Problem Based Learning Report

SENG3160, The University of Newcastle

Atharv Naphade

c3386230

# Contents

[Contents 1](#_Toc207665192)

[1. Abstract 2](#_Toc207665193)

[2. Literature Review 3](#_Toc207665194)

[2.1. Introduction 3](#_Toc207665195)

[2.2. Areas of Software Testing 4](#_Toc207665196)

[2.2.1. Unit and Integration Testing 4](#_Toc207665197)

[2.2.2. System Testing 5](#_Toc207665198)

[2.2.3. Acceptance Testing 6](#_Toc207665199)

[2.2.4. Contract Testing 6](#_Toc207665200)

[2.2.5. Non-functional Testing 6](#_Toc207665201)

[3. Testing Plan Proposal 7](#_Toc207665202)

[3.1. End-to-End tests for the Web client 7](#_Toc207665203)

[3.2. GraphQL & SQLite DB 8](#_Toc207665204)

[3.3. Cloud-independent testing for microservices 8](#_Toc207665205)

[3.4. Excluding Unit Tests 8](#_Toc207665206)

[4. Conclusion 9](#_Toc207665207)

[5. Bibliography 10](#_Toc207665208)

# Abstract

This report illustrates a comprehensive literature review of a few key software testing techniques, unit testing, integration testing, system testing, acceptance testing, contract testing, and non-functional testing.

# Literature Review

**Figure 2.1.**

**Different kinds of tests shown in relation to points/artifacts in the software development life cycle in conjunction to which they are performed**

Note. From The Art of Software Testing (3rd ed., p. 117), by Myers et al., 2012, John Wiley & Sons.

## Introduction

Testing is important component of software engineering as it ensures reliability, maintainability and performance. There are various types of testing techniques and processes that are undertaken at different points in the software development life cycle.

For any given software project, it is important to strategically identify areas that are testable and will potentially lead to improvements, mitigations, or provide insight into the health of the project at any given point.

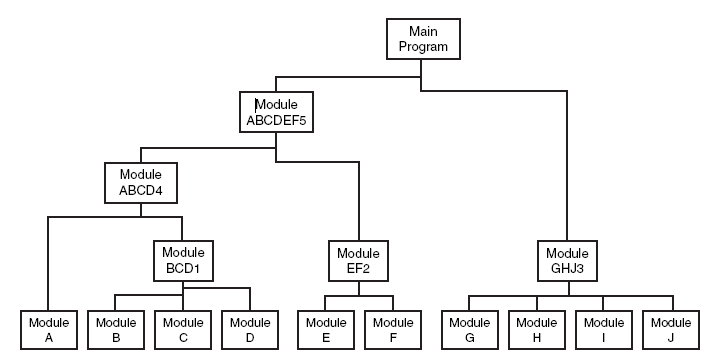
The following sections provide details on some of the popular types of software testing approaches.

## Areas of Software Testing

### Unit and Integration Testing

Module testing (or unit testing) is a process of testing the individual sub-programs, subroutines, classes, or procedures in a program (Myers et al., 2012, p. 85). Unit testing is usually undertaken parallelly to writing code, this allows for quick feedback and concrete definitions for each component’s behaviour such as inputs, outputs and edge cases. Myers et al., 2012 presents three motivations for why unit testing is used (p. 85) –

* It aids in managing other kinds of testing that involve multiple units working together, as it allows for working on the knowledge that each of the individual units are functioning well.
* It simplifies debugging by allowing developers to immediately direct their attention to concise units of the software program, in case of errors.
* It is a time and cost-efficient form of testing as it allows for parallelly running tests on all units, as they are detached from each other’s state.

Note. From Software Testing (2nd ed., p. 109), by R. Patton, 2005, Sams Publishing.

**Figure 2.2.**

Representation of modules within a software program, shown as being integrated for testing.

Integration testing is very closely linked to unit testing, as it involves testing the identified units’ interactions and finding any interfacing issues. Figure 2.2. illustrates how several units/modules in a software program can be organized together in a hierarchical manner. Examples of frameworks that enable unit testing are JUnit, Vitest, pytest, etc.

### System Testing

Following from Unit and Integration testing, System testing involves testing the proper functioning of the software program and its dependencies with all identified units running in unison. System testing also relies on a set of expected behaviours or measurable objectives (Myers et al., 2012, p. 120). It is although important to distinguish between objectives and client requirements, since the latter is part of Acceptance Testing, as is made clear in Figure 2.1.

### Acceptance Testing

The crucial stage in any procurement is the moment when the vendor offers the product to the buyer for inspection to determine whether the contract has been satisfied (UCC 2-606, as cited in Brannigan, 1985). This rather universal definition helps in understanding Acceptance testing fundamentally. When put in the context of software systems, Acceptance testing refers to tallying the functionality of the system with the initial client requirements, as is also defined in Figure 2.1. Successful adherence to the client requirements or otherwise can be demonstrated with high-fidelity end-to-end tests using modern testing frameworks that allow for browser automation and other platform specific UI automation. Examples of such frameworks include Cypress, Appium, UI Automator etc.

### Contract Testing

Contract testing is a methodology for ensuring that two separate systems (such as two microservices) are compatible and can communicate with one other (Fellows, 2023). Contracts tests provide a way to test, solidify and ensure compatibility of either requirements (consumer) or features (provider) between two products/systems such as APIs, microservices, etc. Fellows, 2023 in a blog article posted on pactflow.io, defines three strategies for contract testing with Pactflow

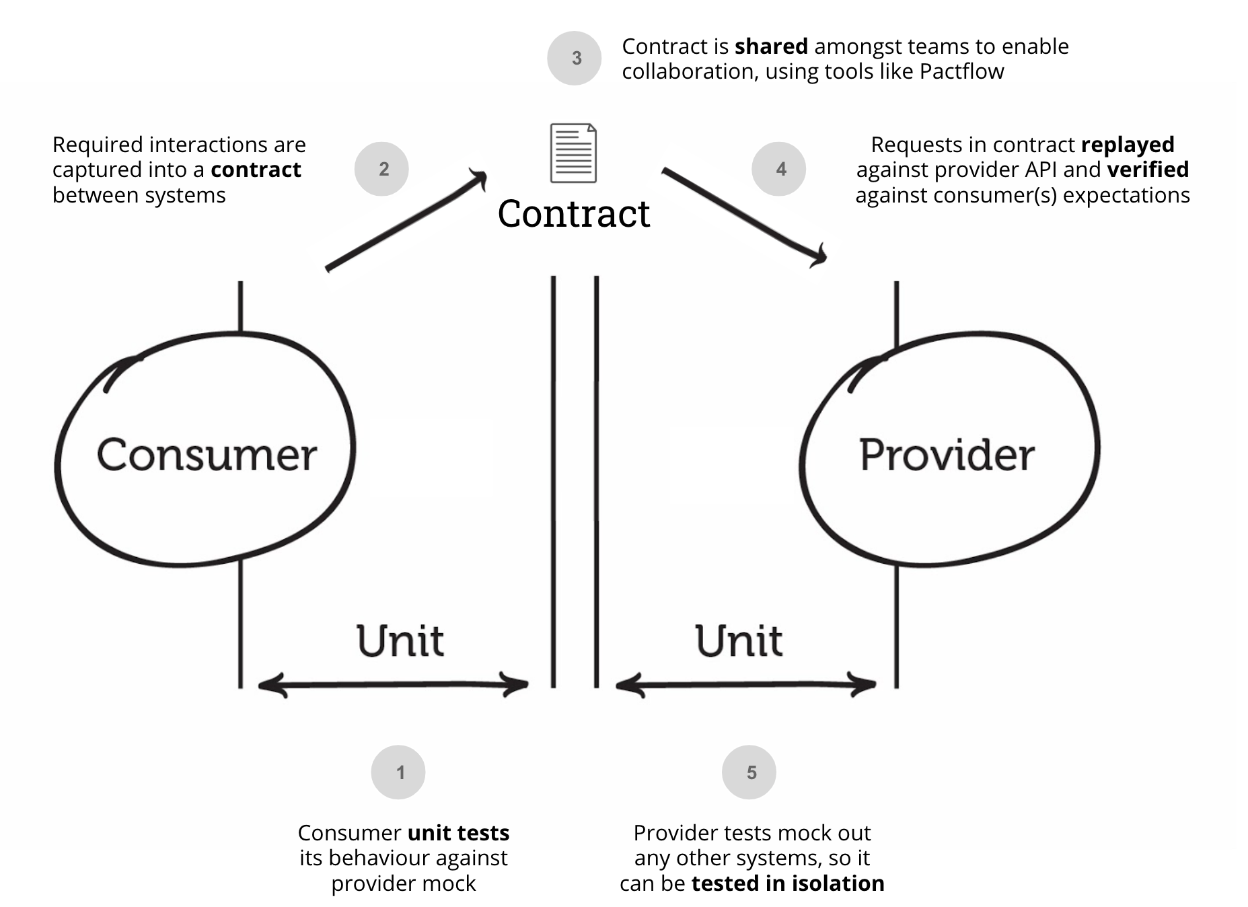
Note. From What is Contract Testing & How is it Used? | Pactflow, by Fellows M., 2023

Figure 2.3.

Consumer-driven contract testing implementation in Pactflow.

### Non-functional Testing

# Testing Plan Proposal

## End-to-End tests for the Web client

This will involve setting up a testing mode; capable of running the app in a high-fidelity environment, which would run the tests in complete isolation. Cypress Studio will be a crucial component on developing the tests, as it allows for recording UI flows and adding assertions for E2E tests by simply interacting with the app, as a user would in real life usage (Cypress Documentation, 2025). These recorded UI flows are then serialised and can replayed in CI tests.

The following defines the scope that these tests will aim to cover -

* Anonymous user register and login – 1.2.
* CRUD Teams – 1.9.
* CRUD Sessions – 1.4., 1.5.
* CRUD Activities – 1.6., 1.7., 1.8.
* Navigate the SPA in both arbitrary and expected manners – 2.5.

CRUD testing on all sections of the web app will start with seed data and involve valid, invalid and repetitive inputs for every operation. All tests will be asserted by checking for visual UI updates such as “New Activity created”, “Activity deleted”, “Team player deleted”, etc.

## API Testing

API Testing for CoachCraft’s GraphQL endpoints implemented with Fastify.

### Test Environment Setup

The testing environment setup will be fully isolated with mocked substitutes for any external services such as AUTH, to a reasonable degree. The following is an implementation outline -

* For every run of the test suite, create new SQLite files/DBs from existing schemas.
* Populate test DB from dummy data, dummy data from a pre-generated dataset which should be committed to the repository
* Implement “knobs” or configuration environment variables within the backend which will allow for easily switching to a testing mode.
* Implement a mock layer for authentication with AWS Cognito, consisting of the GraphQL endpoints blindly accepting the credentials for purposes of the test.

### Designing Tests

Designing test assertions for this test suite will consist of executing CRUD operations on all DB schema entities within scope of the end user. For each one of such entities, “happy” and “unhappy” input paths will be used (synthesized.io, 2024). “Happy” inputs will consist of generally expected, type and length accurate inputs, with assertions for expected server responses such as, server response codes and data formats/structures. “Unhappy” inputs will consist of a range of unexpected inputs such as,

* Mild – wrong type, incorrect length, etc
* Moderate – null type, incorrect data format/structure, etc.
* Bad faith – “fuzzed” inputs, code injection attempts, buffer overflow attack attempts, etc.

### Tooling

A set of off-shelf tools and frameworks’ features will be used for implementing the proposed test suite.

* fastify.inject() for simulating HTTP requests to the GraphQL endpoint (fastify.dev, n.d.). It allows for testing endpoints without starting up the HTTP server, easing setup.
* Vitest for assertions, structure test status logs, parallelism support for fast execution (vitest.dev, 2025).
* GitHub Actions for executing the test suite on various triggers such as, on creating PRs, merges into the main branch, etc.

## Excluding Unit Tests

# Conclusion

# Bibliography

Brannigan, V. (1985). Acceptance Testing - The Critical Problem in Software Acquisition. *IEEE Transactions on Biomedical Engineering, 1985-04, Vol.BME-32 (4), p.295-299*.

Cypress Documentation. (2025, August 22). *Cypress Studio: Build Tests by Interacting with Your App | Cypress Documentation*. https://docs.cypress.io/app/guides/cypress-studio

fastify.dev. (n.d.). *Testing | Fastify*. Retrieved September 2, 2025, from https://fastify.dev/docs/latest/Guides/Testing/

Fellows, M. (2023, January 5). *Contract Testing vs. Schema Testing | Pactflow*. https://pactflow.io/blog/contract-testing-using-json-schemas-and-open-api-part-1/

Fellows, M. (2023, September 2). *What is Contract Testing & How is it Used? | Pactflow*. https://pactflow.io/blog/what-is-contract-testing/

Myers, G. J. ., Sandler, Corey., & Badgett, Tom. (2012). The Art of Software Testing. In *The Art of Software Testing* (3rd ed.). John Wiley & Sons.

Patton, Ron. (2005). Software Testing. In *Software Testing* (2nd ed.). Sams Publishing.

synthesized.io. (2024, September 12). *Balancing Happy Paths, Golden Paths, and Negative Testing with AI-Driven Test Data Management - Blog - Synthesized*. https://www.synthesized.io/post/balancing-happy-path-negative-testing

vitest.dev. (2025, May 17). *Parallelism | Guide | Vitest*. https://vitest.dev/guide/parallelism