IS1S481 Coursework 2

Jake Real - 23056792

10/03/2024

Contents

Program UML

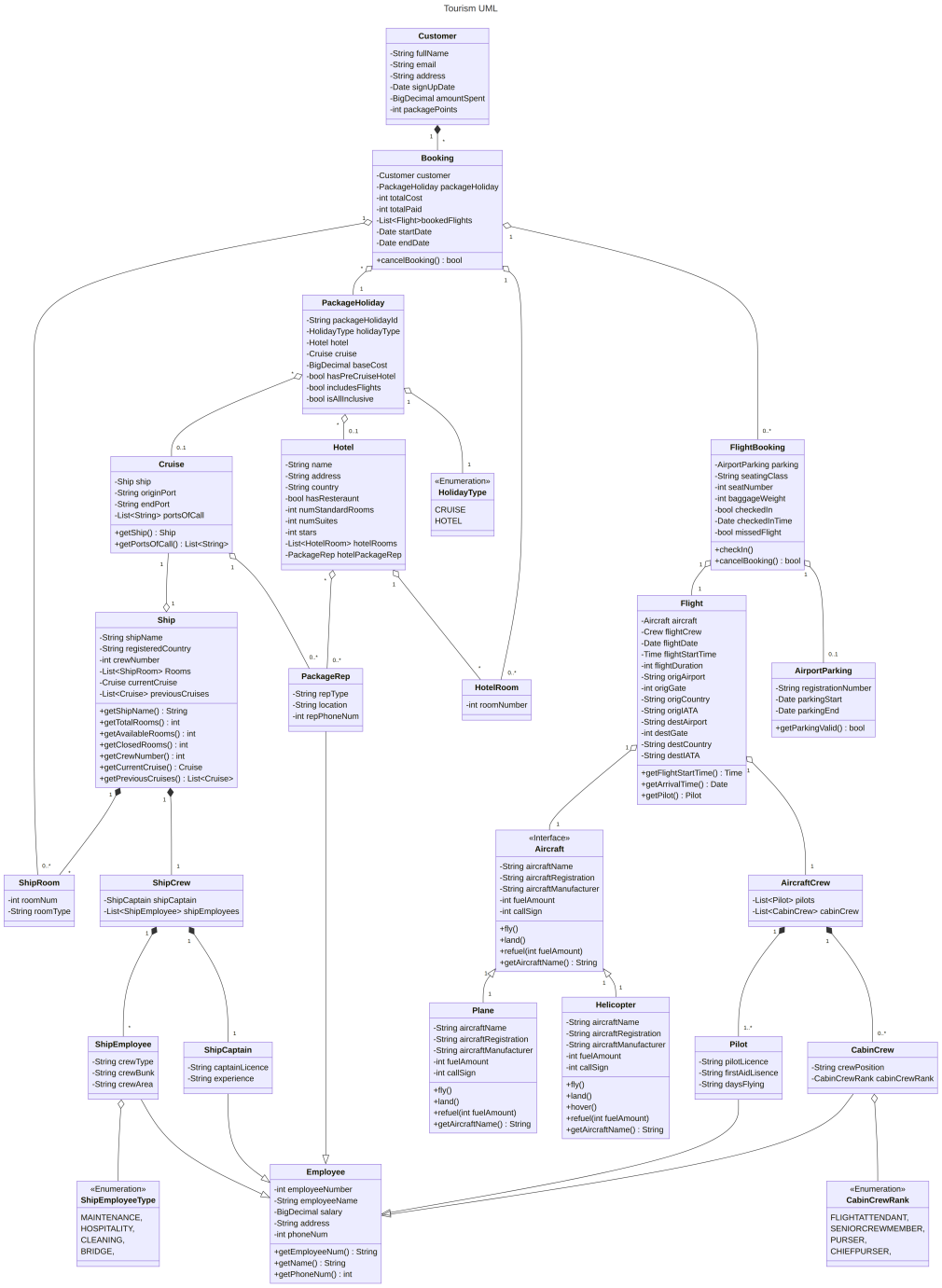

Figure 1: UML Representation of Program

Testing Report

Introduction

Testing is the process examining software's behaviour to check that it meets the required specifications of the service without errors arising. The process provides information concerning the software under the conditions that the test is run under; however, it only guarantees that the software will not fail under those specific conditions. Thus, it is imperative that the testing strategy for the company is comprehensive and covers most facets of the service to avoid undetected faults. Achieving large coverage manually would require considerable amounts of people hours within the company. Therefore, to tackle this problem, this report outlines a testing strategy making use of both manual and automated, computer-operated test strategies to achieve high coverage testing without major slowdowns in releases, or considerable expansions to testers and the quality assurance team.

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 to 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. Overall, automated testing is an extremely valuable area with a significant role in the project's development process.

Testing pyramid

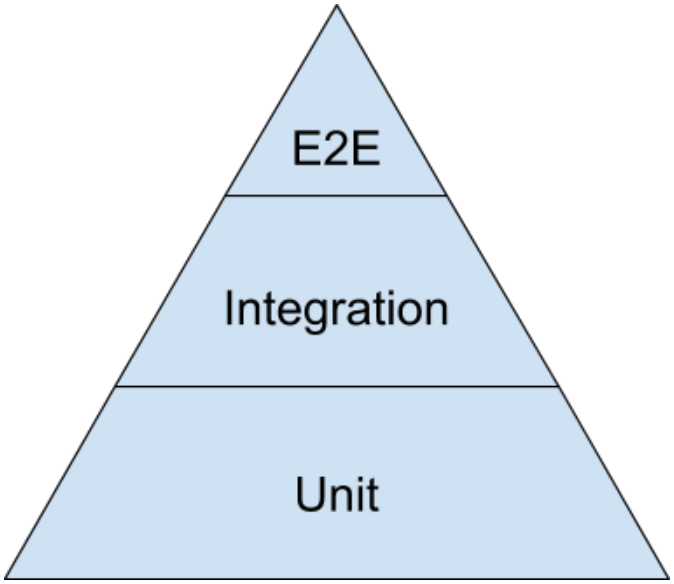

Figure 2: Testing Pyramid (Wacker, 2015)

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, lacking coverage on 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 will will

explain each layer in further depth and provide example implementations.

Unit tests

The first stages of testing occur during the development and programming process. The company uses unit tests as its foundation; these tests involve writing isolated automated tests targeting small sections of the program, known as units. A single project can have many unit tests; a common amount is one unit test per production class, however even more can be added. Creation of the unit test involves developing a criteria, referred to as the test case. The criteria should not be too strict, otherwise the test would break every time code is changed, but it should not be too general either. A good way to implement unit tests is through check observable behaviours; Martin Fowler (2018b), states, 'think about, if I enter values x and y, will the result be z? instead of, if I enter x and y, will the method call class A first, then call class B and then return the result of class A plus the result of class B?'. Testing lots of small units that integrate to form a complex program reduces the amount of uncertain variables compared to a large monolithic end-to-end tests. Unit tests are used in most of the companies production classes to ensure good coverage of the entire codebase. Within the company, the Java unit testing framework `JUnit` is 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, 2018a). 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. In Valley Cruises application, narrow integration tests will be used with the service's database for customer, holiday, flight, and cruise information. Furthermore, narrow integration tests can be used for any file input or output in Valley cruises operations; these input and output operations may be used to read images, Welsh language localisation text, or configuration files for the service.

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;

import static org.junit.jupiter.api.Assertions.assertEquals;


/**
 * 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");
```

```
22
23      Customer customer;
24      // CustomerDatabase customerDatabase;
25
26      /**
27       * Start the database
28       */
29      @BeforeAll
30      public static void startContainer() {
31          postgres.start();
32      }
33
34      /**
35       * Stop the database container
36       */
37      @AfterAll
38      public static void stopContainer() {
39          postgres.stop();
40      }
41
42      /**
43       * Connect to the database and proceed to clean it of previous
44       * entries
45       */
46      @BeforeEach
47      public void setUpCleanDb() {
48          customerDatabase = new CustomerDatabase(postgres.getJbdcUrl());
49          customerDatabase.deleteAll();
50      }
51
52      /**
53       * Test that {@code customerDatabase} integrates with and adds
54       * the customer to the database
55       */
56      @Test
57      @DisplayName("Database create customer")
58      public void dbCreateCustomer() {
59          customer = new Customer("Jake Real", "jakeemail@email.com",
60              "3 Cool Road, Swandiff"
61          );
62          // Add the created customer to the database
63          customerDatabase.addCustomer(customer)
64          // Test that customer was created in the database
65          assertEquals("Jake Real", customerDatabase.getCustomerByName(
66              "Jake Real").getFullName()
67          );
68      }
69
70      /**
71       * Test that {@code customerDatabase} integrates with and deletes
72       * the customer from the database
```

```
73          */
74         @Test
75         @DisplayName("Database delete customer")
76         public void dbDeleteCustomer() {
77             customer = new Customer("Jake Real", "jakeemail@email.com",
78                 "3 Cool Road, Swandiff"
79             );
80             // Add the created customer to the database
81             customerDatabase.addCustomer(customer);
82             // Test that customer was created in the database
83             assertEquals("Jake Real", customerDatabase.getCustomerByName(
84                 "Jake Real").getFullName()
85             );
86             // Delete the customer from the database
87             customerDatabase.deleteCustomerById(customer.getId);
88             // Check that there are no customers
89             assertEquals(0, customerDatabase.getAllCustomers().size());
90         }
91
92     }
```

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.

Contract Testing

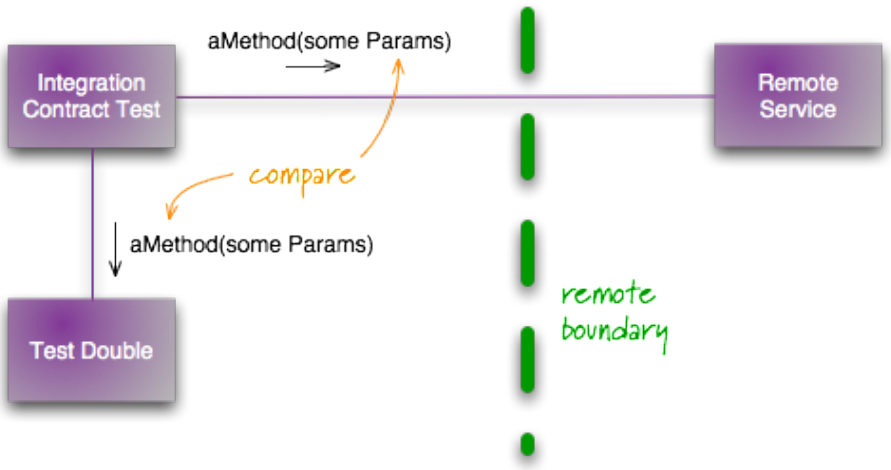

Figure 3: Contract testing (Fowler, 2011)

Integration testing is very effective when the tester controls the external services. However, speed and reliability decreases when tests occur between internal external, remote services. The external service can be slow, update with breaking changes, or have downtime. These are very undesirable in a test. To solve this issue, a test double is created through stubbing the remote service; this is when the class is emulated by the test software, returning predefined information when called. However, it is important that the test double is accurate to the remote service otherwise the developer will not be aware of when changes need to made to code to reflect the service's changes. Therefore, contract test's are used. Martin Fowler (2011) describes them as tests that, 'check all the calls against your test doubles return the same results as a call to the external service world'. Failures in the contact tests should alert a developer to look at the codebase and the external service to check they still integrate well. The test double should then be updated accordingly. Within the Valley Cruises service, potential uses of the contract test would be interacting with plane tracking APIs, postcode APIs for customer registration, and interacting with civil aviation authorities for pilot licencing.

End-to-end Testing

End-to-end tests cover the entire application, including the user interface. The user interface of the application is automatically run through by the testing application. Inputs should correctly trigger their actions, the correct data should be displayed, and the UI states should change when expected to. (Fowler, 2013) Furthermore, the layout of the frontend can be visually tested. As the testing tool proceeds through the service, screenshots of layouts are taken; these screenshots are compared to previous iterations and control screenshots. Any discrepancies are flagged to the developers for manual checking (Vaidya, 2023). End-end-testing's major advantage is that it exercises the whole application and its connections connected; errors in components that other tests were unable to find are discovered. However, end-to-end tests are far more brittle than other testing categories. Browser peculiarities, timings, model dialogues, and animations can create false positives that have to be debugged. Thus, these tests take more time and specialists to develop. Moreover, they must be regularly maintained when changes are made to user interface and data. These tests are far more expensive computationally, taking longer to complete. Therefore, far fewer of these tests are created and run, in accordance with the testing pyramid, so high-value scenarios are prioritised. In Valley Cruises' case, browsing the holidays, creating an account, adding the holiday to the cart, paying for the holiday, and managing holidays in your account would make good areas to use end-to-end testing on.

End-to-end testing for this project will be implemented by a framework supported in Java - `Selenium`. `Selenium` uses the browser to automatically call a website, interact and enter data into it, and check that the correct changes are made. Whilst doing this, `Selenium` can take screenshots of the process, run in a headless, non-graphical browser, or run on a server with no graphical user interface.

Example of using `Selenium` to test logging in to Valley Cruises,

```java
import io.github.bonigarcia.wdm.WebDriverManager;
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.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class LoginTest {
    private WebDriver driver;
    private String username = "testing";
    private String password = "testing";

    @BeforeAll
    public static void setUpDriver() throws Exception {
        WebDriverManager.firefoxdriver.setup();
    }

    @BeforeEach
    public void setUp() {
        driver = new FirefoxDriver();
    }

    @AfterEach
    public void tearDown() {
        driver.quit();
    }

    @Test
    public void loginWebsite() {
        driver.get("https://www.valleycruises.com/login");

        WebElement usernameField = driver.findElement(
            By.id("login-username")
        );
        WebElement passwordField = driver.findElement(
            By.id("login-password")
        );
        WebElement loginButton = driver.findElement(
            By.id("login-button")
        );

        usernameField.sendKeys(username);
        passwordField.sendKeys(password);
        loginButton.click();
```

```
50
51          WebElement usernameLabel = driver.findElement(
52              By.id("label-username")
53          );
54          assertEquals(usernameField.getText(), username);
55      }
56  }
```

Manual Testing

Manual testing is when testers go through the program using its features, checking for correct behaviour. The testers follow a test plan that describes the scope, methodology, test schedule, what resources are used and tools that are allocated for the test; This plan forms the general structure of the test. The tester then follows multiple more detailed test cases. These outline the: steps taken by the tester, the expected outcomes of said tests, and after testing, the actual outcomes of the test, and whether it passed or failed. Manual testing is most suitable for testing areas where automation and computers fail; these are accessibility testing, computers are unable emulate how a visually impaired people will screen readers to access a service; layout testing, computers struggle to find visual defects in programs so manually testing on different layouts and screens is preferred; and exploratory testing, that emphasises tester freedom resulting in tests that cover areas that automation might miss.

Static Testing

Finally, static testing is a subset of manual testing in which the program is not executed.

Table 1: Tests for password input in frontend,

Test number	Test data	Reason	Expected outcome	Actual outcome
1	Password = testing123@@	Test that a password meeting the requirements of containing more 7 characters or more, 2 or more numbers and symbols is accepted	Valid password	Valid password
2	Password = testing123	Test that a password not meeting the requirements of containing more 7 characters or more, 2 or more numbers and symbols is not accepted	Invalid password	Invalid password

Test number	Test data	Reason	Expected outcome	Actual outcome
3	Password = testing@@#	Test that a password not meeting the requirements of containing more 7 characters or more, 2 or more numbers and symbols is not accepted	Invalid password	Invalid password
4	Password = testing	Test that a password not meeting the requirements of containing more 7 characters or more, 2 or more numbers and symbols is not accepted	Invalid password	Invalid password

Table 2: Tests for searching by price in frontend,

Test number	Test data	Reason	Expected outcome	Actual outcome
1	Search = £2000	Test that a valid search amount correctly returns a search query	Valid search	Valid search
2	Search = -£2000	Test that a negative money search amounts does not return a search query as negative money is not possible	Invalid search	Invalid search

Dynamic Testing

Static testing

Conclusion

In conclusion, the company uses a mixed testing strategy of automated and manual tests with a bias to automation. The structure of the automated tests follows the testing pyramid, with many unit tests, a medium number of integration and contract tests, and finally a fewer amount of end-to-end tests. This strategy results in a bottom-up approach to automated testing where most errors should be caught before the final end-to-end testing, ensuring that they are caught quickly before causing costing considerable resources to fix. The manual testing process covers areas that are difficult or too in depth to be automated, such as layout testing, accessibility testing, and mimicing a user's flow through the program. Overall, the testing strategy of the company attempts to ensure a quick throughput of

software, whilst keeping good testing coverage of the released program in an attempt to catch errors prior to public release.

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. (2011) *Contract test*. Available at: https://martinfowler.com/bliki/ContactTest.html.

Fowler, M. (2013) *Broad stack test*. Available at: https://martinfowler.com/bliki/BroadStackTest.html.

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

Fowler, M. (2018b) *The practical test pyramid*. Available at: https://martinfowler.com/articles/practical-test-pyramid.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.

Vaidya, N. (2023) *How ot run visual tests with selenium: tutorial*. Available at: https://www.browsersta

ck.com/guide/run-visual-tests-with-selenium.

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.