Software Testing Assignment 1

Alireza Dastmalchi Saei

Student ID: *993613026*

University: *University of Isfahan*

Course: *Software Testing Course*

November 24, 2023

1 Introduction

In this comprehensive report, we embark on an exploration and evaluation of a Train Reservation System using a dual-approach testing methodology. Leveraging the power of JUnit and Gherkin, we conduct a thorough examination of the system's functionalities, ensuring its robustness, reliability, and adherence to business requirements.

JUnit, a widely adopted testing framework for Java applications, provides a solid foundation for scenario-based testing. Through the creation of five detailed scenarios, we utilize JUnit to meticulously assess various aspects of the Train Reservation System. These scenarios encompass critical functionalities such as City and Train Management, Trip Management, Ticket Booking and Cancellation, Delay Management, and Exchange Management (With Gherkin).

Complementing our JUnit tests, we employ Gherkin, a language-agnostic format, to write three additional scenarios. Gherkin facilitates Behavior-Driven Development (BDD) by allowing us to describe system behavior in a natural language format that is easily understandable by both technical and non-technical stakeholders.

This integrated approach ensures a comprehensive evaluation of the Train Reservation System, covering both the low-level unit testing provided by JUnit and the high-level behavioral testing facilitated by Gherkin. By combining these two testing methodologies, we aim to provide a robust and thorough validation of the system's functionality, ensuring its readiness for deployment in real-world scenarios.

2 Section 1: City and Train Management

2.1 Test Case 1.1: Add a New City

Description: Verify that a new city can be added to the system.
Preconditions:

- The system is running.
- The user is authorized to add a new city.
- There is new city to be added

Steps:

1. Add a new city to the system.

Expected Result: The city should be added to the list of cities in the system.

2.2 Test Case 1.2: Add a New Train

Description: Verify that a new train can be added to the system.
Preconditions:

- The system is running.
- The user is authorized to add a new train.
- There is a new train to be added.

Steps:

1. Add a new train to the system.

Expected Result: The new train should be added to the list of trains in the system.

3 Section 2: Trip Management

3.1 Test Case 2.1: Create a New Trip

Description: Verify that a new trip can be added to the system considering the specific constraints.
Preconditions:

- The system is running.
- The user is authorized to add a new trip.
- The new trip requirements are available to be created.

Steps:

1. Provide valid details for the new trip: Origin
2. Provide valid details for the new trip: Destination
3. Provide valid details for the new trip: Train
4. Provide valid details for the new trip: Departure Time
5. Provide valid details for the new trip: Arrival Time

Expected Result: The new trip should be created and registered in the system.

3.2 Test Case 2.2: Cancel Trip

Description: Verify that a trip can be canceled in the system.
Preconditions:

- The system is running.
- The user is authorized to cancel a trip.

Steps:

1. Select a trip to cancel

Expected Result: The selected trip should be canceled, and all associated tickets should also be canceled.

3.3 Test Case 2.3: Add a trip with conflicting timings to a train

Description: Verify that a trip with conflicting times cannot be added to the train trips list.
Preconditions:

- The system is running.
- The user is authorized to add a trip.

Steps:

1. Add a valid trip to train
2. Add a new trip that has intercepting times with previous trip

Expected Result: The second should not be added to the trips list of the train, and a trip exception must be thrown.

4 Section 3: Ticket Booking and Cancellation

4.1 Test Case 3.1: Book a Ticket

Description: Verify that a new ticket can be booked for a trip if the trip has not reached the maximum number of passengers.
Preconditions:

- The system is running.
- The user is authorized to book a ticket.
- There is an available trip.

Steps:

1. Select and available trip
2. Provide passenger name
3. Book a ticket

Expected Result: A new ticket should be booked for the selected trip.

4.2 Test Case 3.2: Cancel a Ticket

Description: Verify that a booked ticket can be canceled.
Preconditions:

- The system is running.
- The user has a booked ticket.

Steps:

1. Select a booked ticket

Expected Result: The selected ticket should be canceled, and the trip should be updated accordingly.

4.3 Test Case 3.3: Book a Ticket for a full trip

Description: Verify that a ticket for a full trip cannot be created.
Preconditions:

- The system is running.

Steps:

1. Create a ticket for a trip with max passengers

Expected Result: The ticket should not be booked and it must give an error.

4.4 Test Case 3.3: Book a Ticket for a full trip

Description: Verify that a ticket for a full trip cannot be created.
Preconditions:

- The system is running.

Steps:

1. Create a ticket for a trip with max passengers

Expected Result: The ticket should not be booked and it must give an error.

5 Section 4: Delay Management

5.1 Test Case 4.1: Add Departure Delay to a Trip

Description: Verify that a departure delay can be added to a trip, and it updates the real departure time.
Preconditions:

- The system is running.
- There is a trip available for delay.

Steps:

1. Select a trip
2. Add departure delay

Expected Result: The departure delay should be added to the trip, and the real departure time should be updated accordingly.

5.2 Test Case 4.2: Add Arrival Delay to a Trip

Description: Verify that an arrival delay can be added to a trip, and it updates the real arrival time.
Preconditions:

- The system is running.
- There is a trip available for delay.

Steps:

1. Select a trip
2. Add arrival delay

Expected Result: The arrival delay should be added to the trip, and the real arrival time should be updated accordingly.

5.3 Test Case 4.3: Add Departure Delay More Than Duration to Pass Arrival Date

Description: Verify that a departure date cannot pass arrival time after being delayed.
Preconditions:

- The system is running.
- There is a trip available for delay.

Steps:

1. Select a trip
2. Add departure delay more that duration

Expected Result: The departure delay should not be added to the trip, or the arrival time must be updated.

6 Section 5: Exchange Management (Cucumber)

6.1 Gherkin 5.1: Find All Exchangeable Tickets

Someone wants to exchange his/her ticket, available tickets for exchanges will be shown
Given → Exchange requirements are met
When → Request to see available tickets for exchange
Then → A list of all available tickets are shown

6.2 Gherkin 5.2: Get previous and Next Trip of a Train

Someone has a trip of a train and wants to see previous and next trip of that train
Given → Train has more that three trips and has a previous and next trip
When → Request to see available earlier and later trips of a train
Then → The correct successor and predecessor trips are shown

6.3 Gherkin 5.3: Exchange a Ticket successfully

Someone wants to exchange his/her ticket, the exchanging process must be correct
Given → Exchanging requirements are met
When → Request to exchange a ticket from a trip
Then → Exchange must be done for a trip

7 Code Explanation and Report

Overall 14 test have been written (11 with JUnit and 3 with Gherkin). We will go through each one of them and explain in details if required.

7.1 Setting Up Tests

Setting up JUnit and Cucumber in a Maven project involves adding the necessary dependencies to the project's pom.xml file. JUnit is a widely-used testing framework for Java, while Cucumber facilitates behavior-driven development using Gherkin syntax. Below is an example passage describing how to set up these dependencies in a Maven project:

Must add following dependencies:

```
        <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.8.1</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>io.cucumber</groupId>
            <artifactId>cucumber-junit</artifactId>
            <version>5.1.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>io.cucumber</groupId>
            <artifactId>cucumber-java</artifactId>
            <version>7.14.0</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>io.cucumber</groupId>
            <artifactId>cucumber-core</artifactId>
            <version>7.14.0</version>
        </dependency>

    </dependencies>
```

7.2 Preconditions

All JUnit test require and instance of TicketReservationSystem for controlling code, so in our Test class there is a **@beforeeach** for creating the instance before running every unittest.

```
@BeforeEach
public void setUp() {
    // Set up the default ZoneId and create a TicketReservationSystem
    zoneId = ZoneId.systemDefault();
    trs = new TicketReservationSystemImpl(zoneId);
    origin = new CityImpl( name: "City A");
    destination = new CityImpl( name: "City B");
}
```

Figure 1: Method with beforeeach

7.3 Section 1: City and Train Management

7.3.1 Test Case 1.1: Add a New City

The code for testing if a city can be added correctly to the city list is given below:
First, we create a list of cities with a size of 3. Then, a new city with the name of "California" is created and added to city list. With the variable *CityExists* and iterating through the list of all available cities, we can check if the city is in the list or not.

```java
@Test
public void addNewCity(){
    // Add a city to the system before executing the test
    City existingCity1 = new CityImpl( name: "Los Angeles");
    trs.addCity(existingCity1);
    City existingCity2 = new CityImpl( name: "Washington");
    trs.addCity(existingCity2);
    City existingCity3 = new CityImpl( name: "Texas");
    trs.addCity(existingCity3);

    City city = new CityImpl( name: "California");
    trs.addCity(city);
    List<City> cityList = trs.getCities();

    boolean cityExists = false;
    for (City c : cityList) {
        if (c.getName().equals(city.getName())) {
            cityExists = true;
            break;
        }
    }
    assertTrue(cityExists);
}
```

Figure 2: Test Case 1.1

7.3.2 Test Case 1.2: Add a New Train

This method, checks whether a new can be added in our TrainReservationSystem correctly or not. The snippet of code can be seen below:

```java
@Test
public void AddNewTrain(){
    Train train1 = new TrainImpl( name: "Bullet Train",  maxPassengers: 100);
    trs.addTrain(train1);
    Train train2 = new TrainImpl( name: "Bullet Train",  maxPassengers: 100);
    trs.addTrain(train2);
    Train train3 = new TrainImpl( name: "Bullet Train",  maxPassengers: 100);
    trs.addTrain(train3);

    Train train = new TrainImpl( name: "Bullet Train",  maxPassengers: 100);
    trs.addTrain(train);

    List<Train> trains = trs.getAllTrains();

    boolean trainExists = false;
    for (Train t : trains) {
        if (t.getName().equals(train.getName())) {
            trainExists = true;
            break;
        }
    }
    assertTrue(trainExists);
}
```

Figure 3: Test Case 1.2

As can be seen, in this test method, an initial list of 3 trains is created. Then, a new instance of train with the name of *BulletTrain* is created and added to the list of trains. The foreach loop after adding the train, checks for the correction of process of adding new trains. The value of *trainExists* must be true after iterating *trains* list.

7.4 Section 2: Trip Management

7.4.1 Test Case 2.1: Create a New Trip

This unittest is for testing the *createTrip* method in the class "TrainReservationSystem". In the code, 2 trips are created and the expected list length of the trips is asserted to be 2.

```
@Test
public void createNewValidTrip() throws TripException {
    City origin1 = new CityImpl( name: "City A");
    City destination1 = new CityImpl( name: "City B");
    Train train1 = new TrainImpl( name: "Express Train",  maxPassengers: 200);
    Instant departureTime1 = TimeManagement.createInstant( dateTimeString: "2023-11-25 10:00", zoneId);
    Instant arrivalTime1 = TimeManagement.createInstant( dateTimeString: "2023-11-28 14:00", zoneId);

    Trip trip1 = trs.createTrip(origin1, destination1, train1, departureTime1, arrivalTime1);


    City origin2 = new CityImpl( name: "City C");
    City destination2 = new CityImpl( name: "City D");
    Train train2 = new TrainImpl( name: "Local Train",  maxPassengers: 100);
    Instant departureTime2 = TimeManagement.createInstant( dateTimeString: "2023-12-02 08:00", zoneId);
    Instant arrivalTime2 = TimeManagement.createInstant( dateTimeString: "2023-12-02 10:00", zoneId);

    Trip trip2 = trs.createTrip(origin2, destination2, train2, departureTime2, arrivalTime2);

    List<Trip> trips = trs.getAllTrips();
    assertEquals(trips.toArray().length,  actual: 2);
}
```

Figure 4: Test Case 2.1

7.4.2 Test Case 2.2: Cancel Trip

This test case is for checking if a valid trip can be canceled correctly or not.

```
@Test
public void cancelAnExistingTrip() throws TripException {
    Train train = new TrainImpl( name: "Express Train",  maxPassengers: 200);
    Instant departureTime = TimeManagement.createInstant( dateTimeString: "2023-11-25 10:00", zoneId);
    Instant arrivalTime = TimeManagement.createInstant( dateTimeString: "2023-11-28 14:00", zoneId);

    Trip trip = trs.createTrip(origin, destination, train, departureTime, arrivalTime);

    trs.cancelTrip(trip);
    List<Trip> trips = trs.getAllTrips();
    assertEquals(trips.toArray().length,  actual: 0);
}
```

Figure 5: Test Case 2.2

In the code above, first we create an instance of objects needed for creating a trip, After creating the trip, we cancel the same trip created before and then check for the length of trip lists to be reduced by 1 (In this case: 0).

7.5 Section 3: Ticket Booking and Cancellation

7.5.1 Test Case 3.1: Book a Ticket

In this test case, we verify the system's ability to successfully book a ticket for a specified trip. The test involves creating a trip, attempting to book a ticket, and ensuring that the booked ticket is correctly registered in the system.

```java
@Test
public void bookingATicket() throws TripException, ReservationException {
    Train train = new TrainImpl( name: "Express Train",  maxPassengers: 200);
    Instant departureTime = TimeManagement.createInstant( dateTimeString: "2023-11-25 10:00", zoneId);
    Instant arrivalTime = TimeManagement.createInstant( dateTimeString: "2023-11-28 14:00", zoneId);

    Trip trip = trs.createTrip(origin, destination, train, departureTime, arrivalTime);
    trip.bookTicket( passengerName: "Alireza");

    List<Ticket> tickets = trs.getAllBookedTickets();
    List<Ticket> actual_tickets = trip.getBookedTickets();

    assertEquals(tickets, actual_tickets);
}
```

Figure 6: Test Case 3.1

7.5.2 Test Case 3.2: Cancel a Ticket

This test case assesses the functionality of canceling a previously booked ticket. After booking a ticket in a given trip, the system is tested to confirm that the cancellation process correctly removes the ticket from the booked tickets list.

```java
@Test
public void cancelATicket() throws TripException, ReservationException{
    Train train = new TrainImpl( name: "Express Train",  maxPassengers: 200);
    Instant departureTime = TimeManagement.createInstant( dateTimeString: "2023-11-25 10:00", zoneId);
    Instant arrivalTime = TimeManagement.createInstant( dateTimeString: "2023-11-28 14:00", zoneId);

    Trip trip = trs.createTrip(origin, destination, train, departureTime, arrivalTime);
    trip.bookTicket( passengerName: "Alireza");

    List<Ticket> tickets = trs.getAllBookedTickets();
    trip.cancelTicket(tickets.get(0));

    boolean ticketIsCancelled = trip.getBookedTickets().isEmpty();

    List<Ticket> canceled = trs.getAllCancelledTickets();

    assertTrue(ticketIsCancelled);
    assertFalse(canceled.isEmpty());
}
```

Figure 7: Test Case 3.2

7.5.3 Test Case 3.3: Book a Ticket for a Full Trip

This test examines the system's behavior when attempting to book a ticket for a trip that has reached its full capacity. It checks whether the system appropriately handles scenarios where the available seats on a train are already fully booked.

```
@Test
public void bookAnInvalidTicket() throws TripException, ReservationException{
    Train train = new TrainImpl( name: "Express Train",  maxPassengers: 3);
    Instant departureTime = TimeManagement.createInstant( dateTimeString: "2023-11-25 10:00", zoneId);
    Instant arrivalTime = TimeManagement.createInstant( dateTimeString: "2023-11-28 14:00", zoneId);


    Trip trip = trs.createTrip(origin, destination, train, departureTime, arrivalTime);
    trip.bookTicket( passengerName: "Alireza Saei");
    trip.bookTicket( passengerName: "David Goggins");
    trip.bookTicket( passengerName: "Donald Trump");

    assertThrows(ReservationException.class, () -> {
        trip.bookTicket( passengerName: "Andrew Tate");
    });
}
```

Figure 8: Test Case 3.3

7.6 Section 4: Delay Management

7.6.1 Test Case 4.1: Add Departure Delay to a Trip

This test case verifies the system's ability to handle and correctly reflect a departure delay for a given trip. The test involves creating a trip, adding a departure delay, and checking if the system accurately records and applies the delay to the trip's schedule.

```
@Test
public void addDepartureDelayToATrip() throws TripException{
    Train train = new TrainImpl( name: "Express Train",  maxPassengers: 3);
    Instant departureTime = TimeManagement.createInstant( dateTimeString: "2023-11-25 10:00", zoneId);
    Instant arrivalTime = TimeManagement.createInstant( dateTimeString: "2023-11-28 14:00", zoneId);

    Trip trip = trs.createTrip(origin, destination, train, departureTime, arrivalTime);
    Duration dur_day = Duration.ofDays(1);
    Duration dur_hour = Duration.ofHours(2);
    Duration dur_merged = dur_day.plus(dur_hour);
    trip.addDepartureDelay(dur_merged);

    assertTrue(trs.getAllTrips().get(0).isDelayed());
    assertEquals(trip.getDepartureDelay(), dur_merged);
    assertEquals(trip.findRealDepartureTime(), TimeManagement.createInstant( dateTimeString: "2023-11-26 12:00", zoneId));
    assertEquals(trip.getPlannedDepartureTime(), departureTime);
}
```

Figure 9: Test Case 4.1

7.6.2 Test Case 4.2: Add Arrival Delay to a Trip

In this test case, the system's response to adding an arrival delay to a trip is evaluated. After introducing an arrival delay, the test checks whether the system appropriately updates the trip's arrival time, considering the delay.

```java
@Test
public void addArrivalDelayToATrip() throws TripException{
    Train train = new TrainImpl( name: "Express Train",  maxPassengers: 10);
    Instant departureTime = TimeManagement.createInstant( dateTimeString: "2023-11-25 10:00", zoneId);
    Instant arrivalTime = TimeManagement.createInstant( dateTimeString: "2023-11-28 14:00", zoneId);

    Trip trip = trs.createTrip(origin, destination, train, departureTime, arrivalTime);
    Duration dur_day = Duration.ofDays(1);
    Duration dur_hour = Duration.ofHours(2);
    Duration dur_merged = dur_day.plus(dur_hour);
    trip.addArrivalDelay(dur_merged);

    // There is an error in isDelayed() for checking delay for arrivalTime
    assertTrue(trs.getAllTrips().get(0).isDelayed());
    assertEquals(trip.getArrivalDelay(), dur_merged);
    assertEquals(trip.findRealArrivalTime(), TimeManagement.createInstant( dateTimeString: "2023-11-29 16:00", zoneId));
    assertEquals(trip.getPlannedArrivalTime(), arrivalTime);
}
```

Figure 10: Test Case 4.2

7.6.3 Test Case 4.3: Add Departure Delay More Than Duration to Pass Arrival Date

This test examines the system's behavior when attempting to add a departure delay that surpasses the duration needed to reach the trip's arrival date. The goal is to ensure that the system handles such cases gracefully, preventing unrealistic delays that would exceed the arrival date.

```
@Test
public void addDepartureDelayToATripMoreThanDuration() throws TripException{
    Train train = new TrainImpl( name: "Express Train",  maxPassengers: 10);
    Instant departureTime = TimeManagement.createInstant( dateTimeString: "2023-11-25 10:00", zoneId);
    Instant arrivalTime = TimeManagement.createInstant( dateTimeString: "2023-11-27 10:00", zoneId);

    Trip trip = trs.createTrip(origin, destination, train, departureTime, arrivalTime);
    Duration dur = Duration.ofDays(4);
    trip.addDepartureDelay(dur);

    assertTrue(trs.getAllTrips().get(0).isDelayed());
    assertEquals(trip.getDepartureDelay(), dur);
    assertEquals(trip.findRealDepartureTime(), TimeManagement.createInstant( dateTimeString: "2023-11-29 10:00", zoneId));
    assertEquals(trip.getPlannedDepartureTime(), departureTime);
    // Now departure time is after arrival time that is not true
    assertTrue(trip.findRealArrivalTime().isAfter(trip.findRealDepartureTime()));
}
```

Figure 11: Test Case 4.2

7.7 Section 5: Exchange Management

7.7.1 Gherkin 5.1: Find All Exchangeable Tickets

This Gherkin scenario outlines the steps to find all tickets that are available for exchange within the system. It likely involves identifying criteria for exchange eligibility and presenting a list of tickets that meet these criteria.

```
@Given("Exchange requirements are met")
public void exchange_requirements_are_met() throws TripException{
    trs = new TicketReservationSystemImpl(zoneId);

    City origin1 = new CityImpl( name: "City A");
    City destination1 = new CityImpl( name: "City B");
    Train train1 = new TrainImpl( name: "Express Train",  maxPassengers: 20);
    Instant departureTime1 = Instant.parse( text: "2023-11-25T10:00:00Z");
    Instant arrivalTime1 = Instant.parse( text: "2023-11-28T14:00:00Z");
    trip1 = trs.createTrip(origin1, destination1, train1, departureTime1, arrivalTime1);

    Train train2 = new TrainImpl( name: "Bullet Train",  maxPassengers: 20);
    Instant departureTime2 = Instant.parse( text: "2023-11-27T10:00:00Z");
    Instant arrivalTime2 = Instant.parse( text: "2023-11-29T14:00:00Z");
    trip2 = trs.createTrip(origin1, destination1, train2, departureTime2, arrivalTime2);
}

 Alireza Saei
@When("Request to see available tickets for exchange")
public void request_to_see_available_tickets_for_exchange() throws ReservationException{
    Ticket t1 = trip1.bookTicket( passengerName: "Alireza");
    tripsList = trs.findPossibleExchanges(t1);
}

 Alireza Saei
@Then("A list of all available tickets are shown")
public void a_list_of_all_available_tickets_are_shown() { assertEquals(tripsList.toArray().length,  actual: 1); }
```

Figure 12: Gherkin 5.1

7.7.2 Gherkin 5.2: Get previous and Next Trip of a Train

This Gherkin scenario focuses on retrieving information about the previous and next trips of a train. It involves interacting with the system to obtain details about the chronological order of train trips, helping users plan and manage their travel effectively.

```
@Given("Train has more that three trips and has a previous and next trip")
public void train_has_more_that_three_trips_and_has_a_previous_and_next_trip() throws TripException {
    trs = new TicketReservationSystemImpl(zoneId);
    train = new TrainImpl( name: "Bullet Train",  maxPassengers: 20);

    City origin1 = new CityImpl( name: "City A");
    City destination1 = new CityImpl( name: "City B");
    Instant departureTime1 = Instant.parse( text: "2023-12-10T10:00:00Z");
    Instant arrivalTime1 = Instant.parse( text: "2023-12-15T14:00:00Z");
    trip1 = trs.createTrip(origin1, destination1, train, departureTime1, arrivalTime1);

    // Created 2 more (Deleted for screenshot)
}

 Alireza Saei
@When("Request to see available earlier and later trips of a train")
public void request_to_see_available_earlier_and_later_trips_of_a_train() throws TripException{
    previous_trip = trs.findPreviousTripOfTrain(train, trip2);
    next_trip = trs.findNextTripOfTrain(train, trip2);
}

 Alireza Saei
@Then("The correct successor and predecessor trips are shown")
public void the_correct_successor_and_predecessor_trips_are_shown() {
    assertTrue(previous_trip.isPresent());
    assertTrue(next_trip.isPresent());

    assertEquals(previous_trip.get(), trip1);
    assertEquals(next_trip.get(), trip3);
}
```

Figure 13: Gherkin 5.2

7.7.3 Gherkin 5.3: Exchange a Ticket successfully

This Gherkin scenario describes the steps for successfully exchanging a ticket. It includes the process of selecting a ticket for exchange, verifying its eligibility, and executing the exchange operation, ensuring that the system handles the exchange seamlessly.

```
@Given("Exchanging requirements are met")
public void exchanging_requirements_are_met() throws TripException, ReservationException{
    trs = new TicketReservationSystemImpl(zoneId);

    City origin1 = new CityImpl( name: "City A");
    City destination1 = new CityImpl( name: "City B");
    Train train1 = new TrainImpl( name: "Express Train",  maxPassengers: 20);
    Instant departureTime1 = Instant.parse( text: "2023-11-25T10:00:00Z");
    Instant arrivalTime1 = Instant.parse( text: "2023-11-28T14:00:00Z");
    trip = trs.createTrip(origin1, destination1, train1, departureTime1, arrivalTime1);

    t = trip.bookTicket( passengerName: "Alireza Saei");
    trip.bookTicket( passengerName: "Jeff Bezos");
    trip.bookTicket( passengerName: "Arthur Morgan");
}

 Alireza Saei
@When("Request to exchange a ticket from a trip")
public void request_to_exchange_a_ticket_from_a_trip() throws ReservationException {
    exchanged_ticket = t.exchangeTicket(trip);
}

 Alireza Saei
@Then("Exchange must be done for a trip")
public void exchange_must_be_done_for_a_trip() { assertTrue(t.isCancelled()); }
```

Figure 14: Gherkin 5.3

8 Explanation Video Link

Click here to see the video recorded for this Assignment.