# Programmentwurf
# Cookbook

Name: Schmidt Marco
Martrikelnummer:

Abgabedatum: 28.05.2023

# Kapitel 1: Einführung

## Übersicht über die Applikation

Die Applikation ist ein digitales Rezeptebuch, das es Benutzern ermöglicht, ihre Rezepte an einem zentralen Ort zu speichern, zu organisieren und zu verwalten. Benutzer können neue Rezepte hinzufügen, bestehende Rezepte suchen, bearbeiten und löschen. Die Rezepte werden in einer Textdatei gespeichert, die von der Applikation gelesen und geschrieben wird.

Eine besondere Funktion der Applikation ist die Möglichkeit, eine Einkaufsliste zu erstellen, die auf den Zutaten der ausgewählten Rezepte basiert. Benutzer können die Zutaten für ein Gericht in die Einkaufsliste einlegen und die Anzahl der Personen angeben, für die sie kochen möchten. Die Applikation berechnet automatisch die benötigte Menge an Zutaten basierend auf der Anzahl der Personen und fügt sie der Einkaufsliste hinzu. Dies spart Benutzern Zeit und Mühe beim Schreiben von Einkaufslisten und stellt sicher, dass sie alle benötigten Zutaten haben, wenn sie kochen möchten.

Die Applikation soll Benutzern helfen, ihre Rezepte besser zu organisieren und zu finden. Oft haben Benutzer viele Rezepte in verschiedenen Büchern, auf Zetteln oder im Internet gespeichert. Die Applikation bietet eine zentrale Plattform, auf der Benutzer ihre Rezepte speichern und organisieren können. Benutzer können auch nach Rezepten suchen, indem sie nach Zutaten oder Schlagwörtern suchen, was das Auffinden von Rezepten erleichtert.

Insgesamt bietet die Applikation eine praktische und effektive Möglichkeit, Rezepte zu speichern, zu organisieren und zu verwalten, sowie das Schreiben von Einkaufslisten zu erleichtern.

## Wie startet man die Applikation?

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

## Wie testet man die Applikation?

*[Wie testet man die Applikation? Welche Voraussetzungen werden benötigt? Schritt-für-Schritt-Anleitung]*

# Kapitel 2: Clean Architecture

## Was ist Clean Architecture?

Unter Clean Architecture versteht man ein Softwareentwurfsprinzip, das von Robert C. Martin erfunden wurde. Es hilft dabei Softwareanwendungen so zu strukturieren, dass sie leicht lesbar, wartbar und erweiterbar sind.

Dabei besteht Clean Architecture aus verschiedenen Schichten, die jeweils eine spezifische Funktion erfüllen. Die schichten lauten: Benutzerschnittstelle, Schnittstellenadapter, Entitäten und externen Bibliotheken und Frameworks.

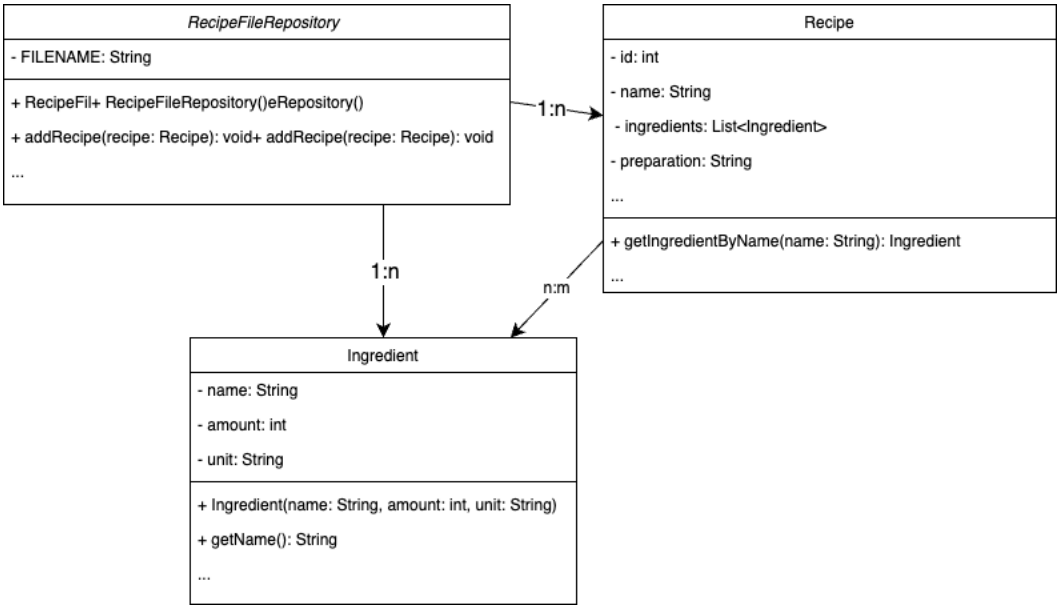
Das Ziel der Clean Architecture ist es, eine klare Trennung zwischen den verschiedenen Schichten zu schaffen, um die Abhängigkeiten zu minimieren und die Wartbarkeit zu verbessern. Dadurch wird es einfacher, Änderungen an der Anwendung vorzunehmen, ohne dass sich dies auf andere Teile der Anwendung auswirkt.

## Analyse der Dependency Rule

*[(1 Klasse, die die Dependency Rule einhält und eine Klasse, die die Dependency Rule verletzt); jeweils UML der Klasse 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]*

Die Dependency Rule (auch Abhängigkeitsregel genannt) ist ein Prinzip der Softwarearchitektur, das besagt, dass Abhängigkeiten zwischen Komponenten nur in eine Richtung zeigen sollten. Konkret bedeutet dies, dass eine Komponente A von einer anderen Komponente B abhängen kann, aber B nicht von A abhängen sollte.

## *Positiv-Beispiel: Dependency Rule*




**Analyse der Abhängigkeiten der Klasse RecipeFileRepository**

Die Klasse RecipeFileRepository hängt von der Klasse Recipe und der Klasse Ingredient ab, da sie Instanzen von Recipe und Ingredient erstellt, um sie in einer Datei zu speichern, zu suchen, zu aktualisieren und zu löschen. Die Klasse Recipe hängt von der Klasse Ingredient ab, da sie eine Liste von Ingredient-Objekten enthält.

Insgesamt erfüllen die drei Klassen die Dependency Rule, da die Abhängigkeiten nur in eine Richtung zeigen: RecipeFileRepository hängt von Recipe und Ingredient ab, aber Recipe und Ingredient hängen nicht von RecipeFileRepository ab.

### *Negativ-Beispiel: Dependency Rule*

//TODO

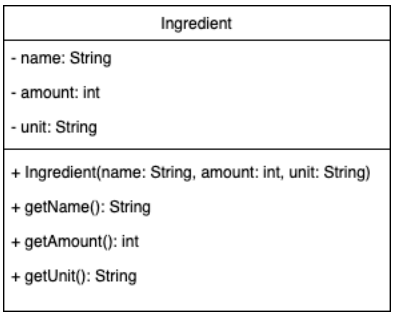
## Analyse der Schichten

Die Clean-Architecture besteht aus vier Schichten: der Domain-Schicht, der Use-Case-Schicht, der Interface-Adapter-Schicht und der Framework- und Driver-Schicht. Jede Schicht hat eine bestimmte Verantwortung und Abhängigkeiten zwischen den Schichten sollten immer von innen nach außen verlaufen.

### *Schicht: Domain-Schicht*

Die Klasse **Ingredient** repräsentiert eine Zutat in einem Rezept. Sie hat drei private Attribute: **name**, **amount** und **unit**, die den Namen, die Menge und die Einheit der Zutat speichern. Die Klasse hat auch drei öffentliche Methoden, **getName()**, **getAmount()** und **getUnit()**, die jeweils den Namen, die Menge und die Einheit der Zutat zurückgeben. Die Methode **toString()** gibt eine String-Darstellung der Zutat zurück.

Die Klasse **Ingredient** gehört zur Domain-Schicht der Clean-Architecture. Sie enthält nur die grundlegenden Informationen über eine Zutat und hat keine Abhängigkeiten zu anderen Schichten. Die Klasse ist einfach und gut strukturiert und kann leicht erweitert werden, wenn weitere Informationen über eine Zutat benötigt werden.



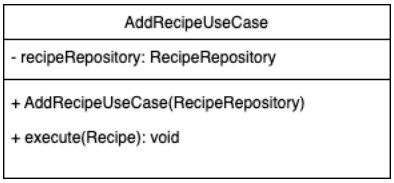

### *Schicht: Use-Case-Schicht*

Die Klasse **AddRecipeUseCase** gehört zur Use-Case-Schicht der Clean-Architecture. Sie implementiert die Geschäftslogik für das Hinzufügen eines Rezepts und verwendet dafür das **RecipeRepository**-Interface.

Die Klasse hat ein privates Attribut **recipeRepository**, das eine Instanz des **RecipeRepository**-Interface speichert. Der Konstruktor der Klasse nimmt eine Instanz des **RecipeRepository**-Interface als Parameter.

Die Klasse hat eine öffentliche Methode **execute()**, die ein **Recipe**-Objekt als Parameter nimmt und das Rezept zum **RecipeRepository** hinzufügt. Die Methode ruft die **addRecipe()**-Methode des **RecipeRepository**-Interface auf, um das Rezept hinzuzufügen.

Insgesamt ist die Klasse **AddRecipeUseCase** gut strukturiert und folgt den Prinzipien der Clean-Architecture. Sie hat keine Abhängigkeiten zur Interface-Adapter-Schicht und kann leicht durch eine andere Implementierung des **RecipeRepository**-Interface ersetzt werden, ohne dass Änderungen an der Klasse vorgenommen werden müssen.




# Kapitel 3: SOLID

## Analyse Single-Responsibility-Principle (SRP)

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

Das Single-Responsibility-Principle (SRP) ist ein Prinzip der objektorientierten Programmierung, das besagt, dass eine Klasse nur eine einzige Verantwortlichkeit haben sollte. Das bedeutet, dass eine Klasse nur für eine bestimmte Aufgabe oder Funktion zuständig sein sollte und keine anderen Aufgaben oder Funktionen übernehmen sollte.

### *Positiv-Beispiel*

//TODO UML Diagramm

#### Beschreibung der Aufgabe

Die Klasse **Ingredient** repräsentiert eine einzelne Zutat, die in einem Rezept verwendet werden kann. Die Klasse hat drei private Attribute, **name**, **amount** und **unit**, die den Namen, die Menge und die Einheit der Zutat darstellen. Die Klasse hat vier öffentliche Methoden, **getName()**, **getAmount()**, **getUnit()** und **toString()**, die jeweils den Namen, die Menge, die Einheit und eine String-Darstellung der Zutat zurückgeben.

Die Klasse erfüllt das Single-Responsibility-Prinzip (SRP), da sie nur für die Darstellung einer einzelnen Zutat zuständig ist und keine anderen Verantwortlichkeiten hat. Die Klasse hat keine Abhängigkeiten zu anderen Klassen und ist gut isoliert, was dazu beitragen kann, die Wartbarkeit und Erweiterbarkeit der Anwendung zu verbessern.

### *Negativ-Beispiel*

/TODO UML

**Beschreibung der Aufgabe**

Die Klasse **RecipeBookApplication** ist eine Anwendungsklasse, die für die Verwaltung von Rezepten und Einkaufslisten zuständig ist. Die Klasse hat sechs private Attribute, die jeweils eine Use-Case-Klasse darstellen, die für die Durchführung von Aufgaben wie dem Hinzufügen, Suchen, Aktualisieren und Löschen von Rezepten zuständig sind. Die Klasse hat auch zehn private Methoden, die jeweils für eine bestimmte Aufgabe zuständig sind, wie z.B. das Hinzufügen eines Rezepts oder das Suchen nach Rezepten.

Das Problem mit dieser Klasse ist, dass sie mehrere Verantwortlichkeiten hat und somit das Single-Responsibility-Prinzip (SRP) nicht erfüllt. Die Klasse ist für die Verwaltung von Rezepten und Einkaufslisten zuständig, was zwei unterschiedliche Verantwortlichkeiten sind. Die Klasse hat auch eine hohe Anzahl von Abhängigkeiten zu anderen Klassen, was darauf hinweist, dass sie nicht gut isoliert ist.

Eine mögliche Lösung für dieses Problem wäre die Aufteilung der Verantwortlichkeiten in separate Klassen. Zum Beispiel könnte eine Klasse **RecipeManager** erstellt werden, die für die Verwaltung von Rezepten zuständig ist, und eine separate Klasse **ShoppingListManager**, die für die Verwaltung von Einkaufslisten zuständig ist. Dies würde dazu beitragen, die Verantwortlichkeiten klar zu definieren und die Wartbarkeit und Erweiterbarkeit der Anwendung zu verbessern.

## Analyse Open-Closed-Principle (OCP)

*[jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML der Klasse 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)?]*

Das Open-Closed-Prinzip (OCP) ist ein Prinzip der objektorientierten Programmierung, das besagt, dass eine Softwarekomponente (z.B. eine Klasse, ein Modul oder eine Funktion) offen für Erweiterungen sein sollte, aber geschlossen für Änderungen. Das bedeutet, dass eine Komponente so gestaltet werden sollte, dass sie leicht erweitert werden kann, ohne dass ihre bestehende Funktionalität geändert werden muss.

### *Positiv-Beispiel*

Die Klasse **Recipe** erfüllt das Open-Closed-Prinzip, da sie offen für Erweiterungen ist, aber geschlossen für Änderungen bleibt. Die Klasse bietet eine Reihe von Methoden und Attributen an, die es anderen Klassen ermöglichen, mit Rezepten zu arbeiten, ohne dass die Klasse selbst geändert werden muss.

Zum Beispiel können andere Klassen die **getIngredients()**-Methode verwenden, um eine Liste der Zutaten eines Rezepts abzurufen, oder die **getPreparation()**-Methode, um die Zubereitung eines Rezepts abzurufen. Die Klasse bietet auch eine **getIngredientByName()**-Methode an, die es anderen Klassen ermöglicht, eine Zutat anhand ihres Namens abzurufen.

### *Negativ-Beispiel*

Die Klasse **ShoppingListFileRepository** implementiert das **ShoppingListRepository**-Interface und bietet Methoden zum Speichern und Laden von **ShoppingList**-Objekten. Die Implementierung dieser Methoden erfolgt durch das Lesen und Schreiben von **ShoppingListItem**-Objekten in eine Textdatei.

Das Open-Closed-Prinzip (OCP) wird von der Klasse **ShoppingListFileRepository** nicht erfüllt, da sie Änderungen an der Implementierung erfordert, um neue Arten von **ShoppingList**-Objekten zu unterstützen. Insbesondere verwendet die Klasse die **ShoppingListItem**-Klasse, um die Elemente der Einkaufsliste zu speichern und zu laden. Wenn sich die Implementierung von **ShoppingList** ändert und **ShoppingListItem** nicht mehr verwendet wird, muss die Implementierung von **ShoppingListFileRepository** geändert werden.

Um das OCP zu erfüllen, könnte die Klasse **ShoppingListFileRepository** abstrakte Methoden und Interfaces verwenden, um die Implementierung von **ShoppingList** zu abstrahieren. Zum Beispiel könnte sie ein **ShoppingListSerializer**-Interface definieren, das Methoden zum Serialisieren und Deserialisieren von **ShoppingList**-Objekten bereitstellt. Eine konkrete Implementierung dieses Interfaces könnte dann verwendet werden, um **ShoppingList**-Objekte in eine Textdatei zu schreiben und aus einer Textdatei zu lesen. Auf diese Weise würde die Klasse **ShoppingListFileRepository** offen für Erweiterungen bleiben, da sie neue Implementierungen von **ShoppingList** unterstützen könnte, ohne ihre eigene Implementierung zu ändern.

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

*[jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP oder DIP); jeweils UML der Klasse und Begründung, warum man hier das Prinzip erfüllt/nicht erfüllt wird]*

*[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]*

Das Liskov-Substitution-Prinzip (LSP) besagt, dass eine Unterklasse in der Lage sein sollte, anstelle ihrer Basisklasse verwendet zu werden, ohne dass dadurch die Korrektheit des Programms beeinträchtigt wird. Mit anderen Worten, eine Unterklasse sollte alle Methoden und Eigenschaften ihrer Basisklasse erben und diese Methoden und Eigenschaften sollten in der Unterklasse auf die gleiche Weise funktionieren wie in der Basisklasse. Wenn eine Unterklasse eine Methode ihrer Basisklasse überschreibt, sollte sie dies nur tun, um das Verhalten der Methode zu erweitern, nicht um es zu ändern.

Das Interface-Segregation-Prinzip (ISP) besagt, dass eine Klasse nicht gezwungen werden sollte, Methoden zu implementieren, die sie nicht benötigt. Mit anderen Worten, ein Interface sollte nur Methoden enthalten, die für die Implementierung der Klasse relevant sind. Wenn eine Klasse ein Interface implementiert, das Methoden enthält, die sie nicht benötigt, führt dies zu unnötigem Code und erhöht die Komplexität der Klasse.

Das Dependency-Inversion-Prinzip (DIP) besagt, dass eine Klasse nicht von einer konkreten Implementierung abhängig sein sollte, sondern von einer abstrakten Schnittstelle. Mit anderen Worten, eine Klasse sollte nicht direkt auf eine andere Klasse zugreifen, sondern auf eine abstrakte Schnittstelle, die von der anderen Klasse implementiert wird. Dadurch wird die Flexibilität und Wiederverwendbarkeit des Codes erhöht, da die Implementierung der abstrakten Schnittstelle ausgetauscht werden kann, ohne dass die Klasse, die darauf zugreift, geändert werden muss.

## *Positiv-Beispiel*

Das **RecipeFileRepository**-Klasse implementiert das **RecipeRepository**-Interface, das die abstrakte Schnittstelle definiert, die von der Klasse verwendet wird. Dadurch wird sichergestellt, dass die Klasse nicht von einer konkreten Implementierung abhängt, sondern von einer abstrakten Schnittstelle.

Das **RecipeRepository**-Interface definiert alle Methoden, die von der **RecipeFileRepository**-Klasse implementiert werden müssen, um als **RecipeRepository** zu fungieren. Durch die Verwendung des Interfaces wird sichergestellt, dass die **RecipeFileRepository**-Klasse alle erforderlichen Methoden implementiert, um als **RecipeRepository** verwendet zu werden.

Das DIP wird auch dadurch eingehalten, dass die **RecipeFileRepository**-Klasse keine Abhängigkeiten zu anderen Klassen hat, die nicht von einer abstrakten Schnittstelle abhängen. Die Klasse verwendet keine konkreten Implementierungen von anderen Klassen, sondern nur abstrakte Schnittstellen.

Insgesamt zeigt das UML-Diagramm, dass die **RecipeFileRepository**-Klasse das Dependency-Inversion-Prinzip einhält, indem sie von einer abstrakten Schnittstelle abhängt und keine Abhängigkeiten zu konkreten Implementierungen anderer Klassen hat.

## *Negativ-Beispiel*

Die Klasse **RecipeBookApplication** ist ein negatives Beispiel für das Dependency-Inversion-Prinzip (DIP), da sie direkt von konkreten Implementierungen von **RecipeRepository** und **Shopping-ListRepository** abhängt, anstatt von abstrakten Schnittstellen.

Die **RecipeBookApplication**-Klasse erstellt Instanzen von **RecipeFileRepository** und **ShoppingListFileRepository**, die konkrete Implementierungen von **RecipeRepository** und **ShoppingListRepository** sind. Dadurch wird das DIP verletzt, da die Klasse direkt von konkreten Implementierungen abhängt, anstatt von abstrakten Schnittstellen.

Um das DIP einzuhalten, sollte die **RecipeBookApplication**-Klasse abstrakte Schnittstellen anstelle von konkreten Implementierungen verwenden. Zum Beispiel könnte sie **RecipeRepository** und **ShoppingListRepository** als abstrakte Schnittstellen definieren und dann Instanzen von Klassen erstellen, die diese Schnittstellen implementieren. Dadurch würde die Klasse nur von abstrakten Schnittstellen abhängen und nicht von konkreten Implementierungen.

# Kapitel 4: Weitere Prinzipien

## Analyse GRASP: Geringe Kopplung

*[jeweils eine bis jetzt noch nicht behandelte Klasse als positives und negatives Beispiel geringer Kopplung; jeweils UML Diagramm mit zusammenspielenden Klassen, Aufgabenbeschreibung und Begründung für die Umsetzung der geringen Kopplung bzw. Beschreibung, wie die Kopplung aufgelöst werden kann]*

### *Positiv-Beispiel*

### *Negativ-Beispiel*

## Analyse GRASP: Hohe Kohäsion

*[eine Klasse als positives Beispiel hoher Kohäsion; UML Diagramm und Begründung, warum die Kohäsion hoch ist]*

Die Klasse **Ingredient** hat nur drei Attribute (**name**, **amount** und **unit**) und vier Methoden (**Ingredient()**, **getName()**, **getAmount()** und **getUnit()**, **toString()**). Die Verantwortlichkeiten innerhalb der Klasse sind eng miteinander verbunden und konzentrieren sich auf ein gemeinsames Ziel, nämlich die Repräsentation einer Zutat.

Die Methoden **getName()**, **getAmount()** und **getUnit()** sind alle eng mit dem Attribut verbunden, das sie zurückgeben, und haben keine Auswirkungen auf andere Teile der Klasse oder andere Klassen. Die Methode **toString()** gibt eine Zeichenfolge zurück, die alle Attribute der Klasse enthält, was auch eng mit dem Zweck der Klasse verbunden ist.

Daher hat die Klasse **Ingredient** eine hohe Kohäsion, da die Verantwortlichkeiten innerhalb der Klasse eng miteinander verbunden sind und sich auf ein gemeinsames Ziel konzentrieren.

## Don’t Repeat Yourself (DRY)

*[ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher); begründen und Auswirkung beschreiben]*

```
private void searchRecipes() {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter search query: ");
    String query = scanner.nextLine();
    List<Recipe> recipes = searchRecipeUseCase.execute(query);
    if (recipes.isEmpty()) {
        System.out.println("No recipes found");
    } else {
        System.out.println("Found " + recipes.size() + " recipes:");
        for (int i = 0; i < recipes.size(); i++) {
            System.out.println((i + 1) + ". " + recipes.get(i).getName());
        }
        int choice;
        while (true) {
            System.out.print("Select a recipe (or enter 0 to cancel): ");
            try {
                choice = Integer.parseInt(scanner.nextLine());
                if (choice == 0) {
                    return;
                }
                if (choice < 1 || choice > recipes.size()) {
                    throw new NumberFormatException();
                }
                break;
            } catch (NumberFormatException e) {
                System.out.println("Invalid input, please enter a number
between 1 and " + recipes.size() + " (or enter 0 to cancel)");
            }
        }
        Recipe selectedRecipe = recipes.get(choice - 1);
        System.out.println(selectedRecipe);
    }
}
```

...

```
private void updateRecipe() {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter recipe name: ");
    String name = scanner.nextLine();
    List<Recipe> recipes = searchRecipeUseCase.execute(name);
    if (recipes.isEmpty()) {
        System.out.println("Recipe not found");
    } else {
        Recipe recipe = recipes.get(0);
        System.out.println("Current recipe:");
        System.out.println(recipe);
        System.out.println("Enter new recipe information (leave blank to
keep current information):");
        System.out.print("Name: ");
        String newName = scanner.nextLine();
        if (!newName.isBlank()) {
            recipe.setName(newName);
        }
        System.out.print("Ingredients (comma-separated): ");
        String newIngredientsString = scanner.nextLine();
        if (!newIngredientsString.isBlank()) {
            List<String> newIngredientsStringList =
List.of(newIngredientsString.split(","));
            List<Ingredient> newIngredients = new ArrayList<>();
            for (String ingredientString : newIngredientsStringList) {
                Ingredient ingredient = new
Ingredient(ingredientString.trim(), 0, "");
                newIngredients.add(ingredient);
```

```
private Recipe choseRecipes(List<Recipe> recipes){
    Scanner scanner = new Scanner(System.in);
    System.out.println("Found " + recipes.size() + " recipes:");
    for (int i = 0; i < recipes.size(); i++) {
        System.out.println((i + 1) + ". " + recipes.get(i).getName());
    }
    int choice;
    while (true) {
        System.out.print("Select a recipe (or enter 0 to cancel): ");
        try {
            choice = Integer.parseInt(scanner.nextLine());
            if (choice == 0) {
                return null;
            }
            if (choice < 1 || choice > recipes.size()) {
                throw new NumberFormatException();
            }
            break;
        } catch (NumberFormatException e) {
            System.out.println("Invalid input, please enter a number
between 1 and " + recipes.size() + " (or enter 0 to cancel)");
        }
    }
    return recipes.get(choice - 1);
}
```

…

```
private void searchRecipes() {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter search query: ");
    String query = scanner.nextLine();
    List<Recipe> recipes = searchRecipeUseCase.execute(query);
    if (recipes.isEmpty()) {
        System.out.println("No recipes found");
    } else {
        Recipe selectedRecipe = choseRecipes(recipes);
        if(!(selectedRecipe == null)) System.out.println(selectedRecipe);
    }
}
```

…

```
private void updateRecipe() {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter search query: ");
    String query = scanner.nextLine();
    List<Recipe> recipes = searchRecipeUseCase.execute(query);
    if (recipes.isEmpty()) {
        System.out.println("No recipes found");
    } else {
        Recipe recipe = choseRecipes(recipes);
        if (recipe == null) {
            return;
        }
        …
    }
}
```

*Abbildung 1: Code nach DRY*

Die Methode **choseRecipes(List<Recipe> recipes)** wurde hinzugefügt, um Code-Duplizierungen in den Methoden **searchRecipes()** und **updateRecipe()** zu entfernen. Beide Methoden enthielten ähnlichen Code, um den Benutzer aufzufordern, eine ausgewählte Rezept-ID einzugeben. Durch das Extrahieren dieses Codes in eine separate Methode kann dieser Code an einer einzigen Stelle gewartet werden, anstatt an mehreren Stellen im Code dupliziert zu werden.

Die Auswirkungen dieser Änderung sind, dass der Code jetzt einfacher zu warten und zu aktualisieren ist, da der gemeinsame Code in einer separaten Methode extrahiert wurde. Wenn Änderungen an diesem Code vorgenommen werden müssen, müssen sie nur an einer Stelle vorgenommen werden, anstatt an mehreren Stellen im Code. Dies verbessert die Wartbarkeit und Lesbarkeit des Codes und reduziert die Wahrscheinlichkeit von Fehlern und Bugs im Code.

# Kapitel 5: Unit Tests

## 10 Unit Tests

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

| Unit Test | Beschreibung |
|---|---|
| IngredientTest#testGetName() | Überprüft, ob die Methode **getName()** in **Ingredient** den korrekten Namen der Zutat zurückgibt. |
| IngredientTest#testGetAmount() | Überprüft, ob die Methode **getAmount()** in **Ingredient** die korrekte Menge der Zutat zurückgibt. |
| IngredientTest#testGetUnit() | Überprüft, ob die Methode **getUnit()** in **Ingredient** die korrekte Einheit der Zutat zurückgibt. |
| IngredientTest#testToString() | Überprüft, ob die Methode **toString()** in **Ingredient** die korrekte Zeichenfolge zurückgibt, die den Namen, die Menge und die Einheit der Zutat enthält. |
| RecipeTest#testGetIngredients() | Überprüft, ob die Methode **getIngredients()** die korrekte Liste von Zutaten zurückgibt. |
| RecipeTest#testGetPreparation() | Überprüft, ob die Methode **getPreparation()** die korrekte Zubereitungsanleitung zurückgibt. |
| RecipeTest#testToString() | Überprüft, ob die Methode **toString()** die korrekte Zeichenfolge zurückgibt, die alle Eigenschaften des Rezepts enthält. |
| RecipeFileRepositoryTest# testAddRecipe() | Überprüft, ob die Methode **addRecipe()** ein neues Rezept zur Datei hinzufügt und das hinzugefügte Rezept korrekt ist. |
| RecipeFileRepositoryTest# testSearchRecipes() | Überprüft, ob die Methode **searchRecipes()** eine Liste von Rezepten zurückgibt, die den Suchkriterien entsprechen. |
| RecipeFileRepositoryTest# testDeleteRecipe() | Überprüft, ob die Methode **deleteRecipe()** ein Rezept aus der Datei löscht. |

## ATRIP: Automatic

Ich habe Machen zur Automatisierung der Unit Test genutzt. Maven ist ein Build-Management-Tool, das es Entwicklern ermöglicht, ihre Projekte zu automatisieren und zu standardisieren. Einer der Hauptvorteile von Maven ist die Möglichkeit, automatisch Unit-Tests auszuführen.

Maven verwendet das Surefire-Plugin, um Unit-Tests automatisch auszuführen. Das Surefire-Plugin sucht nach Testklassen im Standardverzeichnis **src/test/java** und führt alle Methoden aus, die mit **@Test** annotiert sind.

Um die Unit-Tests automatisch auszuführen, muss lediglich der Befehl **mvn test** in der Kommandozeile ausgeführt werden. Maven kompiliert den Quellcode, erstellt die Testklassen und führt dann alle Tests aus. Das Ergebnis wird auf der Konsole angezeigt und in einem Bericht im Verzeichnis **target/surefire-reports** gespeichert.

Durch die Automatisierung der Unit-Tests mit Maven können Entwickler sicherstellen, dass ihre Code-Änderungen keine unerwarteten Auswirkungen auf die Funktionalität des Projekts haben. Die automatisierten Tests können schnell und einfach ausgeführt werden, um sicherzustellen, dass der Code fehlerfrei ist und wie erwartet funktioniert.

## ATRIP: Thorough

*[jeweils 1 positives und negatives Beispiel zu 'Thorough'; jeweils Code-Beispiel, Analyse und Begründung, was professionell/nicht professionell ist]*

## ATRIP: Professional

*[jeweils 1 positives und negatives Beispiel zu 'Professional'; jeweils Code-Beispiel, Analyse und Begründung, was professionell/nicht professionell ist]*

## Code Coverage

Current scope: all classes

Overall Coverage Summary

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| all classes | 35,7% (5/14) | 39,1% (36/92) | 29,9% (160/535) |

Coverage Breakdown

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| <empty package name> | 0% (0/1) | 0% (0/12) | 0% (0/59) |
| domain | 100% (4/4) | 92,9% (26/28) | 84,6% (55/65) |
| framework | 50% (1/2) | 58,8% (10/17) | 63,6% (105/165) |
| ui | 0% (0/1) | 0% (0/20) | 0% (0/214) |
| usecase | 0% (0/6) | 0% (0/15) | 0% (0/32) |

Current scope: all classes | domain

Coverage Summary for Package: domain

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| domain | 100% (4/4) | 92,9% (26/28) | 84,6% (55/65) |

| Class | Class, % | Method, % | Line, % |
|---|---|---|---|
| Ingredient | 100% (1/1) | 100% (5/5) | 100% (8/8) |
| Recipe | 100% (1/1) | 83,3% (10/12) | 76,3% (29/38) |
| ShoppingList | 100% (1/1) | 100% (4/4) | 88,9% (8/9) |
| ShoppingListItem | 100% (1/1) | 100% (7/7) | 100% (10/10) |

| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| all | 35% (5/14) | 39% (36/91) | 29% (160/534) |
| domain | 100% (4/4) | 92% (26/28) | 84% (55/65) |
| Ingredient | 100% (1/1) | 100% (5/5) | 100% (8/8) |
| Recipe | 100% (1/1) | 83% (10/12) | 76% (29/38) |
| RecipeRepository | 100% (0/0) | 100% (0/0) | 100% (0/0) |
| ShoppingList | 100% (1/1) | 100% (4/4) | 88% (8/9) |
| ShoppingListItem | 100% (1/1) | 100% (7/7) | 100% (10/10) |
| ShoppingListRepository | 100% (0/0) | 100% (0/0) | 100% (0/0) |
| framework | 50% (1/2) | 62% (10/16) | 64% (105/164) |
| RecipeFileRepository | 100% (1/1) | 83% (10/12) | 78% (105/133) |
| ShoppingListFileRepository | 0% (0/1) | 0% (0/4) | 0% (0/31) |
| ui | 0% (0/1) | 0% (0/20) | 0% (0/214) |
| ConsoleUI | 0% (0/1) | 0% (0/20) | 0% (0/214) |
| usecase | 0% (0/6) | 0% (0/15) | 0% (0/32) |
| AddRecipeUseCase | 0% (0/1) | 0% (0/2) | 0% (0/3) |
| DeleteRecipeUseCase | 0% (0/1) | 0% (0/2) | 0% (0/6) |
| RandomRecipeUseCase | 0% (0/1) | 0% (0/2) | 0% (0/3) |
| SearchRecipeUseCase | 0% (0/1) | 0% (0/2) | 0% (0/3) |
| ShoppingListUseCase | 0% (0/1) | 0% (0/5) | 0% (0/11) |
| UpdateRecipeUseCase | 0% (0/1) | 0% (0/2) | 0% (0/6) |
| RecipeBookApplication | 0% (0/1) | 0% (0/12) | 0% (0/59) |

Code Coverage ist ein Maß dafür, wie viel von des Code von Tests abgedeckt wird. Es gibt an, wie viele Zeilen Code, Methoden und Zweige von Ihren Tests ausgeführt werden. Ein höherer Code Coverage-Wert bedeutet, dass mehr Teile des Codes von Tests abgedeckt werden. In meinem Fall habe ich 35,7% meiner Klassen einen Test geschrieben und für 39,1% der Methoden, was 29.9% aller Line entspricht. Im Package domain habe 100% der Klassen Test geschrieben und 92.9 der Methoden damit abgedeckt. Dafür sieht man auch das ich keine einzigen Test zum Package ui und usecase geschrieben habe. Wodurch Fehler in diesen packages nicht automatisiert entdeckt werden können.

## Fakes und Mocks

*[Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten; zusätzlich jeweils UML Diagramm der Klasse]*

# Kapitel 6: Domain Driven Design

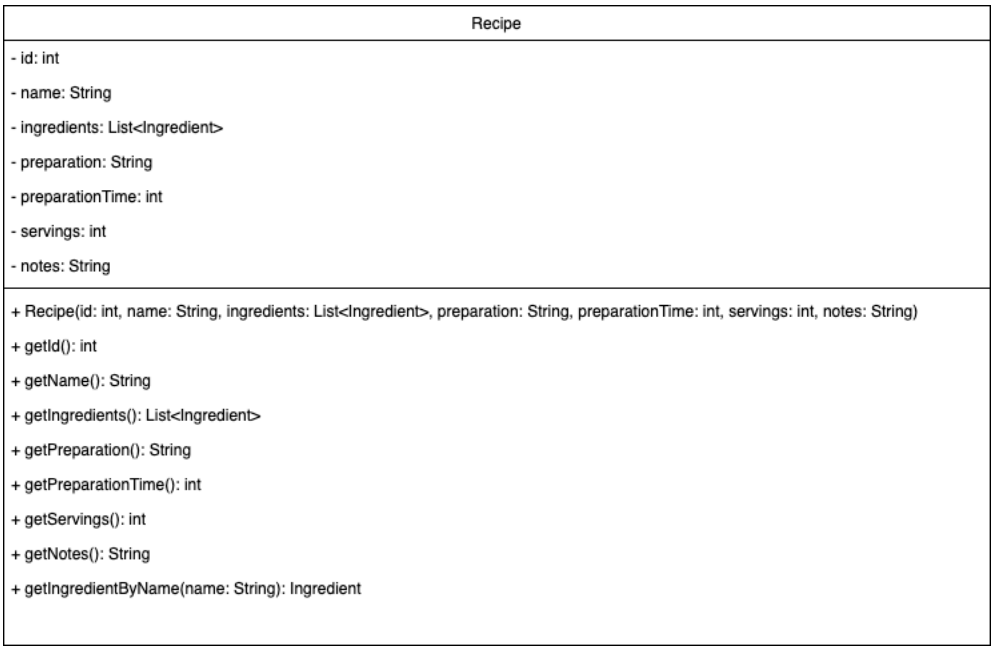
## Ubiquitous Language

| Bezeichnung | Bedeutung | Begründung |
|---|---|---|

| | | |
|---|---|---|
| Ingredient | Ein Bestandteil eines Rezepts, der eine bestimmte Menge und Einheit hat. | "Ingredient" ist ein wichtiger Begriff in der Domäne der Rezeptverwaltung und sollte daher in der Ubiquitous Language verwendet werden, um sicherzustellen, dass alle Beteiligten dasselbe Verständnis für diesen Begriff haben. |
| Recipe | Eine Sammlung von Zutaten und Anweisungen zur Zubereitung eines Gerichts. | "Recipe" ist ein wichtiger Begriff in der Domäne der Rezeptverwaltung und sollte daher in der Ubiquitous Language verwendet werden, um sicherzustellen, dass alle Beteiligten dasselbe Verständnis für diesen Begriff haben. |
| ShoppingList | Eine Liste von Zutaten, die für die Zubereitung von Gerichten benötigt werden. | "ShoppingList" ist ein wichtiger Begriff in der Domäne der Rezeptverwaltung und sollte daher in der Ubiquitous Language verwendet werden, um sicherzustellen, dass alle Beteiligten dasselbe Verständnis für diesen Begriff haben. |
| ShoppingListItem | Ein Eintrag in einer Einkaufsliste, der eine bestimmte Menge und Einheit eines Bestandteils enthält. | "ShoppingListItem" ist ein wichtiger Begriff in der Domäne der Rezeptverwaltung und sollte daher in der Ubiquitous Language verwendet werden, um sicherzustellen, dass alle Beteiligten dasselbe Verständnis für diesen Begriff haben. |

## Entities

Bei Domain Driven Design (DDD) sind Entities Objekte, die eine Identität haben und über die Zeit hinweg bestehen bleiben. Sie repräsentieren Konzepte in der Domäne, die wichtig genug sind, um als eigenständige Objekte behandelt zu werden. Entities haben normalerweise eine Lebensdauer, die unabhängig von der Lebensdauer der Anwendung ist.
In meinem Code ist die Klasse "Recipe" zum Beispiel eine Entity. Diese Klasse repräsentiert ein Rezept, das ein eigenständiges Konzept in der Domäne der Rezeptverwaltung ist. Ein Rezept hat eine Identität (durch die ID repräsentiert), die über die Zeit hinweg bestehen bleibt. Ein Rezept hat auch eine Lebensdauer, die unabhängig von der Lebensdauer der Anwendung ist.



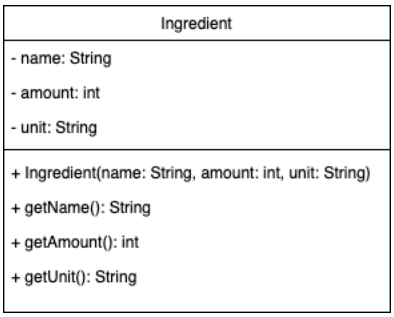

Die "Recipe"-Klasse hat eine Identität (durch die ID repräsentiert) und enthält auch eine Liste von "Ingredient"-Objekten. Die Klasse hat auch Methoden, um auf ihre Eigenschaften zuzugreifen und eine Methode, um ein "Ingredient" anhand seines Namens zu finden.

Die Verwendung von Entities wie der "Recipe"-Klasse kann dazu beitragen, die Komplexität von Anwendungen zu reduzieren, indem sie wichtige Konzepte in der Domäne als eigenständige Objekte behandeln. Entities können auch dazu beitragen, die Wartbarkeit von Anwendungen zu verbessern, indem sie eine klare Struktur für die Organisation von Code bereitstellen.

## Value Objects

Bei DDD sind Value Objects Objekte, die keine Identität haben, sondern nur durch ihre Eigenschaften definiert werden. Sie repräsentieren Konzepte in der Domäne, die wichtig genug sind, um als eigenständige Objekte behandelt zu werden, aber keine Identität benötigen. Value Objects haben normalerweise eine Lebensdauer, die von der Lebensdauer der Anwendung abhängt.

In meinem Code ist die Klasse "Ingredient" ein Value Object. Diese Klasse repräsentiert einen Bestandteil eines Rezepts und wird nur durch ihren Namen, ihre Menge und ihre Einheit definiert. Ein "Ingredient" hat keine Identität, da es durch diese Eigenschaften eindeutig identifiziert wird.

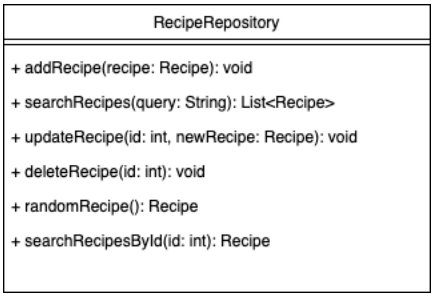


Die "Ingredient"-Klasse hat keine Identität, sondern wird nur durch ihre Eigenschaften definiert. Die Klasse hat auch Methoden, um auf ihre Eigenschaften zuzugreifen.

Die Verwendung von Value Objects wie der "Ingredient"-Klasse kann dazu beitragen, die Komplexität von Anwendungen zu reduzieren, indem sie wichtige Konzepte in der Domäne als eigenständige Objekte behandeln. Value Objects können auch dazu beitragen, die Wartbarkeit von Anwendungen zu verbessern, indem sie eine klare Struktur für die Organisation von Code bereitstellen.

## Repositories

Bei DDD sind Repositories Objekte, die eine Schnittstelle zwischen der Domäne und der Datenquelle bilden. Sie ermöglichen es, Objekte aus der Domäne in der Datenquelle zu speichern und abzurufen, ohne dass die Domäne direkt auf die Datenquelle zugreifen muss. Repositories sind in der Regel für eine bestimmte Entität oder Aggregat-Typ verantwortlich.
Diese Klasse "RecipeRepository" ist ein solches Repositorie, das für das Speichern und Abrufen von "Recipe"-Objekten verantwortlich und bildet eine Schnittstelle zwischen der Domäne und der Datenquelle.




Die "RecipeRepository"-Schnittstelle enthält Methoden, um ein "Recipe" hinzuzufügen, "Recipe"-Objekte anhand eines Suchbegriffs abzurufen, ein "Recipe" zu aktualisieren, ein "Recipe" zu löschen, ein zufälliges "Recipe" abzurufen und ein "Recipe" anhand seiner ID abzurufen.

Die Verwendung von Repositories wie der "RecipeRepository"-Schnittstelle kann dazu beitragen, die Komplexität von Anwendungen zu reduzieren, indem sie eine klare Trennung zwischen der Domäne und der Datenquelle bereitstellen. Repositories können auch dazu beitragen, die Wartbarkeit von Anwendungen zu verbessern, indem sie eine klare Struktur für die Organisation von Code bereitstellen.

## Aggregates

Bei DDD sind Aggregates Gruppen von zusammengehörigen Objekten, die als eine Einheit behandelt werden. Sie bestehen aus einer Wurzel-Entität (Root Entity), die die anderen Objekte im Aggregate verwaltet. Aggregates stellen eine konsistente Grenze um eine Gruppe von Objekten dar, die zusammengehören und gemeinsam eine konsistente In-Memory-Transaktion bilden.
Ein Aggregate ist sinnvoll, wenn es eine Gruppe von zusammengehörigen Objekten gibt, die als eine Einheit behandelt werden müssen und eine Wurzel-Entität haben. Die Wurzel-Entität ist für die Verwaltung der anderen Objekte im Aggregate verantwortlich und stellt sicher, dass sie konsistent bleiben.

In meinem Code gibt es jedoch keine Gruppe von Objekten, die als eine Einheit behandelt werden müssen, und daher ist kein Aggregate erforderlich.
Am Beispiel "ShoppingList" : Die "ShoppingList" ist eine Entität, die eine Liste von "ShoppingListItem"-Objekten enthält. Jedes "ShoppingListItem" ist eine eigenständige Entität, die keine Beziehung zu anderen "ShoppingListItem"-Objekten hat. Stattdessen wird die "ShoppingList" als eigenständige Entität behandelt und über die Repository-Schnittstelle verwaltet. Die "ShoppingListItem"-Objekte werden als eigenständige Entitäten behandelt und haben keine Beziehung zu anderen "ShoppingListItem"-Objekten.

## Kapitel 7: Refactoring

### Code Smells

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

### 2 Refactorings

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

## Kapitel 8: Entwurfsmuster

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

### Entwurfsmuster: [Name]

### Entwurfsmuster: [Name]