Ministry of Education, Culture, and Research of the Republic of Moldova
Technical University of Moldova
The Faculty of Computers, Informatics, and Microelectronics

REPORT

SOLID Principles

Laboratory work no. 1

Executed by:	Cornievschi Bogdan
st. gr.	FAF-222
Verified by:	Furdui Alexandru
	univ. assist.

Chişinău – 2024

1. Introduction

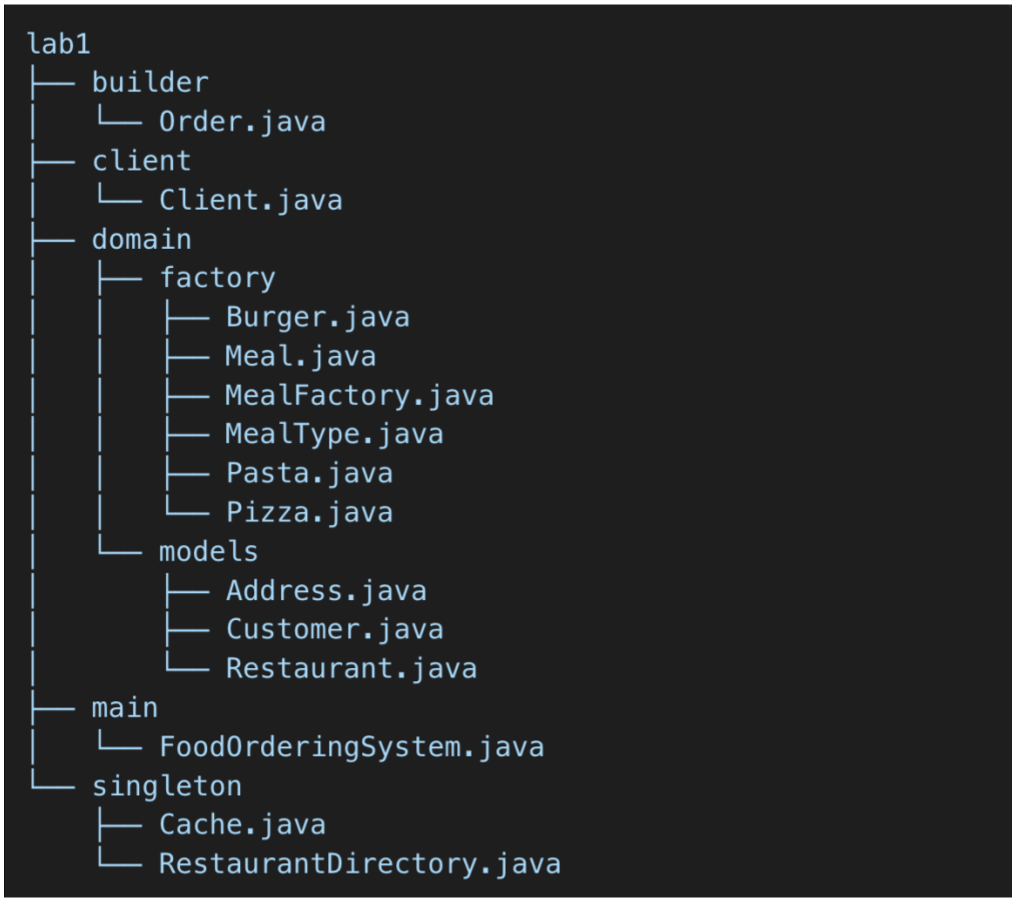
The goal of this laboratory work was to create a sample project using an object-oriented programming (OO) language and a suitable Integrated Development Environment (IDE) or editor. The project focused on implementing at least three creational design patterns to manage object instantiation within the selected domain. I selected three creational design patterns which are – singleton pattern, builder pattern and factory method pattern.

2. Domain of the project

The domain of this project is an online food ordering system in which a user can place orders to various restaurants. The system includes representations of customer, restaurant meals and orders for my example code.

3. Project structure

- **builder**: Contains the Order class, which is responsible for managing the order creation process. The OrderBuilder pattern is used to facilitate the construction of complex order objects in a readable and flexible manner.
- **client**: Contains the Client class, which represents the interaction point for customers. This class simulates how a customer places an order, interacts with the restaurant directory, and uses different components of the system.
- **domain**: Contains sub-packages that represent the core entities and factory logic for the food ordering system:
  - **factory**: Contains factory classes and associated types responsible for creating instances of meals. Classes like MealFactory, Burger, Pizza, and Pasta fall under this package. The MealFactory class abstracts the instantiation of various Meal types, promoting flexibility in the system.
  - **models**: Contains data models for the key elements in the project, such as Customer, Address, and Restaurant. These models represent the main objects involved in the food ordering process, encapsulating their respective attributes and behaviors.
- **main**: Contains the FoodOrderingSystem class, which serves as the entry point for running the application. This class initializes the components and demonstrates the interaction between various elements of the system.
- **singleton**: Contains singleton classes like RestaurantDirectory and Cache to ensure consistent access and centralized management of shared resources. These classes use the Singleton pattern to restrict instantiation to a single instance across the system, ensuring data consistency and efficient resource use.



```
lab1
├── builder
│   └── Order.java
├── client
│   └── Client.java
├── domain
│   ├── factory
│   │   ├── Burger.java
│   │   ├── Meal.java
│   │   ├── MealFactory.java
│   │   ├── MealType.java
│   │   ├── Pasta.java
│   │   └── Pizza.java
│   └── models
│       ├── Address.java
│       ├── Customer.java
│       └── Restaurant.java
├── main
│   └── FoodOrderingSystem.java
└── singleton
    ├── Cache.java
    └── RestaurantDirectory.java
```

Figure 1 – project structure

4. Factory method pattern

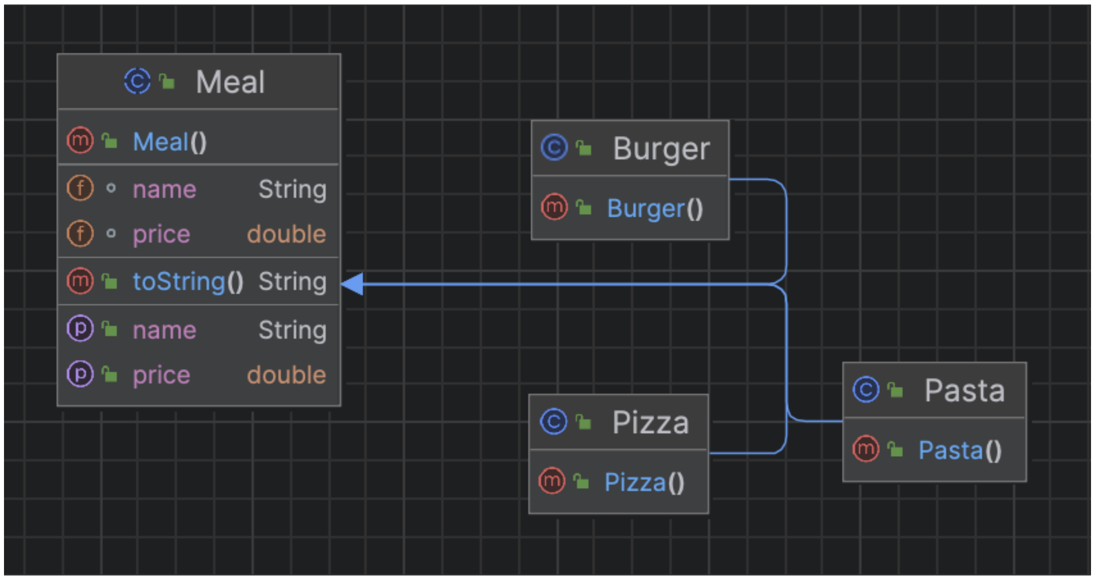


Figure 2 – diagram of implementation of factory method pattern in my project

Factory Method Pattern: The factory method is creational pattern that uses to create objects without specification of the exact calss object it will be created. It gives an interface for creating the class instance but lets its subclasses to choose which created object type.

Base Class (Meal):

This Meal class would be a superclass that meets the common properties of every type meal, like name and price… (Methods, toString).

Concrete Subclasses (Burger, Pizza and Pasta class)

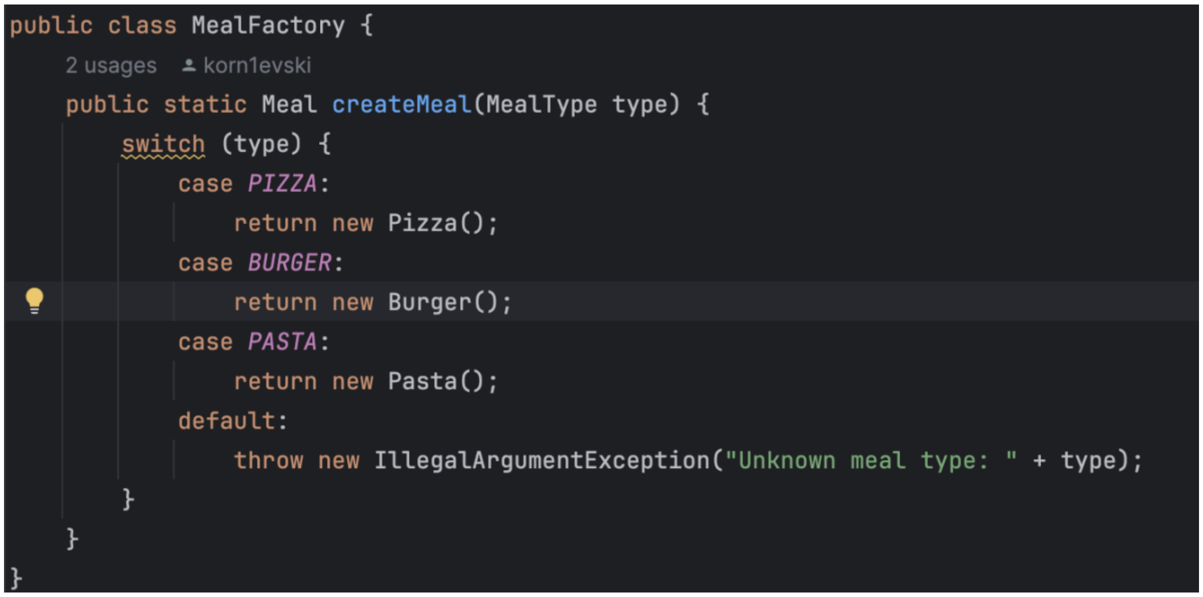
These classes extend the Meal abstract class and encapsulate specific implementation of distinct meal kinds.

Each of these classes (Burger, Pizza, Pasta) is an example for a meal type that can be made.

Factory Class (MealFactory):

Factory (Meal Factory in the diagram) - used to create the actual meal subclasses (Burger, Pizza and Pasta) (Although this is not explicitly shown directly on the picture)

The MealFactory can make its decision according to the requests and create a Burger, a Pizza or Pasta; this way the client will handle with using it without knowing how to instantiate it.



```java
public class MealFactory {
    2 usages   korn1evski
    public static Meal createMeal(MealType type) {
        switch (type) {
            case PIZZA:
                return new Pizza();
            case BURGER:
                return new Burger();
            case PASTA:
                return new Pasta();
            default:
                throw new IllegalArgumentException("Unknown meal type: " + type);
        }
    }
}
```

Figure 3 – factory method pattern implementation in my project

5. Builder pattern

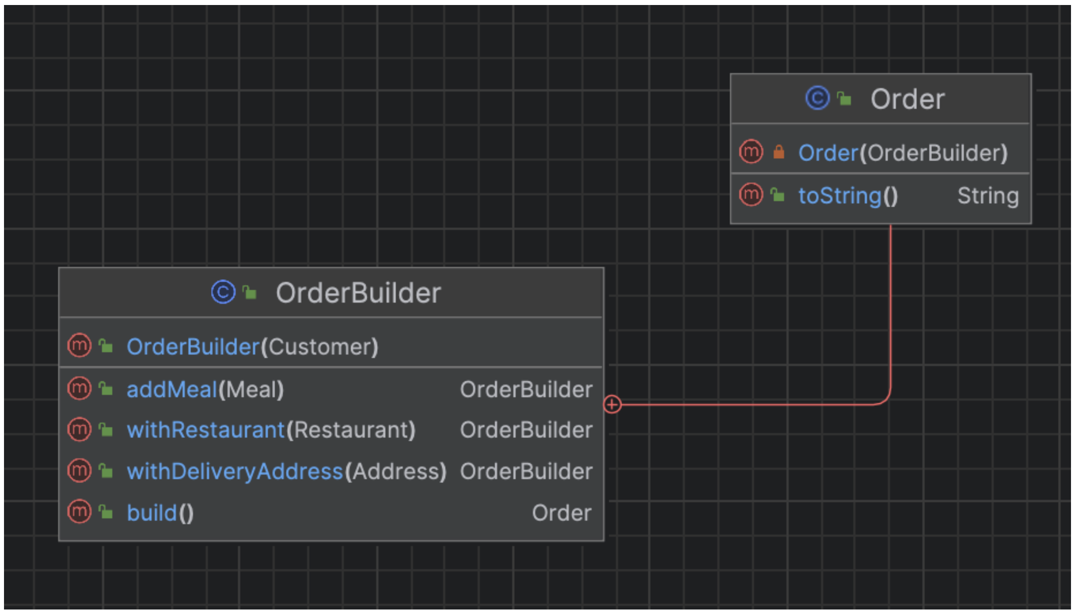
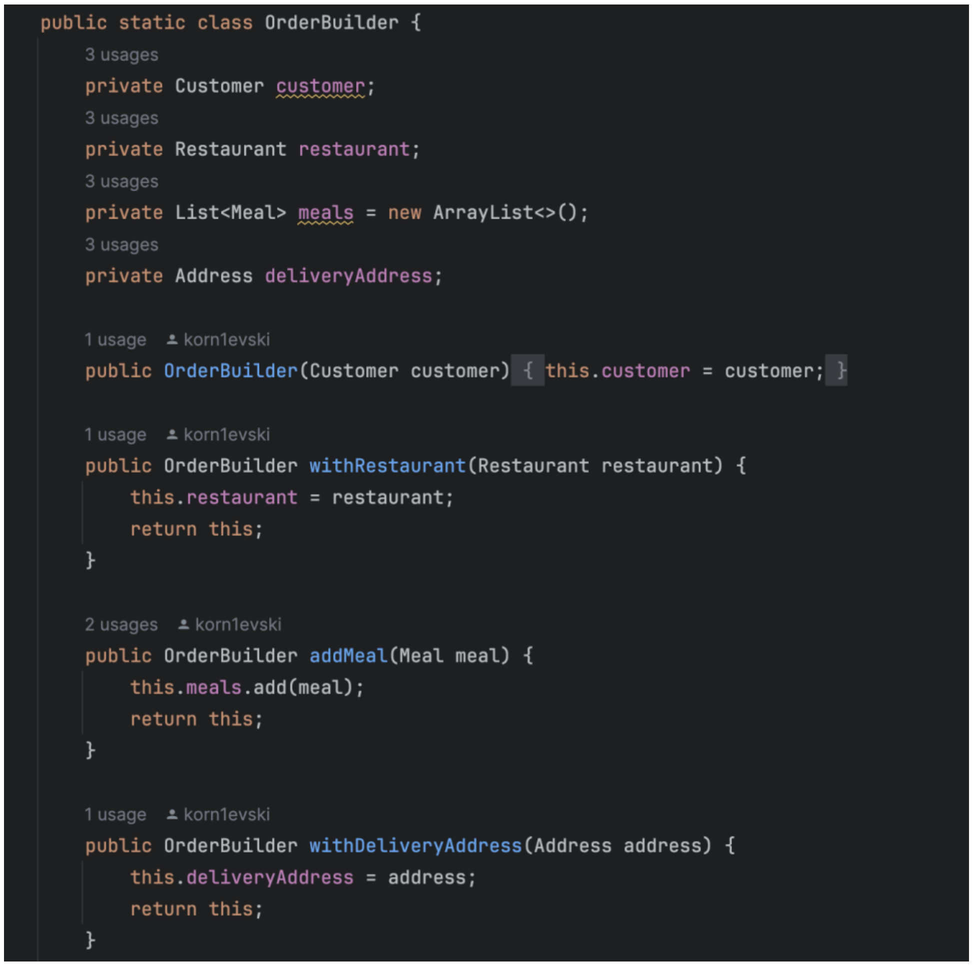

Order
Order(OrderBuilder)
toString() String
OrderBuilder
OrderBuilder(Customer)
addMeal(Meal) OrderBuilder
withRestaurant(Restaurant) OrderBuilder
withDeliveryAddress(Address) OrderBuilder
build() Order

Figure 4 – diagram of implementation of builder pattern in my project

The Builder Pattern is a creational software design pattern which aims to solve this problem by providing a way of building steps into the object construction. It allows to build different shapes of an object with the same code. This is especially useful when there are many attributes that may need separate configurations for each one.

OrderBuilder Class: This class will be an entity wrapping most functions I use to create. Order object with allowed configuration options (meals, restaurant, delivery address) and it has fluent interface which allows us to chain definitions of those configurations. Some of them like addMeal, withRestaurant, andwithDeliveryAddress method helps to build a open an Order order as per required. The methods all return an instance of OrderBuilder and so I can chain calls to them with.setData() which makes it simple to set the attributes that I want.

Order Class: Building the Order class with OrderBuilder. This shows the end result which has all of the settings that were made whilst building it. The constructor of the Order requires an OrderBuilder, so one can only be created with a named initializer and thus all code related to that object is located in it.



```
public static class OrderBuilder {
    3 usages
    private Customer customer;
    3 usages
    private Restaurant restaurant;
    3 usages
    private List<Meal> meals = new ArrayList<>();
    3 usages
    private Address deliveryAddress;

    1 usage  korn1evski
    public OrderBuilder(Customer customer) { this.customer = customer; }

    1 usage  korn1evski
    public OrderBuilder withRestaurant(Restaurant restaurant) {
        this.restaurant = restaurant;
        return this;
    }

    2 usages  korn1evski
    public OrderBuilder addMeal(Meal meal) {
        this.meals.add(meal);
        return this;
    }

    1 usage  korn1evski
    public OrderBuilder withDeliveryAddress(Address address) {
        this.deliveryAddress = address;
        return this;
    }
```

Figure 5 – builder pattern implementation in my project

6. Singleton pattern

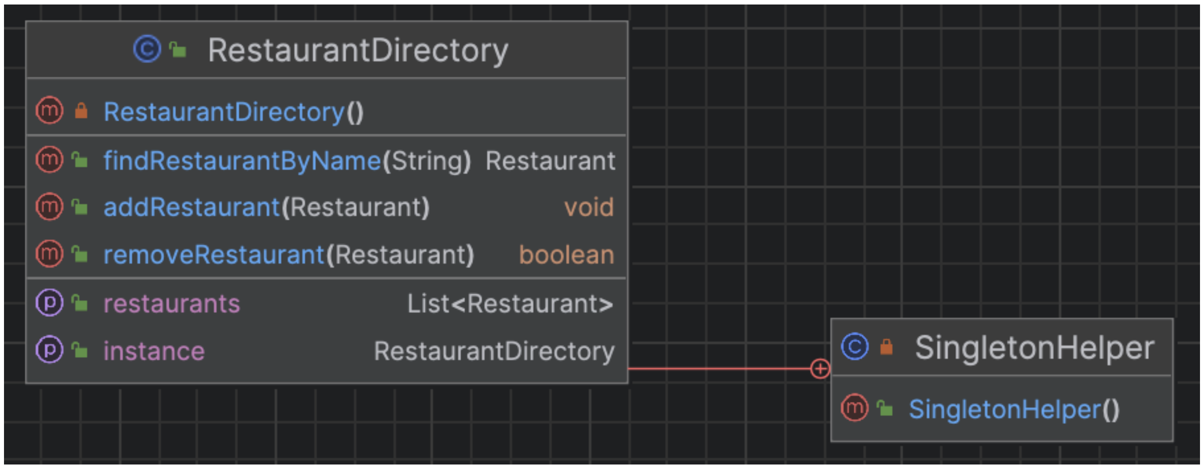

RestaurantDirectory
RestaurantDirectory()
findRestaurantByName(String) Restaurant
addRestaurant(Restaurant) void
removeRestaurant(Restaurant) boolean
restaurants List<Restaurant>
instance RestaurantDirectory
SingletonHelper
SingletonHelper()

Figure 6 – diagram of implementation of singleton pattern in my project

The **RestaurantDirectory class** used in the food ordering system is a singleton which allows only one instance of directory throughout the application. The following is a class diagram that outlines in detail the more specific components of RestaurantDirectory, such as:

Private Constructor: It specifies constructor is private and it can not be called from outside of the class. This ensures that the sole instance is internally maintained.

SingletonHelper Class: Use nested static SingletonHelper class. The above approach is threadsafe due to lazy loading, because instance will not be created until the first request comes.

Instance Management Methods:

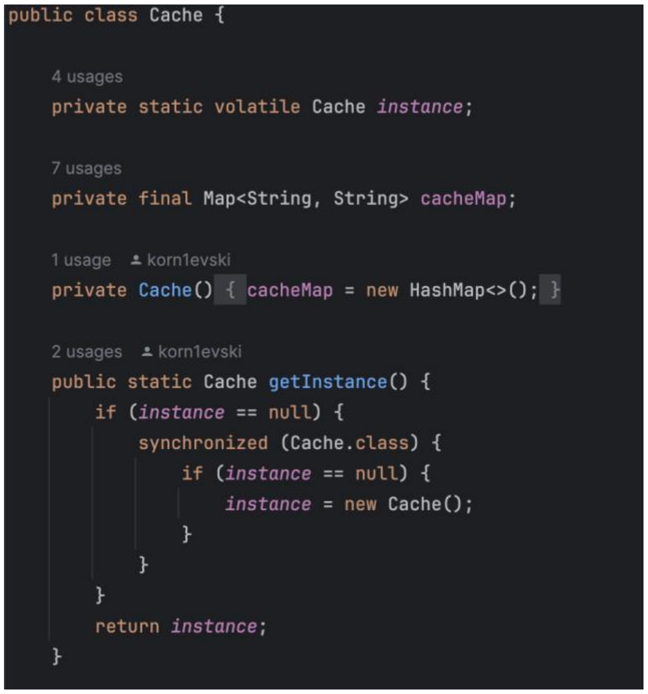
addRestaurant(Restaurant): This method is used to add a restaurant in the directory.

Also **removeRestaurant(Restaurant)** method is removes the restaurant from list if exists.

Use **findRestaurantByName(String)** for search restaurants by name.

A **getListRestaurants()** return a collection of available restaurants method was made an immutable list in collections. I wrapped it with a call to Collections. unmodifiableList() so nothing can accidentally modify it, thereby retaining better control over data integrity.

The RestaurantDirectory implements the Singleton Pattern so that it can hold all of its available restaurants in a single location and using Singletons increases data consistency across the system, lending globally consistent data access to this list.



```
public class Cache {

    4 usages
    private static volatile Cache instance;

    7 usages
    private final Map<String, String> cacheMap;

    1 usage  korn1evski
    private Cache() { cacheMap = new HashMap<>(); }

    2 usages  korn1evski
    public static Cache getInstance() {
        if (instance == null) {
            synchronized (Cache.class) {
                if (instance == null) {
                    instance = new Cache();
                }
            }
        }
        return instance;
    }
```

Figure 7– singleton pattern implementation in my project

7. Conclusion

To conclude, this laboratory work demonstrated the advantages of creational design patterns such as Singleton, Factory Method and Builder for better system scalable, maintainableability and flexibility. This way of using these patterns helped me to unify resource management, abstract complex creation logic and bring clarity for constructing the more complicated objects.