Contents

Program UML

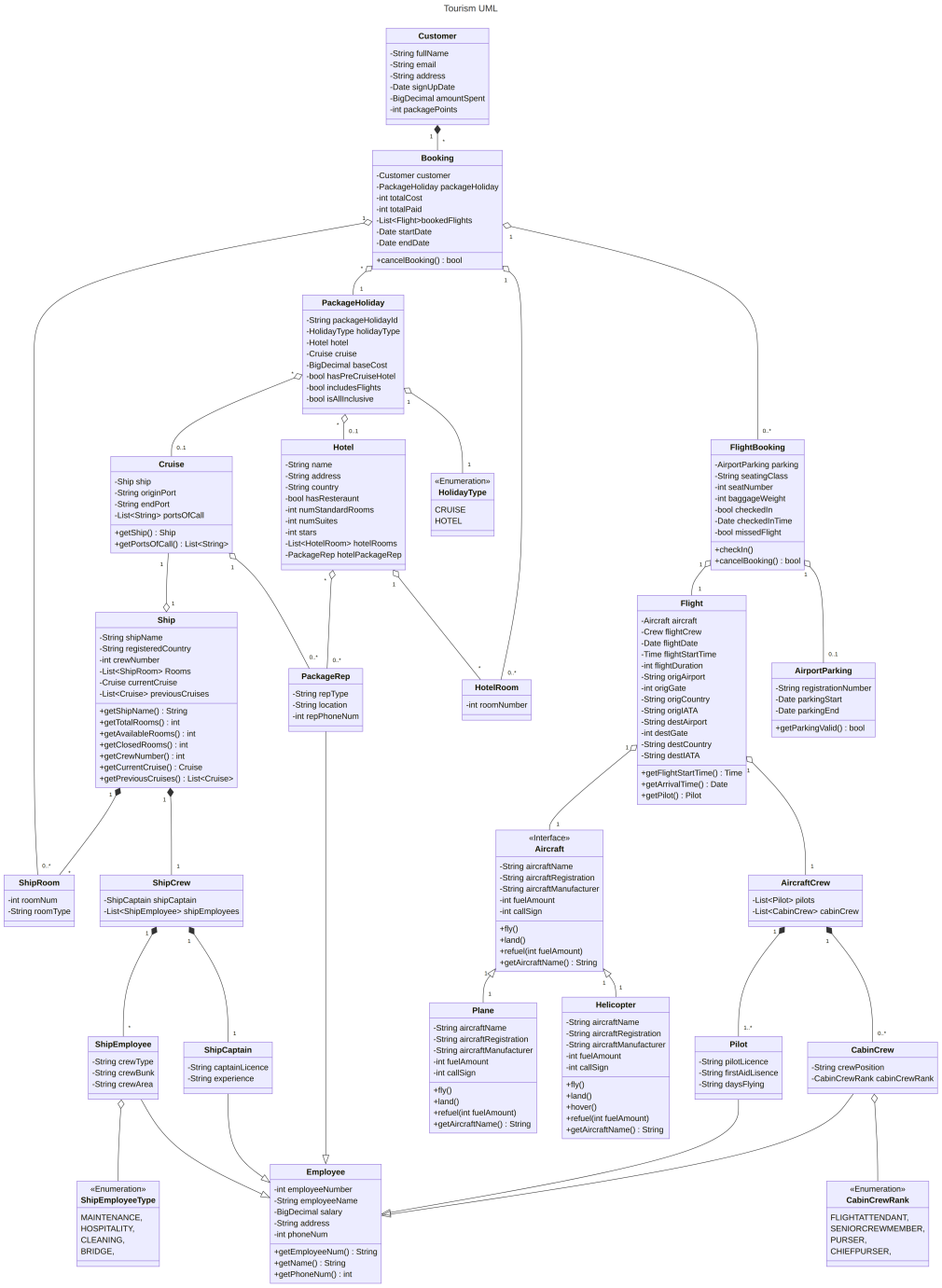

Figure 1: UML Representation of Program

Testing Report

Introduction

Automated Testing

The project will make extensive use of automated testing throughout development and deployment. Automated testing allows a developer to monitor if any defects are present in their codebase without manually testing and searching for errors, saving time. A report by the National Institute of Standards & Technology (2002) reports software bugs are estimated to have cost the United States of America $59.5 billion annually. They estimated that approximately $22.5 billion, a third of the cost, could be saved through improvements in testing infrastructure. The impact of automated testing is not only observed on the macro scale, but also on the team scale. Salesforce, customer relationship management software provider, reported the following reductions after implementing automated testing: staff involved in application deployment reduced by 65%, two to 3 hours of final testing became 10 minutes of automated tests, 3 to 4 hours of post-release testing was handled by 45 minutes automated testing, the patch release team reduced by 80%, and savings of 300 hours per major release (Cohn, 2009). On the other hand, automated testing has the following disadvantages: it has higher initial costs than manual testing, knowledge of the test tools is required, the tests require maintenance, and implementing testing requires either a developer or testing specialist (Umar and Chen, 2019). Despite these disadvantages, when automated testing is implemented early into a project's development its advantages far outweigh the detriments. Catching errors early without manual testing reduces the cost of finding and then solving them, in addition to improving the structure of the project, as seen in processes like test-driven development that will be discussed later. Overall, automated testing is an extremely valuable area with a significant role in the project's development process.

- Ensure that caught as they can become costly with time

The time invested by creating these automated tests can be recouped by the time saved in manual testing stages.

Testing pyramid

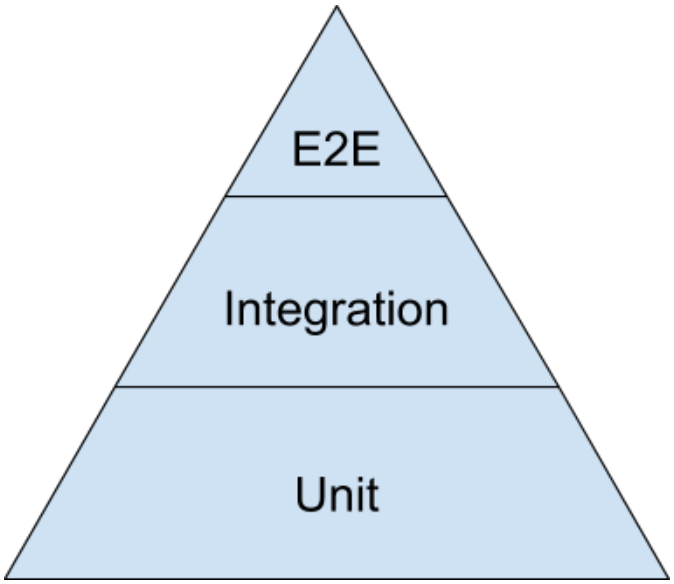

Figure 2: Testing Pyramid (Wacker, 2015)

The automated 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 development 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 such as databases, APIs and user input through the user interface.

The explanation on the testing pyramid provided short insights into the stages. The following report

will delve into the details and implementation of these layers.

Unit tests

The first stages of testing 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.

`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;

```

```
14
15      @Test
16      @DisplayName("Check-in process with an on-time flight booking")
17      public void checkIn() {
18          LocalDateTime earlyDateTime = LocalDateTime.now().plusYears(1);
19          /*
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  }
```

Integration tests

Integration, or service layer, testing involves verifying that independent components of the program successfully interact and connect together as expected (Contan et al., 2018). These independent components could be databases, files, APIs, user inputs, and so on. It is one level above the isolation

of unit tests, but below end-to-end testing. Integration testing is split into 2 areas, narrow and broad integration testing. The former tests: exercise the code that interacts with the separate component, use test doubles - not production or development services in the test, and number in higher numbers than the latter (Fowler, 2018). Whereas, broad testing requires live versions of every service, testing all code paths; not just those that interact with separate services. Therefore, narrow integration tests will run quicker and provide earlier feedback at the expense of the test's depth and code coverage. Thus, it is important to use both tests; a sizeable amount of narrow test regularly running will spot most integration errors. Then, the broad tests can be run during less frequently, such as during deployment pipelines, to spot any errors that passed through the narrow tests.

Narrow integration tests can be implemented through many libraries: `JUnit`, `Spring`, `testcontainers`, or mocking. Valley Cruises' service would require an SQL database for customer information. This external service would require integration testing with the Customer class and code that reads the customer information from the database. Using the library `testcontainers`, we create a temporary isolated SQL database through docker, we then connect to it, interact with the database through the customer's SQL classes, and check if those interactions propagate to the database. If the tests succeed, the database and program are successfully integrated.

Integration test using `testcontainer` and the `Customer` class (testcontainers, [no date]),

```java
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.testcontainers.containers.PostgreSQLContainer;


/**
 * Test the program's integration with a database
 */
public class CustomerIntegrationTest {

    /*
     Create a temporary database container to test a database's
     integration with the program
    */
    static PostgreSQLContainer<?> postgres =
            new PostgreSQLContainer<>("postgres:16-alpine");

    Customer customer;
    // CustomerDatabase customerDatabase;

    /**
     * Start the database
     */
    @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  }
```

In a broad integration test, the process of creating a customer and booking the customer on a holiday and that integration between these two concepts could be tested. Alternatively, APIs that are external to Valley Cruise's network such as potential flight tracking APIs would be included in broad tests due to a lack of control over them.

End to end testing

Input validation etc.

Test driven development

Implementation with continuous integration services

Conclusion

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 fowler 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. # Manual Testing

References

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

Contan, A., Dehelean, C. and Miclea, L. (2018) 'Test automation pyramid from theory to practice'. in *2018 IEEE international conference on automation, quality and testing, robotics (AQTR)*., pp. 1–5. doi: 10.1109/AQTR.2018.8402699.

Fowler, M. (2018) *Integration test*. Available at: https://martinfowler.com/bliki/IntegrationTest.html.

Greene, S. and Fry, C. (2007) 'Large scale agile transformation in an on-demand world'. in *Agile 2007*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 136–142. Available at: https://doi.ieeecomputersociety.org/10.1109/AGILE.2007.38.

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.

National Institute of Standards & Technology (2002) *The economic impacts of inadequate infrastructure for software testing*.

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

Umar, M.A. and Chen, Z. (2019) 'A study of automated software testing: Automation tools and frameworks'. 8, pp. 217–225. doi: 10.5281,zenodo.3924795.

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.