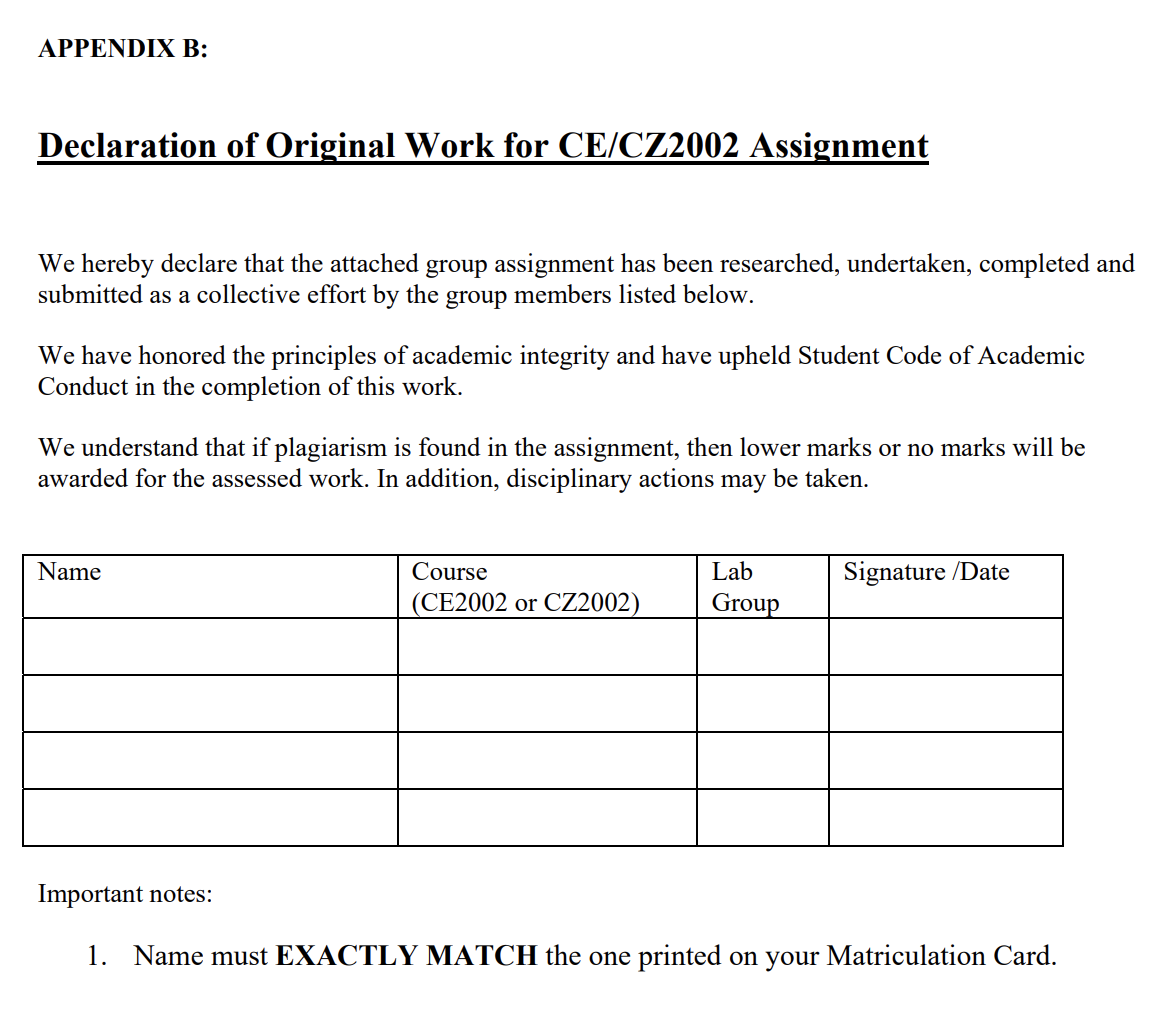
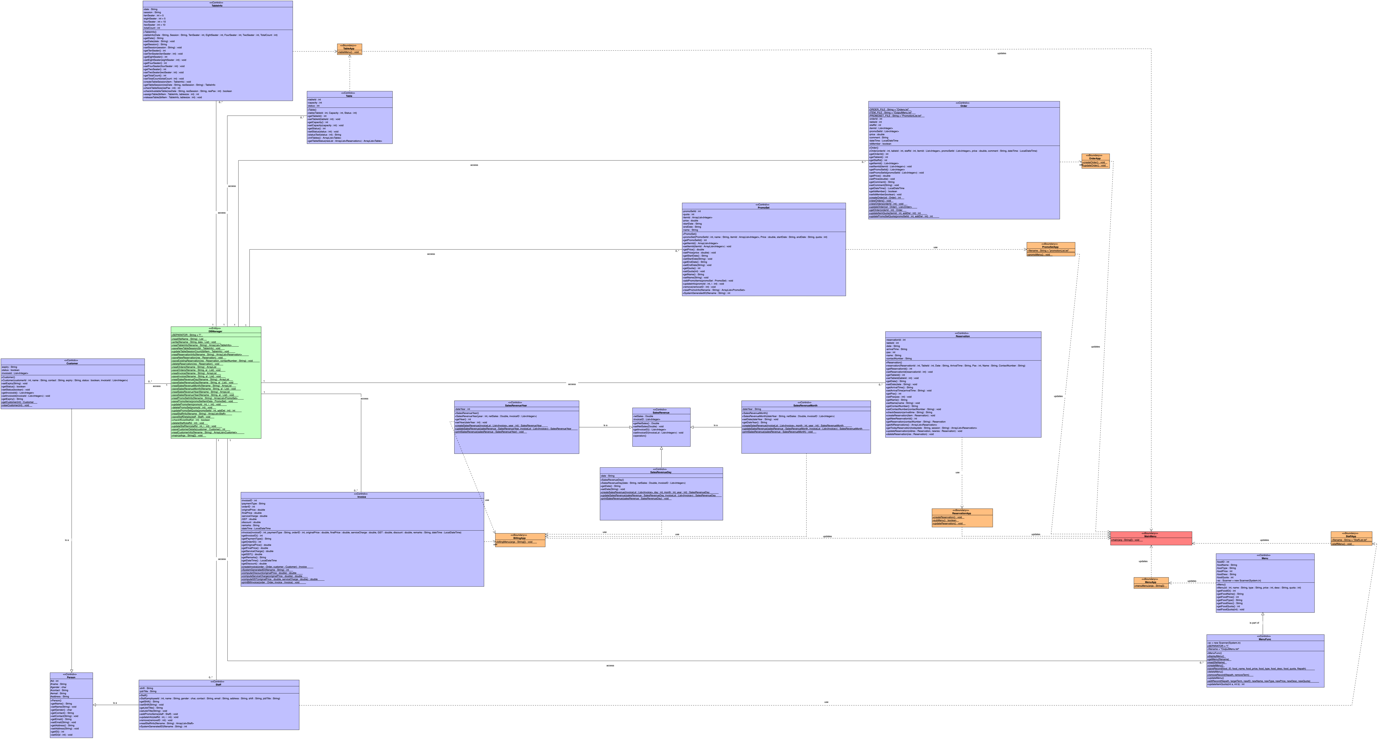
* ­­­­Scanned Copy of Declaration of Original Work for CZ2002 Assignment



**UML Class Diagram**



**UML Sequence Diagram**

* Insert picture here

**Design Considerations**

This Restaurant Reservation and Point of Sale System is built with the concept of three-tier architecture which separates the entire application into meaningful parts. So for our code, the classes will be divided into 3 components which is the boundary class (presentation layer), control class (business layer) and entity class (data layer). Each layer will have its own specific task to perform.

First of all, for the boundary class, it will be in charge of displaying information on the console for the user to see. As for the control class, it acts as the middleman to process information that are either passed in by the boundary or fetched by the entity before returning back to the boundary. Finally, the data layer will then read/write information into the database (text file).

The main consideration of utilizing this architecture is to maximize efficiency and allows all members to build their features at the same time without worrying about any conflicts in code. In addition, it will be much simpler to maintain code and to add in new features.

1. Single Responsibility Principle (SRP)

This states that each class and function should have responsibility of a single part of the feature. By doing so, whenever there is a adjustment in requirements of a class, the developer will then be able to quickly identify the other class that is linked and make the necessary changes.

For example, the Reservation class in our project is broken up into smaller classes. The ReservationApp class responsibility will be to request information or to display information from the Reservation class. As for the Reservation class, it will then perform the necessary processing and fetches data from the DBManager class. The DBManager class acts like the middleman between the database and the business layer class to perform database operations.



1. Open-Closed Principle (OCP)

This states that the software entities (classes or methods) should be open for extension but closed for modification. In simpler terms, this means that we should aim to write code that modifies one part for all the other classes which refers to the implementation of inheritance or abstract classes. For example, we have a Person class (Parent) where the Staff and Customer (Sub class) inherit variables from. If there’s a change in a data type for a certain variable in the Person Class, the Staff and Customer class will be updated also.

1. Liskov Substitution Principle (LSP)

This states that the classes should be replaceable with their sub classes without causing the application to break. Our application will not be too affected by it as we are using the 3 tier architecture approach. Besides that, we have at most 1-2 parent and child class which does not use any method overriding. Therefore, compared to other architectural approaches, it does not break the principle as easily.

1. Interface Segregation Principle (ISP)

This states that no client should be forced to depend on methods it does not used. For our application, each method performs its own specific method and calls other functions if necessary. For example, in the viewing of invoice method, if the user were to enter the selection for the payment, the method will then call the function to perform the task. In our application, we have clearly defined smaller and specific methods. By doing so, it helps to cut down on the code length and also does not rely on other functions to work.

1. Don’t Repeat Yourself (DRY)

This states that there should not be a repetition of code throughout the application. While implementing the validation for the presentation layer, we had decided to create a validation class whereby each class will be able to call from a function to validate the input. By doing so, each class do not have to repeat the same set of validation function over and over again and it shortens the code count.

**Use of OO Concepts**

Abstraction/Encapsulation/Information Hiding

This means the hiding of the complexity of an object by breaking it down into sub classes. As we are using the 3 tier architecture, the advantage is that each layer is an abstraction by itself. For example, the presentation layer is only required to know how to take in input and display the necessary information. Whereas for the business layer, it is only required to know how to process data. For further elaboration, instead of having just Order class to handle everything, we had decided to break it down to OrderApp & Order class. The OrderApp is the presentation layer and Order is the business layer as mentioned above.

Besides that, as it is a layered architecture, it provides a complete encapsulation whereby the user will never be able to reach the DBManager as it is handled by the control class which acts like a middleman. In addition, for all our classes, we have declared private variables so that it is only visible to the current class. Therefore, the user will not be able to access other class variables which makes it more secure.

Inheritance

Sharing of main variables in the parent class where the subclass inherits from. Only used for Person class (Parent) and Staff/Employee class (Sub class). The variables that is shared to the sub class are id, name, gender, contact, email, address. This is the only part of this project whereby it makes sense to use inheritance.

Polymorphism

Each features are of its own, there is no need to use method overloading or method overriding. Hence it is not implemented in our application.

New Features

1. Membership feature where it registers new users with a minimum spending of $100 and provides a 10% discount for members.
2. Reservation table that keep tracks of the number for different seats for all reservations date and session.

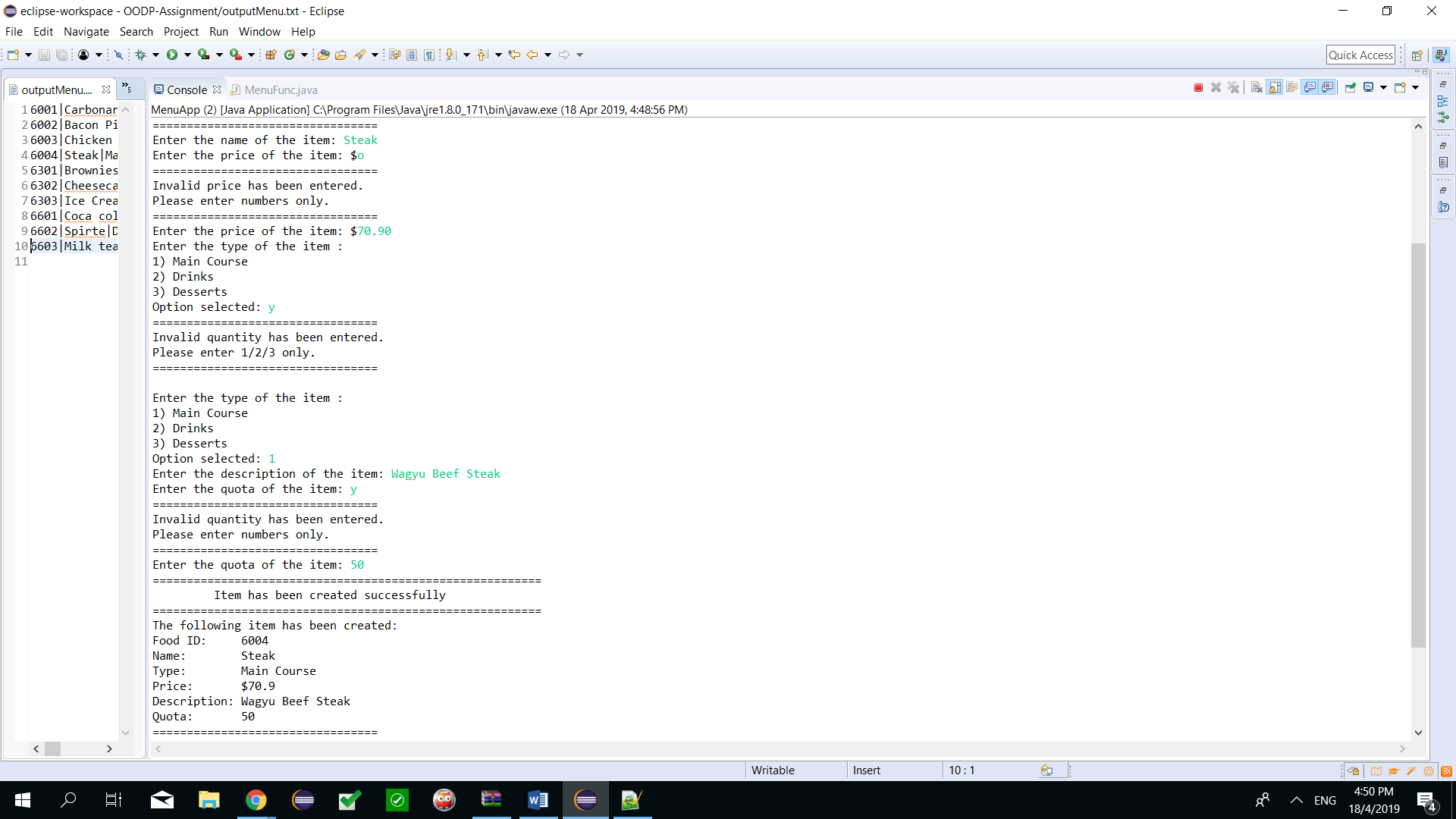
**Test Cases: Order**

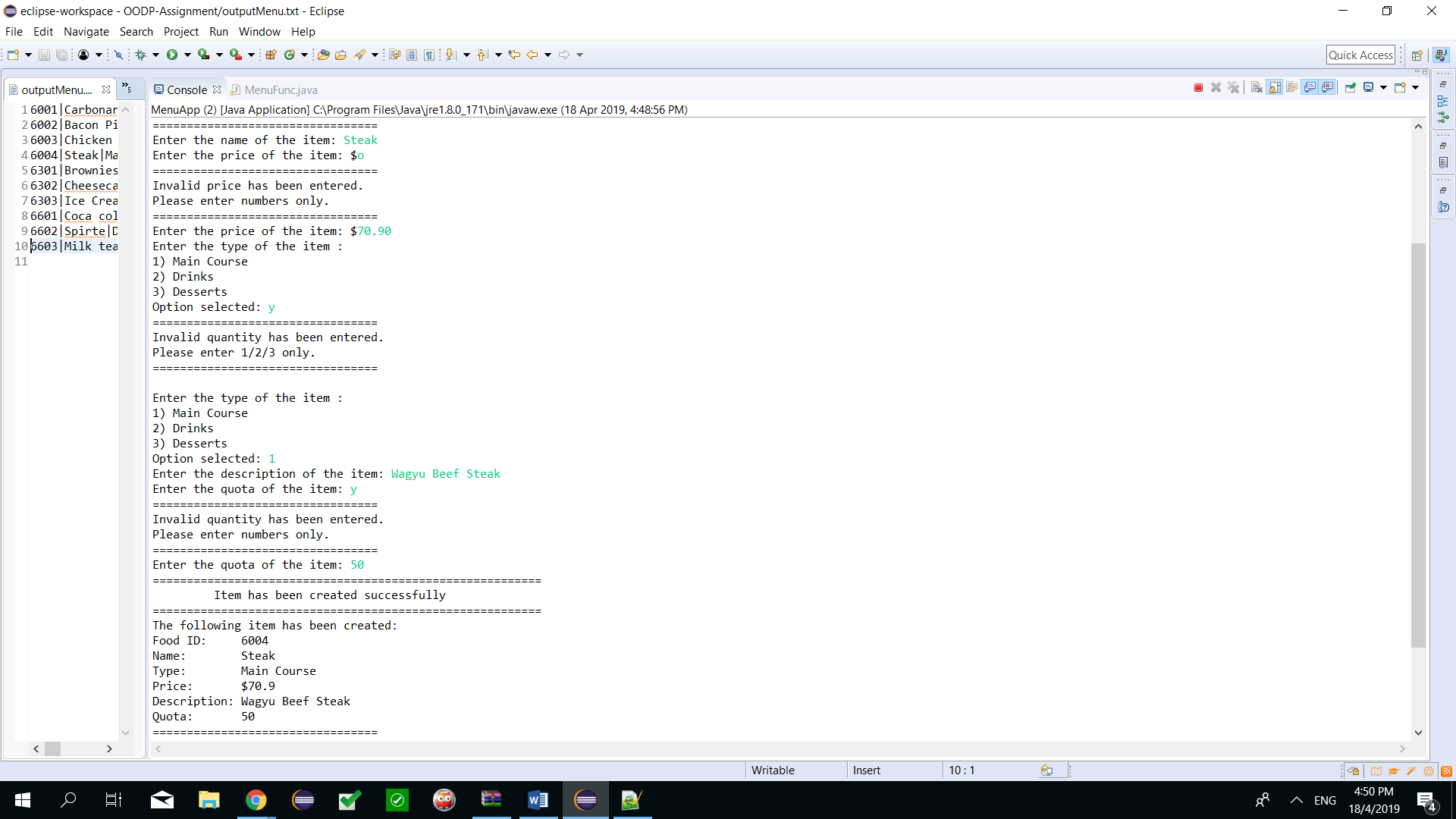
**Test Cases: Reservation**

**Test Cases: Menu Items**

**1) Create new menu item.**

All the possible errors are demonstrated below in Fig 1.1

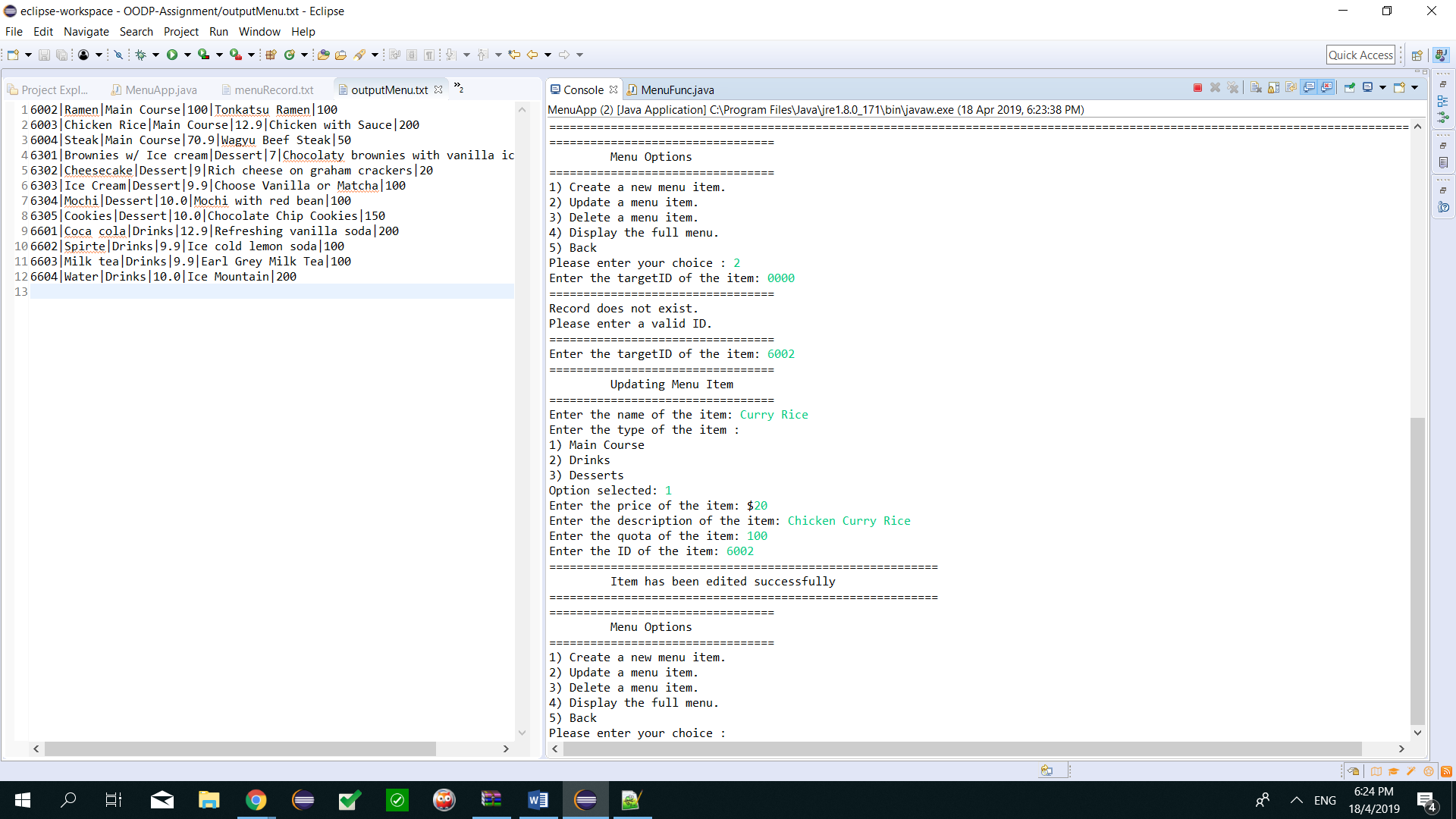




**Fig 1.1: Test case with all possible errors caught**

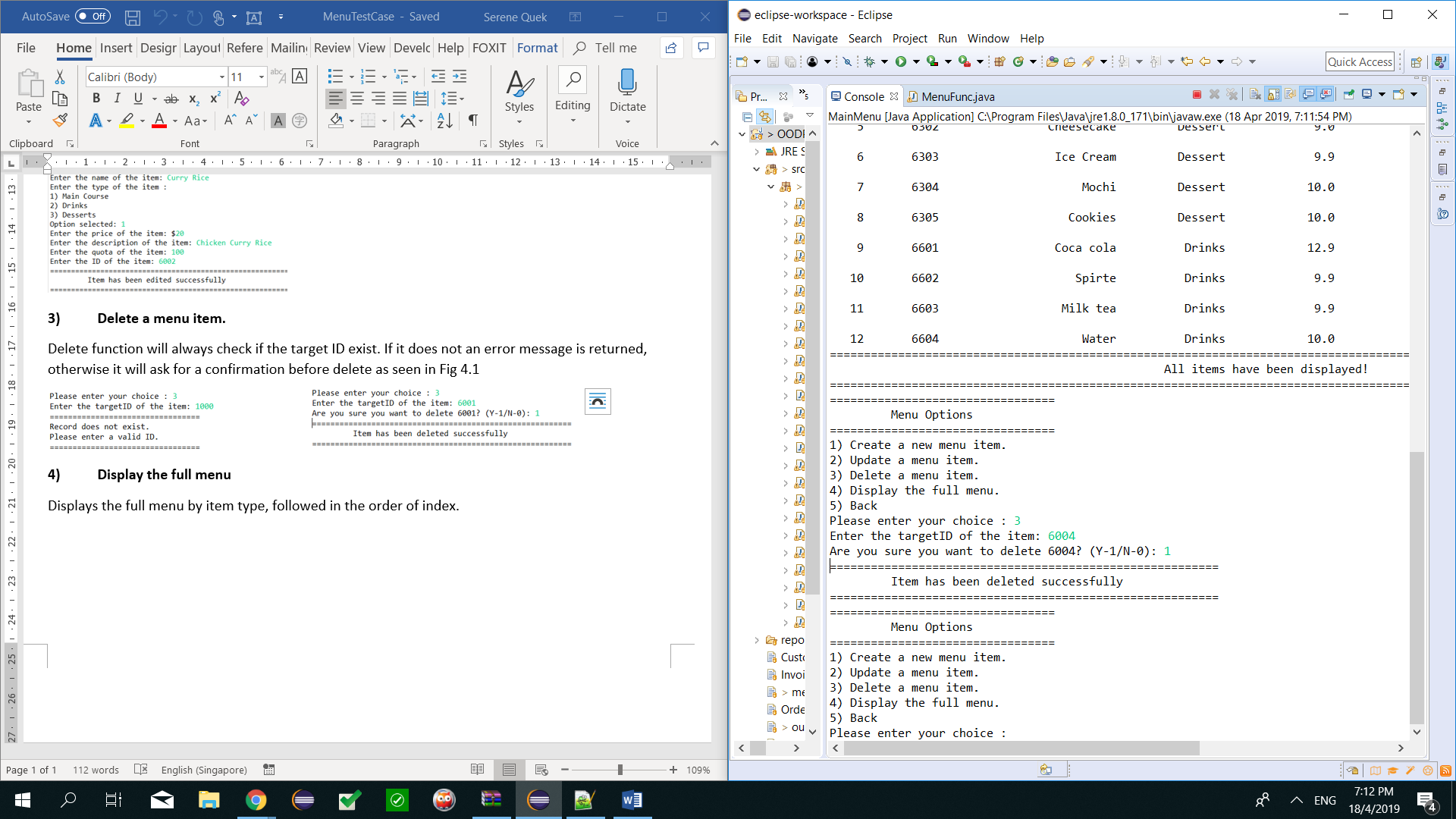
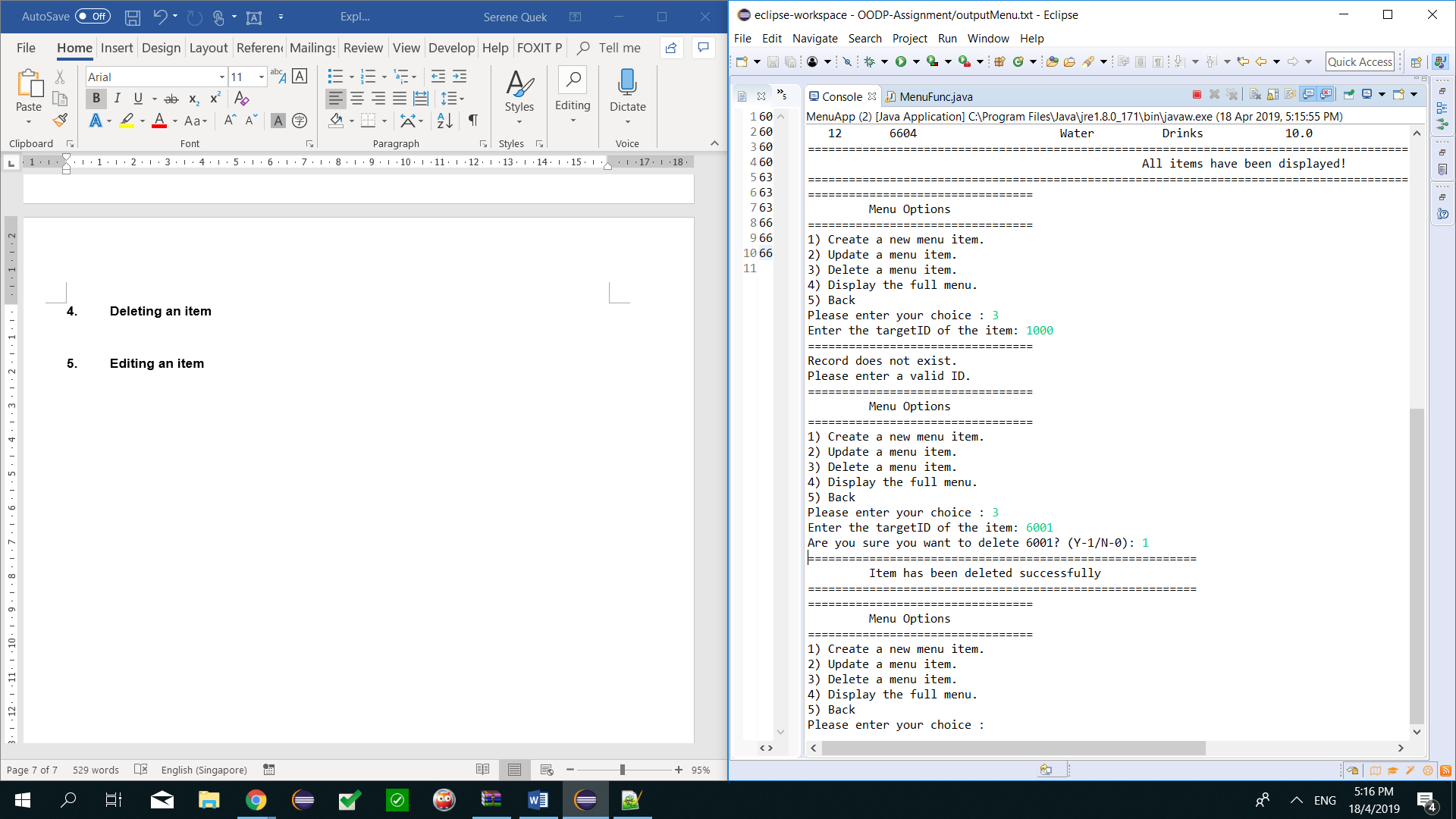
**2) Update a menu item.**

Editing an item would require the user to enter a targetID and if it does not exist will prompt for another ID that has to be valid.



**3) Delete a menu item.**

Delete function will always check if the target ID exist. If it does not an error message is returned, otherwise it will ask for a confirmation before delete as seen in Fig 4.1



**4) Display the full menu**

Displays the full menu by item type, followed in the order of index.

**Test Cases: Promotion item**

**Test Cases: Staff**

**Test Cases: Billing**

**Test Case: Membership**