# Contents

# Program UML

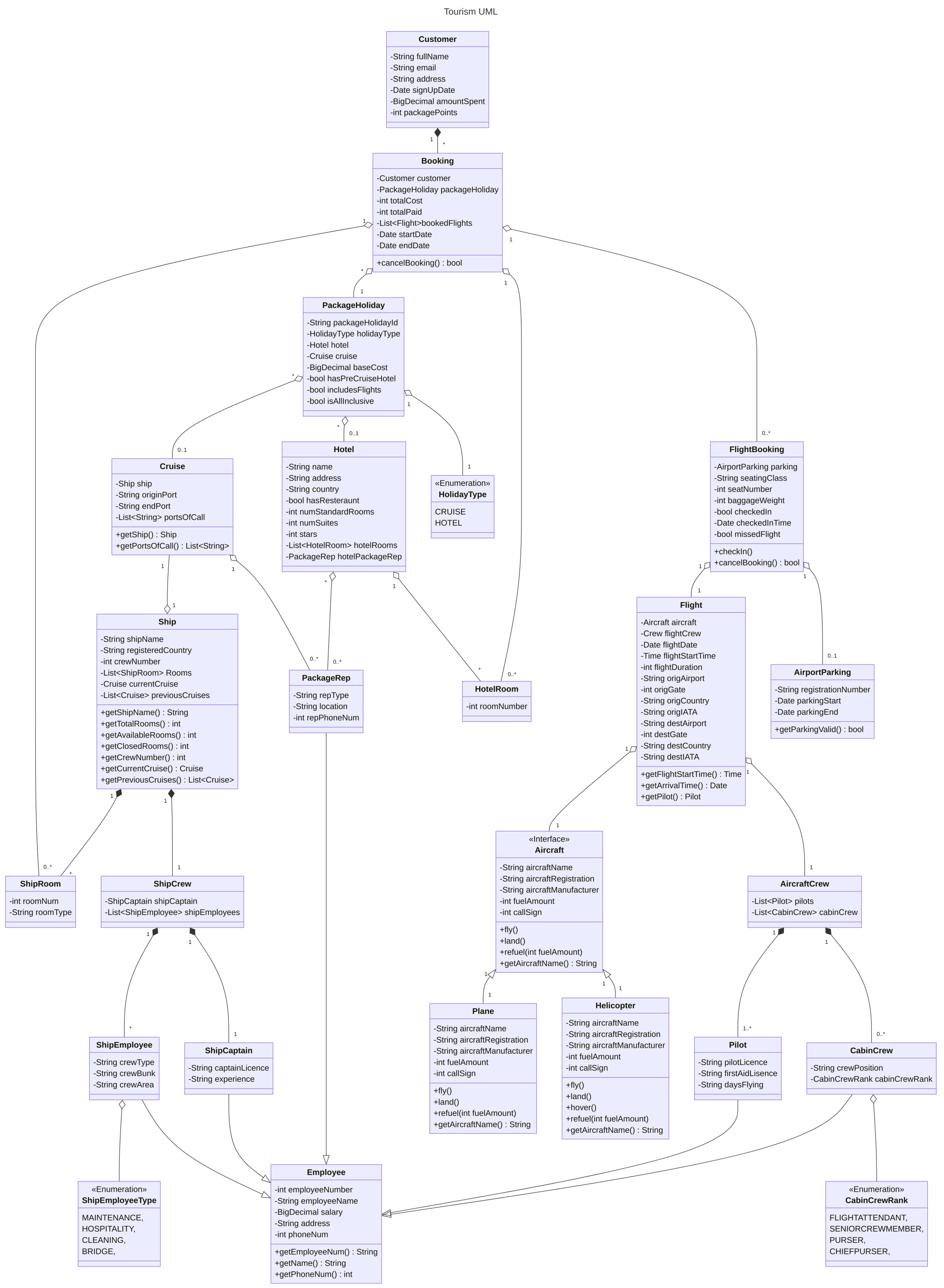


**Figure 1:** UML Representation of Program

# 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 maintenace, 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 advatages far outweight the detriments. Catching errors early without manual testing reduces the cost of finding and then solving them, in addition to improving the strucutre 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 signifigant 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**

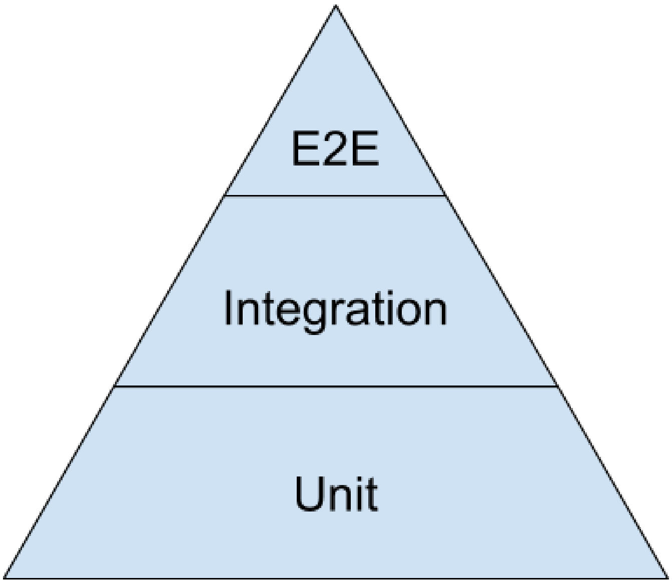


**Figure 2:** Testing Pyramid (Wacker, 2015)

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.

### Integration tests

#### Narrow

#### Wide

### 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 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);
        /*
```

```java
         Create a mocked class of testFlight testFlight could use
         an API for plane tracking, so creating a stub ensures that
         the test is independent
        */
        testFlight = Mockito.mock(Flight.class);
        // Set what to return when getFlightStart() is called
        Mockito.when(testFlight.getFlightStart()).thenReturn(
                earlyDateTime
        );

        // Call the check-in process
        testFlightBooking = new FlightBooking(
                testFlight, "economy", 12, 200
        );
        assertTrue(testFlightBooking.checkIn());
    }

    @Test
    @DisplayName("Check-in process with a late flight booking")
    public void checkInLate() {
        LocalDateTime lateDateTime = LocalDateTime.now().minusYears(1);

        // Create another mocked class
        testFlight = Mockito.mock(Flight.class);
        // Set return time to the future to fail check-in
        Mockito.when(testFlight.getFlightStart()).thenReturn(
                lateDateTime
        );

        // Call the check-in process
        testFlightBooking = new FlightBooking(
                testFlight, "economy", 12, 200
        );
        assertFalse(testFlightBooking.checkIn());
    }
}
```

Narrow integration test with a PostgreSQL database.

```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 {
```

```java

    /*
     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
    public static void startContainer() {
        postgres.start();
    }

    /**
     * Stop the database container
     */
    @AfterAll
    public static void stopContainer() {
        postgres.stop();
    }

    /**
     * Connect to the database and proceed to clean it of previous
     * entries
     */
    @BeforeEach
    public void setUpCleanDb() {
        customerDatabase = new CustomerDatabase(postgres.getJbdcUrl());
        customerDatabase.deleteAll();
    }

    /**
     * Test that {@code customerDatabase} integrates with and adds
     * the customer to the database
     */
    @Test
    @DisplayName("Database create customer")
    public void dbCreateCustomer() {
        customer = new Customer("Jake Real", "jakeemail@email.com",
                "3 Cool Road, Swandiff"
        );
        // Add the created customer to the database
        customerDatabase.addCustomer(customer)
        // Test that customer was created in the database
        assertEquals("Jake Real", customerDatabase.getCustomerByName(
```

```
                    "Jake Real").getFullName()
            );
        }

        /**
         * Test that {@code customerDatabase} integrates with and deletes
         * the customer from the database
         */
        @Test
        @DisplayName("Database delete customer")
        public void dbDeleteCustomer() {
            customer = new Customer("Jake Real", "jakeemail@email.com",
                    "3 Cool Road, Swandiff"
            );
            // Add the created customer to the database
            customerDatabase.addCustomer(customer);
            // Test that customer was created in the database
            assertEquals("Jake Real", customerDatabase.getCustomerByName(
                    "Jake Real").getFullName()
            );
            // Delete the customer from the database
            customerDatabase.deleteCustomerById(customer.getId);
            // Check that there are no customers
            assertEquals(0, customerDatabase.getAllCustomers().size());
        }

}
```

## Manual Testing

## References

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

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][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.

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.