IS1S481 – Coursework 2

Student ID: 23056793

Part A: Class Diagram & OOP Outline Skeleton Framework

Part B: Testing Framework Report

Table of Contents

Part A – OOP Task

A1: Class Diagram

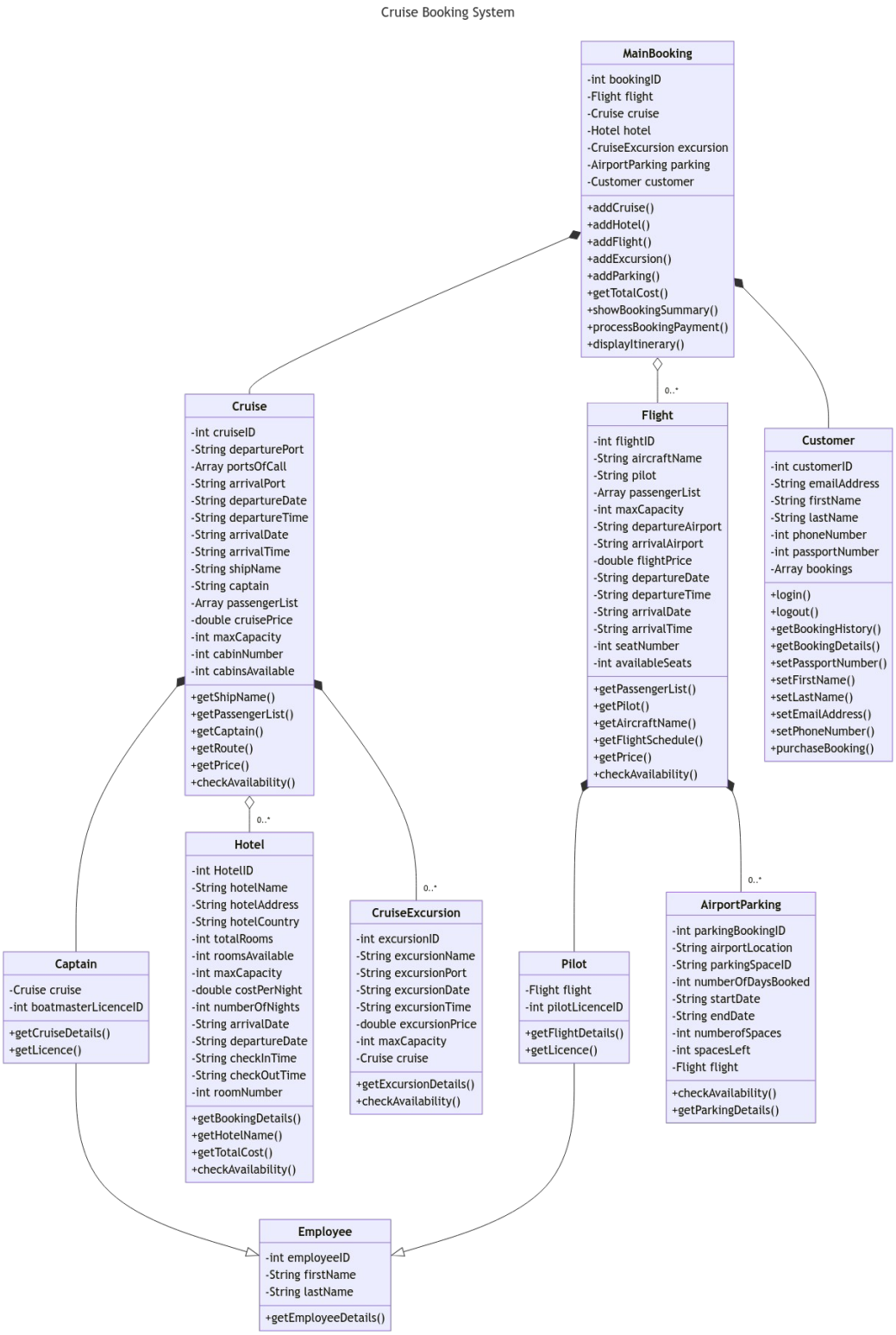

Figure 1: UML Class Diagram of Cruise Booking System

A2: OOP Outline Skeleton Framework

```java
package com.CruiseBooking;
public class MainBooking {
    private int bookingID;
    private Flight flight;
    private Cruise cruise;
    private Hotel hotel;
    private CruiseExcursion excursion;
    private Customer customer;
    private AirportParking airportParking;

    public MainBooking(int bookingID) {
        this.bookingID = bookingID;
    }

    public static void main(String[] args) {
        MainBooking booking = new MainBooking(123);
        System.out.println(booking);
    }

    // Add a cruise to the main booking
    public void addCruise() {

    }

    // Add a flight to the main booking
    public void addFlight() {

    }

    // Add a hotel to the main booking
    public void addHotel() {

    }

    // Add a cruise excursion to the main booking
    public void addExcursion() {

    }

    // Add airport parking to the main booking
    public void addParking() {

    }

    // Add up the prices of each part of the booking and display
    public void getTotalCost() {
```

```
    }

    // Display summary of purchased booking
    public void showBookingSummary() {

    }

    // Handle payment process
    public void processBookingPayment() {

    }

    // Collate dates and times of each section of the booking and display to user
    public void displayItinerary() {

    }
}
```

```
package com.CruiseBooking;
public class Customer {
    private int customerID;
    private String emailAddress;
    private String firstName;
    private String lastName;
    private String phoneNumber;
    private int passportNumber;
    private String[] bookings;

    public Customer(int customerID, String emailAddress, String firstName, String lastName, String
phoneNumber, int passportNumber, String[] bookings) {
        this.customerID = customerID;
        this.emailAddress = emailAddress;
        this.firstName = firstName;
        this.lastName = lastName;
        this.phoneNumber = phoneNumber;
        this.passportNumber = passportNumber;
        this.bookings = bookings;
    }

    public static void main(String[] args) {
        Customer customer = new Customer(123, "joe.bloggs@mail.com", "Joe", "Bloggs", "01234567890",
123456789, null);
        System.out.println(customer);
    }

    // Getters

    // if user is logged out, check if username and password match stored credentials
```

```
    // if true, login user. If false, display error message
    public void login() {

    }

    // if user is logged in, log user out
    public void logout() {

    }

    // check bookings associated with customer's ID and display in a list
    public void getBookingHistory() {

    }

    // Setters

    // Sets user's passport number to user input
    public void setPassportNumber() {

    }

    // Sets user's first name to user input
    public void setFirstName() {

    }

    // Sets user's surname to user input
    public void setLastName() {

    }

    // Sets user's email to user input
    public void setEmailAddress() {

    }

    // Sets user's phone number to user input
    public void setPhoneNumber() {

    }

    // Send card details, name, address etc. to main booking to process payment
    public void purchaseBooking() {

    }

}
```

```
package com.CruiseBooking;
public class AirportParking {
    private int parkingBookingID;
    private String airportLocation;
    private String parkingSpaceID;
    private int numberOfDaysBooked;
    private String startDate;
    private String endDate;
    private int numberOfSpaces;
    private int spacesLeft;
    private Flight flight;

    public AirportParking(int parkingBookingID, String airportLocation, String parkingSpaceID, int
numberOfDaysBooked, String startDate, String endDate, int numberOfSpaces,  int spacesLeft) {
        this.parkingBookingID = parkingBookingID;
        this.airportLocation = airportLocation;
        this.parkingSpaceID = parkingSpaceID;
        this.numberOfDaysBooked = numberOfDaysBooked;
        this.startDate = startDate;
        this.endDate = endDate;
        this.numberOfSpaces = numberOfSpaces;
        this.spacesLeft = spacesLeft;
    }

    public static void main(String[] args) {
        AirportParking airportParking = new AirportParking(123, "EGFF", "A135", 14, "12/07/2024",
"26/07/2024", 600, 57);
        System.out.println(airportParking);
    }

    // If spacesLeft = 0 return "no spaces remaining"
    public void checkAvailability() {

    }

    // Display parking booking details to customer
    public void getParkingDetails() {

    }
}
```

```
package com.CruiseBooking;
public class Flight {
    private int flightID;
    private String aircraftName;
    private String pilot;
    private String[] passengerList;
```

```
    private int maxCapacity;
    private String departureAirport;
    private String arrivalAirport;
    private double flightPrice;
    private String departureDate;
    private String departureTime;
    private String arrivalDate;
    private String arrivalTime;
    private int seatNumber;
    private int availableSeats;

    public Flight(int flightID, String aircraftName, String pilot, String[] passengerList, int maxCapacity,
String departureAirport, String arrivalAirport, double flightPrice, String departureDate, String
departureTime, String arrivalDate, String arrivalTime, int seatNumber, int availableSeats) {
        this.flightID = flightID;
        this.aircraftName = aircraftName;
        this.pilot = pilot;
        this.passengerList = passengerList;
        this.maxCapacity = maxCapacity;
        this.departureAirport = departureAirport;
        this.arrivalAirport = arrivalAirport;
        this.flightPrice = flightPrice;
        this.departureDate = departureDate;
        this.departureTime = departureTime;
        this.arrivalDate = arrivalDate;
        this.arrivalTime = arrivalTime;
        this.seatNumber = seatNumber;
        this.availableSeats = availableSeats;
    }

    public static void main(String[] args) {
        Flight flight = new Flight(123, "Boeing 747", "Joe Bloggs", null, 150, "CWL", "BGI", 500,
"14/07/2024", "05:00", "26/07/2024", "12:00", 14, 24);
        System.out.println(flight);
    }

    // Returns list of passengers to backend user
    public void getPassengerList() {

    }

    // Returns pilot name to backend user
    public void getPilot() {

    }

    // Returns aircraft name to backend user
    public void getAircraftName() {
```

```
    }

    // Displays price to customer
    public void getPrice() {

    }

    // If availableSeats = 0 return "No seats remaining"
    public void checkAvailability() {

    }
}
```

```
package com.CruiseBooking;
public class Hotel {
    private int hotelID;
    private String hotelName;
    private String hotelAddress;
    private String hotelCountry;
    private int totalRooms;
    private int roomsAvailable;
    private int maxCapacity;
    private double costPerNight;
    private int numberOfNights;
    private String arrivalDate;
    private String departureDate;
    private String checkInTime;
    private String checkOutTime;
    private int roomNumber;

    public Hotel(int hotelID, String hotelName, String hotelAddress, String hotelCountry, int totalRooms,
int roomsAvailable, int maxCapacity, double costPerNight, int numberOfNights, String arrivalDate,
String departureDate, String checkInTime, String checkOutTime, int roomNumber) {
        this.hotelID = hotelID;
        this.hotelName = hotelName;
        this.hotelAddress = hotelAddress;
        this.hotelCountry = hotelCountry;
        this.totalRooms = totalRooms;
        this.roomsAvailable = roomsAvailable;
        this.maxCapacity = maxCapacity;
        this.costPerNight = costPerNight;
        this.numberOfNights = numberOfNights;
        this.arrivalDate = arrivalDate;
        this.departureDate = departureDate;
        this.checkInTime = checkInTime;
        this.checkOutTime = checkOutTime;
        this.roomNumber = roomNumber;
    }
```

```
    public static void main(String[] args) {
        Hotel preCruiseHotel = new Hotel(123, "Fawlty Towers", "Torquay", "UK", 10, 3, 20, 50, 1,
"14/07/2024", "15/07/2024", "14:00", "10:00", 2);
        System.out.println(preCruiseHotel);
    }

    // Display hotel details to customer
    public void getBookingDetails() {

    }

    // Display name of hotel to user
    public void getHotelName() {

    }

    // Calculate total cost of hotel stay based on hotel and number of nights
    public void getTotalCost() {

    }

    // Return "No rooms available" if roomsAvailable is 0
    public void checkAvailability() {

    }
}
```

```
package com.CruiseBooking;
public class Cruise {
    private int cruiseID;
    private String departurePort;
    private String[] portsOfCall;
    private String arrivalPort;
    private String departureDate;
    private String departureTime;
    private String arrivalDate;
    private String arrivalTime;
    private String shipName;
    private String captain;
    private String[] passengerList;
    private double cruisePrice;
    private int maxCapacity;
    private int cabinNumber;
    private int cabinsAvailable;

    public Cruise(int cruiseID, String departurePort, String[] portsOfCall, String arrivalPort, String
departureDate, String departureTime, String arrivalDate, String arrivalTime, String shipName, String
```

```
captain, String[] passengerList, double cruisePrice, int maxCapacity, int cabinNumber, int
cabinsAvailable) {
        this.cruiseID = cruiseID;
        this.departurePort = departurePort;
        this.portsOfCall = portsOfCall;
        this.arrivalPort = arrivalPort;
        this.departureDate = departureDate;
        this.departureTime = departureTime;
        this.arrivalDate = arrivalDate;
        this.arrivalTime = arrivalTime;
        this.shipName = shipName;
        this.captain = captain;
        this.passengerList = passengerList;
        this.cruisePrice = cruisePrice;
        this.maxCapacity = maxCapacity;
        this.cabinNumber = cabinNumber;
        this.cabinsAvailable = cabinsAvailable;
    }

    public static void main(String[] args) {
        Cruise cruise = new Cruise(123, "Bridgetown", null, "Port of Spain", "15/07/2024", "09:00",
"25/07/2024", "11:00", "Boaty McBoatFace", "Jack Sparrow", null, 1500, 750, 345, 100);
        System.out.println(cruise);
    }

    // Return name of ship to backend user
    public void getShipName() {

    }

    // Return list of passengers to backend user
    public void getPassengerList() {

    }

    // Return name of captain to backend user
    public void getCaptain() {

    }

    // Show route to customer
    public void getRoute() {

    }

    // Calculate and display price to customer
    public void getPrice() {

    }
```

```
    // If cabinsAvailable = 0, return "No availability"
    public void checkAvailability() {

    }
}
```

```
package com.CruiseBooking;
public class CruiseExcursion {
    private int excursionID;
    private String excursionName;
    private String excursionPort;
    private String excursionDate;
    private String excursionTime;
    private double excursionPrice;
    private int maxCapacity;
    private Cruise cruise;

    public CruiseExcursion(int excursionID, String excursionName, String excursionPort, String
excursionDate, String excursionTime, double excursionPrice, int maxCapacity) {
        this.excursionID = excursionID;
        this.excursionName = excursionName;
        this.excursionPort = excursionPort;
        this.excursionDate = excursionDate;
        this.excursionTime = excursionTime;
        this.excursionPrice = excursionPrice;
        this.maxCapacity = maxCapacity;
    }

    public static void main(String[] args) {
        CruiseExcursion excursion = new CruiseExcursion(123, "Excursion", "Bridgetown", "17/07/2024",
"09:00", 51.99, 25);
        System.out.println(excursion);
    }
}
```

```
package com.CruiseBooking;

public class Employee {
    private int employeeID;
    private String firstName;
    private String lastName;

    public Employee(int employeeID, String firstName, String lastName) {
        this.employeeID = employeeID;
        this.firstName = firstName;
        this.lastName = lastName;
```

```
    }

    public static void main(String[] args) {
        Employee employee = new Employee(123, "Joe", "Bloggs");
        System.out.println(employee);
    }

    // Returns all employee params to backend user
    public void getEmployeeDetails() {

    }
}
```

```
package com.CruiseBooking;

public class Pilot extends Employee {
    private Flight flight;
    private int pilotLicenceID;

    public Pilot(int employeeID, String firstName, String lastName, int pilotLicenceID) {
        super(employeeID, firstName, lastName);
        this.pilotLicenceID = pilotLicenceID;
    }

    public static void main(String[] args) {
        Pilot pilot = new Pilot(123, "Joe", "Bloggs", 123);
        System.out.println(pilot);
    }
}
```

```
package com.CruiseBooking;

public class Captain extends Employee {
    private Cruise cruise;
    private int boatmasterLicenceID;

    public Captain(int employeeID, String firstName, String lastName, int boatmasterLicenceID) {
        super(employeeID, firstName, lastName);
        this.boatmasterLicenceID = boatmasterLicenceID;
    }

    public static void main(String[] args) {
        Captain captain = new Captain(123, "Joe", "Bloggs", 123);
        System.out.println(captain);
    }
}
```

Part B – Report

Abstract

Our software development company has faced significant backlash due to the release of a flawed national software system, resulting in erroneous blame on businesses and customers, leading to legal repercussions. Valley Cruises, a prominent client, has raised concerns about our capabilities in light of these events. In response, this report delves into our testing strategies and policies, aiming to address the shortcomings and propose enhancements to mitigate similar issues in the future.

Introduction

Software testing is the act of examining the behaviour of software through varying methods of validation. This can include manual or automated testing, such as unit testing, integration testing or different types of input validation.

In the wake of the British Post Office Horizon scandal, testing for bugs in software has never been more at the forefront of the public consciousness. Over 900 sub-postmasters were wrongly convicted of theft, fraud or false accounting by the Post Office over 16 years due to major accounting bugs in its Horizon system. Following the successful overturning of the convictions, the UK Government has now allocated £1 billion in order to compensate wrongly convicted sub-postmasters, offering £600,000 per settlement (*The Guardian*, 2024).

Testing in this manner also saves on both time and cost to companies; a 2021 report from the Consortium for Information and Software Quality found that in 2020, the total losses by US companies due to poor software quality totalled approximately $2.08 trillion. In addition, a 2002 report from the National Institute of Standard Technology states that on average, it takes 15.3 hours to fix a bug found in post-production, whereas this time decreases to an average of 4.9 hours for the average bug found in early testing. It is clear that a robust testing strategy will drastically increase the quality of our codebase, lower costs, save a considerable amount of time, and most importantly, rebuild trust in our clients.

1. Our Testing Strategy

1.1 Input Validation

Input validation is where we test data given to us by a user against certain constraints. For example, we may wish for an uploaded image to conform to a .jpg format, or ensure a password contains at least 15 characters. We employ input validation in all our projects to ensure any data inputs are both correct and useful. Depending on the nature of the data, there are several types of input validation that we employ, such as data type validation or range validation. Further types of input validation exist and are categorised by the way we wish for the inputted data to be validated.

Below is an example of input validation: the method checks the user's username and password against what is stored on the server, and processes the request if they match and throws an error if they do not.

```
// Java Servlet Input Validation Example
protected void doPost(
    HttpServletRequest request,
    HttpServletResponse response
    )
    throws ServletException, IOException {
    String username = request.getParameter("username");
    String password = request.getParameter("password");

    // Validate username and password inputs
    if (isValidInput(username) && isValidInput(password)) {
        // Process the request
    } else {
        // Handle invalid input error
        response.sendRedirect("error.jsp");
    }
}

private boolean isValidInput(String input) {
    // Validate input against certain criteria (e.g., length, format, etc.)
    // Return true if input is valid, false otherwise
}
```

Figure 2: An example of input validation

1.1.1 Types of Input Validation

Data type validation is used on input fields requiring a specific data type in order for the entered data to be considered valid. The code will check that the input entered matches the requested data type. For example, if an input field is requesting a user to enter their age, it will only accept the numerical characters 0-9 as valid inputs.

Range validation is used where an input is required to be within a specific range in order for the data to be considered valid. For example, a user may need to select an age between 50 and 80 to be able to apply for a life insurance product.

1.2 Manual vs Automated Testing

Any robust testing strategy should contain both manual and automated testing; each have an important role in the testing process. Manual testing involves users manually exploring the software following a particular path and checking for any issues along the way, often simulating the way a user would navigate the software. A testing plan is put in place beforehand which a tester follows before indicating whether the expected behaviour has occurred or not, often in a table or using a tool such as Azure Test Plans. Manual testing tools give developers the ability to organise manual tests and list expected behaviours so that testers can reference these when carrying out their work.

User Acceptance Testing (UAT) is another form of manual testing where testers verify that the software has met the needs of the customer. Finally, exploratory testing allows testers to manually test without the use of testing guidelines, leaving them free to explore the software naturally and log any bugs or issues as they arise.

								Test Run	*Tester*	Test Run	*Tester*
								Date	*Version*	*Date*	*Version*
ID	Function	Priority	Regr.	Description	Inputs	Expected results	Notes, data files, etc	Result	Comments	Result	Comments
1	Opening files	1	r1	Opening documents of different sizes	Open documents of different sizes. Including empty document and document which size exeeds available memory. Open each file, add a new row (few words) at the end of the document and save the file.	The documents can be opened and edited successfully, if document is too big for the system to handle, a proper error message is shown. Verify the contents of the saved file.	Test data files: empty.txt, small.txt, big.txt, huge.txt	Pass	Error message could be more informative and e.g. prompt the user to shut down the jEdit if memory limit is exceeded. No data loss experinced, though.		
2	Opening files	1	r1								
3	Saving files	1	r2								
4	Autosave	2									
5											
6											
7											
8											
9											
10											
11											
12											
13											
14											
15											
16											
17											
18											
...											
n											

Figure 3: Example of a manual testing template

Manual testing is also advantageous when it comes to testing UI/UX and accessibility issues; as programs often fail to test these in an automated way, these areas are often outside the scope of automated testing tools and as such require a dedicated tester.

Automated testing has the advantage of being able to generate tests for a development team, generally saving significant amounts of time and money, however it often requires knowledge of automated testing software such as Selenium, PyTest, JUnit, Cypress or others, depending on the project stack. However, the time spent writing tests in the testing software can sometimes negate the time that would have been saved in manual testing. Using software that supports macros can assist with this, such as Selenium. A macro records a tester’s mouse and keyboard movements for the software’s expected behaviour and then plays these actions back automatically. If the software behaves as expected the test passes, otherwise it fails.

1.3 Automated testing in detail

A good automated testing strategy is often visualised in the form of a three-tier test automation pyramid. Originally designed by Mike Cohn in 2009, a test automation pyramid denotes a plan of the types of tests a development team would perform when forming their testing strategy. For example, at the bottom of the below pyramid is unit testing, in the middle is integration testing, and finally, at the top is end-to-end testing (E2E).

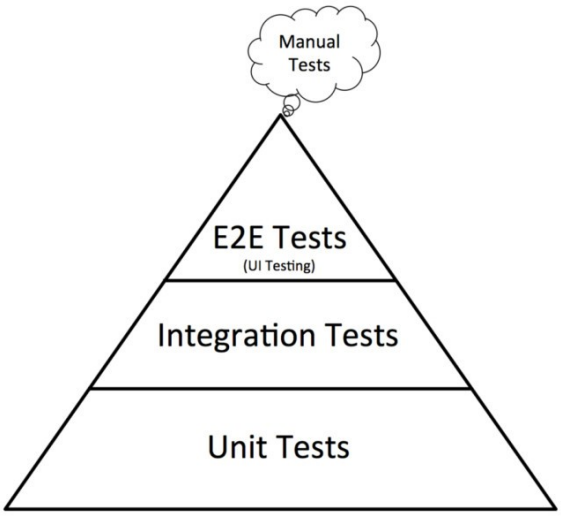

Figure 4: A typical example of a testing pyramid. Manual testing is not part of the pyramid but is included as part of the testing process.

The size of each of the units in the pyramid also correspond to the amount of resources that should be devoted to each type of test. That is, the most resources should be spent on unit tests and the least on E2E tests, with integration tests in the middle to bridge the gap between the small scope of unit tests and the large scope of E2E. This is due to the time and costs associated with excessive E2E testing which can be better spent on unit testing which will target specific parts of the codebase. This model is one of many examples of building an automated testing strategy; other models include inverting the pyramid, or including additional layers to suit the nature of a project, such as including beta testing for a video game.

1.3.1 Unit Testing

Unit testing is where individual components of a program are tested in isolation to ensure the components – known as units – work as expected. Given that unit testing is the main type of automated testing we focus on, it is expected that our unit tests cover as many units within the software as possible so as to ensure better testing coverage. To capture as many bugs as possible, it is significantly more effective to use multiple unit tests across the application rather than using one large E2E test. We generally aim for 80% test coverage using unit testing.

Below is an example of a Java unit test using JUnit to test the addition function on a calculator app:

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

class CalculatorTest {
    @Test
    void testAddition() {
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 3);
        assertEquals(5, result, "2 + 3 should equal 5");
    }
}
```

Drawing 1: Example unit test

1.3.2 Integration Testing

Integration testing tests how different sections of the code that makes up the whole software interact with each other and typically occurs before E2E testing and after unit testing. Units which have undergone unit testing are grouped together and tested to ensure these groups successfully interact with each other. For example, for Valley Cruises' application, we may create a group consisting of a payments API, a database and a user interface once all have been unit tested, and then use integration testing to ensure these can all successfully interact with one another. As integration tests are further up the pyramid, fewer resources should be allocated to these than unit tests. Further, while unit tests should ideally try to cover every path through the code, integration testing should not attempt this; as their scope is larger they will naturally take longer to run than unit tests.

Below is an example to test endpoints using Springboot and JUnit:

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;

@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
class IntegrationTest {
    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    void testHelloEndpoint() {
        String response = restTemplate.getForObject("/hello", String.class);
        assertEquals("Hello, World!", response);
    }
}
```

Drawing 2: Example of integration test

1.3.3 End-to-end Testing

E2E testing tests the entire software from beginning to end. The goal of E2E testing is to simulate the experience of a user going through the application from start to finish, ensuring there are no issues or bugs during this process. Automated testing software automatically runs through the entire software, testing inputs, UI and data displayed to the end user. At the end of the test, the software lists all discovered issues to the developer team. As E2E testing is found at the top of the pyramid, this type of testing should be where the least amount of resources are allocated. This is because E2E testing has the broadest scope and covers entire applications, and so would be costly if all the tests were this kind. It is also much easier for this kind of test to go wrong if a small amount of code in the software is edited after writing the test. To this end, E2E tests are not ideal when trying to capture more specific errors that would be better suited to the scope of a unit or integration test, but are much better in capturing a broader view of how the whole application works with all its moving parts. In the case of Valley Cruises, we would begin the E2E test at creating a customer account and end it with the customer successfully receiving the booking summary of their booked cruise.

The below example uses Selenium WebDriver and JUnit to test a user login:

```
import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.By;

class EndToEndTest {
    @Test
    void testLoginPage() {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        WebDriver driver = new ChromeDriver();

        driver.get("https://example.com/login");
        WebElement usernameInput = driver.findElement(By.id("username"));
        WebElement passwordInput = driver.findElement(By.id("password"));
        WebElement submitButton = driver.findElement(By.id("submit"));

        usernameInput.sendKeys("testuser");
        passwordInput.sendKeys("testpassword");
        submitButton.click();

        String pageTitle = driver.getTitle();
        assertEquals("Welcome Page", pageTitle);

        driver.quit();
    }
}
```

Drawing 3: Example E2E test

2. A Case for Test-Driven Development

While our testing strategy is strong, this report has found that our developers do not conform to a particular software development process. It is suggested that we undergo a transition to begin adopting a methodology of Test-Driven Development (TDD).

TDD requires developers to convert all software requirements into test cases before the development of software. The software is then tracked through the use of repeated testing using these test cases. This would ensure a significantly more robust approact to our development process than our current model of developing the software first then creating the tests later. The testing cycle looks like this (*Beck, 2002*):

1. Create a list of tests for the new feature. This means listing all the expected variants in desired new software behaviour and forces the developer to write these down before beginning to code, and to utilise user stories.

2. Add one test from the list. The developer writes one automated test that passes when the desired variant in the behaviour is observed.

3. Run all tests. The new test should fail for expected reasons. This is to rule out the possibility that the previous test will always pass and demonstrates that new code is required for the new feature.

4. Write the simplest code that passes the new test. No code should be added beyond the scope of the test; hard-coding is acceptable as long as the test passes.

5. All tests should now pass. If there are any tests that fail, amend the code until all tests pass. This is to make sure all code meets the requirements of the test.

6. Refactor as needed, using tests after each refactor to ensure that functionality is preserved. Remove hard-coded data at this point. The code should be made more readable and easier to maintain. It is essential to re-run the tests to ensure no functionality has broken.

7. Add the next test on the list. Repeat this process until all test have been included and pass.

Each test should be small and commits made often; this is to minimise the amount of excessive debugging by the developer when a simple undo can be done instead.

There are several benefits to TDD, not least including more comprehensive test coverage, better documented code and a reduction in constant debugging. Implementing TDD will significantly improve the quality of our software going forward and will restore our clients' confidence in our team.

Conclusion

In conclusion, this report found that while our testing strategy is comprehensive – consisting of both manual and automated testing – a lack of structure to our testing process resulted in serious errors in our code, undermining Valley Cruises' confidence in us as a competent development team. By implementing TDD as a structured methodology to our work, we aim to restore lost confidence in us and assure our clients that we can deliver quality software solutions.

References

Bansal, A. (2021). *Best Practices For Unit Testing In Java | Baeldung*. [online] www.baeldung.com. Available at: https://www.baeldung.com/java-unit-testing-best-practices [Accessed 15 Mar. 2024].

Beck, K. (2002). *Test-driven development by example*. Boston: Addison-Wesley.

Cohn, M. (2009). *The Forgotten Layer of the Test Automation Pyramid*. [online] Mountain Goat Software. Available at: https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid [Accessed 15 Mar. 2024].

Courea, E. (2024). Ministers to quash convictions of hundreds of post office operators. *The Guardian*. [online] 13 Mar. Available at: https://www.theguardian.com/uk-news/2024/mar/13/ministers-to-quash-convictions-post-office-operators-horizon-scandal [Accessed 15 Mar. 2024].

Hakoune, R. (2022). *Test case template for software feature troubleshooting*. [online] monday.com Blog. Available at: https://monday.com/blog/task-management/test-case-template/ [Accessed 15 Mar. 2024].

Krasner, H. (2021). *The Cost of Poor Software Quality in the US: A 2020 Report*. [online] *Consortium for Information & Software Quality*. Available at: https://www.it-cisq.org/cisq-files/pdf/CPSQ-2020-report.pdf [Accessed 15 Mar. 2024].

Obregon, A. (2023). *Effective Java Testing Strategies: Unit, Integration, and End-to-End Testing Techniques*. [online] Medium. Available at: https://medium.com/@AlexanderObregon/effective-java-testing-strategies-unit-integration-and-end-to-end-testing-techniques-5ade73ab259a [Accessed 15 Mar. 2024].

RTI (2002). *The Economic Impacts of Inadequate Infrastructure for Software Testing*. [online] *National Institute of Standards & Technology*. National Institute of Standards & Technology. Available at: https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf [Accessed 15 Mar. 2024].

Varkholyak, V. (2018). *What is Test Driven Development (TDD)? Software Development Blog*. [online] Lasoft. Available at: https://lasoft.org/blog/test-driven-development-tdd/ [Accessed 15 Mar. 2024].