Programmentwurf
Einkaufsliste

Name: Hirsch, Jonas
Martrikelnummer: [3448019]

Abgabedatum: [27.05.2022]

Kapitel 1: Einführung

Übersicht über die Applikation

Die Applikation besitzt die Funktion verschiedene Einkaufslisten zu erstellen und abzurufen. Dabei werden aus einer zuvor erstellten Liste, Produkte und Lebensmittel ausgewählt und der Einkaufsliste hinzugefügt. Die Produkte und Lebensmittel werden den Listen einzeln hinzugefügt und in einer Json-Datei gespeichert. Eine weitere Funktion ist das Speichern von Rezepten, wobei man die Zutaten des Rezepts direkt einer zuvor ausgewählten Einkaufsliste hinzugefügt werden. Der Zweck der Applikation ist das einfache Erstellen von Einkaufslisten und der Ausgabe der Einkaufsliste in der Kommandozeile oder die Speicherung als Json-Datei.

Wie startet man die Applikation?

Die Applikation wird über die.Einkaufsliste.exe ausgeführt oder in VisualStudio durch das Starten des Projekts Einkaufsliste. Anschließend wird eine Eingabe geöffnet, in der ein Befehl eingegeben werden kann.

Um ein Produkt oder ein Lebensmittel der Auswahlliste hinzuzufügen können die Befehle „createProduct“ und „createFood“. Nachdem Ausführen der Befehle werden verschiedene Angaben über das Produkt/Lebensmittel ausgegeben, wie der Name und der Preis. Dann wird das Produkt/Lebensmittel in die Auswahlliste hinzugefügt. Weiter Befehle für Produkte und Lebensmittel sind das Ausgeben der Auswahlliste mit dem Befehl „getProductList“/ „getFoodList“. Auch können über die Befehle „deleteProduct“ und „deleteFood“ Produkte und Lebensmittel aus der Auswahlliste entfernt werden.

Das Erstellen einer Einkaufsliste ist etwas komplizierter, wie das Erstellen von Produkten und Lebensmitteln. Zuerst muss über den Befehl „createShoppingList“ eine Einkaufsliste erstellt werden. Der Name für die Einkaufsliste wird über die Eingabe übergeben. Nach der Erstellung der Einkaufsliste können die Produkte und Lebensmittel über die Befehle „addProductToShoppingList“ und „addFoodToShoppingList“ der Einkaufsliste hinzugefügt werden. Falls der Inhalt einer Einkaufsliste angezeigt werden soll, kann der Befehl „getShoppingList“ verwendet werden. Ein weiterer Befehl ist der Befehl für das Löschen einer Einkaufsliste „deleteShoppingList“.

Über den Befehl „createRecipe“ erstellt ein Rezept, wobei zusätzlich neben dem Namen innerhalb der Kommandozeile eine Eingabe der Zutaten verlangt wird. Für die Eingabe der Zutaten wird eine Liste der vorhandenen Lebensmittel ausgegeben und eine Eingabe bereitgestellt, in der die Zutaten einzeln eingegeben werden müssen. Erstellte Rezepte können über den Befehl „getRecipe“ angezeigt werden. Rezepte können ebenfalls über den Befehl „deleteRecipe“ gelöscht werden. Um ein Rezept einer Einkaufsliste hinzuzufügen kann der Befehl „`addToShoppingList`“ verwendete werden. Bei diesem Befehl muss zuerst der Name des Rezepts angegeben werden und anschließend der Name der Einkaufsliste, zu welcher die Lebensmittel hinzugefügt werden sollen.

Wie testet man die Applikation?

Die Applikation kann in Visual Studio (2022) getestet werden. Zum Starten der Tests kann über den Tab „Test“ am oberen Rand von Visual Studio alle Tests ausgeführt werden, oder der Test Explorer aufgerufen werden. Über den Test Explorer können die verschiedenen Tests auch einzeln ausgeführt werden. Ein Shortcut für das Ausführen aller Tests ist „Strg+R, Strg+A“ und ein Shortcut für den Test Explorer ist „Strg+E, Strg+T“.

Kapitel 2: Clean Architecture

Was ist Clean Architecture?

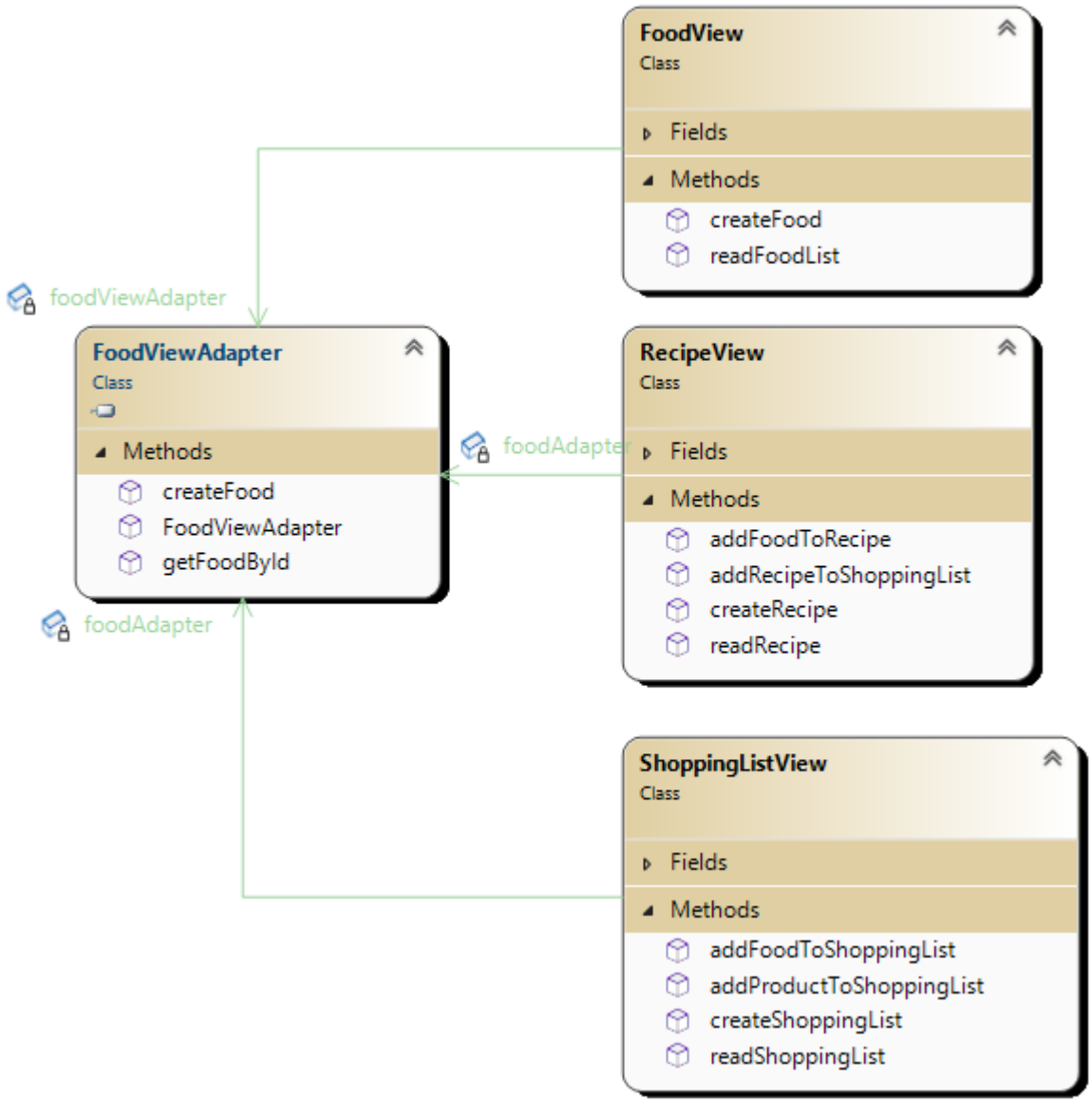
Clean Architecture ist ein Konzept zur Strukturierung von Projekten, wobei ein Projekt in vier Schichten aufgeteilt wird. Die inneren Schichten sollten dabei nicht von den äußeren Schichten in abhängig sein. Die innerste Schicht ist die Domain Schicht, in der der Code enthalten ist, welcher direkt mit der Domäne zu tun hat, weswegen häufig Entities in dieser Schicht enthalten sind. Die zweite Schicht ist die Application Schicht, in der die Use Cases der Applikation implementiert sind. Um Daten und Aufrufe zu vermitteln wird die dritte Schicht verwendet. Dabei wird ein externes Format so umgewandelt, dass es in der Applikation genutzt werden kann. Gleichzeitig werden auch interne Formate umgewandelt, um diese an externe Plugins weitergeben zu können. Die letzte und äußerste Schicht ist die Plugin Schicht. Diese Schicht enthält die verschiedenen Plugins, Frameworks, Datentransportmittel und andere Werkzeuge, welche nicht selbst implementiert werden. In der Schicht sollte nur Code geschrieben werden, welcher die Adapter nutzt, um mit der Applikation zu kommunizieren. Es sollte auf jeden Fall vermieden werden in dieser Schicht Anwendungslogik zu implementieren. Das Ziel der Clean Architetcture ist es das die eigentliche Applikation unabhängig von bestimmten Technologien oder Frameworks ist. Die Unabhängigkeit sorgt dafür, dass Frameworks schnell ausgetauscht werden können und die Anwendung langsamer altert, da diese wenig direkte Abhängigkeiten zu schnell alternden Frameworks besitzt.

Analyse der Dependency Rule

1.Positiv-Beispiel: Dependency Rule

FoodViewAdapter:

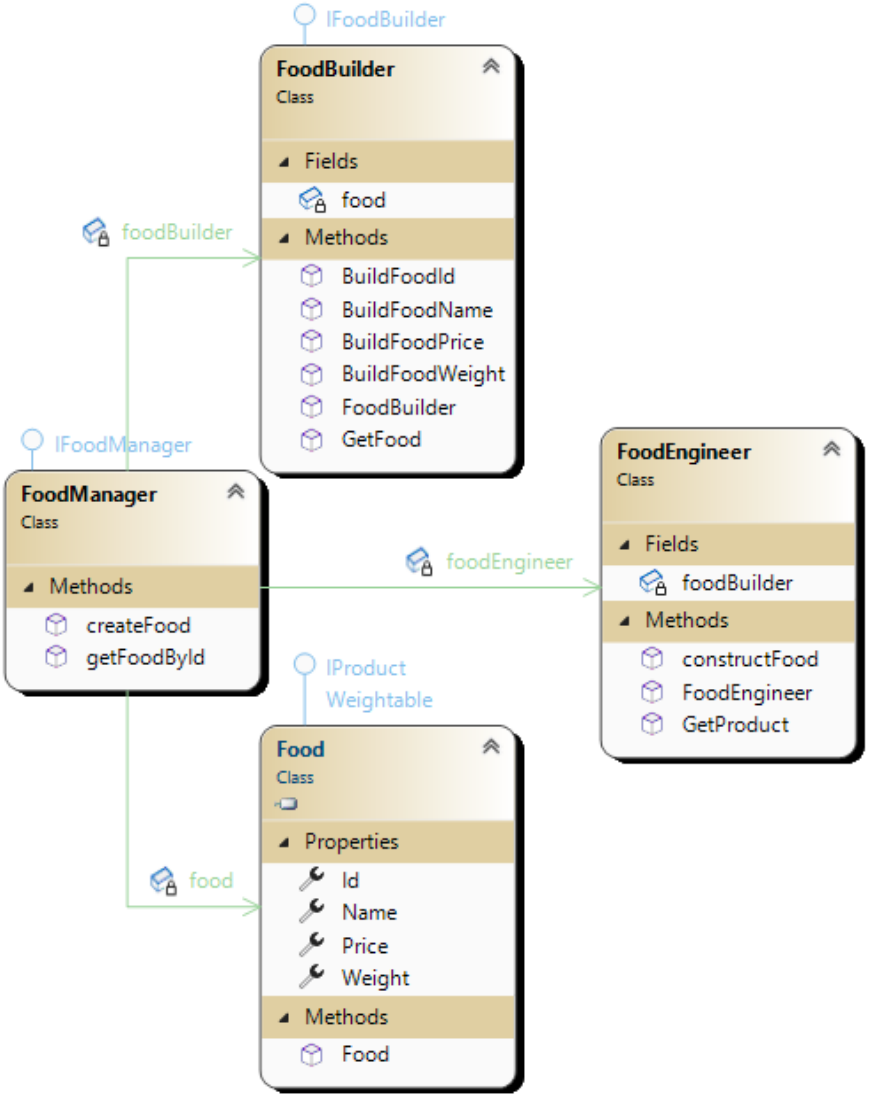
Die Klasse FoodViewAdapter ist Teil der Adapter Schicht und hat die Aufgabe die Aufrufe der Benutzerschnittelle, welche mit Nahrungsmitteln zu tun haben, an die innen liegenden Schichten zu vermitteln. Die Klassen, welche von FoodViewAdapter abhängig sind, sind die Klassen FoodView, ShoppingListView und RecipeView, welcher Teil der Plugin Schicht sind. Die Dependency Rule wird eingehalten,da alle Abhängigkeiten von innen nach außengehen.


FoodView
Class
Fields
Methods
createFood
readFoodList
foodViewAdapter
FoodViewAdapter
Class
Methods
createFood
FoodViewAdapter
getFoodById
foodAdapter
RecipeView
Class
foodAdapter
Fields
Methods
addFoodToRecipe
addRecipeToShoppingList
createRecipe
readRecipe
ShoppingListView
Class
Fields
Methods
addFoodToShoppingList
addProductToShoppingList
createShoppingList
readShoppingList

2.Positiv-Beispiel: Dependency Rule

FoodManager:

Die Klasse FoodManager ist ebenfalls ein positives Beispiel für die Dependency Rule, da die Klasse ausschließlich Abhängigkeiten zu Klassen entweder in der Application Schicht (FoodBuilder und FoodEngineer) oder der Domain Schicht (Food) besitzt. Somit existieren keine Abhängigkeiten zu Klassen äußerer Schichten und die Dependency Rule ist daher eingehalten.



Negativ-Beispiel: Dependency Rule

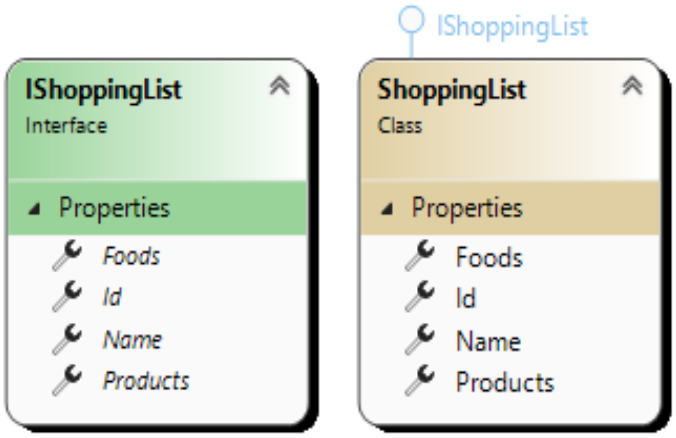
Da die Solution in verschiedene Projekte aufgeteilt ist, welche die verschiedenen Schichten repräsentieren kann, sichergestellt werden, dass die inneren Schichten keine Abhängigkeiten zu den äußeren Schichten besitzen, in dem die Projektreferenzen der einzelnen Projekte betrachtet werden.

Analyse der Schichten

Schicht: [Domaine Code]

ShoppingList:

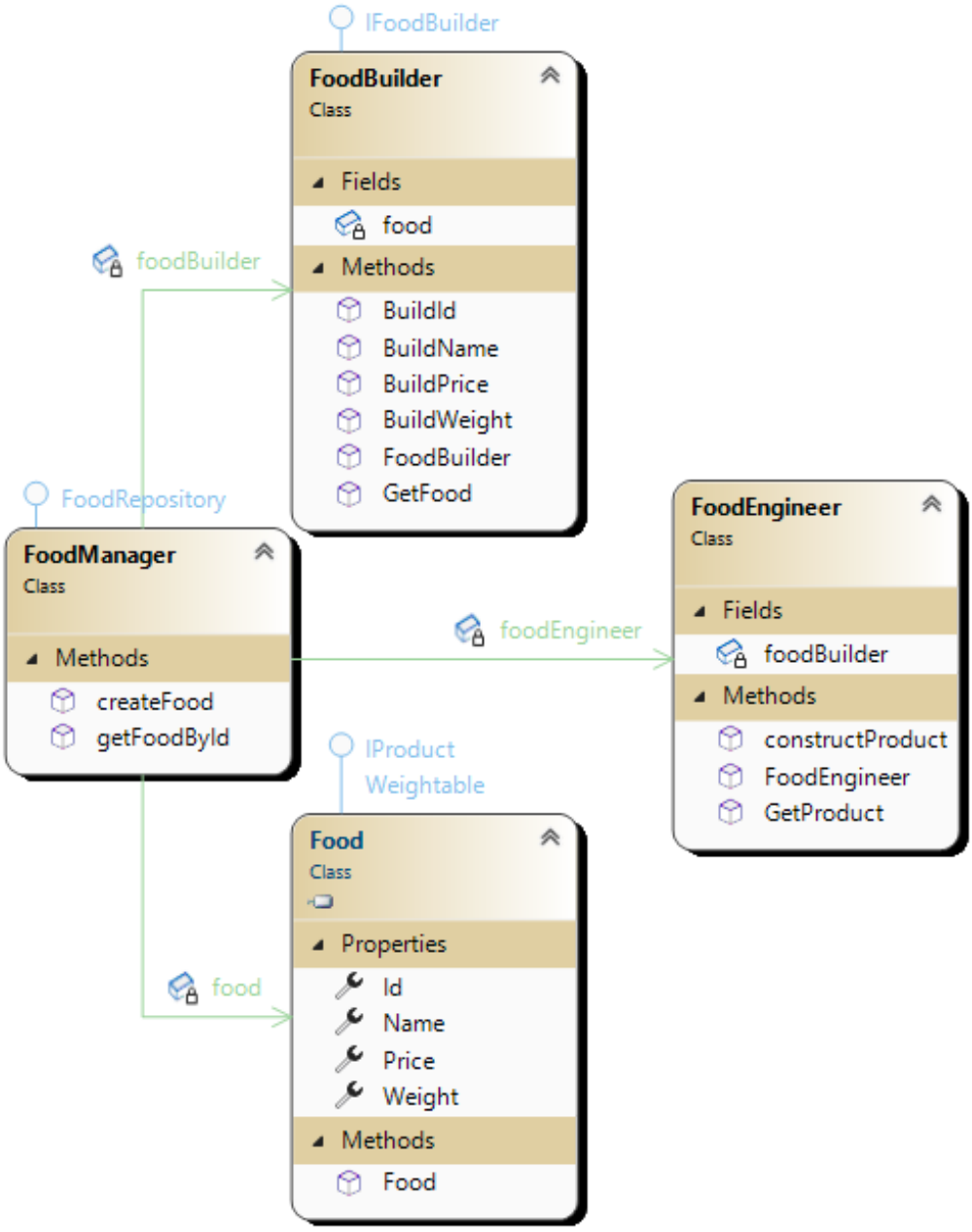
Die ShoppingList.cs Klasse wird genutzt, um das Objekt der Einkaufsliste mit verschiedenen Parametern zu erstellen und somit kann das Objekt in der gesamten Solution genutzt werden. Die ShoppingList.cs Klasse beschreibt dabei ein Objekt, welches in der Domäne vorkommt, weswegen die Klasse Teil des Domaine Code ist.



Schicht: [Application Code]

FoodManager:

In der FoodManager Klasse sind die verschiedenen Use Cases enthalten, welche in Zusammenhang mit Nahrungsmitteln stehen. Die implementierten Funktionen sind das Erstellen von Nahrungsmitteln und das Suchen eines Nahrungsmittels aus einer Liste über die ID. Für das Erstellen eines Nahrungsmittels werden die Klassen FoodBuilder und FoodEngineer verwendet, um ein Nahrungsmittel zu erstellen. Die Beiden Klassen FoodBuilder und FoodEngineer sind Teil des Application Codes, da sie nicht Teil der Fachdomäne sind. Ebenfalls wird die Klasse Food benötigt, welche Teil des Domain Codes ist, da es sich um ein Entity handelt, welches zu der Fachdomäne gehört. Das Entity Food entspricht der Rückgabe der beiden Funktionen in der FoodManager Klasse.


IFoodBuilder
FoodBuilder
Class
Fields
food
Methods
BuildId
BuildName
BuildPrice
BuildWeight
FoodBuilder
GetFood
foodBuilder
FoodRepository
FoodManager
Class
Methods
createFood
getFoodById
foodEngineer
FoodEngineer
Class
Fields
foodBuilder
Methods
constructProduct
FoodEngineer
GetProduct
IProduct
Weightable
Food
Class
Properties
Id
Name
Price
Weight
Methods
Food
food

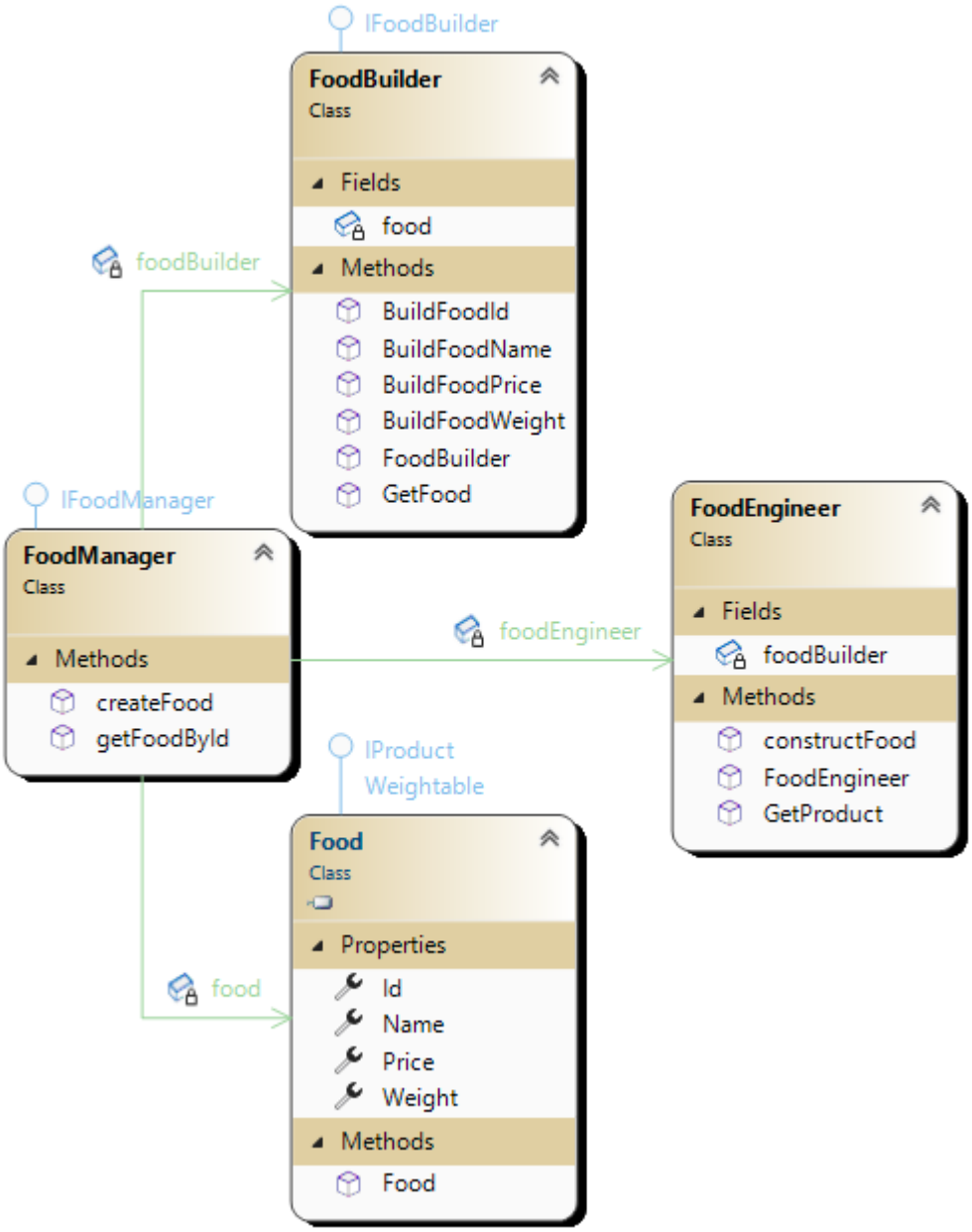
Kapitel 3: SOLID

Analyse Single-Responsibility-Principle (SRP)

Positiv-Beispiel

FoodManager:

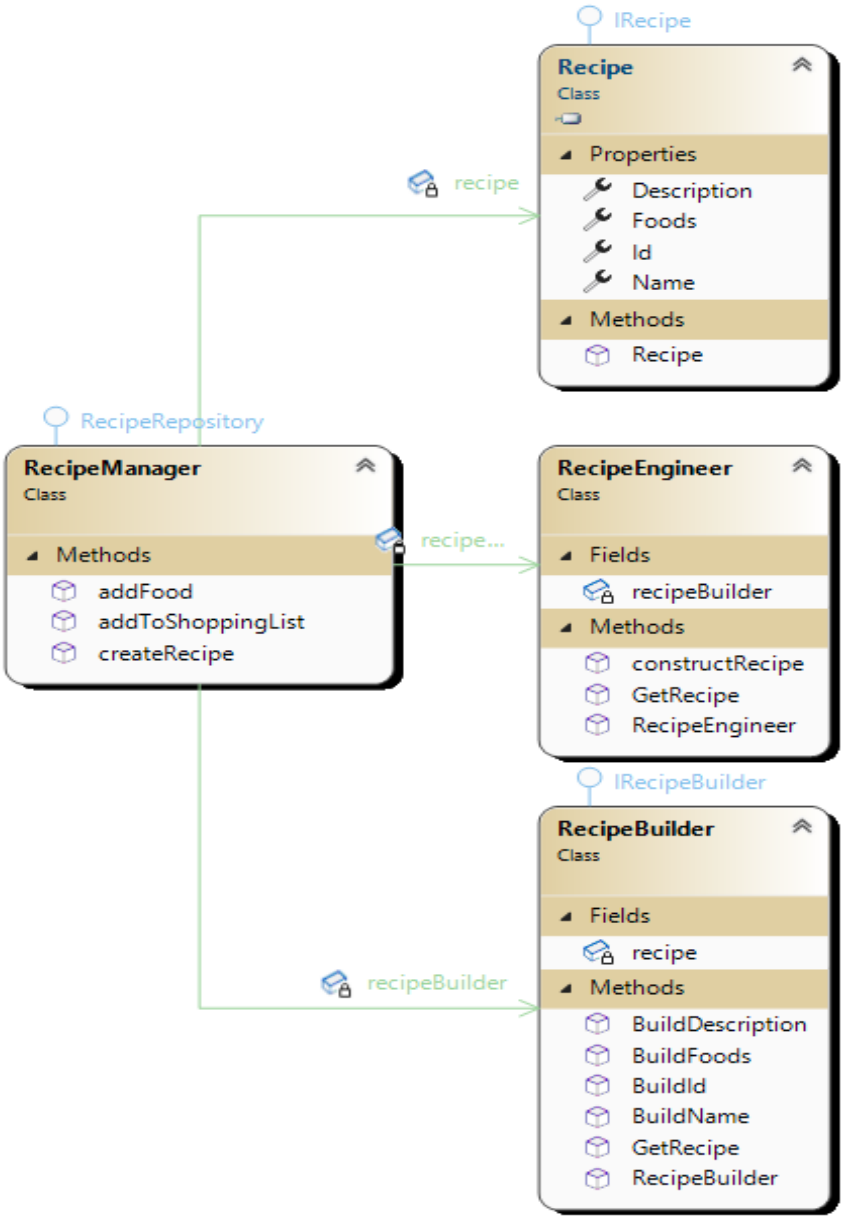
Die Klasse FoodManager beinhaltet die verschiedenen Funktionen, welche in Zusammenhang mit Nahrungsmitteln stehen. Die implementierten Funktionen sind das Erstellen von Nahrungsmitteln und das Suchen eines Nahrungsmittels aus einer Liste über die ID. Das Ergebnis der beiden Funktionen ist das Entity Food, welches ein Nahrungsmittel darstellt.



Negativ-Beispiel

RecipeManager:

In dem RecipeManager sind die verschiedenen Funktionen enthalten, welche mit den Rezepten zu tun haben. Die implementierten Funktionen der Klasse sind das Erstellen von Rezepten (createRecipe), das Hinzufügen von Zutaten zu schon vorhandenen Rezepten (addFood) und das Hinzufügen von Zutaten eines Rezepts zu einer Einkaufsliste (addToShoppingList). Die Klasse verstößt gegen das Single-Responsibility-Principle, da die Funktion addToShoppingList eine Einkaufsliste zurückgibt und somit eher eine Funktion ist, die zu den Funktionen der Einkaufsliste geht. Ein Lösungsweg, um diesen Verstoß zu beheben ist das die Funktion in die Klasse ShoppingListManager verschoben wird.

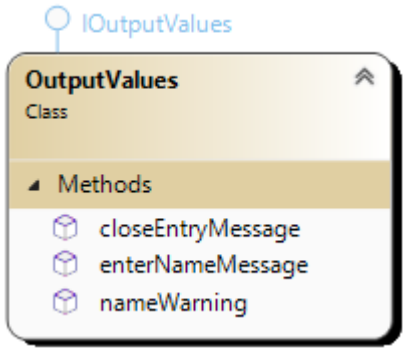


Analyse Open-Closed-Principle (OCP)

Positiv-Beispiel

OutputValues:

Ein positives Beispiel ist die Klasse OutputValues. Diese Klasse ist dafür zuständig Konsolen ausgaben zu erstellen, welche in verschiedenen Klassen des Programms aufgerufen werden müssen. So kann die Klasse für zum Beispiel Ausgaben für die Bitte für das Eingeben eines Namens machen oder auch einen Infotext ausgeben, welcher benötigt wird, um anzugeben, wie man eine Auswahl von Produkten oder Lebensmitteln beendet. Da innerhalb der Klasse keine weiteren Klassen genutzt werden, muss wenn anderer Klassen verändert werden kein Code verändert werden und es können weitere Funktionen eingefügt werden, ohne dass es an anderen Stellen der Klasse Code geändert werden muss.

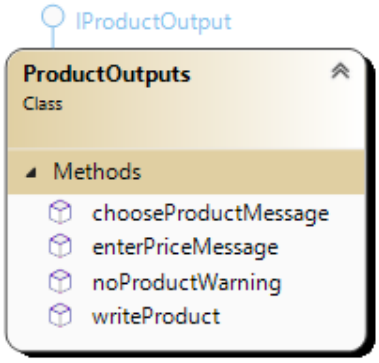


Negativ-Beispiel

ProductOutputs:

Die Klasse ProductOutputs verletzt das Open-Closed-Principle, da wenn das Entity Product, weitere Parameter erhält, innerhalb der Klasse ProductOutputs die Methode writeProduct, welche die Parameter des übergebenen Product ausgibt, angepasst werden muss, so dass dieser Wert ebenfalls in der Konsole ausgegeben wird. Ein Parameter, welcher dem Entity Product hinzugefügt werden könnte, ist zum Beispiel der Hersteller, von dem das Produkt produziert wird.

Eine Möglichkeit die Verletzung zu beheben, ist das die eine Liste der Parameter des Entity Product erstellt wird mit den Funktionen GetType().GetProperties(). Diese Liste kann verwendet werden, um anschließend die einzelnen Parameter auszugeben, indem die Funktionen GetName() und GetValue() verwendet werden, um den Namen und den Wert des Parameters zu erhalten. Ein Problem was hierbei aufkommt ist, dass die Parameter weitere Klassen sein können, wie zum Beispiel der Parameter Price des Entity Product, weswegen für diese Klassen ebenfalls die Parameter einzeln über die gleiche Methode erhalten und ausgegeben werden müssen.



Analyse Liskov-Substitution- (LSP), Interface-Segreggation- (ISP), Dependency-Inversion-Principle (DIP)

Positiv-Beispiel

Interface Segregation Principle:

Food:

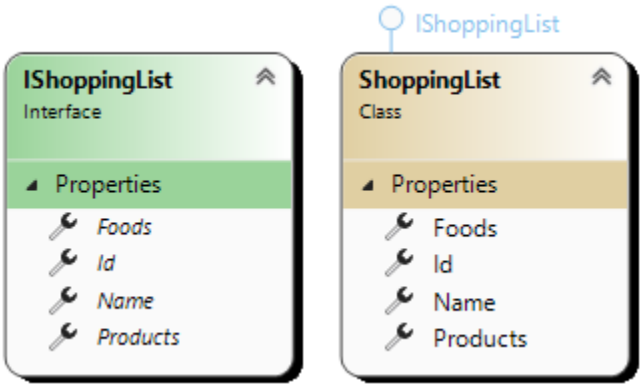
Das Entity Food besitzt die beiden Interfaces IEntity, IProduct und Weightable. Da ein Lebensmittel ein Entity ist wird das Interface IEntity genutzt genutzt. Ebenfalls ist ein Lebensmittel ein Produkt, weshalb das Interface IProduct für das Entity verwendet wird. Zusätzlich ist für ein Lebensmittel das Gewicht wichtig, weswegen zusätzlich das Interface Weightable verwendet wird. Die Klasse erfüllt das Interface Segregation Principle, da es zwei modulare Interfaces implementiert.

Negativ-Beispiel

Interface Segregation Principle:

ShoppingList:

Die Klasse ShoppingList erfüllt das Interface Segregation Principle nicht, da das Interface IShoppingList in mehrere modularere Interfaces unterteilt werden kann. Das Interface IShoppingList besitzt die Parameter Id, Name, Foods und Products. Der Parameter Foods ist eine Liste von Guid's von Lebensmitteln und Products ist eine Liste von Guid's von Produkten. Für diese Parameter kann es sinnvoll sein, eigene Interfaces zu erstellen, da der Parameter Foods zum Beispiel ebenfalls in dem Interface IRecipe verwendet wird.



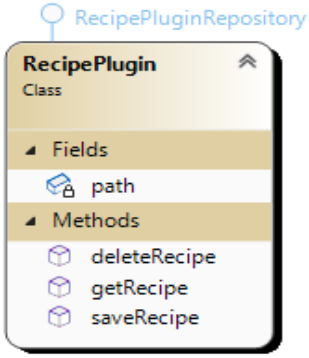
Kapitel 4: Weitere Prinzipien

Analyse GRASP: Geringe Kopplung

Positiv-Beispiel

RecipePlugin:

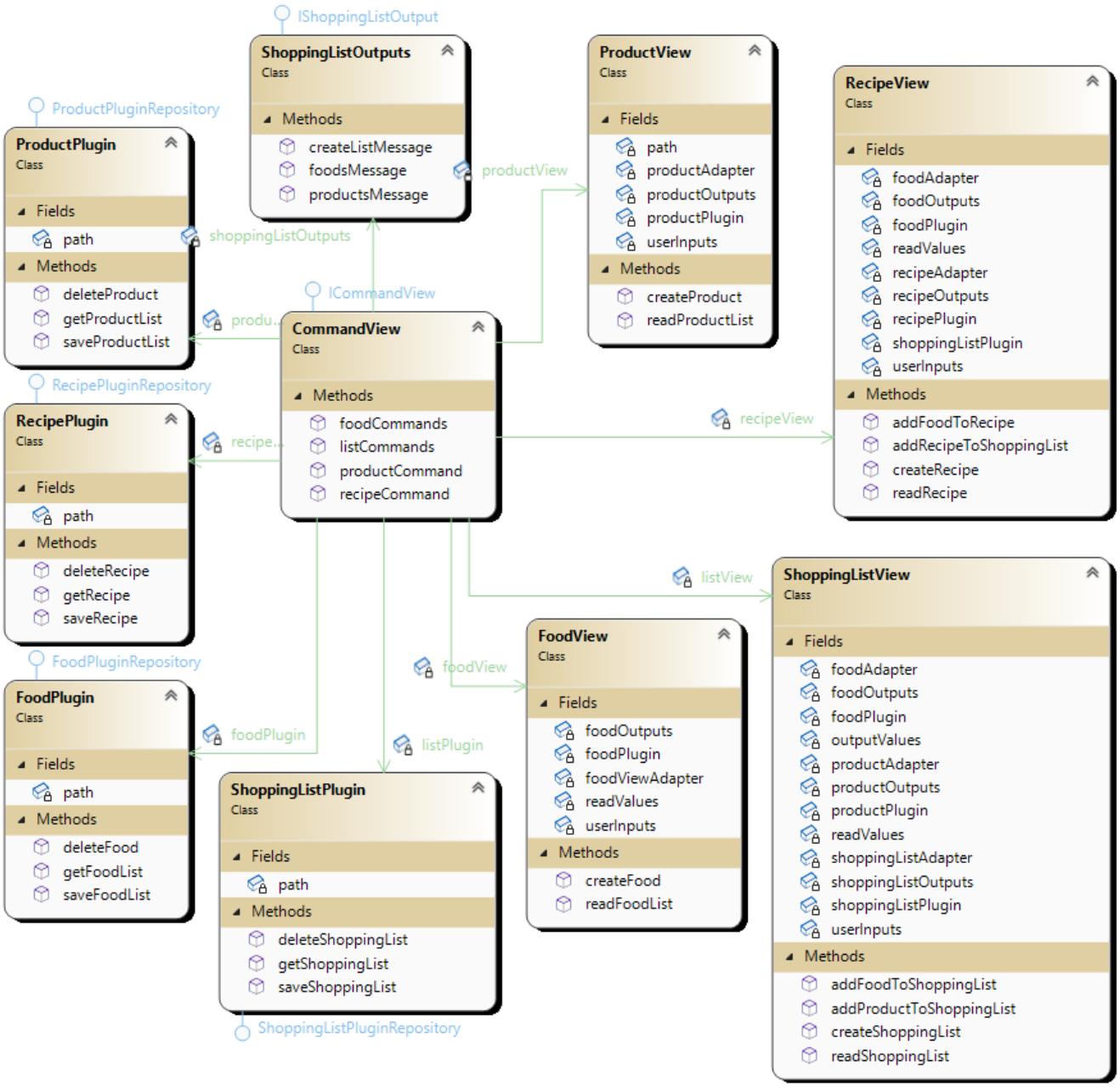
Die Klasse RecipePlugin hat die Aufgabe die erstellten Rezepte in eine JSON-Datei zu speichern, erstellte Rezepte zu löschen, oder zu laden. Die Kopplung innerhalb der Klasse ist gering, da die Klasse von keiner anderen Klasse abhängig ist und nur das Interface RecipePluginService implementiert. Die Methoden innerhalb der Klasse sind ebenfalls voneinander unabhängig und können somit beliebig erweitert oder verändert werden. Ebenfalls kann die Klasse um weitere Funktionen erweitert werden, ohne dass die Applikation dadurch nicht mehr funktionieren würde.



Negativ-Beispiel

CommandView:

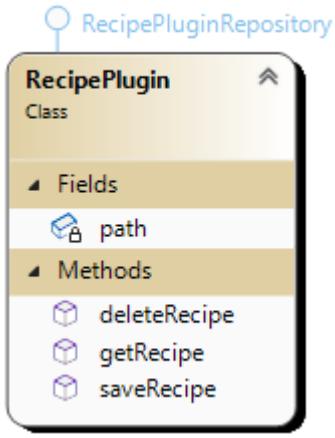
Die Klasse CommandView ist von vielen verschiedenen Klassen abhängig. Die Aufgabe der Klasse ist es, die Benutzereingaben, welche an die Methoden der Klasse übergeben werden, den Mehtoden der anderen Klassen zuzuordnen und diese Methoden aufzurufen. Einige Klassen wie zum Beispiel RecipePlugin und RecipeView oder FoodPlugin und FoodView zu gemeinsam in einer Methode genutzt werden. Da die Klasse CommandView stark abhängig von den verschiedenen Klassen und deren Funktionen ist kommt es in der Klasse zu einer hohen Kopplung, da wenn Änderungen an den Methoden der Klassen, von denen die CommandView Klasse abhängt, gemacht werden es zu Problem in der CommandView Klasse kommen kann, da bestimme Funktionen nicht mehr richtig funktionieren. Eine Lösung wäre es weitere Klassen zu erstellen, welche die Klassen, von denen die CommandView Klasse abhängt, zusammenfassen und somit die Änderungen nur in diese Klassen bearbeitet werden müssen. Die Klassen, von denen die CommandView Klasse abhängt, von denen die CommandView Klasse abhängt, welche zusammengefasst werden sollten, sind die Klassen, welche in derselben Methode der CommandView Klasse aufgerufen werden wie zum Beispiel die RecipePlugin Klasse und die RecipeView Klasse oder die FoodPlugin Klasse und die FoodView Klasse.


IShoppingListOutput
ShoppingListOutputs
Class
Methods
createListMessage
foodsMessage
productsMessage
productView
ProductView
Class
Fields
path
productAdapter
productOutputs
productPlugin
userInputs
Methods
createProduct
readProductList
RecipeView
Class
Fields
foodAdapter
foodOutputs
foodPlugin
readValues
recipeAdapter
recipeOutputs
recipePlugin
shoppingListPlugin
userInputs
Methods
addFoodToRecipe
addRecipeToShoppingList
createRecipe
readRecipe
ProductPluginRepository
ProductPlugin
Class
Fields
path
Methods
deleteProduct
getProductList
saveProductList
shoppingListOutputs
ICommandView
produ...
CommandView
Class
Methods
foodCommands
listCommands
productCommand
recipeCommand
recipeView
RecipePluginRepository
RecipePlugin
Class
Fields
path
Methods
deleteRecipe
getRecipe
saveRecipe
recipe...
listView
ShoppingListView
Class
Fields
foodAdapter
foodOutputs
foodPlugin
outputValues
productAdapter
productOutputs
productPlugin
readValues
shoppingListAdapter
shoppingListOutputs
shoppingListPlugin
userInputs
Methods
addFoodToShoppingList
addProductToShoppingList
createShoppingList
readShoppingList
FoodView
Class
Fields
foodOutputs
foodPlugin
foodViewAdapter
readValues
userInputs
Methods
createFood
readFoodList
foodView
FoodPluginRepository
FoodPlugin
Class
Fields
path
Methods
deleteFood
getFoodList
saveFoodList
foodPlugin
listPlugin
ShoppingListPlugin
Class
Fields
path
Methods
deleteShoppingList
getShoppingList
saveShoppingList
ShoppingListPluginRepository

Analyse GRASP: Hohe Kohäsion

RecipePlugin:

Hohe Kohäsion bedeutet, dass die verschiedenen Methoden und Parameter innerhalb einer Klasse logisch und semantisch zusammenhängen. Ein Positives Beispiel einer Klasse mit hoher Kohäsion ist die Klasse Recipe Plugin. Die Methode der Klasse sind logisch und semantische zusammenhängend, da alle Funktionen mit den JSON-Dateien des Objekts Recipe zusammenhängen, um diese Dateien zu speichern, zu löschen oder zu laden. Auch der Parameter path ist logisch im Zusammenhang der Methoden, da dieser den Speicherort der Dateien angibt, wo diese gespeichert, gelöscht oder geholt werden.



Don’t Repeat Yourself (DRY)

Commit: Entfernen von dupliziertem Code (25.05.2022) (`e6235266539881aa6554c7025762ccde446b6f12`)

In der Klasse RecipeView ist Code implementiert, welcher für die Auswahl von mehreren Lebensmitteln aus einer Liste genutzt wird. Dieser Code wird in zwei der Methoden genutzt, wobei der Code somit doppelt in der Klasse verwendet wird. Aus diesem Grund wird der Code in eine eigene Methode ausgelagert, welche in einer anderen Klasse liegt. Ein weiteres Stück Code, welches dupliziert in der Klasse vorkommt ist das Abfragen eines Namens, weswegen dieser Code ebenfalls in eine eigene Methode in der gleichen Klasse ausgelagert wird, wie der Code für die Auswahl der Lebensmittel. Dies bringt den Vorteil, dass der Code, falls es zu Änderungen kommt, nur an einer Stelle geändert werden muss. Ebenfalls ist der Code leichter zu lesen, da die Klasse kürzer ist und die Methoden besser benannt werden können.

Vorher:

```
public class RecipeView
{
    ReadValues readValues = new ReadValues();
    RecipePlugin recipePlugin = new RecipePlugin();
    ShoppingListPlugin shoppingListPlugin = new ShoppingListPlugin();
    FoodPlugin foodPlugin = new FoodPlugin();
    OutputValues outputValues = new OutputValues();
    RecipeOutputs recipeOutputs = new RecipeOutputs();
    FoodOutputs foodOutputs = new FoodOutputs();
    RecipeManager recipeManager = new RecipeManager();
    UserInputs userInputs = new UserInputs();
    FoodManager foodManager = new FoodManager();
    public void createRecipe()
    {
        List<Food> foods = new List<Food>();
        List<Guid> foodIds = new List<Guid>();
        List<Food> foodList = foodPlugin.getFoodList();
        string food = "";
        string name = "";

        outputValues.enterNameMessage();
        name = readValues.ReadString();

        foodOutputs.chooseFoodMessage();

        foreach (var fd in foodList)
        {
            Console.WriteLine(fd.Name);
        }
        while (food != null && food != "q")
        {
            food = readValues.ReadString();
            if (food != "q" && food != null)
            {
                var addedFood = foodList.FirstOrDefault(f => f.Name == food);
                foods.Add(addedFood);
                outputValues.closeEntryMessage();
            }
        }

        recipeOutputs.enterDescriptionMessage();
        string desc = readValues.ReadString();

        if (name != null && name != "")
        {
            Recipe recipe = recipeManager.createRecipe(name, foodIds, desc);
            recipePlugin.saveRecipe(recipe);
        }
    }

    public void readRecipe(string name = null)
    {
        if (name == null)
        {
            ReadValues readValues = new ReadValues();
            outputValues.enterNameMessage();
            name = readValues.ReadString();
        }
        Recipe recipe = recipePlugin.getRecipe(name);
```

```
        List<Food> foods = foodPlugin.getFoodList();
        recipeOutputs.foodsMessage();
        foreach (Guid foodId in recipe.Foods)
        {
            Food food = foodManager.getFoodById(foodId, foods);
            foodOutputs.writeFood(food);
        }
    }

    public void addFood(string recipeName)
    {
        int count = 0;
        string food = "";
        Food addedFood = new Food();
        Recipe recipe = recipePlugin.getRecipe(recipeName);
        List<Food> foods = new List<Food>();
        List<Food> foodList = foodPlugin.getFoodList();

        foreach (var fd in foodList)
        {
            Console.WriteLine(fd.Name);
        }
        while (food != null && food != "q")
        {
            food = readValues.ReadString();
            if (food != null && food != "q")
            {
                addedFood = foodList.FirstOrDefault(f => f.Name == food);
                if (addedFood != null)
                {
                    recipeManager.addFood(recipe, addedFood);
                }
                outputValues.closeEntryMessage();
            }

            recipePlugin.saveRecipe(recipe);
        }

        public void addToShoppingList(string recipeName = null, string listName =
null)
        {
            if (recipeName == null)
            {
                outputValues.enterNameMessage();
                recipeName = readValues.ReadString();
            }
            Recipe recipe = recipePlugin.getRecipe(recipeName);

            if (listName == null)
            {
                outputValues.enterNameMessage();
                listName = readValues.ReadString();
            }
            ShoppingList shoppingList = shoppingListPlugin.getShoppingList(listName);

            if (shoppingList != null)
            {
                shoppingList = recipeManager.addToShoppingList(recipe, shoppingList);
                shoppingListPlugin.saveShoppingList(shoppingList);
            }
```

```
            }
        }
    }
```

Nachher:

```
public class RecipeView
    {
        ReadValues readValues = new ReadValues();
        RecipePlugin recipePlugin = new RecipePlugin();
        ShoppingListPlugin shoppingListPlugin = new ShoppingListPlugin();
        FoodPlugin foodPlugin = new FoodPlugin();
        RecipeOutputs recipeOutputs = new RecipeOutputs();
        FoodOutputs foodOutputs = new FoodOutputs();
        RecipeManager recipeManager = new RecipeManager();
        UserInputs userInputs = new UserInputs();
        FoodManager foodManager = new FoodManager();
        public void createRecipe()
        {
            List<Food> foods = new List<Food>();
            List<Guid> foodIds = new List<Guid>();
            List<Food> foodList = foodPlugin.getFoodList();
            string name = "";

            name = userInputs.getName();

            foodOutputs.chooseFoodMessage();

            foods = userInputs.chooseFoods(foodList);
            foreach(Food food in foods)
            {
                foodIds.Add(food.Id);
            }

            recipeOutputs.enterDescriptionMessage();
            string desc = readValues.ReadString();

            if(name != null && name != "")
            {
                Recipe recipe = recipeManager.createRecipe(name, foodIds, desc);
                recipePlugin.saveRecipe(recipe);
            }
        }

        public void readRecipe(string name = null)
        {
            if (name == null)
            {
                ReadValues readValues = new ReadValues();
                name = userInputs.getName();
            }
            Recipe recipe = recipePlugin.getRecipe(name);
            List<Food> foods = foodPlugin.getFoodList();
            recipeOutputs.foodsMessage();
            foreach (Guid foodId in recipe.Foods)
            {
                Food food = foodManager.getFoodById(foodId, foods);
                foodOutputs.writeFood(food);
            }
        }

        public void addFood(string recipeName)
        {
            Recipe recipe = recipePlugin.getRecipe(recipeName);
            List<Food> foods = new List<Food>();
```

```
            List<Food> foodList = foodPlugin.getFoodList();

            foods = userInputs.chooseFoods(foodList);

            foreach(Food fd in foods)
            {
                recipeManager.addFood(recipe, fd.Id);
            }
            recipePlugin.saveRecipe(recipe);
        }

        public void addToShoppingList(string recipeName = null, string listName = null)
        {
            if (recipeName == null)
            {
                recipeName = userInputs.getName();
            }
            Recipe recipe = recipePlugin.getRecipe(recipeName);

            if (listName == null)
            {
                listName = userInputs.getName();
            }
            ShoppingList shoppingList = shoppingListPlugin.getShoppingList(listName);

            if(shoppingList != null)
            {
                shoppingList = recipeManager.addToShoppingList(recipe, shoppingList);
                shoppingListPlugin.saveShoppingList(shoppingList);
            }

        }
    }
```

Kapitel 5: Unit Tests

10 Unit Tests

Unit Test	Beschreibung
FoodTest#CreateFood	Testet, ob ein Lebensmittel korrekt erstellt wird.
JsonFoodTest#DeleteFood	Tetstet, ob ein vorhandener Eintrag in der Json Datei Food.json korrekt gelöscht wird.
JsonFoodTest#GetFood	Testet, ob die Liste von Lebensmittel richtig geladen wird.
ShoppingListTest#CreateList	Testet, ob der eine neue Einkaufliste mit den gegebenen Werten erstellt wird.
FoodCommandTest#WrongArgument	Testet, ob ein falscher Befehl den erwarteten Wert zurückgibt.
RecipeTest#AddFood	Testet, ob eine Zutat der Zutatenliste des Rezepts hinzugefügt wird.
RecipeTest#AddToShoppingList	Testet, ob die Zutaten des Rezepts einer ausgewählten Einkaufsliste hinzugefügt werden können.
ShoppingListTest#AddProduct	Testet, ob der Einkaufsliste ein Produkt hinzugefügt werden kann.
JsonShoppingListTest#DeleteShoppingList	Testet, ob eine Einkaufsliste gelöscht werden kann.
JsonShoppingListTest#GetShoppingList	Testet, ob eine ausgewählte Einkaufsliste ausgegeben wird.

ATRIP: Automatic

Die Tests können schnell mit einem Knopfdruck alle im Test-Explorer auf einmal ausgeführt werden. Während der Ausführung der Tests müssen keine Eingaben getätigt werden und die Tests sind alle erfolgreich.

ATRIP: Thorough

Ein positives Beispiel sind die Tests zum Erstellen einer Einkaufsliste. Bei den Tests wird sowohl überprüft, ob eine Einkaufsliste erfolgreich bei richtigen Eingaben erstellt wird (ShoppingListTest#CreateShoppingList), als auch dass der Erstellvorgang abgebrochen wird, wenn kein Name eingegeben wird (ShoppingListTest#CreateShoppingListNoName).

```
[TestMethod]
  public void CreateList()
  {
      //arrange
      string name = "TestListe";

      //act
      ShoppingList shoppingList = listManager.createShoppingList(name);

      //assert
      Assert.IsNotNull(shoppingList);
  }

  [TestMethod]
  public void CreateListNoName()
  {
      //arrange
      string name = "";

      //act
      ShoppingList shoppingList = listManager.createShoppingList(name);


      //assert
      Assert.IsNull(shoppingList);
  }
```

Ein negatives Beispiel sind die Tests zum Erstellen von Lebensmitteln. Bei diesen Tests wird neben dem erfolgreichen Erstellen von einem Lebensmittel (FoodTest#CreateFood) nur getestet, wie sich das Programm verhält, wenn kein Name eingegeben wird (FoodTest#CreateFoodNoName). Andere Eingaben, wie das Gewicht oder der Preis werden vernachlässigt.

```
  [TestMethod]
  public void CreateFood()
  {
      //arrange

      //act
      Food food = foodManager.createFood(apple.Name, apple.Weight, apple.Price);

      //assert
      Assert.AreEqual(apple.ToString(), food.ToString());
  }
  [TestMethod]
  public void CreateFoodNoName()
  {
      //arrange
      expectedFoods.Add(apple);
      StringReader priceReader = new StringReader("");
      Console.SetIn(priceReader);
      //act
      Food food = foodManager.createFood("", apple.Weight, apple.Price);

      //assert
      Assert.IsNull(food);

  }
```

ATRIP: Professional

FoodCommandTest:

Ein Positives Beispiel ist die Testklasse FoodCommandTest. In dieser Testklasse wird getestet, ob die Methode foodCommands der Klasse CommandView funktionieren. Ein Grund, warum die Testklasse professional ist, ist, dass ein Mock der Klasse CommandView verwendet, damit der die Klasse getrennt von den restlichen Klassen getestet werden kann. Ebenfalls wird darauf geachtet, dass nur unbedingt notwendiger Code doppelt in den Testmethoden vorkommt, wie zum Beispiel der StringWriter, welcher einzeln aufgerufen werden muss. Anderer Code wie der CommandViewMock und der der String für das erwartete Ergebnis (expected) werden in der Startup Methode, welche mit [TestInitialize] initialisiert wird, initialisiert und der Kompletten Klasse zur Verfügung gestellt.

```
public class FoodCommandTest
    {
        CommandViewMock commandViewMock;
        string output;
        string expected;
        [TestInitialize]
        public void Startup()
        {
            output = "";
            expected = "true\r\n";
            commandViewMock = new CommandViewMock();
        }
        [TestMethod]
        public void GetFoodCommand()
        {
            //arrange
            string command = "getFoodList";
            //act
            using (StringWriter sw = new StringWriter())
            {
                Console.SetOut(sw);
                commandViewMock.foodCommands(command);
                output = sw.ToString();
            }

            //assert
            Assert.AreEqual(expected, output);
        }
        [TestMethod]
        public void DeleteFoodCommand()
        {
            //arrange
            string command = "deleteFood";
            //act
            using (StringWriter sw = new StringWriter())
            {
                Console.SetOut(sw);
                commandViewMock.foodCommands(command);
                output = sw.ToString();
            }

            //assert
            Assert.AreEqual(expected, output);
        }
```

```
    [TestMethod]
    public void CreateFoodCommand()
    {
        //arrange
        string command = "createFood";
        //act
        using (StringWriter sw = new StringWriter())
        {
            Console.SetOut(sw);
            commandViewMock.foodCommands(command);
            output = sw.ToString();
        }

        //assert
        Assert.AreEqual(expected, output);
    }
    [TestMethod]
    public void WrongArgument()
    {
        //arrange
        string command = "Food";
        string expected = "Kein echter Befehl\r\n";
        //act
        using (StringWriter sw = new StringWriter())
        {
            Console.SetOut(sw);
            commandViewMock.foodCommands(command);
            output = sw.ToString();
        }

        //assert
        Assert.AreEqual(expected, output);
    }
}
```

ReadValueTest:

Ein negatives Beispiel ist die Testklasse ReadValueTest. In dieser Klasse wird nicht darauf geachtet, ob Code in den Testmethoden doppelt vorkommt. So ist zu sehen, dass in jeder Methode einzeln die Klasse ReadValues initialisiert wird und auch in jeder Klasse ein eigener StringReader verwendet wird, obwohl man diese Klassen nur einmal in einer Statup-Methode, welche mit [TestInitialize] initialisiert wird, initialisieren müsste und dann in jeder Klasse nutzen könnte.

```
[TestClass]
    public class ReadValuesTest
    {

        [TestMethod]
        public void ReadStringTest()
        {
            //arrange
            ReadValues readValues = new ReadValues();
            string testString = "testString";


            //act
            StringReader stringReader = new StringReader(testString);
            Console.SetIn(stringReader);
            string output =  readValues.ReadString();

            //assert
            Assert.AreEqual(testString, output);
        }
        [TestMethod]
        public void ReadDoubleTest()
        {
            //arrange
            ReadValues readValues = new ReadValues();
            double testDouble = 10.00;
            StringReader doubleReader = new StringReader(testDouble.ToString());
            Console.SetIn(doubleReader);

            //act
            double output = readValues.ReadDouble();
            //assert
            Assert.AreEqual(testDouble, output);
        }
        [TestMethod]
        public void ReadIntTest()
        {
            //arrange
            ReadValues readValues = new ReadValues();
            int testInt = 10;
            StringReader intReader = new StringReader(testInt.ToString());
            Console.SetIn(intReader);

            //act
            int output = readValues.ReadInt();

            //assert
            Assert.AreEqual(testInt, output);
        }
    }
```

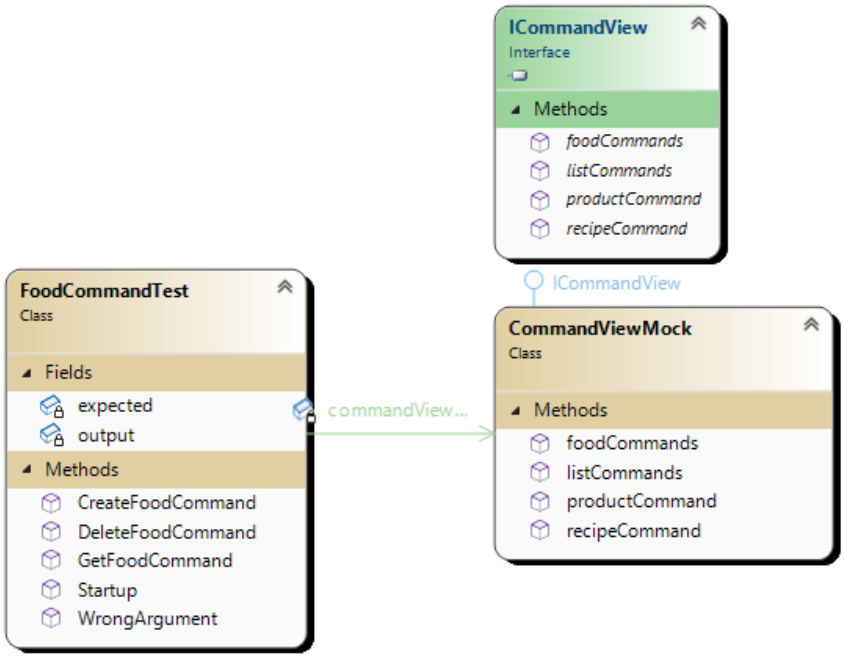
Code Coverage

Die Code Coverage in dem gesamten Projekt beträgt 91,78% und wurde mit dem Tool Clean Code Coverage gemessen. Um auf die 91,78% zu errechnen, wurden die Code Coverages der einzelnen Projekte addiert und durch die Anzahl der Projekte geteilt. Es kann keine 100-prozentige Code Coverage in den Projekten erreicht werden, da es in C# nur möglich ist eine Eingabe für das Testen einer Konsolenapplikation zu übergeben. Somit können nicht alle Pfade des Codes getestet werden.

Fakes und Mocks

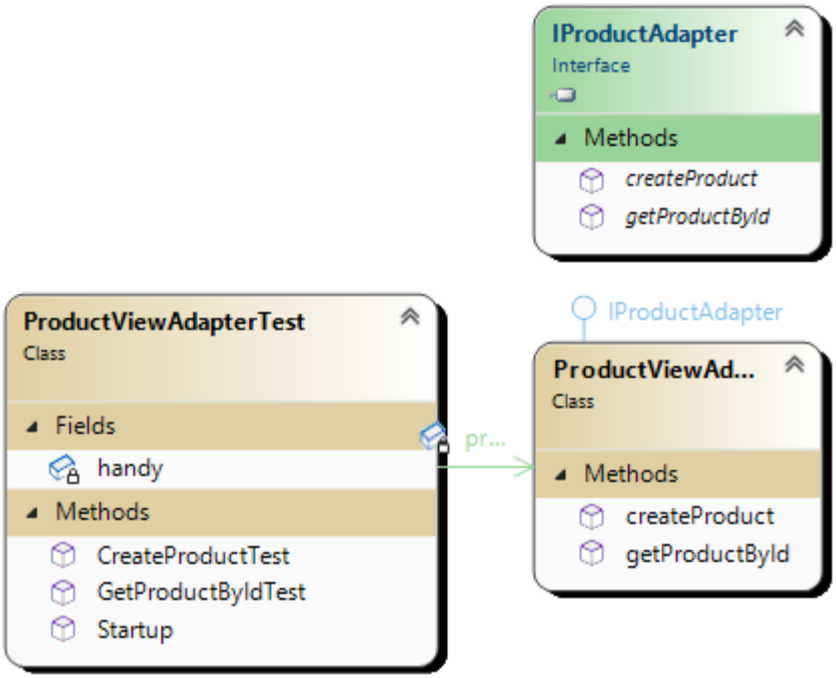
1. CommandViewMock

Damit die Klasse CommandView getestet werden kann wird ein Mock-Objekt des Interfaces ICommandView, welches die Klasse CommandView verwendet. Innerhalb der Test Klassen FoodCommandTest, ProductCommandTest, ShoppingListCommandTest und RecipeCommandTest wird das Mock-Objekt genutzt, um zu prüfen, dass die einzelnen Kommandos, welche eingegeben werden können zu den richtigen Funktionen führt, ohne dass diese Funktionen aufgerufen werden. Aus diesem Grund sind die Funktionen durch Konsolenausgaben ersetzt worden, welche in den Tests gelesen werden und mit einem erwarteten Wert verglichen werden. Als Beispiel wird das Class Diagram der Test Klasse FoodCommandTest genutzt.



2. ProductViewAdapterMock

Um die Klasse ProductViewAdapter zu testen wird ein Mock-Objekt des Interfaces IProductAdapter, welches die Klasse ProductViewAdapter nutzt.



Kapitel 6: Domain Driven Design

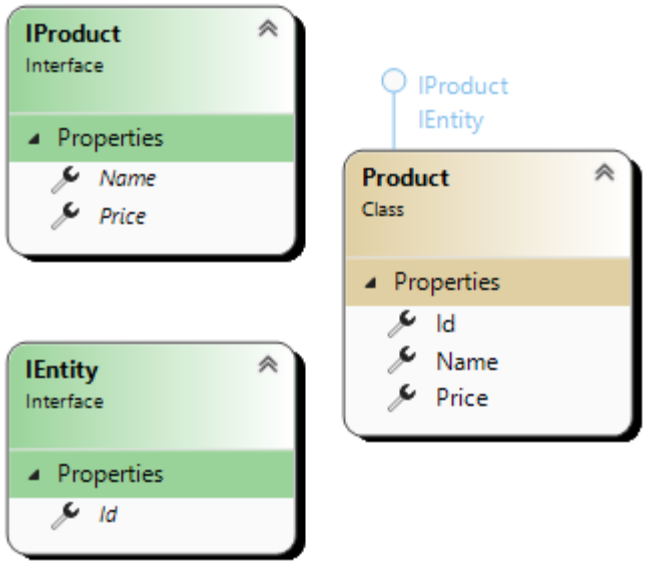
Ubiquitous Language

Bezeichung	Bedeutung	Begründung
ShoppingList	Einkaufsliste	ShoppingList ist der englische Begriff für Einkaufsliste und kann somit von Domänenexperten und Entwicklern verstanden werden.
Recipe	Rezept	Recipe ist der englische Begriff für Rezept und kann somit von Domänenexperten und Entwicklern verstanden werden.
Food	Lebensmittel/Zutat	Food ist der englische Begriff für Lebensmittel oder Zutat und kann somit von Domänenexperten und Entwicklern verstanden werden.
Product	Produkt	Product ist der englische Begriff für Produkt und kann somit von Domänenexperten und Entwicklern verstanden werden.

Entities

Product:

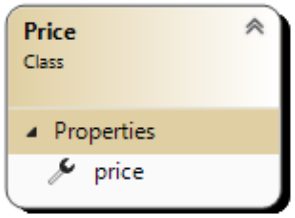
Das Objekt Product ist ein Entity, da es Parameter besitzt, welche Potenziell geändert werden können. So kann zum Beispiel der Preis wegen eines Sonderangebotes verändert werden.



Value Objects

Price:

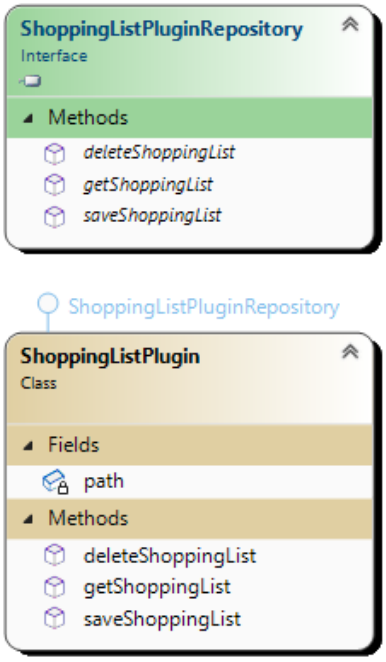
Der Price ist ein Value Object, welches aus einem Double besteht. Dieses Value Object wird eingesetzt, um bei den Objekten Food und Product den Preis anzugeben. Der Einsatz ist sinnvoll, da in zukünftigen Implementierungen der Preis zwischen den Objekten Product oder Food verglichen werden sollen, wobei durch die Verwendung von Price schnell festgestellt werden kann, ob es sich um denselben Preis handelt.



Repositories

ShoppingListPluginRepository:

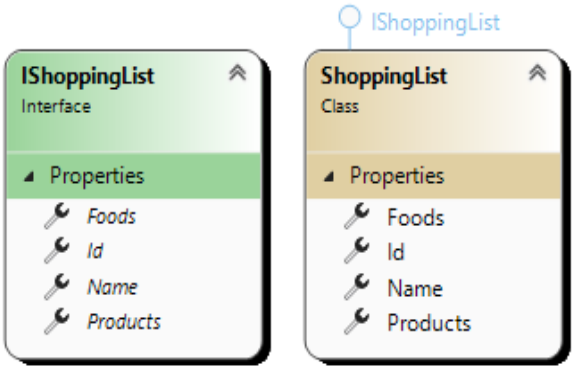
Das Interface ShoppingListPluginRepository wird in dem Domain Code implementiert und steht in engem Zusammenhang mit dem Aggregate ShoppingList. ShoppingListPluginRepository wird der Klasse ShoppingListPlugin genutzt, in welcher die einzelnen Methoden implementiert werden. Durch die Klasse ShoppingListPlugin bekommt der Domain Code Zugriff auf die persistente Speicherung. Das Repository wird eingesetzt, um das Aggregate ShoppingList persistent zu speichern. Ebenfalls kann das Aggregate durch das Repository gelöscht, ausgelesen oder verändert werden. Dadurch, dass die Klasse ShoppingListPlugin in der Plugin Schicht implementiert wird, bleibt der konkrete technische Zugriff auf den Speicher durch das Repository verborgen und die Domäne bleibt von technischen Details unbeeinflusst.



Aggregates

ShoppingList:

Die Einkaufliste besteht aus einer ID, einem Namen, einer Liste an Produkt Id‘s und einer Liste an Lebensmittel Id‘s. Das Aggregat ShoppingList wird eingesetzt, da eine Einkaufsliste aus verschiedenen Produkten und Lebensmitteln besteht. Somit kann das Entity Product und das Value Object Food miteinander verbunden und verwaltet werden, indem die Id’s der Produkte und Lebensmittel in Listen gespeichert werden. Die einzelnen Lebensmittel und Produkte können über die Id’s der Einkaufsliste zugewiesen werden, falls diese aufgerufen werden.



Kapitel 7: Refactoring

Code Smells

Duplicated Code:

Commit: Entfernen von dupliziertem Code (25.05.2022) (`e6235266539881aa6554c7025762ccde446b6f12`)

Die Klassen ProductView, FoodView, RecipeView und ShoppingListView nutzen alle den gleichen Code für das Abfragen eines Namens. Aus diesem Grund wird die Abfrage eines Namens in eine eigene Methode in der Klasse UserInputs ausgelagert und jede Klasse ruft den neuen Code auf, welcher anstatt zweit Zeilen auf eine Zeile reduziert wurde. Ebenfalls wird in den Klassen FoodView und ProductView der gleiche Code für das Abfragen eines Preises verwendet, weshalb dieser Code ebenfalls in die Klasse UserInputs ausgelagert wird und somit zwei Zeile Code in den Klassen gespart werden können. Als Beispiele werden die Klassen ProductView und FoodView verwendet, in denen die Änderungen zu sehen sind. In den Klassen RecipeView und ShoppingListView wird ebenfalls die Auswahl von mehreren Lebensmitteln aus einer Liste ausgelagert, dieser Code ist aber nicht in den unteren Beispielen zu sehen. Ein weiterer Vorteil ist, dass nun der ausgelagerte Code nur an einmal verändert werden muss, falls Änderungen an dem Code gemacht werden müssen und nicht in jeder Klasse einzeln.

Vorher:

ProductView:

```
public class ProductView
    {
        string path = @"C:\Users\user\source\repos\Einkaufsliste\Einkaufsliste\Prod-
ucts.json";
        ReadValues readValues = new ReadValues();
        OutputValues outputValues = new OutputValues();
        ProductOutputs productOutputs = new ProductOutputs();
        ProductPlugin productPlugin = new ProductPlugin();
        ProductManager productManager = new ProductManager();
        UserInputs userInputs = new UserInputs();
        public void createProduct()
        {
            Price price = new Price();
            double priceDouble;
            string name;
            List<Product> products = productPlugin.getProductList();


            outputValues.enterNameMessage();
            name = readValues.ReadString();

            productOutputs.enterPriceMessage();
            priceDouble = readValues.ReadDouble();
            price.price = priceDouble; outputValues.enterNameMessage();
            name = readValues.ReadString();

            productOutputs.enterPriceMessage();
            priceDouble = readValues.ReadDouble();
            price.price = priceDouble;


            Product product = productManager.createProduct(name, price);
            products.Add(product);
            productPlugin.saveProductList(products);
        }


        public void readProductList(List<Product> products = null)
        {
            if (products == null)
            {
                products = productPlugin.getProductList();
            }
            foreach (Product product in products)
            {
                productOutputs.writeProduct(product);
            }
        }
    }
```

FoodView:

```
public class FoodView
    {
        FoodOutputs foodOutputs = new FoodOutputs();
        ReadValues readValues = new ReadValues();
        FoodPlugin foodPlugin = new FoodPlugin();
        FoodManager foodManager = new FoodManager();
        OutputValues outputValues = new OutputValues();
        UserInputs userInputs = new UserInputs();
        public void createFood()
        {
            Price price = new Price();
            double priceDouble;
            int weight;
            string name;
            List<Food> foods = foodPlugin.getFoodList();

            outputValues.enterNameMessage();
            name = readValues.ReadString();

            foodOutputs.enterPriceMessage();
            priceDouble = readValues.ReadDouble();
            price.price = priceDouble;

            foodOutputs.enterWeightMessage();
            weight = readValues.ReadInt();

            Food food = foodManager.createFood(name, weight, price);
            foodOutputs.writeFood(food);
            foods.Add(food);
            foodPlugin.saveFood(foods);
        }

        public void readFoodList(List<Food> foods = null)
        {
            if (foods == null)
            {
                foods = foodPlugin.getFoodList();
            }
            foreach (Food food in foods)
            {
                foodOutputs.writeFood(food);
            }
        }
    }
```

Nachher:

ProductView:

```
public class ProductView
    {
        string path = @"C:\Users\user\source\repos\Einkaufsliste\Einkaufsliste\Prod-
ucts.json";
        ReadValues readValues = new ReadValues();
        OutputValues outputValues = new OutputValues();
        ProductOutputs productOutputs = new ProductOutputs();
        ProductPlugin productPlugin = new ProductPlugin();
        ProductManager productManager = new ProductManager();
        UserInputs userInputs = new UserInputs();
        public void createProduct()
        {
            Price price = new Price();
            double priceDouble;
            string name;
            List<Product> products = productPlugin.getProductList();


            name = userInputs.getName();

            price = userInputs.getPrice();


            Product product = productManager.createProduct(name, price);
            products.Add(product);
            productPlugin.saveProductList(products);
        }


        public void readProductList(List<Product> products = null)
        {
            if (products == null)
            {
                products = productPlugin.getProductList();
            }
            foreach (Product product in products)
            {
                productOutputs.writeProduct(product);
            }
        }
    }
```

FoodView:

```
public class FoodView
{
    FoodOutputs foodOutputs = new FoodOutputs();
    ReadValues readValues = new ReadValues();
    FoodPlugin foodPlugin = new FoodPlugin();
    FoodManager foodManager = new FoodManager();
    OutputValues outputValues = new OutputValues();
    UserInputs userInputs = new UserInputs();
    public void createFood()
    {
        Price price = new Price();
        double priceDouble;
        int weight;
        string name;
        List<Food> foods = foodPlugin.getFoodList();

        name = userInputs.getName();

        price = userInputs.getPrice();

        foodOutputs.enterWeightMessage();
        weight = readValues.ReadInt();

        Food food = foodManager.createFood(name, weight, price);
        foodOutputs.writeFood(food);
        foods.Add(food);
        foodPlugin.saveFood(foods);
    }

    public void readFoodList(List<Food> foods = null)
    {
        if (foods == null)
        {
            foods = foodPlugin.getFoodList();
        }
        foreach (Food food in foods)
        {
            foodOutputs.writeFood(food);
        }
    }
}
```

UserInputs:

```
public class UserInputs
{
    ReadValues  readValues = new ReadValues();
    OutputValues outputValues = new OutputValues();
    public string getName()
    {
        outputValues.enterNameMessage();
        string name = readValues.ReadString();
        return name;
    }
    public Price getPrice()
    {
        Price price = new Price();
        Console.WriteLine("Please enter the price.");
        double priceDouble = readValues.ReadDouble();
        price.price = priceDouble;
        return price;
    }
    public List<Food> chooseFoods(List<Food> foodList)
    {
        string foodString = "";
        List<Food> foods = new List<Food>();
        Food addedFood = new Food();
        foreach (var fd in foodList)
        {
            Console.WriteLine(fd.Name);
        }
        while (foodString != null && foodString != "q")
        {
            foodString = readValues.ReadString();
            if (foodString != null && foodString != "q")
            {
                addedFood = foodList.FirstOrDefault(f => f.Name == foodString);
                if (addedFood != null)
                {
                    foods.Add(addedFood);
                }
                outputValues.closeEntryMessage();
            }
        }
        return foods;
    }
}
```

Long Method:

Commit: Entfernen von dupliziertem Code (25.05.2022) (`e6235266539881aa6554c7025762ccde446b6f12`)

In der Klasse RecipeView existiert die Methode createRecipe, welche vor den Änderungen eine Größe von 33 Zeilen besitzt. Da innerhalb der Methode eine große Schleife genutzt wird, welche für die Auswahl von Nahrungsmittel dient, wurde die Schleife in eine eigene Methode in der Klasse UserInputs ausgelagert. Durch das Auslagern der Schleife als Methode und benennen der Methode der Schleife ist der Code der Methode createRecipe wesentlich übersichtlicher und einfacher zu lesen. Nach den Änderungen ist die Methode createRecipe nur noch 23 Zeilen groß.

Vorher:

```
public void createRecipe()
{
    List<Food> foods = new List<Food>();
    List<Food> foodList = foodPlugin.getFoodList();
    string name = "";
    string name = "";

    outputValues.enterNameMessage();
     name = readValues.ReadString();

    foodOutputs.chooseFoodMessage();

    foreach (var fd in foodList)
    {
        Console.WriteLine(fd.Name);
    }
    while (foodString != null && foodString != "q")
    {
        foodString = readValues.ReadString();
        if (foodString != "q" && foodString != null)
        {
            var addedFood = foodList.FirstOrDefault(f -> f.Name == food);
            foods.Add(addedFood);
            outputValues.closeEntryMessage();
        }
    }

    recipeOutputs.enterDescriptionMessage();
    string desc = readValues.ReadString();

    if(name != null && name != "")
    {
        Recipe recipe = recipeManager.createRecipe(name, foodIds, desc);
        recipePlugin.saveRecipe(recipe);
    }
}
```

Nachher:

Klasse RecipeView:

```
public void createRecipe()
{
    List<Food> foods = new List<Food>();
    List<Guid> foodIds = new List<Guid>();
    List<Food> foodList = foodPlugin.getFoodList();
    string name = "";

    name = userInputs.getName();

    foodOutputs.chooseFoodMessage();

    foods = userInputs.chooseFoods(foodList);
    foreach(Food food in foods)
    {
        foodIds.Add(food.Id);
    }

    recipeOutputs.enterDescriptionMessage();
    string desc = readValues.ReadString();

    if(name != null && name != "")
    {
        Recipe recipe = recipeManager.createRecipe(name, foodIds, desc);
        recipePlugin.saveRecipe(recipe);
    }
}
```

Klasse UserInputs:

```
public List<Food> chooseFoods(List<Food> foodList)
{
    string foodString = "";
    List<Food> foods = new List<Food>();
    Food addedFood = new Food();
    foreach (var fd in foodList)
    {
        Console.WriteLine(fd.Name);
    }
    while (foodString != null && foodString != "q")
    {
        foodString = readValues.ReadString();
        if (foodString != null && foodString != "q")
        {
            addedFood = foodList.FirstOrDefault(f => f.Name == foodString);
            if (addedFood != null)
            {
                foods.Add(addedFood);
            }
            outputValues.closeEntryMessage();
        }
    }
    return foods;
}
```

2 Refactorings

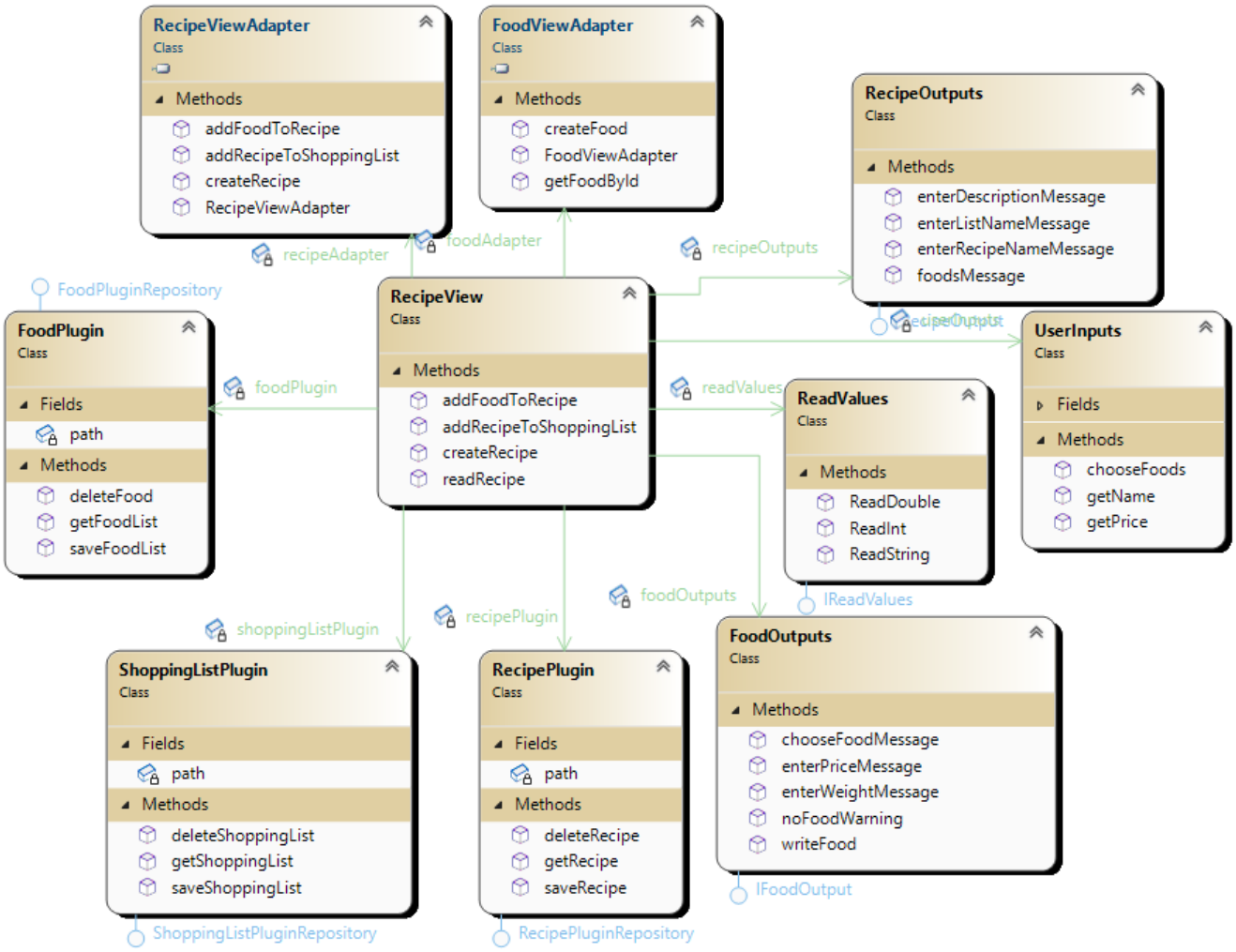
Extract Method:

Commit: Entfernen von dupliziertem Code (25.05.2022) (`e6235266539881aa6554c7025762ccde446b6f12`)

Da in der Klasse RecipeView zweimal derselbe Code in zwei unterschiedlichen Methoden für das Auswählen von Lebensmitteln existiert. Auch kommt der gleiche Code in der Klasse ShoppingListView vor. Aus diesem Grund wurde dieser Code in eine eigene Methode in eine andere Klasse ausgelagert, da falls sich der Code für die Auswahl der Lebensmittel ändern sollte, der Code nur an einer der Stellen verändert werden, anstatt an mehreren. Ebenfalls kann falls an einer anderen Stelle eine Auswahl von mehreren Lebensmitteln gebraucht, wird diese Methode aufgerufen werden, anstatt dass der Code noch einmal neu geschrieben werden muss.

Vorher:

Nachher:

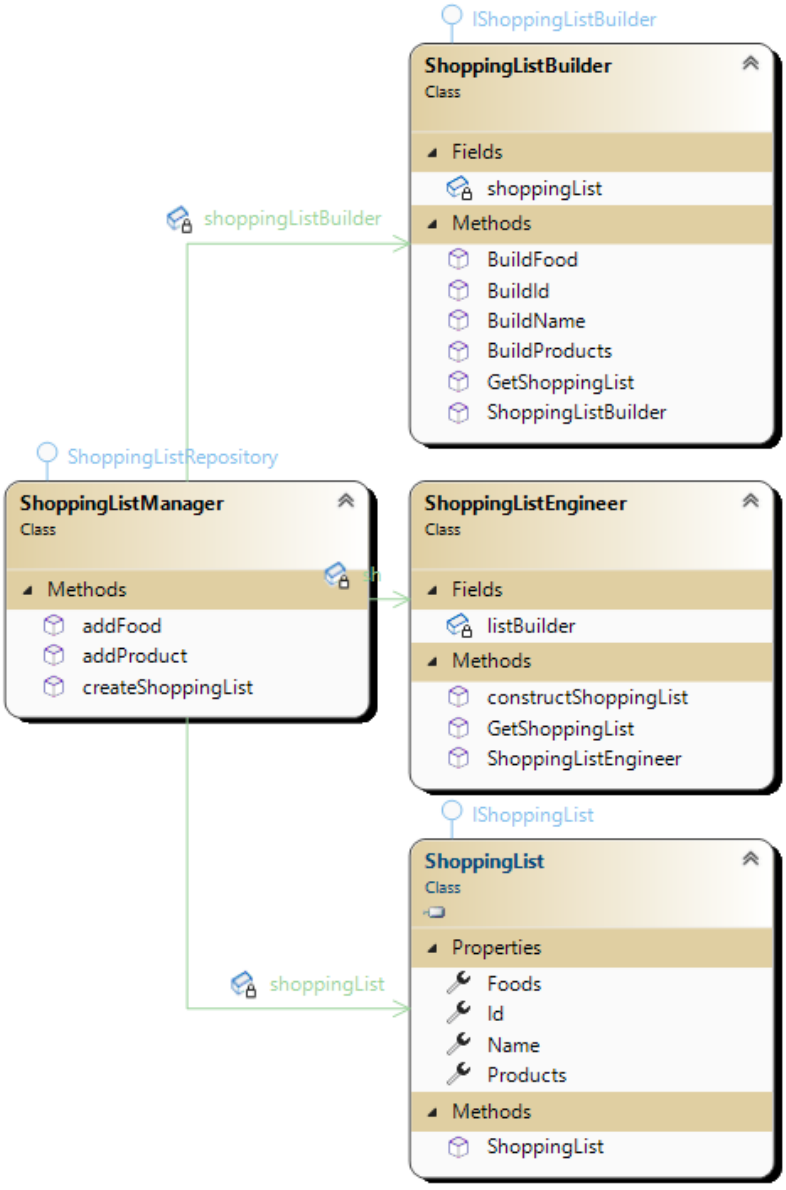

RecipeViewAdapter
Class
Methods
addFoodToRecipe
addRecipeToShoppingList
createRecipe
RecipeViewAdapter
FoodViewAdapter
Class
Methods
createFood
FoodViewAdapter
getFoodById
RecipeOutputs
Class
Methods
enterDescriptionMessage
enterListNameMessage
enterRecipeNameMessage
foodsMessage
recipeAdapter
foodAdapter
recipeOutputs
FoodPluginRepository
FoodPlugin
Class
Fields
path
Methods
deleteFood
getFoodList
saveFoodList
RecipeView
Class
Methods
addFoodToRecipe
addRecipeToShoppingList
createRecipe
readRecipe
UserInputs
Class
Fields
Methods
chooseFoods
getName
getPrice
foodPlugin
readValues
ReadValues
Class
Methods
ReadDouble
ReadInt
ReadString
IReadValues
shoppingListPlugin
recipePlugin
foodOutputs
ShoppingListPlugin
Class
Fields
path
Methods
deleteShoppingList
getShoppingList
saveShoppingList
RecipePlugin
Class
Fields
path
Methods
deleteRecipe
getRecipe
saveRecipe
FoodOutputs
Class
Methods
chooseFoodMessage
enterPriceMessage
enterWeightMessage
noFoodWarning
writeFood
IFoodOutput
ShoppingListPluginRepository
RecipePluginRepository

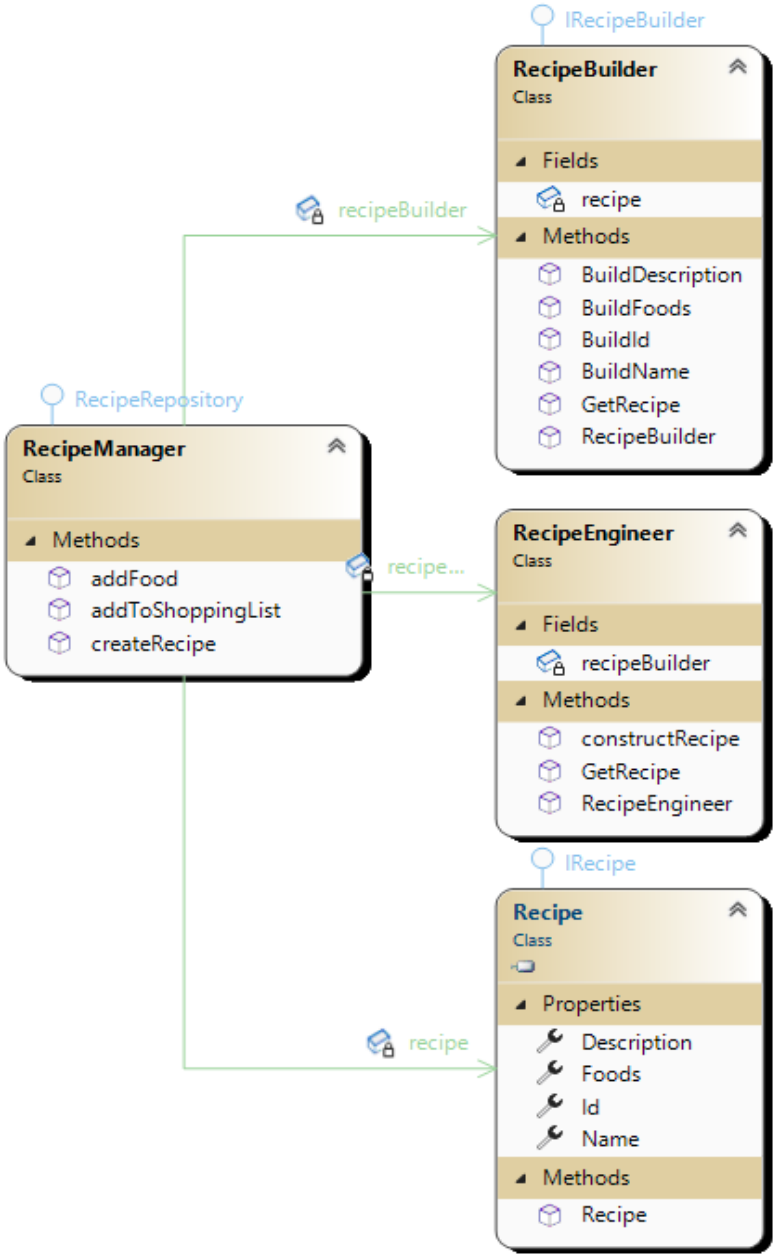
Rename Method:

Commit: Umbenennen von Methodennamen (26.05.2022) (`ab8a6b2c4741cf5c57fac8fc8939447111d9c20b`)

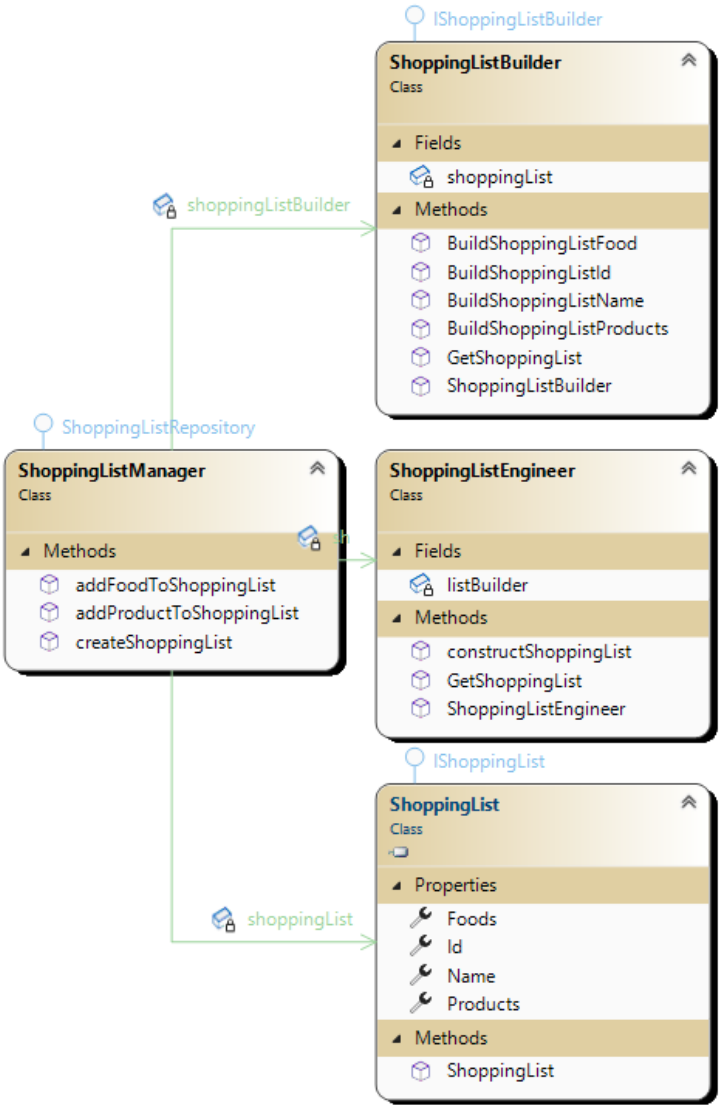
Da innerhalb des Projektes es bei einigen Methoden nicht komplett klar ist, welche Funktion die Methode genau ausführt, werden die Methoden umbenannt. So werden zum Beispiel die Methoden addFood und addProduct der Klasse ShoppingListManager umbenannt in addFoodToShoppingList und addProductToShoppingList. Auch in der Klasse RecipeManager werden die Methoden addFood und addToShoppingList zu addFoodToRecipe und addRecipeToShoppingList umbenannt.

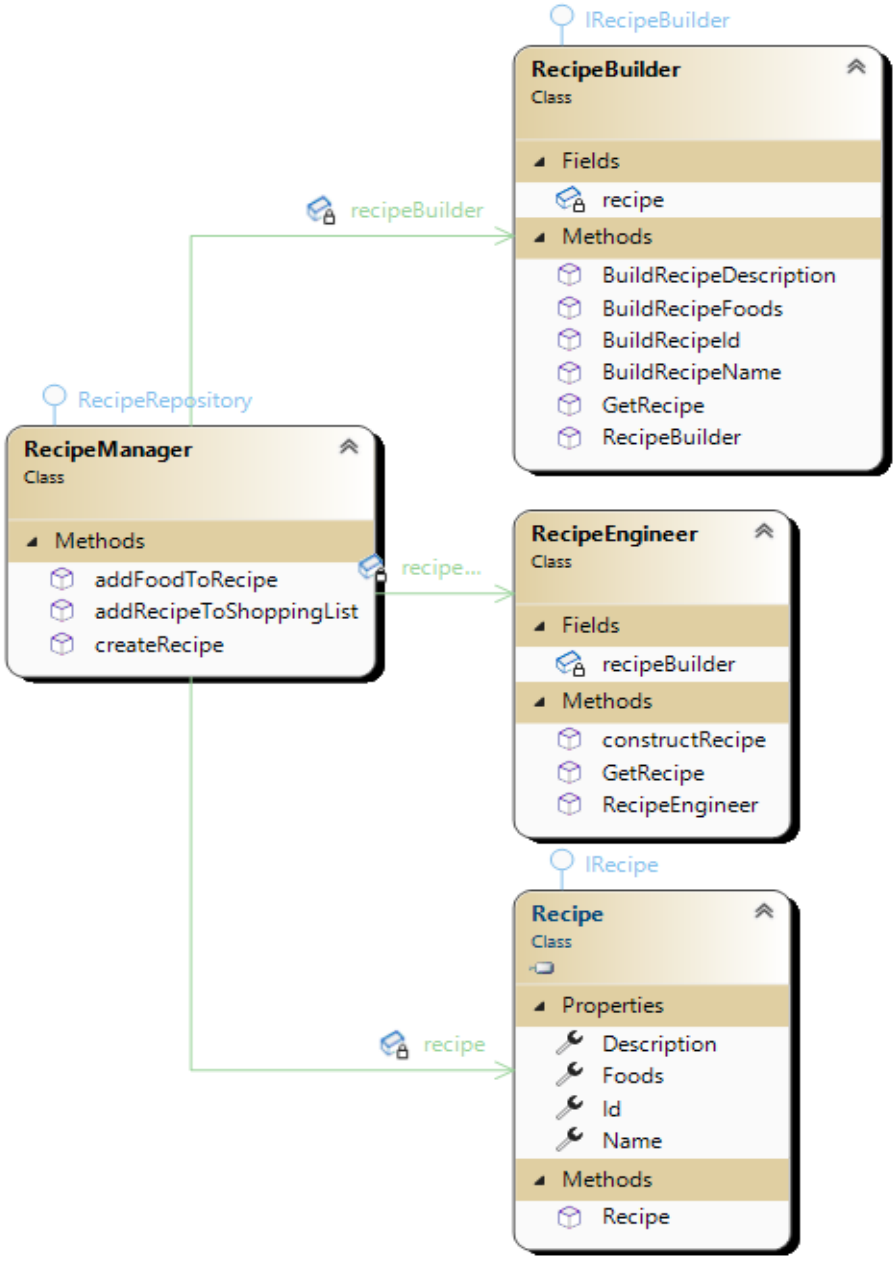
Vorher:




IRecipeBuilder
RecipeBuilder
Class
Fields
recipe
Methods
BuildDescription
BuildFoods
BuildId
BuildName
GetRecipe
RecipeBuilder
recipeBuilder
RecipeRepository
RecipeManager
Class
Methods
addFood
addToShoppingList
createRecipe
recipe...
RecipeEngineer
Class
Fields
recipeBuilder
Methods
constructRecipe
GetRecipe
RecipeEngineer
IRecipe
Recipe
Class
Properties
Description
Foods
Id
Name
Methods
Recipe
recipe

Nachher:

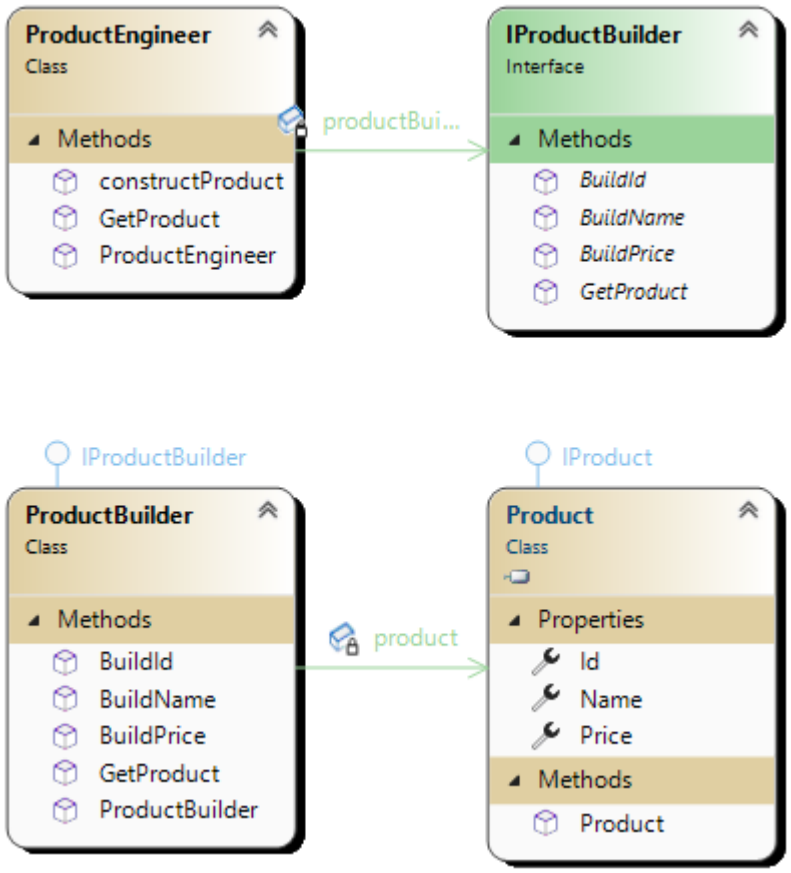

IShoppingListBuilder
ShoppingListBuilder
Class
Fields
shoppingList
Methods
BuildShoppingListFood
BuildShoppingListId
BuildShoppingListName
BuildShoppingListProducts
GetShoppingList
ShoppingListBuilder
shoppingListBuilder
ShoppingListRepository
ShoppingListManager
Class
Methods
addFoodToShoppingList
addProductToShoppingList
createShoppingList
ShoppingListEngineer
Class
Fields
listBuilder
Methods
constructShoppingList
GetShoppingList
ShoppingListEngineer
IShoppingList
ShoppingList
Class
Properties
Foods
Id
Name
Products
Methods
ShoppingList
shoppingList


IRecipeBuilder
RecipeBuilder
Class
Fields
recipe
Methods
BuildRecipeDescription
BuildRecipeFoods
BuildRecipeId
BuildRecipeName
GetRecipe
RecipeBuilder
recipeBuilder
RecipeRepository
RecipeManager
Class
Methods
addFoodToRecipe
addRecipeToShoppingList
createRecipe
recipe...
RecipeEngineer
Class
Fields
recipeBuilder
Methods
constructRecipe
GetRecipe
RecipeEngineer
IRecipe
Recipe
Class
Properties
Description
Foods
Id
Name
Methods
Recipe
recipe

Kapitel 8: Entwurfsmuster

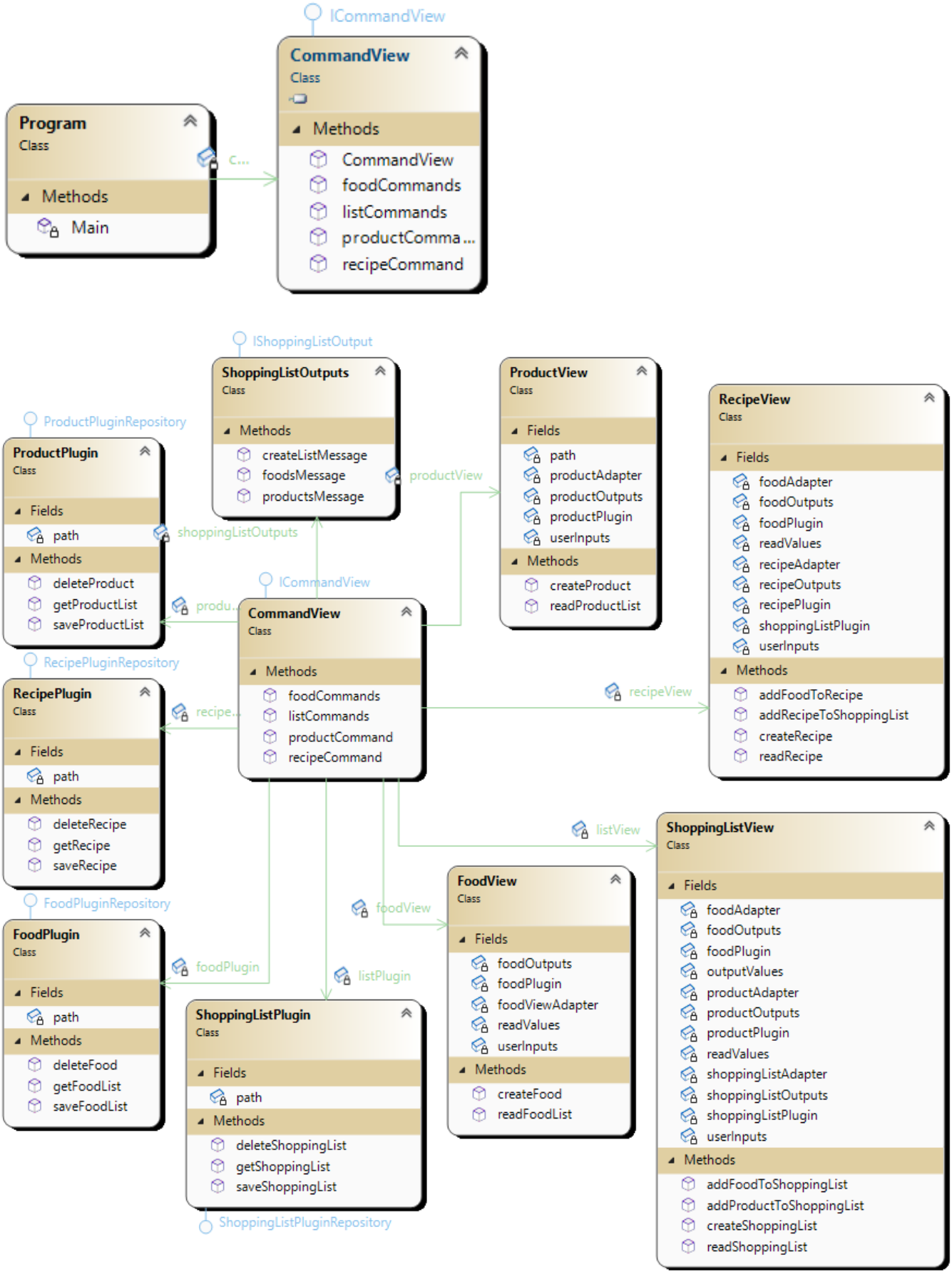
Entwurfsmuster: [Builder Pattern]

Das Builder Pattern wird genutzt, um die verschiedenen Objekte der Domäne innerhalb des Codes zu definieren. Der Einsatz des Builder Pattern ist sinnvoll, da so die Objekte schnell und einfach um verschiedene Arten der Objekte erweitert werden können, durch das Hinzufügen weitere Builder-Klassen, welche das Interface IProductBuilder implementieren. Auch kann sichergegangen werden, dass die erstellten Objekte alle Parameter, welche zur Erstellung der Objekte benötigt werden, erhalten sind und somit keine Fehlerhaften Objekte erstellt werden.



Entwurfsmuster: [Facade]

Als zweites Entwurfsmuster wurde das Muster Facade genutzt. Ein Beispiel ist hierfür die Klasse CommandView, in der die Klassen FoodPlugin, FoodView, ProductPlugin, ProductView, ShoppingListPlugin, ShoppingListView, RecipePlugin und RecipeView erstellt werden, um somit die Benutzeraufrufe zu den jeweiligen Klassen zu delegieren. Durch diese Klasse bleibt die Main Methode der Klasse Program in dem Projekt Einkaufsliste übersichtlich und versteckt die Komplexität des Programmes vor dem*der Nutzer*in.


ICommandView
CommandView
Class
Methods
CommandView
foodCommands
listCommands
productComma...
recipeCommand
Program
Class
Methods
Main
c...
IShoppingListOutput
ShoppingListOutputs
Class
Methods
createListMessage
foodsMessage
productsMessage
productView
ProductView
Class
Fields
path
productAdapter
productOutputs
productPlugin
userInputs
Methods
createProduct
readProductList
RecipeView
Class
Fields
foodAdapter
foodOutputs
foodPlugin
readValues
recipeAdapter
recipeOutputs
recipePlugin
shoppingListPlugin
userInputs
Methods
addFoodToRecipe
addRecipeToShoppingList
createRecipe
readRecipe
ProductPluginRepository
ProductPlugin
Class
Fields
path
Methods
deleteProduct
getProductList
saveProductList
shoppingListOutputs
produ...
ICommandView
CommandView
Class
Methods
foodCommands
listCommands
productCommand
recipeCommand
RecipePluginRepository
RecipePlugin
Class
Fields
path
Methods
deleteRecipe
getRecipe
saveRecipe
recipe...
recipeView
listView
ShoppingListView
Class
Fields
foodAdapter
foodOutputs
foodPlugin
outputValues
productAdapter
productOutputs
productPlugin
readValues
shoppingListAdapter
shoppingListOutputs
shoppingListPlugin
userInputs
Methods
addFoodToShoppingList
addProductToShoppingList
createShoppingList
readShoppingList
FoodPluginRepository
FoodPlugin
Class
Fields
path
Methods
deleteFood
getFoodList
saveFoodList
foodPlugin
foodView
FoodView
Class
Fields
foodOutputs
foodPlugin
foodViewAdapter
readValues
userInputs
Methods
createFood
readFoodList
listPlugin
ShoppingListPlugin
Class
Fields
path
Methods
deleteShoppingList
getShoppingList
saveShoppingList
ShoppingListPluginRepository