Declaration of Original Work for SC2002 Assignment

We hereby declare that the attached group assignment has been researched, undertaken, completed, and submitted as a collective effort by the group members listed below.

We have honoured the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work. In addition, disciplinary actions may be taken.

Name	Course	Lab Group	Signature/ Date
Solis Aaron Mari Santos	SC2002	FDAB	Solis Aaron Mari Santos / 25-04-24
Dana Yak	SC2002	FDAB	Dana Yak / 25-04-24
Isaac Wong Jia Kai	SC2002	FDAB	Isaac Wong Jia Kai / 25-04-24
Tan Pei Wen Jamie	SC2002	FDAB	Tan Pei Wen Jamie / 25-04-24
Toh Jun Sheng	SC2002	FDAB	Toh Jun Sheng / 25-04-24

1. Introduction

This report delves into the design considerations implemented in our Fastfood Ordering and Management System (FOMS) project. It covers the design pattern, the SOLID design principles, and the four Object-Oriented Programming (OOP) concepts that we adopted to ensure reusability, extensibility, and maintainability of our program. This report also includes a detailed UML Class Diagram, Test Cases and Results, as well as a Reflection section highlighting the difficulties we encountered, the knowledge we acquired, and suggestions for further improvements. The source code for our project can be accessed here.

2. Design Considerations

2.1 Design Pattern

2.1.1 Model-View-Controller (MVC)

In this project, we implemented the Model-View-Controller architectural pattern, which organises the system into three main components: Model, View, and Controller, with additional packages like Enums, Exceptions, Interfaces, Main, and Services. This approach creates a clear separation of concerns, allowing us to manage and modify each component independently without affecting the others. For example, a change in how data is stored and managed (Model) can be done without having to make changes to how the user interface looks (View) or how user actions are handled (Controller). This architectural pattern improves code organisation and readability, making it easier for developers to understand, maintain, and extend the current system.

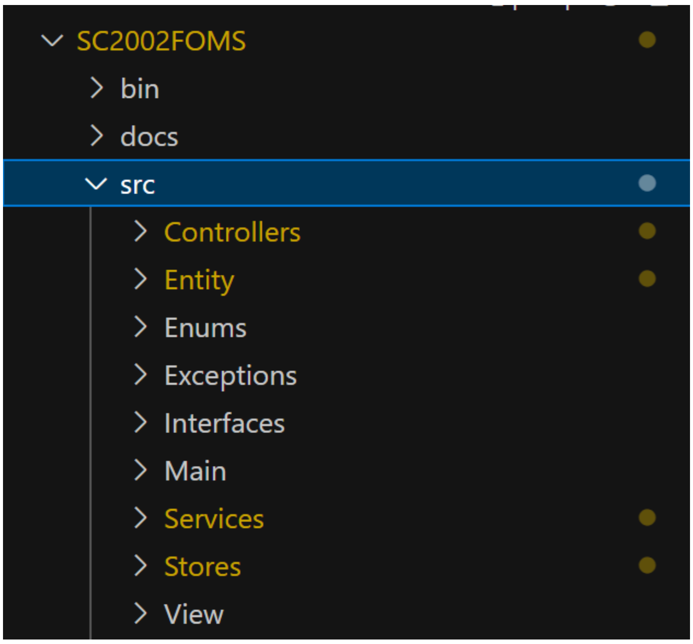

Figure 1: MVC design pattern

Our Model component encompasses the Entity and Stores packages (see Figure 1), which handle the data structures, database operations, and logic. They manage the backend, allowing data to be updated when changes are made. For example, the NewOrder class in the Entity package is able to manage the state of the cart object, allowing customers to add, edit, and remove items in the cart object.

The View package is responsible for presenting information to the user. It displays the data or information from the Model to the user and sends user input to the Controller for processing. In our case, the classes in the View package are responsible for displaying an output to users. For example, the MenuDisplay prints the specific branch menu to Customers.

The Controller package acts as an intermediary between the Model and the View. It handles user input, updates the Model based on that input, and updates the view to reflect changes in the Model. For example, the CustomerController in the Controller package receives the input from the user and calls the NewOrder in the Entity package to initiate a new Cart object for Customers to add their orders.

2.2 Design Principles (SOLID)

2.2.1 Single Responsibility Principle

“There should never be more than ONE reason for a class to change”.
The FOMS is organised into distinct packages, each tasked with handling a single responsibility. Within these packages, each class is designed to perform a specific task related to its package's responsibility. For example, the View package has separate view classes, MenuDisplay and StaffDisplay. While both are responsible for presenting information to users, the MenuDisplay displays menu items to Customers, and the StaffDisplay displays the staff list to Managers. By adhering to this principle, we avoid creating a monolithic ‘God’ class that tries to encompass multiple responsibilities. This fosters better code organisation, ensuring the system remains adaptable to future changes and enhancements.

2.2.2 Open-Closed Principle (OCP)

“A module should be open for extension but closed for modification”.
In our system, we utilise interfaces to establish clear contracts for the behaviour expected from different modules, thus creating a structured way to add or change functionalities through extension rather than extensive modifications to the existing codebase. For example, we implemented an IPayment interface to enable new payment methods to be added to our system without having to make major modifications to the codebase. Hence, we encourage new functionalities to be added with ease.

2.2.3 Liskov Substitution Principle (LSP)

"Subtypes must be substitutable for their base types without altering the correctness of the program."
To ensure that our system does not violate the LSP, we enforce the derived classes to have pre-conditions no stronger than the base class methods, and post-conditions no weaker than the base class method. In this way, the derived classes never introduce new requirements that are not presented in the base class, and never compromise the deliverables the base class provides. This is achieved by utilising inheritance meaningfully and ensuring that the overridden methods follow the same contract (i.e. same parameters and return types) as the base class methods.

2.2.4 Interface Segregation Principle (ISP)

“Many client-specific interfaces are better than one general purpose interface”.
As much as possible, we create multiple smaller interfaces that are specific to the needs of individual modules instead of a large, general-purpose interface. Creating interfaces with methods specific to each module's requirements ensures that classes are not reliant on interfaces they do not utilise, thereby fostering a more modular and cohesive design. In our system, instead of a single comprehensive interface for AdminController, we divided it into separate interfaces such as IBranchManagement, IPaymentManagement, and IStaffManagement to handle specific Admin actions. Thus, if a Manager is promoted to a “Senior Manager” and has access to Add/Remove payment methods, they can directly implement the IPaymentManagement interface, maintaining the principle of ISP.

2.2.4 Dependency Inversion Principle (DIP)

By implementing interfaces, we are able to achieve loose coupling and high cohesion between classes. The high-level classes depend on interfaces rather than on concrete implementations. Because interfaces have a lower chance of being modified compared to concrete implementations, establishing dependencies with interfaces instead of concrete classes reduces the likelihood of needing to make changes to high-level classes when implementations change. For example, we use the IOrderManager interface to define the methods (Figure 2) for StaffController and ManagerController (high-level classes) without specifying the exact implementation. This approach allows us to make changes to displayNewOrder and processOrder easily without modifying the dependent classes.

```java
package Interfaces;
import java.io.IOException;
public interface IOrderManager {
    void displayNewOrder(String branch) throws IOException;
    void processOrder(String branch) throws IOException;
    void viewDetails() throws IOException;}
```

Figure 2: IOrderManager Interface

2.3 Object-Oriented Concepts

2.3.1 Abstraction

Abstraction simplifies the complexity of our FOMS by hiding complex implementation details and showing only the essential features. This was achieved through creating an IPayment interface (Figure 3), which abstracts away the implementation details of how the processPayment() works. The interface thus provides a standardised contract for payment processing, enabling the implementation of different payment methods without the need to understand how payments are processed. Moreover, the MVC design we adopted groups the classes based on their roles in the FOMS, allowing us to locate classes easily in a well-organised structure. This is also a form of abstraction.

```
package Interfaces;
public interface IPayment {
    public abstract void processPayment();
}
```

Figure 3: IPayment Interface

2.3.2 Encapsulation

Encapsulation helps protect an object's data by making its attributes private, which means they can only be accessed or modified through public getter and setter methods. For example, the Branch class (Figure 4) in the Stores package contains private attributes such as the name and location which can only be accessed through the class’s public get methods (Figure 5). This approach ensures that the internal state of an object is controlled and prevents unauthorised access or modification of its data by other classes.

```
public class Branch {
    private String name;
    private String location;
    private int staffQuota;
    private BranchStatus branchStatus;
}
```

Figure 4: Private Attributes in Branch Class

```
    public String getName(){
        return name;}
    public String getLocation(){
        return location;}
    public int getStaffQuota(){
        return staffQuota;}
    public BranchStatus getBranchStatus(){
        return branchStatus;}
}
```

Figure 5: Getter Methods in Branch Class

2.3.3 Inheritance

Inheritance allows us to derive new classes from existing classes. This is an "is-a" relationship between two classes where the child class inherits all the attributes and methods from the parent class. By applying this concept, we can create new classes without duplicating the code extensively, especially when two or more classes share similar behaviours and attributes. For example, since Managers can perform all actions that the Staff can, the Manager Class inherits from the Staff Class, thereby minimising the need to duplicate code in the Manager class.

2.3.4 Polymorphism

Polymorphism allows objects of different types to be treated as instances of a common superclass, promoting code reusability and a more straightforward approach to managing user functionalities in our system. For example, the abstract start() method defined in EmployeeController is overwritten in the subclasses StaffController and AdminController. ManagerController inherits from StaffController and overwrites start() with their own implementation once again, these instances of overwriting showing polymorphism, where start() is a common functionality.

2.4 Assumptions Made

We assume that the users understand how to navigate the FOMS and what they can do according to their roles (Customer, Staff, Manager, Admin). Additionally, we assume the data in our database is consistent, indicating that it has been validated and handled before being used in our system. This means that the data should not cause any errors during run time, such as missing values or incorrect data types. Lastly, we also assume that the customer could remember their OrderID if they do not opt for a receipt.

3. UML Class Diagram

Figure 6 illustrates the UML Class Diagram of our FOMS, showing the class relationships and dependencies between different classes and packages. A clearer diagram image is available here.

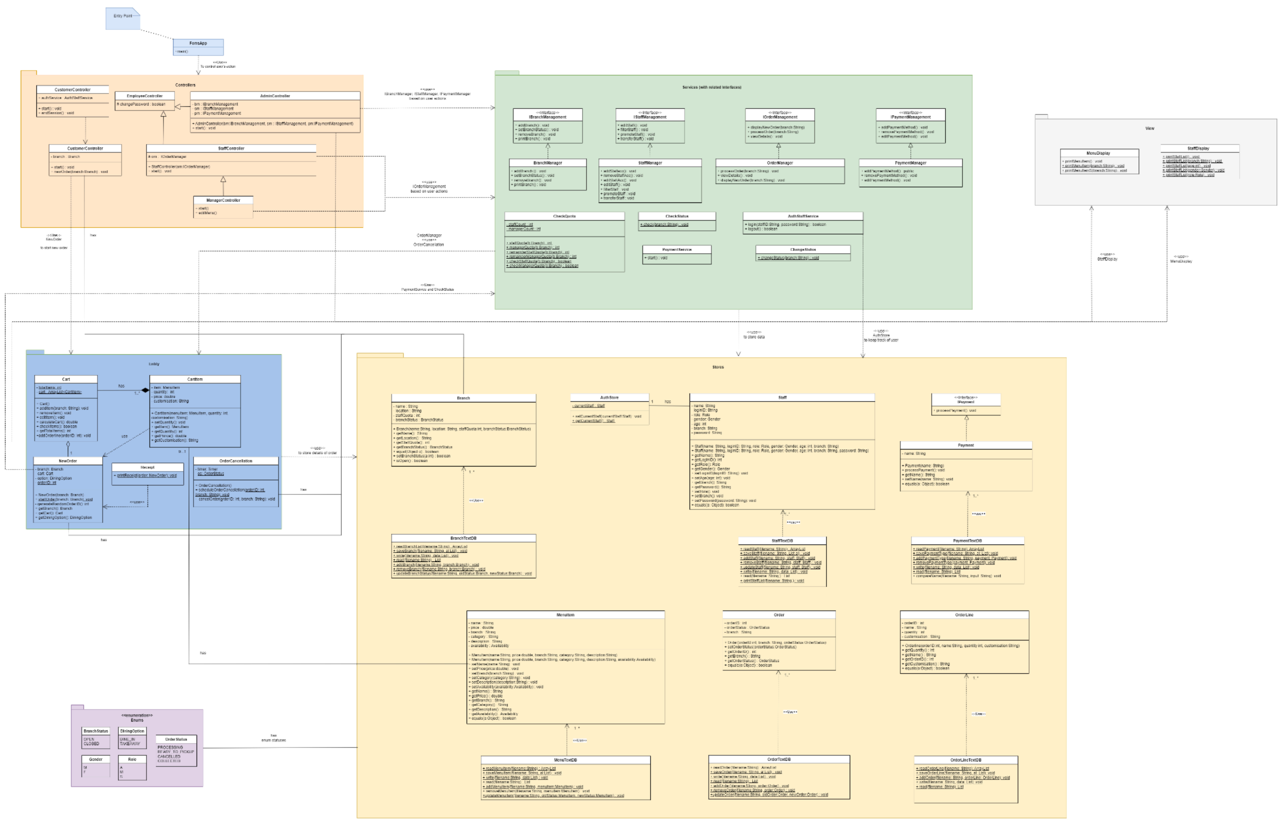

Figure 6: UML Diagram

4. Tests Cases and Results

4.1 Manager's action: Menu Management

Test Case	Test Description	Expected Result	Pass/ Fail
1	(a) Add a new menu item with a unique name, price, description, and Category. (b) **Verify** that the menu item is successfully added.	System prints "Item has been successfully added", then prints out the updated menu.	Pass
2	(a) Update the price and description of an existing menu item. (b) **Verify** that the changes are reflected in the menu.	User chooses an item to update, the system prints out the item's current details before asking if the user wants to update each of the attributes. Once done, the system prints the updated menu.	Pass
3	(a) Remove an existing menu item. **(b) Verify** that the menu item is no longer available.	System prompts users to choose which item they want to remove, then prints "Item removed successfully" when done and prints	Pass

		the updated menu.	

4.2 Order Processing

Test Case	Test Description	Expected Result	Pass/ Fail
4	(a) Place a new order with multiple food items, customise some items, and choose the takeaway option. (b) **Verify** that the order is created successfully.	Menu is displayed and the user chooses the actions to do (add item, customise, payment type, dine in/ takeout). After confirming the cart and payment, the system prints the list of items ordered, price and orderID.	Pass
5	(a) Place a new order with the dine-in option. **(b) Verify** that the order is created with the correct preferences.	The same as case 4.	Pass

4.3 Payment Integration

Test Case	Test Description	Expected Result	Pass/ Fail
6	(a) Simulate payment using a credit/debit card. (b) **Verify** that the payment is processed successfully.	System prints "Processing Debit/Credit payment…" "Successful".	Pass
7	(a) Simulate payment using an online payment platform (e.g., PayPal). (b) Verify that the payment is processed successfully.	For example, if the online payment platform is ApplePay, the system prints "Processing ApplePay payment…" "Successful"	Pass

4.4 Order Tracking

Test Case	Test Description	Expected Result	Pass/ Fail
8	(a) Track the status of an existing order using the order ID. **(b) Verify** that the correct status is displayed.	When the user "logs-in" with their Order ID, the system prints out the Order Status.	Pass

4.5 Staff Actions

Test Case	Test Description	Expected Result	Pass/ Fail
9	(a) Login as a staff member and display new orders. (b) **Verify** that the staff can see all the new orders in the branch he/she is working.	Staff logs in and chooses option 1 to display all new orders, which is then printed out.	Pass

10	(a) Process a new order, updating its status to "Ready to pickup." **(b) Verify** that the order status is updated correctly.	Staff logs in then chooses option 2 to process order and this changes the order status to “Ready to pickup”. This is verified by logging in as the customer with the Order ID and choosing option 2 to check order status, which reflects “Ready to pickup”.	Pass

4.6 Manager Actions

Test Case	Test Description	Expected Result	Pass/ Fail
11	(a) Login as a manager and display the staff list in the manager’s branch. (b) Verify that the staff list is correctly displayed.	The Staff List for the branch the manager belongs to. We open the excel list to verify if the printed list tallies with the names under the specific branch.	Pass
12	(a) A manager should be able to process order as described in Test Case 9	Managers can do the same things a Staff does, so choosing option 2 then inputting the OrderID to process order would update and print “READY_TO_PICKUP” status.	Pass

4.7 Admin Actions

Test Case	Test Description	Expected Result	Pass/ Fail
13	(a) Close a branch. **(b) Verify** that the branch does not display in Customer’s Interface anymore.	Choose which branch to open/close. If a branch is already open/ close, it prints a message to say so. If a branch is being closed, when a customer logs in afterwards, the branch does not show up as an option anymore.	Pass
14	(a) Login as an admin and display the staff list with filters (branch, role, gender, age). (b) **Verify** that the staff list is correctly filtered.	Filtered list is printed out and can be checked against the datafile.	Pass
15	(a) Assign managers to branches with the quota/ratio constraint. (b) **Verify** that managers are assigned correctly.	If the branch chosen has already reached the limit, error message “Too many managers in X. Remove some managers.” is printed.	Pass
16	(a) Promote a staff member to a Branch Manager. (b) Verify that the staff is promoted successfully.	System asks for confirmation before promoting a staff member, then prints “Staff has been promoted successfully.”	Pass

17	(a) Transfer a staff/manager among branches. (b) **Verify** that the transfer is reflected in the system.	Same process as test case 15.	Pass

4.8 Customer Interface

Test Case	Test Description	Expected Result	Pass/ Fail
18	(a) Place a new order, check the order status using the order ID, and collect the food. (b) **Verify** that the order status changes from "Ready to pickup" to "completed."	If the order is already processed and the customer checks OrderStatus, "Ready to pick up" is printed. When the customer chooses "Collect Order" and enters their OrderID, the status is printed as "COMPLETED".	Pass

4.9 Error Handling

Test Case	Test Description	Expected Result	Pass/ Fail
19	(a) Attempt to add a menu item with a duplicate name. (b) **Verify** that an appropriate error message is displayed.	"Item already exists" is printed	Pass
20	(a) Attempt to process an order without selecting any items. (b) Verify that an error message prompts the user to select items.	Prompts user to select an orderID, will not be brought to the next instance if no valid ID is selected, and has the option to go back to the previous page with -1 input.	Pass

4.10 Extensibility

Test Case	Test Description	Expected Result	Pass/ Fail
21	(a) Add a new payment method. (b) **Verify** that the new payment method is successfully added.	Prints the updated list of payment methods after successfully adding it.	Pass
22	(a) Open a new branch. (b) Verify that the new branch is added without affecting existing functionalities.	Asks for Name, Location, Staff Quota. Prints "Branch has been successfully added".	Pass

4.11 Order Cancellation

Test Case	Test Description	Expected Result	Pass/ Fail

23	(a) Place a new order and let it remain uncollected beyond the specified timeframe. (b) **Verify** that the order is automatically cancelled and removed from the "Ready to pick up" list.	After staff processes the order, the status is changed to “Ready to pickup”. If a customer does not collect within a specific timeframe, the order is cancelled and status is changed to “CANCELLED”.	Pass

4.13 Login System

Test Case	Test Description	Expected Result	Pass/ Fail
24	(a) Attempt to log in with incorrect credentials as a staff member. (b) Verify that an appropriate error message is displayed.	User fails to log in, system prints “ Invalid UserID/Password. Invalid credentials, please try again.”	Pass
25	(a) Log in as a staff member, change the default password, and log in again with the new password. (b) **Verify** that the password change functionality works as expected.	Successful login with the new password.	Pass

4.14 Staff List Initialization

Test Case	Test Description	Expected Result	Pass/ Fail
26	(a) Upload a staff list file during system initialisation. (b) Verify that the staff list is correctly initialised based on the uploaded file.	Admin to login and display staff list, then check against the data file to see that it is correct.	Pass

4.15. Data Persistence

Test Case	Test Description	Expected Result	Pass/ Fail
27	(a) Perform multiple sessions of the application, adding, updating, and removing menu items. (b) Verify that changes made in one session persist and are visible in subsequent sessions.	Changes to menu items remain even when we rerun the code.	Pass

5. Reflection

5.1 Difficulties Encountered and How We Conquered Them

- We faced issues when collaborating in Github as we could not find a way to clone the repository using Eclipse. Hence, we shifted our workspace to Visual Studio Code (VS Code) to facilitate easy commits to the existing code using push and pull operations.
- At the start, it is hard to structure our classes in the FOMS while taking into consideration the SOLID principles. As such, we aimed to code the program first, gradually made adjustments and improvements to the existing code before implementing the SOLID principle.
- For example, when we faced uncertainty in adhering to the SRP, we came up with multiple classes and interfaces to distribute the responsibilities so that one class only has one responsibility.
- Preservation of Global System State: We addressed the challenge of maintaining crucial system states (e.g., order status updates, menu modifications) across user sessions by making the code update the datafiles to retain information.
- User Interface Discrepancies: We instituted an interface and hierarchical access restrictions to harmonise displays and streamline user interactions.

5.2 Knowledge Learnt from Course

- Practical application of the OOP concepts we have learned in the course.
- Incorporating the SOLID design principles to improve code maintainability and scalability.
- Gain familiarity with using Java as an object-oriented programming language to develop an OO application.

5.3 Areas of Improvement/ Other Design Considerations

- Employing more interfaces would further promote code flexibility and a greater degree of decoupling between implementations and specific types, facilitating multiple classes to implement common behaviours (Polymorphism).
- Creating a single DataStore class with HashMaps pointing to each datalist instead of separate databases. This would streamline access to essential data elements, enhancing system performance and responsiveness, particularly for frequently accessed key-value pairs if the datalist were bigger, enhancing the efficiency of data retrieval and manipulation processes within the database operations.
- Include more specific exception handling to tackle more specific exception cases. For example, having an exception catch if gibberish were to be typed as the name of a new branch.