Application Testing

Benjamin Roberts

Level 4 Software Development

# Table of Contents

[Table of Contents 2](#_Toc178628100)

[Table of Figures 3](#_Toc178628101)

[Introduction 4](#_Toc178628102)

[Testing Frameworks 6](#_Toc178628103)

[Software 6](#_Toc178628104)

[Language Libraries 6](#_Toc178628105)

[Testing Methodologies 8](#_Toc178628106)

[Strategical Approach: Shift-Left Testing 8](#_Toc178628107)

[Behaviour Driven Development 10](#_Toc178628108)

[Test Driven Development 12](#_Toc178628109)

[Functional Testing 13](#_Toc178628110)

[Unit Testing 13](#_Toc178628111)

[System Testing 17](#_Toc178628112)

[Integration Testing 21](#_Toc178628113)

[Non-Functional Testing 23](#_Toc178628114)

[Performance Testing 23](#_Toc178628115)

[Destructive Testing 23](#_Toc178628116)

[Security Testing (Penetration) 23](#_Toc178628117)

[Conclusion 24](#_Toc178628118)

[Bibliography 25](#_Toc178628119)

[Criteria 27](#_Toc178628120)

# Table of Figures

[Figure 1 - Internal Email Communication with IT Development & Testing Leads (names redacted for privacy) 4](#_Toc178628055)

[Figure 2- Email Response from IT with starting point suggestions 5](#_Toc178628056)

[Figure 3 - Testing Framework Software 6](#_Toc178628057)

[Figure 4- Testing Framework Language libraries 7](#_Toc178628058)

[Figure 5- Microsoft Teams meeting for discussing approaches to Software Testing 8](#_Toc178628059)

[Figure 6- Traditional view o Shift-Left testing, with various forms of testing applied (Wikimedia.org, 2024) 9](#_Toc178628060)

[Figure 7 - Escalating costs of bugs/faults as they are found later in the Development process (Deepsource, 2019) 9](#_Toc178628061)

[Figure 8 - Table Breakdown of Gherkin Keywords (Cucumber, 2024) 10](#_Toc178628062)

[Figure 9- Generic example of a complete Gherkin Test case (SmartBear, 2023a) 11](#_Toc178628063)

[Figure 10 - Example of a workplace Test case written in Gherkin language (with sensitive information redacted) 11](#_Toc178628064)

[Figure 11- Visual of Test Driven Development (IBM, 2024) 12](#_Toc178628065)

[Figure 12 - Typical Hierarchy to order different types of Functional Testing is conducted (GeeksForGeeks, 2024b) 13](#_Toc178628066)

[Figure 13 - Unit Test cases 15](#_Toc178628067)

[Figure 14- Java JUnit representation of a Unit Test my team would write within our Credit Decisioning platform 15](#_Toc178628068)

[Figure 15- Unit Test Results as seen in our Credit Decisioning platform 16](#_Toc178628069)

[Figure 16 - Flowchart of the workflow for conducting System Testing (GeeksForGeeks, 2024b) 17](#_Toc178628070)

[Figure 17- Generic Postman Test to check HTTP Status Code received 17](#_Toc178628071)

[Figure 18 - System Test cases 18](#_Toc178628072)

[Figure 19 - Using Postman to load Test data for System Testing 19](#_Toc178628073)

[Figure 20 - Postman Test code in practice (with sensitive information redacted) 19](#_Toc178628074)

[Figure 21 - Results for System Test Case 1 20](#_Toc178628075)

[Figure 22 - Results for System Test Case 2 20](#_Toc178628076)

[Figure 23 - Results for System Test Case 3 20](#_Toc178628077)

[Figure 24- Visual Representation of Integration Testing (SmartBear, 2024a) 21](#_Toc178628078)

# Introduction

This report covers my research into Testing Frameworks and Methodologies, undertaken as an upskilling exercise so I can expand my knowledge and toolkit for ensuring the quality of software I work on. This will cover:

* A **theory review** of various Testing Frameworks and Methodologies accepted by the industry (with references)
* Provide details on which **Frameworks and Methodologies** are **used within my team**
* Expand on this via reaching out to our dedicated IT Testing teams, who I know engage in more testing methodologies then my team do, which will allow a view of testing in the wider organisation

Also covered are my efforts to apply these Testing Frameworks and Methodologies. To do this I put together a Test Plan for testing a deployed version of an implementation of our Credit Decisioning system. Within this plan I cover:

* **System Testing** - conducted via the Postman software, which makes use of JavaScript test cases via an implementation of the Chai library for running test cases on the HTTP response received from a web application
* **Integration testing** – for testing that our Credit Decisioning system integrates with 3rd party API's successfully (i.e. no errors returned for multiple real-world scenarios)
* **Performance Testing** - via testing peak volumes the system can handle

***Note***: As showing specific workings of our Credit Decisioning system would be considered trade secrets, certain information (e.g. code showing data structures) will be censored.

To get started with this, I consulted our IT Development and Testing team leads for information on what methodologies and frameworks were used within the workplace. I approached these stakeholders as I knew they had a higher level of technical expertise then my team (who are a more hybrid team).

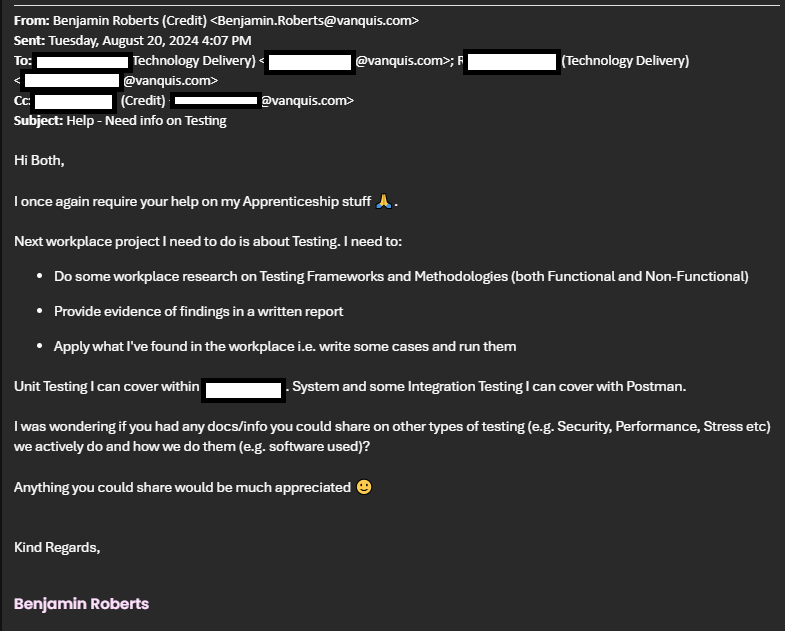


Figure 1 - Internal Email Communication with IT Development & Testing Leads (names redacted for privacy)

Below shows the response I got from the stakeholders. They provided suggestions of where to start with methodologies and frameworks for my theory review, along with suggestions of tools I could look at for conducting testing. Also was an invitation to discuss further in person.

A screenshot of a computer

Description automatically generated

Figure 2- Email Response from IT with starting point suggestions

With this starting point, I’ve broken down the theory review into multiple sections below.

# Testing Frameworks

Software Testing Frameworks are a set of guidelines or rules used for creating and designing test cases and are comprised of tools and practices used to run these test cases. These could be providing functionality to users for writing tests, methods of managing test data, processes for storing results and more (Smartbear, 2024b). Applied, these refer to the tools we use to conduct testing activities.

Various software and language libraries have been developed to support teams conducting testing. Some examples of each are covered below:

## Software

|  |  |  |
| --- | --- | --- |
| **Tool** | **Description** | **Types of Testing used for** |
| Postman | A software designed to support teams with API development by providing tools to document and test API interfaces offered over the web (Postman, 2024) | * Integration * System * Performance * Destructive |
| JMeter | An open source application built in pure Java designed for load testing functional behaviour and measuring performance (Apache, 2024) | * Integration * System * Performance |
| Selenium | Automated web browser input tool, allowing teams to create scripts that automate interaction with a Web App | * System * End to End * User Acceptance * Cross-Browser |
| Burp Suite | Provides a suite of tools for security testing web applications, used for identifying vulnerabilities within the application. Includes features for SQL Injection, Cross-site scripting and more (PortSwigger, 2024) | * Security |

Figure 3 - Testing Framework Software

## Language Libraries

|  |  |  |
| --- | --- | --- |
| **Language** | **Library** | **Functionality provided** |
| Java | JUnit | Unit Testing functionality, with integration with the IDE and Continuous Integration tools |
| Mockito | Mocking services, allowing certain method behaviour to be set to always return an expected result. |
| C# | NUnit | Unit Testing functionality, with integration with the IDE and Continuous Integration tools |
| moq | Mocking services, allowing certain method behaviour to be set to always return an expected result. |
| Javascript / NodeJs | Chai | Unit Testing functionality, with integration with the IDE and Continuous Integration tools |
| Python | Pytest | Unit Testing functionality, with integration with the IDE and Continuous Integration tools |

Figure 4- Testing Framework Language libraries

# Testing Methodologies

Software Testing Methodologies are various strategies/approaches used for testing an application for ensuring it looks and behaves as per requirements (SmartBear, 2024a). These form the strategic approach to how developers/teams will test a piece of software. The types of testing can be split into types:

1. **Functional**, including Unit, Integration, System and User Acceptable Testing
2. **Non**-**Functional**, including Performance, Security and Accessibility Testing

Aside from these different types of testing, an overall strategical plan for testing is required to be able to apply these different forms of testing effectively when developing software.

## Strategical Approach: Shift-Left Testing

To get started, I took up an offer from one of our technical IT Delivery Leads to discuss approaches to testing one-to-one in greater detail. I set up a Web Meeting for this:

A screenshot of a computer

Description automatically generated

Figure 5- Microsoft Teams meeting for discussing approaches to Software Testing

Our Delivery Lead gave me good information to get started on and recommendations on what to investigate. One highlight was the “Shift-Left” approach to testing. This is an approach that aims to perform testing earlier in the development cycle, rather than having isolated phases of development and testing (IBM, 2023).

Shift Left Testing is accomplished by using various kinds of testing that can be executed throughout the software’s development, as demonstrated in the figure below:

A diagram of a software development process

Description automatically generated

Figure 6- Traditional view o Shift-Left testing, with various forms of testing applied (Wikimedia.org, 2024)

Here, we see high level requirements broken down as the software development cycle continues, until we get down to the design and coding levels. From there, we immediately start different forms of testing, therefore moving testing from the end of the lifecycle to be directly integrated with the development. This is the “V-Model”, which allows development team to verify early on that their solutions match the requirements provided and validate they solve the business problems that the software is intended to address (IBM, 2023).

A key benefit of moving testing earlier into the cycle like this is that it allows bugs/faults to be identified earlier in the software development process. Bugs/Faults found later in the process lead to more costs as the software needs to be sent back to the development teams to fix, or in a worst-case scenario can create production level incidents that can cost the business significantly (ether financially or reputationally). The below figure visualises how cost to fix bug/faults increases as they are found later in the process:

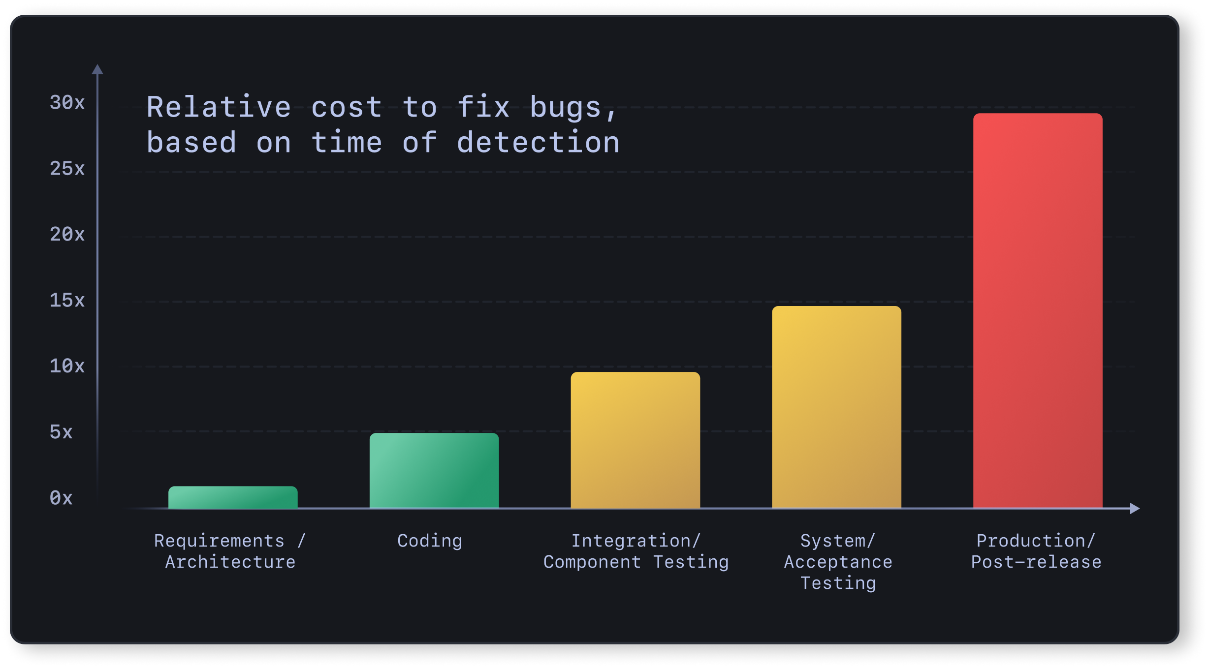


Figure 7 - Escalating costs of bugs/faults as they are found later in the Development process (Deepsource, 2019)

Dedicated end to end testing phases are still important under this strategy, but Shift-left allows potentially costly bugs to be found earlier in the process.

### Behaviour Driven Development

Behaviour Driven Development is one way to apply this shift left strategy by bringing user requirements directly into the development and testing processes and using them as a basis for writing test cases. The idea is to write test cases in a plain English language that bridges the gap between technical and non-technical stakeholders, which simplifies communicating test results and is effective for getting business approvals to move to deployment (North, 2006). The test cases then form requitements the development team can use to develop from and communicate the status of those features back to the business easily. Test cases are therefore considered very early in the software development cycle i.e. testing “shifts-left”.

When meeting with internal technical stakeholders who lead our testing teams, they advised this is something we apply internally in our test cases via the “**Gherkin**” language. Gherkin is a plain text language design to be easy for non-programmers to use while also being concise enough to write effective test scenarios that emulate real-world business use cases (SmartBear, 2023a). The syntax makes use of various keywords to structure the test case, which every line must start with (Cucumber, 2024). Common ones I observed in our internal plans are:

|  |  |
| --- | --- |
| **Keyword** | **Meaning** |
| **Scenario/Example** | * “Scenario” is a synonym of “Example” i.e. Gherkin treats them as the same meaning * Used for describing a specific business rule the functionality under test is for |
| **Feature** | * Provides a high-level Description of a software feature (i.e. a piece of functionality to be delivered). * Useful for grouping related scenarios * Must always be the 1st Keyword |
| **Given** | * Provides the initial context of a test case * Idea is to put the system in a known state before any other interactions * Form the preconditions that must be met before executing a test case |
| **When** | * Describes an event/action taken by the user or system i.e. what the user/system would be doing when the test case is run |
| **Then** | * Describes the expected outcomes of the test case |
| **And** | * Used to join multiple “**Given**” statements * Also can be used to join multiple “**Then**” statements * Useful for making the test case more readable |
| **But** | * Similar to “**And**”, but can be used when the test case not intending for an action to happen e.g.   + *Then I should see something*   + *But I shouldn't see something else* |

Figure 8 - Table Breakdown of Gherkin Keywords (Cucumber, 2024)

Put together, a Gherkin test case example is:

A screenshot of a computer screen

Description automatically generated

Figure 9- Generic example of a complete Gherkin Test case (SmartBear, 2023a)

I found in my organisation, our testing teams use Azure DevOps test plans, with the Gherkin language being used to structure the test cases. Below is an example of a Gherkin test case from one of our internal test plans:

A screenshot of a computer

Description automatically generated

Figure 10 - Example of a workplace Test case written in Gherkin language (with sensitive information redacted)

### Test Driven Development

Test Driven development is a precursor to Behaviour-Driven Development, with North (2005) using their experience with teaching Test Driven Development to develop the Behaviour-Driven Development methodology. It is similar in context due to that, although they are not mutually exclusive (Cucumber, 2024)

Visualised, the workflow looks like this:



Figure 11- Visual of Test Driven Development (IBM, 2024)

The approach works by writing test cases for a feature before writing any code for the feature itself, knowing that the test cases will fail initially. The Development will then write just enough code to allow the test case to pass, therefore bringing part of the testing earlier into the Software Development cycle. By writing the test cases upfront like this, the development team can maximise the code coverage.

Once these test pass, then the development team is free to refactor the code, which could be for optimisation or for conforming to coding standards set by the client. With the Tests already in place, they are free to make these changes and rapidly re-run the tests to confirm their changes have not introduced a defect into the code. This also allows for quick regression testing of existing functionality once a new feature request comes in.

## Functional Testing

Functional Testing is about the application against the business requirements set e.g. does a specific feature perform per expectations (SmartBear, 2024a). Different forms of Functional testing are covered below.

Typically, there is an order to how these forms of Functional testing are done, as seen below:

A diagram of a software testing process

Description automatically generated

Figure 12 - Typical Hierarchy to order different types of Functional Testing is conducted (GeeksForGeeks, 2024b)

### Unit Testing

Unit Testing involved testing small pieces of code in isolation, allowing the behaviour of the component under test to be proven without any dependency on other parts of the software (AWS, 2024). This is the typical 1st level of testing and is often performed by the development team (SmartBear, 2024a). The advantage of this is that it allows a developer to confirm that their written code performs the function it’s expected to without having to design specific test data to handle other components that could block access to the function under test.

My team do this kind of testing within our Credit Decisioning platform as a standard part of our development process. Each piece of functionality has multiple unit tests created for it to demonstrate every possible outcome the component under test could return.

**Test Plan**

Our internal Credit Decisioning engine uses a low code solution provided by a 3rd party supplier, which I’ve had prior training in. To expand my horizons, I’ve instead looked to conduct Unit Tests on a component from a previous personal project that was written in Java, allowing me to practice this form of testing in a more traditional coding space using the JUnit library.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Class** | **Type** | **Component Under Test (CUT)** | **Test Case (Gherkin)** |
| 1 | Functional | Unit | Scorecards - e1b09\_score method | **Given** an applicant is applying for a loan  **When** they reach the Scorecard module  **And** the Champion Scorecard is run  **And** the value for the E1B09 variable is -1  **Then** the output of “e1b09\_score” will be -20 |
| 2 | Functional | Unit | Scorecards - e1b09\_score method | **Given** an applicant is applying for a loan  **When** they reach the Scorecard module  **And** the Champion Scorecard is run  **And** the value for the E1B09 variable is 0  **Then** the output of “e1b09\_score” will be -20 |
| 3 | Functional | Unit | Scorecards - e1b09\_score method | **Given** an applicant is applying for a loan  **When** they reach the Scorecard module  **And** the Champion Scorecard is run  **And** the value for the E1B09 variable is 1  **Then** the output of “e1b09\_score” will be 10 |
| 4 | Functional | Unit | Scorecards - e1b09\_score method | **Given** an applicant is applying for a loan  **When** they reach the Scorecard module  **And** the Champion Scorecard is run  **And** the value for the E1B09 variable is 2  **Then** the output of “e1b09\_score” will be 10 |
| 5 | Functional | Unit | Scorecards - e1b09\_score method | **Given** an applicant is applying for a loan  **When** they reach the Scorecard module  **And** the Champion Scorecard is run  **And** the value for the E1B09 variable is 3  **Then** the output of “e1b09\_score” will be 20 |
| 6 | Functional | Unit | Scorecards - e1b09\_score method | **Given** an applicant is applying for a loan  **When** they reach the Scorecard module  **And** the Champion Scorecard is run  **And** the value for the E1B09 variable is 4  **Then** the output of “e1b09\_score” will be 20 |
| 7 | Functional | Unit | Scorecards - e1b09\_score method | **Given** an applicant is applying for a loan  **When** they reach the Scorecard module  **And** the Champion Scorecard is run  **And** the value for the E1B09 variable is 5  **Then** the output of “e1b09\_score” will be 30 |
| 8 | Functional | Unit | Scorecards - e1b09\_score method | **Given** an applicant is applying for a loan  **When** they reach the Scorecard module  **And** the Champion Scorecard is run  **And** the value for the E1B09 variable is 5  **Then** the output of “e1b09\_score” will be 30 |
| 9 | Functional | Unit | Scorecards - e1b09\_score method | **Given** an applicant is applying for a loan  **When** they reach the Scorecard module  **And** the Champion Scorecard is run  **And** the value for the E1B09 variable is 7  **Then** the output of “e1b09\_score” will be 18 |
| 10 | Functional | Unit | Scorecards - e1b09\_score method | **Given** an applicant is applying for a loan  **When** they reach the Scorecard module  **And** the Champion Scorecard is run  **And** the value for the E1B09 variable is greater than 7 (e.g. 8)  **Then** the output of “e1b09\_score” will be 18 |
| 11 | Functional | Unit | Scorecards - e1b09\_score method | **Given** an applicant is applying for a loan  **When** they reach the Scorecard module  **And** the Champion Scorecard is run  **And** the value for the E1B09 variable is greater than 7 (e.g. 8)  **Then** the output of “e1b09\_score” will be 18 |
| 12 | Functional | Unit | Scorecards - e1b09\_score method | **Given** an applicant is applying for a loan  **When** they reach the Scorecard module  **And** the Champion Scorecard is run  **And** the value for the E1B09 variable has a value that falls outside the provided business logic (e.g. -999)  **Then** the output of “e1b09\_score” will be -20 |
| 13 | Functional | Unit | Scorecards - e1b09\_score method | **Given** an applicant is applying for a loan  **When** they reach the Scorecard module  **And** the Champion Scorecard is run  **And** the value for the E1B09 variable has a NULL value  **Then** the output of “e1b09\_score” will be -20 |

Figure 13 - Unit Test cases

The below figure shows these test cases being applied in a JUnit class called **ChampionTest**, which calls the static “e1b09\_score” method from the Champion class:

A screenshot of a computer program

Description automatically generated

Figure 14- Java JUnit representation of a Unit Test my team would write within our Credit Decisioning platform

Unit Tests work on the concept of Assertions, which define a TRUE/FALSE condition that must evaluate to TRUE for the Unit Test to pass. Different Frameworks provide various ways to define these Assertions, but a common one is “AssertEquals”, which tests if the output of the component under test equals an expected value. In this example, the 1st parameter passed into the “AssertEquals” method represents the expected value, while the 2nd parameter is the component under test. The method being passed in as the 2nd parameter returns a value back to “AssertEquals”, which can then determine if the test passes or fails by comparing the 2 parameters. Developer Tools the present the result back to the developer e.g. in our Credit Decisioning platform, this gets presented within a specific view available to the platform, seen below:

A screenshot of a computer

Description automatically generated

Figure 15- Unit Test Results as seen in our Credit Decisioning platform

Successful tests appear as regular text within this view, while failed tests will appear in red with a “FAIL” indicator next to them.

Once developed, Unit Tests can be run repeatedly. This makes them useful for confirming a component still works as intended if changes are made to it later.

### System Testing

Systems testing is done on a completely integrated system to check the developed software meets the original business requirements and is typically done after Unit and Integration testing (GeeksForGeeks, 2024b). This process is core for ensuring the final product can deliver value to the business.

Typically, this testing is done by a dedicated test team, rather than the developers (SmartBear, 2024a). This separation gives the testers independence to test the software from an impartial perspective (GeeksForGeeks, 2024b), which can allow them to find issues that may not occur to the development team.

If any gaps/faults between the system and the requirements are found, these are communicated back to the development team to investigate.

Regression testing can also be done as part of this, which involved re-running test cases on existing features to ensure they have not created new defects in existing functionality.

Below shows a flowchart for the Process:

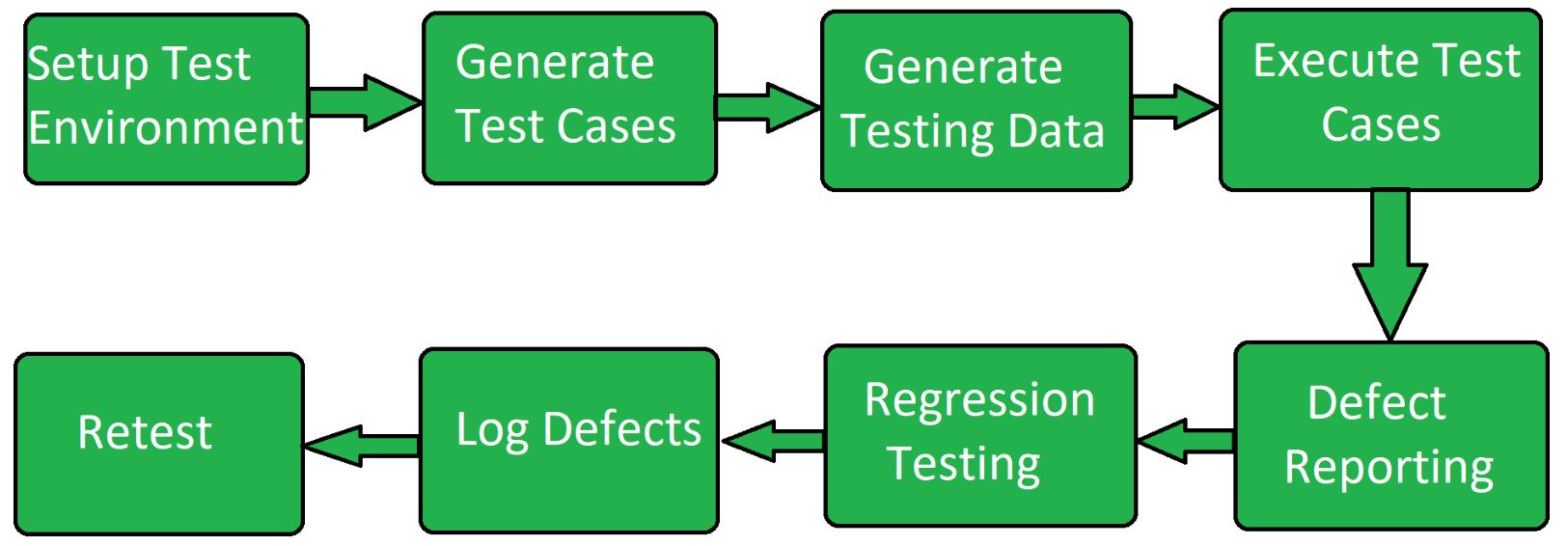


Figure 16 - Flowchart of the workflow for conducting System Testing (GeeksForGeeks, 2024b)

My team primarily uses the Postman software to conduct this kind of testing, specifically for its ability to send HTTP requests to API endpoints, which allows us to submit test cases to our Credit Decisioning engine. Our use case however is to rely on MI Files generated by the system that get loaded into a SQL Database. We use these files to validate the system works as expected via recreating the entire business logic in SQL.

Our current approach does not take full use of this software, as has it’s own ability to run Test Scripts on the results of an API call, which can be used to conduct system testing e.g. the figure below is an auto-generated one to confirm the response received from a HTTP Service has the 200 status code, indicating a success:

A screenshot of a computer program

Description automatically generated

Figure 17- Generic Postman Test to check HTTP Status Code received

**Test Plan**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Class** | **Type** | **Component Under Test (CUT)** | **Test Case (Gherkin)** |
| 1 | Functional | System | Policy Rule – Existing Customer Decline | **Given** a single applicant is applying for a loan **When** the applicant is an is an existing customer **And** the applicants name, current postcode and date of birth match exactly to the existing customer database **Then** that applicant will be declined  **And** a HTTP Status Code 200 shall be received **And** *[Redacted]* policy code shall be returned to the front-end system |
| 2 | Functional | System | Policy Rule – Exclusions Decline | **Given** a single applicant is applying for a loan **When** the applicant is on the exclusion list **And** the applicants name, current postcode and date of birth match exactly to the exclusion list **Then** that applicant will be declined  **And** a HTTP Status Code 200 shall be received **And** *[Redacted]* policy code shall be returned to the front-end system |
| 3 | Functional | System | Data – Existing Customer agreement | **Given** a single applicant is applying for a loan **When** the applicant is an is an existing customer **And** the applicants name, current postcode and date of birth match exactly to the existing customer database  **Then** a HTTP Status Code 200 shall be received  **And** the current agreement number for the applicant will be returned in the response |

**Note**: Exact response structure these tests expect are redacted as this is sensitive information.

Figure 18 - System Test cases

Before I could do anything, I needed some test data to match against (as a precondition for these tests to be valid). I have built functionality in the past within our Credit Decisioning system for loading data into the back-end database the components in these test cases use. This works by using a HTTP POST Request with a request body containing the data to load. The figure below shows this in practice:

A screenshot of a computer

Description automatically generated

Figure 19 - Using Postman to load Test data for System Testing

To run these cases, I also wrote functions in Postman’s Scripts tab to check the JSON Response received for the expected data specified in the test case (the structure of which is similar for all 3 cases). Using the 1st test case as an example, these work by:

1. Wrapping within the “pm.test” method, which Postman uses to capture the result for it’s graphical representation of the results
2. Parsing the JSON message received and committing to a JavaScript array
3. Using a forEach loop on the above array to check for the expected data outputs
4. Using the “pm.expect” method to return a PASS/FAIL result back to the Postman UI, which is presented to the user

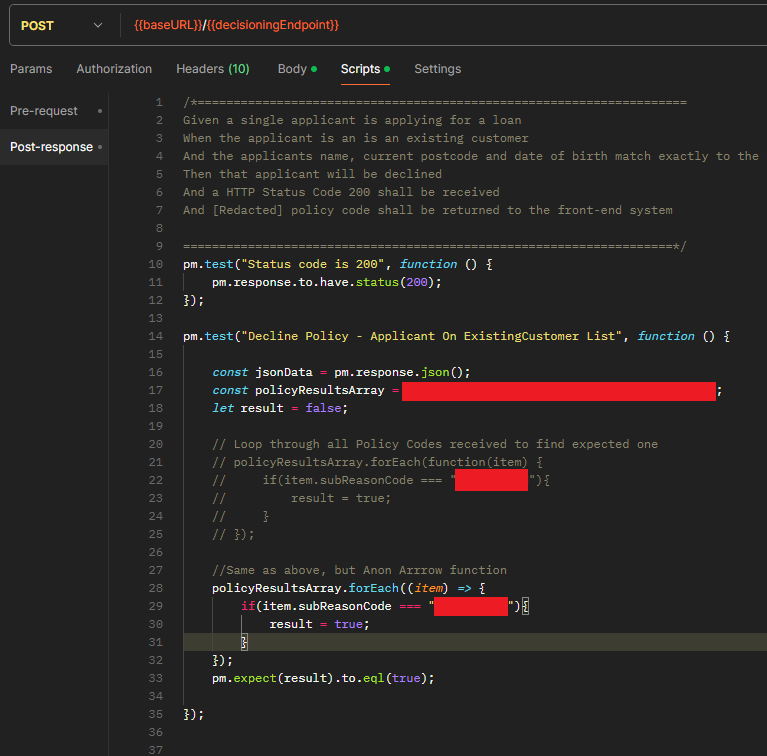


Figure 20 - Postman Test code in practice (with sensitive information redacted)

Once Postman is used to send an HTTP request, these test cases are run automatically and presented in a menu, as seen below for my 3 test cases:

A black screen with a green border

Description automatically generated

Figure 21 - Results for System Test Case 1

A black surface with white text

Description automatically generated

Figure 22 - Results for System Test Case 2

A black screen with white text

Description automatically generated

Figure 23 - Results for System Test Case 3

By using Postman’s functionality for Test Scripts like this, I see an opportunity to improve how my team conducts testing. System testing is core to use because it validates the business logic we implement performs as expected. Our current approach of relying on MI files loaded into a SQL database however creates an overhead due to the time it takes to receive and load the files into a database. This creates inefficiency and slows down testing. We may be able to use Postman like this to better automate our testing.

### Integration Testing

Integration Testing tests the interface between 2 systems or modules, looking for potential issues that would impact the functionality of both systems/modules. This is typically done after Unit Testing and before System Testing (GeeksForGeeks, 2024a). Unit Tests would instead make use of mocking services to force specific responses from integrated systems where the components under test would require then, which keeps the scopes of the test scenarios separate.

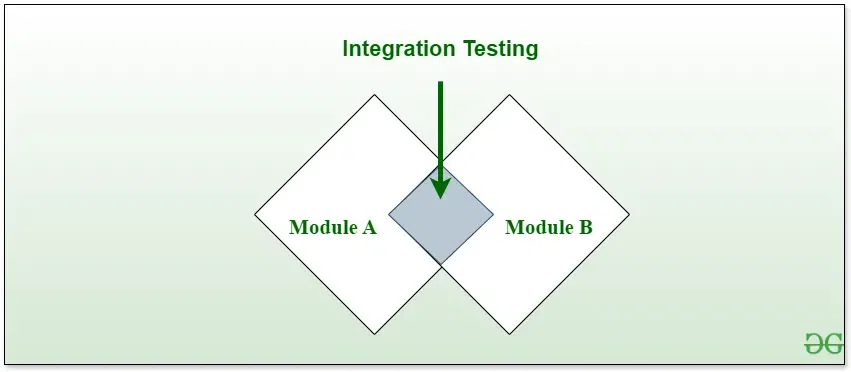


Figure 24- Visual Representation of Integration Testing (SmartBear, 2024a)

An applied example would be testing the communication one system has with integrated web services e.g. our internal Credit Decisioning engine integrates with API’s offered by the 3 main credit bureaus in the UK: Experian, TransUnion and Equifax. These providers return data on a person’s Credit file, which can be used for assessing a person’s credit worthiness in a more informed manner then solely relying on data provided by a person or from internal data the business holds.

For the benefits of the example to materialise however, the business needs a degree of confidence that the Credit Decisioning system can retrieve a proper response from the source API. Test cases therefore need to cover:

1. A successful connection can be made to the credit bureau API
2. The data returned is per expectations, based on the input provided e.g. if a person were to provide multiple addresses, the decisioning engine sends all of them to the credit bureau API and all are searched on the credit bureau’s service
3. The response from the credit bureau API can be successfully interpreted by the originating system

**Test Plan**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Class** | **Type** | **Component Under Test (CUT)** | **Test Case (Gherkin)** |
| 1 | Functional | Integration | DelphiSelect (Quotation) | **Given** a single applicant is applying for a loan **When** a single address is given on their application **And** the applicant is not decline on any pre-bureau policy rules **Then** the system will make an HTTP API call to Experian DelphiSelect **And** a successful Response will be received **And** the provided address will be present on the searched addresses in the Delphiselect response |
| 2 | Functional | Integration | DelphiSelect (Quotation) | **Given** a single applicant is applying for a loan **When** multiple addresses are given on their application **And** the applicant is not decline on any pre-bureau policy rules **Then** the system will make an HTTP API call to Experian DelphiSelect **And** a successful Response will be received **And** the provided address will be present on the searched addresses in the Delphiselect response |
| 3 | Functional | Integration | DelphiSelect (Quotation) | **Given** joint applicants are applying for a loan **When** multiple addresses are given on their application **And** the applicant is not decline on any pre-bureau policy rules **Then** the system will make an HTTP API call to Experian DelphiSelect **And** a successful Response will be received **And** the provided address will be present on the searched addresses in the Delphiselect response |

## Non-Functional Testing

Non-Functional Testing focuses on the operational aspects of the software, rather than the features of the software e.g. testing how the software performs with multiple users in parallel (SmartBear, 2024a). Various types of this are covered below.

### Performance Testing

### Destructive Testing

### Security Testing (Penetration)

Security Testing (also called Penetration Testing) is about discovering vulnerabilities within the application that a malicious user could take advantage of for various ends e.g. accessing user data, sabotage etc.

# Conclusion

* Many forms of testing, many more beyond what is covered here
* Strategic Focus is to bring in as early as possible, costly otherwise
* Many frameworks to do this with

# Bibliography

Apache, 2024. *Apache JMeter.* [Online]   
Available at: https://jmeter.apache.org/  
[Accessed 27 09 2024].

AWS, 2024. *What is Unit Testing?.* [Online]   
Available at: https://aws.amazon.com/what-is/unit-testing/#:~:text=Unit%20testing%20is%20the%20process,test%20for%20each%20code%20unit.  
[Accessed 20 09 2024].

Cucumber, 2024. *Gherkin Reference.* [Online]   
Available at: https://cucumber.io/docs/gherkin/reference/  
[Accessed 20 09 2024].

Cucumber, 2024. *Understanding the Differences Between BDD & TDD.* [Online]   
Available at: https://cucumber.io/blog/bdd/bdd-vs-tdd/  
[Accessed 20 09 2024].

Deepsource, 2019. *The exponential cost of fixing bugs.* [Online]   
Available at: https://deepsource.com/blog/exponential-cost-of-fixing-bugs  
[Accessed 20 09 2024].

GeeksForGeeks, 2024a. *Integration Testing – Software Engineering.* [Online]   
Available at: https://www.geeksforgeeks.org/software-engineering-integration-testing/   
[Accessed 20 09 2024].

GeeksForGeeks, 2024b. *System Testing – Software Engineering.* [Online]   
Available at: https://www.geeksforgeeks.org/system-testing/  
[Accessed 20 09 2024].

IBM, 2023. *What is shift-left testing?.* [Online]   
Available at: https://www.ibm.com/topics/shift-left-testing  
[Accessed 13 09 2024].

IBM, 2024. *5 steps of test-driven development.* [Online]   
Available at: https://developer.ibm.com/articles/5-steps-of-test-driven-development/  
[Accessed 30 09 2024].

North, 2006. *Introducing BDD.* [Online]   
Available at: https://dannorth.net/introducing-bdd/  
[Accessed 20 09 2024].

PortSwigger, 2024. *Burp Suite Professional Features.* [Online]   
Available at: https://portswigger.net/burp/pro/features  
[Accessed 30 09 2024].

Postman, 2024. *Tools.* [Online]   
Available at: https://www.postman.com/product/tools/  
[Accessed 20 09 2024].

SmartBear, 2023a. *Writing scenarios with Gherkin syntax.* [Online]   
Available at: https://support.smartbear.com/cucumberstudio/docs/bdd/write-gherkin-scenarios.html  
[Accessed 20 09 2024].

SmartBear, 2024a. *Software Testing Methodologies.* [Online]   
Available at: https://smartbear.com/learn/automated-testing/software-testing-methodologies/  
[Accessed 20 09 2024].

Smartbear, 2024b. *Test Automation Frameworks.* [Online]   
Available at: https://smartbear.com/learn/automated-testing/test-automation-frameworks/  
[Accessed 27 09 2024].

Wikimedia.org, 2024. *Traditional-Shift-Left.* [Online]   
Available at: https://upload.wikimedia.org/wikipedia/commons/2/2c/Traditional-Shift-Left.jpg  
[Accessed 20 09 2024].

# Criteria

**Knowledge**

* K12 - Software testing frameworks and methodologies

**Skills**

* S13 - Follow testing frameworks and methodologies
* S5 - Conduct a range of test types, such as Integration, System, User Acceptance, Non-Functional, Performance and Security testing

**Behaviours**

* B10 - Committed to continued professional development
* B4 - Works collaboratively with a wide range of people in different roles, internally and externally, with a positive attitude to inclusion and diversity
* B7 - Communicates effectively in a variety of situations to both a technical and non- technical audience
* B5 - Acts with integrity with respect to ethical, legal and regulatory ensuring the protection of personal data, safety and security
* B1 - Works independently and takes responsibility. For example, has a disciplined and responsible approach to risk, and stays motivated and committed when facing challenges
* B6 - Shows initiative for solving problems within their own remit, being resourceful when faced with a problem to solve
* B8 - Shows curiosity to the business context in which the solution will be used, displaying an inquisitive approach to solving the problem. This includes the curiosity to explore new opportunities, and techniques; the tenacity to improve methods and maximise performance of the solution; and creativity in their approach to solutions
* B9 - Demonstrates creativity and tenacity in their approach to solutions and the methods used to come to a solution, for example sees the task through to the end by devising new solutions and despite obstacles and problems along the way