PURPOSE OF THE UNIT STANDARD

- Demonstrating an understanding of evaluating requirements for test design.
- Explaining test design methodology and techniques.
- Applying requirement-based test design techniques.
- Using state models.
- Using cases for discovering and testing requirements.
Learning Outcomes

- Functional requirements are identified and explained in terms of test design requirements.
- Quality requirements are identified and explained in terms of outcomes required.
- Test design requirements are documented according to organisational requirements.
- Software requirements are evaluated to assess completeness, precision and testability.
- Strategies for designing tests are explained with examples.
Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test.[1] Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects) and verifying that the software product is fit for use.

Software testing involves the execution of a software component or system component to evaluate one or more properties of interest. In general, these properties indicate the extent to which the component or system under test

- meets the requirements that guided its design and development,
- responds correctly to all kinds of inputs,
- performs its functions within an acceptable time,
- is sufficiently usable,
- can be installed and run in its intended environments, and
- achieves the general result its stakeholders desire.

As the number of possible tests for even simple software components is practically infinite, all software testing uses some strategy to select tests that are feasible for the available time and resources. As a result, software testing typically (but not exclusively) attempts to execute a program or application with the intent of finding software bugs (errors or other defects). The job of testing is an iterative process as when one bug is fixed, it can illuminate other, deeper bugs, or can even create new ones.

Software testing can provide objective, independent information about the quality of software and risk of its failure to users or sponsors.

Software testing can be conducted as soon as executable software (even if partially complete) exists. The overall approach to software development often determines when and how testing is conducted. For example, in a phased process, most testing occurs after system requirements have been defined and then implemented in testable programs. In contrast, under an agile approach, requirements, programming, and testing are often done concurrently.

Software Quality
SOFTWARE QUALITY is the degree of conformance to explicit or implicit requirements and expectations.

Explanation

- **Explicit:** clearly defined and documented
- **Implicit:** not clearly defined and documented but indirectly suggested
- **Requirements:** business/product/software requirements
- **Expectations:** mainly end-user expectations

Note: Some people tend to think quality as compliance to only explicit requirements and not implicit requirements. We tend to think of such people as lazy.

Quality: The degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations.

Software quality: The totality of functionality and features of a software product that bear on its ability to satisfy stated or implied needs.

As with any definition, the definition of ‘software quality’ is also varied and debatable. Some even say that ‘quality’ cannot be defined and some say that it can be defined but only in a particular context. Some even state confidently that ‘quality is lack of bugs. Whatever the definition, it is true that quality is something we all aspire to.

Software quality has many dimensions. See Dimensions of Quality.

In order to ensure software quality, we undertake Software Quality Assurance and Software Quality Control.
Test design

Definition

Test design could require all or one of:

• knowledge of the software, and the business area it operates on,

• knowledge of the functionality being tested,

• knowledge of testing techniques and heuristics.

planning skills to schedule in which order the test cases should be designed, given the effort, time and cost needed or the consequences for the most important and/or risky features.

Well-designed test suites will provide for an efficient testing. The test suite will have just enough test cases to test the system, but no more. This way, there is no time lost in writing redundant test cases that would unnecessarily consume time each time they are executed. In addition, the test suite will not contain brittle or ambiguous test cases.

Automatic test design

Entire test suites or test cases exposing real bugs can be automatically generated by software using model checking or symbolic execution. Model checking can ensure all the paths of a simple program are exercised, while symbolic execution can detect bugs and generate a test case that will expose the bug when the software is run using this test case.

However, as good as automatic test design can be, it is not appropriate for all circumstances. If the complexity becomes too high, then human test design must come into play as it is far more flexible, and it can concentrate on generating higher level test suites.

Introduction to Testing

• Testing is a set of activities which are decided in advance i.e before the start of development and organized systematically.

• In the literature of software engineering various testing strategies to implement the testing are defined.

• All the strategies give a testing template.

Following are the characteristic that process the testing templates:
• The developer should conduct the successful technical reviews to perform the testing successful.
• Testing starts with the component level and work from outside toward the integration of the whole computer-based system.
• Different testing techniques are suitable at different point in time.
• Testing is organized by the developer of the software and by an independent test group.
• Debugging and testing are different activities, then also the debugging should be accommodated in any strategy of testing.
• Difference between Verification and Validation
<table>
<thead>
<tr>
<th>Verification</th>
<th>Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled.</td>
<td>Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled.</td>
</tr>
<tr>
<td>It estimates an intermediate product.</td>
<td>It estimates the final product.</td>
</tr>
<tr>
<td>It describes whether the outputs are as per the inputs or not.</td>
<td>It explains whether they are accepted by the user or not.</td>
</tr>
<tr>
<td>Verification is done before the validation.</td>
<td>It is done after the verification.</td>
</tr>
<tr>
<td>Plans, requirement, specification, code is evaluated during the verifications.</td>
<td>Actual product or software is tested under validation.</td>
</tr>
<tr>
<td>It manually checks the files and document.</td>
<td>It is a computer software or developed program based checking of files and document.</td>
</tr>
</tbody>
</table>
Testing is organized by the developer of the software and by an independent test group.

Debugging and testing are different activities, then also the debugging should be accommodated in any strategy of testing.

Difference between Verification and Validation

Strategy of testing

A strategy of software testing is shown in the context of spiral.

**Fig. - Testing Strategy**

**Unit testing**
Unit testing starts at the centre and each unit is implemented in source code.

**Integration testing**
An integration testing focuses on the construction and design of the software.

**Validation testing**
Check all the requirements like functional, behavioral and performance requirement are validated against the construction software.

**System testing**
System testing confirms all system elements and performance are tested entirely.
Testing strategy for procedural point of view

As per the procedural point of view the testing includes following steps.

1) Unit testing
2) Integration testing
3) High-order tests
4) Validation testing

These steps are shown in following figure:

![Diagram showing testing steps](image-url)
Learning Outcomes

- Test design requirements are described with examples.
- Test scenarios are identified and explained in terms of model-based requirements.
- Test cases are developed and prioritized according to test procedure specifications.
- The scripts are created and specified in terms of expected results and execution conditions.
Black Box Testing

BLACK BOX TESTING, also known as Behavioral Testing, is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester. These tests can be functional or non-functional, though usually functional.

This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see. This method attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors

**black box testing**: Testing, either functional or non-functional, without reference to the internal structure of the component or system.

**black box test design technique**: Procedure to derive and/or select test cases based on an analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.

**Example**

A tester, without knowledge of the internal structures of a website, tests the web pages by using a browser; providing inputs (clicks, keystrokes) and verifying the outputs against the expected outcome.
Levels Applicable To

Black Box Testing method is applicable to the following levels of software testing:

- Integration Testing
- System Testing
- Acceptance Testing

The higher the level, and hence the bigger and more complex the box, the more black box testing method comes into use.

Techniques

Following are some techniques that can be used for designing black box tests.

- **Equivalence Partitioning**: A black-box test technique in which test cases are designed to exercise equivalence partitions by using one representative member of each partition.

- **Boundary Value Analysis**: A black-box test technique in which test cases are designed based on boundary values.

- **Cause-Effect Graphing**: It is a software test design technique that involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph, and generating test cases accordingly.

Advantages

- Tests are done from a user’s point of view and will help in exposing discrepancies in the specifications.

- Tester need not know programming languages or how the software has been implemented.

- Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias.

- Test cases can be designed as soon as the specifications are complete.

Disadvantages

- Only a small number of possible inputs can be tested, and many program paths will be left untested.

- Without clear specifications, which is the situation in many projects, test cases will be difficult to design.
• Tests can be redundant if the software designer/developer has already run a test case.

• Ever wondered why a soothsayer closes the eyes when foretelling events? So is almost the case in Black Box Testing.

White Box Testing

WHITE BOX TESTING (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester. The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential. White box testing is testing beyond the user interface and into the nitty-gritty of a system.

This method is named so because the software program, in the eyes of the tester, is like a white/transparent box; inside which one clearly sees.

**white-box testing**: Testing based on an analysis of the internal structure of the component or system.

**white-box test design technique**: Procedure to derive and/or select test cases based on an analysis of the internal structure of a component or system.

**Example**

A tester, usually a developer as well, studies the implementation code of a certain field on a webpage, determines all legal (valid and invalid) AND illegal inputs and verifies the outputs against the expected outcomes, which is also determined by studying the implementation code.

White Box Testing is like the work of a mechanic who examines the engine to see why the car is not moving.
Levels Applicable To

White Box Testing method is applicable to the following levels of software testing:

- **Unit Testing**: For testing paths within a unit.

- **Integration Testing**: For testing paths between units.

- **System Testing**: For testing paths between subsystems.

However, it is mainly applied to Unit Testing.

**Advantages**

- Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.

- Testing is more thorough, with the possibility of covering most paths.

**Disadvantages**

- Since tests can be very complex, highly skilled resources are required, with a thorough knowledge of programming and implementation.

- Test script maintenance can be a burden if the implementation changes too frequently.

- Since this method of testing is closely tied to the application being tested, tools to cater to every kind of implementation/platform may not be readily available.

**Gray Box Testing**

GRAY BOX TESTING is a software testing method which is a combination of Black Box Testing method and White Box Testing method. In Black Box Testing, the internal structure of the item being tested is unknown to the tester and in White Box Testing the internal structure is known. In Gray Box Testing, the internal structure is partially known. This involves having access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level.

Gray Box Testing is named so because the software program, in the eyes of the tester is like a gray/semi-transparent box; inside which one can partially see.
Example
An example of Gray Box Testing would be when the codes for two units/modules are studied (White Box Testing method) for designing test cases and actual tests are conducted using the exposed interfaces (Black Box Testing method).

Levels Applicable To
Though Gray Box Testing method may be used in other levels of testing, it is primarily used in Integration Testing.

Agile Testing
AGILE TESTING is a method of software testing that follows the principles of agile software development.

This article on Agile Testing assumes that you already understand Agile software development methodology (Scrum, Extreme Programming, or other flavors of Agile).

Agile Testing Manifesto
This is adapted from agilemanifesto.org and it might look a little silly to copy everything from there and just replace the term “development” with “test/ing” but here it is for your refreshment.

We are uncovering better ways of testing

software by doing it and helping others do it.

Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.
Agile Testing Values

Individuals and interactions over processes and tools: This means that flexible people and communication are valued over rigid processes and tools. However, this does not mean that agile testing ignores processes and tools. In fact, agile testing is built upon very simple, strong and reasonable processes like the process of conducting the daily meeting or preparing the daily build. Similarly, agile testing attempts to leverage tools, especially for test automation, as much as possible. Nevertheless, it needs to be clearly understood that it is the testers who drive those tools and the output of the tools depend on the testers (not the other way around).

Working software over comprehensive documentation: This means that functional and usable software is valued over comprehensive but unusable documentation. Though this is more directed to upfront requirement specifications and design specifications, this can be true for test plans and test cases as well. Our primary goal is the act of testing itself and not any elaborate documentation merely pointing toward that goal. However, it is always best to have necessary documentation in place so that the ‘picture ’is clear, and the ‘picture ’remains with the team if/when a member leaves.

Customer collaboration over contract negotiation: This means that the client is engaged frequently and is closely in touch with the progress of the project (not through complicated progress reports but through working pieces of software). This does put some extra burden on the customer who has to collaborate with the team at regular intervals (instead of just waiting till the end of the contract, hoping that deliveries will be made as promised). But this frequent engagement ensures that the project is heading in the right direction and not toward the building of a frog when a fish is expected.

Responding to change over following a plan: This means accepting changes as being natural and responding to them without being afraid of them. It is always nice to have a plan beforehand, but it is not very nice to stick to a plan, at whatever the cost, even when situations have changed. Let’s say you write a test case, which is your plan, assuming a certain requirement. Now, if the requirement changes, you do not lament over the wastage of your time and effort. Instead, you promptly adjust your test case to validate the changed requirement. And, of course, only a FOOL would try to run the same old test case on the new software and mark the test as FAIL.

Principles Behind Agile Testing Manifesto

Behind the Agile Testing Manifesto are the following principles which some agile practitioners, unfortunately, fail to understand or implement. We urge you to go through each principle and digest them thoroughly if you intend to embrace Agile Testing. On the right column, the original principles have been re-written specifically for software testers.
<table>
<thead>
<tr>
<th>We follow these principles:</th>
<th>What it means for Software Testers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our highest priority is to satisfy the customer through the early and continuous delivery of valuable software.</td>
<td>Our highest priority is to satisfy the customer through the early and continuous delivery of high-quality software.</td>
</tr>
<tr>
<td>Welcome changing requirements, even late in testing. Agile processes harness change for the customer’s competitive advantage.</td>
<td>Welcome changing requirements, even late in testing. Agile processes harness change for the customer’s competitive advantage.</td>
</tr>
<tr>
<td>Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.</td>
<td>Deliver high-quality software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.</td>
</tr>
<tr>
<td>Business people and developers must work together daily throughout the project.</td>
<td>Business people, developers, and testers must work together daily throughout the project.</td>
</tr>
</tbody>
</table>
Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

Importance of Testing Methodologies

As software applications get ever more complex and intertwined and with the large number of different platforms and devices that need to get tested, it is more important than ever to have a robust testing methodology for making sure that software products/systems being developed have been fully tested to make sure they meet their specified requirements and can successfully operate in all the anticipated environments with the required usability and security.

<table>
<thead>
<tr>
<th>Functional Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Testing</td>
</tr>
<tr>
<td>Integration Testing</td>
</tr>
<tr>
<td>System Testing</td>
</tr>
<tr>
<td>Acceptance Testing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-Functional Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Testing</td>
</tr>
<tr>
<td>Security Testing</td>
</tr>
<tr>
<td>Usability Testing</td>
</tr>
<tr>
<td>Compatibility Testing</td>
</tr>
</tbody>
</table>

This page describes the various components of a thorough testing methodology and illustrates how Spira Test is best suited to help you implement and manage them on your projects.

It is a complete solution that includes requirements management, test, release management and defect tracking all fully integrated from day one.

Highly intuitive web application that provides a complete picture of a project’s status and health yet requires only a web-browser.

Ability to leverage your existing technology investments.

Functional Testing
The functional testing part of a testing methodology is typically broken down into four components - unit testing, integration testing, system testing and acceptance testing – usually executed in this order. Each of them is described below:

**Unit Testing**

The Unit testing part of a testing methodology is the testing of individual software modules or components that make up an application or system. These tests are usually written by the developers of the module and in a test-driven-development methodology (such as Agile, Scrum or XP) they are actually written before the module is created as part of the specification. Each module function is tested by a specific unit test fixture written in the same programming language as the module.

```
public class SampleTestFixture {
    [SetUp]
    public void Init()
    {
        //Do Nothing
    }

    /// <summary>
    /// Sample test that asserts a failure
    /// </summary>
    [Test, 
    SpiraTestCase (<TEST_CASE_ID>)]
    public void _01_SampleFailure()
```

includes support for the management and execution of automated unit tests. With its library of plugins and extensions for all of the major unit test frameworks, Spira Test allows the test manager to ensure that there is full coverage of all program code and that all unit tests have passed.

**Integration Testing**

A test level that focuses on interactions between components or systems. The Integration testing part of a testing methodology is the testing of the different modules/components that have been successfully unit tested when integrated together to perform specific tasks and activities (also known as scenario testing). This testing is usually done with a combination of automated functional tests and manual testing depending on how easy it is to create automated tests for specific integrated components.

Spira Test includes support for storing, managing and coordinating integration tests across various modules and components. With its library of plugins and extensions for different automated functional
testing tools including our Rapise automation platform, Spira Test is the perfect solution for managing your integration testing.

**System Testing**

A test level that focuses on verifying that a system as a whole meets specified requirements. The system testing part of a testing methodology involves testing the entire system for errors and bugs. This test is carried out by interfacing the hardware and software components of the entire system (that have been previously unit tested and integration tested), and then testing it as a whole.

9This testing is listed under the black-box testing method, where the software is checked for user-expected working conditions as well as potential exception and edge conditions.

SpiraTest includes support for storing, managing and coordinating system tests across all the modules and components that make up a system. SpiraTest support data-driven testing where test cases are defined with input parameters and different combinations of test data can be passed to the manual and automated tests. This ensures that both expected, and exception cases can be tested using the same test frameworks.

**Acceptance Testing**

A test level that focuses on determining whether to accept the system. The acceptance testing part of a testing methodology is the final phase of functional software testing and involves making sure that all the product/project requirements have been met and that the end-users and customers have tested the system to make sure it operates as expected and meets all their defined requirements:
SpiraTest provides a powerful manual testing solution that helps you coordinate and manage your acceptance testing activities, with all the test scripts, assignments, test results and associated defects/bugs all tracked in a single unified system. With SpiraTest’s requirements test coverage capabilities you can validate that all of your requirements have been fully tested to your customer’s satisfaction.

**Non-Functional Testing**

Testing performed to evaluate that a component or system complies with non-functional requirements. In most testing methodologies, functional testing involves testing the application against the business requirements. Functional testing is done using the functional specifications provided by the client or by using the design specifications like use cases provided by the design team.

On the other hand, non-functional testing involves testing the application against the non-functional requirements, which typically involve measuring/testing the application against defined technical qualities, for example: vulnerability, scalability, usability. Some examples of non-functional testing are described below:

**Performance, Load, Stress Testing**

There are several different types of performance testing in most testing methodologies, for example: performance testing is measuring how a system behaves under an increasing load (both numbers of users and data volumes), load testing is verifying that the system can operate at the required response times when subjected to its expected load, and stress testing is finding the failure point(s) in the system when the tested load exceeds that which it can support.
SpiraTest includes support for storing, managing and coordinating your performance, load and stress testing activities. With its library of plugins and extensions for different automated performance testing tools including LoadRunner, JMeter and NeoLoad, SpiraTest is the perfect solution for managing your performance testing.

**Security, Vulnerability Testing**

Previously, security was something that was tested after-the-fact. With the rise in cyber-crime and the awareness of the risks associated with software vulnerabilities, application security is now something that needs to be designed and developed at the same time as the desired business functionality. Security testing tests the software for confidentiality, integrity, authentication, availability, and non-repudiation. Individual tests are conducted to prevent any unauthorized access to the software code.

**Usability Testing**

Testing to evaluate the degree to which the system can be used by specified users with effectiveness, efficiency and satisfaction in a specified context of use. The usability testing part of a testing methodology looks at the end-user usability aspect of the software. The ease with which a user can access the product forms the main testing point. Usability testing looks at five aspects of testing, - learnability, efficiency, satisfaction, memorability, and errors.

**Compatibility Testing**

The compatibility part of a testing methodology tests that the product or application is compatible with all the specified operating systems, hardware platforms, web browsers, mobile devices, and other designed third-party programs. Compatibility tests check that the product works as expected across all the different hardware/software combinations and that all functionality is consistently supported.
SpiraTest makes the managing and tracking of your cross-platform testing easy, it enabled you to quickly configure different test plans for the different hardware/software combinations and make sure that all of the requirements and test cases have been executed (and passed) against all of the specified combinations and configurations.

Learning Outcomes

- Equivalence class partitioning is identified to meet test requirements.
- Boundary value analysis is carried out by applying the value rule and principles.
- A case-effect graph is developed outlining the cases and effects.
- Decision tables are explained in terms of their application.
- Decision tables are constructed showing test scenarios and decision trees

**Equivalence partitioning:**

Dividing the test input data into a range of values and selecting one input value from each range is called Equivalence Partitioning. This is a black box test design technique used to calculate the effectiveness of test cases and which can be applied to all levels of testing from unit, integration, system testing and so forth.

We cannot test all the possible input domain values, because if we attempted this, the number of test cases would be too large. In this method, input data is divided into different classes, each class representing the input criteria from the equivalence class. We then select one input from each class.

This technique is used to reduce an infinite number of test cases to a finite number, while ensuring that the selected test cases are still effective test cases which will cover all possible scenarios.

Let’s take a very basic and simple example to understand the Equivalence Partitioning concept:

- If one application is accepting input range from 1 to 100, using equivalence class we can divide inputs into the classes, for example, one for valid input and another for invalid input and design one test case from each class.
In this example test cases are chosen as below:

TRY REQTEST FREE - #1 BUG TRACKING TOOL

One is for valid input class i.e. selects any value from input between ranges 1 to 100. So here we are not writing hundreds of test cases for each value. Any one value from this equivalence class should give you the same result.

One is for invalid data below lower limit i.e. any one value below 1.

One is for invalid data above upper limit i.e. any value above 100.

Boundary value analysis:

• For the most part, errors are observed in the extreme ends of the input values, so these extreme values like start/end or lower/upper values are called Boundary values and analysis of these Boundary values is called “Boundary value analysis”. It is also sometimes known as ‘range checking’.

• Boundary value analysis is another black box test design technique and it is used to find the errors at boundaries of input domain rather than finding those errors in the centre of input.

• Equivalence Partitioning and Boundary value analysis are linked to each other and can be used together at all levels of testing. Based on the edges of the equivalence classes, test cases can then be derived.

• Each boundary has a valid boundary value and an invalid boundary value. Test cases are designed based on the both valid and invalid boundary values. Typically, we choose one test case from each boundary.

Finding defects using Boundary value analysis test design technique is very effective and it can be used at all test levels. You can select multiple test cases from valid and invalid input domains based on your needs or previous experience but remember you do have to select at least one test case from each input domain.

TRY REQTEST FREE FOR TEST MANAGEMENT

Let’s take same above example to understand the Boundary value analysis concept:

• One test case for exact boundary values of input domains each means 1 and 100.

• One test case for just below boundary value of input domains each means 0 and 99.

• One test case for just above boundary values of input domains each means 2 and 101.
Test case writing techniques for dynamic testing

- Equivalence partitioning
- Boundary Value Analysis
- Decision table
- Cause and effect graph technique
- State transition diagram
- Orthogonal array testing (OATS)
- Error guessing.

We have some really good papers 1st, 2nd and 3rd points (Equivalence portioning, BVA and decision tables) here in STH. I am going to discuss point 4 which is Cause and effect graph.

Introduction

Cause and effect graph are a dynamic test case writing technique. Here causes are the input conditions and effects are the results of those input conditions.

Cause-Effect Graphing is a technique which starts with a set of requirements and determines the minimum possible test cases for maximum test coverage which reduces test execution time and ultimately cost.

The goal is to reduce the total number of test cases still achieving the desired application quality by covering the necessary test cases for maximum coverage.

But at the same time obviously, there are some downsides of using this test case writing technique. It takes time to model all your requirements into this cause-effect graph before writing test cases.

The Cause-Effect graph technique restates the requirements specification in terms of the logical relationship between the input and output conditions. Since it is logical, it is obvious to use Boolean operators like AND, OR and NOT.

Notations we are going to use:
AND – For effect E1 to be true, both the causes C1 and C2 should be true

OR – For effect E1 to be true, either of causes C1 OR C2 should be true

NOT – For Effect E1 to be True, Cause C1 should be false

MUTUALLY EXCLUSIVE – When only one of the causes will hold true.

Now let’s try to implement this technique with some example.

1. Draw a cause and effect graph based on a requirement/situation
2. Cause and Effect graph is given, draw a decision table based on it to draw the test case.

Let’s see both of them one by one.

Let’s draw a cause and effect graph based on a situation
Situation:

The “Print message” is software that read two characters and, depending on their values, messages must be printed.

The first character must be an “A” or a “B”.

The second character must be a digit.

If the first character is an “A” or “B” and the second character is a digit, the file must be updated.

If the first character is incorrect (not an “A” or “B”), the message X must be printed.

If the second character is incorrect (not a digit), the message Y must be printed.

Solution:

The causes for this situation are:

C1 – First character is A
C2 – First character is B
C3 – the Second character is a digit

The effects (results) for this situation are

E1 – Update the file
E2 – Print message “X”
E3 – Print message “Y”

First, draw the causes and effects as shown below:
Key – Always go from effect to cause (left to right). That means, to get effect “E”, what causes should be true.

In this example, let’s start with Effect E1.

Effect E1 is to update the file. The file is updated when

– The first character is “A” and the second character is a digit
– The first character is “B” and the second character is a digit
– The first character can either be “A” or “B” and cannot be both.

Now let’s put these 3 points in symbolic form:

For E1 to be true – following are the causes:

– C1 and C3 should be true
– C2 and C3 should be true
– C1 and C2 cannot be true together. This means C1 and C2 are mutually exclusive.

Now let’s draw this:
So as per the above diagram, for E1 to be true the condition is 

\[(C1 \lor C2) \land C3\]

The circle in the middle is just an interpretation of the middle point to make the graph less messy. There is a third condition where C1 and C2 are mutually exclusive. So the final graph for effect E1 to be true is shown below:

So as per the above diagram, for E1 to be true the condition is

\[(C1 \land C2) \lor C3\]
The circle in the middle is just an interpretation of the middle point to make the graph less messy.

There is a third condition where C1 and C2 are mutually exclusive. So, the final graph for effect E1 to be true is shown below:

For Effect E3.
E3 states to print message “Y”. Message Y will be printed when Second character is incorrect.
Which means Effect E3 will hold true when C3 is invalid. So, the graph for Effect E3 is shown as (In Green line)

This completes the Cause and Effect graph for the above situation.
Move to draw the Decision table based on the above graph.

**Writing Decision table based on Cause and Effect graph**

First, write down the Causes and Effects in a single column shown below

<table>
<thead>
<tr>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>C3</td>
</tr>
<tr>
<td>E1</td>
</tr>
<tr>
<td>E2</td>
</tr>
<tr>
<td>E3</td>
</tr>
</tbody>
</table>

Key is the same. Go from bottom to top which means traverse from effect to cause.

Start with Effect E1. For E1 to be true, the condition is \((C1 \lor C2) \land C3\).

Here we are representing True as 1 and False as 0

First, put Effect E1 as True in the next column as

<table>
<thead>
<tr>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>C3</td>
</tr>
<tr>
<td>E1</td>
</tr>
<tr>
<td>E2</td>
</tr>
<tr>
<td>E3</td>
</tr>
</tbody>
</table>

Now for E1 to be “1” (true), we have the below two conditions –
- C1 AND C3 will be true
- C2 AND C3 will be true

<table>
<thead>
<tr>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>C3</td>
</tr>
<tr>
<td>E1</td>
</tr>
<tr>
<td>E2</td>
</tr>
<tr>
<td>E3</td>
</tr>
</tbody>
</table>

For E2 to be True, either C1 or C2 has to be false shown as
For E3 to be true, C3 should be false.

<table>
<thead>
<tr>
<th>Actions</th>
<th>TC1</th>
<th>TC2</th>
<th>TC3</th>
<th>TC4</th>
<th>TC5</th>
<th>TC6</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So, it’s done. Let’s complete the graph by adding 0 in the blank column and including the test case identifier.

<table>
<thead>
<tr>
<th>Actions</th>
<th>TC1</th>
<th>TC2</th>
<th>TC3</th>
<th>TC4</th>
<th>TC5</th>
<th>TC6</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C2</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C3</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Writing Test cases from the decision table

I am writing a sample test case for test case 1 (TC1) and Test Case 2 (TC2).

<table>
<thead>
<tr>
<th>TC ID</th>
<th>TC Name</th>
<th>Description</th>
<th>Steps</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>TC1_FileUpdate Scenario1</td>
<td>Validate that system updates the file when first character is A and second character is a digit.</td>
<td>1. Open the application. 2. Enter first character as “A” 3. Enter second character as a digit</td>
<td>File is updated.</td>
</tr>
<tr>
<td>TC2</td>
<td>TC2_FileUpdate Scenario2</td>
<td>Validate that system updates the file when first character is B and second character is a digit.</td>
<td>1. Open the application. 2. Enter first character as “B” 3. Enter second character as a digit</td>
<td>File is updated.</td>
</tr>
</tbody>
</table>

In a similar fashion, you can create other test cases.

(A test case contains many other attributes like preconditions, test data, severity, priority, build, version, release, environment etc. I assume all these attributes to be included when you write the test cases in actual situation)
Guidelines for Cause-Effect Functional Testing Technique:

- If the variables refer to physical quantities, domain testing and equivalence class testing are indicated.

- If the variables are independent, domain testing and equivalence class testing are indicated.

- If the variables are dependent, decision table testing is indicated.

- If the single-fault assumption is warranted, boundary value analysis (BVA) and robustness testing are indicated.

- If the multiple-fault assumption is warranted, worst-case testing, robust worst-case testing and decision table testing are identical.

- If the program contains significant exception handling, robustness testing and decision table testing are indicated.

- If the variables refer to logical quantities, equivalence class testing, and decision table testing are indicated.

Decision tables: A table used to show sets of conditions and the actions resulting from them, are a concise visual representation for specifying which actions to perform depending on given conditions. They are algorithms whose output is a set of actions. The information expressed in decision tables could also be represented as decision trees or in a programming language as a series of if-then-else and switch-case statements.

Each decision corresponds to a variable, relation or predicate whose possible values are listed among the condition alternatives. Each action is a procedure or operation to perform, and the entries specify whether (or in what order) the action is to be performed for the set of condition alternatives the entry corresponds to.

To make them more concise, many decision tables include in their condition alternatives a don't care symbol. This can be a hyphen [1][2][3] or blank,[4] although using a blank is discouraged as it may merely indicate that the decision table has not been finished.
One of the uses of decision tables is to reveal conditions under which certain input factors are irrelevant on the actions to be taken, allowing these input tests to be skipped and thereby streamlining decision-making procedures.

Aside from the basic four quadrant structure, decision tables vary widely in the way the condition alternatives and action entries are represented. Some decision tables use simple true/false values to represent the alternatives to a condition, other tables may use numbered alternatives, and some tables even use fuzzy logic or probabilistic representations for condition alternatives. In a similar way, action entries can simply represent whether an action is to be performed, or in more advanced decision tables, the sequencing of actions to perform.
Use state models
Specific Outcome 4

Learning Outcomes

• State models are explained with examples of their use.
• Tests are carried out based on state models.
Model-based testing

Model-based testing is testing based on or involving models. An application of model-based design for designing and optionally also executing artifacts to perform software testing or system testing. Models can be used to represent the desired behavior of a system under test (SUT), or to represent testing strategies and a test environment. The picture on the right depicts the former approach.

A model describing a SUT is usually an abstract, partial presentation of the SUT’s desired behavior. Test cases derived from such a model are functional tests on the same level of abstraction as the model. These test cases are collectively known as an abstract test suite. An abstract test suite cannot be directly executed against an SUT because the suite is on the wrong level of abstraction. An executable test suite needs to be derived from a corresponding abstract test suite.

The executable test suite can communicate directly with the system under test. This is achieved by mapping the abstract test cases to concrete test cases suitable for execution. In some model-based testing environments, models contain enough information to generate executable test suites directly. In others, elements in the abstract test suite must be mapped to specific statements or method calls in the software to create a concrete test suite. This is called solving the “mapping problem”. In the case of online testing, abstract test suites exist only conceptually but not as explicit artifacts.

Tests can be derived from models in different ways. Because testing is usually experimental and based on heuristics, there is no known single best approach for test derivation. It is common to consolidate all test derivation related parameters into a package that is often known as “test requirements”, “test purpose” or even “use case(s)”. This package can contain information about those parts of a model that should be focused on, or the conditions for finishing testing.

Because test suites are derived from models and not from source code, model-based testing is usually seen as one form of black-box testing.
State Transition in Testing

State Transition testing is defined as the testing technique in which changes in input conditions cause's state changes in the Application under Test (AUT). A black-box test technique in which test cases are designed to exercise elements of a state transition model.

It is a black box testing technique in which the tester analyzes the behavior of an application under test for different input conditions in a sequence. In this technique, tester provides both positive and negative input test values and record the system behavior.

It is the model on which the system and the tests are based. Any system where you get a different output for the same input, depending on what has happened before, is a finite state system.

State Transition Testing Technique is helpful where you need to test different system transitions.

Use State Transition

This can be used when a tester is testing the application for a finite set of input values.

When the tester is trying to test sequence of events that occur in the application under test. I.e., this will allow the tester to test the application behavior for a sequence of input values.

When the system under test has a dependency on the events/values in the past.

Not Rely On State Transition

When the testing is not done for sequential input combinations.

If the testing is to be done for different functionalities like exploratory testing

Four Parts Of State Transition Model:

There are 4 main components of the State Transition Model as below

1) States that the software might get

2) Transition from one state to another

3) Events that origin a transition like closing a file or withdrawing money
4) Actions that result from a transition (an error message or being given the cash.)

**State Transition Diagram and State Transition Table**

There are two main ways to represent or design state transition, State transition diagram, and state transition table.

In state transition diagram the states are shown in boxed texts, and the transition is represented by arrows. It is also called State Chart or Graph. It is useful in identifying valid transitions.

In state transition table all the states are listed on the left side, and the events are described on the top. Each cell in the table represents the state of the system after the event has occurred. It is also called State Table. It is useful in identifying invalid transitions.

**How to Make a State Transition (Examples of a State Transition)**

Example 1:

Let's consider an ATM system function where if the user enters the invalid password three times the account will be locked.

In this system, if the user enters a valid password in any of the first three attempts the user will be logged in successfully. If the user enters the invalid password in the first or second try, the user will be asked to re-enter the password. And finally, if the user enters incorrect password 3rd time, the account will be blocked.
In the diagram whenever the user enters the correct PIN he is moved to Access granted state, and if he enters the wrong password he is moved to next try and if he does the same for the 3rd time the account blocked state is reached.
Learning Outcomes

- The use of natural languages is explained in terms of defining requirements.
- The model-based requirement process, development and testing is explained in terms of its use.
- Use cases are explained in terms of their fit components and scope.
- Use cases are used to find missing requirements
Natural Language Processing

Natural language processing is one of the more efficient ways to program computers to interpret qualitative data. With the right solution, you can train a computer to perform sentiment analysis on large quantities of text-based responses, quickly gleaning insights on respondents' emotional reactions to, or opinions about, a particular subject or experience. Having a dependable NLP solution for textual analysis not only reduces the time it takes to read, and score written or free-form survey responses, but can also reduce human error and bias in your analysis.

In our work with the lab we developed a solution that saved time and improved consistency, as well as enabled us to home in on responses from the experiment's most and least satisfied users. As with a net promoter score, the most valuable responses to surveys are typically those on either end of the spectrum. By grouping responses as either highly positive or negative, we could begin to do text mining on the words most commonly used to express sentiment. These words could help tell a better story and provide context on how users responded to the experiments.

Another team objective was to apply the NLP algorithm to survey data from other lab experiments with minimal tweaking. With these objectives in mind, we began testing existing algorithms and developing our own.

How to build the algorithm

There were several approaches we could have taken to build and apply a natural language processing algorithm for sentiment analysis. We decided to start by building our own model from scratch, and experiment with using different types of existing data sets to train the model. We also looked at the success rates for existing sentiment analysis models and used them as benchmarks against which to compare our work. Ultimately, we went through three iterations of models and data sets to arrive at the solution.

First iteration: algorithm

To build the algorithm, we reviewed a number of existing NLP Python packages to mine ideas for how to handle language nuances. For example, the algorithm needed to be able to understand the logic of negation words, such as “not,” in front of a positive word. Once we created our base algorithm, we followed this process to train it:
Once we had our base algorithm written, we started experimenting with various datasets to train it. First, we tried a Twitter sentiment analysis dataset, but it only had a 57% accuracy rate when used to analyze data from the election survey. To us, this highlighted some discrepancies between common words used in tweets and the survey responses for the elections experiment. For example, words related to the topic of notifications such as “alerts” or “auto-updating” were used in the survey responses but not covered in the tweets.

Next we tried training the algorithm model on our own data set from the election survey, and we obtained an accuracy score of 81%. Better! However, when testing the same algorithm to analyze responses from an earlier experiment the lab ran for the Brexit vote, the accuracy dropped several points to 78%. While this method of training the algorithm picked up more nuances and specifics of the lab’s experiments, the scope was still limited to words highly associated with experiments surrounding the topic of the election.

**Second iteration: The VADER algorithm**

To validate our own algorithm, we tested the same election survey data set with another algorithm called VADER. The VADER algorithm was created by researchers at Georgia Tech and has been trained through crowdsourcing, asking surveyed users to rate a series of words, emoticons, slang, and acronyms. The dataset that VADER was trained on includes over 7,000 words in its lexicon. We decided to switch to the VADER algorithm instead of using our own original base algorithm because it would allow us to accurately analyze a wider scope of words.

**Third iteration: Adapting the VADER algorithm**

To further improve the VADER algorithm’s accuracy, we added terms specific to the lab to its lexicon, based on the team’s input. For example, its lexicon did not include words such as “convenience”, or “up-to-date”, which were important keywords to add as they are benefits of participating in the experiments.
The results with the adapted VADER algorithm were the best of the three iterations. Although this score was a bit lower than the one, we produced with our survey-trained model, it pulled through when we tested it to analyze the survey results from the Brexit experiment, where the accuracy score was 88%. Thus, this meant that VADER was flexible enough to accurately handle and score sentiment for future experiments that may differ from the scope of US politics coverage.

Although we had set out to use VADER to validate the algorithm, we learned that adapting the VADER algorithm with words specific to the lab’s surveys turned out to be the most efficient solution. VADER is the algorithm we now use in our analyses. However, as more survey responses come in from new experiments, we could feed them into our own survey-trained model, enabling our model to potentially perform better than VADER over time as the experiment-specific data set grows larger.

The Outcome

Through the use of natural language processing, it substantially reduced the time it took to tag and grade the sentiment of survey responses. If the team had read and manually tagged every one of the 1,400 election survey responses, the work would have taken about five hours. But with the algorithm, it could tag and grade the responses in less than five minutes.

In addition to reducing the time required for analysis, the algorithm allowed to home in on some of the most commonly used words associated with the experiment, segmented by positive and negative sentiment. For example, positive reactions were associated with people who liked the convenience of the live updates. Their responses included words such as “easy”, “live”, “check”, and “updated.” These key words gave us some hints about what users liked about the experiment.

Developing an NLP solution is a valuable investment and requires a team with the appropriate skillset. Although it is time consuming to build, it can significantly reduce the time required for analyses. We now use the modified VADER sentiment scoring algorithm to analyze free-form survey data from lab experiments.

Still, it’s important to note that developing NLP methods is an iterative process. In order to improve accuracy over time, it’s necessary to continuously add important keywords to its lexicon and allow the algorithm to evolve and adapt alongside the content included in the experiments.
Model-based testing (MBT) is an increasingly widely used technique for automating the generation and execution of tests. There are several reasons for the growing interest in using model-based testing:

- The complexity of software applications continues to increase, and the user’s aversion to software defects is greater than ever, so our functional testing has to become more and more effective at detecting bugs;

- The cost and time of testing is already a major proportion of many projects (sometimes exceeding the costs of development), so there is a strong push to investigate methods like MBT that can decrease the overall cost of test by designing tests automatically as well as executing them automatically.

- The MBT approach and the associated commercial and open source tools are now mature enough to be applied in many application areas, and empirical evidence is showing that it can give a good ROI;

- Model-based testing renews the whole process of functional software testing: from business requirements to the test repository, with manual or automated test execution. It supports the phases of designing and generating tests, documenting the test repository, producing and maintaining the bi-directional traceability matrix between tests and requirements, and accelerating test automation.

What is MBT

Model-based testing refers to the processes and techniques for the automatic derivation of abstract test cases from abstract formal models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases. Therefore, the key points of model-based testing are the modeling principles for test generation, the test generation strategies and techniques, and the concretization of abstract tests into concrete, executable tests.

Design a Test Model. The model, generally called the test model, represents the expected behavior of the system under test (SUT). Standard modeling languages such as UML are used to formalize the control points and observation points of the system, the expected dynamic behavior of the system, the business entities associated with the test, and some data for the initial test configuration. Model elements such as transitions or decisions are linked to the requirements, in order to ensure bi-directional traceability between the requirements and the model, and later to the generated test cases. Models must be precise and complete enough to allow automated derivation of tests from these models;
Select some Test Generation Criteria. There are usually an infinite number of possible tests that could be generated from a model, so the test analyst chooses some Test Generation Criteria to select the highest priority tests, or to ensure good coverage of the system behaviors. One common kind of test generation criteria is based on structural model coverage, using well known test design strategies such as equivalence partitioning, cause effect testing, pair-wise testing, process cycle coverage, or boundary value analysis. Another useful kind of test generation criteria ensures that the generated test cases cover all the requirements, perhaps with more tests generated for requirements that have a higher level of risk. In this way, model-based testing can be used to implement a requirement and risk-based testing approach. For example, for a noncritical application, the test analyst may choose to generate just one test for each of the nominal behaviors in the model and each of the main error cases; but for one of the more critical requirements, she/he could apply more demanding coverage criteria such as all loop free paths, to ensure that the businesses processes associated with that part of the test model are more thoroughly tested;

Generate the tests. This is a fully automated process that generates the required number of (abstract) test cases from the test model. Each generated abstract test case is typically a sequence of high-level SUT actions, with input parameters and expected output values for each action. These generated test sequences are similar to the high-level test sequences that would be designed manually in action-word testing [2]. They are easily understood by humans and are complete enough to be directly executed on the SUT by a manual tester.

The test model allows computing the expected results and the input parameters. Data table may be used to link some abstract value from the model with some concrete test value. To make them executable using a test automation tool, a further concretization phase automatically translates each abstract test case into a concrete (executable) scripts, using a user defined mapping from abstract data values to concrete SUT values, and a mapping from abstract operations into SUT GUI actions or API calls. For example, if the test execution is via the GUI of the SUT, then the action words are linked to the graphical object map, using a test robot such as HP QuickTest Professional, IBM Rational Functional Tester or the open source robot Selenium. If the test execution of the SUT is API-based, then the action words need to be implemented on this API. This can be a direct mapping or a more complex automation layer. The expected results part of each abstract test case is translated into oracle code that will check the SUT outputs and decide on a test pass/fail verdict.
The tests generated from the test model may be structured into multiple test suites and published into standard test management tools such as HP Quality Center, IBM Rational Quality Manager or the open source tool Test Link. Maintenance of the test repository is done by updating the test model, then automatically regenerating and republishing the test suites into the test management tools;

**Execute the Tests.** The generated concrete tests are typically executed either manually or within a standard automated test execution environment, such as HP QuickTest Professional or IBM Rational Functional Tester. Either way, the result is that the tests are executed on the SUT, and we find that some tests pass, and some tests fail. The failing tests indicate a discrepancy between the SUT and the expected results designed in the test model, which then needs to be investigated to decide whether the failure is caused by a bug in the SUT, or by an error in the model and/or the requirements. Experience shows that model-based testing is good at finding SUT errors but is also highly effective at exposing requirements errors even far before executing a single test.