Creational patterns

Singleton pattern

All repository classes are going to be singleton classes. Wherever they are needed, they should be accessed through the static methods to get instance of needed repository class. This pattern has been implemented on our project.

Prototype pattern

Our application has no use for object cloning, which is why this pattern is not implemented.

Factory method pattern

The class "Direction" implements the factory method pattern. In this way, we differentiate two types of directions, long and short. This makes it easier to later add more direction classes. This pattern has been implemented on our project.

Abstract factory pattern

This pattern can be used by adding more classes for recipe types. Say we have classes Vegetarian, Pescatarian and NonVegetarian. If our application provided the option to differentiate users by the food they eat and we added an option for them to have personalized meal recommendations, we could use this pattern to avoid using "if-else" statements. It would be implemented by adding a new interface "IFactory" which would be mutual for the factory classes. Moreover, we would need additional sub-classes (ex. VegetarianFactory) and they would implement the IFactory interface.

Builder pattern

Builder pattern is not implemented in our project. For an example, if we added an option of auto-filling comments (in the Feedback class) based on the rating of a certain meal, this pattern could be used. The user would be able to decide whether they wanted to write their own comment, or have it auto-generated. We would have to add two new classes that would implement methods on their own way. Next, we would add an interface IBuilder which would build objects based on what the user chooses.

Structure patterns

Adapter pattern

This pattern can be used if we added an option with which users could add pictures of meals they have made. The new interface "IReproduction" would analize and control extensions of the pictures, making them all usable (so we would not have to restrict users on using only .jpg extensions).

Facade pattern

Facade pattern could be used if we added more classes and methods that would define certain ingredients. Lets say we added classes "LiquidIngredient" and "SolidIngredient". This pattern could be implemented by adding a new class "IngredientFacade" that would implement methods addLiquidIngredient and addSolidIngredient. As this would need many unnecessary modifications in our project, we have not implemented it.

Decorator pattern

Say we wanted to allow certain users to edit existing and add optional steps to a recipe. New interface "IStep" with methods "editStep" and "addOptionalStep" would be added to our project. We would also need new classes that would represent these new implementations. Using this pattern, the directions could now be modified subsequently.

Bridge pattern

Lets say we have a class "Guest" and "RegisteredUser" and we want to differentiate the types of recipes they can search. Implementing this pattern, we would add a new interface "ISearch" that would hold the method implementation based on what kind of user is using the application. We would also need to add a new class "Bridge". As we did not want to implement this difference in our program, we have not used this pattern.

Composite pattern

Say we have an abstract class "User" and classes "Guest" and "RegisteredUser" which inherit the class "User". If we wanted to show recommended recipes on the start page for these users, we could use saved recipes for registered users, and show random recipes for guest users. To implement this pattern, we would need to change the class "User" to an interface "IUser" which would implement the method "giveRecommendedRecipes" and the given classes would inherit the interface.

Proxy pattern

Proxy pattern has been implemented on our project and is shown on the new class diagram. Proxy class controls the modification of recipes based on the given access level.

Flyweight pattern

Flyweight pattern is used to split up recipes based on the ingredients the user has which will be displayed on the page: whether he has some missing ingredients or if he has all of them. This pattern has been implemented on our project.