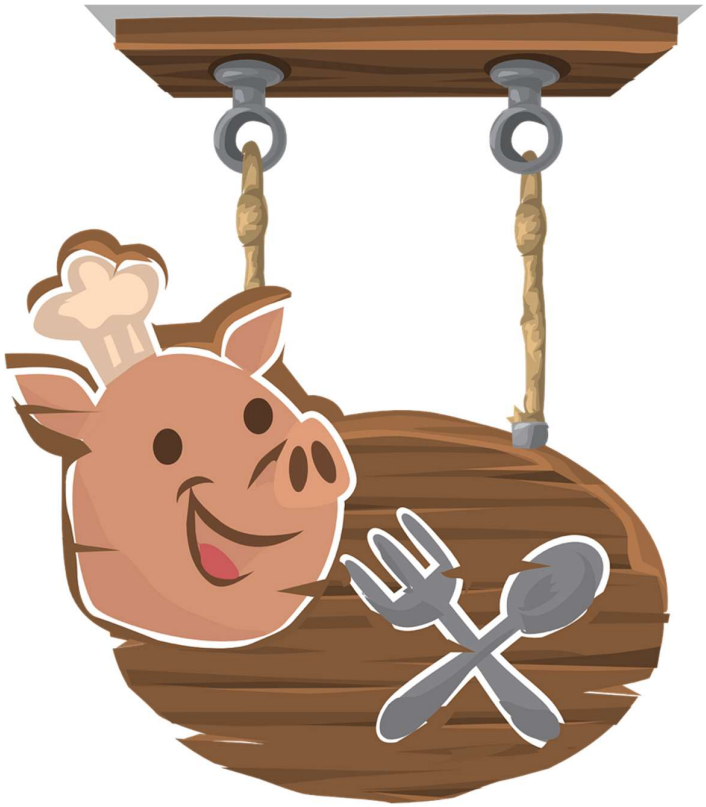


CS3343 Software Design

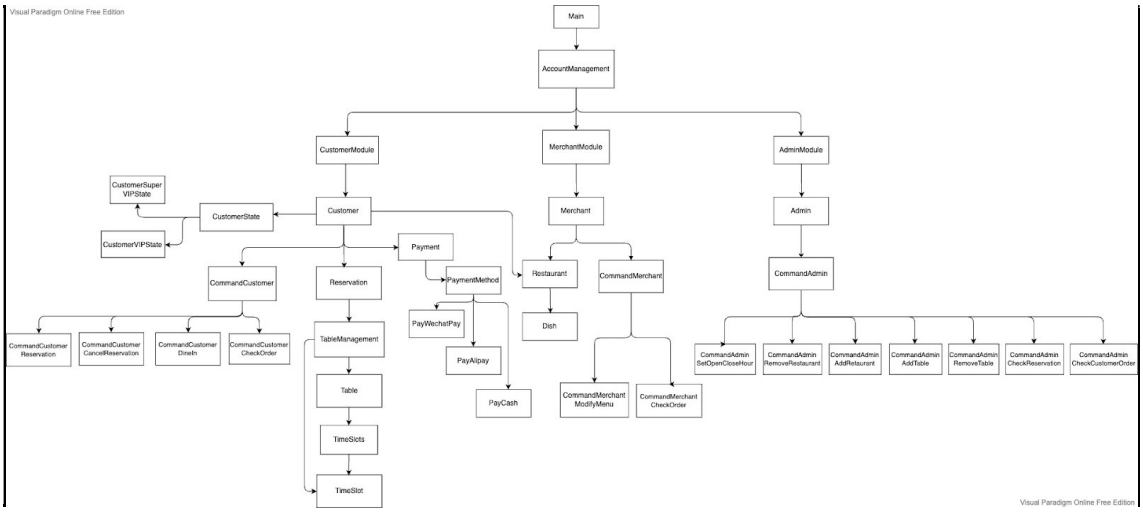
Test Report

Group 12

Name	Student ID	Title
CHENG Yin		Project Manager
SONG Tao		Assistant Project Manager
LUO Peiyuan		Programmer
ZHOU Junchen		Programmer
DU Wenxi		Tester
FENG Yong		Tester

Table of Contents

1. Testing Module Organization



The diagram describes the testing structure of our GoGoEat system. There are three important modules under the Main and AccountManagement module, which are the Customer module, Merchant module, and Admin module.

2. Testing Strategy

We were programming and testing simultaneously with the V-model development process. As soon as we got the user requirement and architecture, we started black box testing design. Then coming to the detailed design, we did unit testing when unit codes were done and did integration testing when branches were defined. With this process, we achieved saving time and improved the project code quality.

In this project, we used multiple testing strategies as listed below:

1. Bottom-Up Approach
2. Black-box Testing

Due to the wide scope, it is difficult to express our system into a tree structure, so the bottom-up approach and white-box testing are insufficient. We also test some significant units apart from the bottom parts of units to increase the CC/DC coverage. Apart from white-box testing, black-box testing is also used in integration testing and system testing.

2.1 Bottom-up testing

Bottom-up testing is a specific type of integration testing that tests the lowest components of a code base first. More generally, it refers to a middle phase in software testing that involves taking integrated code units and testing them together before testing an entire system or code base.

To conclude our test coverage, we achieved more than 50% of the coverage for both unit and integration testing, testing all the methods except those for printing purposes only.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
GoGoEat	69.9 %	9,021	3,880	12,901
src	69.9 %	9,021	3,880	12,901
GoGoEat	53.9 %	4,486	3,830	8,316
TestGoGoEat	98.9 %	4,535	50	4,585

The black-box testing approach, which will be introduced later, will cover the remaining test cases together with inputting and outputting.

2.2 Black-box Testing

Due to the broad scope of the project and the massiveness of the test scenarios, we used the black-box testing to provide the minimum test cases that would cover most of the test scenarios.

- **Default Value Strategy:**
  The default value strategy determines a default value for each parameter, starts the test with a test case in which all parameters take default values, varies one parameter from its default value after each test, and repeats the above step until every value of every parameter has appeared in the tests at least once.

- **Category-Partition Method (CPM)**
  The Category-Partition Method (CPM) required us to analyze the specification to identify the individual testable function units, parameters of the function units, and entities in the environment; classify the specifications into categories and partition the categories into choices; determine the constraints among the choices; write the test specification, and express them in Test Specification Language (TSL); evaluate the output of each test frame, and convert the test frames into test cases.

3. Unit Test Cases Analysis

3.1 TimeSlots.java

Branch Coverage:

TimeSlots.java	98.6 %	484	7	491
TimeSlots	98.6 %	484	7	491
addSlot(TimeSlot)	86.0 %	37	6	43
remove(TimeSlot)	96.4 %	27	1	28
getCloseTime()	100.0 %	2	0	2
getOpenTime()	100.0 %	2	0	2
setOpenAndCloseTime(String, String)	100.0 %	15	0	15
TimeSlots()	100.0 %	8	0	8
TimeSlots(String)	100.0 %	12	0	12
checkAvailable(TimeSlot)	100.0 %	54	0	54
checkReservedStatus(LocalTime)	100.0 %	52	0	52
checkReserver(LocalTime)	100.0 %	27	0	27
checkValidReserver(LocalTime, String)	100.0 %	31	0	31
getAvailableSlots()	100.0 %	118	0	118
getDate()	100.0 %	3	0	3
getReservationStartEndInDay()	100.0 %	39	0	39
hasReserved(LocalTime)	100.0 %	31	0	31
isEmpty()	100.0 %	5	0	5

Test 1

Purpose: To check whether the function ***setOpenAndCloseTime()*** can successfully set the time limit of the time slot if the user wants to add a timeslot in the future.

```
@Test
public void timeSlotTest1() throws ExTimeFormatInvalid {
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1;
    try {
        ts1 = new TimeSlot("09:00:00","21:00:00","tester1");
    } catch (Exception e) {
        assertEquals(true, e instanceof ExTimeSlotInvalid);
    }
}

@Test
public void timeSlotTest2() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    assertEquals(true, tss.addSlot(ts1));
}

@Test
public void timeSlotTest3() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    TimeSlot ts2 = new TimeSlot("12:00:00","13:00:00","tester2");
    tss.addSlot(ts1);
    assertEquals(false, tss.addSlot(ts2));
}

@Test
public void timeSlotTest4() {
    TimeSlots tss = new TimeSlots("2022-10-27");
    LocalDate thisday = LocalDate.parse("2022-10-27");
    assertEquals(thisday, tss.getDate());
}

@Test
public void timeSlotTest5() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots("2022-10-27");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    tss.addSlot(ts1);
     LocalTime time = LocalTime.parse("14:00:00");
    assertEquals(null, tss.checkReserver(time));
}
```

Test 2

Purpose: Check the function ***addSlot()*** can successfully add a new slot into the timeslots ArrayList or not.

```
@Test
public void timeSlotTest1() throws ExTimeFormatInvalid {
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1;
    try {
        ts1 = new TimeSlot("09:00:00","21:00:00","tester1");
    } catch (Exception e) {
        assertEquals(true, e instanceof ExTimeSlotInvalid);
    }
}

@Test
public void timeSlotTest2() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    assertEquals(true, tss.addSlot(ts1));
}

@Test
public void timeSlotTest3() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    TimeSlot ts2 = new TimeSlot("12:00:00","13:00:00","tester2");
    tss.addSlot(ts1);
    assertEquals(false, tss.addSlot(ts2));
}

@Test
public void timeSlotTest4() {
    TimeSlots tss = new TimeSlots("2022-10-27");
    LocalDate thisday = LocalDate.parse("2022-10-27");
    assertEquals(thisday, tss.getDate());
}

@Test
public void timeSlotTest5() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots("2022-10-27");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    tss.addSlot(ts1);
     LocalTime time = LocalTime.parse("14:00:00");
    assertEquals(null, tss.checkReserver(time));
}
```

Test 3

Purpose: Check the function ***addSlot()*** can prevent adding duplicate timeslot or not.

```
@Test
public void timeSlotTest1() throws ExTimeFormatInvalid {
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1;
    try {
        ts1 = new TimeSlot("09:00:00","21:00:00","tester1");
    } catch (Exception e) {
        assertEquals(true, e instanceof ExTimeSlotInvalid);
    }
}

@Test
public void timeSlotTest2() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    assertEquals(true, tss.addSlot(ts1));
}

@Test
public void timeSlotTest3() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    TimeSlot ts2 = new TimeSlot("12:00:00","13:00:00","tester2");
    tss.addSlot(ts1);
    assertEquals(false, tss.addSlot(ts2));
}

@Test
public void timeSlotTest4() {
    TimeSlots tss = new TimeSlots("2022-10-27");
    LocalDate thisday = LocalDate.parse("2022-10-27");
    assertEquals(thisday, tss.getDate());
}

@Test
public void timeSlotTest5() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots("2022-10-27");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    tss.addSlot(ts1);
     LocalTime time = LocalTime.parse("14:00:00");
    assertEquals(null, tss.checkReserver(time));
}
```

Test 4

Purpose: To check whether the function ***getDate()*** can successfully output the correct date that the user has set before.

```
@Test
public void timeSlotTest1() throws ExTimeFormatInvalid {
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1;
    try {
        ts1 = new TimeSlot("09:00:00","21:00:00","tester1");
    } catch (Exception e) {
        assertEquals(true, e instanceof ExTimeSlotInvalid);
    }
}

@Test
public void timeSlotTest2() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    assertEquals(true, tss.addSlot(ts1));
}

@Test
public void timeSlotTest3() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    TimeSlot ts2 = new TimeSlot("12:00:00","13:00:00","tester2");
    tss.addSlot(ts1);
    assertEquals(false, tss.addSlot(ts2));
}

@Test
public void timeSlotTest4() {
    TimeSlots tss = new TimeSlots("2022-10-27");
    LocalDate thisday = LocalDate.parse("2022-10-27");
    assertEquals(thisday, tss.getDate());
}

@Test
public void timeSlotTest5() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots("2022-10-27");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    tss.addSlot(ts1);
     LocalTime time = LocalTime.parse("14:00:00");
    assertEquals(null, tss.checkReserver(time));
}
```

Test 5

Purpose: To check whether the function ***checkReserver()*** can successfully output the correct reservation owner number if the timeslot has been reserved before.

```
@Test
public void timeSlotTest1() throws ExTimeFormatInvalid {
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1;
    try {
        ts1 = new TimeSlot("09:00:00","21:00:00","tester1");
    } catch (Exception e) {
        assertEquals(true, e instanceof ExTimeSlotInvalid);
    }
}

@Test
public void timeSlotTest2() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    assertEquals(true, tss.addSlot(ts1));
}

@Test
public void timeSlotTest3() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    TimeSlot ts2 = new TimeSlot("12:00:00","13:00:00","tester2");
    tss.addSlot(ts1);
    assertEquals(false, tss.addSlot(ts2));
}

@Test
public void timeSlotTest4() {
    TimeSlots tss = new TimeSlots("2022-10-27");
    LocalDate thisday = LocalDate.parse("2022-10-27");
    assertEquals(thisday, tss.getDate());
}

@Test
public void timeSlotTest5() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots("2022-10-27");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    tss.addSlot(ts1);
     LocalTime time = LocalTime.parse("14:00:00");
    assertEquals(null, tss.checkReserver(time));
}
```

Test 6

Purpose: To check whether the function ***checkReserver()*** can successfully output the null result if the timeslot has not been reserved before.

```
@Test
public void timeSlotTest6() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots("2022-10-27");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    tss.addSlot(ts1);
     LocalTime time = LocalTime.parse("12:30:00");
    assertEquals("tester1", tss.checkReserver(time));
}

@Test
public void timeSlotTest7() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots("2022-10-27");
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1 = new TimeSlot("10:00:00","11:00:00","tester1");
    TimeSlot ts2 = new TimeSlot("11:25:00","13:00:00","tester2");
    tss.addSlot(ts1);
    tss.addSlot(ts2);

    assertEquals("13:00-19:00", tss.getAvailableSlots());
}

@Test
public void timeSlotTest8() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot timeslot = new TimeSlot("12:00:00","13:00:00","0");
    assertEquals(true, ts.addSlot(timeslot));
}
```

Test 7

Purpose: To check whether the function ***getAvailableSlots()*** can output the correct timeslot when some slots are unavailable or have been booked.

```
@Test
public void timeSlotTest6() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots("2022-10-27");
    TimeSlot ts1 = new TimeSlot("12:00:00","13:00:00","tester1");
    tss.addSlot(ts1);
     LocalTime time = LocalTime.parse("12:30:00");
    assertEquals("tester1", tss.checkReserver(time));
}

@Test
public void timeSlotTest7() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots tss = new TimeSlots("2022-10-27");
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot ts1 = new TimeSlot("10:00:00","11:00:00","tester1");
    TimeSlot ts2 = new TimeSlot("11:25:00","13:00:00","tester2");
    tss.addSlot(ts1);
    tss.addSlot(ts2);

    assertEquals("13:00-19:00", tss.getAvailableSlots());
}

@Test
public void timeSlotTest8() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot timeslot = new TimeSlot("12:00:00","13:00:00","0");
    assertEquals(true, ts.addSlot(timeslot));
}
```

Test 8

Purpose: To check whether the function ***remove()*** can successfully delete the reserved timeslot.

```
@Test
public void timeSlotTest8() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot timeslot = new TimeSlot("12:00:00","13:00:00","0");
    assertEquals(true, ts.addSlot(timeslot));
}
```

Test 9

Purpose: Using a ***LocalTime*** as a variable to check the function ***hasReserved()*** can successfully identify whether a time point has been booked or not.

```
@Test
public void timeSlotTest9() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots();
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot timeslot1 = new TimeSlot("12:00:00","13:00:00","0");
    ts.addSlot(timeslot1);
    TimeSlot timeslot2 = new TimeSlot("12:00:00","13:00:00","0");
    assertEquals(false, ts.addSlot(timeslot2));
}
```

Test 10

Purpose: To check the function ***checkReservedStatus()***. If the timeslot is booked, then it will return -1.

```
@Test
public void timeSlotTest10() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots("2022-10-27");
    TimeSlot timeslot1 = new TimeSlot("12:00:00","13:00:00","0");
    ts.addSlot(timeslot1);
    LocalTime time = LocalTime.parse("12:30:00");
    assertEquals("0", ts.checkReserver(time));
}
```

Test 11

Purpose: To check the function ***checkReservedStatus()***. If the timeslot start time is reached, then it will return 0.

```
@Test
public void timeSlotTest11() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots("2022-10-27");
    TimeSlot timeslot1 = new TimeSlot("12:00:00","13:00:00","0");
    ts.addSlot(timeslot1);
    LocalTime time = LocalTime.parse("13:30:00");
    assertEquals(null, ts.checkReserver(time));
}
```

Test 12

Purpose: To check the function ***getAvailableSlots()***.

```
@Test
public void timeSlotTest12() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots("2022-10-27");
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot timeslot1 = new TimeSlot("10:00:00","11:00:00","5");
    TimeSlot timeslot2 = new TimeSlot("11:35:00","13:00:00","4");
    ts.addSlot(timeslot1);
    ts.addSlot(timeslot2);
    assertEquals("11:00-11:35, 13:00-19:00", ts.getAvailableSlots());
}
```

Test 13

Purpose: To check the function ***hasReserved()*** without reservation.

```
@Test
public void timeSlotTest13() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots("2022-10-27");
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot timeslot1 = new TimeSlot("10:00:00","11:00:00","5");
    ts.addSlot(timeslot1);
    LocalTime time = LocalTime.parse("10:30:00");
    ts.remove(timeslot1);
    assertEquals(false, ts.hasReserved(time));
}
```

Test 14

Purpose: To check the function ***hasReserved()*** with reservation.

```
@Test
public void timeSlotTest14() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots("2022-10-27");
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot timeslot1 = new TimeSlot("10:00:00","11:00:00","5");
    TimeSlot timeslot2 = new TimeSlot("11:35:00","13:00:00","4");
    ts.addSlot(timeslot1);
    ts.addSlot(timeslot2);
    LocalTime time = LocalTime.parse("12:30:00");
    assertEquals(true, ts.hasReserved(time));
}
```

Test 15

Purpose: To check the function ***checkReservedStatus()*** while the reservation is expired.

```
@Test
public void timeSlotTest15() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots("2022-10-27");
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot timeslot = new TimeSlot("11:35:00","13:00:00","4");
    ts.addSlot(timeslot);
    LocalTime time = LocalTime.parse("15:30:00");
    assertEquals(-1, ts.checkReservedStatus(time));
}
```

Test 16

Purpose: To check the function ***checkReservedStatus()*** while the reservation is on time.

```
@Test
public void timeSlotTest16() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots("2022-10-27");
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot timeslot = new TimeSlot("11:35:00","13:00:00","4");
    ts.addSlot(timeslot);
    LocalTime time = LocalTime.parse("11:35:00");
    assertEquals(0, ts.checkReservedStatus(time));
}
```

Test 17

Purpose: To check the function ***checkReservedStatus()*** when it's not the reservation time.

```
@Test
public void timeSlotTest17() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots("2022-10-27");
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot timeslot1 = new TimeSlot("10:00:00","11:00:00","5");
    TimeSlot timeslot = new TimeSlot("11:35:00","13:00:00","4");
    ts.addSlot(timeslot1);
    ts.addSlot(timeslot);
    LocalTime time = LocalTime.parse("11:10:00");
    assertEquals(1, ts.checkReservedStatus(time));
}
```

Test 18

Purpose: To check the function ***checkValidReserver()*** with valid reservation time.

```
@Test
public void timeSlotTest18() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots("2022-10-27");
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot timeslot = new TimeSlot("11:35:00","13:00:00","4");
    ts.addSlot(timeslot);
    LocalTime time = LocalTime.parse("11:35:00");
    assertEquals(true, ts.checkValidReserver(time, "4"));
}
```

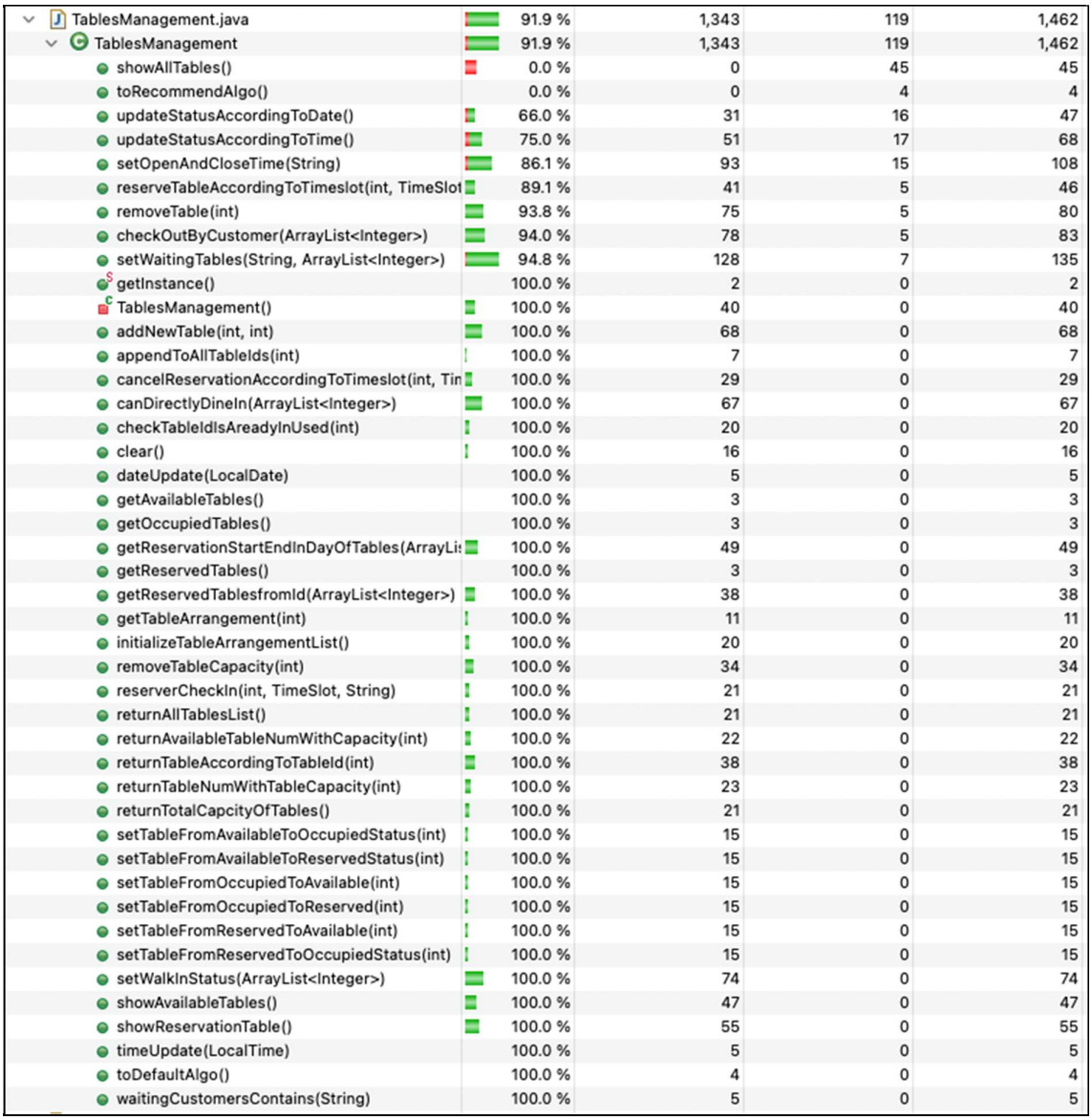
Test 19

Purpose: To check the function ***checkValidReserver()*** with invalid reservation time.

```
@Test
public void timeSlotTest19() throws ExTimeSlotInvalid, ExTimeFormatInvalid {
    TimeSlots ts = new TimeSlots("2022-10-27");
    TimeSlots.setOpenAndCloseTime("10:00:00", "19:00:00");
    TimeSlot timeslot = new TimeSlot("11:35:00","13:00:00","4");
    ts.addSlot(timeslot);
    LocalTime time = LocalTime.parse("15:35:00");
    assertEquals(false, ts.checkValidReserver(time, "4"));
}
```

3.2 TablesManagement.java

Branch Coverage:



TablesManagement.java	91.9 %	1,343	119	1,462
TablesManagement	91.9 %	1,343	119	1,462
showAllTables()	0.0 %	0	45	45
toRecommendAlgo()	0.0 %	0	4	4
updateStatusAccordingToDate()	66.0 %	31	16	47
updateStatusAccordingToTime()	75.0 %	51	17	68
setOpenAndCloseTime(String)	86.1 %	93	15	108
reserveTableAccordingToTimeslot(int, TimeSlot	89.1 %	41	5	46
removeTable(int)	93.8 %	75	5	80
checkOutByCustomer(ArrayList<Integer>)	94.0 %	78	5	83
setWaitingTables(String, ArrayList<Integer>)	94.8 %	128	7	135
getInstance()	100.0 %	2	0	2
TablesManagement()	100.0 %	40	0	40
addNewTable(int, int)	100.0 %	68	0	68
appendToAllTableIds(int)	100.0 %	7	0	7
cancelReservationAccordingToTimeslot(int, Tin	100.0 %	29	0	29
canDirectlyDineIn(ArrayList<Integer>)	100.0 %	67	0	67
checkTableIdIsAreadyInUsed(int)	100.0 %	20	0	20
clear()	100.0 %	16	0	16
dateUpdate(LocalDate)	100.0 %	5	0	5
getAvailableTables()	100.0 %	3	0	3
getOccupiedTables()	100.0 %	3	0	3
getReservationStartEndInDayOfTables(ArrayLis	100.0 %	49	0	49
getReservedTables()	100.0 %	3	0	3
getReservedTablesfromId(ArrayList<Integer>)	100.0 %	38	0	38
getTableArrangement(int)	100.0 %	11	0	11
initializeTableArrangementList()	100.0 %	20	0	20
removeTableCapacity(int)	100.0 %	34	0	34
reserverCheckIn(int, TimeSlot, String)	100.0 %	21	0	21
returnAllTablesList()	100.0 %	21	0	21
returnAvailableTableNumWithCapacity(int)	100.0 %	22	0	22
returnTableAccordingToTableId(int)	100.0 %	38	0	38
returnTableNumWithTableCapacity(int)	100.0 %	23	0	23
returnTotalCapcityOfTables()	100.0 %	21	0	21
setTableFromAvailableToOccupiedStatus(int)	100.0 %	15	0	15
setTableFromAvailableToReservedStatus(int)	100.0 %	15	0	15
setTableFromOccupiedToAvailable(int)	100.0 %	15	0	15
setTableFromOccupiedToReserved(int)	100.0 %	15	0	15
setTableFromReservedToAvailable(int)	100.0 %	15	0	15
setTableFromReservedToOccupiedStatus(int)	100.0 %	15	0	15
setWalkInStatus(ArrayList<Integer>)	100.0 %	74	0	74
showAvailableTables()	100.0 %	47	0	47
showReservationTable()	100.0 %	55	0	55
timeUpdate(LocalTime)	100.0 %	5	0	5
toDefaultAlgo()	100.0 %	4	0	4
waitingCustomersContains(String)	100.0 %	5	0	5

Because some functions inside Tablesmanagement are not suited for testing by directly inputting some fake value, or it is meaningless to test, for example, ***checkOutByCustomer()*** needs other classes, and some initial setup of the whole system, or ***getAvailableTables()*** only returns an ArrayList. Therefore the testing cases only cover the functions which are crucial and unique to the ***TablesManagement*** class itself.

Test 1: Testing the ***addNewTable()*** function

```
@Test
public void tableManagementTest1() {
    try {
        tm.addNewTable(0, 4);
        tm.addNewTable(0, 4);
    } catch (Exception e) {
        assertEquals(true, e instanceof ExTableIdAlreadyInUse);
    }
}
```

```
@Test
void AddTableWithExistID() {
    try {
        initialization.initialize();
        tm.addNewTable(1, 999);
    } catch (Exception e) {
        assertEquals(true, e instanceof ExTableIdAlreadyInUse);
        int capacity = tm.returnTableAccordingToTableId(1).getTableCapacity();
        assertEquals(false, capacity == 999);
    }
}
```

```
@Test
void AddRemoveTable() throws ExTableIdAlreadyInUse, ExTableNotExist, ExUnableToRemoveTable {
    initialization.initialize();
    tm.addNewTable(20, 4);
    Object t = tm.returnTableAccordingToTableId(20);
    assertEquals(true, t instanceof Table);
    assertEquals(true, ((Table) t).getTableCapacity() == 4);

    tm.removeTable(20);
    Object t1 = tm.returnTableAccordingToTableId(20);
    assertEquals(false, t1 instanceof Table);
}
```

Test 2: Testing Remove Table

```
@Test
void removeUnexistTable() throws ExTableIdAlreadyInUse, ExUnableToRemoveTable {
    initialization.initialize();
    try {
        tm.removeTable(20);
    } catch (Exception e) {
        assertEquals(true, e instanceof ExTableNotExist);
    }
}
```

Test 3: Testing the ***showAvailableTables*** function

```
@Test
public void tableManagementTest2() throws ExTableIdAlreadyInUse, ExTableNotExist, ExUnableToRemoveTable {
    tm.addNewTable(20, 8);
    tm.removeTable(20);
    tm.addNewTable(1, 4);
    String result = "\nBelow is the available tables: \n" + "Num of Available 4-Seats Table: 1 \n";
    assertEquals(result, tm.showAvailableTables());
}
```

```
@Test
public void tableManagementTest7() throws ExTableIdAlreadyInUse {
    tm.addNewTable(0, 4);
    String correct = "\nBelow is the available tables: \n" + "Num of Available 4-Seats Table: 1 \n";
    String result = tm.showAvailableTables();
    assertEquals(correct, result);
}
```

```java
@Test
public void tableManagementTest8() throws ExTableIdAlreadyInUse {
    tm.addNewTable(1, 4);
    String correct = "\nBelow is the available tables: \n" + "Num of Available 4-Seats Table: 1 \n";
    assertEquals(correct, tm.showAvailableTables());
}
```


```java
@Test
public void tableManagementTest8() throws ExTableIdAlreadyInUse {
    tm.addNewTable(1, 4);
    String correct = "\nBelow is the available tables: \n" + "Num of Available 4-Seats Table: 1 \n";
    assertEquals(correct, tm.showAvailableTables());
}
```


Test 4: Testing ***showReservationTable*** function

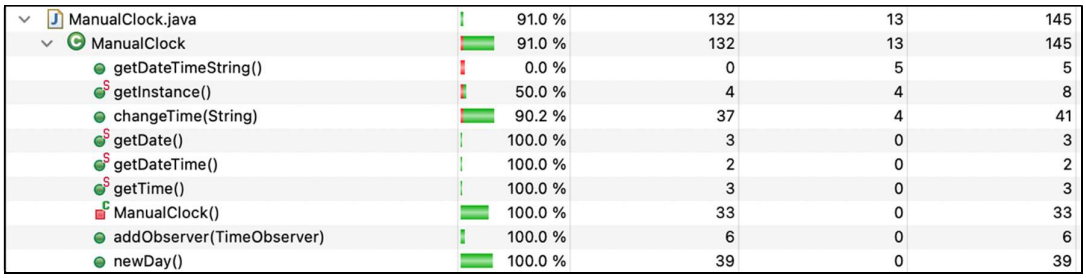
```java
@Test
public void testshowReservationTables()
        throws ExTableIdAlreadyInUse, ExPeopleNumExceedTotalCapacity, ExCustomersIdNotFound {
    initialization.initialize();
    TablesManagement.getInstance().clear();
    tm.addNewTable(11, 2);
    tm.addNewTable(12, 4);
    String Msg = "\nTable(s) for tomorrow reservation and available time slots: \n"
            + "2-Seats Table with ID of 11 is available tmr for the timeslots: 00:00-23:59 \n"
            + "4-Seats Table with ID of 12 is available tmr for the timeslots: 00:00-23:59 \n";
    String actualMsg = tm.showReservationTable();
    assertEquals(Msg, actualMsg);

}
```

Test 5: Testing ***returnTotalCapacityOfTables*** function

```java
@Test
public void testGetTotalCapacity()
        throws ExTableIdAlreadyInUse, ExPeopleNumExceedTotalCapacity, ExCustomersIdNotFound {
    tm.addNewTable(11, 2);
    tm.addNewTable(12, 4);

    assertEquals(6, tm.returnTotalCapcityOfTables());

}
```

3.3 ManualClock.java

Branch Coverage:



ManualClock.java	91.0 %	132	13	145
ManualClock	91.0 %	132	13	145
getDateTimeString()	0.0 %	0	5	5
getInstance()	50.0 %	4	4	8
changeTime(String)	90.2 %	37	4	41
getDate()	100.0 %	3	0	3
getDateTime()	100.0 %	2	0	2
getTime()	100.0 %	3	0	3
ManualClock()	100.0 %	33	0	33
addObserver(TimeObserver)	100.0 %	6	0	6
newDay()	100.0 %	39	0	39

Test 1

Purpose: To test whether the function ***getTime()*** can successfully get the current time in reality.

```
@Test
public void manualClockTest1() {
    ManualClock mc = ManualClock.getInstance();
    mc.changeTime("23:59");
    assertEquals(LocalTime.of(23, 59), ManualClock.getTime());
}
```

Test 2

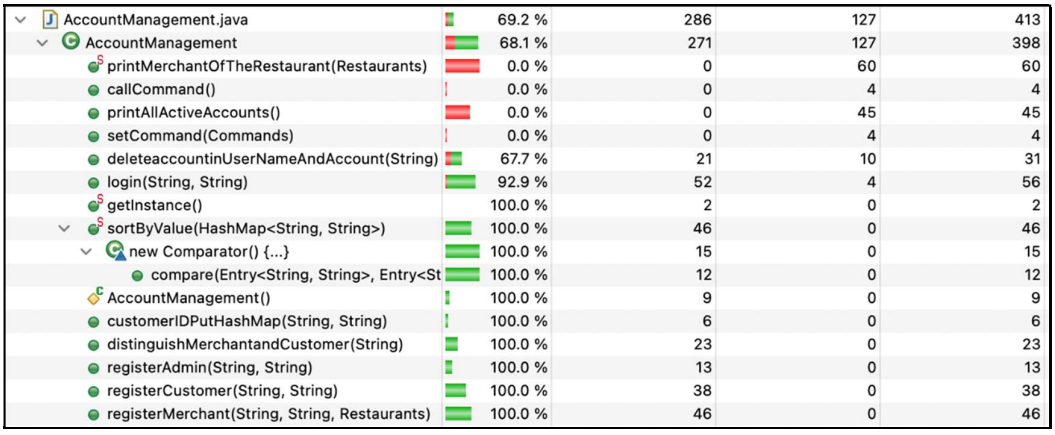
Purpose: To test whether the function ***getDate()*** can set a new date in the system according to the input value.

```
@Test
public void manualClockTest2() {
    ManualClock mc = ManualClock.getInstance();
    LocalDate first = LocalDate.now();
    mc.newDay();
    assertEquals(first.plusDays(1), ManualClock.getDate());
}
```

3.4 AccountManagement.java

- Login Account
- Delete Account
- Sorting Algorithm
- Distinguish User Module

Branch Coverage:



AccountManagement.java	69.2 %	286	127	413
AccountManagement	68.1 %	271	127	398
printMerchantOfTheRestaurant(Restaurants)	0.0 %	0	60	60
callCommand()	0.0 %	0	4	4
printAllActiveAccounts()	0.0 %	0	45	45
setCommand(Commands)	0.0 %	0	4	4
deleteaccountinUserNameAndAccount(String)	67.7 %	21	10	31
login(String, String)	92.9 %	52	4	56
getInstance()	100.0 %	2	0	2
sortByValue(HashMap<String, String>)	100.0 %	46	0	46
new Comparator() {...}	100.0 %	15	0	15
compare(Entry<String, String>, Entry<St	100.0 %	12	0	12
AccountManagement()	100.0 %	9	0	9
customerIDPutHashMap(String, String)	100.0 %	6	0	6
distinguishMerchantandCustomer(String)	100.0 %	23	0	23
registerAdmin(String, String)	100.0 %	13	0	13
registerCustomer(String, String)	100.0 %	38	0	38
registerMerchant(String, String, Restaurants)	100.0 %	46	0	46

Test 1: Login for a non-existing account:

```
@Test
void LoginAccountNotExist() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse {
    initialization.initialize();
    String username = "y3";
    String password = "t123";
    String cID = acMag.login(username, password);
    assertEquals(null, cID);
}
```

Test 2: Login for an existing customer account:

```
@Test
void LoginExistCustomer() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse {
    initialization.initialize();
    String username = "yinch33";
    String password = "t123";
    String cID = acMag.login(username, password);
    assertEquals("C0001", cID);
}
```

Test 3: Login for an existing merchant account:

```
@Test
void LoginExistCustomer() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse {
    initialization.initialize();
    String username = "yinch33";
    String password = "t123";
    String cID = acMag.login(username, password);
    assertEquals("C0001", cID);
}
```

Test 4: Login for an existing admin account:

```
@Test
void LoginAdmin() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse {
    initialization.initialize();
    String username = "admin";
    String password = "t123";
    String aID = acMag.login(username, password);
    assertEquals("A0001", aID);
}
```

Test 5: Login with the wrong password:

```
@Test
void LoginExistCustomerWithWrongPW() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse {
    initialization.initialize();
    String password = "wrongPassword";

    String username = "yinch33";
    String cID = acMag.login(username, password);
    assertEquals(null, cID);

    username = "KFCWorker";
    String mID = acMag.login(username, password);
    assertEquals(null, mID);

    username = "admin";
    String aID = acMag.login(username, password);
    assertEquals(null, aID);
}
```

Test 6: Delete user:

```
@Test
void deleteUser() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse {
    initialization.initialize();
    String username = "yinch33";
    String password = "t123";
    acMag.deleteaccountinUserNameAndAccount(username);
    String cID = acMag.login(username, password);
    assertEquals(null, cID);
}
```

Test 7: Sorting accounts

```
@Test
void testsortByValue() {
    HashMap<String, String> hashmap =new HashMap<String, String>();
    hashmap.put("Custom","C0001");
    hashmap.put("Admin","A0001");
    Set<Entry<String, String>> result=AccountManagement.sortByValue(hashmap);
    LinkedHashMap<String, String> sortedByValue = new LinkedHashMap<String, String>();
    sortedByValue.put("Admin", "A0001");
    sortedByValue.put("Custom", "C0001");
    Set<Entry<String, String>> smallset=sortedByValue.entrySet();
    AccountManagement.sortByValue(hashmap);
    assertEquals(smallset,result);
}
```

Test 8: Testing distinguish accounts function

```
@Test
void testdistinguishMerchantandCustomer1() {
    String userid="A0001";
    AccountManagement ac=AccountManagement.getInstance();
    AdminModule am=AdminModule.getInstance();
    assertEquals(am,ac.distinguishMerchantandCustomer(userid));
}

@Test
void testdistinguishMerchantandCustomer2() {
    String userid="C0001";
    AccountManagement ac=AccountManagement.getInstance();
    CustomerModule cm=CustomerModule.getInstance();
    assertEquals(cm,ac.distinguishMerchantandCustomer(userid));
}

@Test
void testdistinguishMerchantandCustomer3() {
    String userid="M0001";
    AccountManagement ac=AccountManagement.getInstance();
    MerchantModule cm=MerchantModule.getInstance();
    assertEquals(cm,ac.distinguishMerchantandCustomer(userid));
}

@Test
void testdistinguishMerchantandCustomer4() {
    String userid="S0001";
    AccountManagement ac=AccountManagement.getInstance();
    assertEquals(null,ac.distinguishMerchantandCustomer(userid));
}
```

3.5 Dish.java

Branch Coverage:

Dish.java	100.0 %	26	0	26
Dish	100.0 %	26	0	26
Dish(String, double)	100.0 %	9	0	9
getdishname()	100.0 %	3	0	3
getdishPrice()	100.0 %	3	0	3
setDishName(String)	100.0 %	4	0	4
setDishPrice(double)	100.0 %	4	0	4
toString()	100.0 %	3	0	3

Test 1: Edit dish name:

```
@Test
public void testEditDishName() {
    Dish dishA = new Dish("A Dish", 12.9);
    dishA.setDishName("Dish B");
    assertEquals("Dish B", dishA.getdishname());
}
```

Test 2: Edit dish price:

```
@Test
public void testEditDishPrice() {
    Dish dishA = new Dish("A Dish", 12.9);
    dishA.setDishPrice(129.9);
    assertEquals(129.9, dishA.getdishPrice());
}
```

3.6 Customer.java

Test 1: Get user name:

```
@Test
void testgetUsername() {
    Customers cus = new Customers("yinch33", "C0001");
    String result = cus.getUsername();
    assertEquals("yinch33", result);
}
```

Test 2: Get user ID:

```
@Test
void testgetUserId() {
    Customers cus = new Customers("yinch33", "C0001");
    String result = cus.getUserId();
    assertEquals("C0001", result);
}
```

4. Integration Test

4.1 White-Box Testing

The structure of the white-box integration testing:

- Login Module
  - Register
  - Login
  - Delete Account
- Admin Module
  - Set Open Close Time
  - Add / Remove Restaurant
  - Add / Remove Table
  - Check Reservation
- Customer Module
  - Dine In
  - Reserve
  - Cancel Reserve
  - Check Out
  - Update State
- Merchant Module
  - Modify Dish
- Table Management Module
- Table Arrangement Algorithm
  - Default Table Arrangement Algorithm
  - Recommended Arrangement

4.1.1 Login Module

- Register Part

Branch Coverage for ***GenerateCustomerId.java***:

GenerateCustomerId.java	100.0 %	45	0	45
GenerateCustomerId	100.0 %	45	0	45
getInstance()	100.0 %	8	0	8
GenerateCustomerId()	100.0 %	9	0	9
getNextId()	100.0 %	25	0	25

Branch Coverage for ***GenerateMerchantId.java***:

GenerateMerchantId.java	100.0 %	45	0	45
GenerateMerchantId	100.0 %	45	0	45
getInstance()	100.0 %	8	0	8
GenerateMerchantId()	100.0 %	9	0	9
getNextId()	100.0 %	25	0	25

Test 1: Register for new customer:

```
@Test
void registerNewCustomer() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse {
    initialization.initialize();
    String username = "test1";
    String password = "test1";
    acMag.registerCustomer(username, password);
    String cID = acMag.login(username, password);
    assertEquals(true, (db.matchCId(cID) instanceof Customers));
}
```

Test 2: Register for an existing customer:

```
@Test
void registerExistCustomer() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse {
    initialization.initialize();
    String username = "yinch33";
    String password = "t123";
    boolean result = acMag.registerCustomer(username, password);
    assertEquals(false, result);
}
```

Test 3: Register for new merchant:

```
@Test
void registerNewMerchant() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse {
    initialization.initialize();
    String username = "testWorker";
    String password = "t123";
    Restaurants r = new Restaurants("test");
    acMag.registerMerchant(username, password, r);
    String mID = acMag.login(username, password);
    assertEquals(true, (db.matchMId(mID) instanceof Merchants));
}
```

Test 4: Register for existing merchant:

```
@Test
void registerExistMerchant() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse {
    initialization.initialize();
    String username = "KFCWorker";
    String password = "t123";
    Restaurants r = new Restaurants("KFC");
    boolean result = acMag.registerMerchant(username, password, r);
    assertEquals(false, result);
}
```

4.1.2 Admin Module

4.1.2.1 Set Open and Close Time

Admin set open and close time without overlap:

```
@Test
void adminSetOpenCloseHourWithoutReservationOverlap()
        throws ExTableIdAlreadyInUse, ExUnableToSetOpenCloseTime, ExTimeFormatInvalid {
    initialization.initialize();
    String timeString = "09:00-21:00";
    boolean result = tm.setOpenAndCloseTime(timeString);
    assertEquals(true, result);
    assertEquals("09:00", TimeSlots.getOpenTime().toString());
    assertEquals("21:00", TimeSlots.getCloseTime().toString());
}
```

Admin set open and close time with reservation overlap:

```
@Test
void adminSetOpenCloseHourWithReservationOverlap() throws ExTableIdAlreadyInUse, ExTimeFormatInvalid,
        ExTableNotExist, ExTimeSlotAlreadyBeReserved, ExTimeSlotInvalid {
    initialization.initialize();
    String timeString = "09:00-21:00";
    ArrayList<Integer> table = new ArrayList<Integer>();
    table.add(1);
    Reservation r = new Reservation("C0001", "08:00-10:00", table, ManualClock.getDate());
    try {
        tm.setOpenAndCloseTime(timeString);
    } catch (Exception e) {
        assertEquals(true, e instanceof ExUnableToSetOpenCloseTime);
    } finally {
        assertEquals("00:00", TimeSlots.getOpenTime().toString());
        assertEquals("23:59", TimeSlots.getCloseTime().toString());
        r.cancel();
    }
}
```

4.1.2.2 Add / Remove Restaurant

```
@Test
void AddRemoveRestaurant() {
    //add
    class AddRestaurant_stub extends CommandAdminAddRestaurant {
        AddRestaurant_stub() {
            super();
        }
        public void addRestaurant(Restaurants r) {
            super.addRestaurant(r);
        }
    }
    AddRestaurant_stub cmdAdd = new AddRestaurant_stub();
    Restaurants r = new Restaurants("test");
    cmdAdd.addRestaurant(r);
    assertEquals(true, db.getListofRestaurants().contains(r));
    //remove
    class removeRestaurant_stub extends CommandAdminRemoveRestaurant {
        removeRestaurant_stub() {
            super();
        }
        public void removeRestaurant(Restaurants r) {
            super.deleteRestaurant(r);
        }
    }
    removeRestaurant_stub cmdRemove = new removeRestaurant_stub();
    cmdRemove.removeRestaurant(r);
    assertEquals(false, db.getListofRestaurants().contains(r));
}
```

4.1.2.3 Add / Remove Table

Test 1: Add existing table:

```
@Test
void AddTableWithExistID() {
    try {
        initialization.initialize();
        tm.addNewTable(1,999);
    } catch (Exception e) {
        assertEquals(true, e instanceof ExTableIdAlreadyInUse);
        int capacity = tm.returnTableAccordingToTableId(1).getTableCapacity();
        assertEquals(false, capacity == 999);
    }
}
```

Test 2: Add or remove table with valid input ID:

```
@Test
void AddRemoveTable() throws ExTableIdAlreadyInUse, ExTableNotExist, ExUnableToRemoveTable {
    initialization.initialize();
    tm.addNewTable(20,4);
    Object t = tm.returnTableAccordingToTableId(20);
    assertEquals(true, t instanceof Table);
    assertEquals(true, ((Table)t).getTableCapacity() == 4);

    tm.removeTable(20);
    Object t1 = tm.returnTableAccordingToTableId(20);
    assertEquals(false, t1 instanceof Table);
}
```

Test 3: Remove table which is non-existing:

```
@Test
void removeUnexistTable() throws ExTableIdAlreadyInUse, ExUnableToRemoveTable {
    initialization.initialize();
    try {
        tm.removeTable(20);
    } catch (Exception e) {
        assertEquals(true, e instanceof ExTableNotExist);
    }
}
```

4.1.2.4 Check Reservation

```
@Test
void checkReservation() throws ExCustomersIdNotFound, ExTableNotExist, ExTimeSlotAlreadyBeReserved,
        ExTimeSlotInvalid, ExTimeFormatInvalid, ExTableIdAlreadyInUse {
    initialization.initialize();
    String username = "tester";
    String password = "tester";
    acMag.registerCustomer(username, password);
    String cID = acMag.login(username, password);
    Customers tester = db.matchCId(cID);
    ArrayList<Integer> desiredTable = new ArrayList<Integer>();
    desiredTable.add(1);
    Reservation testR = new Reservation(cID, "12:00-13:00", desiredTable, ManualClock.getDate());
    tester.setReserve(testR);

    String expected = "";
    expected += String.format("\nThe upcoming reservation of Customer %s for %s is: ", cID, "tomorrow");
    expected += "\n[1] [Table with ID: 1] [Time Slot: 12:00-13:00]";

    assertEquals(expected, tester.getReserveInfo());

    tester.getReservation().cancel();
    tester.clearReservation();
    assertEquals(null, tester.getReservation());
}
```

4.1.3 Customer Module

4.1.3.1 Dine in

Test 1: Test Successfully Dine in operation

```
@Test
void testDineinOperation() {
    class customerstub extends Customers {
        public customerstub(String username, String userid) {
            super(username, userid);
        }

        public boolean checkisReserved() {
            return false;
        }
    }

    class cmd_stub extends CommandCustomerDineIn {
        public cmd_stub(Customers customer) {
            super(customer);
        }

        public boolean dineInOperation() {
            return super.dineInOperation();
        }
    }
    Customers cus = new customerstub("yinch33", "C0001");
    cmd_stub customerdinein = new cmd_stub(cus);
    boolean result = customerdinein.dineInOperation();
    assertEquals(false, result);
}
```

```
@Test
void testDirectWalkin() {
    Customers cus = new Customers("yinch33", "C0001");
    class cmd_stub extends CommandCustomerDineIn {
        public cmd_stub(Customers customer) {
            super(customer);
        }

        public boolean directWalkIn(ArrayList<Integer> array) {
            return super.directWalkIn(array);
        }
    }
    cmd_stub customerdinein = new cmd_stub(cus);
    ArrayList<Integer> array = new ArrayList<Integer>();
    array.add(2);
    boolean result = customerdinein.directWalkIn(array);
    assertEquals(true, result);
}
```

Test 2: Testing check in operation

Check in when walk-in: Add Occupied Table Id to customer

```
@Test
void testaddCheckInInfo() {
    Customers cus = new Customers("yinch33", "C0001");
    class cmd_stub extends CommandCustomer {
        public cmd_stub(Customers customer) {
            super(customer);
        }

        @Override
        public void exe() throws ExUnableToSetOpenCloseTime, ExTableIdAlreadyInUse, ExTableNotExist,
                ExTimeSlotNotReservedYet, ExCustomersIdNotFound, ExTimeSlotAlreadyBeReserved {
        }
    }
    Commands cmd = new cmd_stub(cus);
    ArrayList<Integer> array = new ArrayList<Integer>();
    array.add(1);
    CommandCustomerDineIn.addCheckInInfo(array);
    assertEquals(array, cus.getOccupiedTableId());
}
```

4.1.3.2 Reserve

Branch Coverage:

Reservation.java	93.0 %	226	17	243
Reservation	93.0 %	226	17	243
reserve(ArrayList<Integer>, TimeSlot)	63.0 %	29	17	46
Reservation(String, String, ArrayList<Integer>,	100.0 %	38	0	38
cancel()	100.0 %	25	0	25
checkValid(LocalDate)	100.0 %	18	0	18
computeReserveString()	100.0 %	62	0	62
getReservedTableIDs()	100.0 %	3	0	3
getReservedTimeSlot()	100.0 %	3	0	3
getReserveString()	100.0 %	5	0	5
toString()	100.0 %	40	0	40

Test 1: Testing ***toString()*** function

```
@Test
void testReservationToString1() throws ExTableNotExist, ExTimeSlotAlreadyBeReserved,
    ExTimeSlotInvalid, ExTimeFormatInvalid {

    ArrayList<Integer> table = new ArrayList<Integer>();
    Reservation r=new Reservation("C0001", "08:00-10:00", table, ManualClock.getDate());
    String result=r.toString();
    String s="\n[C0001] Error: Reservation not made.\n";
    assertEquals(s,result);
}

@Test
void testReservationToString2() throws ExTableNotExist, ExTimeSlotAlreadyBeReserved,
    ExTimeSlotInvalid, ExTimeFormatInvalid, ExTableIdAlreadyInUse {

    initialization.initialize();
    ArrayList<Integer> table = new ArrayList<Integer>();
    table.add(1);
    Reservation r = new Reservation("C0001", "08:00-10:00", table, ManualClock.getDate());
    String result = r.toString();
    String s = "\n[C0001] Reservation made, Reserved tables: "
            +"\n[1] [Table with ID: 1] [Time Slot: 08:00-10:00]";
    assertEquals(s,result);
}
```

Test 2: Testing to print out the reservation reminder of the customer

```
@Test
public void testCustomerGetReservationReminder() throws ExTableIdAlreadyInUse, ExTableNotExist,
    ExTimeSlotAlreadyBeReserved, ExTimeSlotInvalid, ExTimeFormatInvalid {

    initialization.initialize();
    TablesManagement.getInstance().clear();

    tm.addNewTable(11, 2);
    tm.addNewTable(12, 4);
    Customers c= new Customers("tester","C0008");
    ArrayList<Integer> reserveTd=new ArrayList<Integer>();
    reserveTd.add(11);
    reserveTd.add(12);
    LocalDate thisday = LocalDate.parse("2022-10-27");
    Reservation r=new Reservation("C0008","11:00-12:00",reserveTd,thisday);
    c.setReserve(r);
    String Msg= String.format("\nReminder: You have a reservation for %s: ","tomorrow");
    Msg += "\n[1] [Table with ID: 11] [Time Slot: 11:00-12:00]";
    Msg += "\n[2] [Table with ID: 12] [Time Slot: 11:00-12:00]";


    String actualMsg=c.getReserveReminder();
    assertEquals(Msg,actualMsg);
}
```

Test 3: Testing ***checkValid*** function

```
@Test
public void testcheckValid1() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse, ExTableNotExist,
        ExTimeSlotAlreadyBeReserved, ExTimeSlotInvalid, ExTimeFormatInvalid {
    initialization.initialize();

    Customers c = db.matchCId("C0001");
    LocalDate currDate = LocalDate.of(2022, 12, 4);
    String customerID = "C0001";
    String timeSlotString = "12:00-13:00";
    LocalDate date = LocalDate.of(2022, 12, 1);

    ArrayList<Integer> desiredTableIDs = new ArrayList<>();
    desiredTableIDs.add(1);
    desiredTableIDs.add(3);
    desiredTableIDs.add(7);

    Reservation r = new Reservation(customerID, timeSlotString, desiredTableIDs, date);

    assertEquals(-1, r.checkValid(currDate));
}
```

```
@Test
public void testcheckValid2() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse, ExTableNotExist,
        ExTimeSlotAlreadyBeReserved, ExTimeSlotInvalid, ExTimeFormatInvalid {
    initialization.initialize();

    Customers c = db.matchCId("C0001");
    LocalDate currDate = LocalDate.of(2022, 12, 1);
    String customerID = "C0001";
    String timeSlotString = "12:00-13:00";
    LocalDate date = LocalDate.of(2022, 12, 1);

    ArrayList<Integer> desiredTableIDs = new ArrayList<>();
    desiredTableIDs.add(1);
    desiredTableIDs.add(3);
    desiredTableIDs.add(7);

    Reservation r = new Reservation(customerID, timeSlotString, desiredTableIDs, date);

    assertEquals(0, r.checkValid(currDate));
}
```

```
@Test
public void testcheckValid3() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse, ExTableNotExist,
        ExTimeSlotAlreadyBeReserved, ExTimeSlotInvalid, ExTimeFormatInvalid {
    initialization.initialize();

    LocalDate currDate = LocalDate.of(2022, 11, 30);
    String customerID = "C0001";
    String timeSlotString = "12:00-13:00";
    LocalDate date = LocalDate.of(2022, 12, 1);

    ArrayList<Integer> desiredTableIDs = new ArrayList<>();
    desiredTableIDs.add(1);
    desiredTableIDs.add(3);
    desiredTableIDs.add(7);

    Reservation r = new Reservation(customerID, timeSlotString, desiredTableIDs, date);

    assertEquals(1, r.checkValid(currDate));
}
```

Test 4: Testing to obtain the reservation information of the customer

```
@Test
public void testCustomerGetReservationInfo() throws ExTableIdAlreadyInUse, ExTableNotExist,
    ExTimeSlotAlreadyBeReserved, ExTimeSlotInvalid, ExTimeFormatInvalid {

    initialization.initialize();
    TablesManagement.getInstance().clear();
    tm.addNewTable(11, 2);
    tm.addNewTable(12, 4);
    Customers c= new Customers("tester","C0008");
    ArrayList<Integer> reserveTd=new ArrayList<Integer>();
    reserveTd.add(11);
    reserveTd.add(12);
    LocalDate thisday = LocalDate.parse("2022-10-27");
    Reservation r=new Reservation("C0008","11:00-12:00",reserveTd,thisday);
    c.setReserve(r);
    String Msg= String.format("\nThe upcoming reservation of Customer %s for %s is: ", "C0008","tomorrow");
    Msg += "\n[1] [Table with ID: 11] [Time Slot: 11:00-12:00]";
    Msg += "\n[2] [Table with ID: 12] [Time Slot: 11:00-12:00]";


    String actualMsg=c.getReserveInfo();
    assertEquals(Msg,actualMsg);
}
```

Test 5: Testing successfully make a reservation

```
@Test
public void testReserveCreated() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse {
    initialization.initialize();

    Customers c = db.matchCId("C0001");
    CommandCustomerReservation cmdCustomerReserve = new CommandCustomerReservation(c);

    String customerID = "C0001";
    String timeSlotString = "12:00-13:00";
    LocalDate date = ManualClock.getDate();

    ArrayList<Integer> desiredTableIDs = new ArrayList<>();
    desiredTableIDs.add(1);
    desiredTableIDs.add(3);
    desiredTableIDs.add(7);

    Object r = cmdCustomerReserve.setReserve(customerID, timeSlotString, desiredTableIDs, date);

    assertEquals(true, r instanceof Reservation);

}
```

Test 6: Test ***checkIsReserved*** function

```
@Test
void testcheckisReserved1() {
    class customerstub extends Customers {
        private Reservation reserve = null;

        public customerstub(String username, String userid) {
            super(username, userid);

        }

    }
    customerstub cus = new customerstub("yinch33", "C0001");
    boolean result = cus.checkisReserved();
    assertEquals(false, result);
}
```

Test 7: Testing make reservation unsuccessful

```
@Test
public void testReserveTimeSlotInvalid() throws ExTableIdAlreadyInUse, ExCustomersIdNotFound {
    initialization.initialize();

    Customers c = db.matchCId("C0001");
    CommandCustomerReservation cmdCustomerReserve = new CommandCustomerReservation(c);

    String customerID = "C0001";
    String timeSlotString = "12:00-17:00";
    LocalDate date = ManualClock.getDate();

    ArrayList<Integer> desiredTableIDs = new ArrayList<>();
    desiredTableIDs.add(1);
    desiredTableIDs.add(3);
    desiredTableIDs.add(7);

    Object r = cmdCustomerReserve.setReserve(customerID, timeSlotString, desiredTableIDs, date);
    assertEquals(null, r);
}
```

```
@Test
public void testReserveTimeFormatInvalid() throws ExTableIdAlreadyInUse, ExCustomersIdNotFound {
    initialization.initialize();

    Customers c = db.matchCId("C0001");
    CommandCustomerReservation cmdCustomerReserve = new CommandCustomerReservation(c);

    String customerID = "C0001";
    String timeSlotString = "12:00 - 13:00";
    LocalDate date = ManualClock.getDate();

    ArrayList<Integer> desiredTableIDs = new ArrayList<>();
    desiredTableIDs.add(1);
    desiredTableIDs.add(3);
    desiredTableIDs.add(7);

    Object r = cmdCustomerReserve.setReserve(customerID, timeSlotString, desiredTableIDs, date);

    assertEquals(null, r);
}
```

```
@Test
public void testReserve() throws ExTableNotExist, ExTimeSlotAlreadyBeReserved, ExTimeSlotInvalid,
        ExTimeFormatInvalid, ExCustomersIdNotFound, ExTableIdAlreadyInUse {
    initialization.initialize();

    Customers c = db.matchCId("C0001");

    String customerID = "C0001";
    String timeSlotString = "12:00-13:00";
    LocalDate date = ManualClock.getDate();

    ArrayList<Integer> desiredTableIDs = new ArrayList<>();
    desiredTableIDs.add(1);
    desiredTableIDs.add(3);
    desiredTableIDs.add(7);

    Reservation r = new Reservation(customerID, timeSlotString, desiredTableIDs, date);
    c.setReserve(r);

    assertEquals(desiredTableIDs, r.getReservedTableIDs());

    assertEquals("12:00", c.getReservation().getReservedTimeSlot().getStart().toString());
    assertEquals("13:00", c.getReservation().getReservedTimeSlot().getEnd().toString());
    assertEquals("C0001", c.getReservation().getReservedTimeSlot().getCustomerID());

}
```

4.1.3.3 Cancel Reserve

Test 1: Cancel existing reservation:

```
@Test
public void testCancelReservation()
        throws ExTableIdAlreadyInUse, ExCustomersIdNotFound, ExUnableToSetOpenCloseTime, ExTableNotExist,
        ExTimeSlotNotReservedYet, ExTimeSlotAlreadyBeReserved, ExTimeSlotInvalid, ExTimeFormatInvalid {
    initialization.initialize();

    Customers c = db.matchCId("C0001");
    CommandCustomerCancelReservation cmdCancelReserve = new CommandCustomerCancelReservation(c);

    String customerID = "C0001";
    String timeSlotString = "12:00-13:00";
    LocalDate date = ManualClock.getDate();

    ArrayList<Integer> desiredTableIDs = new ArrayList<>();
    desiredTableIDs.add(1);
    desiredTableIDs.add(3);
    desiredTableIDs.add(7);

    Reservation r = new Reservation(customerID, timeSlotString, desiredTableIDs, date);
    c.setReserve(r);

    assertEquals(true, c.getReservation() instanceof Reservation);

    cmdCancelReserve.exe();

    assertEquals(null, c.getReservation());
}
```

Test 2: Cancel non-existing reservation:

```
@Test
public void testCancelReservationNotReserved() throws ExCustomersIdNotFound, ExTableIdAlreadyInUse,
        ExTableNotExist, ExTimeSlotNotReservedYet, ExUnableToSetOpenCloseTime {
    initialization.initialize();
    Customers c = db.matchCId("C0001");
    c.clearReservation();
    assertEquals(null, c.getReservation());
    CommandCustomerCancelReservation cmdCancelReserve = new CommandCustomerCancelReservation(c);
    cmdCancelReserve.exe();
}
```

4.1.3.4 Ordering

Test 1: Update the pending orders when not confirming order.

```
@Test
public void testCustomerUpdateOrders() throws ExTableIdAlreadyInUse, ExTableNotExist,
    ExTimeSlotAlreadyBeReserved, ExTimeSlotInvalid, ExTimeFormatInvalid {

    initialization.initialize();
    TablesManagement.getInstance().clear();
    tm.addNewTable(11, 2);
    tm.addNewTable(12, 4);

    Customers c= new Customers("tester","C0008");
    Dish d=new Dish("H",35);
    Restaurants r= new Restaurants("H");

    c.addPendingOrder(d);
    c.updateOrder(r);

    Multimap<Dish, Restaurants> DishToRestaurant =ArrayListMultimap.create();
    DishToRestaurant.put(d, r);
    boolean res=c.isDishToRestaurant(DishToRestaurant);

    assertEquals(true,res);

    Dish d1=new Dish("M",35);
    Restaurants r1= new Restaurants("M");
    c.addPendingOrder(d1);
    c.updateOrder(r1);
    DishToRestaurant.put(d1, r1);
}
```

4.1.3.5 Check Out

```
@Test
void testCheckoutbyCustomer() throws ExTableIdAlreadyInUse {
    initialization.initialize();
    TablesManagement tm=TablesManagement.getInstance();
    ArrayList<Integer> array=new ArrayList<Integer>();
    array.add(1);
    tm.checkOutByCustomer(array);
    assertEquals(true, tm.getAvailableTables().contains(tm.returnTableAccordingToTableId(1)));
    assertEquals(false, tm.getOccupiedTables().contains(tm.returnTableAccordingToTableId(1)));
}
```

```
@Test
public void testCheckOutByCustomer() throws ExTableIdAlreadyInUse, ExPeopleNumExceedTotalCapacity, ExCustomersIdNotFound{
    tm.addNewTable(11, 2);
    tm.addNewTable(12, 4);
    String username = "tester1";
    String password = "tester1";
    acMag.registerCustomer(username, password);
    String cID = acMag.login(username, password);
    ArrayList<Integer> defaultArrangment1=new ArrayList<>();
    tm.toDefaultAlgo();
    defaultArrangment1=tm.getTableArrangement(6);
    tm.setWalkInStatus(defaultArrangment1);
    ArrayList<Integer> defaultArrangment2=new ArrayList<>();
    tm.toDefaultAlgo();
    defaultArrangment2=tm.getTableArrangement(6);
    Customers c = db.matchCId(cID);
    class cmd_stub extends CommandCustomer {
        public cmd_stub(Customers customer) {
            super(customer);
        }

        @Override
        public void exe() throws ExUnableToSetOpenCloseTime, ExTableIdAlreadyInUse, ExTableNotExist,
                ExTimeSlotNotReservedYet, ExCustomersIdNotFound, ExTimeSlotAlreadyBeReserved {
        }
    }
   Commands cmd = new cmd_stub(c);
   tm.setWaitingTables(cID,defaultArrangment2);
   ArrayList<Integer> allId=new ArrayList<Integer>();
   allId.add(11);
   allId.add(12);
   tm.checkOutByCustomer(allId);
   boolean containsC = tm.waitingCustomersContains(cID);
   assertEquals(true, containsC);
}
```

4.1.3.6 Update State

Test 1: Test to update the state of the customer based on bill amount

```
@Test
public void testCustomerUpdateState() throws ExTableIdAlreadyInUse, ExTableNotExist,
    ExTimeSlotAlreadyBeReserved, ExTimeSlotInvalid, ExTimeFormatInvalid {

    Customers c= new Customers("tester","C0008");
    c.setBillAmount(89);
    c.updateState();
    CustomerState cs=new CustomerSuperVIPstate();
    assertEquals(cs.getdiscount(),c.getState().getdiscount());
    Customers c1= new Customers("tester1","C0009");
    c1.setBillAmount(85);
    c1.updateState();
    CustomerState cs1=new CustomerVIPstate();
    assertEquals(cs1.getdiscount(),c1.getState().getdiscount());
}
```

4.1.4 Merchant Module

4.1.4.1 Modify Dish

Test 1: Find dish:

```
@Test
public void testFindDish() {
    String dishName;
    double dishPrice;
    dishName = "ABC Dish";
    dishPrice = 199.9;
    merchant.addtoMenu(dishName, dishPrice);;

    assertEquals("ABC Dish", merchant.findDish(dishName).toString());
}
```

Test 2: Add dish:

```
@Test
public void testAddDish() {
    String dishName = "ABC Dish";
    double dishPrice = 199.9;
    merchant.addtoMenu(dishName, dishPrice);
    ArrayList<Dish> menu = merchant.getRestaurantOwned().getMenu();
    Dish dish = merchant.getRestaurantOwned().getDishbyName(dishName);
    assertEquals(true, menu.contains(dish));
}
```

```
@Test
public void testAddDishtoMenu() {
    Dish dishA = new Dish("A Dish", 12.9);
    Restaurants r = merchant.getRestaurantOwned();
    r.adddishtoMenu(dishA);
    assertEquals(true, merchant.getRestaurantOwned().getMenu().contains(dishA));
}
```

Test 3: Remove existing dish:

```
@Test
public void testAddRemoveDishfromMenu() {
    Dish dishA = new Dish("A Dish", 12.9);
    Restaurants r = merchant.getRestaurantOwned();
    r.adddishtoMenu(dishA);
    assertEquals(true, merchant.getRestaurantOwned().getMenu().contains(dishA));

    r.deletedishfromMenu(dishA);
    assertEquals(false, merchant.getRestaurantOwned().getMenu().contains(dishA));
}
```

Test 4: Remove non-existing dish:

```
@Test
public void testRemoveNotExistDishfromMenu() {
    Dish dishA = new Dish("A Dish", 12.9);
    Restaurants r = merchant.getRestaurantOwned();
    r.deletedishfromMenu(dishA);

    assertEquals(false, merchant.getRestaurantOwned().getMenu().contains(dishA));
}
```

Test 5: Get non-existing dish:

```
@Test
public void testGetDishByNameNotExist() {
    Restaurants r = merchant.getRestaurantOwned();
    assertEquals(null, r.getDishbyName("X Dish"));
}
```

Test 6: Get updated menu:

```
@Test
public void testUpdateMenu() {
    Restaurants r = merchant.getRestaurantOwned();
    ArrayList<Dish> newMenu = new ArrayList<>();

    Dish dishA = new Dish("A Dish", 100);
    Dish dishB = new Dish("B Dish", 200.5);
    Dish dishC = new Dish("C Dish", 199.5);

    newMenu.add(dishA);
    newMenu.add(dishB);
    newMenu.add(dishC);

    r.updateMenu(newMenu);
    assertEquals(newMenu, r.getMenu());
}
```

Test 7: Get total price:

```
@Test
public void testCountPrice() {
    Restaurants r = merchant.getRestaurantOwned();

    Dish dishA = new Dish("A Dish", 100);
    Dish dishB = new Dish("B Dish", 200.5);
    Dish dishC = new Dish("C Dish", 199.5);

    r.adddishtoMenu(dishA);
    r.adddishtoMenu(dishB);
    r.adddishtoMenu(dishC);

    ArrayList<Dish> orders = new ArrayList<>();
    orders.add(dishA);
    orders.add(dishB);
    orders.add(dishC);

    assertEquals(500, r.countPrice(orders));
}
```

4.1.5 Table Management

Test 1: Table Status Handling

```
@Test
public void testSetTableFromAvailableToOccupied() throws ExTableIdAlreadyInUse {
    tm.addNewTable(11, 2);
    tm.addNewTable(12, 4);
    Table t1 = tm.returnTableAccordingToTableId(11);
    Table t2 = tm.returnTableAccordingToTableId(12);
    tm.setTableFromAvailableToOccupiedStatus(11);
    tm.setTableFromAvailableToOccupiedStatus(12);
    ArrayList<Table> occupiedTableList = tm.getOccupiedTables();
    boolean testRes1 = occupiedTableList.contains(t1);
    boolean testRes2 = occupiedTableList.contains(t2);
    assertEquals(true, testRes1);
    assertEquals(true, testRes2);
}
```

```
@Test
public void testSetTableFromOccupiedToAvailable() throws ExTableIdAlreadyInUse {
    tm.addNewTable(11, 2);
    tm.addNewTable(12, 4);
    tm.setTableFromAvailableToOccupiedStatus(11);
    tm.setTableFromAvailableToOccupiedStatus(12);
    Table t1 = tm.returnTableAccordingToTableId(11);
    Table t2 = tm.returnTableAccordingToTableId(12);
    tm.setTableFromOccupiedToAvailable(11);
    tm.setTableFromOccupiedToAvailable(12);
    ArrayList<Table> availableTableList = tm.getAvailableTables();
    boolean testRes1 = availableTableList.contains(t1);
    boolean testRes2 = availableTableList.contains(t2);
    assertEquals(true, testRes1);
    assertEquals(true, testRes2);
}
```

```
@Test
public void testSetTableFromAvailableToReserved() throws ExTableIdAlreadyInUse {
    tm.addNewTable(11, 2);
    tm.addNewTable(12, 4);
    Table t1 = tm.returnTableAccordingToTableId(11);
    Table t2 = tm.returnTableAccordingToTableId(12);
    tm.setTableFromAvailableToReservedStatus(11);
    tm.setTableFromAvailableToReservedStatus(12);
    ArrayList<Table> reservedTableList = tm.getReservedTables();
    boolean testRes1 = reservedTableList.contains(t1);
    boolean testRes2 = reservedTableList.contains(t2);
    assertEquals(true, testRes1);
    assertEquals(true, testRes2);
}
```

```
@Test
public void testSetTableFromReservedToAvailable() throws ExTableIdAlreadyInUse {
    tm.addNewTable(11, 2);
    tm.addNewTable(12, 4);
    tm.setTableFromAvailableToReservedStatus(11);
    tm.setTableFromAvailableToReservedStatus(12);
    Table t1 = tm.returnTableAccordingToTableId(11);
    Table t2 = tm.returnTableAccordingToTableId(12);
    tm.setTableFromReservedToAvailable(11);
    tm.setTableFromReservedToAvailable(12);
    ArrayList<Table> availableTableList = tm.getAvailableTables();
    boolean testRes1 = availableTableList.contains(t1);
    boolean testRes2 = availableTableList.contains(t2);
    assertEquals(true, testRes1);
    assertEquals(true, testRes2);
}
```

```java
@Test
public void testSetTableFromReservedToOccupied() throws ExTableIdAlreadyInUse {
    tm.addNewTable(11, 2);
    tm.addNewTable(12, 4);
    Table t1 = tm.returnTableAccordingToTableId(11);
    Table t2 = tm.returnTableAccordingToTableId(12);
    tm.setTableFromAvailableToReservedStatus(11);
    tm.setTableFromAvailableToReservedStatus(12);
    ArrayList<Table> reservedTableList = tm.getReservedTables();
    boolean testRes1 = reservedTableList.contains(t1);
    boolean testRes2 = reservedTableList.contains(t2);
    assertEquals(true, testRes1);
    assertEquals(true, testRes2);
    tm.setTableFromReservedToOccupiedStatus(11);
    tm.setTableFromReservedToOccupiedStatus(12);
    ArrayList<Table> occupiedTableList = tm.getOccupiedTables();
    boolean testRes3 = occupiedTableList.contains(t1);
    boolean testRes4 = occupiedTableList.contains(t2);
    assertEquals(true, testRes3);
    assertEquals(true, testRes4);
}
```

```java
@Test
public void testSetTableFromOccupiedToReserved() throws ExTableIdAlreadyInUse {
    tm.addNewTable(11, 2);
    tm.addNewTable(12, 4);
    tm.setTableFromAvailableToOccupiedStatus(11);
    tm.setTableFromAvailableToOccupiedStatus(12);
    Table t1 = tm.returnTableAccordingToTableId(11);
    Table t2 = tm.returnTableAccordingToTableId(12);
    ArrayList<Table> occupiedTableList = tm.getOccupiedTables();
    boolean testRes3 = occupiedTableList.contains(t1);
    boolean testRes4 = occupiedTableList.contains(t2);
    assertEquals(true, testRes3);
    assertEquals(true, testRes4);
    tm.setTableFromOccupiedToReserved(11);
    tm.setTableFromOccupiedToReserved(12);
    ArrayList<Table> reservedTableList = tm.getReservedTables();
    boolean testRes1 = reservedTableList.contains(t1);
    boolean testRes2 = reservedTableList.contains(t2);
    assertEquals(true, testRes1);
    assertEquals(true, testRes2);
}
```

Test 2: Reserve Table according to Timeslot

```java
@Test
public void tableManagementTest3() throws ExTableIdAlreadyInUse,
        ExTimeSlotInvalid, ExTimeFormatInvalid, ExTableNotExist, ExTimeSlotAlreadyBeReserved {
    tm.addNewTable(0, 4);
    TimeSlot ts = new TimeSlot("12:00:00", "13:00:00", "0");
    boolean result = tm.reserveTableAccordingToTimeslot(0, ts);
    assertEquals(true, result);

}
```

```java
@Test
public void tableManagementTest4()
        throws ExTimeSlotInvalid, ExTimeFormatInvalid, ExTableNotExist, ExTableIdAlreadyInUse {
    initialization.initialize();
    TimeSlot ts = new TimeSlot("12:30:00", "13:00:00", "1");
    try {
        tm.reserveTableAccordingToTimeslot(1, ts);
        tm.reserveTableAccordingToTimeslot(1, ts);
    } catch (Exception e) {
        assertEquals(true, e instanceof ExTimeSlotAlreadyBeReserved);
    }
}
```

Test 3: Cancel Reservation according to timeslot

```
@Test
public void tableManagementTest5() throws ExTimeSlotInvalid, ExTimeFormatInvalid, ExTableNotExist,
        ExTimeSlotAlreadyBeReserved, ExTableIdAlreadyInUse {
    initialization.initialize();
    TimeSlot ts = new TimeSlot("12:00:00", "13:00:00", "1");
    try {
        tm.cancelReservationAccordingToTimeslot(1, ts);
    } catch (Exception e) {
        assertEquals(true, e instanceof ExTimeSlotNotReservedYet);
    }

}
```

Test 4: Find number of available table corresponding to the capacity

```
@Test
public void tableManagementTest6() throws ExTableIdAlreadyInUse {
    tm.addNewTable(1, 4);
    tm.addNewTable(2, 5);
    assertEquals(1, tm.returnAvailableTableNumWithCapacity(4));
}
```

Test 5: Operation for reserved customer to dine in

```
@Test
public void tableManagementTest9_reservedDineIn() throws ExTableIdAlreadyInUse, ExCustomersIdNotFound,
        ExTableNotExist, ExTimeSlotAlreadyBeReserved, ExTimeSlotInvalid, ExTimeFormatInvalid,
        ExUnableToSetOpenCloseTime,
        ExTimeSlotNotReservedYet {
    initialization.initialize();
    ManualClock mc = ManualClock.getInstance();
    String username = "tester";
    String password = "tester";
    acMag.registerCustomer(username, password);
    String cID = acMag.login(username, password);
    Customers tester = db.matchCId(cID);
    ArrayList<Integer> desiredTable = new ArrayList<Integer>();
    desiredTable.add(1);
    desiredTable.add(2);
    desiredTable.add(3);
    Reservation testR = new Reservation(cID, "12:00-13:00", desiredTable, ManualClock.getDate().plusDays(1));
    tester.setReserve(testR);
    mc.newDay();
    mc.changeTime("11:30");
    assertEquals(true, tester.checkisReserved());
    assertEquals(false, tester.isReserveTime());
    mc.changeTime("12:15");
    assertEquals(true, tester.checkisReserved());
    assertEquals(true, tester.isReserveTime());
    for (Integer i : tester.getReservedTableIDs()) {
        tm.reserverCheckIn(i, tester.getReservationTimeSlot(), cID);
    }
    ArrayList<Integer> occupiedTableId = tester.getOccupiedTableId();
    assertEquals(true, occupiedTableId.contains(1));
    assertEquals(true, occupiedTableId.contains(2));
    assertEquals(true, occupiedTableId.contains(3));
    assertEquals(false, occupiedTableId.contains(4));
}
```

Test 6: Test table operation when customer direct walk in

```
@Test
public void testcanDirectlyWalkIn() throws ExTableIdAlreadyInUse, ExPeopleNumExceedTotalCapacity {
    tm.addNewTable(11, 2);
    tm.addNewTable(12, 4);
    ArrayList<Integer> defaultArrangment1 = new ArrayList<>();
    tm.toDefaultAlgo();
    defaultArrangment1 = tm.getTableArrangement(3);
    boolean res1 = tm.canDirectlyDineIn(defaultArrangment1);
    assertEquals(true, res1);
    tm.setWalkInStatus(defaultArrangment1);
    ArrayList<Integer> defaultArrangment2 = new ArrayList<>();
    tm.toDefaultAlgo();
    defaultArrangment2 = tm.getTableArrangement(3);
    boolean res2 = tm.canDirectlyDineIn(defaultArrangment2);
    assertEquals(false, res2);
}
```

Test 7: Set waiting table

```
@Test
public void testSetWaitingTableInTableManagement()
        throws ExPeopleNumExceedTotalCapacity, ExTableIdAlreadyInUse, ExCustomersIdNotFound {
    initialization.initialize();
    String username = "tester";
    String password = "tester";
    acMag.registerCustomer(username, password);
    String cID = acMag.login(username, password);
    ArrayList<Integer> defaultArrangment1 = new ArrayList<>();
    tm.toDefaultAlgo();
    defaultArrangment1 = tm.getTableArrangement(8);
    tm.setWalkInStatus(defaultArrangment1);
    ArrayList<Integer> defaultArrangment2 = new ArrayList<>();
    tm.toDefaultAlgo();
    defaultArrangment2 = tm.getTableArrangement(16);
    Customers c = db.matchCId(cID);
    class cmd_stub extends CommandCustomer {
        public cmd_stub(Customers customer) {
            super(customer);
        }

        @Override
        public void exe() throws ExUnableToSetOpenCloseTime, ExTableIdAlreadyInUse, ExTableNotExist,
                ExTimeSlotNotReservedYet, ExCustomersIdNotFound, ExTimeSlotAlreadyBeReserved {
        }
    }
    Commands cmd = new cmd_stub(c);
    tm.setWaitingTables(cID, defaultArrangment2);
    boolean containsC = tm.waitingCustomersContains(cID);
    assertEquals(true, containsC);
}
```

4.1.6 Table Arrangement Algorithms

There are two algorithms of the table arrangement:

- Default Arrangement
- Recommended Arrangement

4.1.5.1 Default Table Arrangement Algorithm

Branch Coverage:

DefaultTableArrangementAlgorithm.java	100.0 %	215	0	215
DefaultTableArrangementAlgorithm	100.0 %	215	0	215
getInstance()	100.0 %	2	0	2
getTableArrangementResult(int, ArrayList<Tab	100.0 %	147	0	147
initializeTableArrangementList(ArrayList<Intege	100.0 %	19	0	19
returnTableNumWithTableCapacity(int, ArrayLis	100.0 %	22	0	22
returnTotalCapcityOfTables(ArrayList<Table>)	100.0 %	20	0	20

Test 1: Initializing Table Arrangement List

```
@Test
public void testDefaultInitializeTableArrangementList() {
    DefaultTableArrangementAlgorithm dtaa = DefaultTableArrangementAlgorithm.getInstance();
    ArrayList<Integer> tableCapacityTypeList = new ArrayList<>();
    tableCapacityTypeList.add(2);
    tableCapacityTypeList.add(4);
    tableCapacityTypeList.add(8);

    ArrayList<Integer> tableArrangementResults = dtaa.initializeTableArrangementList(tableCapacityTypeList);

    ArrayList<Integer> ExpectedTableArrangementResults = new ArrayList<>();
    ExpectedTableArrangementResults.add(0);
    ExpectedTableArrangementResults.add(0);
    ExpectedTableArrangementResults.add(0);

    assertEquals(ExpectedTableArrangementResults, tableArrangementResults);
}
```

Test 2: Returning Number of Tables of the Table Capacity

```
@Test
public void testDefaultReturnTableNumWithTableCapacity() {
    DefaultTableArrangementAlgorithm dtaa = DefaultTableArrangementAlgorithm.getInstance();
    ArrayList<Table> allTables = new ArrayList<>();

    Table t1 = new Table(11, 2);
    Table t2 = new Table(12, 4);
    Table t3 = new Table(13, 8);
    Table t4 = new Table(14, 8);

    allTables.add(t1);
    allTables.add(t2);
    allTables.add(t3);
    allTables.add(t4);

    assertEquals(2, dtaa.returnTableNumWithTableCapacity(8, allTables));
}
```

Test 3: Calculate the Overall capacity of all the tables

```
@Test
public void testDefaultReturnTotalCapcityOfTables() {
    DefaultTableArrangementAlgorithm dtaa = DefaultTableArrangementAlgorithm.getInstance();
    ArrayList<Table> allTables = new ArrayList<>();

    Table t1 = new Table(11, 2);
    Table t2 = new Table(12, 4);
    Table t3 = new Table(13, 8);
    Table t4 = new Table(14, 8);

    allTables.add(t1);
    allTables.add(t2);
    allTables.add(t3);
    allTables.add(t4);

    assertEquals(22, dtaa.returnTotalCapcityOfTables(allTables));
}
```

Test 4: Obtain the result of arrangement for default arrangement successfully

```
@Test
public void testDefaultGetTableArrangementResult() throws ExPeopleNumExceedTotalCapacity {
    DefaultTableArrangementAlgorithm dtaa = DefaultTableArrangementAlgorithm.getInstance();

    int peopleNum = 11;

    ArrayList<Integer> tableCapacityTypeList = new ArrayList<>();

    tableCapacityTypeList.add(8);
    tableCapacityTypeList.add(4);
    tableCapacityTypeList.add(2);

    ArrayList<Table> allTables = new ArrayList<>();

    Table t1 = new Table(11, 2);
    Table t2 = new Table(12, 4);
    Table t3 = new Table(13, 8);
    Table t4 = new Table(14, 8);
    Table t5 = new Table(15, 4);

    allTables.add(t1);
    allTables.add(t2);
    allTables.add(t3);
    allTables.add(t4);
    allTables.add(t5);

    ArrayList<Table> availableTables = new ArrayList<>();

    availableTables.add(t1);
    availableTables.add(t3);
    availableTables.add(t4);
    availableTables.add(t5);

    ArrayList<Integer> tableArrangementResults = new ArrayList<>();


    tableArrangementResults = dtaa.getTableArrangementResult(peopleNum, availableTables, tableCapacityTypeList, allTables);

    ArrayList<Integer> ExpectedTableArrangementResults = new ArrayList<>();
    ExpectedTableArrangementResults.add(1);
    ExpectedTableArrangementResults.add(1);
    ExpectedTableArrangementResults.add(0);

    assertEquals(ExpectedTableArrangementResults, tableArrangementResults);
}
```

Test 5:

Obtain the result of arrangement for default arrangement when the last table capacity type, i.e., the capacity of 2, available can fit all the people in the inputted number of people

```
@Test
public void testDefaultGetTableArrangementResultPPLNumEqualCapacityLastTable() throws ExPeopleNumExceedTotalCapacity {
    DefaultTableArrangementAlgorithm dtaa = DefaultTableArrangementAlgorithm.getInstance();

    int peopleNum = 2;

    ArrayList<Integer> tableCapacityTypeList = new ArrayList<>();

    tableCapacityTypeList.add(8);
    tableCapacityTypeList.add(4);
    tableCapacityTypeList.add(2);

    ArrayList<Table> allTables = new ArrayList<>();

    Table t1 = new Table(11, 8);
    Table t2 = new Table(12, 8);
    Table t3 = new Table(13, 2);

    allTables.add(t1);
    allTables.add(t2);
    allTables.add(t3);

    ArrayList<Table> availableTables = new ArrayList<>();

    availableTables.add(t3);

    ArrayList<Integer> tableArrangementResults = new ArrayList<>();


    tableArrangementResults = dtaa.getTableArrangementResult(peopleNum, availableTables, tableCapacityTypeList, allTables);

    ArrayList<Integer> ExpectedTableArrangementResults = new ArrayList<>();
    ExpectedTableArrangementResults.add(0);
    ExpectedTableArrangementResults.add(0);
    ExpectedTableArrangementResults.add(1);

    assertEquals(ExpectedTableArrangementResults, tableArrangementResults);
}
```

Test 6:

Obtain the result of arrangement for default arrangement when the table capacity available can fit all the people in the inputted number of people and the table is not the last table capacity

```
@Test
public void testDefaultGetTableArrangementResultPPLNumEqualCapacityNotLastTable() throws ExPeopleNumExceedTotalCapacity {
    DefaultTableArrangementAlgorithm dtaa = DefaultTableArrangementAlgorithm.getInstance();

    int peopleNum = 4;

    ArrayList<Integer> tableCapacityTypeList = new ArrayList<>();

    tableCapacityTypeList.add(8);
    tableCapacityTypeList.add(4);
    tableCapacityTypeList.add(2);

    ArrayList<Table> allTables = new ArrayList<>();

    Table t1 = new Table(11, 8);
    Table t2 = new Table(12, 8);
    Table t3 = new Table(13, 2);

    allTables.add(t1);
    allTables.add(t2);
    allTables.add(t3);

    ArrayList<Table> availableTables = new ArrayList<>();

    availableTables.add(t3);

    ArrayList<Integer> tableArrangementResults = new ArrayList<>();


    tableArrangementResults = dtaa.getTableArrangementResult(peopleNum, availableTables, tableCapacityTypeList, allTables);

    ArrayList<Integer> ExpectedTableArrangementResults = new ArrayList<>();
    ExpectedTableArrangementResults.add(0);
    ExpectedTableArrangementResults.add(0);
    ExpectedTableArrangementResults.add(2);

    assertEquals(ExpectedTableArrangementResults, tableArrangementResults);
}
```

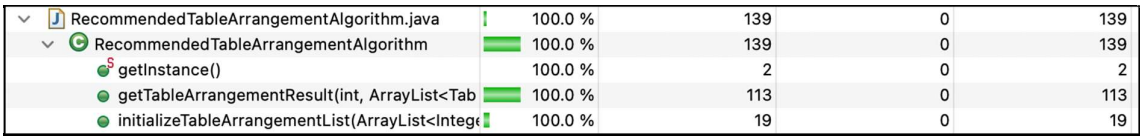
Test 7:

Obtain the result of arrangement for default arrangement when the table capacity available can fit all the people in the inputted number of people and the table is not the last table capacity

```
@Test
public void testDefaultGetTableArrangementResultPPLNumExceedCapacity() throws ExPeopleNumExceedTotalCapacity {
    DefaultTableArrangementAlgorithm dtaa = DefaultTableArrangementAlgorithm.getInstance();

    int peopleNum = 9;

    ArrayList<Integer> tableCapacityTypeList = new ArrayList<>();

    tableCapacityTypeList.add(8);
    tableCapacityTypeList.add(4);
    tableCapacityTypeList.add(2);

    ArrayList<Table> allTables = new ArrayList<>();

    Table t1 = new Table(11, 2);
    Table t2 = new Table(12, 2);
    Table t3 = new Table(13, 2);
    Table t4 = new Table(14, 4);
    Table t5 = new Table(15, 8);

    allTables.add(t1);
    allTables.add(t2);
    allTables.add(t3);
    allTables.add(t4);
    allTables.add(t5);

    ArrayList<Table> availableTables = new ArrayList<>();

    availableTables.add(t5);

    ArrayList<Integer> tableArrangementResults = new ArrayList<>();


    tableArrangementResults = dtaa.getTableArrangementResult(peopleNum, availableTables, tableCapacityTypeList, allTables);

    ArrayList<Integer> ExpectedTableArrangementResults = new ArrayList<>();
    ExpectedTableArrangementResults.add(1);
    ExpectedTableArrangementResults.add(0);
    ExpectedTableArrangementResults.add(1);

    assertEquals(ExpectedTableArrangementResults, tableArrangementResults);
}
```

4.1.5.2 Recommended Arrangement

Branch Coverage:



RecommendedTableArrangementAlgorithm.java	100.0 %	139	0	139
RecommendedTableArrangementAlgorithm	100.0 %	139	0	139
getInstance()	100.0 %	2	0	2
getTableArrangementResult(int, ArrayList<Tab	100.0 %	113	0	113
initializeTableArrangementList(ArrayList<Integ	100.0 %	19	0	19

Test 1: Initializing Table Arrangement List

```
@Test
public void testRecommendedInitializeTableArrangementList() {
    RecommendedTableArrangementAlgorithm rtaa = RecommendedTableArrangementAlgorithm.getInstance();
    ArrayList<Integer> tableCapacityTypeList = new ArrayList<>();
    tableCapacityTypeList.add(2);
    tableCapacityTypeList.add(4);
    tableCapacityTypeList.add(8);

    ArrayList<Integer> tableArrangementResults = rtaa.initializeTableArrangementList(tableCapacityTypeList);

    ArrayList<Integer> ExpectedTableArrangementResults = new ArrayList<>();
    ExpectedTableArrangementResults.add(0);
    ExpectedTableArrangementResults.add(0);
    ExpectedTableArrangementResults.add(0);

    assertEquals(ExpectedTableArrangementResults, tableArrangementResults);
}
```

Test 2: Obtain the optimized arrangement (recommended arrangement) result successfully

```
@Test
public void testRecommendedGetTableArrangementResult() {
    // Has optimised Recommended Arrangement
    RecommendedTableArrangementAlgorithm dtaa = RecommendedTableArrangementAlgorithm.getInstance();

    int peopleNum = 11;

    ArrayList<Integer> tableCapacityTypeList = new ArrayList<>();

    tableCapacityTypeList.add(8);
    tableCapacityTypeList.add(4);
    tableCapacityTypeList.add(2);

    ArrayList<Table> allTables = new ArrayList<>();

    Table t1 = new Table(11, 2);
    Table t2 = new Table(12, 4);
    Table t3 = new Table(13, 8);
    Table t4 = new Table(14, 8);
    Table t5 = new Table(15, 4);

    allTables.add(t1);
    allTables.add(t2);
    allTables.add(t3);
    allTables.add(t4);
    allTables.add(t5);

    ArrayList<Table> availableTables = new ArrayList<>();

    availableTables.add(t1);
    availableTables.add(t3);
    availableTables.add(t4);
    availableTables.add(t5);

    ArrayList<Integer> tableArrangementResults = new ArrayList<>();

    tableArrangementResults = dtaa.getTableArrangementResult(peopleNum, availableTables, tableCapacityTypeList, allTables);

    ArrayList<Integer> ExpectedTableArrangementResults = new ArrayList<>();
    ExpectedTableArrangementResults.add(2);
    ExpectedTableArrangementResults.add(0);
    ExpectedTableArrangementResults.add(0);

    assertEquals(ExpectedTableArrangementResults, tableArrangementResults);
}
```

Test 3: No optimized arrangement

```
@Test
public void testRecommendedGetTableArrangementResultPPLNumExceedCapacity() {
    // No optimised Recommended Arrangement
    RecommendedTableArrangementAlgorithm dtaa = RecommendedTableArrangementAlgorithm.getInstance();

    int peopleNum = 31;

    ArrayList<Integer> tableCapacityTypeList = new ArrayList<>();

    tableCapacityTypeList.add(8);
    tableCapacityTypeList.add(4);
    tableCapacityTypeList.add(2);

    ArrayList<Table> allTables = new ArrayList<>();

    Table t1 = new Table(11, 2);
    Table t2 = new Table(12, 4);
    Table t3 = new Table(13, 8);
    Table t4 = new Table(14, 8);
    Table t5 = new Table(15, 4);

    allTables.add(t1);
    allTables.add(t2);
    allTables.add(t3);
    allTables.add(t4);
    allTables.add(t5);

    ArrayList<Table> availableTables = new ArrayList<>();

    availableTables.add(t1);
    availableTables.add(t3);
    availableTables.add(t4);
    availableTables.add(t5);

    ArrayList<Integer> tableArrangementResults = new ArrayList<>();

    tableArrangementResults = dtaa.getTableArrangementResult(peopleNum, availableTables, tableCapacityTypeList, allTables);

    assertEquals(null, tableArrangementResults);
}
```

4.2 Black-Box Testing

Advantage of Black-Box testing:

- The test is unbiased because the designer and the tester are independent of each other.
- The tester does not need knowledge of any specific programming languages.
- The test is done from the point of view of the user, not the designer.
- Test cases can be designed as soon as the specifications are complete.

4.2.1 Sample Test Cases

Test Case 1:

- Input: Super VIP, Dine-in, Pepper-Lunch, {Pork, Turkey}, Wechat Pay

1	1	2	1	3	2	3	2

- Output: $86.24, "You will be redirected to WeChat to continue payment."

```
Please choose your option: [1 Dine-in | 2 Reserve]: 1

Available restaurants:
[1] Pepper-Lunch
[2] Tam-Jai-Mi-Xian
[3] McDonald's
[4] KFC

Please choose the restaurant to order:
1

You have chosed restaurant Pepper-Lunch.

Menu:
[1] Beef
[2] Pork
[3] Turkey

Please choose from the menu of restaurant Pepper-Lunch (separate by a COMMA):
2

Your pending orders:
[1] Pork

Do you want to add or delete order? [1 ADD | 2 DELETE | 3 Continue]: 1

Menu:
[1] Beef
[2] Pork
[3] Turkey

Input the dish number to add: (separate by a COMMA): 3

Your pending orders:
[1] Pork
[2] Turkey

Do you confirm to order? [1 YES | 2 NO]: 2

Your pending orders:
[1] Pork
[2] Turkey

Do you want to add or delete order? [1 ADD | 2 DELETE | 3 Continue]: 3
Official Orders:
[1] Pork
[2] Turkey

You are a Super VIP! You get 80% discount.

Your bill is $86.24.
Please choose a payment method [1 Alipay | 2 WeChat Pay | 3 Cash]: 2

You will be redirected to WeChat to continue payment.

You have completed payment with WeChat Pay. Thank you!
```

Test Case 2:

- Input: Member, Dine-in, KFC, { Beef-Egg-Burger, Beef-Egg-Burger, Beef-Egg-Burger}, Wechat Pay

1	1	2	1	3	2	3	2

- Output: $86.24, “You will be redirected to WeChat to continue payment.”

```
Please choose your option: [1 Dine-in | 2 Reserve]: 1

Available restaurants:
[1] Pepper-Lunch
[2] Tam-Jai-Mi-Xian
[3] McDonald's
[4] KFC

Please choose the restaurant to order:
4

You have chosed restaurant KFC.

Menu:
[1] Filet-O-Fish
[2] Beef-Egg-Burger

Please choose from the menu of restaurant KFC (separate by a COMMA):
2

Your pending orders:
[1] Beef-Egg-Burger

Do you want to add or delete order? [1 ADD | 2 DELETE | 3 Continue]: 1

Menu:
[1] Filet-O-Fish
[2] Beef-Egg-Burger

Input the dish number to add: (separate by a COMMA): 2

Your pending orders:
[1] Beef-Egg-Burger
[2] Beef-Egg-Burger

Do you confirm to order? [1 YES | 2 NO]: 2

Your pending orders:
[1] Beef-Egg-Burger
[2] Beef-Egg-Burger

Do you want to add or delete order? [1 ADD | 2 DELETE | 3 Continue]: 1

Menu:
[1] Filet-O-Fish
[2] Beef-Egg-Burger

Input the dish number to add: (separate by a COMMA): 2

Your pending orders:
[1] Beef-Egg-Burger
[2] Beef-Egg-Burger
[3] Beef-Egg-Burger
```

```
Do you confirm to order? [1 YES | 2 NO]: 1
Official Orders:
[1] Beef-Egg-Burger
[2] Beef-Egg-Burger
[3] Beef-Egg-Burger

You are a member. Consume up to $88 to get 80% discount!

Your bill is $39.00.
```

4.2.2 Invalid Input

4.2.2.1 Invalid Input 1: Integer out of given choice

Example 1:

- Input: 3
- Expected output: Ask for valid input again.
- Actual output:

```
Please choose your option: [1 Dine-in | 2 Reserve]: 3

Please choose your option: [1 Dine-in | 2 Reserve]:
```

Example 2:

- Input: 5
- Expected output: Ask for valid restaurant input again.
- Actual output:

```
Please choose your option: [1 Dine-in | 2 Reserve]: 2

Available restaurants:
[1] Pepper-Lunch
[2] Tam-Jai-Mi-Xian
[3] McDonald's
[4] KFC

Please choose the restaurant to order:
5
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 4 out of bounds for length 4
        at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
        at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
        at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:248)
        at java.base/java.util.Objects.checkIndex(Objects.java:372)
        at java.base/java.util.ArrayList.get(ArrayList.java:459)
        at CustomerModule.Ordering(CustomerModule.java:88)
        at CustomerModule.run(CustomerModule.java:62)
        at Main.main(Main.java:31)
```

4.2.2.2 Invalid Input 2: Not an integer

Example 1:

- Input: 1.5
- Expected output: Ask for an integer.
- Actual output: Error

```
Please choose your option: [1 Dine-in | 2 Reserve]: 1.5
Exception in thread "main" java.lang.NumberFormatException: For input string: "1.5"
        at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.base/java.lang.Integer.parseInt(Integer.java:652)
        at java.base/java.lang.Integer.parseInt(Integer.java:770)
        at CustomerModule.run(CustomerModule.java:54)
        at Main.main(Main.java:31)
```

Example 2:

- Input: a
- Expected output: Ask for an integer.
- Actual output: Error

```
Please choose your option: [1 Dine-in | 2 Reserve]: a
Exception in thread "main" java.lang.NumberFormatException: For input string: "a"
        at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.base/java.lang.Integer.parseInt(Integer.java:652)
        at java.base/java.lang.Integer.parseInt(Integer.java:770)
        at CustomerModule.run(CustomerModule.java:54)
        at Main.main(Main.java:31)
```

5. System Test

5.1 Testing Approach

The testing approach we adopted for system testing is Category-Partition Method (CPM) in Black-box Testing. The restrictions for test cases are written in Test Specification Language, then we write test cases based on the TSL.

By using Category-Partition Method (CPM), we firstly analyze the specification to identify the individual testable function units, parameters of the function units, and entities in environment which affect operation. Secondly we classify the specifications into categories and partition the categories into choices. Thirdly we determine the constraints among the choices, for example, how one choice in one category affect the other choices the another category. Then we write and process the test specification, and express them in TSL. Next, we evaluate the expect output of each test frame. Finally we convert the test frames into test cases.

In the below part of system test, we will divide the system test by different login in identities, which are customer, merchant, and admin. And for each identity, the testing TSL, test frames and test cases will be shown in detail.

5.2 Test Cases for System Testing

5.2.1 Customer

5.2.1.1 Customer TSL

Service:

Dine In	[property dine in]
Reservation	[property reservation]
Cancel Reservation	[property cancel]
Log Out	[property logout]

Reserved:

Reserved	[property isReserved]
No reservation	[property noReserved]

Choose Table:

Valid table	[if dine in & noReserved] [property have table]
Invalid table	[if dine in & noReserved] [property no table]

Action:

Walk in	[if isReserved or have table] [property walk in]
Leave	[if dine in]
Queue	[if no table]

Restaurant:

Pepper – Lunch	[if walk in]
Tam-Jai-Mi-Xian	[if walk in]
McDonald's	[if walk in]
KFC	[if walk in]

Reserve Time:

Valid time	[if reservation & noReserved]
Invalid time	[if reservation & noReserved]

5.2.1.2 Customer Test Frames

	Service	Reserved	Choose Table	Action	Restaurant	Reserve Time	Dish	Expect Behaviour
1	Cancel Reservation	Reserved	/	/	/	/	/	Cancel Success!
2	Reservation	No reservation	/	/	/	Valid Time	/	Reserve Success. And Reservation Reminder
3	Reservation	No reservation	/	/	/	Invalid Time	/	Invalid Time Slot! And ask to retry.
4	Dine in	Reserved	/	Walk in	KFC	/	…	Return the bill and bill number.
5	Dine in	Reserved	/	Walk in	Tam-Jai -Mi-Xian	/	…	Return the bill and bill number.
6	Dine in	Reserved	/	Walk in	McDonald's	/	…	Return the bill and bill number.
7	Dine in	Reserved	/	Walk in	Pepper -Lunch	/	…	Return the bill and bill number.
8	Dine in	No reservation	Valid table	Walk in	Tam-Jai -Mi-Xian	/	…	Return the bill and bill number.
9	Dine in	No reservation	Valid table	Walk in	KFC	/	…	Return the bill and bill number.
10	Dine in	No reservation	Valid table	Walk in	McDonald's	/	…	Return the bill and bill number.
11	Dine in	No reservation	Valid table	Walk in	Pepper -Lunch	/	…	Return the bill and bill number.
12	Dine in	No reservation	Valid table	Leave	/	/	/	Check out to the command bar.
13	Dine in	No reservation	Invalid table	Leave	/	/	/	Check out to the command bar.
14	Dine in	No reservation	Invalid table	Queue	/	/	/	Return the waiting number.

5.2.1.3 Coverage

Coverage: 100%

We manually list out all valid combinations of parameters "Service", "Reserved", "Choose Table", "Action", "Restaurant", and "Reserve Time".

5.2.1.4 Test Cases and Actual Output

Test Case 1:

	Service	Reserved	Choose Table	Action	Restaurant	Reserve Time	Dish	Expect Behaviour
1	Cancel Reservation	Reserved	/	/	/	/	/	Cancel Success!

Actual Output of Test Case 1:

```
Reminder: You have a reservation for tomorrow:
[1] [Table with ID: 2] [Time Slot: 10:00-11:00]

Commands:
[1] Dine in
[3] Cancel Reservation
[5] Logout

Please select your operation: 3

Cancel Success!
```

Test Case 2:

2	Reservation	No Reservation	/	/	/	Valid Time	/	Reserve Success. And remind the reserved time.

Actual Output of Test Case 2:

```
Commands:
[1] Dine in
[2] Reserve
[5] Logout

Please select your operation: 2

Table(s) for tomorrow reservation and available time slots:
2-Seats Table with ID of 1 is available tmr for the timeslots: 00:00-23:59
2-Seats Table with ID of 2 is available tmr for the timeslots: 00:00-23:59
2-Seats Table with ID of 3 is available tmr for the timeslots: 00:00-23:59
2-Seats Table with ID of 4 is available tmr for the timeslots: 00:00-23:59
2-Seats Table with ID of 5 is available tmr for the timeslots: 00:00-23:59
4-Seats Table with ID of 6 is available tmr for the timeslots: 00:00-23:59
4-Seats Table with ID of 7 is available tmr for the timeslots: 00:00-23:59
4-Seats Table with ID of 8 is available tmr for the timeslots: 00:00-23:59
8-Seats Table with ID of 9 is available tmr for the timeslots: 00:00-23:59
8-Seats Table with ID of 10 is available tmr for the timeslots: 00:00-23:59


Please input time slot to reserve (Format: xx:xx-xx:xx): 10:00-11:00
Please input the table ids you want to reserve: (separate by comma): 1

Table with id of 1 is succesfully reserved between 10:00-11:00.

Reserve Success.
[1] [Table with ID: 1] [Time Slot: 10:00-11:00]
```

Test Case 3:

3	Reservation	No Reservation	/	/	/	Invalid Time	/	Invalid Time Slot! And ask to retry

Actual Output of Test Case 3:

```
Commands:
[1] Dine in
[2] Reserve
[5] Logout

Please select your operation: 2

Table(s) for tomorrow reservation and available time slots:
2-Seats Table with ID of 1 is available tmr for the timeslots: 00:00-23:59
2-Seats Table with ID of 2 is available tmr for the timeslots: 00:00-23:59
2-Seats Table with ID of 3 is available tmr for the timeslots: 00:00-23:59
2-Seats Table with ID of 4 is available tmr for the timeslots: 00:00-23:59
2-Seats Table with ID of 5 is available tmr for the timeslots: 00:00-23:59
4-Seats Table with ID of 6 is available tmr for the timeslots: 00:00-23:59
4-Seats Table with ID of 7 is available tmr for the timeslots: 00:00-23:59
4-Seats Table with ID of 8 is available tmr for the timeslots: 00:00-23:59
8-Seats Table with ID of 9 is available tmr for the timeslots: 00:00-23:59
8-Seats Table with ID of 10 is available tmr for the timeslots: 00:00-23:59


Please input time slot to reserve (Format: xx:xx-xx:xx): 1234
Please input the table ids you want to reserve: (separate by comma): 3
Invalid Time Slot!!

Please try to reserve again :-)
```

Test Case 4:

4	Dine in	Reserved/ No reservation	/	Walk in	A restaurant	/	Some dish	Return the bill and bill number.

Actual Output of Test Case 4:

```
Commands:
[1] Dine in
[2] Reserve
[5] Logout

Please select your operation: 1

Below is the available tables:
Num of Available 2-Seats Table: 4
Num of Available 4-Seats Table: 3
Num of Available 8-Seats Table: 0

Please input the number of people: 2

Your arranged tables are:
[1] [2-Seats] Tables

You can now directly walk in.

Commands:
[1] Walk in
[2] Leave

Please choose your operation: 1
Successfully Dine In!

Available restaurants:
[1] Pepper-Lunch
[2] Tam-Jai-Mi-Xian
[3] McDonald's
[4] KFC
[5] 12

Please choose the restaurant to order: 1

You have chosed restaurant Pepper-Lunch.

Menu:
[1] Beef
[2] Pork
[3] Turkey

Please choose from the menu of restaurant Pepper-Lunch (separate by a COMMA): 1

Your pending orders:
[1] Beef

Do you want to confirm order?

Commands:
[1] Yes: Please input 'True'/'true'/'TRUE'
[2] No: Please input 'False'/'false'/'FALSE'

Your option: true

Your orders are:
[1] Beef

You are a member. Consume up to $88 to get 80% discount!

Your bill is $59.90.

Your bill number is: B0004
```

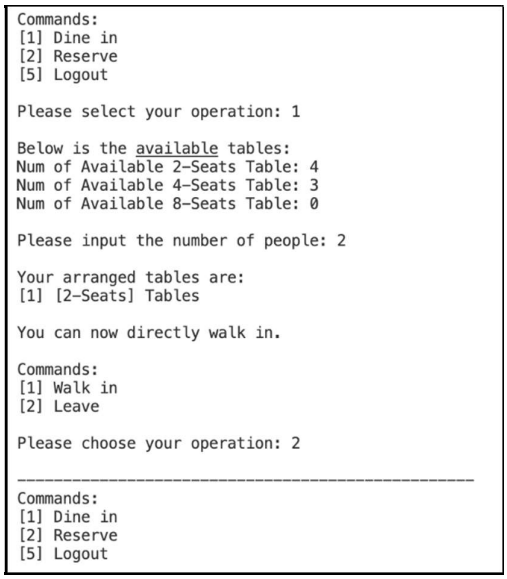
Specific Cases for Dish

Restaurant	Dish	Membership	Expected payment	Actual Price
Tam-Jai-Mi-Xian	[1] Lettuce-Mixian	VIP	$29.9	$29.90
Pepper-Lunch	[1] Beef [2] Pork	SuperVIP	$91.84	$91.84
Pepper-Lunch	[1] Beef [2] Pork	VIP	$91.84	$91.84
KFC	[1] Filet-O-Fish [2] Filet-O-Fish	SuperVIP	$19.2	$19.2
Mcdonald's	[1] Americano [2] Latte [3] Americano [4] Latte	VIP	$72	$72
KFC	[1] Filet-O-Fish	VIP	$12	$12
Pepper-Lunch	[1] Turkey	VIP	$52.9	$52.9
KFC	[1] Filet-O-Fish [2] Beef-Egg-Burger [3] Beef-Egg-Burger [4] Filet-O-Fish	VIP	$50	$50
Tam-Jai-Mi-Xian	[1] Lettuce-Mixian [2] Lettuce-Mixian [3] Lettuce-Mixian [4] Pork-Mixian	SuperVIP	$99.68	$99.68
Mcdonald's	[1] Hot-Chocolate [2] Latte [3] Americano [4] Americano [5] Americano [6] Hot-Chocolate	SuperVIP	$83.2	$83.2

Test Case 5:

12	Dine in	No reservation	Valid table	Leave	/	/	/	Check out to the command bar.

Actual Output of Test Case 5:

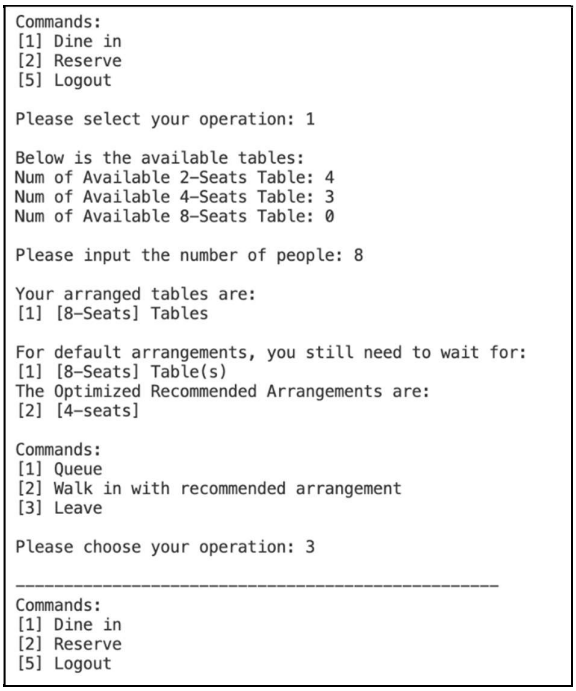


```
Commands:
[1] Dine in
[2] Reserve
[5] Logout

Please select your operation: 1

Below is the available tables:
Num of Available 2-Seats Table: 4
Num of Available 4-Seats Table: 3
Num of Available 8-Seats Table: 0

Please input the number of people: 2

Your arranged tables are:
[1] [2-Seats] Tables

You can now directly walk in.

Commands:
[1] Walk in
[2] Leave

Please choose your operation: 2

--------------------------------------------------
Commands:
[1] Dine in
[2] Reserve
[5] Logout
```

Test Case 6:

13	Dine in	No reservation	Invalid table	Leave	/	/	/	Check out to the command bar.

Actual Output of Test Case 6:



```
Commands:
[1] Dine in
[2] Reserve
[5] Logout

Please select your operation: 1

Below is the available tables:
Num of Available 2-Seats Table: 4
Num of Available 4-Seats Table: 3
Num of Available 8-Seats Table: 0

Please input the number of people: 8

Your arranged tables are:
[1] [8-Seats] Tables

For default arrangements, you still need to wait for:
[1] [8-Seats] Table(s)
The Optimized Recommended Arrangements are:
[2] [4-seats]

Commands:
[1] Queue
[2] Walk in with recommended arrangement
[3] Leave

Please choose your operation: 3

--------------------------------------------------
Commands:
[1] Dine in
[2] Reserve
[5] Logout
```

Test Case 7:

14	Dine in	No reservation	Invalid table	Queue	/	/	/	Return the waiting number.

Actual Output of Test Case 7:

```
Commands:
[1] Dine in
[2] Reserve
[5] Logout

Please select your operation: 1

Below is the available tables:
Num of Available 2-Seats Table: 4
Num of Available 4-Seats Table: 3
Num of Available 8-Seats Table: 0

Please input the number of people: 8

Your arranged tables are:
[1] [8-Seats] Tables

For default arrangements, you still need to wait for:
[1] [8-Seats] Table(s)
The Optimized Recommended Arrangements are:
[2] [4-seats]

Commands:
[1] Queue
[2] Walk in with recommended arrangement
[3] Leave

Please choose your operation: 1
For selected arrangements, you still need to wait for:
[1] [8-Seats] Table
```

5.2.2 Merchant

5.2.2.1 Merchant TSL

Service:

Modify menu	[property modify]
Check order	[property check order]
Log out	[property logout]

Menu:

Add dish	[if modify] [property add]
Remove dish	[if modify] [property remove]
Edit dish	[if modify] [property edit]

Remove:

Valid number	[if remove]
Invalid number	[if remove]

Edit:

Name	[if edit] [property edit name]
Price	[if edit] [property edit price]

Dish Name:

Valid dish name	[if add or edit name]
Invalid dish name	[if add or edit name]

Dish Price:

Valid price	[if add or edit price]
Invalid price	[if add or edit price]

Customer ID:

Existing ID	[if check order] [property id]
Invalid ID	[if check order]

Bill:

Have bill	[if id]
No bill	[if id]

5.2.2.2 Merchant Test Frames

	Service	Menu	Remove	Edit	Dish Name	Dish Price	Customer ID	Bill	Expect Behaviour
1	Modify Menu	Add dish	/	/	Valid	Valid	/	/	Add dish success.
2	Modify Menu	Edit Dish	/	Price	/	Valid	/	/	Edit dish success.
3	Check Order	/	/	/	/	/	Existing ID	Have bill	Return the bill.
4	Modify Menu	Add dish	/	/	Invalid	Valid	/	/	Ask for dish name again.
5	Check Order	/	/	/	/	/	Existing ID	No bill	Cannot find the bill.
6	Modify Menu	Add dish	/	/	Invalid	Invalid	/	/	Ask for the dish name and price again.
7	Modify Menu	Edit Dish	/	Price	/	Invalid	/	/	Ask for the dish price again.
8	Modify Menu	Remov e dish	Valid number	/	/	/	/	/	Remove dish success.
9	Modify Menu	Edit Dish	/	Name	Valid	/	/	/	Edit dish success.
10	Check Order	/	/	/	/	/	Invalid ID	No bill	Ask for the customer id again.
11	Check Order	/	/	/	/	/	Invalid ID	Have bill	Ask for the customer id again.
12	Modify Menu	Add dish	/	/	Valid	Invalid	/	/	Ask for the dish price again.
13	Modify Menu	Remov e dish	Invalid number	/	/	/	/	/	Ask for the dish number again.
14	Modify Menu	Edit Dish	/	Name	Invalid	/	/	/	Ask for the dish name again.

5.2.2.3 Triple Wise Coverage

Tests: 14
Number of Possible Tests : 30
Coverage : 100%
Valid combinations of 3 values :

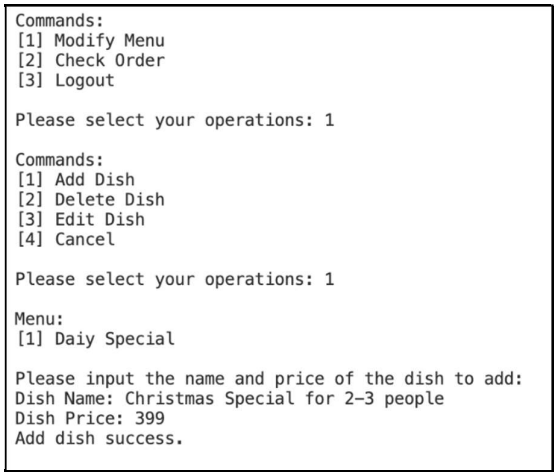
- covered : 414
- total : 414
- uncovered : 0

5.2.2.4 Test Cases and Actual Output

Test Case 1:

	Service	Menu	Remove	Edit	Dish Name	Dish Price	Customer ID	Bill	Expect Behaviour
1	Modify Menu	Add dish	/	/	Valid	Valid	/	/	Add dish success.

Actual Output of Test Case 1:



Test Case 2:

2	Modify Menu	Edit Dish	/	Price	/	Valid	/	/	Edit dish success.

Actual Output of Test Case 2:

```
Commands:
[1] Modify Menu
[2] Check Order
[3] Logout

Please select your operations: 1

Commands:
[1] Add Dish
[2] Delete Dish
[3] Edit Dish
[4] Cancel

Please select your operations: 3

Menu:
[1] Daily Special
[2] Christmas Special for 2-3 people

Please input the name of the dish: Daily Special

Commands:
[1] Edit Dish Name
[2] Edit Dish Price
[3] Cancel

Please select your operations: 2

Input the new price: 60
Edit Dish Price success.
```

Test Case 3:

5	Check Order	/	/	/	/	/	Existing ID	No bill	Cannot find the bill.

Actual Output of Test Case 3:

```
Commands:
[1] Modify Menu
[2] Check Order
[3] Logout

Please select your operations: 2

Please input customer's id: C0005

Customer Name: 123
Customer ID: C0005

--------------------------------------------------
Commands:
[1] Modify Menu
[2] Check Order
[3] Logout
```

Test Case 4:

7	Modify Menu	Edit Dish	/	Price	/	Invalid	/	/	Ask for the dish price again.

Actual Output of Test Case 4:

```
Commands:
[1] Modify Menu
[2] Check Order
[3] Logout

Please select your operations: 1

Commands:
[1] Add Dish
[2] Delete Dish
[3] Edit Dish
[4] Cancel

Please select your operations: 3

Menu:
[1] Daily Special
[2] Christmas Special for 2-3 people

Please input the name of the dish: Daily Special

Commands:
[1] Edit Dish Name
[2] Edit Dish Price
[3] Cancel

Please select your operations: 2

Input the new price: $
Error! Wrong input for selection! Please input an integer!
```

Test Case 5:

8	Modify Menu	Remove dish	Valid number	/	/	/	/	/	Remove dish success.

Actual Output of Test Case 5:

```
Commands:
[1] Modify Menu
[2] Check Order
[3] Logout

Please select your operations: 1

Commands:
[1] Add Dish
[2] Delete Dish
[3] Edit Dish
[4] Cancel

Please select your operations: 2

Menu:
[1] Daily Special
[2] Christmas Special for 2-3 people

Please input the numbering of the dish to remove: 2
Delete Dish success.
```

Test Case 6:

12	Modify Menu	Add dish	/	/	Valid	Invalid	/	/	Ask for the dish price again.

Actual Output of Test Case 6:

```
Commands:
[1] Modify Menu
[2] Check Order
[3] Logout

Please select your operations: 1

Commands:
[1] Add Dish
[2] Delete Dish
[3] Edit Dish
[4] Cancel

Please select your operations: 1

Menu:
[1] 1234
[2] 1
[3] 111

Please input the name and price of the dish to add:
Dish Name: Christmas Special for 2-3 people
Dish Price: $399
Please input a double.

Please input the name and price of the dish to add:
Dish Name:
```

Test Case 7:

14	Modify Menu	Edit Dish	/	Name	Invalid	/	/	/	Ask for the dish name again.

Actual Output of Test Case 7:

```
Commands:
[1] Modify Menu
[2] Check Order
[3] Logout

Please select your operations: 1

Commands:
[1] Add Dish
[2] Delete Dish
[3] Edit Dish
[4] Cancel

Please select your operations: 3

Menu:
[1] Daiy Special
[2] Christmas Special for 2-3 people

Please input the name of the dish: 1
Wrong Dish Name! Please try again.
```

5.2.3 Admin

5.2.3.1 Admin TSL

Service:

Change Opening Time	[Property Change Opening Hour]
Check Order	[Property Check]
Check Reservation	[Property Check]
Add Restaurant	[Property Add Restaurant]
Remove Restaurant	[Property Remove Restaurant]
Add Table	[Property Add]
Remove Table	[Property Remove]
Log Out	[single] [Property Log Out]

Correct Input Format

Yes	[if Not Add Restaurant or Log Out]
No	[error] [if Not Add Restaurant or Log Out]

Time Overlap

Yes	[error] [if Change Opening Hour]
No	[if Change Opening Hour]

ID / Name Exists

Yes	[if Not Change Opening Hour or Log Out]
No	[if Not Change Opening Hour or Log Out] [error]

Match Found

Yes	[if Check]
No	[if Check]

5.2.3.2 Admin Test Frames

	Service	Correct Input Format	Time verlap	ID / Name Exists	Match Found	Expected Behavior
1	Change Opening Time	Yes	No	/	/	Change Success
2	Change Opening Time	Yes	Yes	/	/	Change Fail, ask for retry
3	Change Opening Time	No	/	/	/	Change Fail, ask for retry
4	Check Order	Yes	/	Yes	Yes	Print order info
5	Check Order	Yes	/	Yes	No	Order not found
6	Check Order	Yes	/	No	/	ID / Name not found, ask for retry
7	Check Order	No	/	/	/	Invalid format, ask for retry
8	Check Reservation	Yes	/	Yes	Yes	Print reservation info
9	Check Reservation	Yes	/	Yes	No	Reservation not found
10	Check Reservation	Yes	/	No	/	ID / Name not found, ask for retry
11	Check Reservation	No	/	/	/	Invalid format, ask for retry
12	Add Restaurant	/	/	/	/	Add success
13	Remove Restaurant	Yes	/	Yes	/	Remove success
14	Remove Restaurant	Yes	/	No	/	ID / Name not found, ask for retry
15	Remove Restaurant	No	/	/	/	Invalid format, ask for retry
16	Add Table	Yes	/	Yes	/	Add Fail, duplicate ID, ask for retry
17	Add Table	Yes	/	No	/	Add Success

18	Add Table	No	/	/	/	Invalid format, ask for retry
19	Remove Table	Yes	/	Yes	/	Remove Success
20	Remove Table	Yes	/	No	/	ID not found, ask for retry
21	Remove Table	No	/	/	/	Invalid format, ask for retry
22	Log Out	/	/	/	/	Logged out

5.2.3.3 Coverage

Coverage: 100%

We manually list out all valid combinations of parameters “Service”, “Correct Input Format”, “Time Overlap”, “ID / Name Exists”, “Match Found”

5.2.3.4 Test Cases and Actual Output

Test Case 1:

	Service	Correct Input Format	Time verlap	ID / Name Exists	Match Found	Expected Behavior
1	Change Opening Time	Yes	No	/	/	Change Success

Actual Output of Test Case 1:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 1

Please input new opening and closing hour (format: xx:xx-xx:xx): 08:00-23:00

Change opening and closing time success!
The new opening hour is 08:00-23:00.
```

Test Case 2:

3	Change Opening Time	No	/	/	/	Change Fail, ask for retry

Actual Output of Test Case 2:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 1

Please input new opening and closing hour (format: xx:xx-xx:xx): 8:00-23:00

Error! Time format invalid. Valid format: xx:xx-xx:xx
```

Test Case 3:

4	Check Order	Yes	/	Yes	Yes	Print order info

Actual Output of Test Case 3:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 2

Please input the CustomerId to check order: C0005

Official Orders:
[1] Beef
[2] Beef
```

Test Case 4:

5	Check Order	Yes	/	Yes	No	Order not found

Actual Output of Test Case 4:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 2

Please input the CustomerId to check order: C0001
There is no orders made by this customer.
```

Test Case 5:

6	Check Order	Yes	/	No	/	ID / Name not found, ask for retry

Actual Output of Test Case 5:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 2

Please input the CustomerId to check order: C0010
Error! Customer ID: C0010 not found!
```

Test Case 6:

7	Check Order	No	/	/	/	Invalid format, ask for retry

Actual Output of Test Case 6:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 2
\
Please input the CustomerId to check order: A
Error! Customer ID: \A not found!
```

Test Case 7:

8	Check Reservation	Yes	/	Yes	Yes	Print reservation info

Actual Output of Test Case 7:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 3

Please input the CustomerId to check reservation info: C0005

The upcoming reservation of Customer C0005 for tomorrow is:
[1] [Table with ID: 1] [Time Slot: 10:00-11:30]
```

Test Case 8:

9	Check Reservation	Yes	/	Yes	No	Reservation not found

Actual Output of Test Case 8:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 3

Please input the CustomerId to check reservation info: C0008
There is no reservation made by this customer.
```

Test Case 9:

10	Check Reservation	Yes	/	No	/	ID / Name not found, ask for retry

Actual Output of Test Case 9:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 3

Please input the CustomerId to check reservation info: C00000
Error! Customer ID: C00000 not found!
```

Test Case 10:

11	Check Reservation	No	/	/	/	Invalid format, ask for retry

Actual Output of Test Case 10:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 3

Please input the CustomerId to check reservation info: abcde
Error! Customer ID: abcde not found!
```

Test Case 11:

12	Add Restaurant	/	/	/	/	Add success

Actual Output of Test Case 11:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 4

Please input the name of the new Restaurant: Starbuck's
Add new restaurant success.

List of Restaurants:
[1] Pepper-Lunch
[2] Tam-Jai-Mi-Xian
[3] McDonald's
[4] Ganki Sushi
[5] KFC
[6] Starbuck's
```

Test Case 12:

13	Remove Restaurant	Yes	/	Yes	/	Remove success

Actual Output of Test Case 12:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 5

Available restaurants:
[1] Pepper-Lunch
[2] Tam-Jai-Mi-Xian
[3] McDonald's
[4] KFC
[5] Ganki Sushi

Please input the name of the Restaurant to remove: KFC
Delete restaurant success.

List of Restaurants:
[1] Pepper-Lunch
[2] Tam-Jai-Mi-Xian
[3] McDonald's
[4] Ganki Sushi
```

Test Case 13:

15	Remove Restaurant	No	/	/	/	Invalid format, ask for retry

Actual Output of Test Case 13:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 5

Available restaurants:
[1] Pepper-Lunch
[2] Tam-Jai-Mi-Xian
[3] McDonald's
[4] Ganki Sushi
[5] KFC
[6] Starbuck's

Please input the name of the Restaurant to remove: 8
No such restaurant. Please check again.
```

Test Case 14:

16	Add Table	Yes	/	Yes	/	Add Fail, duplicate ID, ask for retry

Actual Output of Test Case 14:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 6

Please input the new tableId: 1

Please input the capacity of new table: 2
Can't add such table because Table with ID of 1 is already in use!
```

Test Case 15:

17	Add Table	Yes	/	No	/	Add Success

Actual Output of Test Case 15:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 6

Please input the new tableId: 20

Please input the capacity of new table: 2
Successfully add table with ID of 20, capacity of 2
```

Test Case 16:

18	Add Table	No	/	/	/	Invalid format, ask for retry

Actual Output of Test Case 16:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 6

Please input the new tableId: abc
Error! Wrong input for selection! Please input an integer!
```

Test Case 17:

19	Remove Table	Yes	/	Yes	/	Remove Success

Actual Output of Test Case 17:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 7

List of all Tables:
2-Seats Table | ID: 1
2-Seats Table | ID: 2
2-Seats Table | ID: 3
2-Seats Table | ID: 4
2-Seats Table | ID: 5
2-Seats Table | ID: 20
4-Seats Table | ID: 6
4-Seats Table | ID: 7
4-Seats Table | ID: 8
8-Seats Table | ID: 9
8-Seats Table | ID: 10


Please input the TableId to delete table: 20
Successfully delete the table with id of 20
```

Test Case 18:

20	Remove Table	Yes	/	No	/	ID not found, ask for retry

Actual Output of Test Case 18:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 7

List of all Tables:
2-Seats Table | ID: 1
2-Seats Table | ID: 2
2-Seats Table | ID: 3
2-Seats Table | ID: 4
2-Seats Table | ID: 5
4-Seats Table | ID: 6
4-Seats Table | ID: 7
4-Seats Table | ID: 8
8-Seats Table | ID: 9
8-Seats Table | ID: 10


Please input the TableId to delete table: 90
Table with ID: 90 does not exist!
```

Test Case 19:

21	Remove Table	No	/	/	/	Invalid format, ask for retry

Actual Output of Test Case 19:

```
Commands:
[1] Set Food Court's Opening and Closing Time
[2] Check Customer's Orders
[3] Check Customer's Reservation
[4] Add Restaurant
[5] Remove Restaurant
[6] Add Table
[7] Remove Table
[8] Logout

Please select your operations: 7

List of all Tables:
2-Seats Table | ID: 1
2-Seats Table | ID: 2
2-Seats Table | ID: 3
2-Seats Table | ID: 4
2-Seats Table | ID: 5
4-Seats Table | ID: 6
4-Seats Table | ID: 7
4-Seats Table | ID: 8
8-Seats Table | ID: 9
8-Seats Table | ID: 10


Please input the TableId to delete table: a
Error! Wrong input for selection! Please input an integer!
```

5.3 Sample Bug Report

Issue #C1

Title:	**Invalid Integer input in "Please choose the restaurant to order:"**
Reported Date:	11 November 2022
Severity Level:	**Minor**
Module:	CommandCustomerDineIn
Assigned To:	**CHENG Yin**
Status:	Resolved

Problem Description:

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 4 out of bounds for length 4
        at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
        at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
        at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:248)
        at java.base/java.util.Objects.checkIndex(Objects.java:372)
        at java.base/java.util.ArrayList.get(ArrayList.java:459)
        at CustomerModule.Ordering(CustomerModule.java:88)
        at CustomerModule.run(CustomerModule.java:62)
        at Main.main(Main.java:31)
```

Should output "Please choose the restaurant to order:" instead of terminating the program.

Steps to reproduce the bug:

```
Available restaurants:
[1] Pepper-Lunch
[2] Tam-Jai-Mi-Xian
[3] McDonald's
[4] KFC

Please choose the restaurant to order:
```

1. Login as Customers.
2. Choose to dine in.
3. Enter the number of people.
4. Choose to walk in.
5. Enter "5" under the above situation.

Comment By Developer:
Added a Try-Catch block to catch the IndexOutOfBoundsException and looped to another round of asking, "Please choose the restaurant to order:" if the customer input an invalid input.

Issue #M1

Title:	**Error in Delete Dish module when the menu is empty**
Reported Date:	13 November 2022
Severity Level:	**Minor**
Module:	Merchants, CommandMerchantModifyMenu
Assigned To:	**CHENG Yin**
Status:	Resolved

Problem Description:

```
Commands:
[1] Add Dish
[2] Delete Dish
[3] Edit Dish
[4] Cancel

Please select your operations: 2

Menu:

Please input the numbering of the dish to remove: 3
Index 2 out of bounds for length 0
```

Should output “Menu is empty, please try again” instead of outputting an error.

Steps to reproduce the bug:

1. Login as Merchants.
2. Choose to “[1] Modify Menu”.
3. Choose to Delete Dish as the above figure shows.
4. Continue the previous step until the menu is empty.
5. Then choose to Delete Dish as the above figure shows.

Comment By Developer:
Add two if clauses to restrict the Merchants.removeDish() to be executed only when the restaurant's menu is not empty.

End of Report