Nanyang Technological University
School of Computing Science and Engineering

SC2002 Project Report
Fast Food Order Management System

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 honored 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
Denzel Elden Wijaya (U2320123A)	SC2002	SCE2	[illegible] 26/4/2024
Federrico Hansen Budianto (U2320343B)	SC2002	SCE2	[illegible] 26/4/2024
Melisa Lee (U2320732A)	SC2002	SCE2	[illegible] 26/4/2024
Rivaldo Billy Sebastian (U2040150J)	SC2002	SCE2	[illegible] 26/4/2024

1 Design Considerations

1.1 Approach

FOMS is a Java Command Line Interface (CLI) application. The application is aimed at streamlining the process of ordering, payment, and order management for both customers and staff at Fast-food restaurants. The system is expected to facilitate staff management, menu browsing, order customization, and payment processing.

The application employs the entity, controller, and boundary (ECB) class stereotypes. Users interact with the app via menus, represented as boundary classes. These menus request data from managers, functioning as controller classes. Managers handle logic implementation, error checking, and exception handling to ensure smooth app operation. Data is retrieved from entity classes, responsible for data storage. These abstractions facilitate straightforward adjustments and expansions.

1.2 Assumptions

Assumptions to be made on our FOMS application.

1. When editing the Excel files, we need to commit first for them to be used in subsequent steps.
2. The gender of a staff member cannot be changed.
3. The `userID` will be set to `100` as the default starting point. It will then be incremented after subsequent orders.
4. The password for all user will be set to `password` as default.
5. Even though our payment system is easily extensible, no staff member can add other payments here.
6. The addition of a menu will be automatically incremented as the bottommost item in the Excel file.

1.3 SOLID Design Principles

SOLID represents the initial five principles of object-oriented design (OOD) formulated. These principles provide guidelines for constructing software that can be maintained and expanded as the project evolves. Embracing these principles not only aids in circumventing code deficiencies but also facilitates code refactoring and the creation of adaptable applications. The SOLID principles are:

- Single-responsibility Principle
- Open-closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

1.3.1 Single Responsibility Principle

The single responsibility principle (SRP) ensures that a class is never overloaded with responsibilities. Each class should do at most one operation and serve one purpose. This is to ensure that each class is as small as possible, making the code not only easier to read and understand, but also more maintainable.

One implementation in our code is on `MenuUI` class. This class used by these two classes `CustomerMainPage` which is the page for customer and `StaffLogin` for staff. Also for staff, these three `ManagerMainPage`, `StaffMainPage`, and `AdminMainPage` have a dependency relation with `StaffLogin` class. These classes serves only one specific nature task. The tests ensure that our login systems will always work across various changes to the code base.

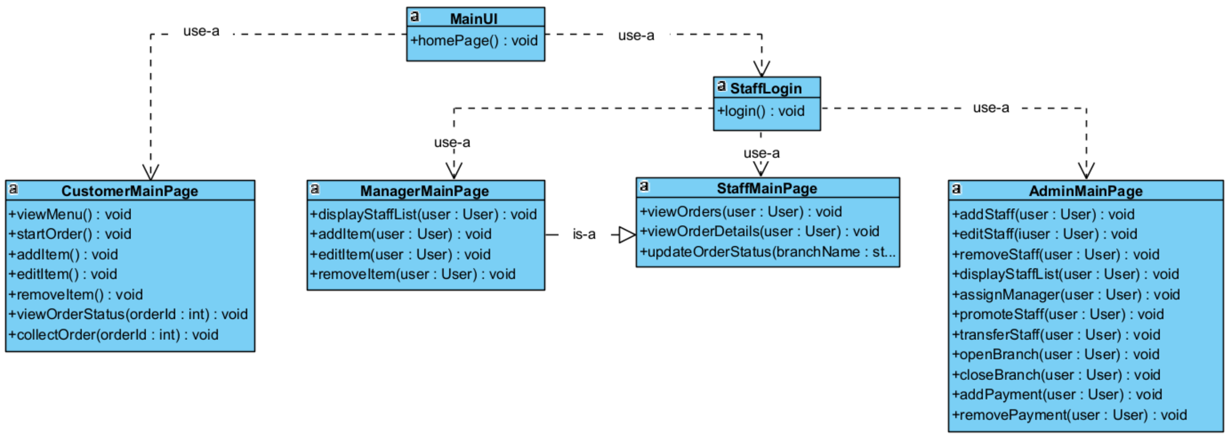

MainUI
+homePage() : void
use-a
use-a
StaffLogin
+login() : void
use-a
use-a
use-a
CustomerMainPage
+viewMenu() : void
+startOrder() : void
+addItem() : void
+editItem() : void
+removeItem() : void
+viewOrderStatus(orderId : int) : void
+collectOrder(orderId : int) : void
ManagerMainPage
+displayStaffList(user : User) : void
+addItem(user : User) : void
+editItem(user : User) : void
+removeItem(user : User) : void
is-a
StaffMainPage
+viewOrders(user : User) : void
+viewOrderDetails(user : User) : void
+updateOrderStatus(branchName : st...
AdminMainPage
+addStaff(user : User) : void
+editStaff(iuser : User) : void
+removeStaff(user : User) : void
+displayStaffList(user : User) : void
+assignManager(user : User) : void
+promoteStaff(user : User) : void
+transferStaff(user : User) : void
+openBranch(user : User) : void
+closeBranch(user : User) : void
+addPayment(user : User) : void
+removePayment(user : User) : void

Figure 1: `MenuUI` and its dependencies

1.3.2 Open-Closed Principle

The open-close principle (OCP) states that a class should be close to modification but open to extension. It means that new functionalities should be added without modifying previous classes. This can be done through polymorphism and inheritance. The extension should be able to be added without touching the superclass.

One of our implementation involves the `PaymentController` class. Within this implementation, three classes, `CashController`, `CreditCardController`, and `PayNowController` inherit from PaymentController. This inheritance structure enables the creation of additional classes that inherit from `PaymentController`, facilitating extension. However, it's important to note that the other classes inheriting from `PaymentController` cannot be modified.

1.3.3 Liskov Substitution Principle

The Liskov Substitution Principle asserts that subclasses inheriting from a superclass must be capable of performing all the functionalities of their superclass. This means that objects of the superclass should be substitutable with instances of its subclass without causing any disruptions to the application.

One of the way we implement this is at the `StaffMainPage` and `ManagerMainPage` class. Staff and manager have same class, however, manager enable to do more features than staff. Therefore, we create `ManagerMainPage` class, a subclass of `StaffMainPage`. This ensures that a manager can

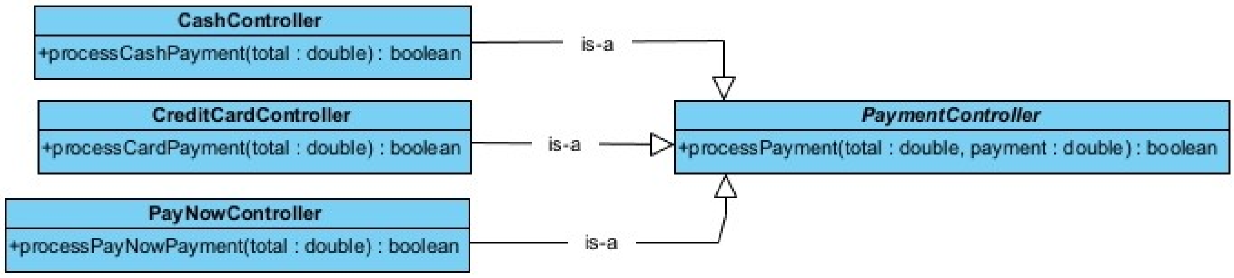

CashController
+processCashPayment(total : double) : boolean
CreditCardController
+processCardPayment(total : double) : boolean
PayNowController
+processPayNowPayment(total : double) : boolean
is-a
is-a
is-a
PaymentController
+processPayment(total : double, payment : double) : boolean

Figure 2: `PaymentController` and its inheritances.

perform all the tasks that a staff member can, without introducing any new issues that the staff member cannot handle.

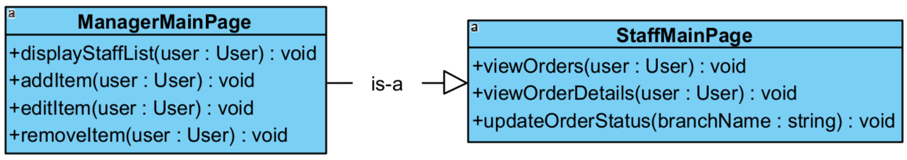

ManagerMainPage
+displayStaffList(user : User) : void
+addItem(user : User) : void
+editItem(user : User) : void
+removeItem(user : User) : void
is-a
StaffMainPage
+viewOrders(user : User) : void
+viewOrderDetails(user : User) : void
+updateOrderStatus(branchName : string) : void

Figure 3: `ManagerMainPage` is a subclass of `StaffMainPage`.

1.3.4 Interface Segregation Principle

Interface segregation dictates that when employing interfaces, each interface should be as concise as possible. It's preferable to have several specific interfaces rather than a single broad interface, and a class should implement multiple interfaces rather than one extensive interface.

One of the implementations we used in the code is on `OrderStatusController` class. This class realizes three interfaces, named `OrderCheckOutInterface`, `OrderCollectionInterface`, and `OrderProcessInterface`, which includes all the methods the class uses. This ensures that we don't rely on methods in interfaces that a class doesn't utilize.

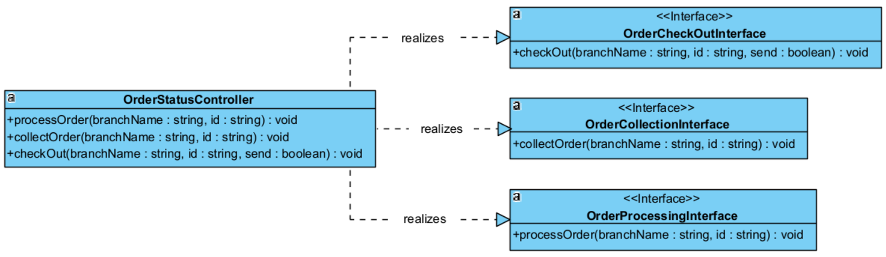

OrderStatusController
+processOrder(branchName : string, id : string) : void
+collectOrder(branchName : string, id : string) : void
+checkOut(branchName : string, id : string, send : boolean) : void
realizes
realizes
realizes
<<Interface>>
OrderCheckOutInterface
+checkOut(branchName : string, id : string, send : boolean) : void
<<Interface>>
OrderCollectionInterface
+collectOrder(branchName : string, id : string) : void
<<Interface>>
OrderProcessingInterface
+processOrder(branchName : string, id : string) : void

Figure 4: `ManagerMainPage` is a subclass of `StaffMainPage`.

1.3.5 Dependency Injection Principle

.

The principle of dependency injection advocates for designing systems where high-level modules are not tightly coupled to low-level ones. Instead of depending on specific concrete classes, they should rely on higher-level interfaces. This approach promotes loose coupling, making the code more flexible and easier to maintain.

In our codebase, we've applied this principle effectively. For instance, consider the viewOrder-Status method, which implements the CustomerViewInterface and StaffViewInterface. By adhering to these interfaces, any future modifications or extensions to the UI view, whether in the customer or staff perspective, can be seamlessly integrated. If there's a need for a new version of the view in the future, implementing these interfaces in a new class provides direct access for modifications without impacting existing code significantly.

This approach not only enhances maintainability but also fosters scalability and adaptability in the system architecture. By relying on interfaces rather than concrete implementations, we ensure that changes to the underlying functionality can be made with minimal disruption to the overall system.

1.4 APIE Design Principles

1.4.1 Abstraction

Implementing abstraction in the `Payment` class provides a flexible foundation for future enhancements. By abstracting the specific payment methods, such as credit cards or PayPal, we create a system that can seamlessly accommodate the integration of new payment methods at a later stage, facilitated by an administrator. This abstraction shields the core functionality of the Payment class from the intricacies of individual payment processes. It essentially establishes a standardized interface, allowing for easy addition of new payment methods without the need for extensive modifications to the existing codebase. In essence, this approach ensures that the Payment class remains adaptable and scalable, ready to embrace emerging payment technologies or preferences in the ever-evolving landscape of digital transactions.

1.4.2 Polymorphism

Polymorphism allows for more generic and flexible code because it enables you to write code that can work with objects of various types without knowing their specific class at compile time. One of the things is that it is extendable, we can add new classes with little or no modification to the existing code. This means that the application can grow over time by adding new function in a new classes that implement existing methods. Another thing is maintainibility, changes to the method implementation in the superclass or interface can be independently extended in subclasses without affecting other subclasses. This isolation of changes minimise the risk of destroying existing code. In our code, Polymorphism is used under the Payment Method, as when we called $processPayment$ method from ths superclass $PaymentController$ to the childClass, such as $cashController$, $CreditCardController$, $PayNowController$.

1.4.3 Inheritance

Inheritance serve as a generelization of a certain class to it's subclasses. It helps to provide the same attributes and methods among the subclasses. This will help identify the hierarchy of the classes. When changes need to be made to common functionality, they can be implemented in just one place: the superclass. This update automatically propagates to all subclasses, which improves code maintainability and reduces the chance of errors. Some of the implementation involves `PayNowController, CashController, and CreditCardController` that inherits from

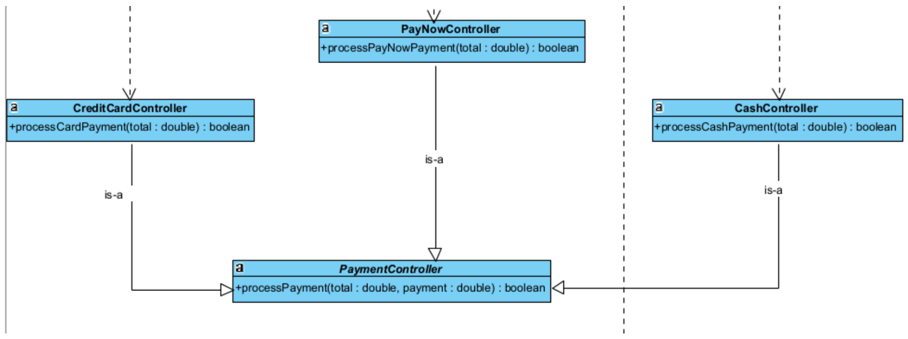
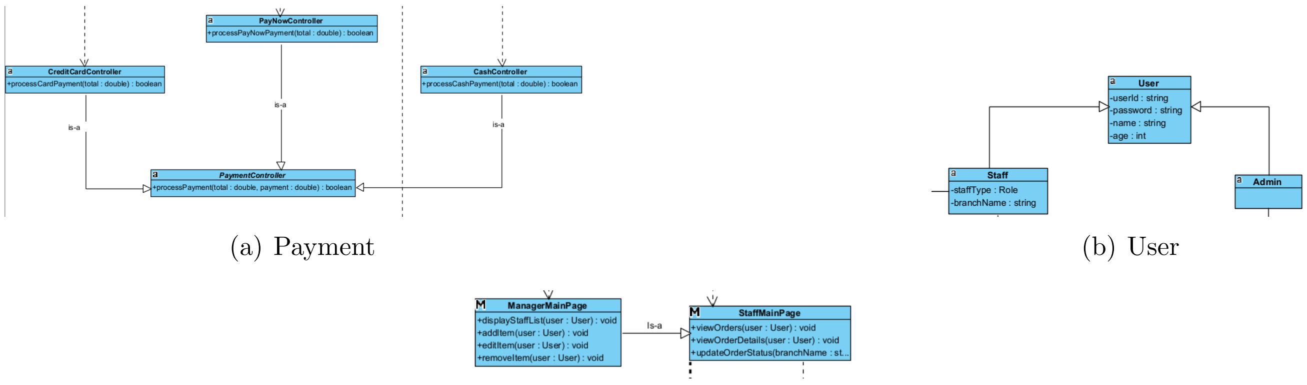

PayNowController
+processPayNowPayment(total : double) : boolean
CreditCardController
+processCardPayment(total : double) : boolean
CashController
+processCashPayment(total : double) : boolean
is-a
is-a
is-a
PaymentController
+processPayment(total : double, payment : double) : boolean

Figure 5: `Polymorphism`

`PaymentController`. Also, `Staff` and `Admin` who inherits from `User` and `ManagerMainPage` inherits from `StaffMainPage`. The payment method uses inheritance to allow it extend the class so that when a new payment method is added, it does not affect the existing system. Inheritance makes it possible to override methods of the superclass in a subclass to provide specific functionality. While the inherited method provides a general procedure, the overridden method in the subclass can refine or replace this procedure with a specific one.


PayNowController
+processPayNowPayment(total : double) : boolean
CreditCardController
+processCardPayment(total : double) : boolean
CashController
+processCashPayment(total : double) : boolean
is-a
PaymentController
+processPayment(total : double, payment : double) : boolean
User
-userId : string
-password : string
-name : string
-age : int
Staff
-staffType : Role
-branchName : string
Admin
ManagerMainPage
+displayStaffList(user : User) : void
+addItem(user : User) : void
+editItem(user : User) : void
+removeItem(user : User) : void
is-a
StaffMainPage
+viewOrders(user : User) : void
+viewOrderDetails(user : User) : void
+updateOrderStatus(branchName : st...

(a) Payment (b) User

(c) Page

1.4.4 Encapsulation and Information Hiding

Encapsulation, a foundational concept in object-oriented programming (OOP), entails concealing an object's internal workings and permitting only designated operations to manipulate it. In the realm of OOP, objects are instances of classes, and encapsulation empowers developers to regulate access to the data and functions linked with those objects. Such encapsulation practices effectively segregate roles and responsibilities, ensuring a robust division of control within the system. This is vital for maintaining integrity and security in operations. This is facilitated by using access modifiers in the programming language, such as private, protected, and public, which help define the scope and accessibility of methods and variables. In our program, encapsulation is implemented within private methods of Admin or Managers, such as adding payments for the admin or displaying staff lists for the manager. For instance, when adding or deleting payments, access is granted to Excel data using ArrayLists, which are kept private and static. Consequently, managers or staff members are barred from performing such actions. Similarly, in the managers' domain, staff members lack the authority to display the staff members in a particular branch.

1.5 Other Implementation

1.5.1 Data Serialization and Deserialization using Apache POI

To extract the data from Microsoft Excel to be read in java file we need an extra library, namely Apache POI. Apache POI is a java library that is used to handle Microsoft Office documents. It allows Java applications to read and write Excel documents, handling both .xls formats via HSSF and .xlsx formats via XSSF. Apache POI excels in reading data from Excel files, enabling access to spreadsheet elements such as sheets, rows, and cells, and accommodating various data types and formulas. Additionally, it supports writing to Excel files, crucial for applications requiring dynamic data output for analysis or reporting. This includes the ability to create and modify sheets, cells, and their contents, along with applying cell styles and formatting.

Serialization in Apache POI involves converting application data into a format suitable for Excel. This process starts with creating a Workbook object, populating it with data, and finally writing it to an OutputStream, effectively turning the workbook into an Excel file. This capability extends to handling advanced Excel features like comments, hyperlinks, pictures, and data validations. By bridging the gap between Java applications and Excel, Apache POI serves as a powerful tool for data interchange and comprehensive reporting, making it indispensable for developers integrating office software capabilities into their applications. Furthermore, we are using

1. org.apache.poi.openxml4j.exceptions.InvalidFormatException;
2. org.apache.poi.ss.usermodel.*
3. org.apache.poi.xssf.usermodel.XSSFWorkbook
4. org.apache.poi.xssf.usermodel.XSSFSheet

.

to extract and write back to the file.

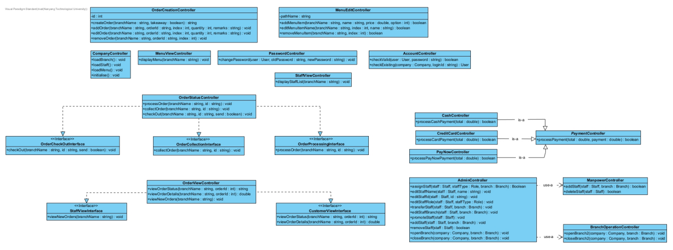
2 Unified Modeling Language

The Unified Modeling Language (UML) is a standardized modeling language that comprises a set of integrated diagrams. It aids system and software developers in specifying, visualizing, constructing, and documenting software system artifacts. UML embodies best engineering practices for modeling large and complex systems and is crucial in object-oriented software development. Using graphical notations, UML facilitates communication, exploration of potential designs, and validation of software architecture. UML aims to offer a universal notation usable across all object-oriented methods, integrating successful elements from previous notations. It caters to a wide range of applications, providing constructs for various systems and activities, including distributed systems, analysis, system design, and deployment.

We implemented Entity-Boundary-Controller (EBC) approach in our application UML. EBC is an architectural pattern used in object-oriented programming (OOP) to organize classes within a system based on their responsibilities in realizing use cases. It typically involves structuring classes into three categories: Entity classes (representing domain objects and data), Boundary classes (representing interactions with external systems or users), and Control classes (containing the application logic).

The EBC pattern is related to the Model-View-Controller (MVC) pattern. Both patterns aim to separate concerns in software architecture, but they do so in different ways. MVC separates an application into three interconnected components: Model, View, and Controller. The Model represents the application's data and business logic, the View represents the presentation layer (user interface), and the Controller acts as an intermediary that handles user input, updates the Model, and updates the View accordingly.

The sections below showcase our Entity, Boundary, Controller UML, and how we combine them into one. For the detailed UML, you may see it on GitHub : SC2002 SCE2 Group 2 Project



Visual Paradigm Standard(rival(Nanyang Technological University))
OrderCreationController
-id : int
+createOrder(branchName : string, takeaway : boolean) : string
+addOrder(branchName : string, orderId : string, index : int, quantity : int, remarks : string) : void
+editOrder(branchName : string, orderId : string, index : int, quantity : int, remarks : string) : void
+removeOrder(branchName : string, orderId : string, index : int) : void
MenuEditController
-pathName : string
+addMenuItem(branchName : string, name : string, price : double, option : int) : boolean
+editMenuItemName(branchName : string, index : int, name : string) : boolean
+removeMenuItem(branchName : string, index : int) : boolean
CompanyController
+loadBranch() : void
+loadStaff() : void
+loadMenu() : void
+initialise() : void
MenuViewController
+displayMenu(branchName : string) : void
PasswordController
+changePassword(user : User, oldPassword : string, newPassword : string) : void
AccountController
+checkValid(user : User, password : string) : boolean
+checkExisting(company : Company, loginId : string) : User
StaffViewController
+displayStaffList(branchName : string) : void
OrderStatusController
+processOrder(branchName : string, id : string) : void
+collectOrder(branchName : string, id : string) : void
+checkOut(branchName : string, id : string, send : boolean) : void
<<Interface>>
OrderCheckOutInterface
+checkOut(branchName : string, id : string, send : boolean) : void
<<Interface>>
OrderCollectionInterface
+collectOrder(branchName : string, id : string) : void
<<Interface>>
OrderProcessingInterface
+processOrder(branchName : string, id : string) : void
CashController
+processCashPayment(total : double) : boolean
CreditCardController
+processCardPayment(total : double) : boolean
PayNowController
+processPayNowPayment(total : double) : boolean
is-a
is-a
is-a
PaymentController
+processPayment(total : double, payment : double) : boolean
OrderViewController
+viewOrderStatus(branchName : string, orderId : int) : string
+viewOrderDetails(branchName : string, orderId : int) : double
+viewNewOrders(branchName : string) : void
<<Interface>>
StaffViewInterface
+viewNewOrders(branchName : string) : void
<<Interface>>
CustomerViewInterface
+viewOrderStatus(branchName : string, orderId : int) : string
+viewOrderDetails(branchName : string, orderId : int) : double
AdminController
+assignStaff(staff : Staff, staffType : Role, branch : Branch) : Boolean
+editStaffName(staff : Staff, name : string) : void
+editStaffId(staff : Staff, id : string) : void
+editStaffRole(staff : Staff, staffType : Role) : void
+transferStaff(staff : Staff, branch : Branch) : void
+editStaffBranch(staff : Staff, branch : Branch) : void
+promoteStaff(staff : Staff) : void
+addStaff(staff : Staff, branch : Branch) : void
+removeStaff(staff : Staff) : boolean
+openBranch(company : Company, branch : Branch) : void
+closeBranch(company : Company, branch : Branch) : void
use-a
use-a
ManpowerController
+addStaff(staff : Staff, branch : Branch) : boolean
+deleteStaff(staff : Staff) : boolean
BranchOperationController
+openBranch2(company : Company, branch : Branch) : void
+closeBranch2(company : Company, branch : Branch) : void

Figure 6: Controller UML

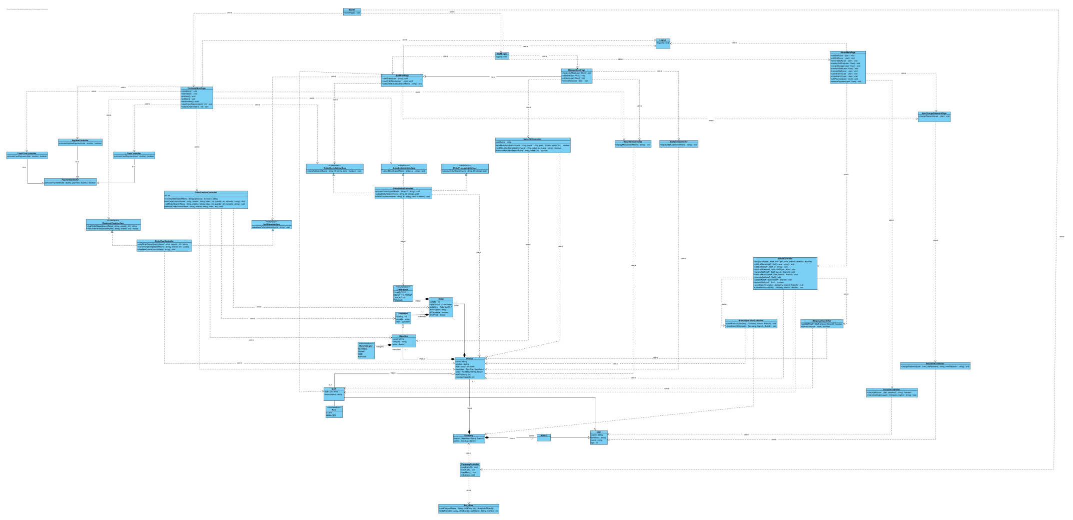
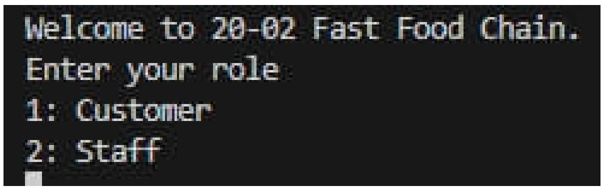


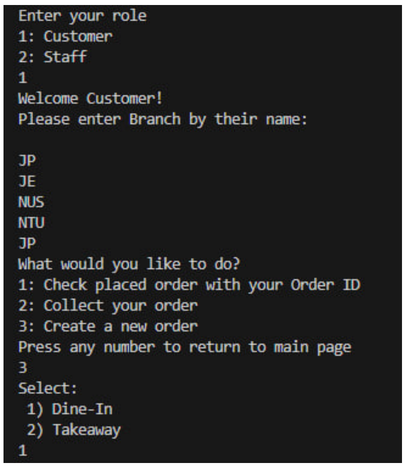
Figure 7: Whole UML

3 Testing



```
Welcome to 20-02 Fast Food Chain.
Enter your role
1: Customer
2: Staff
```

(a) early interface

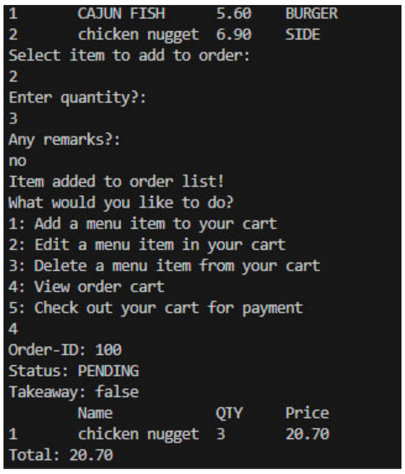


```
Enter your role
1: Customer
2: Staff
1
Welcome Customer!
Please enter Branch by their name:

JP
JE
NUS
NTU
JP
What would you like to do?
1: Check placed order with your Order ID
2: Collect your order
3: Create a new order
Press any number to return to main page
3
Select:
 1) Dine-In
 2) Takeaway
1
```

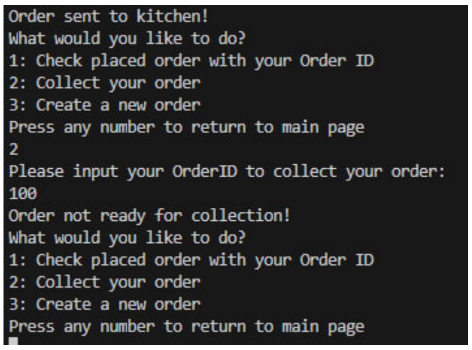
(b) customer ordering

Figure 8: Login page



```
1       CAJUN FISH      5.60    BURGER
2       chicken nugget  6.90    SIDE
Select item to add to order:
2
Enter quantity?:
3
Any remarks?:
no
Item added to order list!
What would you like to do?
1: Add a menu item to your cart
2: Edit a menu item in your cart
3: Delete a menu item from your cart
4: View order cart
5: Check out your cart for payment
4
Order-ID: 100
Status: PENDING
Takeaway: false
        Name            QTY     Price
1       chicken nugget  3       20.70
Total: 20.70
```

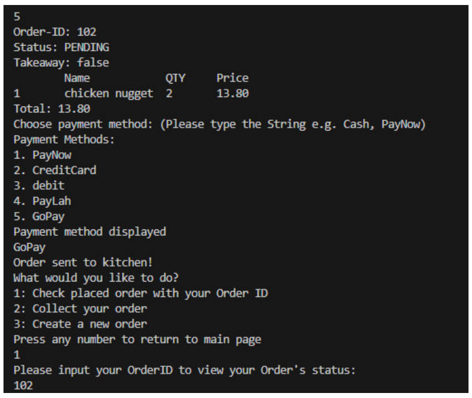
(a) customer waiting



```
Order sent to kitchen!
What would you like to do?
1: Check placed order with your Order ID
2: Collect your order
3: Create a new order
Press any number to return to main page
2
Please input your OrderID to collect your order:
100
Order not ready for collection!
What would you like to do?
1: Check placed order with your Order ID
2: Collect your order
3: Create a new order
Press any number to return to main page
```

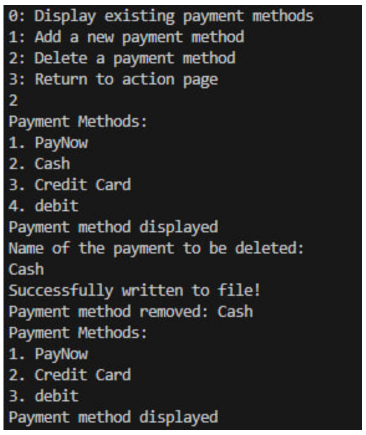
(b) order being sent

Figure 9: Customer order



```
5
Order-ID: 102
Status: PENDING
Takeaway: false
        Name            QTY     Price
1       chicken nugget  2       13.80
Total: 13.80
Choose payment method: (Please type the String e.g. Cash, PayNow)
Payment Methods:
1. PayNow
2. CreditCard
3. debit
4. PayLah
5. GoPay
Payment method displayed
GoPay
Order sent to kitchen!
What would you like to do?
1: Check placed order with your Order ID
2: Collect your order
3: Create a new order
Press any number to return to main page
1
Please input your OrderID to view your Order's status:
102
```

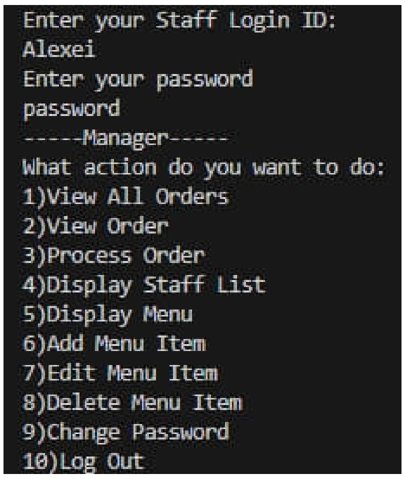
(a) paymentnew



```
0: Display existing payment methods
1: Add a new payment method
2: Delete a payment method
3: Return to action page
2
Payment Methods:
1. PayNow
2. Cash
3. Credit Card
4. debit
Payment method displayed
Name of the payment to be deleted:
Cash
Successfully written to file!
Payment method removed: Cash
Payment Methods:
1. PayNow
2. Credit Card
3. debit
Payment method displayed
```

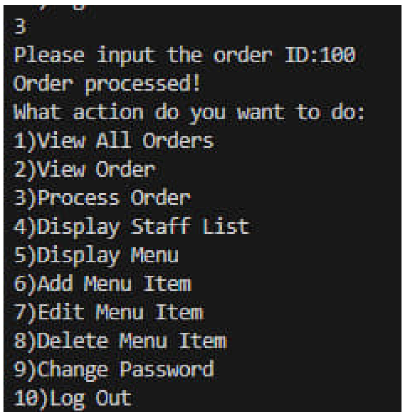
(b) payment method changing

Figure 10: Payment



```
Enter your Staff Login ID:
Alexei
Enter your password
password
-----Manager-----
What action do you want to do:
1)View All Orders
2)View Order
3)Process Order
4)Display Staff List
5)Display Menu
6)Add Menu Item
7)Edit Menu Item
8)Delete Menu Item
9)Change Password
10)Log Out
```

(a) staff login and things



```
3
Please input the order ID:100
Order processed!
What action do you want to do:
1)View All Orders
2)View Order
3)Process Order
4)Display Staff List
5)Display Menu
6)Add Menu Item
7)Edit Menu Item
8)Delete Menu Item
9)Change Password
10)Log Out
```

(b) staffs processing

Figure 11: Staff

4 Reflection

4.1 Difficulties Encountered and Conquering it through learning

Choosing the right model was quite challenging for us, especially given the varying workloads of team members. After much deliberation, we settled on the MVC (Model-View-Controller) model, implementing it through Boundary, Entity, and Controller components. This decision greatly facilitated translating our diagram into code, streamlining our progress.

Adopting the ECB (Entity-Control-Boundary) model marked a significant milestone for our project. It allowed for a more organized codebase and eased the coupling between classes. Utilizing hashmaps to establish connections between branches and team members enhanced our project's functionality. Additionally, we delved into data extraction, serialization, and deserialization, leveraging Excel to interface with Java.

This project served as a rich learning experience. We honed our teamwork skills, delved into designing SOLID models, and built a robust program following the ECB structure. Debugging and running Java files presented its own set of challenges, but through perseverance, we developed a keen eye for spotting and rectifying logic errors in our code.

4.2 Further Improvements

As we continue with the CLI model, our future aspirations include delving deeper into GUI-based applications. Transitioning to GUI would broaden our understanding of application development and enhance user experience. Our current system tackles real-world challenges in food ordering, providing valuable insights into various scenarios encountered during app development.

Expanding beyond CLI opens up avenues for incorporating languages like HTML, CSS, and potentially JavaScript. This multi-faceted approach promises to make the project more dynamic and interactive, offering users a richer experience.