

**NANYANG
TECHNOLOGICAL
UNIVERSITY**
SINGAPORE

SC2002 OBJECT ORIENTED DESIGN & PROGRAMMING

AY 23/34 Semester 2 | FDAC, GROUP 1

FAST FOOD ORDERING MANAGEMENT SYSTEMS

REPORT

Contributors:

Name
Aw Yong Wing Kian, Alvin
Chan Zi Hao
Lee Jedidiah
Siah Yee Long
Koh Huei Shan, Winnie

Project Test Cases:

https://drive.google.com/drive/folders/1k4YUUdaOqtFFQxj8fvv7UxkKE2ijIDOT?usp=drive_link






Project Repository:

<https://github.com/siahyeelong/2002FOMS.git>

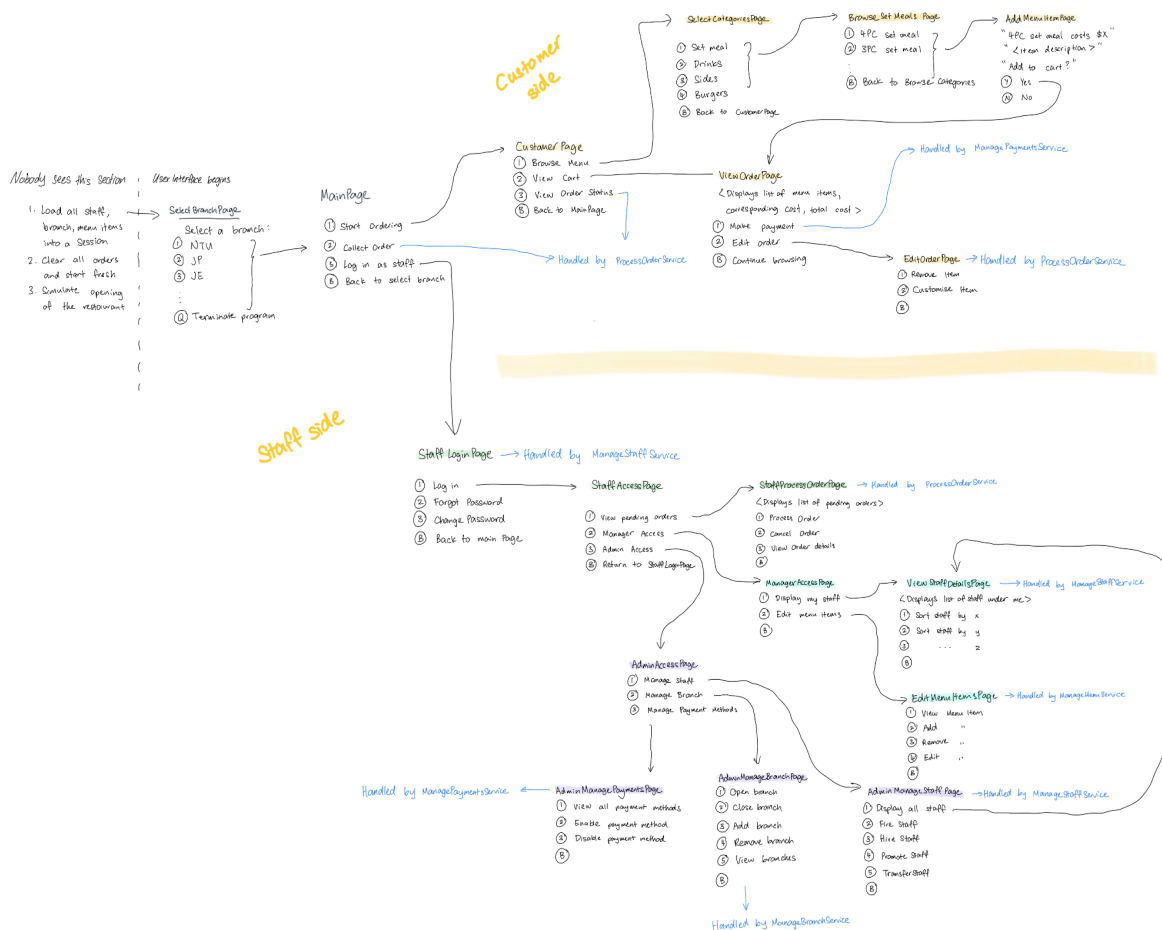
Declaration of Original Work for CE/CZ2002 Assignment	3
Application Overview	4
Design Considerations	4
Approach:	4
Object-oriented concepts adopted:	5
SOLID design principles adopted:	5
Other design considerations:	7
Assumptions	7
Technological Assumptions:	7
Design Assumptions:	8
Operation Assumptions:	8
Detailed UML Class Diagram	9
Test cases performance:	10
Reflection	12

Declaration of Original Work for CE/CZ2002 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 (CE/ CZ2002)	Lab Group	Signature /Date
Aw Yong Wing Kian, Alvin	SC2002	FDAC	 25 th April 2024
Chan Zi Hao	SC2002	FDAC	 25/4/24
Lee Jedidiah	SC2002	FDAC	 25/04/24
Sial Yee Long	SC2002	FDAC	 25 April/2024
Koh Huei Shan, Winnie	SC2002	FDAC	 26/4/24

Application Overview



Design Considerations

Approach:

Our Fast Food Ordering Management Systems (FOMS) has been designed using a modular approach compartmentalising functions into well-defined directories such as entities, services, pages and utilities. The design is organised into a layered architecture resembling a Model View Controller (MVC) where:

- **Entities** represent the Model layer which handles the data and business logic.
- **Pages** provide a View layer which is responsible for the user interface components and interactions.
- **Services** resemble the Controller layers which handle and process the business logic and user requests.

Designing with ease of user navigation in mind, we have mapped out how a user would interact with our programme in <FOMS overview.pdf>

Object-oriented concepts adopted:

- **Abstraction:** Abstraction is present throughout our programme as it is the main part of object-oriented design. Interfaces and abstract classes were used to abstract key portions of functionality within classes. Well-abstracted classes will mean easier code maintainability, feature addition, and being less-prone to errors.
- **Encapsulation:** Encapsulation allows for the proliferation of abstraction throughout the programme. Key attributes of classes are hidden away from other classes and only accessible through getters and setters. Private methods are also used to ensure the overall code is neat and easy to read and hence maintain.
- **Inheritance:** Inheritance was used to enable polymorphism, and to extend functionality beyond its parent class. Overriding of parent attributes also allows the child class to be unique in identity while remaining similar in functionality.
- **Polymorphism:** Polymorphism can be an enabler for extensibility, and is key to satisfy the Open-Closed Principle.

SOLID design principles adopted:

- **Single Responsibility Principle (SRP):** Each class is designed to handle specific responsibilities. An instance on our code would be “LoginDetail” which is dedicated to only providing the login details to the users.

```
public class Staff {  
    private String firstName;  
    private String lastName;  
    private String loginID;  
    private Role role;  
    private boolean gender;  
    private int age;  
    private LoginDetail loginDetail;  
    private Branch branch;  
  
    public LoginDetail(String userID, String initialPassword){  
        this.userID = userID;  
        this.userPassword = initialPassword;  
        if("password" == this.userPassword) this.defaultPassword = true;  
        else this.defaultPassword = false;  
        this.lastLogin = "Never";  
    }  
}
```

- **Open-Closed Principle (OCP):** Utilising interfaces provide us, the developers, the freedom to extend features while remaining closed to modification. This means we can add new classes without changing the code significantly (if at all). An instance will be the iPaymentMethods interface, where new payment methods can be added simply by implementing the interface.

```

public void pay(int customerID, double amount) throws TransactionFailedException{
    if(this.isEnabled){
        try{
            askForCash(amount);
            this.transactionHist.add(new PaymentDetails(customerID, amount, this));
            System.out.println(Logger.ANSI_GREEN+"Payment via " + this.getPaymentTypeName() + " successful."+Logger.ANSI_RESET);
        }
        catch (TransactionFailedException e){
            throw e;
        }
    }
    else
        throw new PaymentServiceDisabledException(Logger.ANSI_RED+this.getPaymentTypeName() + " is currently disabled!" +Logger.ANSI_RESET);
}

```

(cash payment has its dedicated implementation)

- **Liskov Substitution Principle (LSP):** By having an abstract class LoadData with it's inherited child classes LoadStaff, LoadBranch, LoadMenuItem, we ensured the LSP is not violated by keeping arguments, return values, and exceptions thrown in a predictable manner. This allows the substitution of classes where needed without breaking functionality of the program.
- **Interface Segregation Principle (ISP):** This project contains interfaces that are focused in objective, and do not contain methods that will not be used. Examples of interfaces used include iPaymentService, iPage, and iLoginService.
- **Dependency Inversion Principle (DIP):** We designed our modules with abstraction in mind, allowing for better maintainability of our system. High-level modules will not handle low-level jobs, and all low-level jobs are handled through an abstraction layer. In UserInputHelper, user input with its corresponding exceptions are handled within that class, freeing up the need of writing for-loops to iterate through the selection options for Branches every time the user has to select it.

```

public static void removeBranch(Session session) {
    Branch badBranch = UserInputHelper.chooseBranch(session.getAllBranches());
    if(badBranch == null) return;
    if(LoadBranches.removeBranch(badBranch)){
        for(Staff unluckyStaff : session.getAllStaffs()){
            if(unluckyStaff.getBranch() == badBranch){
                // when branch gets deleted, all existing staff just gets displaced into oblivion
                Branch undefinedBranch = new Branch(branchName:"UNDEFINED", location:"beside the dustbin", branchQuota:999, status:"Open");
                unluckyStaff.setBranch(undefinedBranch);
            }
        }
        session.getAllBranches().remove(badBranch);
        System.out.println(x:"Removal of branch successful. All affected staff are gonna starve because of YOU");
        return;
    }
    else{
        System.out.println(x:"Removal of branch failed. phew...");
    }
}

```

(chooseBranch handles the choosing of branch for other methods)

```

public static Branch chooseBranch(ArrayList<Branch> branches) {
    int numOptions = 1;
    for(Branch b : branches){
        System.out.println(numOptions + ". " + b.getBranchName());
        numOptions++;
    }
    int option = getUserChoice(prompt:"Select a branch", numOptions-1);
    if (option == -1) return null;
    return branches.get(option-1);
}

```

Other design considerations:

Session: For every new session of the FOMS run by a user, the current context has to remain consistent throughout the active running session. For example when a staff logs in to NTU branch, the session keeps track of which branch the user is currently at and who the identity of the user is in order to make decisions like providing staff / manager / admin access appropriately. These parameters may change throughout 1 session, but has to remain consistent when changed. For example once a staff logs out, and a customer is now using the FOMS, the session keeps track of the current branch and order objects of this customer.

Data persistence: When multiple instances of the FOMS application is run, the data between the instances must remain persistent. E.g. when a customer places an order in instance 1, and a staff processes the order in instance 2, instance 1 will get updates to show that the order is now ready for pick up. To implement this, we updated the session's information whenever there is an update detected on the csv data files.

Assumptions

Technological Assumptions:

1. Java Runtime Environment: The platform application assumes the Java environment availability for execution which include reliance on Java libraries and the Java Virtual Machine (JVM) for platform independent operation.
2. File-Based Data Management: Our data management is utilising CSV files which assumed these files are accessible and properly formatted. Since we are not using a database, it might affect scalability and performance if it require a huge datasets.
3. Consistent Data Format: The CSV data is assumed to follow a specific schema such as the data being presented in the rows and columns which is important when parsing or writing operations to avoid data corruption or runtime errors.

Design Assumptions:

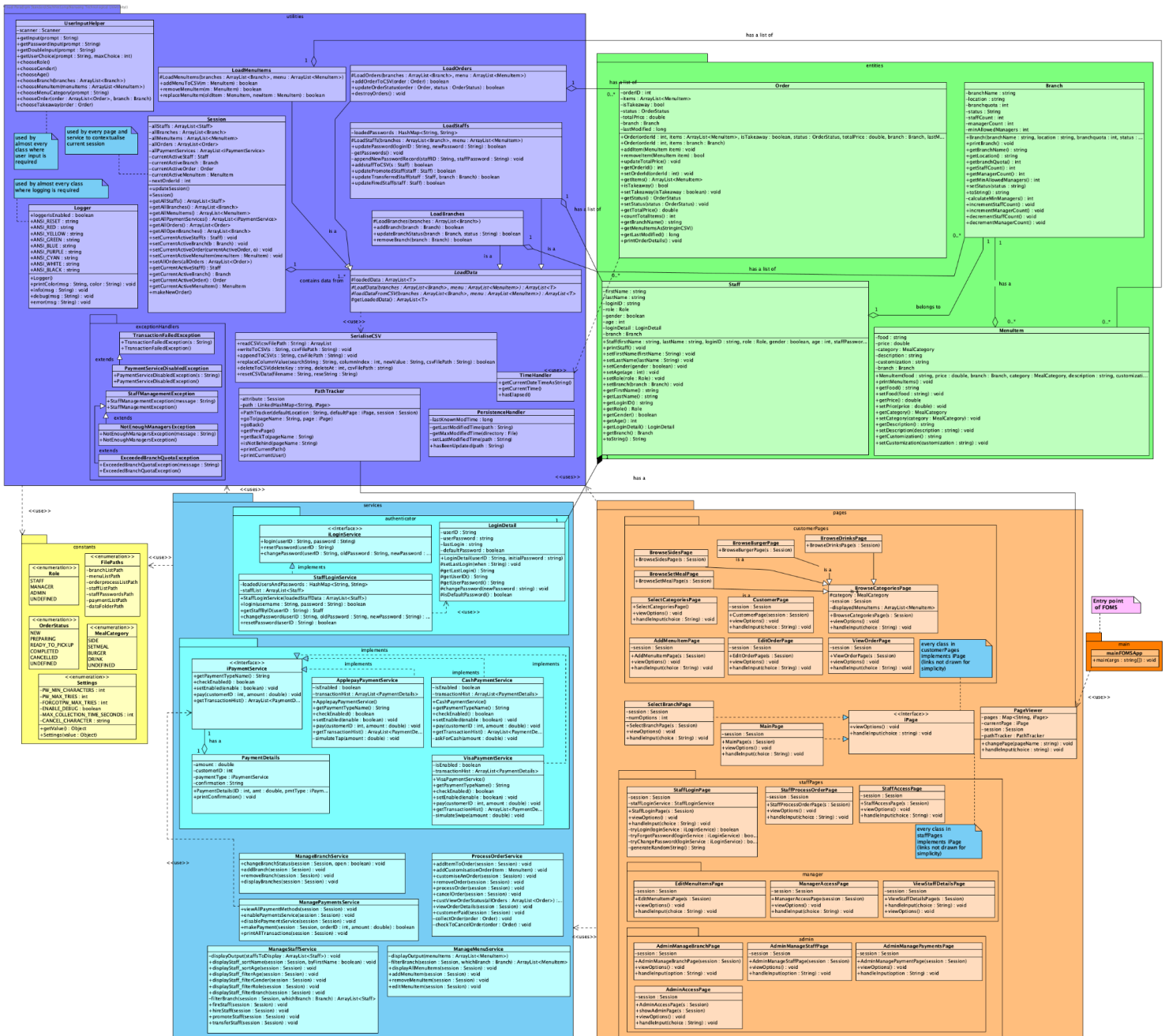
1. Exception Handling: The services package assumes that all critical errors can be managed locally within service methods. An instance will be file not found exceptions and general I/O exceptions are handled without escalating. It is assumed that these do not impact the overall system stability.
2. Error Recovery: System takes on the assumption that writing operations to a file will either succeed or fail in a predictable manner that can be handled by simple retry logic or warning user prompts.

Operation Assumptions:

1. Admins are the highest in rank in the fast food chain. While they may belong to a branch, they can access any branch, and can do whatever a staff and a manager can do. They are not added into the staff count when comparing the branch's staff quota.
2. Adding of a new payment method is done by the developers of the FOMS. As each new payment method will have their distinct way of transacting payments, the feature of extensibility in this project with OCP will be exploited.
3. An admin cannot delete or close a branch while there are still customers ordering.

Detailed UML Class Diagram

(Please view .png for higher resolution image)



Test cases performance:

Test Cases	Test Details	Expected Outcome	Success /Fail
Manager's Action: Menu Management			
1.	Manager add a new menu item with a unique name, price, description, and category.	Customer/staff verify that the menu item is successfully added and reflected.	Pass
2.	Manager update the price and description of an existing menu item.	Customer/staff verify that the changes are reflected in the menu.	Pass
3.	Manager remove an existing menu item.	Customer/staff verify that the menu item is not displayed.	Pass
Order Processing			
4.	Customer place a new order with multiple food items, customize some items, and choose takeaway option.	Manager/staff verify that the order is created successfully and able to process order.	Pass
5.	Customer place a new order with dine-in option.	Manager/staff verify that the order is created with the correct preferences and able to process the order.	Pass
Payment Integration			
6.	Customer make a payment using a credit/debit card (Visa).	Display payment is processed successfully.	Pass
7.	Customer Simulate a payment using an online payment platform (e.g., PayPal).	Display payment is processed successfully.	Pass
Order Tracking			
8.	Customer track the status of an existing order using the order ID placed through the recent order.	Customer verify that the correct status of (preparing as not processed yet) is displayed.	Pass
Staff Actions			
9.	Login as a staff member and display new orders.	Display all the new orders in the branch that the staff is working.	Pass
10.	Manager/staff process a new order which update it's status to "Ready	All user types able to view the order status is updated correctly.	Pass

	to pickup.”		
Manager Actions			
11.	Login as a manager and display the staff list in the manager’s branch.	Manager verify that the staff list is correctly displayed per the branch.	Pass
12.	A manager should be able to perform test case 9.	Verify that manager can see all the new orders in the branch he/she is working.	Pass
Admin Actions			
13.	Admin close a branch	Customer verify by ensuring branch is not displayed on Customer’s interface.	Pass
14.	Login as an admin and display the staff list with the corresponding filters: Branch, Role, Gender and Age.	Display the staff list correctly filtered per the applied filters.	Pass
15.	Admin assign managers to branches with specified quota/ratio	Admin verify that the managers are assigned correctly per the constraint.	Pass
16.	Admin promote a staff to branch manager.	Admin verify that the staff is promoted successfully.	Pass
17.	Admin transfer a staff/manager among branches	Admin verify that the transfer is reflected in the system.	Pass
Customer Interface			
18.	Customer place a new order, check the order status using the order ID, and collect the food.	Customer verify that the order status changes from "Ready to pickup" to "completed."	Pass
Error Handling			
19.	Manager/admin attempt to add a menu item with a duplicate name.	Display an appropriate error message.	Pass
20.	Manager/staff attempt to process an order without selecting any items.	Display an error message prompts the user to select items.	Pass
Extensibility			
21.	Admin add a new payment	Customer able to use new payment method.	Pass

22.	Admin open a new branch	Customer able to see and dine at new branch reflected.	Pass
Order Cancellation			
23.	Customer place a new order and let it remain uncollected beyond the specified timeframe (1 minute).	Customer not being able to pick up order.	Pass
Login System			
24.	User attempt to log in with incorrect credentials as a staff member.	Display an appropriate error message.	Pass
25.	Log in as a staff member, change the default password, and log in again with the new password.	Staff able to login with newly changed password.	Pass
Staff List Initialization			
26.	Upload a staff list file during system initialization.	Verify that the staff list is correctly initialized based on the uploaded file.	Pass
Data Persistence			
27.	Perform multiple sessions of the application, adding, updating, and removing menu items.	Verify that changes made in one session persist and are visible in subsequent sessions.	Pass

Reflection

Reflecting on the initial stages of the FOMS project, the overwhelming complexity of the system was a significant challenge. Initially, our team faced difficulties in grasping the entirety of the application, which led us to adopt a divide-and-conquer strategy. We identified key stakeholders—staff, managers, administrators, and customers—and focused on developing segments catering specifically to each group's needs. This approach allowed us to work efficiently on distinct components before integrating them into a cohesive system.

During the development process, we encountered issues with our initial understanding and implementation of the SOLID design principles. Our first iterations were functional but lacked adherence to these crucial architectural guidelines. This realisation prompted us to revisit our lecture materials, refine our code, and revise our UML diagrams, where we corrected several misrepresentations in the relationships and dependencies.

While UML diagrams serve as an aid for better software design, we realised that an effective workflow is not to design a complete UML first before coding, but rather a synergy of doing both at the same time. Writing code before UML diagrams allows for a more natural flow of ideas, while drawing diagrams before coding allows for clarity and allows us to orient ourselves better.

Among the key takeaways from this project were the importance of effective debugging, enhancing our proficiency in Java, a deeper understanding of SOLID principles, and the crucial skill of designing accurate UML diagrams. Additionally, the project underscored the value of teamwork in navigating complex software development landscapes. Through effective task delegation, our divide-and-conquer approach facilitated a streamline distribution of the workload, adeptly simplifying intricate problems into manageable tasks. Each team member's contribution was vital in transforming initial challenges into a robust and operational system. This experience has not only honed our technical skills but also reinforced the importance of collaboration and structured problem-solving in software development.