BIM492 DESIGN PATTERNS

Project Report

“Recipe Management System”

Name: Erinç Minareci **Number:** 34175008424

Name: Buğra Kaan **Number:** 34462456600

Name: Osman Berkecan Yıldırım **Number:** 28342904046

Faculty/Department: Engineering Faculty /Computer Engineering

2024/ESKİŞEHİR

Introduction

Food is one of the most basic human needs. Enjoying delicious meals has become one of the greatest desires of our time. The increase in culinary options and the diversity of dishes stimulate people's desire to try new foods. That's why we've developed a "Recipe Management System" for better cooks. With this system, you can save recipes you find, classify them, and easily retrieve them when needed. This way, your favorite recipes are always at your fingertips. If you ever want to make changes, the conditions are always available for you to modify them. We hope you enjoy our product.

Patterns We Used in Recipe Management System

1-)Singleton Pattern

The Singleton pattern is particularly useful in scenarios where a single point of control is needed over a resource. Using the Singleton pattern for the RecipeBook class ensures that all parts of the application interact with the same instance of the recipe repository, thereby maintaining consistency and preventing conflicts, such as duplicate recipes or concurrent modifications.

This implementation enhances the manageability and scalability of the system by ensuring that all operations related to recipe management are centralized through a single, globally accessible instance. This approach also simplifies the interaction between different parts of the application and the recipe data.

In the implementation of the Singleton pattern within our Java-based Recipe Management System, the RecipeBook class is designed to ensure that only one instance of this class exists throughout the application. This is achieved by making the constructor private and providing a static method getInstance(), which returns the existing instance of the RecipeBook if it exists or creates a new one if it doesn’t. This setup prevents direct instantiation from outside classes and guarantees that all interactions within the application are managed through this single instance. The RecipeBook class contains a recipeList attribute, which stores a collection of Recipe objects, and provides an addRecipe(Recipe recipe) method, allowing recipes to be added to this list, thus centralizing recipe management in one globally accessible location within the application. This pattern not only ensures consistency and control over the data but also supports easier maintenance and scalability of the system.

2-) Abstract Factory Pattern

In the Recipe Management System, the Abstract Factory Pattern is effectively applied to enhance the system's design and functionality. At its core, this pattern provides a structured approach for creating families of related or dependent objects without directly specifying their concrete classes. Within the system, the Recipe class acts as the Abstract Product, defining a standardized blueprint for Recipe objects that encompasses common methods and fields shared across all recipe types. This abstraction allows for the encapsulation of essential recipe attributes and behaviors in a unified manner. The Concrete Products in the system, namely AsianStyleRecipe, KoreanStyleRecipe, SpecialStyleRecipe, and TurkishStyleRecipe, extend the Recipe abstract class, offering specific implementations tailored to each distinct recipe style. This implementation fosters modularity by compartmentalizing the creation logic of different recipe styles, thus facilitating easy addition or modification of new styles without perturbing existing code segments. Furthermore, the Abstract Factory Pattern affords flexibility, as it enables the system to seamlessly accommodate the introduction of new recipe styles or variations within existing ones, all while minimizing the impact on the overall architecture.

This adaptability is crucial for ensuring the system's responsiveness to evolving requirements and user preferences. Additionally, the pattern enhances maintainability by promoting a clear separation of concerns, with each Concrete Factory responsible for creating a specific family of Recipe objects. This delineation of responsibilities simplifies code organization and reduces the potential for errors during development and maintenance tasks. Moreover, the scalability of the system is greatly facilitated by the Abstract Factory Pattern, as it facilitates the effortless expansion of the system to handle new families of related objects or variations within existing ones. Overall, the abstraction provided by the pattern fosters loose coupling between client code and Concrete Products, resulting in improved code readability and maintainability. Through the careful application of the Abstract Factory Pattern, the Recipe Management System achieves a robust, extensible, and maintainable design that is well-equipped to meet the demands of evolving culinary preferences and requirements.

3-) Command Pattern

The Command Pattern encapsulates requests as objects, allowing for parameterization of clients with different requests, queuing or logging of requests, and support for undoable operations. This pattern enhances modularity, extensibility, and uniformity in executing operations within the system.

The Command interface defines a contract for concrete command classes, requiring them to implement a single method, execute(). This method is responsible for executing the command.

Various concrete command classes, including CreateRecipeCommand, InstructionCommand, ListRecipesCommand, ModificationCommand, RateRecipeCommand, RecipeStyleCommand, SearchRecipeCommand, SelectRecipeCommand, ShowRatingsCommand, and TagCommand, implement the Command interface.

Each concrete command class provides an implementation for the execute() method, encapsulating specific actions to be performed when the command is executed.

The invoker is responsible for utilizing the command objects. It interacts with the command objects through the command interface without needing to know about the concrete implementation of each command.

The receiver represents the objects that perform the actual work when the command is executed. In this codebase, receivers include objects such as Recipe, Tag, Category, etc.

In the CreateRecipeCommand class, the execute() method calls the handleCreateRecipe() method, which encapsulates the creation of a new recipe. It involves setting attributes such as name, description, size, categories, tags, ingredients, and instructions of the recipe.

Similarly, in the ModificationCommand class, the execute() method invokes various methods to modify different aspects of a recipe, such as its name, description, service size, ingredients, instructions, categories, and tags.

The Command Pattern facilitates the encapsulation of operations as objects, promoting modularity, flexibility, and uniformity in executing commands within the Recipe Management System. Each operation is encapsulated within its own class, enhancing code maintainability and extensibility. Additionally, the pattern enables operations to be executed uniformly, regardless of the specific actions required, contributing to the overall robustness of the system.

4-) Factory Method Pattern

The Factory Method pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. In the context of creating recipes, the Factory Method pattern enables flexibility and extensibility by delegating the instantiation of recipe objects to concrete creator classes based on user input.

The Factory Method pattern defines an interface for creating objects in a superclass but delegates the responsibility of instantiation to subclasses.

Factory Method pattern is utilized to create recipe objects based on user input. The RecipeCreator abstract class declares the factory method createRecipes, which returns an object of type Recipe. Concrete creator classes (AsianRecipeFactory, KoreanRecipeFactory, SpecialRecipeFactory, and TurkishRecipeFactory) extend RecipeCreator and implement the createRecipes method to instantiate specific types of Recipe objects (Asian, Korean, Special, or Turkish) based on user input.

RecipeCreator serves as the abstract creator class, defining the factory method createRecipes. Subclasses extend RecipeCreator to provide specific implementations for creating different types of recipes.

AsianRecipeFactory, KoreanRecipeFactory, SpecialRecipeFactory, and TurkishRecipeFactory are concrete creator classes that extend RecipeCreator. Each of these classes overrides the createRecipes method to instantiate and add a specific type of recipe object to the RecipeBook.

5-)Strategy Pattern

The Strategy pattern is a behavioral design pattern that enables defining a family of algorithms, encapsulating each one, and making them interchangeable. In the context of computing and updating ratings for recipes, the Strategy pattern offers flexibility and extensibility by encapsulating different rating computation algorithms into separate classes.

The Strategy pattern defines a family of algorithms, encapsulates each one as an object, and makes them interchangeable within the original context object.

Strategy pattern is applied to compute and update ratings for recipes.

The RatingCompute interface declares methods for the strategy, which are updateRatings and computeImpact. Concrete strategy classes, such as AverageRatingCompute and TotalRatingCompute, implement this interface and provide different implementations for the updateRatings and computeImpact methods.

AverageRatingCompute: This concrete strategy class implements the updateRatings method to calculate the average rating of a recipe and the computeImpact method to return the average rating.

TotalRatingCompute: This concrete strategy class implements the updateRatings method to update the total ratings of a recipe and the computeImpact method to return the total ratings.