Table-Track: A Restaurant Reservation System

Sophia Pasha – 200471640
Seonyu Park – 200330151
Ramanpreet Singh – 200384219

August 1, 2024

Software Testing and Validation (ENSE 375)

University of Regina | Go far, together.

Agenda

- Introduction
- Problem Definition
- Why
- Design Requirements
- Solutions
- Testing and Demonstration
- Project Management
- Demo
- Conclusion and Future Scope

Introduction

- In the modern dining industry, efficient management of reservations is critical for enhancing customer satisfaction and optimizing restaurant operations.
- Many restaurants still rely on outdated, manual reservation systems that are prone to errors, overbooking, and inefficiencies.
- These systems often result in frustrated customers due to lost reservations, long wait times, and poor table management.
- Additionally, restaurant staff spends considerable time managing bookings manually, which detracts from their ability to provide excellent customer service.

Problem Definition

- Table Track will offer a command-line interface for customers and restaurant staff. Customers can make reservations, view available time slots, and receive confirmations, reducing errors and miscommunication. Automating reservations minimizes overbooking, saves time, and allows staff to focus on quality service. Staff can also add, update, and cancel reservations. As the restaurant grows, the system can handle more bookings without extra effort, ensuring consistent service quality. We will use various testing techniques to ensure the software reliability, accuracy, and effectiveness.

Why ?

- Save Time
- To Reduce errors and miscommunication
- Improve Customer Experience
- Reliability and Accuracy

Design Requirements

- Functions
  - Customer:
    - View available reservation times
    - Select an available time and number of people for a table
    - Enter name and contact information
    - Confirm and update reservation
  - Restaurant Staff:
    - Update number of people in a reservation
    - Update time of a reservation
    - View all reservations
    - Cancel a reservation
- Objectives
  - Customers can make reservations
  - Restaurant staff can view, update and cancel reservations

Design Requirements

- Constraints
  - Economic Factors: Our product is free to use, reducing costs for restaurants and increasing accessibility for all types of dining facilities
  - Sustainability and Environmental Factors: Our application reduces the need for paper and other supplies by allowing all reservation to be made on computers, contributing to environmental sustainability
  - Societal Impacts: Our application reduces employees' workload and stress, as customers can book their own reservations, without having to speak to an employee.
  - Reliability: Our system will be designed for reliability, ensuring consistent performance and minimizing time
  - Ethics: Our program will be designed with ethical considerations in mind, ensuring fair access and treatment for all users.

Solutions

- Solution #1

- Our first solution was a simple solution in which the users entered in a date, time and number of people for a reservation. The majority of our test cases failed as we only had the users enter minimal information. This solution did not allow users to view, edit and delete reservations.

Solutions

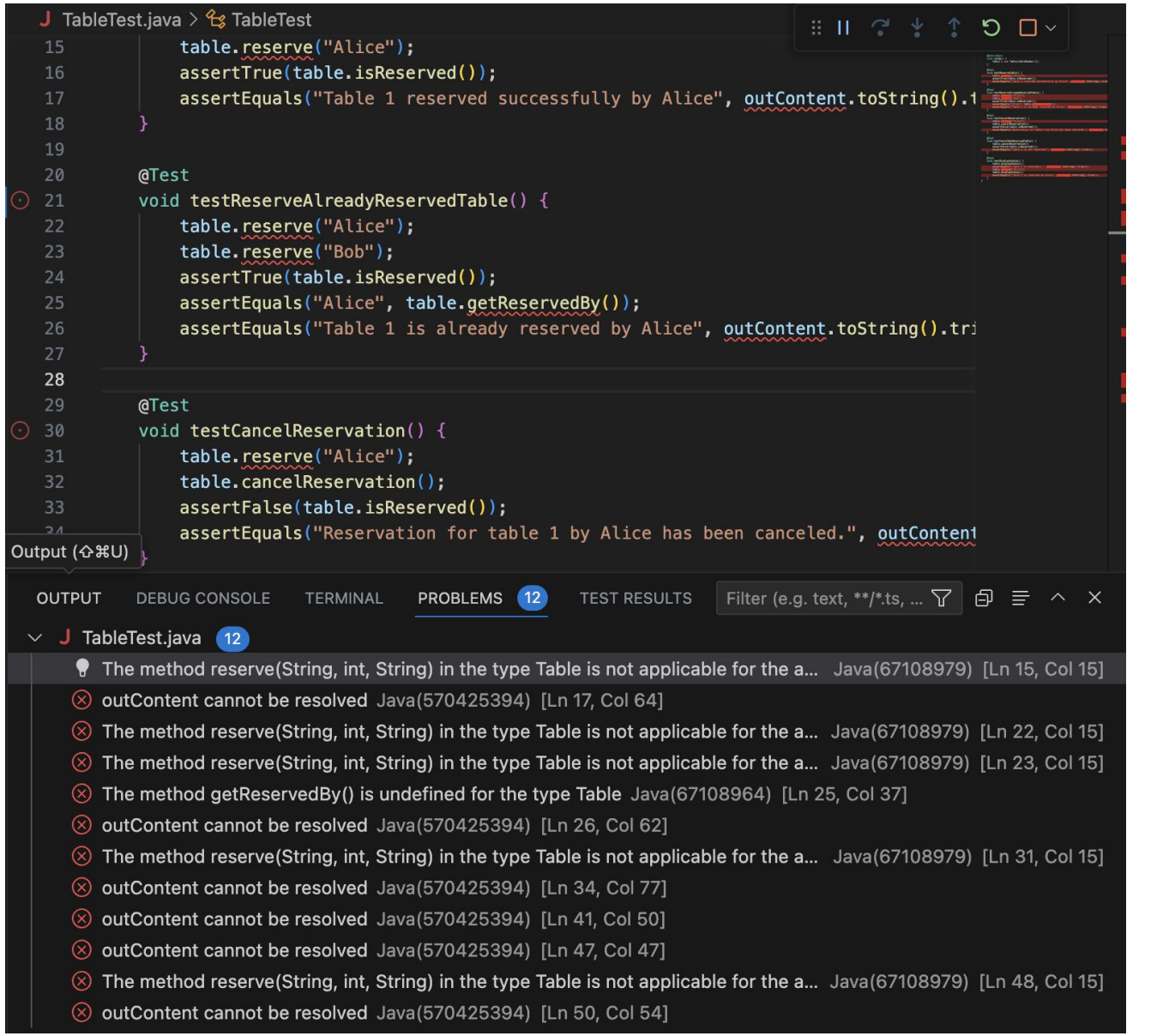
- Solution #2

- We refined our design to allow the user to enter all of their information. With this design, the customer test cases passed, however, the employee test cases failed as we had no way for employees to edit information. We also had no way for the user to choose whether they are an employee or a customer. To improve this design, we need to add a code that employees can enter in order to access the view, edit and delete methods.

Solutions

- Solution #2

  - Some tests are failing



Solutions

- Solution #3

  - our final solution includes the customer entering all of the information needed to book a reservation and allows the customers to view available times and update their reservation. It also allows the employees to make, view, edit and delete reservations.

Solutions

Comparison Table

	Solution 1	Solution 2	Solution 3
Economic factors	5	5	5
	Free to use	Free to use	Free to use
Sustainability and environmental factors	5	5	5
	Reservation done electronically, saving supplies	Reservation done electronically, saving supplies	Reservation done electronically, saving supplies
Societal impacts	2	3	5
	Minimal reduction in workload and stress for employees	Some reduction in workload and stress for employees	Significant reduction in workload and stress for employees. Customers handle most booking details
Reliability	0	0	5
	Low reliability. Most test failed	Improved reliability but still limited functionality	High reliability. All test cases passed. General features for both customers and employees
Ethics	3	4	5
	Limited consideration	Improve ethical consideration with limited options for employees	High ethical consideration. Respects user roles and provide access for employees
Total	15	17	25

Score 1 to 5: 1 is the lowest and 5 is the highest value.

Solutions

Comparison Summary

- Solution 1
  - Economic factors: free to use
  - Sustainability and environmental factors: reservation made on device saving supplies
  - Societal impacts: limited impact as minimal reduction in workload for employees
  - Reliability: low reliability as most test cases failed
  - Ethics: limited consideration as limited functionality
- This solution was the most basic and only allowed users to enter minimal information. Most test cases failed, indicating low reliability and limited functionality.

Solutions

Comparison Summary

- Solution 2
  - Economic factors: free to use
  - Sustainability and environmental factors: reservation made on device saving supplies
  - Societal impacts: improved societal impact as some reduction in workload for employees
  - Reliability: improved reliability as most test cases passed
  - Ethics: improved consideration as better control for employees
- This solution improved on solution 1 by providing employees with a limited set of options chosen by the employees. Although some improved reliability, some test cases failed.

Solutions

Comparison Summary

- Solution 3
  - Economic factors: free to use
  - Sustainability and environmental factors: reservation made on device saving supplies
  - Societal impacts: significant impact with major reduction in employees workload and stress
  - Reliability: high reliability and passed all test cases
  - Ethics: high ethical standards with users' roll

- The final solution provides the highest scores across all factors making it the most suitable solution for the project

Testing Techniques Used:

- Path testing
- Integration testing
- Decision table testing
- Use case testing
- Boundary value testing
- State transition testing
- Equivalence class testing

Path Testing Requirements

- Cover all possible paths in a function
- For the addReservation() function, in the ReservationManager class, three paths can be taken.
  - Valid reservation made
  - Invalid time slot entered
  - Past date entered

Path Testing

```java
@Test
public void testAddReservation_Valid() {
    LocalDate date = LocalDate.of(2024, 8, 10);
    LocalTime time = LocalTime.of(19, 30);
    int id = reservationManager.addReservation("Raman", "111-1111", date, time, 4);

    Reservation reservation = reservationManager.getReservationById(id);
    assertNotNull(reservation, "Reservation should be found."); // there should not be null
    assertEquals("Raman", reservation.getName());
}

@Test
public void testAddReservation_InvalidTimeSlot() {
    LocalDate date = LocalDate.of(2024, 8, 10);
    LocalTime time = LocalTime.of(19, 30);
    reservationManager.addReservation("Sophia", "111-1111", date, time, 4);

    int id = reservationManager.addReservation("Raman", "555-5678", date, time, 2);
    assertEquals(-1, id, "Should return -1 for an unavailable time slot."); // this time slot is already booked by sophia
}

@Test
public void testAddReservation_PastDate() {
    LocalDate pastDate = LocalDate.of(2024, 7, 25);
    LocalTime time = LocalTime.of(19, 30);
    int id = reservationManager.addReservation("Sophia", "111-1111", pastDate, time, 4);

    assertEquals(-1, id, "Should return -1 for a past date.");
}
```

Integration Testing Requirements

- Ensure that the Reservation class and ReservationManager class interact correctly with each other
- the testUpdateReservation_Valid() test case uses the ReservationManager class to add and update a reservation.
- The Reservation class is used to get the reservation ID, date, time and number of people and confirm they have been updated

Integration Testing Requirements

- Valid Reservation: Adding a valid reservation and checking if it is successfully created and stored
- Invalid Reservation Due to Time Slot Conflict: Attempting to add a reservation when the desried time slot is already booked
- Invalid Reservation Due to Past Date: Trying to add a reservation with a date that is in the past
- Valid Cancellation: Canceling an existing reservation and ensuring it is removed from the system
- Invalid Cancellation: Attempting to cancel a non-existent reservation
- Valid Update: Updating an existing reservation and verifying that the updated details are correctly stored
- Invalid Update: Trying to update a non-existent reservation
- Retrieve All Reservation: Adding multiple reservations and retrieving the list of all reservations to ensure they are correctly stored and retrieved.

Integration Testing Results

Test Case	Expected Outcome	Actual Outcome	Status
testAddReservation_Valid	reservation should be created and found	reservation was created and found	pass
testAddReservation_InvalidTimeSlot	should return -1 for an unavailable time slot	return -1 for an unavailable time slot	pass
testAddReservation_PastDate	should return -1 for a past date	return -1 for a past date	pass
testCancelReservation_Valid	reservation should be canceled and not found	reservation was canceled and not found	pass
testCancelReservation_Invalid	cancellation should fail for a non-existent reservation	cancellation failed for a non-existent reservation	pass
testUpdateReservation_Valid	reservation should be updated with new details	reservation was updated with new details	pass
testUpdateReservation_Invalid	update should fail for a non-existent reservation	update failed for a non-existent reservation	pass
testGetAllReservations	should retrieve all reservation	retrieved all reservations	

Integration Testing

Test case	Classes involved	Actual Outcome	Expected Outcome
Update reservation with valid new date	ReservationManager and Reservation	Reservation date is updated and retrievable.	Reservation date should be updated and retrievable.
Update reservation with valid new time	ReservationManager and Reservation	Reservation time is updated and retrievable.	Reservation time should be updated and retrievable.
Update reservation with different guests	ReservationManager and Reservation	Guests is updated and retrievable.	Guest size should be updated and retrievable.
Update reservation with all valid changes	ReservationManager and Reservation	All details is updated and retrievable.	All details should be updated and retrievable.

Integration Testing

```java
@Test
public void testUpdateReservation_Valid() {
    LocalDate date = LocalDate.of(2024, 8, 10);
    LocalTime time = LocalTime.of(19, 30);
    int id = reservationManager.addReservation("Raman", "111-1111", date, time, 4);

    LocalDate newDate = LocalDate.of(2024, 8, 11);
    LocalTime newTime = LocalTime.of(20, 00);
    boolean result = reservationManager.updateReservation(id, newDate, newTime, 5);

    assertTrue(result, "Reservation should be updated.");
    Reservation reservation = reservationManager.getReservationById(id);
    assertNotNull(reservation, "Reservation should be found.");
    assertEquals(newDate, reservation.getReservationDate());
    assertEquals(newTime, reservation.getReservationTime());
    assertEquals(5, reservation.getNumberOfPeople());
}
```

Decision Table Testing

- The purpose of the decision table testing was to ensure that the ReservationManager.java is able to handle various combinations of input and conditions when user is making a reservation.
- The tests will identify and verify how different input conditions affect the system behaviour and the corresponding actions

Decision Table Testing

		Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7
Conditions:	Valid date format	True	True	True	True	False	False	True
	Valid time format	True	False	True	True	True	False	False
	Past Date	False	False	False	True	False	False	True
	Time is already reserved	False	False	True	False	False	False	False
Actions:	Reservation made	X						
	Date is in past				X			X
	Invalid Date format					X	X	
	Invalid Time format		X				X	X
	Time is not available			X				

Decision Table Testing

```java
@Test
public void testValidDate() {
    String input = "10/08/2024\n"; // valid future date
    Util.setScanner(new Scanner(new ByteArrayInputStream(input.getBytes())));
    LocalDate expectedDate = LocalDate.parse("10/08/2024", DateTimeFormatter.ofPattern("dd/MM/yyyy"));
    LocalDate actualDate = Util.getDateFromUser();
    assertEquals(expectedDate, actualDate);
}

@Test
public void testInvalidFormatThenValidDate() {
    String input = "2024-08-10\n10/08/2024\n"; // invalid format followed by valid date
    Util.setScanner(new Scanner(new ByteArrayInputStream(input.getBytes())));
    LocalDate expectedDate = LocalDate.parse("10/08/2024", DateTimeFormatter.ofPattern("dd/MM/yyyy"));
    LocalDate actualDate = Util.getDateFromUser();
    assertEquals(expectedDate, actualDate);
}
```

```java
@Test
public void testValidTime() {
    String input = "14:30\n"; // valid time
    Util.setScanner(new Scanner(new ByteArrayInputStream(input.getBytes())));
    LocalTime expectedTime = LocalTime.parse("14:30", DateTimeFormatter.ofPattern("HH:mm"));
    LocalTime actualTime = Util.getTimeFromUser();
    assertEquals(expectedTime, actualTime);
}
```

```java
@Test
public void testAddReservation_PastDate() {
    LocalDate pastDate = LocalDate.of(2024, 7, 25);
    LocalTime time = LocalTime.of(19, 30);
    int id = reservationManager.addReservation("Sophia", "111-1111", pastDate, time, 4);

    assertEquals(-1, id, "Should return -1 for a past date.");
}
```

Use Case Testing

- Make a reservation as a customer

	Steps	Description
Main Success Scenario: A: Actor S: System	1.	S: Is the user a customer or staff?
	2.	A: Selects Customer
	3.	S: Display menu of options for user
	4.	A: User selects 1 to make a reservation
	5.	A: Enter name
	6.	A: Enter contact information
	7.	A: Enter date
	8.	A: Enter time
	9.	A: Enter number of people
	10.	S: Reservation made. Display reservation ID
	11.	A: Enter 5 to exit

Equivalence Class Testing

- Equivalence class testing was done to ensure that each input is correctly handled, depending on if it is a valid or invalid input.
- Inputs tested:
  - Valid future date
  - Invalid date format
  - Past date
  - Valid time
  - Invalid time
  - Valid number of people
  - Invalid number of people

Equivalence Class Tests and Results

Test Case	Input	Expected Output	Actual Output	Pass?
testValidFutureDate()	12/08/2024	12/08/2024	12/08/2024	✔
testInvalidDateFormat()	31-08-2024	Runtime Exception	Runtime Exception	✔
testPastDate()	25/07/2024	Runtime Exception	Runtime Exception	✔
testValidTimeSlot()	21:00	21:00	21:00	✔
testInvalidTimeFormat()	9 PM	Runtime Exception	Runtime Exception	✔
testValidNumberOfPeople()	5	5	5	✔
testInvalidNumberOfPeople()	0	Runtime Exception	Runtime Exception	✔

Equivalence Class Testing

- Example:

```java
@Test
public void testValidNumberOfPeople() {
    String input = "5\n";
    System.setIn(new ByteArrayInputStream(input.getBytes()));
    Scanner scanner = new Scanner(System.in);
    int expectedNumber = 5;
    int actualNumber = scanner.nextInt();
    assertEquals(expectedNumber, actualNumber);
}

@Test
public void testInvalidNumberOfPeople() {
    String input = "0\n";
    System.setIn(new ByteArrayInputStream(input.getBytes()));
    Scanner scanner = new Scanner(System.in);
    Util.setScanner(scanner);
    assertThrows(RuntimeException.class, () -> {
        int number = scanner.nextInt();
        if (number <= 0) {
            throw new RuntimeException("Invalid number of people");
        }
    });
```

Boundary Value Testing

- The purpose of the boundary value testing was to ensure that the Util.java handles edge case for date and time inputs. Our goal was to verify that the program can handle minimum and maximum valid inputs as well as invalid formats.
- Example:

```java
@Test
public void testMinBoundaryDate() {
    String input = "30/07/2024\n"; // minimum valid date (assuming today is before this date)
    System.setIn(new ByteArrayInputStream(input.getBytes()));
    Util.setScanner(new Scanner(System.in));
    LocalDate expectedDate = LocalDate.parse(text: "30/07/2024", DateTimeFormatter.ofPattern(pattern: "dd/MM/yyyy"));
    LocalDate actualDate = Util.getDateFromUser();
    assertEquals(expectedDate, actualDate);
}

@Test
public void testMaxBoundaryDate() {
    String input = "31/12/2100\n"; // maximum valid date
    System.setIn(new ByteArrayInputStream(input.getBytes()));
    Util.setScanner(new Scanner(System.in));
    LocalDate expectedDate = LocalDate.parse(text: "31/12/2100", DateTimeFormatter.ofPattern(pattern: "dd/MM/yyyy"));
    LocalDate actualDate = Util.getDateFromUser();
    assertEquals(expectedDate, actualDate);
}
```

Boundary Value Testing

Boundary Value Testing Table

The test were to cover the following scenario:

1. Min Date: the earliest valid date
2. Max Date: the latest valid date
3. Min Time: the earliest valid time
4. Max Time: the latest valid time
5. Invalid Time Format
6. Time with Leading Zero: time input a single digit hour without a leading zero

Test Case	Input	Expected Output	Actual Output	Status
testMinBoundaryDate()	30/07/2024	2024-07-30	2024-07-30	pass
testMaxBoundaryDate()	31/12/2100	2100-12-31	2100-12-31	pass
testMinBoundaryTime()	00:00	00:00	00:00	pass
testMaxBoundaryTime()	23:59	23:59	23:59	pass
testInvalidTimeFormat	24:00	00:00	00:00	pass
testTimewithLeadingZero	7:45	07:45	07:45	pass

Boundary Value Testing

Boundary Value Testing Result:

- testMinBoundaryDate(): passed. the system parses the minimum valid date
- testMaxBoundaryDate(): passed. the system handles the maximum valid date
- testMinBoundaryTime(): passed. the system handles the minimum valid time
- testMaxBoundaryTime(): passed. the system handles the minimum valid time
- testvalidTimeFormat(): passed. the system checks invalid format and handles followed by a valid time
- testTimewithLeadingZero(): passed. the system processes a time input with a single digit hour without a leading zero

State Transition Testing

State Transition Testing for the reservation process

The goal of state transition testing for the ReservationManagerStateTransitionTest.java was to ensure that the application is able to handle the different states and transition involved in the reservation process. We wanted to verify that the program correctly transition from initial state to gathering reservation details, checks time slot availability and appropriately handles both available and unavailable time slots and makes a reservation successfully when the slot is available to book or handels reservation failure when the slot is not available to book.

State Transition Testing

Example of ReservationManagerStateTransitionTest.java

```java
@Test
public void testSuccessfulReservation() {
    // Start -> InputDetails
    String input = "Seonyu Park\n123-4567\n30/07/2024\n18:00\n4\n";
    System.setIn(new ByteArrayInputStream(input.getBytes()));
    Util.setScanner(new Scanner(System.in));

    // InputDetails -> CheckAvailability -> ReservationMade
    LocalDate date = LocalDate.parse(text:"30/07/2024", DateTimeFormatter.ofPattern(pattern:"dd/MM/yyyy"));
    LocalTime time = LocalTime.parse(text:"18:00", DateTimeFormatter.ofPattern(pattern:"HH:mm"));
    int reservationId = reservationManager.addReservation(name:"Seonyu Park", contactInfo:"123-4567", date, time,

    // Check if the reservation was successful
    Reservation reservation = reservationManager.getReservationById(reservationId);
    assertNotNull(reservation);
    assertEquals("Seonyu Park", reservation.getName());
    assertEquals(date, reservation.getReservationDate());
    assertEquals(time, reservation.getReservationTime());
    assertEquals(4, reservation.getNumberOfPeople());
}
```

State Transition Testing

State Transition Testing

1. S1: Start -> S2: Gather Reservation Details: event that user initiated the reservation process and provide details such as name, contact number, date, time and number of people)
2. S2: Gather Reservation Details -> S3: Check Slot Availability: event that the program gathers all the necessary details and checks if the desired time slot is available
3. S3: Check Slot Availability -> S4: Reservation Made: event that if the time slot is available, the reservation is successfully booked
4. S3: Check Slot Availability -> S5: Reservation Failed: event that if the slot is not available, the reservation fails
5. S5: Reservation Failed -> S2: Gather Reservation Details for Retry: event that user retries the reservation process by providing new details

State Transition Testing

State Transition Testing Table

State Table for the Reservation Process

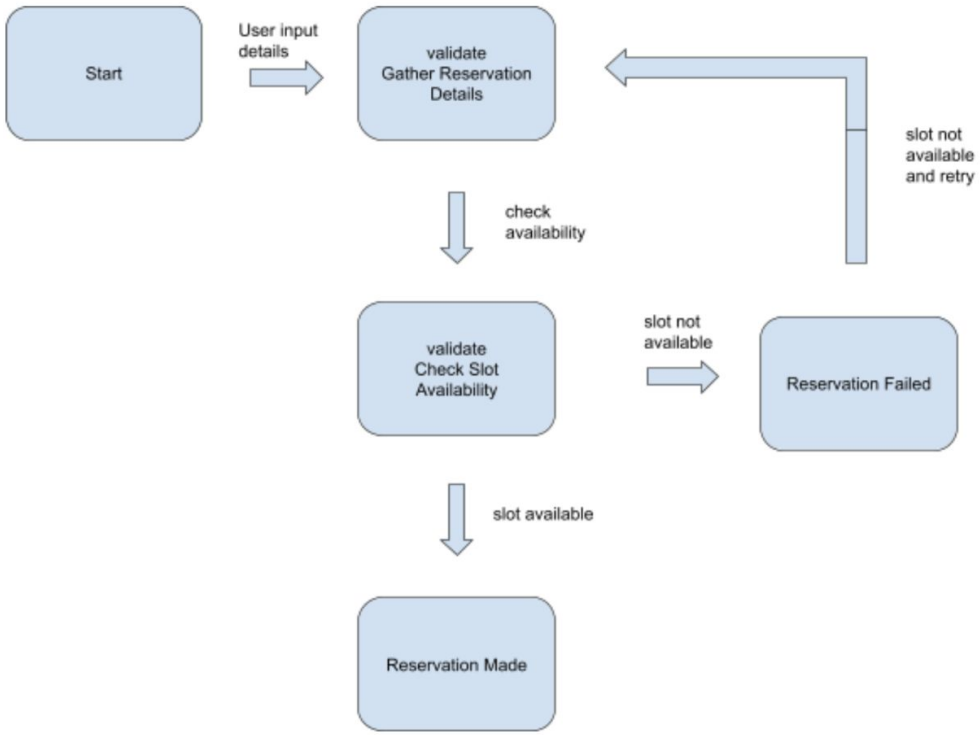
State	Event 1 (input details)	Event 2 (check availability)	Event 3 (slot available)	Event 4 (slot not available)
S1: start	S2 (gather reservation details)	-	-	-
S2: gather reservation details	-	S3 (check availability)	-	-
S3: check slot availability	-	-	S4 (reservation made)	S5 (reservation failed)
S4: reservation made	-	-	-	-
S5: reservation failed	S2 (gather reservation details for retry)	-	-	-

- invalid or Null transitions are represented as -

State Transition Testing

State Transition Diagram

State Transition Diagram for the Reservation Process



State Transition Testing

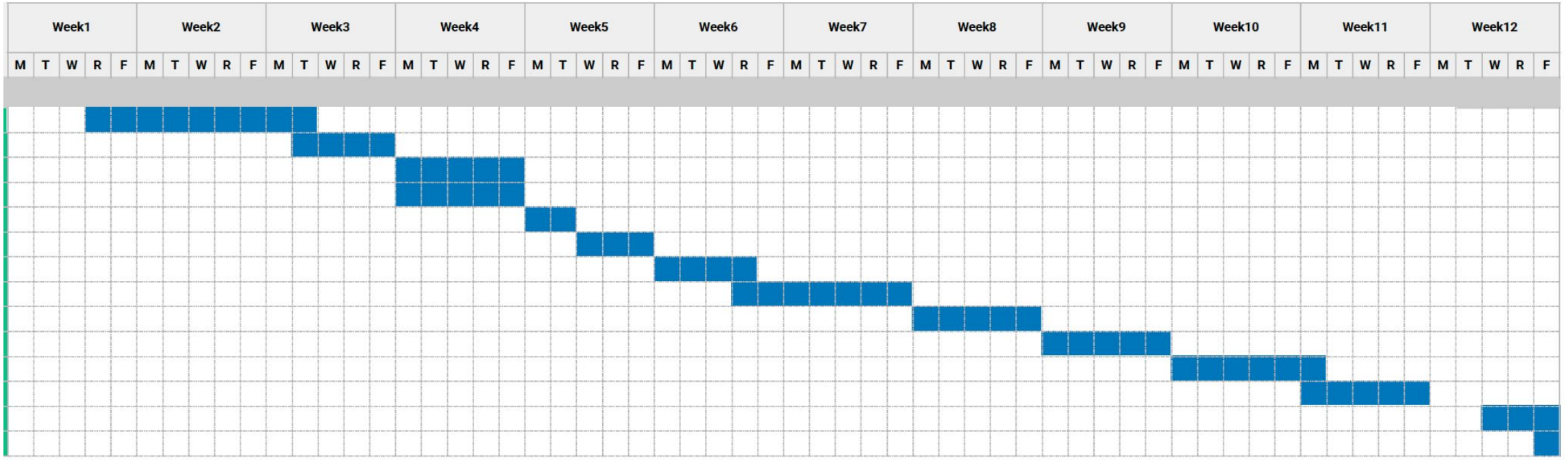
State Transition Testing Results:

Test Case	Expected State Transition	Actual State Transition	Status
testSuccessfulReservation()	S1-S2-S3-S4	S1-S2-S3-S4	pass
testFailedReservationDueToUnavailiableSlot()	S1-S2-S3-S5	S1-S2-S3-S5	pass
testFailedReservationThenSuccessful()	S1-S2-S3-S5-S2-S3-S4	S1-S2-S3-S5-S2-S3-S4	pass

Project Management

Task	Start Date	End Date	Days	Progress
Define the problem	5. 9. 2024	5. 21. 2024	12	100%
Background research	5. 21. 2024	5. 26. 2024	5	100%
Design constraints	5. 26. 2024	5. 31. 2024	5	100%
Specify requriments	5. 26. 2024	5. 31. 2024	5	100%
First prototype	6. 1. 2024	6. 4. 2024	3	100%
Testing	6. 4. 2024	6. 8. 2024	4	100%
Second prototype	6. 8. 2024	6. 13. 2024	5	100%
Testing	6. 13. 2024	6. 21. 2024	8	100%
Third prototype	6. 22. 2024	6. 30. 2024	8	100%
Testing	7. 1. 2024	7. 7. 2024	6	100%
Final implementation	7. 8. 2024	7. 15. 2024	7	100%
Testing	7. 14. 2024	7. 21. 2024	7	100%
Report	7. 24. 2024	7. 27. 2024	3	100%
Presentation	7. 30. 2024	7. 31. 2024	1	100%

Project Management



Demo

Conclusion

- **Enhanced Efficiency**: Table-Track streamlines reservation management, reducing errors and overbooking, thus enhancing customer satisfaction and optimizing restaurant operations.
- **Robust Testing**: Our rigorous testing suite ensures the reliability, accuracy, and effectiveness of the system, covering various testing techniques like path testing, data flow analysis, and integration testing.
- **Employee Empowerment**: By allowing customers to make and update their own reservations, employees can focus more on delivering excellent service, reducing their workload and stress.