Contents

Program UML

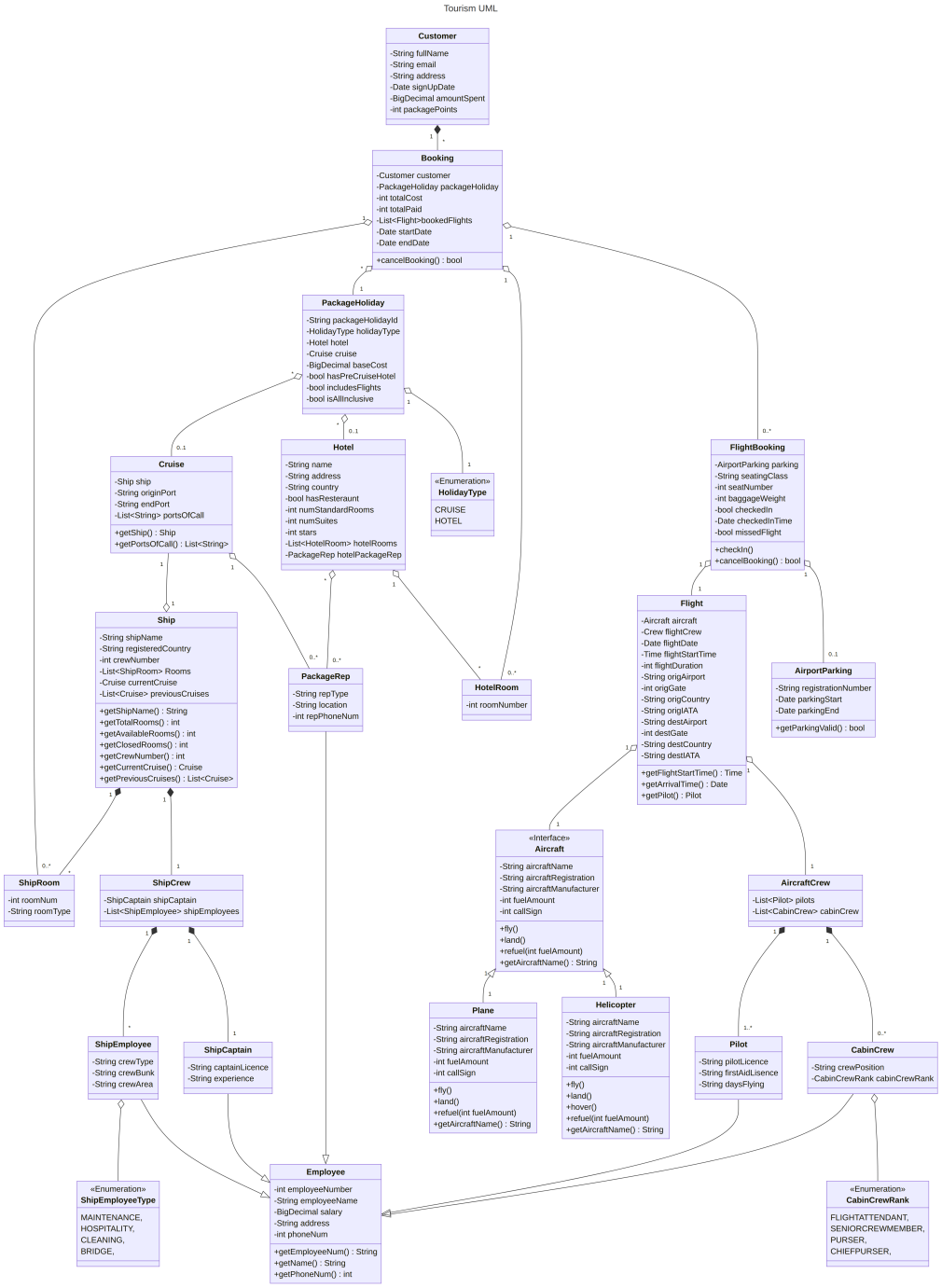

Figure 1: UML Representation of Program

Testing Report

Introduction

Testing pyramid

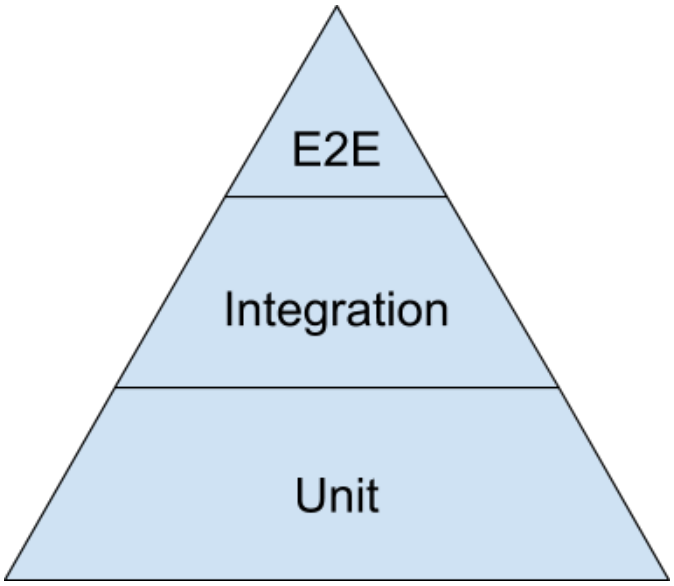

Figure 2: Testing Pyramid (Wacker, 2015)

The test structure of the program will follow the testing pyramid strategy created by Mike Cohn (2009). His initial interpretation was a pyramid with 3 levels. The bottom being unit testing, then service testing, and finally, at the top, user interface testing. Various interpretations have stemmed from Mike Cohn's original design, renaming the pyramid's stages or by adding additional layers. However, it is agreed that the lowest layers receive the highest time and processing investment with the largest number of tests. As you ascend the pyramid's layers, the tests become more infrequent with lower investment. As seen in figure 2, unit tests would have the highest amount of tests; whereas, end-to-end (E2E) and user interface tests number far fewer, running less frequently. This is because end-to-end tests, whilst comprehensive, have several disadvantages. Cohn (2009) lists the drawbacks as, fragility, time cost, and processing cost. E2E tests are brittle; small changes in the program's user interface could break several tests. Repeated fixes create discontent and a reluctance to fix the broken test, leaving it useless. Moreover, effective, non-brittle E2E testing increases time costs, making a large E2E test suite impractical. Finally, E2E testing costs more computationally than unit and integration testing, therefore tests can not run as frequently, decreasing daily coverage compared to unit and integration testing. On the other hand, unit testing, despite being quick, only deals with small independent slices of the program, missing tests concerning external services and how they integrate with each other. Therefore,

service, or integration testing, acts middleman to avoid testing external dependencies through the user interface. External dependencies, such as databases, APIs, or user inputs are tested without the performance degradation of the user interface.

Unit tests

Integration tests

Narrow

Wide

End to end testing

Input validation etc.

Implementation with continuous integration services

Conclusion

First areas of testing.

The first stages of testing will occur during the development and programming process. The project will make extensive use of unit tests; these tests involve writing isolated automated tests that target small sections of the program, known as units.

Creation of the unit test involves developing a criteria, referred to the test case.

The unit returns output that is compared to the test's criteria that is known to be correct by the developers.

Ensuring that these small units of the program correct

Testing lots of small units that integrate to form a complex program reduces the amount of uncertain variables compared to a large monolithic test of the end result.

Furthermore,

Within this project, the Java unit testing framework JUnit will be used.

Building on top of unit testing, continuous integration ensures that all developers within the organisation:

- feature flags

- ui testing
- test lab
- testing pyramid
- e2e testing - operational - max interaction - require running services - automated ui testing server
- integration testing - testing api code interactions, database connection
- black box testing - less interaction -
- martin folwer testing articles

Should the service require any external APIs. These could be tracking APIs offered by various airports, or apis offered by the FIA (British equivalent) to ensure that planes are correctly en route. Then the implementation of contact testing can ensure that updates to API returns and interfaces are quickly recognised and corrected. To ensure that the service does not suffer for too long.

End to end user testing.

`JUnit5` test for checking in process in `FlightBooking.java`,

```java
public boolean checkIn() {
    if (LocalDateTime.now().isAfter(flight.getFlightStart())) {
        hasMissedFlight = true;
        return false;
    }
    hasCheckedIn = true;
    checkInTime = LocalDateTime.now();
    return true;
}
```

The unit test created to test the `checkIn` method in the `FlightBooking` class.

```java
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

import java.time.LocalDateTime;

import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;

class FlightBookingTest {
    private Flight testFlight;
    private FlightBooking testFlightBooking;


    @Test
    @DisplayName("Check-in process with an on-time flight booking")
    public void checkIn() {
        LocalDateTime earlyDateTime = LocalDateTime.now().plusYears(1);
        /*
```

```
20            Create a mocked class of testFlight testFlight could use
21            an API for plane tracking, so creating a stub ensures that
22            the test is independent
23           */
24          testFlight = Mockito.mock(Flight.class);
25          // Set what to return when getFlightStart() is called
26          Mockito.when(testFlight.getFlightStart()).thenReturn(
27              earlyDateTime
28          );
29
30          // Call the check-in process
31          testFlightBooking = new FlightBooking(
32              testFlight, "economy", 12, 200
33          );
34          assertTrue(testFlightBooking.checkIn());
35      }
36
37      @Test
38      @DisplayName("Check-in process with a late flight booking")
39      public void checkInLate() {
40          LocalDateTime lateDateTime = LocalDateTime.now().minusYears(1);
41
42          // Create another mocked class
43          testFlight = Mockito.mock(Flight.class);
44          // Set return time to the future to fail check-in
45          Mockito.when(testFlight.getFlightStart()).thenReturn(
46              lateDateTime
47          );
48
49          // Call the check-in process
50          testFlightBooking = new FlightBooking(
51              testFlight, "economy", 12, 200
52          );
53          assertFalse(testFlightBooking.checkIn());
54      }
55  }
```

Narrow integration test with a PostgreSQL database.

```
1   import org.junit.jupiter.api.Test;
2   import org.junit.jupiter.api.AfterAll;
3   import org.junit.jupiter.api.BeforeAll;
4   import org.junit.jupiter.api.BeforeEach;
5   import org.junit.jupiter.api.DisplayName;
6   import org.testcontainers.containers.PostgreSQLContainer;
7
8
9   /**
10   * Test the program's integration with a database
11   */
12  public class CustomerIntegrationTest {
```

```
13
14      /*
15       Create a temporary database container to test a database's
16       integration with the program
17      */
18      static PostgreSQLContainer<?> postgres =
19          new PostgreSQLContainer<>("postgres:16-alpine");
20
21      Customer customer;
22      // CustomerDatabase customerDatabase;
23
24      /**
25       * Start the database
26       */
27      @BeforeAll
28      public static void startContainer() {
29          postgres.start();
30      }
31
32      /**
33       * Stop the database container
34       */
35      @AfterAll
36      public static void stopContainer() {
37          postgres.stop();
38      }
39
40      /**
41       * Connect to the database and proceed to clean it of previous
42       * entries
43       */
44      @BeforeEach
45      public void setUpCleanDb() {
46          customerDatabase = new CustomerDatabase(postgres.getJbdcUrl());
47          customerDatabase.deleteAll();
48      }
49
50      /**
51       * Test that {@code customerDatabase} integrates with and adds
52       * the customer to the database
53       */
54      @Test
55      @DisplayName("Database create customer")
56      public void dbCreateCustomer() {
57          customer = new Customer("Jake Real", "jakeemail@email.com",
58              "3 Cool Road, Swandiff"
59          );
60          // Add the created customer to the database
61          customerDatabase.addCustomer(customer)
62          // Test that customer was created in the database
63          assertEquals("Jake Real", customerDatabase.getCustomerByName(
```

```
64                 "Jake Real").getFullName()
65             );
66         }
67
68         /**
69          * Test that {@code customerDatabase} integrates with and deletes
70          * the customer from the database
71          */
72         @Test
73         @DisplayName("Database delete customer")
74         public void dbDeleteCustomer() {
75             customer = new Customer("Jake Real", "jakeemail@email.com",
76                 "3 Cool Road, Swandiff"
77             );
78             // Add the created customer to the database
79             customerDatabase.addCustomer(customer);
80             // Test that customer was created in the database
81             assertEquals("Jake Real", customerDatabase.getCustomerByName(
82                 "Jake Real").getFullName()
83             );
84             // Delete the customer from the database
85             customerDatabase.deleteCustomerById(customer.getId);
86             // Check that there are no customers
87             assertEquals(0, customerDatabase.getAllCustomers().size());
88         }
89
90 }
```

References

Cohn, M. (2009) *Succeeding with agile*. Addison-Wesley Professional., pp. 312–323.

Kapelonis, J. (2024) *Stubbing and mocking with mockito and JUnit*. Available at: https://semaphoreci.com/community/tutorials/stubbing-and-mocking-with-mockito-2-and-junit.

testcontainers ([no date][a]) *General container runtime requirements*. Available at: https://java.testcontainers.org/supported_docker_environment.

testcontainers ([no date][c]) *Testcontainers container lifecycle management using JUnit 5*. Available at: https://testcontainers.com/guides/testcontainers-container-lifecycle.

testcontainers ([no date][b]) *Testcontainers container lifecycle management using JUnit 5*. Available at: https://testcontainers.com/guides/testcontainers-container-lifecycle.

Wacker, M. (2015) *Just say no to more end-to-end tests*. Available at: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html.