Table of Contents

[1. Introduction 3](#_Toc98600107)

[1.1 Overview 3](#_Toc98600108)

[1.2 Reference Materials 3](#_Toc98600109)

[1.3 Definitions and Acronyms 3](#_Toc98600110)

[2. Testing Strategy 4](#_Toc98600111)

[2.1 Objectives and Tasks 5](#_Toc98600112)

[2.2 Test Assumptions 5](#_Toc98600113)

[2.3 Test Principles 5](#_Toc98600114)

[2.4 Data Approach 5](#_Toc98600115)

[2.5 Scope and Levels of Testing 5](#_Toc98600116)

[2.5.1 Scope 5](#_Toc98600117)

[3. Unit Testing 5](#_Toc98600118)

[3.1 Definition 5](#_Toc98600119)

[3.2 Popular Frameworks and Why xUnit 6](#_Toc98600120)

[3.3 Installation of xUnit 7](#_Toc98600121)

[3.4 Three Styles of Unit Testing 7](#_Toc98600122)

[3.4.1 Output-Based Style 7](#_Toc98600123)

[3.4.2 State-Based Style 7](#_Toc98600124)

[3.4.3 Communication-Based Style 8](#_Toc98600125)

[3.5 Naming Conventions 8](#_Toc98600126)

[3.5.1 Naming a Test Project 8](#_Toc98600127)

[3.5.2 Naming a Test Class 8](#_Toc98600128)

[3.5.3 Naming a Unit Test Method 9](#_Toc98600129)

[3.6 Test Structure 11](#_Toc98600130)

[3.7 xUnit Attributes 12](#_Toc98600131)

[3.7.1 [Fact] 12](#_Toc98600132)

[3.7.2 [Theory] 12](#_Toc98600133)

[3.7.3 [InlineData] 12](#_Toc98600134)

[3.7.4 [MemberData] 12](#_Toc98600135)

[3.7 Stubs 12](#_Toc98600136)

[3.8 Mocking with MOQ & xUnit 12](#_Toc98600137)

[3.9 Live Unit Testing Approch 12](#_Toc98600138)

[4. System and Integration Testing 12](#_Toc98600139)

[4.1 Definition 13](#_Toc98600140)

[5. Data Base Testing 13](#_Toc98600141)

[5.1 Definition 14](#_Toc98600142)

[7. Appendix Tools and Frameworks 15](#_Toc98600143)

[7.1 Isolation frameworks 15](#_Toc98600144)

[7.2 Test frameworks 15](#_Toc98600145)

[**References:** 15](#_Toc98600146)

# 1. Introduction

## 1.1 Overview

This test plan describes the testing approach and overall framework that will drive the testing of TMS V6.0. This document shall be completed and used by the development team to guide how testing will be managed for this project. The test effort will be prioritized and executed based on the project priorities as defined in the Project Plan and Requirements Specification. This is a living document that may be refined as the project progresses. The QA Manager, Test Team Lead, Product Manager, Project Manager, and Development Manager etc. shall review and approve the definitive version of the Test Strategy document.

The document introduces:

## 1.2 Reference Materials

## 1.3 Definitions and Acronyms

* **Ad Hoc Testing**

Is one of the least formal types of tests run on applications since it requires no preparation or planning. Ad hoc testing can be seen as a light version of error guessing (can be done by the people having enough experience on the system to “guess” the most likely source of errors.), which is a light version of exploratory testing.

At an early possible stage, Ad Hoc testing is done randomly, and it is usually an unplanned activity which does not follow and documentation and test design techniques to create test cases.

* **Scenario**

Detailed description (specific instance) of a use case, including rules, exceptions, boundaries, limits, etc. It is also called Test Condition or Test Possibility.

* **Test Case**

A specific set of test data along with expected results for a particular test objective.

* **Test Coverage**

Describes how much of a system has been tested.

Code coverage (test coverage) = Lines of code executed / Total number of lines.

Another coverage metric is called “branch coverage”, which provides more precise results than code coverage because it helps cope with code coverage’s shortcomings.

Branch coverage = Branches traversed / Total number of branches.

* **Test Design**

Describes how a feature or function shall be tested.

* **Test Plan**
* **Test Strategy**

Describes the scope, approach, resources, and schedule for the testing activities of the project. This includes defining what will be tested, who will perform testing, how testing will be managed, and the associated risks and contingencies. Also referred to as a Test Plan.

* **Test Procedure**

Describes the steps for executing a set of test cases and analyzing their results.

* **Test Script**

Step by step description for specific tests.

* **Use Case**

Describes a sequence of interactions between a system and an external actor that results in the actor accomplishing a task that provides benefit to someone. An actor is a person or other entity external to the software system being specified who interacts with the system to accomplish tasks. Different actors often correspond to different user classes, or roles, identified from the customer community that will use the product.

* **MUT**

Method Under Test, normally refers to a method.

* **SUT**

System Under Test, normally refers to the whole class.

* **Mock**

Is a special kind of test double that allows you to examine interactions between the SUT and its collaborators.

Unit Testing Tips

Run tests after each build.

Group by duration or outcome.

Filter test list.

Save filters with playlists.

The isolation issue:

The London take

The Classical take

# 2. Testing Strategy

Diagram

Description automatically generated

The test pyramid has three main layers:

Unit tests at the base

Service tests in the middles

UI tests at the top

## 2.1 Objectives and Tasks

## 2.2 Test Assumptions

## 2.3 Test Principles

## 2.4 Data Approach

## 2.5 Scope and Levels of Testing

### 2.5.1 Scope

Testing will cover the functional testing of the xxxxxx Functionality for this release is detailed in theXXXX Requirements specifications documents. Installation will be tested on the different platforms as described in the Requirements Specification. The testing for this will cover the installation on these platforms, as well as a set of critical functions to determine that the code will work on all platforms.

Limitations and Exclusions

3. Unit Testing

## 3.1 Definition

A unit test is an automated test that:

* Verifies a small piece of code (also known as a unit), or it could be defined as verifies a single unit of behavior.
* Does it quickly.
* In isolation from other tests.

This unit can be an individual function, object, method, procedure, or module in the software under test.

Writing unit tests to test the individual units makes writing comprehensive tests easier as all the units are put together. It is done as the first level of testing during software development.

Graphical user interface, text, application, website

Description automatically generated

A good unit test should be:

* Automated and repeatable.
* Easy to implement.
* Relevant tomorrow.
* Could be run at the push of a button by anyone.
* Run quickly.
* Consistent in its results.
* Have full control of the unit under test.
* Fully isolated runs independently of other tests.
* Easy to detect what was expected and determine how to pinpoint the problem when it fails.

A good unit test has the following four attributes:

* Protection against regressions
* Resistance to refactoring
* Fast feedback
* Maintainability

## 3.2 Popular Frameworks and Why xUnit

There are some most popular Unit Test Frameworks as below:

* [**xUnit**](https://xunit.net/)**:**-- Used mostly by .Net developers, evolved rapidly. ‘x’ stands for the programming language, e.g., JUnit, NUnit, etc.  
  -- Annotation: [Fact] etc.
* **MSTest:**-- Default test framework of MS Studio.  
  -- Annotation: [TestMethod], [TestClass], [TestInitialize] etc.
* **NUnit:**-- (2006) Version .Net of JUnit. Oldest.  
  -- Annotation: [Test], [TestFeature], [SetUp] etc.

Graphical user interface, text, application, email

Description automatically generated

Table

Description automatically generated

The xUnit is chosen for our project based on the following reasons:

* **Better Isolation:** xUnit framework provides much better isolation of tests in comparison to NUnit and MSTest frameworks. Each test gets a new instance of the test class.
* **Simplified Attribute:** xUnit framework is more extensible since it makes use of [Fact] and [Theory] attributes. Many attributes that were present in NUnit framework e.g. [TestFixture], [TestFixtureSetup], [TestFixtureTearDown] [ClassCleanup], [ClassInitialize], [TestCleanup], etc. are not included in the xUnit framework. Separate setup and teardown are not needed: setup with the constructor and teardown with idisposable.
* **Better Assertion mechanism**: xUnit framework makes use of Assert. Throws instead of [ExpectedException] which is used in NUnit and MSTest. On the other hand, Assert.Throws raises assert even if the exception is generic. This ensures that assert is raised even after the exception is raised.
* **Use test collections to run tests in parallel**. By default, each test class is a unique test collection.

## 3.3 Installation of xUnit

The xUnit framework could be installed via NuGet packages:

* **xunit** (latest version 2.4.1)

https://www.nuget.org/packages/xunit

* **xunit.runner.visualstudio** (latest version 2.4.3)

<https://www.nuget.org/packages/xunit.runner.visualstudio/>

## 3.4 Three Styles of Unit Testing

### 3.4.1 Output-Based Style

This style of unit testing is also known as functional. We feed an input to the SUT and check the output if produces. The only component to verify is its return value.

A picture containing diagram

Description automatically generated

### 3.4.2 State-Based Style

This tyle is about verifying the state of the system after an operation is complete.

A picture containing diagram

Description automatically generated

### 3.4.3 Communication-Based Style

This style uses mocks to verify communications between the system under test and its collaborators.

Diagram

Description automatically generated

## 3.5 Naming Conventions

### 3.5.1 Naming a Test Project

* **Unit test project:**

If the unit tests are one part of the solution, then name this project [ProjectBeingTested].Tests. If there are other types of tests as well, then prefix becomes “[ProjectBeingTested].UnitTests”.

* **Integration test project:**

if we have an integration test project, the prefix “.IntegrationTests” should be added, like [ProjectBeingTested].IntegrationTests.

* **UI test project**:

similar to above, add “.UITests”, like [ProjectBeingTested].UITests.

### 3.5.2 Naming a Test Class

Test class naming convention follows the naming convention of the class being tested, ex: the class under test is “Order” the corresponding test class will be “OrderTests“.

Text

Description automatically generated

### 3.5.3 Naming a Unit Test Method

Test name should:

* Express a specific requirement.
* Could include the expected input or state and the expected result for that input or state.
* Be presented as a statement or fact that expresses workflows and outputs.
* Could include the name of the tested method or class.

Here are some basic rules for naming tests:

* No rigid naming policy. Allow freedom of expression.
* Name the test as describing the scenario to a non-programmer person who is familiar with the problem domain.
* Separate words by underscores.

There are some popular unit tests naming conventions that are found to be used by majority of developers:

* **[MUT]\_[Scenario]\_[ExpectedBehavior]**:

“MethodUnderTest” is the name of the method you are testing. “Scenario” is the condition under which you test the method. “ExpectedBehavior” is what you expect the method under test to do in the current scenario.

For example:

|  |
| --- |
| “IsAdult\_AgeLessThan18\_False” |
| “WithdrawMoney\_InvalidAccount\_ExceptionThrown” |
| “AdmitStudent\_MissingMandatoryFields\_FailToAdmit” |

* **[MUT]\_[ ExpectedResult]\_[ Scenario]**:

This technique and the above have disadvantage that if method names get changes, then the test name should also change, or it becomes difficult to comprehend at a later stage. Following is how tests in first example would read like if named using this technique:

For example:

|  |
| --- |
| “IsAdult\_False\_AgeLessThan18” |
| “WithdrawMoney\_ThrowsException\_IfAccountIsInvalid” |
| “AdmitStudent\_FailToAdmit\_IfMandatoryFieldsAreMissing” |

* **Test\_[Feature being tested]**:

This one makes it easy to read the test as the feature to be tested is written as part of test name. Although, there are arguments that the “test” prefix is redundant. However, some sections of developer love to use this technique. Following is how the above tests would read like if named using this technique:

For example:

|  |
| --- |
| “Test\_IsNotAnAdultIfAgeLessThan18” |
| “Test\_FailToWithdrawMoneyIfAccountIsInvalid” |
| “Test\_StudentIsNotAdmittedIfMandatoryFieldsAreMissing” |

* **[Feature to be tested]**:

The above could be simplified because one is anyway using annotations to identify method as test methods.

For example:

|  |
| --- |
| “IsNotAnAdultIfAgeLessThan18” |
| “FailToWithdrawMoneyIfAccountIsInvalid” |
| “StudentIsNotAdmittedIfMandatoryFieldsAreMissing” |

* **Should\_[ExpectedBehavior]\_When\_[Scenario]**:

It makes it easy to read the tests.

For example:

|  |
| --- |
| “Should\_ThrowException\_When\_AgeLessThan18” |
| “Should\_FailToWithdrawMoney\_ForInvalidAccount” |
| “Should\_FailToAdmit\_IfMandatoryFieldsAreMissing” |

* **When\_[ Scenario]\_Expect\_[ExpectedBehavior]**:

For example:

|  |
| --- |
| “When\_AgeLessThan18\_Expect\_IsAdultAsFalse” |
| “When\_InvalidAccount\_Expect\_WithdrawMoneyToFail” |
| “When\_MandatoryFieldsAreMissing\_Expect\_StudentAdmissionToFail” |

* **Given\_[Preconditions]\_When\_[ Scenario]\_Then\_[ExpectedBehavior]**:

This approach is based on naming convention developed as part of Behavior-Driven Development (BDD). The idea is to break down the tests into three part such that one could come up with preconditions, state under test and expected behavior to be written in above format.

For example:

|  |
| --- |
| “Given\_UserIsAuthenticated\_When\_InvalidAccountNumberIsUsedToWithdrawMoney\_Then\_TransactionsWillFail” |

In conclusion, among above techniques, the recommended formats are:

**[MUT]\_[Scenario]\_[ExpectedBehavior]**

**[Feature to be tested]**

## 3.6 Test Structure

The AAA pattern advocates for splitting each test into three parts: **Arrange - Act - Assert**. It provides a simple, uniform structure for all tests in the suite, and they answer three questions of “Given – When - Then”.

Graphical user interface, text, application

Description automatically generated

* **Arrange:** set up the test date, bring the system under test (SUT) and its dependencies to a desired state.
* **Act**: call methods on the SUT, pass the prepared dependencies, and capture the output value (if any).
* **Assert**: verify the outcome. The outcome may be represented by the return value, the final state of the SUT and its collaborators, or the methods the SUT called on those collaborators.

One way to do that is to put // Arrange, // Act, and // Assert comments before the beginning of each section. Another way is to separate the sections with empty lines, as shown next.

Text

Description automatically generated with medium confidence

Principles when writing the three sections:

* Avoid multiple arrange, act, and assert sections:

Such a test is no longer a unit test rather is an integration test.

* Avoid if statements in tests:

an if statement indicates that the test verified too many things at once.

## 3.7 xUnit Attributes

### 3.7.1 [Fact]

### 3.7.2 [Theory]

### 3.7.3 [InlineData]

### 3.7.4 [MemberData]

## 3.7 Stubs

## 3.8 Mocking with MOQ & xUnit

And stub…

## 3.9 Live Unit Testing Approch

# 4. System and Integration Testing

Diagram

Description automatically generated

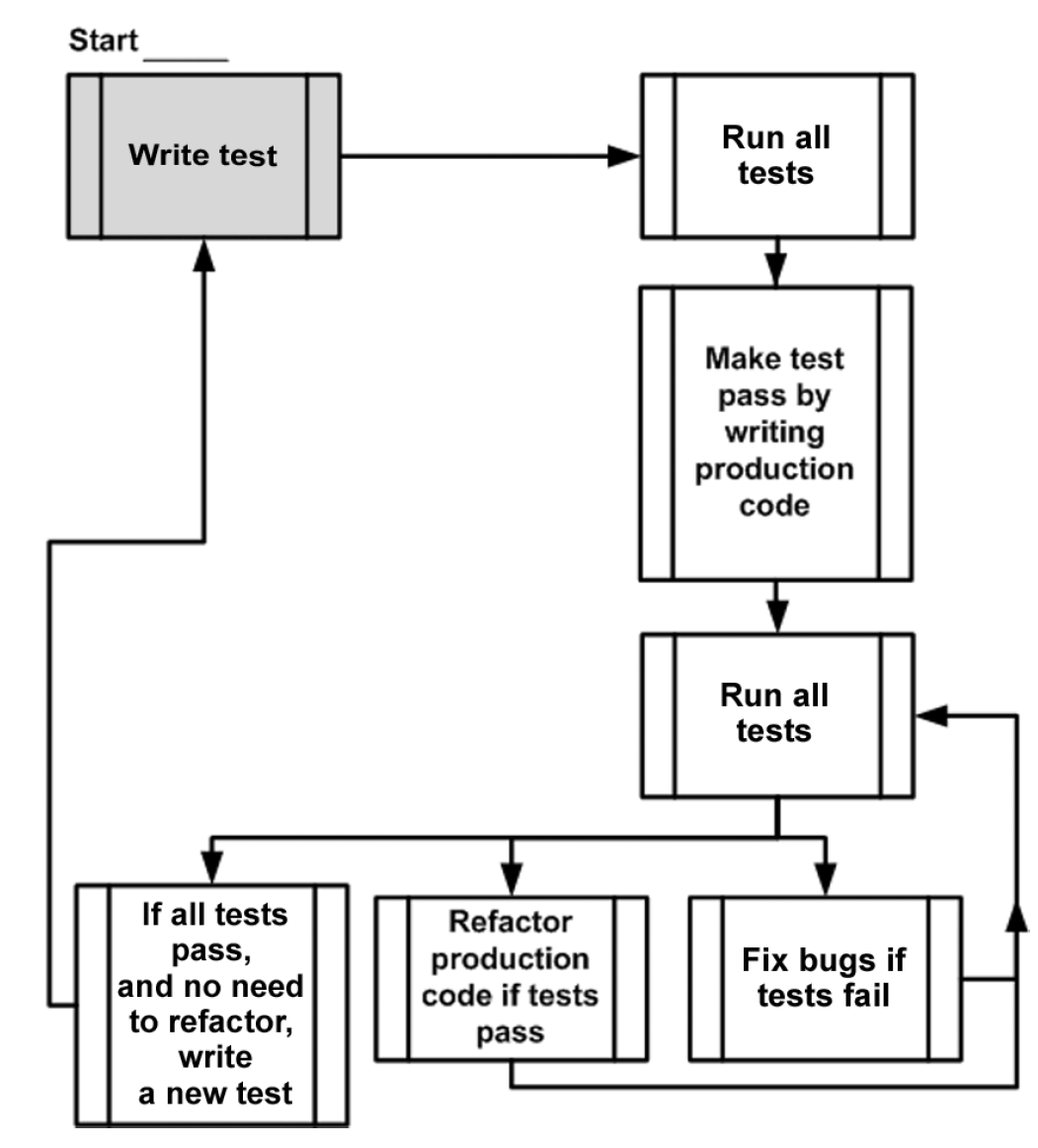
Integration tests cover controllers, while unit tests cover the domain model and algorithms< Trival and overcomplicated code shouldn’t be tested at all.

Performance and Stress Testing  
User Acceptance Testing  
Batch Testing  
Automated Regression Testing  
Beta Testing

## 4.1 Definition

# 5. Data Base Testing

## 5.1 Definition

6. Test Driven Development

Icon

Description automatically generated

An example

Test-driven development is a software development process that relies on tests to

drive the project development. The process consists of three (some authors specify

four) stages, which you repeat for every test case:

Write a failing test to indicate which functionality needs to be added and how

it should behave.

Write just enough code to make the test pass. At this stage, the code doesn’t

have to be elegant or clean.

Refactor the code. Under the protection of the passing test, you can safely

clean up the code to make it more readable and maintainable.

7. Appendix Tools and Frameworks

## 7.1 Isolation frameworks

* Dependencies  
  -- Xunit  
  -- MOQ  
  -- FakeitEasy?

## 7.2 Test frameworks

**References:**

[Xunit.net Doc](https://xunit.net/docs/getting-started/netcore/cmdline)   
[Live Unit Testing Doc](https://docs.microsoft.com/en-us/visualstudio/test/live-unit-testing-intro?view=vs-2022)