

CECS 575 - Group 7

Food Ordering System

Assignment 2

Contents

Logger class - Singleton Pattern	2
1.1: Class Diagram:	2
1.2: Collaboration Diagram:	3
1.3: Description:	4
Select a Meal Category / Plan - Factory Pattern	5
2.1.1: Sequence Diagram:	5
2.1.2: Class Diagram:	6
2.1.3: Description:	7
Customize FoodItem - Builder Pattern	8
2.2.1: Class Diagram:	8
2.2.2: Sequence Diagram:	9
2.2.3: Description:	10
Simulating Factory Pattern, Builder Pattern and Singleton in codebase.	11
3.1: Terminal Snapshot:	11
3.2: Main.java - Driver Code:	12
3.3: Logger File:	14

1. Find an application of Singleton design pattern in your application and draw its corresponding class diagram and a collaboration diagram and Implement it in Java. Explain in a text document why singleton should be applied in the scenario and how it improves your design

Logger class - Singleton Pattern

1.1: Class Diagram:

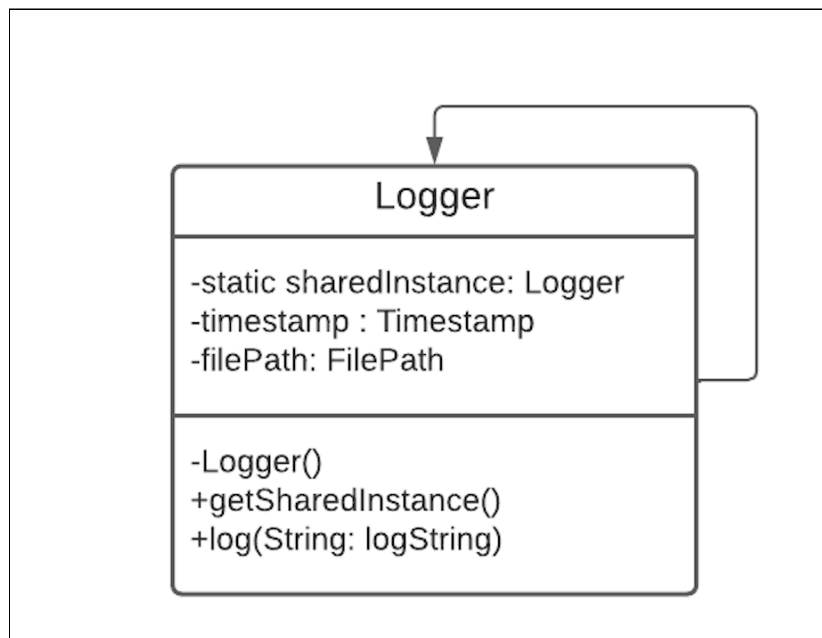


Fig 1.1: Class Diagram for singleton pattern implementation

1.2: Collaboration Diagram:

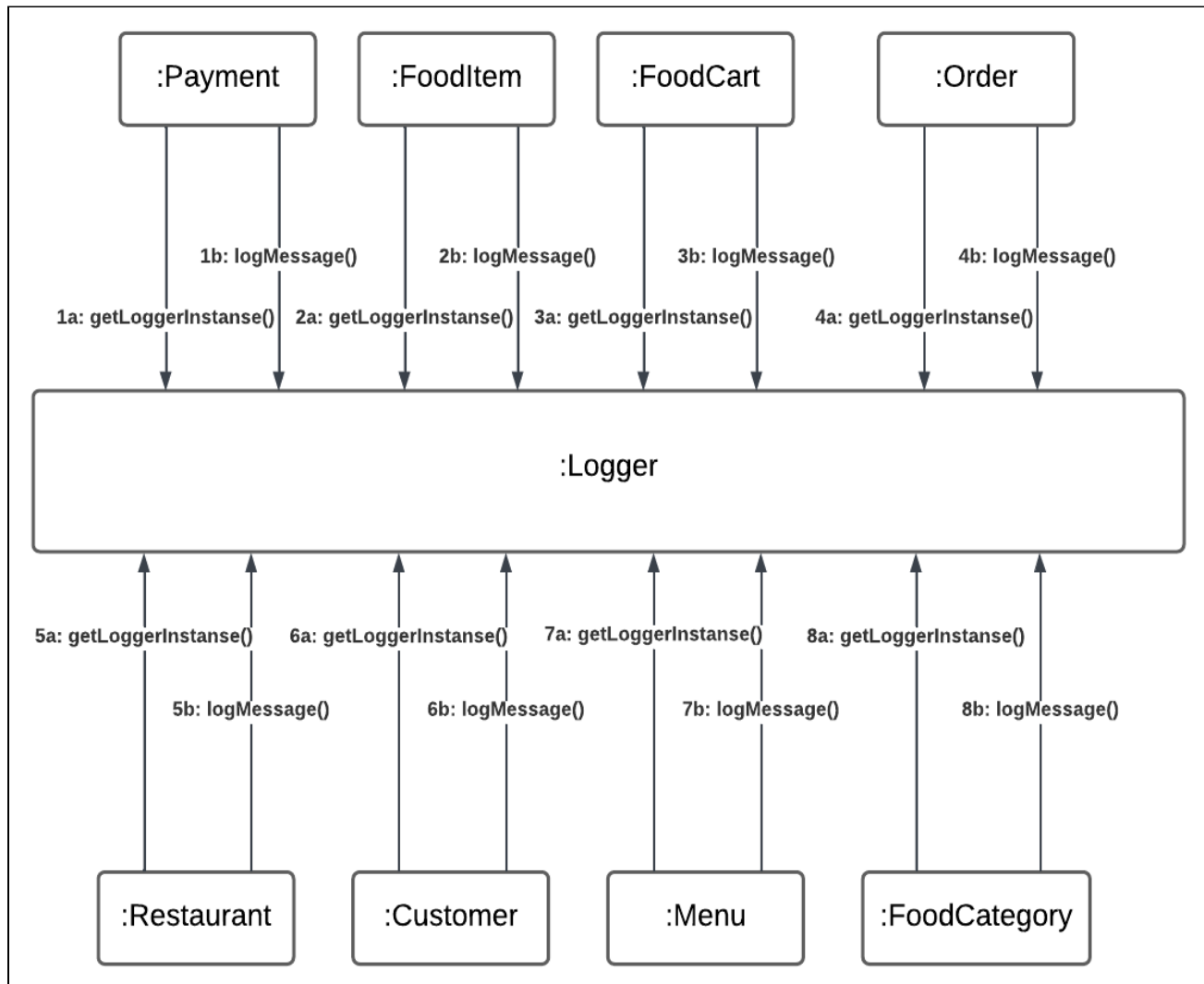


Fig 1.2: Collaboration diagram for singleton pattern implementation

1.3: Description:

Loggers are implemented in a project to log all the important activities / processes carried out during the execution. When we face an exception or if a certain functionality breaks, loggers placed throughout the source code help us trace the bug and identify the actual function or command that caused the exception. It helps developers debug and find the root cause of the issue.

The Logger class that implements the log function, should ideally be a **Singleton**. This makes sure that there are **no parallel overwrites** made to the same log file and there are no race conditions while accessing the same resource. Since the same instance of the Log class is used throughout the project, we can easily update the class configuration and make changes to how the logs are captured in the entire project.

2. Find two compelling scenarios where you can apply creational design patterns (other than singleton) to your system. Draw the corresponding class diagram and sequence diagram for each case and implement it in Java. Explain in a text document how these patterns improve the design of your system

Select a Meal Category / Plan - Factory Pattern

2.1.1: Sequence Diagram:

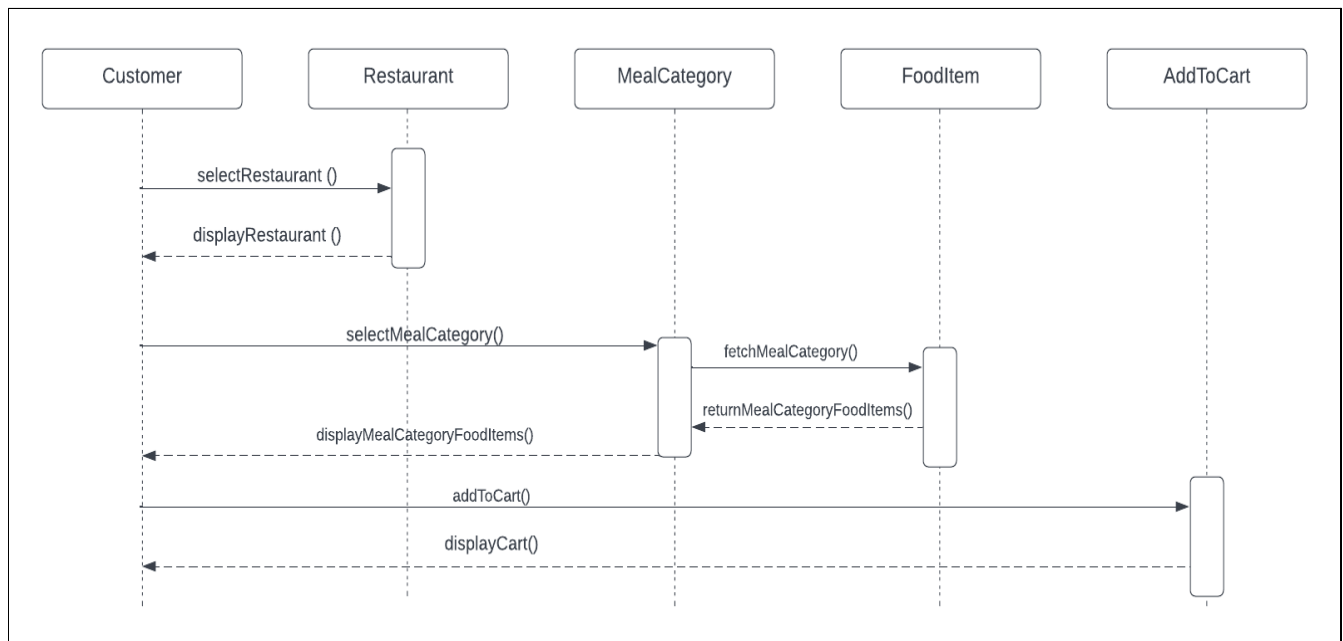


Fig 2.1: Sequence diagram for factory pattern implementation

2.1.2: Class Diagram:

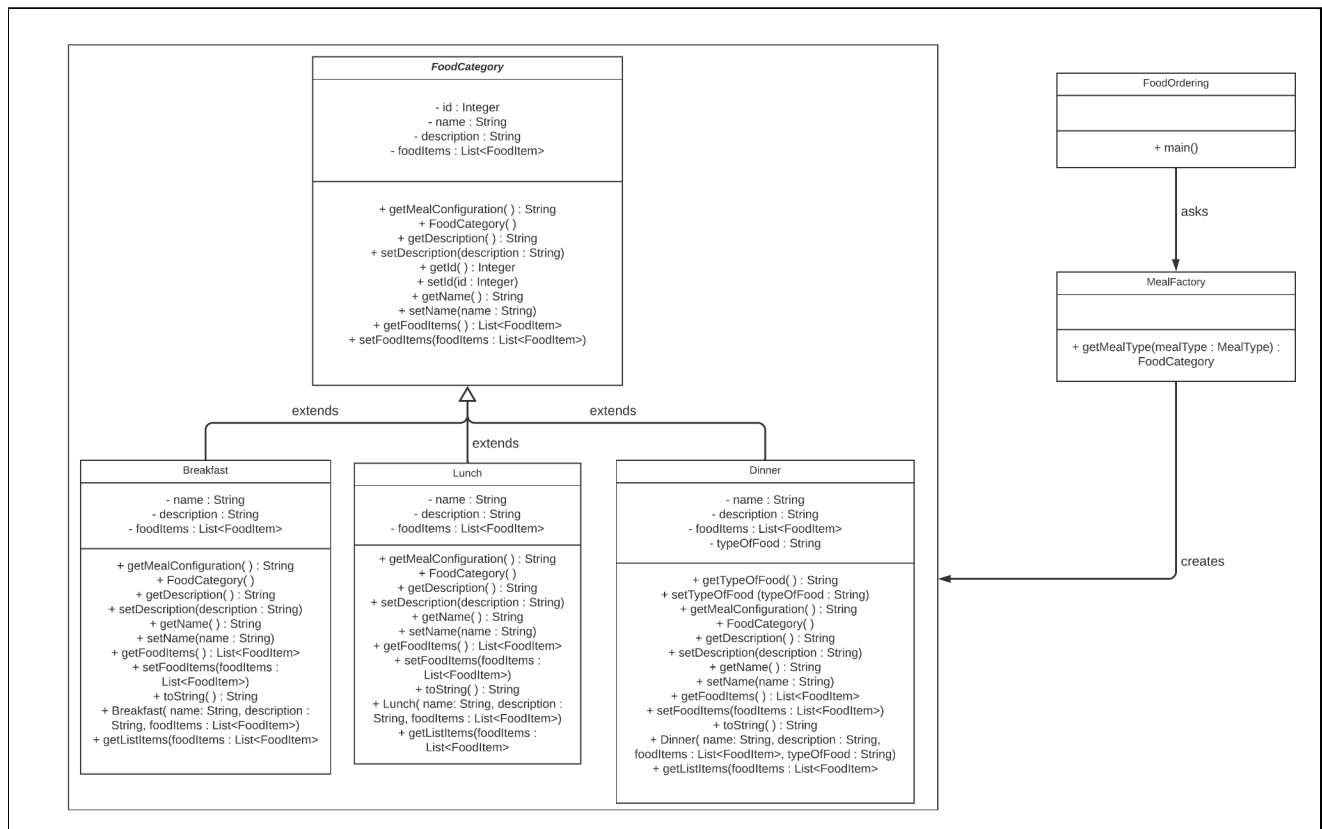


Fig 2.2: Class diagram for factory pattern implementation

2.1.3: Description:

One of the creation patterns used in design patterns is the factory pattern. It makes use of other classes that are of the same kind. Assume we have a meal category that includes breakfast, lunch, and dinner. If the user requires a breakfast meal, he/she must prepare it independently and provide all necessary information. Users must be aware of the information needed to prepare each individual meal. There can be an arbitrary number of users, and if any details in the specific meal class change, all users must update their specific meal information.

For example, if the breakfast information needs to be updated, all users must explicitly state the updated information / criteria when ordering the meal. To get around this, use the meal factory class, which will generate the desired meal object. We built enums for the meal types for better programming practice. We've developed a static type `getMealType()` method that takes the meal type and returns the appropriate meal object. It will construct a breakfast object and return it if it is breakfast. The same goes for lunch and dinner. We call the `MealFactory`'s method `getMealType()` from the caller method, passing the type of meal that is required. If any of the details of one meal change, only that meal's class needs to be updated. The factory pattern's main goal is to free client code of the burden of object generation.

Customize FoodItem - Builder Pattern

2.2.1: Class Diagram:

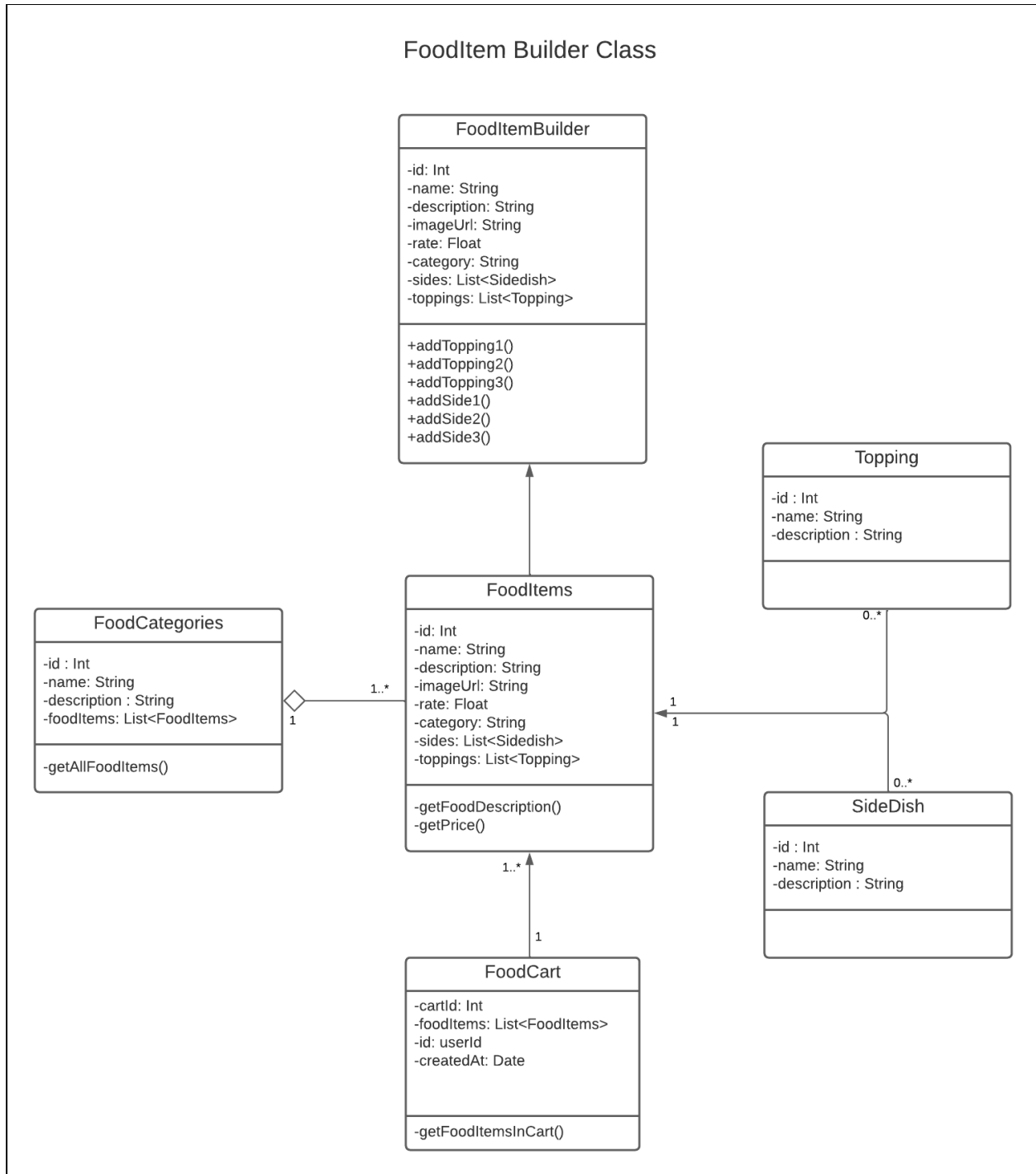


Fig 2.3: Class diagram for builder pattern implementation

2.2.2: Sequence Diagram:

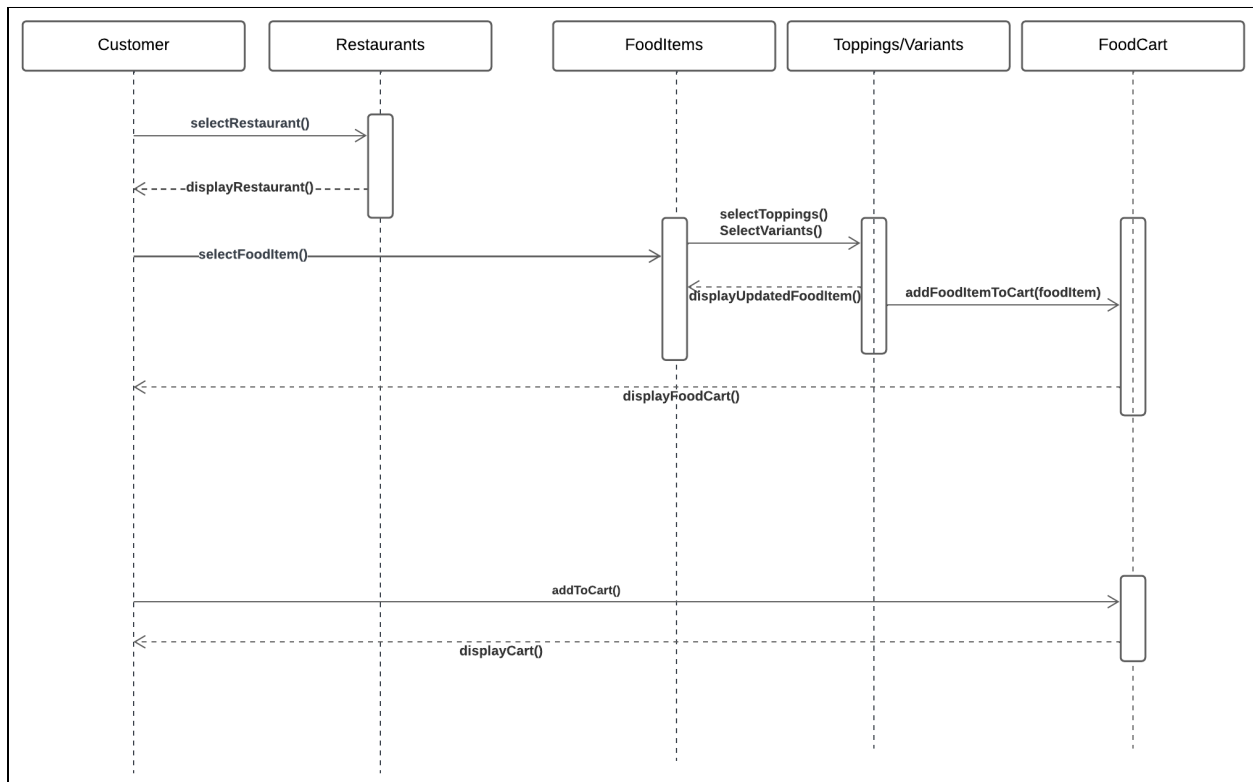


Fig 2.4: Sequence diagram for builder pattern implementation

2.2.3: Description:

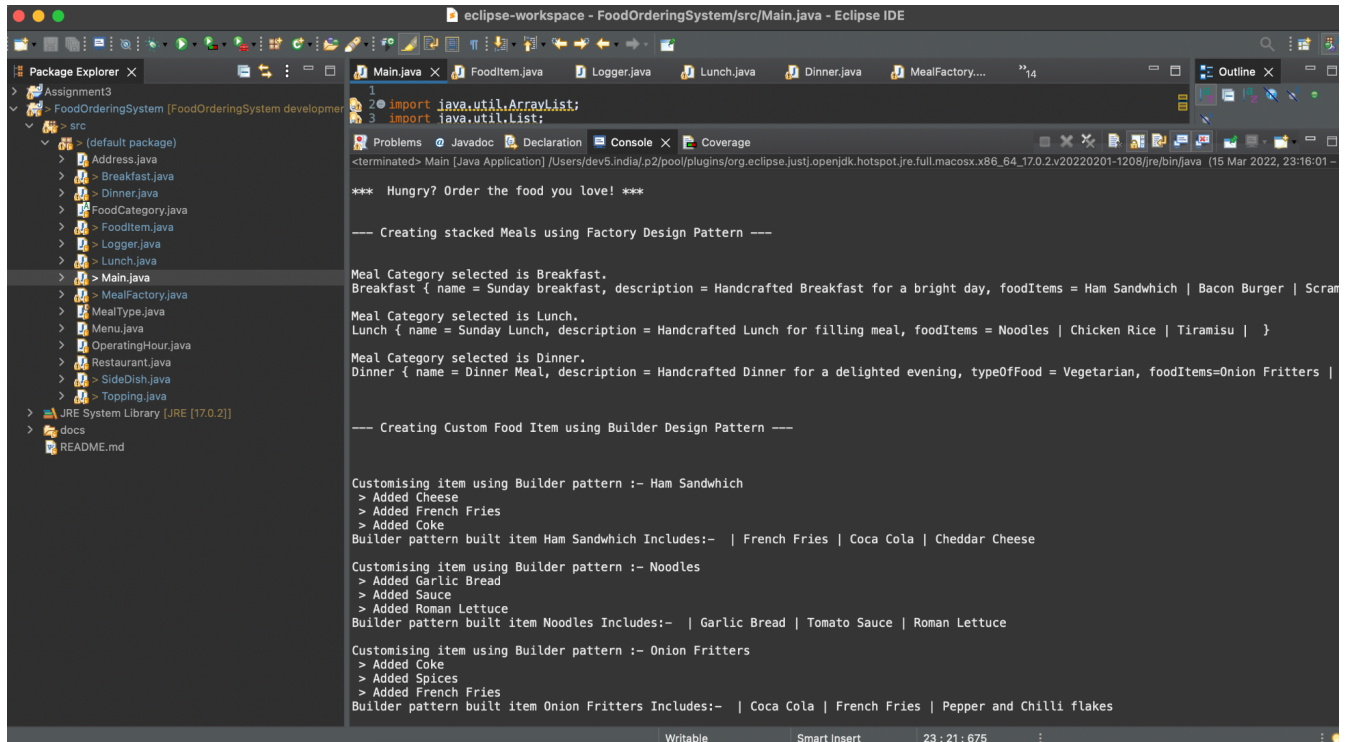
Builder pattern is used to construct complex objects from its representation. When there are multiple attributes that are to be set in the class, but do not need to be set , every attribute builder pattern helps create objects with necessary attributes without the need of creating separate constructors.

In the Food Ordering System, the Meal builder class uses a Builder pattern to construct a Food Item object in which we can customize the food item with various toppings, sides etc.

Every food item can be customized with various toppings and sides. Since users can select any number of toppings or sides, having custom constructors for all these choices seems like a bad design. We use a Food Item Builder class which adds various toppings and sides as incremental function calls to the static builder class defined inside the FoodItem class. Once the items are finalized, we call the build() function to return the Product class (Food Item) from the Concrete Builder (FoodItemBuilder) class.

Simulating Factory Pattern, Builder Pattern and Singleton in codebase.

3.1: Terminal Snapshot:



The screenshot shows the Eclipse IDE interface. The Package Explorer on the left displays the project structure for 'FoodOrderingSystem'. The main editor shows the 'Main.java' file with the following code:

```
1 import java.util.ArrayList;
2 import java.util.List;
3 import java.util.List;
```

The Console window at the bottom shows the output of the application:

```
<terminated> Main [Java Application] /Users/dev5.india/p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_17.0.2.v20220201-1208/jre/bin/java (15 Mar 2022, 23:16:01 -

*** Hungry? Order the food you love! ***

--- Creating stacked Meals using Factory Design Pattern ---

Meal Category selected is Breakfast.
Breakfast { name = Sunday breakfast, description = Handcrafted Breakfast for a bright day, foodItems = Ham Sandwich | Bacon Burger | Scrambled Eggs | }

Meal Category selected is Lunch.
Lunch { name = Sunday Lunch, description = Handcrafted Lunch for filling meal, foodItems = Noodles | Chicken Rice | Tiramisu | }

Meal Category selected is Dinner.
Dinner { name = Dinner Meal, description = Handcrafted Dinner for a delighted evening, typeOfFood = Vegetarian, foodItems=Onion Fritters | }

--- Creating Custom Food Item using Builder Design Pattern ---

Customising item using Builder pattern :- Ham Sandwich
> Added Cheese
> Added French Fries
> Added Coke
Builder pattern built item Ham Sandwich Includes:- | French Fries | Coca Cola | Cheddar Cheese

Customising item using Builder pattern :- Noodles
> Added Garlic Bread
> Added Sauce
> Added Roman Lettuce
Builder pattern built item Noodles Includes:- | Garlic Bread | Tomato Sauce | Roman Lettuce

Customising item using Builder pattern :- Onion Fritters
> Added Coke
> Added Spices
> Added French Fries
Builder pattern built item Onion Fritters Includes:- | Coca Cola | French Fries | Pepper and Chilli flakes
```

3.2: Main.java - Driver Code:

```
import java.util.ArrayList;
import java.util.List;

public class Main {

    public static void main(String[] args) {

        Logger log = Logger.getLoggerInstance();
        log.logMessage("Inside Log");
        System.out.println("\n*** Hungry? Order the food you love! ***\n");

        System.out.println("\n--- Creating stacked Meals using Factory Design Pattern
        ---\n");

        // Breakfast
        //
        FoodCategory bf = MealFactory.getMealType(MealType.breakfast);
        System.out.println('\n'+bf.getMealConfiguration());
        System.out.println(bf.toString());

        // Lunch
        //
        FoodCategory lunch = MealFactory.getMealType(MealType.lunch);
        System.out.println('\n'+lunch.getMealConfiguration());
        System.out.println(lunch.toString());

        // Dinner
        //
        FoodCategory dinner = MealFactory.getMealType(MealType.dinner);
        System.out.println('\n'+dinner.getMealConfiguration());
        System.out.println(dinner.toString());

        System.out.println("\n\n\n--- Creating Custom Food Item using Builder Design
        Pattern ---\n\n\n");

        FoodItem fib1 = new FoodItem.FoodItemBuilder(Integer.valueOf("1"), "Ham
        Sandwhich", "Sides", Double.valueOf("10.88"), "Breakfast")
            .addCheese()
            .addFrenchFries()
            .addCoke()
```

```

        .build();

        FoodItem fil1 = new
FoodItem.FoodItemBuilder(Integer.valueOf("56"), "Noodles", "Maindish", Double.valueOf("19.88"), "
Lunch")

        .addGarlicBread()
        .addSauce()
        .addVeggies()
        .build();
        ;

        FoodItem fid1 = new FoodItem.FoodItemBuilder(Integer.valueOf("105"), "Onion
Fritters", "Sides", Double.valueOf("3.88"), "Dinner")
        .addCoke()
        .addSpices()
        .addFrenchFries()
        .build();

    }
}

```

3.3: Logger File:

```
2022-03-15 23:10:10 Log: Inside Log
2022-03-15 23:10:10 Log: Inside FoodItem Builder
2022-03-15 23:10:10 Log: Inside FoodItem Builder
2022-03-15 23:10:10 Log: Inside FoodItem Builder
2022-03-15 23:16:02 Log: Inside Log
2022-03-15 23:16:02 Log: Returning Breakfast Meal Factory
2022-03-15 23:16:02 Log: Returning Breakfast Meal Factory
2022-03-15 23:16:02 Log: Returning Dinner Meal Factory
2022-03-15 23:16:02 Log: Inside FoodItem Builder
2022-03-15 23:16:02 Log: Created Builder instance successfully
2022-03-15 23:16:02 Log: Inside FoodItem Builder
2022-03-15 23:16:02 Log: Created Builder instance successfully
2022-03-15 23:16:02 Log: Inside FoodItem Builder
2022-03-15 23:16:02 Log: Created Builder instance successfully
2022-03-15 23:27:21 Log: Inside Log
2022-03-15 23:27:21 Log: Returning Breakfast Meal Factory
2022-03-15 23:27:21 Log: Returning Breakfast Meal Factory
2022-03-15 23:27:21 Log: Returning Dinner Meal Factory
2022-03-15 23:27:21 Log: Inside FoodItem Builder
2022-03-15 23:27:21 Log: Created Builder instance successfully
2022-03-15 23:27:21 Log: Inside FoodItem Builder
2022-03-15 23:27:21 Log: Created Builder instance successfully
2022-03-15 23:27:21 Log: Inside FoodItem Builder
2022-03-15 23:27:21 Log: Created Builder instance successfully
```