Assessment 2: Report Automated Software Testing

10774743

GitHub: https://github.com/Plymouth-University/comp2005-assessment2-Esther-Skillman

YouTube Video: https://youtu.be/y6uNerOJWNo

Contents

[Analysis and Discussion (10%) 2](#_Toc166846234)

[Test cases 3](#_Toc166846235)

[Unit tests (20%) 4](#_Toc166846236)

[Integration tests (20%) 5](#_Toc166846237)

[System tests (10%) 5](#_Toc166846238)

[Metrics (such as Code Coverage) (10%) 6](#_Toc166846239)

[Usability Testing (10%) 6](#_Toc166846240)

[How to run tests 10](#_Toc166846241)

[Functional Test Plan 11](#_Toc166846242)

[Use of Tools, Practices, and Systems (such as Version Control, CI / CD) (10%) 13](#_Toc166846243)

[Evaluation (10%) 13](#_Toc166846244)

# Analysis and Discussion (10%)

The testing approach for the API blended manual testing, unit testing with mock objects, and integration testing to ensure robustness. Before starting my test development, manual tests were conducted on the Hospital Maternity Unit API to verify successful project loading. Then following a Test-Driven Development (TDD) approach, basic unit tests with mock objects were implemented to define endpoint requirements. These unit tests were continuously refined, incorporating tags to classify test types, adding assertions for precision, and encompassing various scenarios to enhance code coverage and pre-emptively identify bugs.

Additionally, I conducted manual testing of API endpoints through the browser before integration testing as a faster development option. Integration tests were subsequently introduced to validate endpoints, scrutinizing both success and failure responses. Recognizing the potential variability of real-world data in the API, emphasis was placed on testing status codes rather than specific response content. This comprehensive TDD approach facilitated early bug detection, ensured endpoint functionality, and instilled confidence in the system's behaviour across all testing phases.

The front-end desktop application on the other hand was tested with UAT to validate the basic functionality could be deciphered by a user and to gain feedback in early development to improve the application later in development.

Below shows a class diagram of the APIs and desktop application, and their respective classes and tests.

A diagram of a computer program

Description automatically generated with medium confidence

*Figure 0.1 – UML class diagram*

# Test cases

The approach to testing was done with both unit and integration testing, using a wide range of scenarios of edge, corner, and boundary cases for my API.

*Figure 0.2 – Unit and Integration Testing Usage for the API*

*Figure 0.3 – Types of test usages (edge, corner, and boundary)*

## Unit tests (20%)

As covered in my discussion, unit tests were initialised before developing the API as a Test-Driven Development approach. The advantages to this approach is covered in Fig.0.1 and an example of my implementation of this for F4 can be seen below:

A screenshot of a computer program

Description automatically generated

*Fig 0.4 - Created method with as little code as possible for the TDD.*

A computer screen shot of a program code

Description automatically generated

*Fig 0.5 - Wrote test for respective function.*

With the use of the Mockito library in java, writing unit tests with mock objects allowed me test a wide range of scenarios that would otherwise not be available with the unchanging existing API provided, such as testing for not data retrieved from the /Allocations or /Admissions endpoint.

Taking a closer look at Fig 0.3, during this stage of development I ran into the challenge of having a poor understanding of the API I was trying to mock. Instead of mocking the API I was unfamiliar with the mocking process and mocked the expected response instead. I overcame this challenge by thorough debugging to recognise this gap in my understanding and fix it in future development. Here is the updated code of the same unit test:

A screenshot of a computer program

Description automatically generated

This improved version correctly utilises the mock objects that are called outside of the test with *getAllocations()* for example whilst expecting a valid response.

## Integration tests (20%)

Integration tests were implemented after the MVP of my API with all unit tests working. With an already established framework for the unit tests, shifting these to integration tests provided very time effective and allowed me to test the actual API itself without the mock objects.

A computer screen with many colorful text

Description automatically generated with medium confidence

*Fig 0.6 –GetAdmissionsForInvalidPatient integration test.*

The test above highlights the use of multiple conditions to ensure significant code cover of my API. Robust methods are also demonstrated with the ‘port’ variable that utilises the interface @LocalServerPort as opposed to 8080 in the event the port changes.

## System tests (10%)

This type of black-box testing was mimicked heavily in the integration testing to demonstrate how the API would function in real-life circumstances. To cover these aspects using unit and integration testing covering multiple test cases. By testing various scenarios, including edge cases and unexpected inputs, system testing validated the robustness and reliability of my API across different use cases. This thorough examination using both integration and unit testing ensured that the API met the requirements of end-users, proving reliable in its performance and functionality within the intended production environment.

# Metrics (such as Code Coverage) (10%)

Overall:

A screen shot of a number

Description automatically generated

Integration tests:

A screen shot of a number

Description automatically generated

Unit tests:

A screen shot of a number

Description automatically generated

Looking at the highlighted class of my endpoints, the code coverage of all the tests are incredibly high at 87%, indicating the robustness of my code. Areas for improvement could have been for integration tests to cover more of the code at only 82% however provided more of a challenge due to the inability to use mock objects.

# Usability Testing (10%)

Extensive evidence of usability testing is present (including detailed description of the method used, the number of participants, tasks used, questions used, results and modifications to the software suggested and implemented).

Method used: Microsoft forms Online Survey

Link to survey: <https://forms.office.com/r/1AfJ1NReBK>

Number of participants: 4

Instructions given: To test the software in any way the user sees intuitive.

Questions used:

A screenshot of a computer survey

Description automatically generated

Results and modifications suggested (Qualitative data):

A screenshot of a medical application

Description automatically generatedPrevious implementation:

Blank Search box

Confusing headings

Editable table

Obscure patient ID

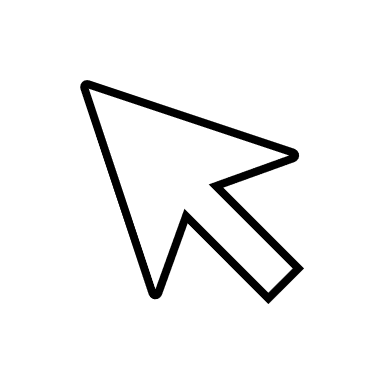
Evidence of Implementation:

A screenshot of a computer

Description automatically generated

Moved + Increased size of font

Explanations for headings

A screenshot of a medical form

Description automatically generated

Tooltip for input help

Non-editable table

# How to run tests

To run both the unit tests and integration tests within the API on IntelliJ:

1. Navigate to the project folder.
2. Select the class for the respective tests you want (unit/integration) or the folder named ‘tests’ to conduct all tests
3. Right-click and select Run.

# Functional Test Plan

Test cases that were passed with unit and integration testing.

**F1 - A list of all admissions for a specific patient:**

Positive Test Case:

* Description: Provide a patient ID and verify that the list of admissions is returned correct.
* Steps:
  + Input a valid patient ID.
  + Send a request to the API endpoint to retrieve admissions for the specified patient.
  + Verify that the response contains the expected admissions data for the patient.
* Expected Outcome: The API returns a list of admissions for the specified patient.
* Result: Pass

Negative Test Case:

* Description: Provide an invalid patient ID and verify that an error response is returned
* Steps:
  + Input an invalid or non-existent patient ID.
  + Send a request to the API endpoint to retrieve admissions for the specified patient.
  + Verify that the response contains an error message indicating that the patient ID is invalid.
* Expected Outcome: The API returns an error message indicating that the patient ID is invalid.
* Result: Pass

Boundary Test Case:

* Description: Test with the maximum and minimum values for the patient ID to ensure robustness.
* Steps:
  + Input the maximum possible patient ID value.
  + Send a request to the API endpoint to retrieve admissions for the specified patient.
  + Verify that the response contains the expected admissions data for the patient.
  + Repeat the above steps for the minimum possible patient ID value.
* Expected Outcome: The API returns a list of admissions for the specified patient, regardless of the patient ID value.
* Result: Pass

**F2 - A list of patients currently admitted:**

Positive Test Case:

* Description: Verify that the API returns a list of patients currently admitted.
* Steps:
  + Send a request to the API endpoint to retrieve the list of currently admitted patients.
  + Verify that the response contains data for at least one admitted patient.
* Expected Outcome: The API returns a list of currently admitted patients.

Negative Test Case:

* Description: Verify that the API responds with an error in the case when there are no patients currently admitted.
* Steps:
  + Send a request to the API endpoint to retrieve the list of currently admitted patients.
  + Verify that the response contains an empty list or an appropriate message indicating no admitted patients.
* Expected Outcome: The API returns an empty list or error indicating no admitted patients.
* Result: Pass

**F3 - Identify which member of staff has the most admissions:**

Positive Test Case:

* Description: Verify that the API correctly identifies the member of staff with the most admissions.
* Steps:
  + Send a request to the API endpoint to identify the member of staff with the most admissions.
  + Verify that the response contains data identifying the staff member with the most admissions.
* Expected Outcome: The API returns data identifying the staff member with the most admissions.
* Result: Pass

Negative Test Case:

* Description: Verify that the API handles the case when there are no admissions recorded.
* Steps:
  + Simulate a mock object where there are no admissions recorded in the system.
  + Send a request to the API endpoint to identify the member of staff with the most admissions.
  + Verify that the response contains an appropriate message indicating no admissions recorded.
* Expected Outcome: The API returns a message indicating no admissions recorded.
* Result: Pass

**F4 - A list of staff who have no admissions:**

Positive Test Case:

* Description: Verify that the API correctly identifies staff members with no admissions.
* Steps:
  + Send a request to the API endpoint to retrieve the list of staff with no admissions.
  + Verify that the response contains data listing staff members with no admissions.
* Expected Outcome: The API returns a list of staff members with no admissions.
* Result: Pass

Negative Test Case:

* Description: Verify that the API handles the case when all staff members have admissions recorded.
* Steps:
  + Simulate a scenario where all staff members have admissions recorded in the system.
  + Send a request to the API endpoint to retrieve the list of staff with no admissions.
  + Verify that the response contains an appropriate message indicating no staff members with no admissions.
* Expected Outcome: The API returns a message indicating no staff members with no admissions.
* Result: Pass

# Use of Tools, Practices, and Systems (such as Version Control, CI / CD) (10%)

Used continuous version control on GitHub, with small incremental changes per commit as an agile software development approach. This significantly helped in the circumstances I attempted to use JaCoCo as a code coverage plugin for my API. However, this raised many conflicts and stopped the API from running, so I reverted back to a previous commit to find another solution – which I settled on the in-built IntelliJ code coverage tool.

I approached both white boxing and black boxing methods for my API. White boxing from my unit and integration tests and black boxing methods incorporated by my usability test and manually testing the endpoints in the browser.

# Evaluation (10%)

Overall, I effectively met all the requirements of the API and front-end application, producing rigorous unit and integration tests that covered 87% of the code, whilst implementing a few tests for the front-end controller to build my experience with the @BeforeAll and @AfterAll tags for unit testing. What can be especially noted is all viable requests from the end-users can be seen to be implemented in the final front-end application.

In future, I would have liked to have done differently is implement continuous integration via GitHub Actions, so tests would have been routinely checked after each sprint. Additionally, improving the robustness of my endpoints to display more data or take more inputs such as a patient name instead of an ID for the admissions for a specific patient, so in the real world the application could be used more easily by employees if they don’t have access to a Patient ID on the system, just their name and NHS number.