

TastyPages



Karin Dirksmüller	kd053
Lara Blersch	lb210
Jan Hübner	jh296
Lukas Karsch	lk224

Hochschule der Medien Stuttgart
Software-Entwicklung 2

Kurzbeschreibung des Projekts

TastyPages ist eine „Rezept-sammle-App“, ähnlich einem Rezeptbuch, wie man es in der Küche hat.

In der App kann man **Rezepte** als Einträge hinterlegen wie man sie auf einer Rezepte Webseite wie Chefkoch.de finden würde. Jedes Rezept enthält einen Namen, ein Bild, Zutaten mit Mengenangaben, Kochanweisungen und Kategorien, über die man Rezepte später schneller finden kann. Rezepte können vom Hauptmenü aus erstellt, abgerufen und verändert werden.

Die zweite Primärfunktion der App ist es, erstellte Rezepte schnell wiederzufinden. Über die **Tags** oder eine Volltextsuche lassen sich Rezepte kategorisieren und gezielt herausuchen.

Die **Zutaten** haben dabei noch eine wichtige Funktion: Sie werden aus einer **Datenbank** ausgewählt, in der die entsprechenden Nährwerte hinterlegt sind. Mit diesen Angaben kann man dann seinen persönlichen **Wochenplan** entwerfen, in dem Rezepte für die Tage der Woche ausgewählt werden können. Die **Nährwerte** aller Rezepte können dann gemeinsam ausgegeben werden und die Rezepte an Ernährungsvorgaben angepasst ausgewählt werden.

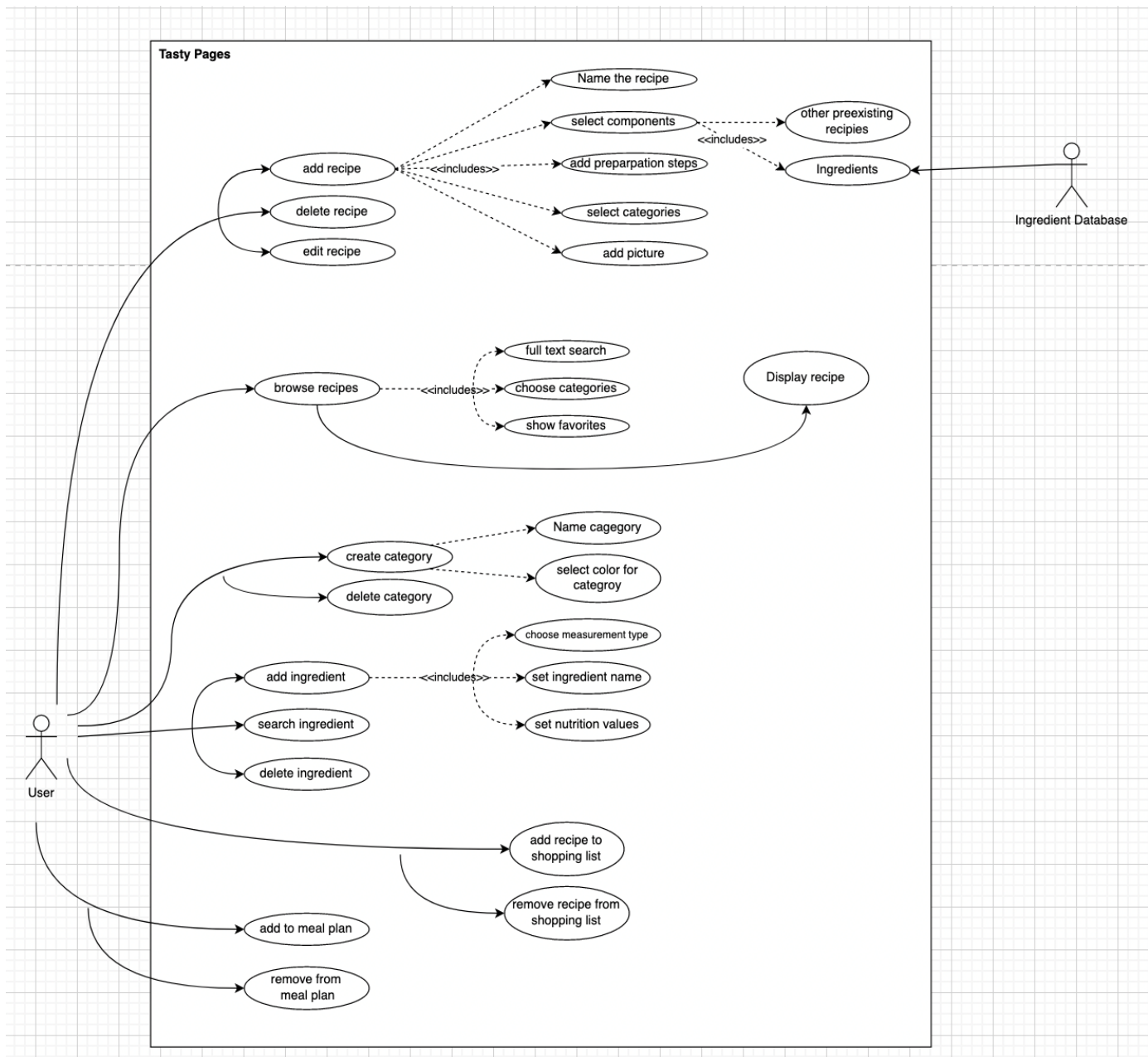
Zudem kann man eine **Einkaufsliste** erstellen, zu der einzelne Zutaten sowie alle Zutaten eines ausgewählten Rezepts hinzugefügt werden können.

Die Main Methode unserer Applikation befindet sich in der Klasse **GUIDriver**.

UML

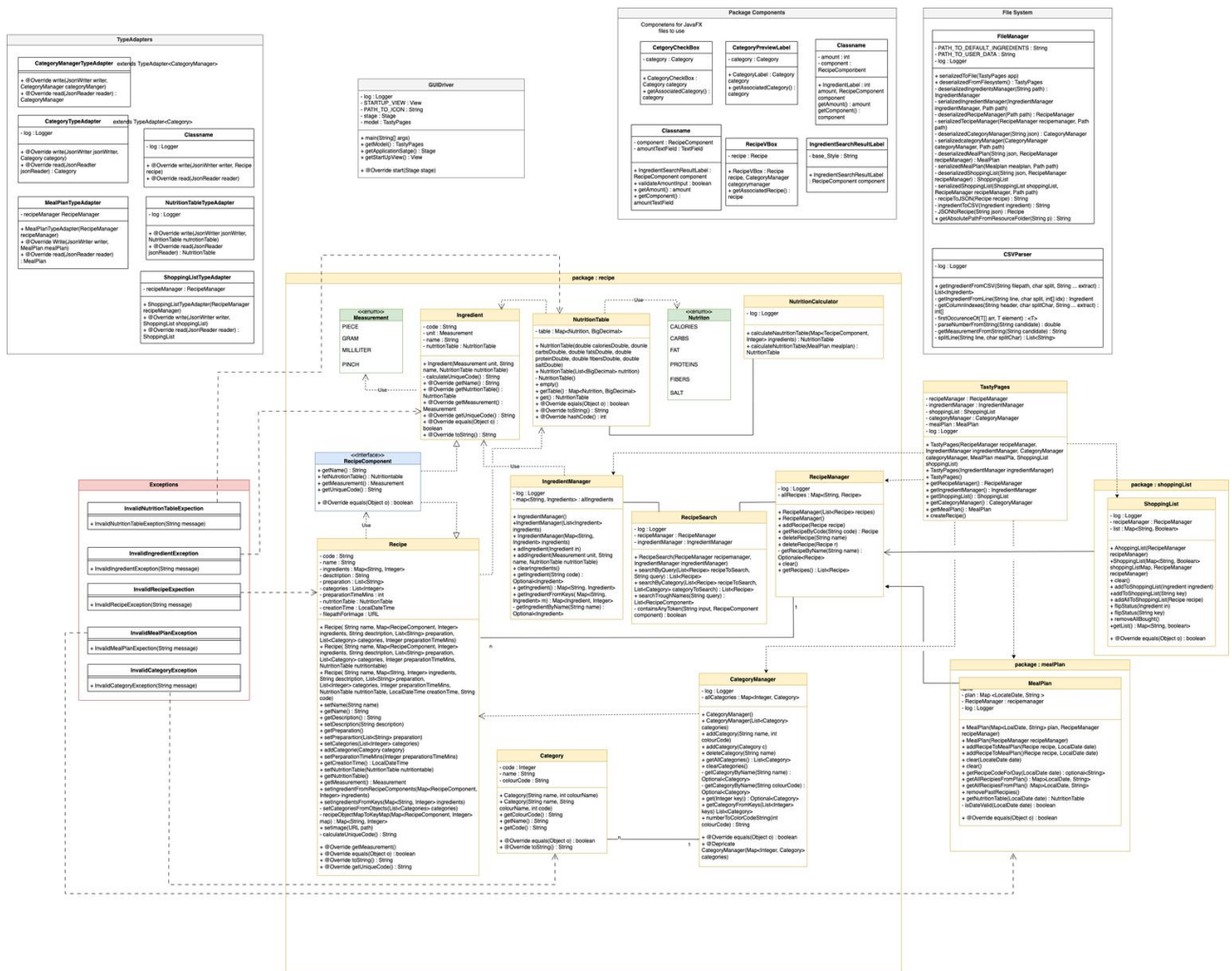
Use case diagram

Da beim Kochen eine Vielzahl an Seiten und Vorgängen Ziel des Benutzers sein können, gibt es mehrere user flows in unserer Applikation. Schon das alleinige Starten der App und das anschließende Auswählen eines Rezepts auf der Startseite könnte das Ziel des Users sein. Genauso wäre denkbar, dass der User zuerst Zutaten und Kategorien anlegt, um anschließend ein Rezept zu erstellen und dieses über die Suchfunktion anzuzeigen. Des Weiteren könnte der Benutzer über den Meal Plan die Rezepte der aktuellen Woche anschauen und eines von ihnen auswählen oder die Shopping List benutzen.



Class Diagram

Das Diagramm stellt nur die Klassen Interaktionen im Base Code (Geschäftslogik) dar. Zusätzlich greifen auch die **GUI Controller** auf fast alle Klassen zu, diese wurden ausgelassen um nicht zu viele Pfeile zu haben. Da wir nach dem MVC-Prinzip arbeiten (Model View Controller), modifizieren die Views lediglich das Model, indem sie Methoden unserer Geschäftslogik aufrufen. Dank dieser Trennung haben wir die Controller nicht auf dem Diagramm abgebildet.

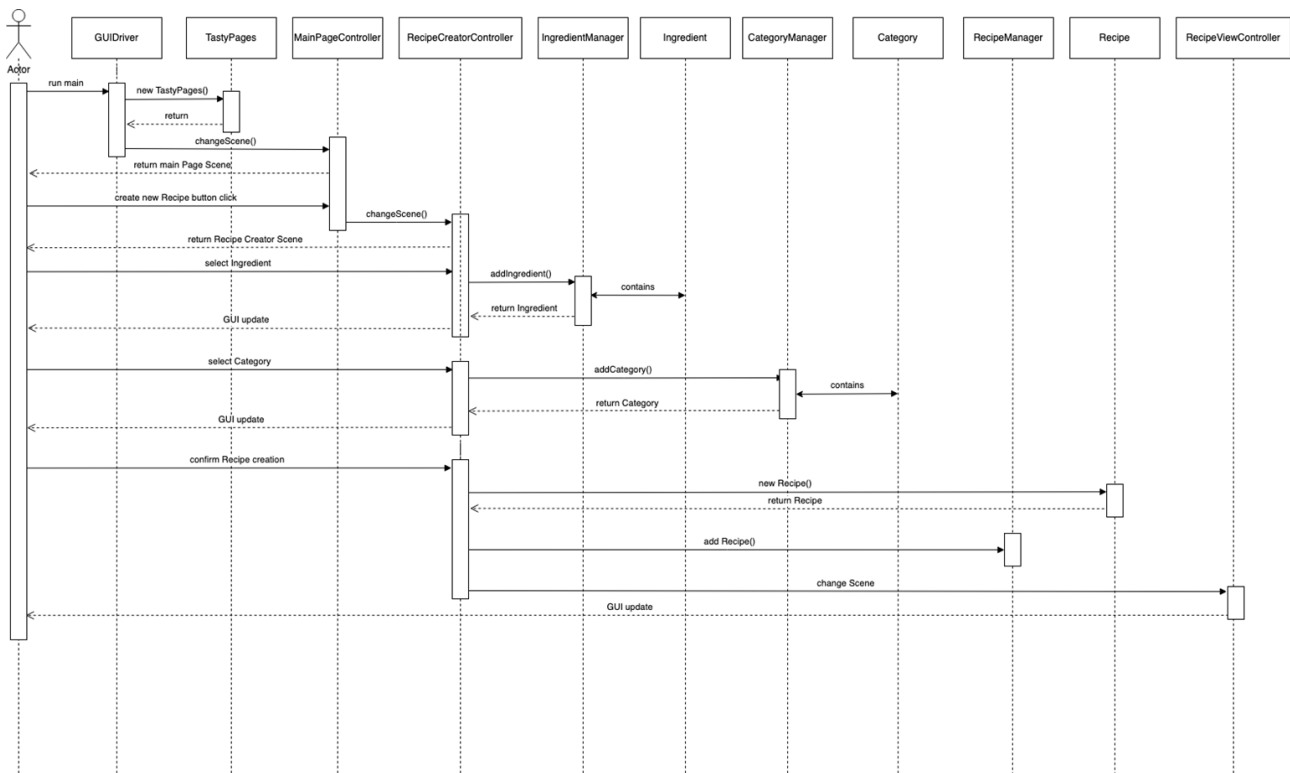


Interaction Sequence Diagram

Das abgebildete Szenario ist „Ein neues Rezept erstellen“.

Die Vorgänge „Ingredient hinzufügen“ und „Category hinzufügen“ können beliebig oft wiederholt werden.

Zusätzlich zu den **Ingredients** und **Categories** werden auch eine ganze Reihe Strings gespeichert, sobald das Rezept erstellt wird. Diese enthalten Angaben wie den Namen des Rezepts, die Kochbeschreibungen und mehr. Das Rezept wird anschließend in einer JSON-Datei gespeichert.



Stellungnahmen

Architektur

Interfaces:

RecipeComponent ist ein Interface, das von **Ingredient** und **Recipe** implementiert wird. **Recipe** ist auch ein **RecipeComponent**, da man auch bereits existierende Rezepte in ein neues Rezept einbauen kann.

Zum Beispiel kann man einen Kuchenteig als Rezept erstellen und diesen Teig dann in mehreren Rezepten als „Zutat“ einsetzen, aber jedes Mal mit anderen Kuchenfüllungen.

Enums:

Das Enum **Measurement** gibt die Maßeinheiten für die Zutaten in allen Rezepten an. Hier sind nicht nur typische Einheiten wie Gramm und Milliliter hinterlegt, sondern auch kochspezifische „Abmessungen“ wie Stück oder eine Prise.

Das Enum **Nutrition** enthält alle Nährwerte, die wir im **NutritionCalculator** verwenden.

Kein Singleton:

Wir hatten zu Beginn mehrere Singletons (**ShoppingList**, **MealPlan**, **Categories**, **RecipeManager**), haben diese allerdings nach und nach zu regulären Klassen umgebaut. Singletons sind auf den ersten Blick überaus praktisch, erlauben aber globalen Zugriff auf Elemente, was Sicherheitsrisiken birgt und falschen Zugriff ermöglicht. Wir haben uns stattdessen für Objektzugriff über Dependency Injection entschieden.

Factory:

In der Klasse **View** beinhaltet die Methode **getScene** eine **Factory**, um den korrekten **Constructor** für die **GUI Controller** aufzurufen.

Vererbung:

Im Package **Components** befinden sich eigens erstellte GUI-Elemente die alle von Basis-JavaFX-Klassen erben.

Ordnerstruktur / Packages

components:

Enthält Bestandteile des GUI. Jedes Element hat eine eigene Klasse, die auf einer oder mehreren Pages im GUI verwendet werden.

controller:

Enthält alle **Controller** für die separaten GUI Pages sowie das **View** Enum.

exceptions:

Enthält eigene **Exceptions**, die im Code aufgerufen werden können.

filesystem:

CSVParser enthält die Daten zum Einbinden der Datenbank. Die Datenbank ist als CSV hinterlegt und enthält die Namen und Nährwerte der Zutaten.

FileManager speichert **Recipes**, **Ingredients**, **MealPlan** und die **ShoppingList** in JSON-Dateien.

helpers:

Helferklassen für kleine Berechnungen und testet, ob Benutzereingaben über das GUI valide sind.

mealPlan:

Enthält die **gleichnamige** Klasse. Man kann dort **Recipes** in eine einwöchige Kalenderansicht (**MealPlan**) einfügen. Die Nährwerte der **Ingredients** aus den eingefügten **Recipes** werden zusammengerechnet und als allgemeine Nährwerte für die Woche ausgegeben.

Recipes:

Enthält Klassen, die für das Anlegen und Verwalten der **Recipes**, **Ingredients** und **Categories** verwendet werden.

Die Klassen **Recipe**, **Category** und **Ingredient** enthalten alle Methoden, die für das jeweilige Objekt wichtig sind, wie beispielsweise Getter und Setter, equals()-Methoden sowie eine Methode für das Abrufen des objektspezifischen Codes, mit dem das Objekt am Ende in den JSON-Dateien referenziert wird.

Die **-Manager** sind zum Verwalten von bereits erstellten Objekten verantwortlich. Objekte der entsprechenden Klasse werden dort entweder in Maps oder Lists gespeichert. Die Manager verfügen über Methoden zum Hinzufügen, Entfernen und Finden von Objekten.

shoppingList:

Enthält die **gleichnamige** Klasse. Sie generiert eine Liste mit Zutaten von Rezepten, die in die **ShoppingList** eingefügt wurden. Nicht mit dem **MealPlan** verbunden. Erlaubt auch das separate Einfügen von eigenen **Ingredients**.

typeAdapters:

Klassen mit deren Hilfe **Gson** (eine JSON-Dependency von Google) JSON-Dateien für die Speicherung im Dateisystem generiert. Der Inhalte der Rezepte, der Inhalt der **ShoppingList** und der Inhalt des **MealPlan** werden separat in einzelnen Dateien gespeichert.

Clean Code

Verzicht auf Public Instanzvariablen in allen Klassen.

Stark eingeschränkte Nutzung von **Static** Variablen und Methoden

Code wurde wenn möglich so geschrieben, dass er an mehreren Stellen **wiederverwendet** werden kann, wodurch doppelte Code Segmente verhindert werden.

Wir haben nur **getter** mit schreibbaren Referenzen wo es nötig ist, ansonsten werden Objekte geklont.

Tests

Tests stehen in src/test.

Mit Hilfe von **JUnit** haben wir ca. 70 Unit Tests geschrieben, welche die gesamte Funktionalität des Datenmodells auf korrekten Ablauf überprüfen – auch bei Edgecases (Negative Tests).

GUI (JavaFX)

Ein großer Teil des GUIs wurde mit **SceneBuilder** erstellt, die Dateien im Package „components“ wurden direkt mit Java Code erstellt und agieren als dynamische GUI-Elemente, welche nach Bedarf in die FXML-Grundgerüste eingefügt werden.

Teilweise werden GUI-Elemente während der Benutzung der Applikation mithilfe der Controller dynamisch erzeugt und geladen. Dies ist insbesondere dann der Fall, wenn Ereignisse von Interaktionen des Users abhängig sind.

Logging/Exceptions

Logs:

Der **Logger** heißt „log“ in allen Klassen. Sowohl für Debug-Zwecke, als auch um dem Nutzer Informationen über den Status des Programms zu geben, befinden sich in allen relevanten Klassen log-Statements.

Wir haben von allen Log-Leveln Gebrauch gemacht:

- log.fatal
- log.error
- log.warn
- log.info
- log.debug
- log.trace

Exceptions:

Die **InvalidCategoryException** tritt auf, wenn man versucht eine **Category** zu erstellen, die es bereits gibt oder wenn die eingegeben Daten für das Erstellen der **Category** unvollständig oder nicht erlaubt sind.

Die **InvalidIngredienteExcepotion** tritt auf, wenn beim Erstellen einer **Ingredient** der Name bereits vergeben wurde.

Die **InvalidMealPlanException** wird geworfen, wenn beim Erstellen eines **MealPlans** NULL Werte übergeben werden oder ein Datum ungültig ist.

Die **InvalidNutritionTableException** wird geworfen, wenn beim Erstellen eines **NutritionTables** ungültige Werte übergeben werden oder kein Wert übergeben wird.

Die **InvalidRecipeException** wird geworfen, wenn beim Erstellen eines **Recipes** ungültige Werte übergeben werden, keine Werte übergeben werden oder wenn beim Zugriff auf den **RecipeManager** ein nicht vergebener RezeptCode verwendet wird.

Die **MissingContentException** ist unsere einzige Checked Exception. Sie wird geworfen, wenn das GUI Daten abrufen und für einen hinterlegten Rezept-, Category- oder Ingredient-Code kein Objekt existiert (aufgrund korrupter Daten oder dem Löschen von Dateien über den Explorer vonseiten des Benutzers).

UML

Wir haben schon recht früh ein **Klassen-Diagramm** und ein **Use-Case-Diagramm** erstellt, die wir im Laufe des Projekts häufig updaten mussten. Unsere Planung in diesen Bereichen

wurde immer wieder umgeworfen und aktualisiert, bis wir zu unserem jetzigen Aufbau gekommen sind.

Das **Interaction-Sequence** Diagramm haben wir erst gegen Ende des Semesters erstellt, da wir angenommen haben, dass unser Aufbau durch das Einfügen der GUI-Elemente noch einmal stark verändert werden würde.

Threads

Unsere Applikation beinhaltet einen Daemon Thread: den MusicThread. Dieser wird mit dem Start der Applikation aufgerufen und spielt dann in Dauerschleife Musik ab, welche durch synchronisierten Objektzugriff auf eine Variable innerhalb des Threads vom GUI aus pausiert werden kann.

Streams und Lambda-Funktionen

In der Klasse **RecipeSearch** verwenden wir einen parallelen Stream. Damit werden **Ingredients** und **Recipes** gleichzeitig nach einem eingegebenen Stichwort durchsucht. Das Ergebnis des Streams wird anschließend Thread-safe gesammelt, um Korruption der Daten zu verhindern:

```
.collect(Collectors.collectingAndThen(
    Collectors.toList(),
    Collections::unmodifiableList)
);
```

Allgemein haben wir in unserem Code viele Streams und Lambda-Funktionen verwendet, da die funktionale Art der Programmierung in Java sowohl in gut lesbarem als auch elegantem und präzisem Code resultiert.

Lambda-Funktionen befinden sich neben den Streams auch in vielen der Controller Klassen, wo sie als EventHandler fungieren. Einige Beispiele sind **CategoryManager**, **MainPageController**, **MealPlanController**, **RecipeCreatorController**, **RecipeEditorController** und mehr.

Factories

Unter (source > main > controllers) in der Klasse **View** haben wir eine Factory. Diese erstellt den korrekten Controller für das GUI, indem sie an den Constructor der **Controller** Klasse die korrekten Parameter übergibt.

Bewertungsbogen

Unser Bewertungsbogen sieht folgendermaßen aus. Er befindet sich ebenfalls im Git-Repository.

Vorname	Nachname	Kürzel	MatrikelProjekt	Arc.	Clean Code	Doku	Tests	GUI	Logging/Except.	UML	Threads	Streams	Profiling	Summe - Projekt	Kommentar	Projekt-Note
Karin	Dirksmüller	kd053	45089 TastyPages	3	3	3	3	3	3	3	2	3	3	29,00	siehe Dokumentation	1,00
Lara	Blersch	lb210	44450 TastyPages	3	3	3	3	3	3	3	2	3	3	29,00	siehe Dokumentation	1,00
Lukas	Karsch	lk224	45259 TastyPages	3	3	3	3	3	3	3	2	3	3	29,00	siehe Dokumentation	1,00
Jan	Hübner	jh296	45204 TastyPages	3	3	3	3	3	3	3	2	3	3	29,00	siehe Dokumentation	1,00

Nachdenkzettel

Die Nachdenkzettel sind im Ordner **Nachdenkzettel** im Git-Repository hinterlegt.