A Report on

BITS Redi Mania - Food Ordering System

Prepared For

Dr. Amit Dua | Dr. Tanmay Mahapatra

(Instructor In-Charge | Course Instructor)

CS F213 - Object-Oriented Programming

Prepared By
GROUP 67

Adarsh Goel - 2020B3A70821P
Akshansh Bhatt - 2019B5A80754P
Varun Dev Bhargava - 2020B4A70848P
Chaitanya Sethi - 2020B3A71961P
Suhani Modi - 2020B2A71136P

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
AUGUST - DECEMBER 2022

Acknowledgement

The success of this report would not have been possible without the assistance and guidance of our teachers. We would like to thank the Vice-Chancellor (BITS Pilani University), Dr Souvik Bhattacharya and the Director of BITS Pilani (Pilani Campus), Prof. Sudhir Kumar Barai. We would thank our instructors and mentors for this assignment dedicated to International Economics, Prof. Dr. Amit Dua (Instructor In-Charge) and Prof. Dr. Tanmay Mahapatra (Course Instructor) for allowing us to conduct this study under the course **CS F213 - Object-Oriented Programming** and for guiding us throughout the semester to make the project comprehensive through his insights and knowledge. Lastly, we would also like to thank our family members and close friends for their moral support.

Table of Contents

ANTI-PLAGIARISM STATEMENT

We hereby declare that this project was solely written by the members of this group. We are aware that the incorporation of code/material from other sources will be treated as plagiarism, subject to the custom and usage of the subject, according to the guidelines provided by the institution.

CONTRIBUTION TABLE

Name of Member	Contributions (Features Implemented/Research Work/Classes/Important Methods)	Signature
Adarsh Goel		
Akshansh Bhatt		
Varun Dev Bhargava		
Chaitanya Sethi		
Suhani Modi		

DESIGN PATTERNS

In software design, a design pattern is a generic, repeatable solution to a common issue. A design pattern isn't a finalized product that can be written in code right away. It is a description or model for problem-solving that may be applied in a variety of circumstances. By offering tried-and-true development paradigms, design patterns help hasten the development process. Effective software design entails taking into account problems that might not show up until later in the implementation process. The reuse of design patterns helps to eliminate subtle issues that can lead to large difficulties and enhances code readability.

The Design Patterns used or that could be used in our code are:

1.) Interpreter Design Pattern: Interpreter pattern provides a way to evaluate language grammar or expression. This type of pattern comes under behavioral patterns. This pattern involves implementing an expression interface that tells to interpret a particular context. Pattern matching can be done using this. A user can search for redi and select it and then see each redi's particular menu using this. However, we have not implemented this.

2.) Iterator Design Pattern - The iterator design pattern allows us to provide iterators to iterate through collections of objects sequentially without exposing their underlying implementation. This type of design pattern can be used for the implementation of a menu in a redi food ordering system. Because of the fact that different food items are on a menu and there is a possibility of multiple traversals, there is a possibility of the internal representation of the code being exposed on the outside. This is where the iterator design pattern is helpful and can be implemented in such a system. However, we have not implemented this.

3.) Observer Design pattern: Observer Pattern is used when there is a one-to-many relationship between objects, such as if one object is modified, its dependent objects are to be notified automatically. In our project, we haven't used this exactly since we are updating the database, but this could have been used to notify the redi owners whenever there was a change in the menu.

4.) Singleton Pattern : This patten ensures a class has only one instance and provides a global point of access to it. This allows you to prevent any other classes from creating an instance of its own. To create an instance, you need to go through the class itself. Whenever an instance of the class is required, a method built inside the class will return the single instance of the object created. The simplest way to implement singleton pattern is to declare the constructor as private instead of public.

SOLID PRINCIPLES

SOLID principles are high-level software design constructs that help us understand the need for specific design patterns and software architecture in general. Solid Principles, unlike Design Patterns, are abstract. Here is a list of all the SOLID principles, their brief description, and how we have used them in our project -

The Single Responsibility Principle (SRP)

The SRP states that a class should do one thing and one thing only. As a commonly used definition, "every class should have only one reason to change". We have kept this principle in mind while designing all our classes. The names of our classes our quite descriptive in themselves. For instance, we have a class, `` `StudentLoginAndRegister` ``, which happens to serve only one functionality- handle the login of existing students and registration of new students in the database (or CSV File). Similarly, we have separated all the data classes and their functionalities. Classes like `` `Customer` ``, `` `Item` ``, `` `Order` ``, and `` `Redi` `` only have simple getters and setters to initialize and retrieve the state and are the fundamental data classes in our design.

Open-Closed Principle (OCP)

The Open-Closed Principle states that classes should be unrestricted for extension but restricted to modification. Most of the classes used in the project are extendable to add new functionalities and override existing ones. Suppose we need to implement different billing plans/methodologies for a student and a faculty member at a redi; we can extend the class `` `Customer` `` into `` `Student` `` and `` `Faculty` ``, both inheriting the fundamental data fields of the `` `Customer` `` class.

Liskov Substitution Principle (LSP)

The Liskov Substitution Principle states that subclasses should be compatible for substitution with their base classes. Since all our classes are pretty fundamental in nature, we don't have such a complex web of inheritance patterns to which this principle could be applied directly. However, if, in the future, we expand the functionalities, this will definitely be taken into account.

Interface Segregation Principle (ISP)

Segregation means keeping things separated, and the Interface Segregation Principle is about separating the interfaces. The principle states that many client-specific interfaces are better than one general-purpose interface. Clients should not be forced to implement a function they do not need. This SOLID principle is also not used in our project because we didn't define any interface of our own.

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP) says that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions. Every class in our project is in accordance with this principle. We have no class which we have extended from a concrete class. We have only extended default interfaces in Java which are required for multithreading functionality (``Runnable``) and for defining custom comparators (``Comparable<T>``).