**CS 3343**

Software Engineering Practice

**Group Project – “Hot Meals”**

Group-1

**Testing Report**

**Group Members**

* Banbah, Kush: 55786740
* Gupta, Aarnav: 55990960
* Jain, Utkarsh: 55992915
* Kasliwal, Aryan Girish: 55972222
* Malhotra, Avi: 55773896
* Rajagopalan, Pratul: 55858290

Table of Contents

[Module Organisation 3](#_Toc89640026)

[Testing Strategy 4](#_Toc89640027)

[Coverage 7](#_Toc89640028)

[Code Refactoring 9](#_Toc89640029)

[Test Cases 12](#_Toc89640030)

# Module Organisation

Diagram

Description automatically generated

The class diagram above helped us understand the dependencies within the system. Please note that the above image is only a reference. For a detailed explanation, please refer to the Design & Analysis report.

The UML class diagram revealed which classes depend on the methods of which other classes. A deep understanding of these dependencies helped us to come up with a testing plan and devise thorough test cases.

# Testing Strategy

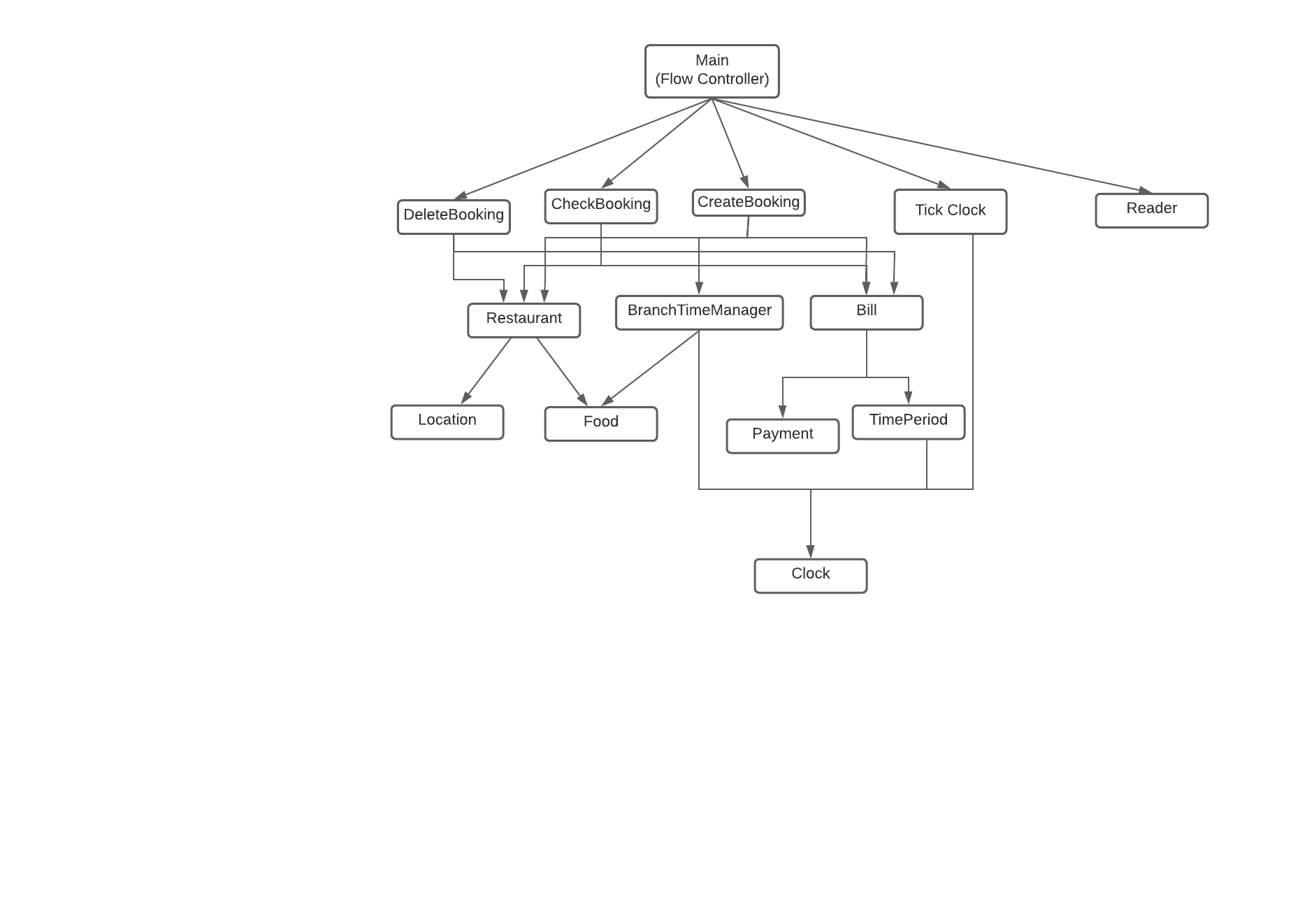
For our testing strategy, we used the bottom up method. We chose this method for many reasons,

* Testing critical functionality: Our code had a lot of base classes that performed a majority of the functionality of the program and hence it made a lot of sense to thoroughly test each critical class first instead of creating test stubs for higher level classes.
* Maintaining parallelism: In the initial phase of the project, all our team members were coding on different classes. Hence it made more sense for each person to devise their own test cases for the classes they wrote and so using the bottom up method made it relatively easy to maintain parallelism.
* Test stubs: Since the button-up approach does not require the use of test stubs for simulation of lower classes, we could ensure that every nook and cranny of the system was being tested before we moved on to integration/system testing.
* Team collaboration: testing and planning the work was quite easy to do, as each person simply wrote tests for their own classes, the bottom-up testing allowed us to begin testing the base classes as soon as we finished coding them.

However, some classes couldn’t be tested and or needed stubs. Classes like Info and Reservation were data classes that just stored values and getters or setters. Reservation used functions from other classes so those classes were tested instead. Lastly, Payment class was our simulation of external API and it had a success rate of 50% in real use. However, this isn’t testable, so a stub was used to set the success as true or false for payment and the classes that relied on it. CreateReservationStub and FlowControllerStub were created to use the new payment stub to simulate success or failure.

Also, to assist with testing we had to create a ScannerTestThread class that closes a scanner after an arbitrary amount of time to help catch the errors while testing. Whenever errors are tested with user input, this class would exit the testing class to see if the error caught was the error intended.

The following diagram depicts our testing plan in its entirety. The higher-level tests of classes require the lower ones to be tested to function properly. Some classes like reader use Location, Restaurant and Reservation by only creating objects and not running any function and hence don’t require testing of those classes to test Reader Class, another example is Location had a function that called Branch Time Manager’s function. This was instead testing in Branch Time Manager instead of Location, Location could work without this function for testing.

It is worth mentioning that the above class hierarchy does not encompass all the classes writing in the project code. Nonetheless, this is by design and is well supported by reasons. For example, testing the CreateBooking class entails a testing of all its child classes and all exception classes are utilized as errors are thrown as and when needed during the testing process.

Moving on, we first began with unit testing on the following classes

* Payment
* Reader
* Food
* Clock
* Location

We then proceeded to integration testing on the following classes:

* Time Period + Clock
* Bill+TimePeriod
* BranchTimeManager + Food + Clock
* Restaurant + Location + Food
* Check Booking + Restaurant+Bill
* Delete Booking + Restaurant + Location + Bill
* Tick Clock + Clock
* Create Reservation + Location + Clock + Food + Restaurant + Payment + Time Period

Then, we proceeded on to do system testing on the whole program using all the classes by writing tests that simulate bookings and reservations made by multiple customers in a day.

# Coverage

A picture containing text

Description automatically generated

We adopted a DC coverage method for our project. Decision Coverage is where we evaluate each decision to be True or False.

The following method is an example from our source code inside the class ‘TimePeriod’ inside the module ‘logistics’. Test cases T79 and T80 cover the follow method and its table is shown below: We do not test if every boolean expression within the if condition is true or false, rather we test if every if condition is true and false. In this case, there is only one variable so res==0 is tested as true and false.

Graphical user interface, text, application

Description automatically generated

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| this | this.start | this.end | timePeriod | timePeriod .start | timePeriod .end | res | Res = 0 ? | return val |
| 11:00-12:00 | 11:00 | 12:00 | 11:00-12:00 | 11:00 | 12:00 | 0 | True | 0 |
| 11:00-12:00 | 11:00 | 12:00 | 11:00-12:00 | 11:00 | 12:00 | 60 | False | 60 |

# Code Refactoring

When the first version of the code was done, it was important for us to refactor it into using better practices. There are four strategies that we had used:

1. **Extract Method:** There were occurrences where, while coding, our method body got too large. A single method, such as addReservation() in the class CreateReservation(). Initially, addReservation() handled all the functions, such as choosing the number of people dining, selecting the menu items, and other things along with that. But, we then broke this method down into multiple methods for better handling and easier testing. The new methods extracted from it are:

* selectLocation(): Called in line 30
* selectDiners(): Called in Line 31
* selectDishes(): Called in Line 32
* selectTime(): Called in Line 37

Graphical user interface, text, application

Description automatically generated

1. **Moving Child Fields to Parent Class:** In the multiple sub-classes of the Food class, such as MainCourse, Dessert and Appetizer, they all had the same private variables, such as name of the food item, time to eat it, and its price, denoted by String name, int eatingTime and int price respectively. We took these variables out of the sub-classes and put them as private variables of their parent class Food.

Text

Description automatically generated

Graphical user interface, application

Description automatically generated with medium confidence

1. **Group Parameters into one Class:** Earlier, we stored multiple variables over multiple classes, constantly passing them through multiple getters here and there, which broke a lot of Software Design principles. To solve that, we made an Info class, which stored all the necessary information, and had the required getters and setters.

Graphical user interface

Description automatically generated with medium confidence

1. **Extracting Method from Code Body:** Initially, there were methods that accessed parameters from other classes in a way that didn’t make a lot of sense. Such as, the findBooking() function used to access the entire list of reservations in the Restaurant class from a different class, until it was moved to Restaurant class itself.

Graphical user interface, text, application

Description automatically generated

# Test Cases

Bill.java - To test if the bill class can correctly display the bill amount, booking ID and the entire bill with the booking details

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| 1T1 | Calculate Bill amount when provided with food items ordered | Bill amount = 60 | As expected |
| 1T2 | Get bookingID from a bill when food items were ordered at a specific location | BookingID = 003 | As expected |
| 1T3 | Show bill for an order of dishes | Prints the bill with prices of the dishes, and details of the booking | As expected |

Reader.java - To test if the reader can handle reading errors and can read the given menu and location files correctly

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| 10T1 | Check if the reader can handle an incorrect file while reading the menu | Output of ‘File Error for menu’ | As expected |
| 10T2 | Check if the reader can handle a file error while reading for the locations | Output of ‘File error for location’ | As expected |
| 10T3 | Check if the reader can handle a reading error while reading for locations | Output of ‘Reading error for location’ | As expected |
| 10T4 | Check if the reader can handle a reading error while reading the locations | Output of ‘Reading error for menu’ | As expected |
| 10T5 | Check if the reader can read the input menu file correctly | The two menus are the same | As expected |
| 10T6 | Check if the reader can read the input location file correctly | The two location lists are the same | As expected |

TimePeriod.java - Checking if our time period class can handle time comparisons

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| 14T1 | Check if the time is within the interval | True | As expected |
| 14T2 | Check if the time given is not in the interval | False | As expected |
| 14T3 | Check if the two time periods are the same | True | As expected |
| 14T4 | Check if two different time periods are the same | False | As expected |
| 14T5 | Get total time in the interval | 60 | As expected |
| 14T6 | Check if a newly created different time period is the same an existing time period | False | As expected |
| 14T7 | Check if a newly created identical time period is the same as an existing time period | True | As expected |

Food.java - Testing the total eating time calculation

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| 7T1 | Testing if the class can calculate the total eating time correctly | 27 minutes | As expected |
| 7T2 | Testing if wrong input will change the eating time | False | As expected |

Clock.java

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| 4T1 | Check if clock is created using int constructor | Clock is made | As expected |
| 4T2 | Check if clock is created using String constructor | Clock is made | As expected |
| 4T3 | Check if Clock to String works with mins less than 10 | String is created with proper format | As expected |
| 4T4 | Check if Clock to String works with mins greater than 10 | String is created with proper format | As expected |
| 4T5 | Check if clock is same as another clock | CompareTo returns 0 to show clocks are same | As expected |
| 4T6 | Check if clock is before another clock | CompareTo returns -60 to show clocks difference | As expected |
| 4T7 | Check if clock is after another clock | CompareTo returns 60 to show clocks difference | As expected |
| 4T8 | Test if a clock equals another clock | .equals returns true since clocks are equal | As expected |
| 4T9 | Test if a clock doesn’t equal another clock | .equals returns false since clocks are not equal | As expected |
| 4T10 | Test if clock clone equals orginal clock | Clone is equal to orginal clock | As expected |
| 4T11 | Test if the iterator has next with no more iterations | Iterator returns false since there are no more iterations | As expected |
| 4T12 | Test if iterator has next with multiple more iterations to go | Iterator returns true since it has more iterations | As expected |
| 4T13 | Test if the next Iterator equals next time | Iterator moves to next time | As expected |
| 4T14 | Test if iterator with multiple iterations equals the result | Iterator moves forward multiple times to result correct time | As expected |
| 4T15 | Test if iterator next works without specifying end time | Iterator functions same as before by setting end time to 11:00pm automatically | As expected |
| 4T16 | Test if iterator works with different tick time and end time | Iterator creates clock of 12:00 | As expected |
| 4T17 | Test if exception is thrown when incorrect input for clock creation | Exception is thrown | As expected |
| 4T18 | Test if exception is thrown when incorrect input for clock creation | Exception is thrown | As expected |
| 4T19 | Test if exception is thrown when Clock is made out of range | Exception is thrown | As expected |
| 4T20 | Test if exception is thrown when clock is created with wrong input | Exception is thrown | As expected |

Payment.java - Testing if the payment can succeed or fail

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| 9T1 | Testing if payment is successful | ‘Payment Success’ | As expected |
| 9T2 | Testing if payment is unsuccessful | ‘Payment failed’ | As expected |

Location.java - Checking current capacity in a location at a specific time

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| 8T1 | Checking current capacity when no reservations have been made | Capacity is zero | As expected |
| 8T2 | Checking current capacity when a reservation for 3 people was made | Capacity is 3 | As expected |
| 8T3 | Checking capacity when a reservation for the total capacity was made | Capacity is 10 | As expected |

Restaurant.java - Testing to see if we can find a booking that was made in the restaurant

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| 11T1 | Checking for a booking for a reservation that was made in the restaurant | Booking is found | As expected |
| 11T2 | Checking for a booking for a reservation not made in a restaurant could be found | Booking not found | As expected |

CheckBookingTest.java - Check if the system is capable of finding bookings and handling exceptions

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| 3T1 | Check without a booking made | ‘Booking not found’ | As expected |
| 3T2 | Check for a booking that has been made | Booking found with details | As expected |
| 3T3 | Check for an incorrect booking ID with wrong input | Booking not found, enter booking ID greater than 0 | As expected |
| 3T4 | Check for an incorrect booking ID | Booking not found | As expected |
| 3T5 | Check for an incorrect booking ID with wrong input – letters instead of numbers | Booking not found, enter a valid number | As expected |

Delete Booking - Checking if the system can handle deletion of bookings and error handling

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| 6T1 | Delete a booking when no bookings are made | Booking not found, could not delete | As expected |
| 6T2 | Find a booking and delete it booking | Booking found and deleted | As expected |
| 6T3 | Find the booking to delete but choose not to delete | Booking found but not deleted | As expected |
| 6T4 | Give incorrect input while finding booking | Enter a valid number | As expected |
| 6T5 | Give incorrect negative input while finding booking | Enter a valid number | As expected |
| 6T6 | Give incorrect booking id while finding booking | Booking not found, could not delete | As expected |

Tick clock – Test if the clock can be ticked forward correctly

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| 13T1 | Give the clock negative input to see if it can handle negative numbers | Error! Please input a postiive number | As expected |
| 13T2 | Give the clock incorrect input to test exception handling | Error! Not a number | As expected |
| 13T3 | Give the clock a correct input of a 10 minute tick | Time is now 11:10 | As expected |

BranchTimeManagerTest – Tests whether the class BranchTimeManager returns the correct ArrayList of TimePeriods to the customer

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected result** | **Actual Output** |
| 2T1 | Create a fake booking and request branch time manager for a list of availabe times. | (11:00-22:32) | As expected |
| 2T2 | Create a fake reservation to block out the complete restaurant location for a timeperiod, then make another reservation to test if the system calculates the blocked time correctly. | (11:00-11:32, 13:00-22:32) | As expected |

CreateReservationTest – Test the class CreateReservation and its methods that allow the user to create a reservation at a restaurant.

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| 5T1 | Adds a list of dishes to a reservation and tests whether a reservation stores its food items correctly | 4 Dishes - dishes 1 2 3 and 4 from the menu | As expected |
| 5T2 | Tests create reservation by passing in a newline character followed by a number | True | As expected |
| 5T3 | Tests create reservation by passing a newline character followed by some non-numeric value | True | As expected |
| 5T4 | Tests create reservation by passing in two newline characters | True | As expected |
| 5T5 | Tests the ability to add the number of guests to the reservation | 5 | As expected |
| 5T6 | Test case when the number of diners exceeds the maximum capacity of the restaurant | "Enter number of people dining in: \r\nError!: Number of diners, 11 exceeds max capacity of 10!\r\n" | As expected |
| 5T7 | Test case when the number of diners is negative | "Enter number of people dining in: \r\nError!: Number of diners, -1 is not a proper amount of diners!\nPlease enter a number from 1 - 10\r\n" | As expected |
| 5T8 | Test case when the number of diners is a non numeric value | True | As expected |
| 5T9 | Tests the input value for start time by the customer | True | As expected |
| 5T10 | Tests when the time input by the customer is not in the given range by the system | True | Ax expected |
| 5T11 | Tests invalid input format of time – 11::35 | True | As expected |
| 5T12 | Tests when the input value of time is not in range | True | As expected |
| 5T13 | Tests the value customer inputs to select location when input value is out of range | True | As expected |
| 5T14 | Tests the value customer inputs to select location when the value is not numeric | True | As expected |
| 5T15 | Tests the value customer inputs to select location when the value is correct | Selected location | As expected |
| 5T16 | Tests the systems ability to store the time period in the reservation once the input time by the customer is verified | True | As expected |
| 5T17 | Tests the systems ability to store the time period in reservation once the input time by the customer is verified | True | As expected |
| 5T18 | Tests the systems ability to create a payment | True | As expected |
| 5T19 | Tests whether a payment has been made when the user selects “Yes” when prompted to make the payment | True | As expected |
| 5T20 | Tests whether a payment has been made when the user selects “No” when prompted to make payment | False | As expected |
| 5T21 | Tests whether a reservation with correct characteristics is being added to the list of reservations | "Enter your name: \r\n  \r\n  Booking is for Kush\r\n  Booking is for 10 guests\r\n  Your booking time is: 11:00 - 12:00\r\n  Main - 10 HKD\n  App - 15 HKD\n  Dessert - 5 HKD\n  Total amount: 30 HKD\n" | As expected |
| 5T22 | Tests when an incomplete booking was made and tried to store to the list of reservations | "Enter your name: \r\n  \r\n  Your booking was not confirmed, please retry ordering to make a new booking!\r\n" | As expected |

SystemTest – Test the entire system using a stub for CreateReservation to set the success of payment through stub

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| 12T1 | Create a reservation | Reservation is made | As expected |
| 12T2 | Create 2 reservations | 2 reservations are made and incorrect times are not shown | As expected |
| 12T3 | Create 2 reservations and check booking | 2 Reservations are made and checked | As expected |
| 12T4 | Tick clock and make 2 reservations are different locations | Clock is ticked forward and reservations are made after the tick at different locations | As expected |
| 12T5 | Create a reservation, check it, delete it, then check it | A resrevation is made, found, deleted and cannot be found anymore | As expected |