

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

Acknowledgement.....	1
Table of Contents.....	2
Anti-Plagiarism Statement.....	3
Contribution Table.....	4
Design Patterns	5
SOLID Principles.....	7
UML Diagrams.....	8

ANTI-PLAGIARISM STATEMENT




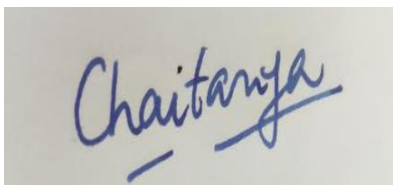

We hereby declare that this project was solely written by the members of this group. We are aware that the incorporation of code/material from other sources will be treated as plagiarism, subject to the custom and usage of the subject, according to the guidelines provided by the institution.

Link for all the Video Recording:

https://drive.google.com/drive/folders/1U_V_vx9T-STaqwDeLCaWZ8y5IkqXd1tR?usp=share_link

How to Run: Open the project in VS Code. Open the Driver class and run it. All other instructions are mentioned in this class.

CONTRIBUTION TABLE

Name of Member	Contributions (Features Implemented/Research Work/Classes/Important Methods)	Signature
Adarsh Goel	Order, Inventory, Redi class	
Akshansh Bhatt	Admin, AdminActivities class	
Varun Dev Bhargava	Timestamp, Item, CustomPair	
Chaitanya Sethi	Customer, Student	
Suhani Modi	StudentActivities, StudentLoginandRegister	

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. Reuse of design patterns helps to eliminate subtle issues that can lead to large difficulties and enhances code readability.

Following is the design pattern that we have implemented in our code:

- 1.) **Singleton Pattern:** This pattern 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.

We have used this pattern to implement all the activities related to the Admin. Since, there can be only one Admin at a time, we created a class Singleton class called Admin with a single object. Now, Admin class extends AdminActivities class which contains all the required methods/functionality that could have been used by the Admin. So, there is only one person who is Admin and can do all the required functions.

The Design Patterns that could have been 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 pattern. This pattern involves implementing an expression interface which 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.
2. **Iterator Design Pattern:** The iterator design pattern allows us to provide iterators to iterate through collections of objects sequentially without exposing its 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 in 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.

3. **Command Design Pattern:** This is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. In this pattern, the request is wrapped under an object which is called Command. Now this object is passed to invoker object. We could have used this pattern to order or cancel food items.

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.

The Single Responsibility Principle: This principle states that A class should have one and only one reason to change, meaning that a class should have only one job. In our code, there are some classes that follows this principle. For e.g.: StudentLoginAndRegister class takes all the students' details from StudentList.csv and registers them into the database. Other classes that follow this principle are Item class, Order class, CustomPair class. There may be some classes which do not follow this principle in our code.

Open and Closed Principle: This principle states that classes should be open for extension and closed to modification. In our code, this is shown through Admin class which extends AdminActivities class. Now, Admin class gains all the functionality of the AdminActivities class and can also add own functionalities.

Liskov's Substitution: The Liskov Substitution Principle states that subclasses should be substitutable for their base classes. This principle can be violated when we use inheritance and pass an object which is referenced by a superclass reference. But since our code is not extensive in terms of inheritance, there is no scope for the violation of Liskov Substitution Principle.

Interface Segregation Principle: This principle states that "Client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use." Since, our code never made use of such an interface, this principle is not violated. Our code made use of Comparable and Runnable interface.

Dependency Inversion Principle: The Dependency Inversion principle states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions. This way, instead of high-level modules depending on low-level modules, both will depend on abstractions. This principle is violated in our code as we store objects of other classes (Redi) in another class (Inventory). This can be resolved by making interface Redi and extending this Redi interface by Shankar, Meera ... classes. In this way, we can store objects of different redis in interface reference.


```

classDiagram
    class StudentLoginAndRegister {
        +StudentLoginAndRegister()
        +register(): void
    }
    class Student {
        +Student(String, String, String, String)
        +emailId: String
        +bitId: String
        +name: String
        +phoneNo: String
        +updateItem(Item, int, String): void
        +generateOrderHistory(): void
        +resetMonthlyexp(): void
        +name: String
        +bitId: String
        +emailId: String
        +phoneNo: String
    }
    class StudentActivities {
        +StudentActivities()
        +takeOrders(): void
        +resetMonthlyExpenditure(): void
        +showOrder(): void
        +placeOrder(): void
    }
    class Admin {
        +Admin()
        +instance: Admin
    }
    class AdminActivities {
        +AdminActivities()
        +addItem(): void
        +removeItem(): void
        +getInsights(): void
        +getRevenueSummary(): void
    }
    class Timestamp {
        +Timestamp()
        +getMonthFromUnixTime(int): int
        +unixTimeToHumanReadable(int): String
        +getHHFromUnixTime(int): int
    }
    class Customer {
        +Customer()
        +registerStudent(String, Student): void
        +showOrderHistory(): void
        +checkPresent(String): boolean
        +showCustomers(): void
    }
    class Driver {
        +Driver()
        +main(String[]): void
    }
    class Redi {
        +Redi(String)
        +totalRevenue: int
        +rediName: String
        +removeItem(String): void
        +placeOrder(Student, String, int): boolean
        +addItem(Item, int): void
        +showInventory(): void
        +maxSoldItem: CustomPair
        +totalRevenue: int
        +rediName: String
    }
    class Inventory {
        +Inventory()
        +getInsights(): void
        +config(): void
        +showRediInventory(): void
        +getRedi(String): Redi?
        +placeOrderAtRedi(Student, String, String, int, int): void
        +getRevenueSummary(): void
    }
    class Item {
        +Item(String, int, String)
        +itemId: String
        +itemName: String
        +itemPrice: int
        +toString(): String
        +itemName: String
        +itemPrice: int
        +itemId: String
    }
    class CustomPair {
        +CustomPair(Item, int)
        +item: Item
        +quantity: int
        +compareTo(CustomPair): int
        +item: Item
        +quantity: int
        +name: String
        +price: int
        +itemId: String
    }
    class Order {
        +Order(int, String, String, String, int)
        +compareTo(Order): int
        +run(): void
        +toString(): String
    }

    StudentLoginAndRegister --> Student : «create»
    Student --> StudentActivities : «create»
    Admin --> AdminActivities : «create»
    Student --> CustomPair : «create»
    CustomPair --> Item : item
    Item --> Inventory : «create»
    Inventory --> Redi : «create»
    Redi --> Inventory : «create»
    Customer --> Item : «create»
    Driver --> Inventory : «create»
    
```

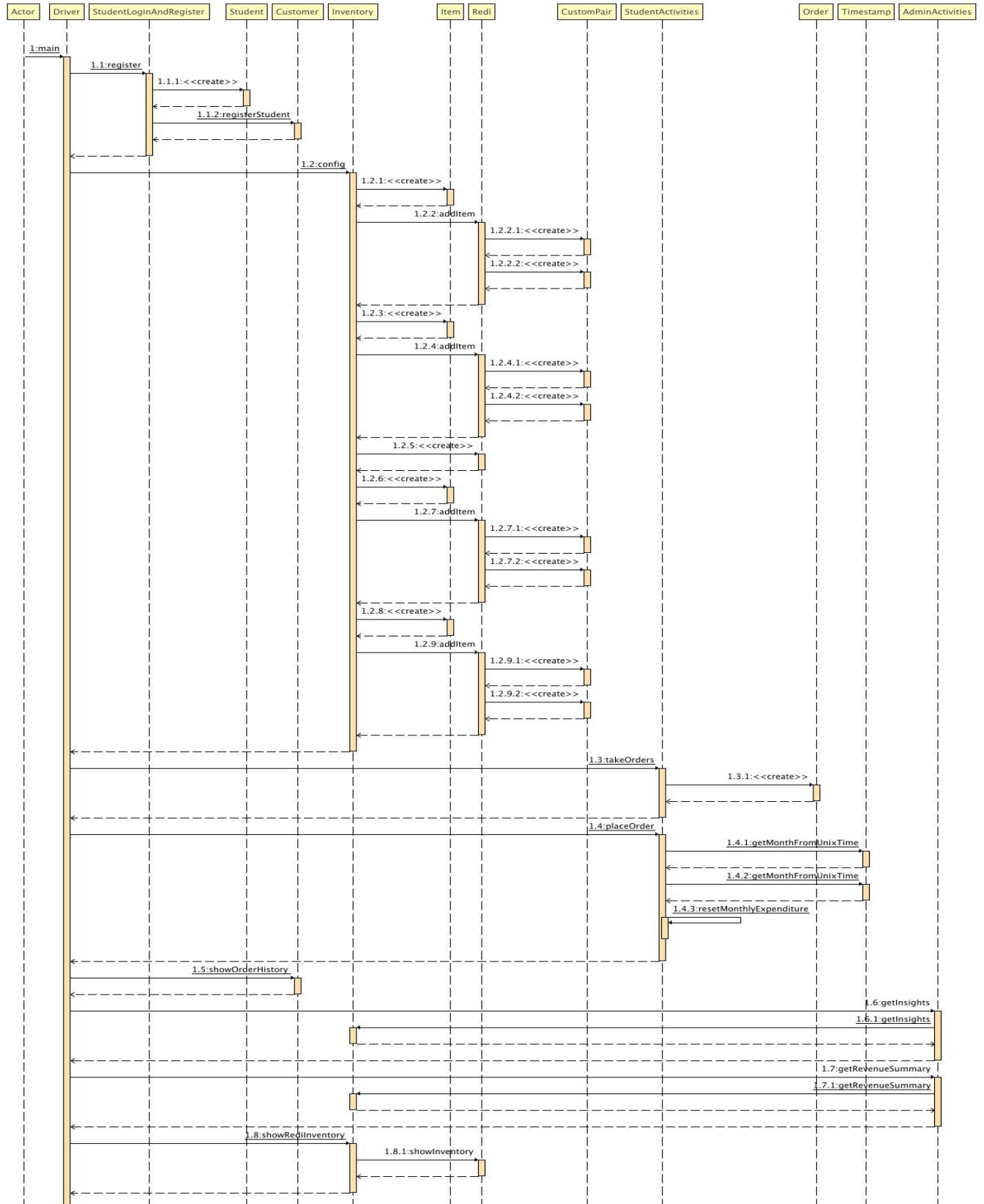
The diagram illustrates the following classes and their components:

- StudentLoginAndRegister**:
 - Method: `StudentLoginAndRegister()`
 - Method: `register(): void`
- Student**:
 - Method: `Student(String, String, String, String)`
 - Attributes: `emailId: String`, `bitId: String`, `name: String`, `phoneNo: String`
 - Methods: `updateItem(Item, int, String): void`, `generateOrderHistory(): void`, `resetMonthlyexp(): void`
 - Attributes: `name: String`, `bitId: String`, `emailId: String`, `phoneNo: String`
- StudentActivities**:
 - Method: `StudentActivities()`
 - Methods: `takeOrders(): void`, `resetMonthlyExpenditure(): void`, `showOrder(): void`, `placeOrder(): void`
- Admin**:
 - Method: `Admin()`
 - Attribute: `instance: Admin`
- AdminActivities**:
 - Method: `AdminActivities()`
 - Methods: `addItem(): void`, `removeItem(): void`, `getInsights(): void`, `getRevenueSummary(): void`
- Timestamp**:
 - Method: `Timestamp()`
 - Methods: `getMonthFromUnixTime(int): int`, `unixTimeToHumanReadable(int): String`, `getHHFromUnixTime(int): int`
- Customer**:
 - Method: `Customer()`
 - Methods: `registerStudent(String, Student): void`, `showOrderHistory(): void`, `checkPresent(String): boolean`, `showCustomers(): void`
- Driver**:
 - Method: `Driver()`
 - Method: `main(String[]): void`
- Redi**:
 - Method: `Redi(String)`
 - Attributes: `totalRevenue: int`, `rediName: String`
 - Methods: `removeItem(String): void`, `placeOrder(Student, String, int): boolean`, `addItem(Item, int): void`, `showInventory(): void`
 - Attributes: `maxSoldItem: CustomPair`, `totalRevenue: int`, `rediName: String`
- Inventory**:
 - Method: `Inventory()`
 - Methods: `getInsights(): void`, `config(): void`, `showRediInventory(): void`, `getRedi(String): Redi?`, `placeOrderAtRedi(Student, String, String, int, int): void`, `getRevenueSummary(): void`
- Item**:
 - Method: `Item(String, int, String)`
 - Attributes: `itemId: String`, `itemName: String`, `itemPrice: int`
 - Method: `toString(): String`
 - Attributes: `itemName: String`, `itemPrice: int`, `itemId: String`
- CustomPair**:
 - Method: `CustomPair(Item, int)`
 - Attributes: `item: Item`, `quantity: int`
 - Method: `compareTo(CustomPair): int`
 - Attributes: `item: Item`, `quantity: int`, `name: String`, `price: int`, `itemId: String`
- Order**:
 - Method: `Order(int, String, String, String, int)`
 - Method: `compareTo(Order): int`
 - Method: `run(): void`
 - Method: `toString(): String`

Relationships and Dependencies:

- StudentLoginAndRegister** creates **Student** (indicated by a dashed arrow labeled «create»).
- Student** creates **StudentActivities** (indicated by a dashed arrow labeled «create»).
- Admin** creates **AdminActivities** (indicated by a dashed arrow labeled «create»).
- Student** creates **CustomPair** (indicated by a dashed arrow labeled «create»).
- CustomPair** has a reference to **Item** (indicated by a solid arrow labeled `item`).
- Item** creates **Inventory** (indicated by a dashed arrow labeled «create»).
- Inventory** creates **Redi** (indicated by a dashed arrow labeled «create»).
- Redi** creates **Inventory** (indicated by a dashed arrow labeled «create»).
- Customer** creates **Item** (indicated by a dashed arrow labeled «create»).
- Driver** creates **Inventory** (indicated by a dashed arrow labeled «create»).

UML SEQUENCE DIAGRAM



UML USE CASE DIAGRAM