






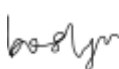

SC2002: Object-Oriented Design & Programming
Building an OO Application: Fastfood Ordering & Management System (FOMS)
FDAA Group 1

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	Lab Group	Signature / Date
Chan Kit Ho	SC2002	FDAA	 25/04/24
Chiang Qin Zhi	SC2002	FDAA	 25/04/24
Lim Yu Kang	SC2002	FDAA	 25/04/2024
Pang Boslyn	SC2002	FDAA	 25/04/2024
Tan Yue Hui	SC2002	FDAA	 25/04/24

1. Design Considerations

1.1 Approach Overview

The development of the FastFood Ordering Management System (FOMS) is predicated on the premise that initial data, stored in files, adheres to a standard format and requires minimal validation upon import. This allows for a straightforward system initialization from pre-defined datasets. The architecture of FOMS is centralised around the InMemoryDatabase class, which utilises hash maps for fast constant-time performance for getting and setting entries, assuming an optimised hashing function is used, so as to easily manage and store all our data. This enables us to tie values such as instances of different classes representing entities such as Staff, BranchManager, Admin, and Account, to keys such as unique staff IDs.

The system's state is transient and maintained only in memory during runtime, with serialisation employed to save the current state to a file upon the conclusion of a session, which can be initiated by a customer or an employee at logout. This approach allows for quick access and modification of data during operation while ensuring data persistence across sessions without the need for a constant database connection or more complex database management systems.

1.2 Assumptions

Data Integrity

It is assumed that the data provided for initial system loading is well-formed and consistent, following the prescribed format to ensure seamless integration into the system.

Controlled Updates

Post-initialization, any updates made to the data files are assumed to be controlled and formatted correctly, as they are managed by the system's functions designed to maintain data integrity.

System Usage

It is assumed that the users of the system, both customers and employees, interact with it through predefined interfaces, ensuring that all interactions are predictable and manageable by the system.

Operational Environment

A stable operational environment is assumed, where standard input/output devices such as consoles are available, and necessary permissions for file read/write operations are granted.

1.3 Application of Object-Oriented Programming Concepts

Abstraction

This concept involves the hiding of complex implementation details and exposing only the necessary parts of the code. In this project, abstraction is realised through the use of interfaces and abstract classes to define what operations can be performed, without specifying how these operations are internally carried out. For instance, the IAdminManagement interface declares methods related to administrative tasks without specifying how these tasks are executed, allowing for detailed implementations in the Admin class. The same can be said for the IMenuManagement and IOrderProcess interfaces implemented by the Branch and Staff classes respectively. Similarly, the abstract Payment class mainly declares the processPayment() method while its complex implementation details are left to the PayPal, PayNow, and BankCard classes.

Encapsulation

This principle is about bundling data with methods that operate on the data and restricting direct access to some of an object's components. Encapsulation ensures that the internal representation of the object is hidden from the outside. For example, the vast majority of all the attributes of all our classes have their visibility set to private, and we have included public getters and setters within these classes only as necessary to ensure that the state of these objects can only be changed in purposeful and controlled ways.

Polymorphism

This allows objects of different classes to be treated as objects of a common super class. It enables a single interface or abstract class to represent different underlying forms allowing for flexible and scalable code. In this project, an example of how polymorphism is utilised is with the different types of employees (Staff, BranchManager, Admin) that can be managed through a common superclass and various different interfaces. The method getBranchName() displays polymorphic behaviour by allowing for different Employee subclasses, as Admins are not associated to a specific branch while the Staff and BranchManagers are. Methods such as getBranchMenu() and getBranchOrders() also display polymorphic behaviour. The

abstract “Payment” class is also what allows for various types of payment subclasses (“Bank Card”, “PayNow” and “PayPa”l) to be implemented with their own processes, and for future payment types to be added easily.

Inheritance

It allows new classes to take on the properties of existing classes. A common parent class like “Employee” can pass on its attributes and methods to child classes like “Staff”, “BranchManager”, and “Admin”. This reduces redundancy in code and allows for enhancements to be made in a single place in the codebase.

Extensibility and Maintainability

The OOP concepts mentioned above ensure a system's extensibility and maintainability by allowing new features to be added with minimal changes to the existing code. Abstraction allows for new functionalities to be added without affecting other parts of the system. Encapsulation means that objects control their own data and behaviour, reducing the likelihood of bugs. Polymorphism and inheritance mean that new types of objects can be added to the system with little or no modification to existing code.

For example, if a new type of employee needs to be added to the system, it can inherit from the “Employee” superclass, necessitating only the unique attributes and behaviours to be defined, while the functions of the employee can be customised through their implementations of the many employee-related interfaces included in our codebase.

As mentioned previously, new types of payment methods can be introduced easily by extending the abstract “Payment” class without altering the logic of existing payment types. The implementation of such a concept through the ICustomerOrderProcess interface even allows for different types of customers in the future other than the regular “Customer” we have in our project, such as a “MemberCustomer” class to allow customers on membership to enjoy a discount on their total price.

1.4 Design Principles

Single Responsibility Principle (SRP)

The Single Responsibility Principle states that a class should have only one reason to change, meaning it should only have one job or responsibility. In our project, we illustrated this principle by ensuring each class encapsulates its own distinct functionality:

- “Account” class handles the attributes and behaviours related to a user's account, like login validation and password management. It does not deal with any other entities or broader system functionalities, adhering to SRP.
- “InMemoryDatabase” acts as a storage class, responsible for managing the addition and retrieval of various entities like Staff, Admin, and BranchManager, but it does not deal with the logic of manipulating these entities beyond storage and retrieval.

Open-Closed Principle (OCP)

The Open-Closed Principle suggests that software entities should be open for extension, but closed for modification. This means that we should be able to add new functionality without changing existing code. This design principle has actually already been extensively discussed in our preceding section, Applications of Object-Oriented Programming Concepts.

Liskov Substitution Principle (LSP)

LSP states that objects of a superclass should be replaceable with objects of its subclasses without breaking the application. This implies that subclasses should not change the expected behaviour of the base class.

For instance, subclasses of “Employee” can be used interchangeably without the system behaving unexpectedly. Anywhere an “Employee” is expected, a “Staff”, “BranchManager”, or “Admin” can be substituted in without issue, which means they adhere to LSP. Particularly, as Admin’s are not associated to a specific branch, they still return an empty branch name string to adhere to LSP.

Interface Segregation Principle (ISP)

ISP dictates that no client should be forced to depend on methods it does not use. Interfaces should be specific to client needs rather than one general-purpose interface.

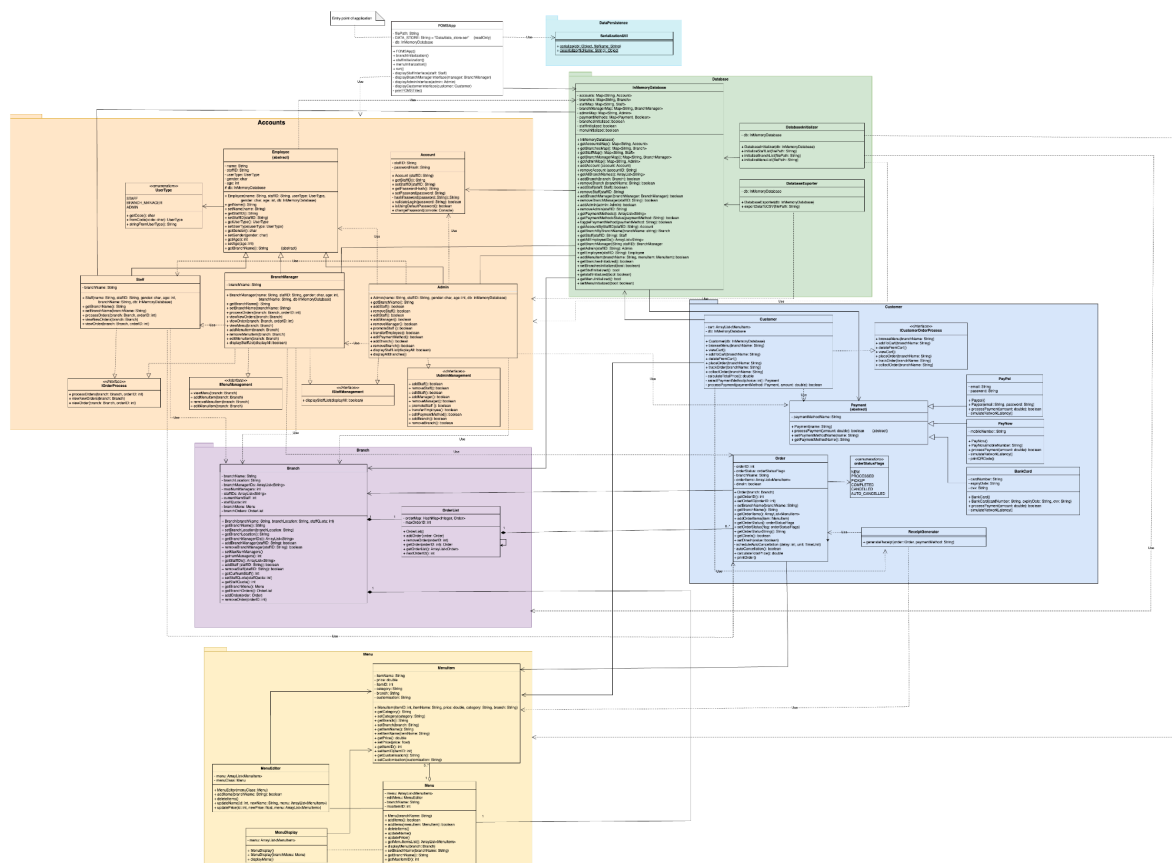
For instance, in our project, interfaces like IAdminManagement and IStaffManagement provide specific methods for different user types, following ISP. Each type of user only interacts with the methods that are relevant to them.

Dependency Inversion Principle (DIP)

There are two parts to DIP: firstly, high-level modules should not depend on low-level modules because both should depend on abstractions, and secondly, abstractions should not depend upon details, because details should depend upon abstractions. For instance, in our project, instead of the high-level (regular) “Customer” class depending on the low-level payment classes of “BankCard”, “PayNow”, and “PayPal”, both depend on abstractions: the payment classes depend on the abstract Payment class which the “Customer” class also depends on, thereby avoiding highly coupled distribution via an abstract intermediary layer.

2. UML Class Diagram

The following UML diagram illustrates the class relationships and dependencies between classes and packages. A higher quality diagram is available in a separate file titled ‘FDDA_Group1_UML’.



3. Testing

The test cases have been organised based on each role. Specifically, actions by customers, staff, managers and admins are sequenced to reflect each user's journey in using FOMS.

3.1 Initialisation & Data Persistence

Test Case No. (in Appendix A of Assignment PDF)	Inputs	Expected Results
26 [Initialisation]	Load files into the system. Branch: branch_list.csv Menu: menu_list.csv Staff: staff_list.csv	Data successfully imported into the system. Branch, menu items and staff list can be displayed.
27 [Data persistence]	Switching between customers and different account types.	Data is serialised after every session.

3.2 Customer's Actions

Test Case No. (in Appendix A of Assignment PDF)	Inputs	Expected Results
4, 6 [Takeaway order + Credit card payment]	Branch: JP Order: Apple Pie, Ice Lemon Tea Payment: PayNow	Order created and can be tracked with order ID (Order 9792).
5, 7 [Dine-in order + Online payment]	Branch: JP Order: Chicken burger, Cheeseburger Customisation: Extra sauce Payment: Bank Card	Order created and can be tracked with order ID (Order 9791).
8 [Track order status]	Order ID of orders made.	Order status is updated.

20 [Empty cart order]	Branch: JE	Error message prompted. Unable to cart out.
18 [Place order, track and pickup]	Place order on customer interface and wait for staff to update order status from NEW to PROCESSED to PICKUP. Indicate on the system that order is collected.	Order created and status is updated whenever there is a new update from staff. Once order is collected, order status would change to COMPLETED.
23 [Place order, pickup time lapse]	OrderID: 9791	Order created and status is updated by staff to PICKUP. After 15 seconds, the order status updates to AUTO_CANCELLED as the customer failed to pick up on time.

3.3 Staff's Actions

Test Case No. (in Appendix A of Assignment PDF)	Inputs	Expected Results
9, 25 [Login, change password & display new orders]	Login as JustinL, and change password.	Successful login with prompt to change password. Passwords cannot be reused. New orders of JP are displayed.
10 [Order processing]	Change order status to PICKUP.	New order status reflected on the customer's end.
24 [Login Error]	1. Login with wrong username	Error message displayed, staff is prompted to attempt to login

	2. Login with wrong password	again
28 [Pickup time lapse]	Update order status to PICKUP.	Order status updated and will be automatically changed to AUTO_CANCELLED in 15 seconds if the customer fails to pick up on time.

3.4 Manager's Actions

Test Case No. (in Appendix A of Assignment PDF)	Inputs	Expected Results
11 [Login & display staff list]	staffID: SteKK Default password: password	Successful login with prompt to change password. Staff list of JP branch displayed.
3 [Remove menu item]	Remove Coffee (item 14) from the JP branch.	Coffee is no longer in the branch menu browsed by customers.
1 [Add new menu item]	Name: Hashbrown Price: \$2.00 Category: Side Branch: JP	Item successfully added and is reflected when customers browse the menu of the JP branch.
2 [Update menu item details]	Branch: JP Item: Cheeseburger (item 6) Price: \$3.00	Change is reflected when customers browse the menu of the JP branch.
12 [Order processing]	Change order status to PICKUP.	New order status reflected on the customer's end.

19 [Adding duplicate menu item]	Menu Item: Hashbrown Branch: JP	Error message displayed due to duplicate menu item.
-----------------------------------	------------------------------------	---

3.5 Admin's Actions

Test Case No. (in Appendix A of Assignment PDF)	Inputs	Expected Results
14 [Login & display staff list]	Login as boss and use a filter for staff display. Branch: JE Role: Manager Gender: Female Age: 22	Successful login with prompt to change password. Staff list is displayed based on the different filters applied.
15 [Assign Managers]	Branch JP requires an additional manager. Add manager: Branch: JP Name: James Lee staffID: JamesL Gender: Male Age: 32	Manager is successfully added and adheres to quota constraints.
16 [Promote staff to manager]	StaffID: JustinL	Change of role of JustinL is reflected in the staff list.
17 [Transfer staff]	Transfer staff from JP to NTU StaffID: Jhn2	Change is reflected in the branch of staff.
21 [Delete payment method]	Payment method: PayPal	Customers are able to select PayNow as payment mode.
13 [Close a branch]	Branch: NTU	Branch is not displayed for

		customer selection and staff has been removed from the master list.
22 [Open new Branch]	Name: JL Location: Jurong Lake Staff quota: 15	New branch is created and is displayed for customer's selection.
Export staff list for administrative purposes	-	Updated staff list will be exported as a csv file.

4. Reflection

4.1 Difficulties Encountered

The development of the FastFood Ordering Management System (FOMS) was not as straightforward as it seemed, and there were various challenges we had to overcome.

One of the main challenges our team faced was reaching an agreement on the design architecture. We encountered debates regarding which types of classes to use, what data structures would be most suitable, and how these classes would interact with each other. These discussions often led to varying approaches, which potentially led to inefficiency in our development process. However, when our team used UML class diagrams as a tool to visualise, we realised that we were able to articulate our design ideas visually, and it helped facilitate communication and collaboration within the team. Furthermore, it enabled us to anticipate potential challenges and dependencies early in our development process. Overall, UML helped simplify complex ideas into clear and concise representations, making it easier for us to understand and discuss various aspects of our project.

Another notable challenge we encountered was adhering to the different Object-Oriented Programming Concepts and SOLID design principles. Though remembering and applying them to use was difficult at first, it enabled us to build a modular and scalable codebase that is easily maintainable and extensible. This increased our understanding of the real-world implementation of the aforementioned design principles and inculcated proper design habits

and patterns. This will be especially useful for the project in our upcoming SC2006 Software Engineering module next semester.

Lastly, we had to face the challenge of exception handling. Securing the robustness of our system required comprehensive exception handling to manage runtime errors, instances of which include `IOExceptions` during file operations due to unexpected data absence, and `InputMismatchExceptions` due to the system's reliance on user input. Ensuring that we had all our bases covered was a lesson in being detail oriented.

4.2 Future Developments

For future applications, we are excited to explore how we can make the system even more user-friendly and interactive by integrating our code into mobile and web-based platforms via a web app. We can make a multi-platform web app through the use of front-end tools such as React Native. Additionally, we would love to explore and implement automated testing to enhance the efficiency of our testing process. Moreover, if we are moving towards an online application, we must understand how to employ stronger and more advanced encryption standards beyond just the password hashing we have implemented, such as password hash salting or implementing the TLS protocol for over Internet communications.

In conclusion, the development of the FOMS has been an invaluable experience that we will continue to look upon with great pride and nostalgia. Little did we know as we embarked on this project that it would not only become a big first step in our OO programming experience, but that the many indispensable OOP concepts and design principles we practised and implemented would hallmark our software engineering careers.