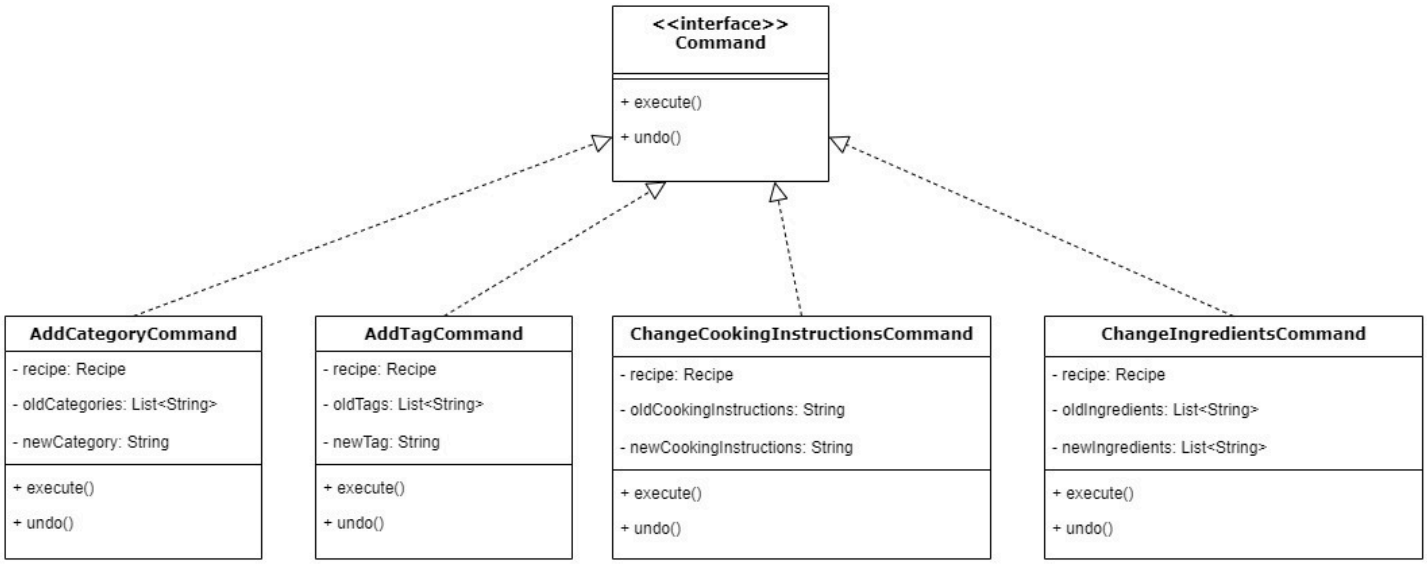
Recipe Management System

Recipe Management System, a console-based Java project, offering functionalities for recipe creation, modification, search, and rating. Users can create recipes by specifying ingredients, cooking instructions, and serving size, and categorize them with tags and categories. The system enables users to search for recipes based on keywords, tags, or categories, and rate recipes on a scale of 1-5 stars. Additionally, users can modify existing recipes by altering ingredients, instructions, or categorization, with an option to undo the last modification. The project includes 3 design patterns. Let's take a look at which one we use and how we use it.

COMMAND PATTERN - The Command pattern encapsulates a request as an object, thereby allowing parameterization of clients with queues, requests, and operations. In our project, the **Command** interface defines the **execute()** and **undo()** methods, which are implemented by concrete command classes such as **AddCategoryCommand**, **AddTagCommand**, **ChangeCookingInstructionsCommand**, and **ChangeIngredientsCommand**. These classes encapsulate specific operations and allow them to be executed and undone independently.

When a user performs an action in the Recipe Management System (such as adding a category, adding a tag, changing cooking instructions, or changing ingredients), a corresponding command object is created and executed. For example, when a user adds a category to a recipe, an **AddCategoryCommand** object is created with the necessary information and executed. The **undo()** method of each command allows reverting the action, providing a way to undo the last modification performed on a recipe.



SINGLETON PATTERN - The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. In our project, the **RecipeManagementSystem** class implements the Singleton pattern, ensuring that only one instance of the Recipe Management System exists throughout the application's lifecycle. *The Singleton instance of **RecipeManagementSystem** is accessed via the **getInstance()** method, which returns the existing instance if it already exists or creates a new one if it doesn't. This ensures that all parts of the application interact with the same instance of the Recipe Management System.*

```
1 usage
public static RecipeManagementSystem getInstance() {
    if (instance == null) {
        instance = new RecipeManagementSystem();
    }
    return instance;
}
```

FACTORY PATTERN - The Factory Method pattern defines an interface for creating objects but allows subclasses to alter the type of objects that will be instantiated. *The Factory Method pattern is utilized in our project through the* ***RecipeFactory*** *interface and its concrete implementation,* ***ConcreteRecipeFactory****. By defining a common interface for creating* ***Recipe*** *objects and delegating the instantiation to concrete subclasses, the Factory Method pattern enables the creation of* ***Recipe*** *objects in a flexible and extensible manner. The* ***ConcreteRecipeFactory*** *provides a method for creating* ***Recipe*** *instances, allowing subclasses to specialize the creation process as needed. This approach promotes code maintainability and scalability, as it allows for the easy addition of new types of recipes without modifying existing client code. Moreover, it adheres to the Open/Closed principle by enabling extension through subclassing while keeping the core* ***RecipeManagementSystem*** *class unchanged. Thus, the Factory Method pattern enhances code organization and facilitates the management of recipe creation logic within the application architecture.*