Verification and Validation

Table of Contents

1. Fundamentals of Testing

Key Learning Outcomes

At the end of this module, you will be able to:

1. Say the importance of testing
2. Analyze the difference between testing and debugging
3. Explain the importance of testing
4. List the relationship between testing and quality assurance
5. Importance of quality embedded system products
6. Explain the consequences of non-tested embedded system products
7. Identify the different seven testing principles used in the embedded system testing
8. Describe the test process
9. Test activities and respective tasks between test processes
10. Explain the Psychological factors that lead to success
11. Differentiate between tester's and developer's mindset

UNIT 1.1: What is Testing?

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the importance of testing in embedded system development.
2. Differentiate between testing and debugging.

1.1.1 What is Testing?

System testing enables us to gauge the quality of software in terms of the number of flaws discovered, the number of tests executed, and the system under test. This can be accomplished for both the non-functional software requirements and characteristics as well as the functional software aspects. Examples of functional software aspects include publishing a report correctly, printing a report quickly, etc.

If testing uncovers few or no defects, it can inspire trust in the software's quality, provided we are satisfied with the testing's level of rigor. Of course, a subpar test might only find a few flaws and give us a false sense of security. If there are any flaws, a well-designed test will find them, and if it passes, we may be more confident in the software and say that the overall risk of using the system has been lowered. If faults are discovered during testing, they can be rectified to improve the quality of the software system, provided the corrections are done correctly.

1.1.2 Identify Typical Objectives of Testing

The main goal of testing is to deliver high-quality products. Do not have unrealistic expectations when someone spends time or money on an application. Testing improves the software and ensures that it meets the users' expectations. Various objectives of testing are identification of bugs and errors, ensuring product quality, justifying requirements, building confidence, and enhancing growth.

Identification of bugs and errors: The tester begins testing once the developer has finished coding. Each module is validated by the QA under various conditions during testing. Then the errors and bugs are compiled together and sent to the developer for correction.

Product Quality: Ensuring product quality is the main goal of testing. Furthermore, testing has its own cycle, with each phase focusing primarily on quality.

Requirements Justification: During testing, the Quality Analyst (QA) team checks the application's compliance with the SRS (System Requirement Specification) document.

Build Confidence: The testing team regularly analyses the features and functionality of the software. It must satisfy commercial needs and promote confidence.

Enhance Growth: A quality product boosts a company's potential, and quality is only achieved through testing.

1.1.3 Approach of Testing

Working on technical knowledge is simply one aspect of testing. It also necessitates levelheadedness. A tester should have an end-user mentality. They should think of all the questions that the end user may have. Examples include: will the end-user like the utilized colour scheme, what happens if the phone is dead, what inquiries can they make during a support session (Testers require this while evaluating chatbots), Is the layout satisfactory, etc. A tester must consider all such scenarios and fulfil the client's expectations.

1.1.4 Differentiate Testing from Debugging

Errors and bugs are identified during testing. Debugging is the process of fixing the bugs and errors identified during the testing process.

Parameters	Testing	Debugging
Basics	It is the process of identifying errors and bugs.	It is the process of fixing the errors and bugs identified during the testing process.
Code Failure	Any failure in the code can be identified through this process.	Code failure can be absolved using this procedure.
Errors	This process displays the errors.	Errors are absolved and eliminated in this process.
Performer	The tester can test any particular software or application.	This step in the debugging process is performed in software or an application by the developer or programmer.
Mode of Operation	Testing can be done manually or automatically.	Debugging must always be done manually because the current errors and bugs are being fixed. A system, piece of software, or application cannot be debugged automatically.
Basis of Operation	The testing process is based on various levels of testing (system testing, integration testing, unit testing, etc.).	The debugging process is built around the various types of issues that can exist in a system.
SDLC	It is a stage in the software development life cycle.	It has absolutely nothing to do with the SDLC. It occurs as a result of the testing process.
Steps	This process consists of both validation and verification of the errors	In this process, the available symptoms are compared to the causes. Does it result in the error correction until then?

Summary

In this unit, we learned about the following concepts:

1. What is Testing?
2. The objectives of testing.
3. The difference between testing and debugging.

UNIT 1.2: Why is Testing Necessary?

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the need for testing.
2. Explain the relationship between testing and quality assurance.
3. Explain the importance of testing for higher quality embedded system products.
4. Explain the impact if the embedded system product is not tested.

1.2.1 Why Testing is Necessary?

Software testing is the process of identifying errors and bugs in software applications and products to ensure efficient performance. Testing is a pivotal step in the process of developing secure and functional software products. This necessitates the need for effective software development. The need for quality assurance in software development arises from the need to understand the different types of software testing.

Generally, the companies that aim to create their own software have in-house software testers. The testers ensure that these software products are ready to be sold in the market by identifying the bugs and errors that impact the quality of the software. Testers verify the functionality of the software and help in changing these software applications into end-user products that function as intended. Although the many different types of software application testing can be difficult to fully comprehend, it is critical for businesses to understand the need for software testing. Also, it is essential to note that the company is responsible for testing the software if they are developing it for the market.

Software testing has already been performed if they are acquiring off-the-shelf software, especially one that has been reviewed and validated. In such cases, software testing is not required, as it would be redundant to repeat the procedure. If they are creating custom software for their specific requirements, they should consider the objective of testing. There may be a number of benefits to a business investing resources in building its own software. Every business should analyse this on its own terms.

1.2.2 Reasons Why Software Testing Is Necessary

Software testing should be considered for the following reasons:

- To gain customer confidence
- To check software adaptability
- To identify errors
- To avoid extra costs
- To accelerate software development
- To avoid risks
- To optimise business

1.2.2.1 To Gain Customer Confidence

Fig. 1.2.2.1 Gaining Customer Confidence

Software testing ensures that the software is user-friendly, thereby making it suitable for use by the intended target audience. Specialized software testers are familiar with the client's requirements. Money spent on developing the software is basically a waste of investment until it can meet the client's requirements. Customer types vary depending on the type of software. For instance, a hospital administrator seeking to determine which medical departments require more resources would have a very different software package compared to the human resources department of a corporation that has to track its employees and their performance. As a result, testers, like developers, specialise in specific types of software designs. This ultimately drives confidence in software testing. The procedure is case-sensitive and takes special care to satisfy customers.

1.2.2.2 To Check Software Adaptability

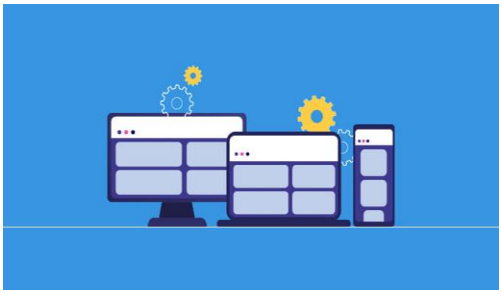


Fig. 1.2.2.2 Software Adaptability

There are numerous devices, browsers, and operating systems available today. It is critical to ensure that the software is compatible with all the platforms to provide a smooth user experience. Functionality can be compromised due to a change in devices, which will ultimately result in a negative user experience. Therefore, testing is essential to ensure that the software is compatible with different platforms, adapts to a change in devices, and eliminates any errors or bugs in the software. Software can be developed to be specifically used only on a desktop or mobile phone. Seeing how the networking landscape and mobile computing are changing, how beneficial would that software be? Customers are less likely to select software that runs well on an operating system like that of a Mac. Only the software with the highest levels of adaptability will actually succeed in the market.

1.2.2.3 To Identify Errors

Fig. 1.2.2.3 Error Identification

It is essential to note that bugs and errors in software are removed through testing before it is delivered to the client. There is always the possibility of bugs and glitches in the software, no matter how skilled the developers are. Software testing will improve the functionality of the software by exposing hidden errors and glitches. Even a single bug can hurt the reputation of the company, and it may take years to gain back the lost confidence. Therefore, it is essential to ensure that the software undergoes rigorous testing before it is delivered. The numerous software tests that the software undergoes will make it valuable to the clients.

.1.2.2.4 To Avoid Extra Costs

Fig. 1.2.2.4 Cost Deduction

The cost of compensating customers for using glitchy software is related to the issues caused by the errors in the software. These extra costs might result in a sizable amount of damage to the company. Not only are the clients dissatisfied with the delivered product, but they also wasted a significant amount of time, which they could have invested elsewhere.

Also, there is an illusion that the testers themselves break the software due to the huge amount of stress that they deal with. Testers feed huge chunks of data to the software applications to ensure their ability to handle huge data loads without compromising their functionality. What purpose does software serve for its clients if it cannot handle large volumes of data? Software testing is the most economical way of handling software applications. As the testers identify errors and bugs before any critical problems can arise, they alleviate the need for a continuous cycle of updates and bug fixes.

1.2.2.5 To Accelerate Software Development

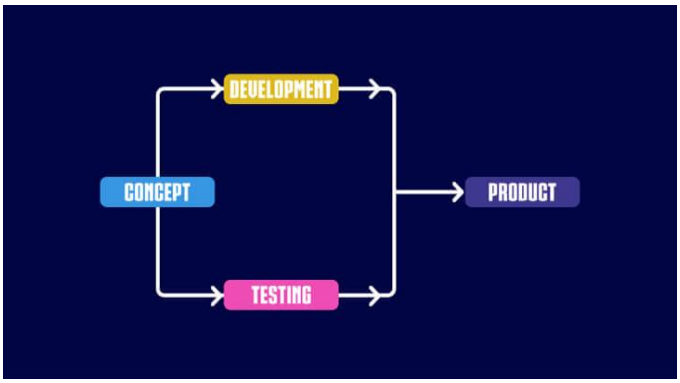


Fig. 1.2.2.5 Software Development

Some companies may skip the software testing process due to the large amount of time that they spent developing the software and to deliver the products to their customers right away. This is not just the result of poor time management by the development team but also a perception issue with how the software testing process is viewed. The benefits of running software development and testing in parallel are that it saves a lot of time and improves efficiency. In order to avoid such pitfalls, it is essential to stage the design process in such a way that the software development and testing processes occur concurrently.

1.2.2.6 To Avoid Risks

Fig. 1.2.2.6 Risk Management

Errors and bugs imply that the software is not secure. These errors can lead to vulnerabilities, especially when the software is designed for organizations and enterprises. This can result in the loss of significant information to rival companies and multiple communication blunders within an organization. Bugs and errors in software applications not only damage the developer's reputation and annoy the clients that use the software, but they also lead to data gaps and privacy leaks that can cost more than the previous issues.

1.2.2.7 To Optimize Business

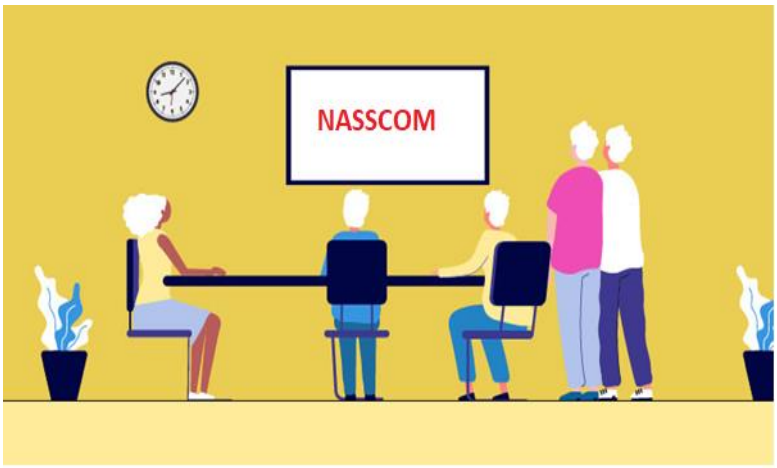


Fig. 1.2.2.7 Business Optimization

Testing enables the product to meet a higher quality standard before going live. It contributes to the company's profitability by lowering support expenses and improving the brand image. In essence, every software developer wants to retain its customers, and every consumer wants to use a service that is dependable and worthwhile. So, offering high-quality software enables a company to establish a solid reputation as a software provider.

1.2.3 Describe the Relationship between Testing and Quality Assurance

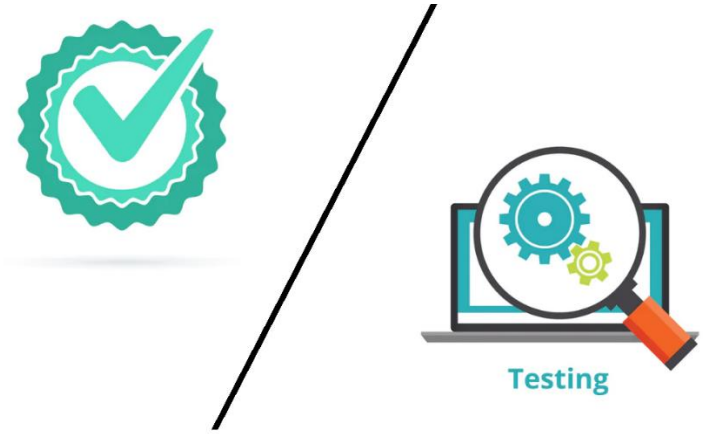


Fig. 1.2.3.1 Testing and Quality Assurance

A comparison between the characteristics of software testing and quality assurance is given below. Software testing is the process of identifying errors and bugs in software applications, whereas quality assurance is performed at each stage of the development process. Software testing is performed to ensure the functionality of the software. On the other hand, quality assurance is performed to ensure that the software application meets industry standards. Therefore, in this process, the verification begins at the inception of the software application.

The software testing process is primarily focused on improving the efficacy of the software by removing bugs from it and verifying its functionality. Whereas, the primary focus on quality assurance is product and solution examination, i.e., whether the product/software works as intended, whether the client requirements are fulfilled, etc. It is a management approach to ensure that the quality objectives of the company are met. Processes, standards, training, and policies required to achieve the quality objectives are part of the quality assurance process. Testing itself is part of the QA strategy.

	Quality Assurance	Testing
Definition	Activities designed to ensure the software corresponds to spec.	The process of exploring a system in order to find defects
Focus	Quality control and meeting the requirements	System inspection and bug finding.
Orientation	Process-oriented	Product-oriented
Activity Type	Preventive	Corrective
Aim	To assure the quality	To control the quality

Fig. 1.2.3.2 Comparison between the characteristics of Software Testing and Quality Assurance

1.2.4 How Testing Contributes to Higher Quality

Review: Testing is routinely performed to ensure the quality of the software. It ensures that errors and bugs can be fixed before they get out of hand and that the software meets industry standards.

Revise: It is essential to analyse what has been effective throughout the development process, employ effective strategies, and determine if innovation can further improve the software.

Remember: It is essential to keep track of what worked well and what did not after delivering quality software. The list should contain both the pros and cons and be used as a reference while developing the next software application.

1.2.5 Impact of Improper Product Testing

Releasing an untested product on the market might have adverse effects on a company. Here are some instances of the ill effects that the companies have faced in the past due to releasing their untested or inadequately tested product in the market:

1. Financial loss: An untested product can cause severe financial loss to a company. For instance, Samsung introduced the Galaxy Note 7 in 2016 without conducting adequate testing. The phone batteries had a manufacturing flaw that made them susceptible to fire and explosion. As a result, they had to recall millions of phones, which resulted in a $5 billion loss in revenue.
2. Damage to brand reputation: Releasing an unsafe or faulty product can harm the reputation of the company. For instance, in 2019, two crashes involving Boeing's 737 Max aircraft occurred as a result of a software problem. 346 people died in these crashes, and this significantly reduced public confidence in Boeing.
3. Legal consequences: Releasing an untested product could also cause legal issues for the company if the product harms consumers. For instance, in 2018, a jury awarded $417 million to a woman who got ovarian cancer from using Johnson & Johnson's talcum powder. The company was convicted for the following reasons: failing to warn consumers about potential side effects and not testing the product adequately.
4. Decrease in customer loyalty: Releasing an untested product can also reduce customer loyalty. Customers with negative encounters with a product may not trust the company again. This can reduce sales and revenue.

Eventually, releasing an untested product can have negative impacts on the company. This can be financial loss, damage to brand reputation, legal consequences, or a decrease in customer loyalty. Before releasing their products on the market, companies must invest adequately in testing to ensure that their products are secure and efficient.

Summary

In this unit, we learned about the following concepts:

1. Need for testing
2. Why are embedded products tested?
3. Difference between software testing and quality assurance
4. Impacts of improper/inadequate product testing

UNIT 1.3: Seven Testing Principles

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the difference between the seven testing principles.

1.3.1 Seven Testing Principles

- Testing shows the presence of defects
- Exhaustive testing is not possible
- Early testing
- Defect clustering
- Pesticide paradox
- Testing is context dependent
- Absence of errors fallacy

Testing shows the presence of defects:
Testing indicates the presence of defects; it cannot confirm that the software is error-free. It reduces the probability of undiscovered defects in the software. In a software, the absence of defects does not conform to its correctness.

Exhaustive Testing is Impossible:
Testing every combination of inputs and preconditions except for trivial cases is not feasible. Instead of exhaustive testing, risks and priorities are used to focus the testing efforts.

Early Testing:
Testing activities should start as early as possible in the software or system development life cycle. It should be focused on defined objectives.

Defect Clustering:
A small number of modules show the greatest operational failures or contain most defects discovered during pre-release testing.

Pesticide Paradox:
Repeating the same test for the same set of test cases will not yield any new bugs. This is known as the "pesticide paradox". The test cases should be reviewed and updated frequently. Also, novel and unique test cases should be built to test various sections of the system or software to uncover newer defects.

Testing is Context Dependent:
Based on the context, different types of software are tested differently. For instance, software for an e-commerce site is tested differently from safety-critical software.

Absence-Of-Errors Fallacy:

In a system that does not meet the user's requirements and is unstable, identifying and fixing bugs does not help.

1.3.2 Exhaustive Testing is Not Possible

As exhaustive testing is not feasible, it is essential to determine the right testing strategy. This should be done based on the application's risk assessment. But how is the risk determined? Let's look at an example. Which process can cause the operating system to malfunction? The answer to this question is too obvious, which is to run numerous applications in parallel. It is highly likely that the defects are found in the multi-tasking activities. Hence, extensive testing is necessary.

1.3.3 Defect Clustering

One of the applications of the Pareto principle to software testing is defect clustering. A small number of modules contain the most defects, i.e., 20 percent of the modules contain 80 percent of the defects. Such modules can be identified only through experience. But, even this approach has its flaws, i.e., repeating the same test for the same set of test cases will not yield any new bugs.

1.3.4 Pesticide Paradox

Insects will build resistance if the same pesticide mixtures are used to get rid of the insects. Hence, pesticides become useless against insects. Similarly, in software testing, repeating the same test for the same set of test cases will not yield any new bugs. Hence, the test cases should be reviewed and updated frequently. Also, novel and unique test cases should be built to test various sections of the system or software to uncover newer defects. Testers cannot solely rely on the existing test methodologies. To make testing more efficient, new ways to enhance current techniques should be discovered. Even after testing the product extensively, one cannot guarantee that it is bug-free. Check out the following YouTube video to see one of the biggest fails by Microsoft during the launch of Windows 98. https://youtu.be/yeUyxjLhAxU

1.3.5 Testing Shows a Presence of Defects

Software testing ensures that the defects are identified but does not guarantee the absence of defects. It lowers the likelihood of undetected errors in the software. But the absence of errors does not guarantee the accuracy of the software. But what if, even after following every procedure in the book, taking all the necessary precautions, and building an error-free software, it does not meet the client's requirements? This brings us to our next principle, the absence of error.

1.3.6 Absence of Error - Fallacy

Even software that is 99% error-free can be unstable. This occurs when testing is done based on incorrect requirements. Software testing is performed not only to ensure that the software is bug-free but also to ensure that the developed software meets the client's requirements. The idea of "absence of errors" is a fallacy if the software does not meet the client's requirements and is unstable. This brings us to our next principle: early testing.

1.3.7 Early Testing

In the software development lifecycle, the testing process should begin as early as possible. This is to ensure that any defects are identified as early as possible, in the requirements or design phase. Fixing a defect in the early stages of the process is much simpler and cheaper compared to fixing these defects in the later stages of the process. So, when should the testing process begin? One should start looking for bugs and errors right from the requirements stage.

1.3.8 Testing is Context Dependent

Testing is context-dependent, which means that different types of software are tested differently. For instance, software for an e-commerce site is tested differently from safety-critical software. Similarly, the software for an ATM machine is tested differently compared to a POS system. As all the developed software is not identical, different techniques, methodologies, approaches, and types of testing can be used based on the application type.

Summary

In this unit, we learned about the following concepts:

1. Seven testing principles
2. Advantages of these testing principles

Reference

https://www.guru99.com/software-testing-seven-principles.html
Book: Fundamentals of Software Testing (Author: Bernard Homes)

UNIT 1.4: Test Process

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the impact of context on the test process.
2. Describe the test activities and respective tasks within the test process.
3. Differentiate the work products that support the test process.
4. Explain the value of maintaining traceability between the test basis and test work products.

1.4.1 What is Test Process?

The collection of tools, techniques, and working methods used to perform a test.

1.4.2 Test Process in Context

In the context of software development, the testing process is a series of activities performed to ensure a set of objectives. The objectives of testing are to ensure that the developed software is error-free, meets the client's requirements, and is of high quality. As shown below, there are five phases in the testing process.

1. Test Planning: In this phase, the scope, objectives, and approach of testing are defined. Also, the testing resources are identified, and timelines are established.
2. Test Design: In this phase, test cases and scenarios are built based on the design specifications and software requirements.
3. Test Execution: In this phase, the test cases are executed, defects are reported, and fixes are verified.
4. Test Reporting: In this phase, the results are documented. This includes metrics such as defects found, test coverage, and the overall test status.
5. Test Closure: In this phase, the quality of the software and the testing process are evaluated. Also, the software is prepared for the next phase, i.e., development or release.

The testing process is crucial in ensuring that the software is reliable, of high quality, and meets the client's requirements.

The list of contextual factors that influence the testing process for an organisation is as follows:

- Project methodology
- Software development lifecycle model
- Test types and test levels
- Project and product risks
- Business domain

The list of contextual factors that influence the organisation's test process is as follows:

- Project methodology
- Software development lifecycle model
- Test types and test levels
- Project and product risks
- Business domain

The operational constraints include:

- Budgets and resources
- Timescales
- Complexity
- Contractual and regulatory requirements
- Organizational policies and practices
- Required internal and external standards

Based on the following factors the organizational test processes are defined:

- Test activities and tasks
- Test work products
- Traceability between the test basis and test work products

The test basis should have a defined, measurable coverage criterion for any type or level of testing being considered. The activities that illustrate the accomplishment of the software test objectives are driven by the coverage criteria, which effectively act as key performance indicators (KPIs). For instance, the test basis for a mobile application may include a list of supported mobile devices and their requirements. Here, each requirement and supported device is an element of the test basis. Under the coverage criteria, for each element of the test basis, at least one test case may be required. Upon execution, the test results inform stakeholders about any failures noted on the supported devices and whether the requirements were met. Further details on test procedures can be found in the ISO standard (ISO/IEC/IEEE 29119-2).

1.4.3 Impact of Context on the Test Process

Context-driven testing is a certain type of software testing. It takes into account the practical application of the product in a real-world scenario, such as in a production or performance environment. It enables the developers to evaluate the software during its development to identify any bugs and errors in order to optimise it before release. In agile software development, context-driven testing is performed with conceptual testing. In context-driven testing, issues related to user-friendly processes or user interfaces are evaluated as opposed to a more technical testing approach that focuses on syntax or code functionality. Unlike other technical testing methodologies, such as white box testing and black box testing, context-driven testing is centred around observing how people use software and determining whether the software functions well in real-world scenarios. Other testing methodologies, such as module testing and integration testing, assess individual code modules or interconnected modules that form a functional component of a software program.

1.4.4 Values of Context-Driven Testing

Context-driven testing is

Skeptical: Skepticism is not the rejection of belief but a rejection of certainty. It helps in observing and analysing more thoroughly, opening our minds to new and different ideas.

Empirical: The role of a human being in context-driven testing is vital. The knowledge and experiences people bring are of great value.

Adaptable: Testing should be focused on adaptability, i.e., not on creating as few tests as possible but as many and challenging ones as possible. It is important to adapt to changing contexts and requirements.

Diversified: It is important to consider how we can run more tests more quickly and more powerfully while increasing our capacity to observe multiple quality criteria.

Humanist: Testing must focus on human values. If quality is valuable to some people, then we need to understand people and their values. So, it is a must to learn to observe people, including customers, managers, programmers, and the project community.

Heuristic: Context-driven testing is heuristic, as it does not follow any process blindly. It helps to identify and understand the best approach to follow depending on the situation.

1.4.5 Test Activities and Tasks Within the Test Process

The primary groups of activities in a test process are:

- Test planning
- Test monitoring and control
- Test analysis
- Test design
- Test implementation
- Test execution
- Test completion

There are multiple activities listed under each activity group. There can also be multiple tasks under each activity within an activity group. This will be different for each project or release. Often these tasks are implemented iteratively, even though they appear to be logically sequential. For instance, Agile development entails continuous, smaller iterations of software design, development, and testing aided by ongoing planning. Therefore, within the development strategy, continuous iterative test activities are included. The logical sequence of activities should be tailored according to the project and system. This is because in the logical sequence, the activities may overlap, be executed in combinations, be executed concurrently, or be omitted.

1.4.5.1 Test Planning

The testing objectives and the strategy to meet these objectives are established through test planning. Examples include defining the test schedule in accordance with the timeline, deciding on the right tasks and test techniques, etc. Feedback from monitoring and control activities is used to update the testing plans.

1.4.5.2 Test Monitoring and Control

Test monitoring is the process of comparing the test plan with the actual test progress based on the defined metrics. Test control involves activities performed to meet the test plan objectives. The test plan objectives may vary from time to time. The evaluation of exit criteria, also referred to as the definition of done in some life cycles, supports test monitoring and control.

- For instance, at a given test level, the evaluation of exit criteria for test execution may include:
    - Verifying logs and test results with the specified test criteria
    - Verifying the quality level of a system or component based on the test results and logs
    - Assessing whether additional tests are required when tests fail to achieve the intended level of product risk coverage
    - Providing test progress reports to convey the test progress to the stakeholders with information on any deviations in comparison to the original test plan and data to support any decision to terminate testing

1.4.5.3 Test Analysis

During test analysis, the test basis is evaluated to determine the testable features and to specify the associated test conditions. Simply put, through test analysis, it is determined what should be tested based on the measurable coverage criteria. The major activities involved in test analysis are:

- Analyzing the test basis to the considered test level, for example,
    - Requirement specifications, such as functional requirements, system requirements, business requirements, epics, user stories, use cases, or similar work products that specify desired non-functional and functional system or component behaviour.
    - Design and implementation information, such as call flows, design specifications, software or system architecture documents or diagrams, interface specifications, modelling diagrams (e.g., UML or entity-relationship diagrams), or similar work products that describe a specific system or component structure.
    - Implementation of the system or component itself, including code, interfaces, database queries and metadata.
    - Risk analysis reports based on structural, non-functional, and functional aspects of the system or component.
- Analyzing the test item and test basis to detect the various types of defects, such as:
    - Ambiguities
    - Omissions
    - Inconsistencies
    - Inaccuracies
    - Contradictions
    - Superfluous statements
- Identifying features and sets of features to be tested.
- Based on the analysis of the test basis, the conditions for each feature should be defined and prioritized. During this process, factors such as structural, non-functional, and functional characteristics; risk levels; and technical and business factors should be considered.
- Capturing bi-directional traceability between each element of the test basis and the associated test conditions

Test analysis can identify test conditions that can double as test objectives in test charters. In experience-based testing, the test cases are the work products. The coverage attained during experience-based testing can be measured when the test objectives are traceable to the test basis. Defect identification through test analysis is a significant benefit when the review process is linked to the test process or when no other review process is used. Thus, test analysis not only ensures that the requirements are consistent, accurate, and complete but also confirms that the requirements are based on the user, consumer, and other stakeholder needs. Acceptance Test Driven Development (ATDD) and Behaviour Driven Development (BDD) are examples of techniques that generate test cases and test conditions from acceptance criteria and user stories before coding. These techniques can be used to verify, validate, and identify defects in these components.

1.4.5.4 Test Design

During test design, high-level test cases, groups of high-level test cases, and other test software are developed from the test conditions. Test design addresses the subject of "how to test," whereas test analysis addresses the subject of "what to test.

Major activities involved in the test design process are listed below:

- Identifying necessary test data to support test cases and test conditions
- Prioritizing and designing test cases and sets of test cases.
- Designing the test environment
- Identifying the necessary tools and infrastructure
- Establishing bi-directional traceability between the test basis, test conditions, test cases, and test procedures.

During test design, test techniques are used to elaborate the test conditions into test cases and sets of test cases. Also, during test design, defects can be identified in the test basis. Identifying defects during test design can be a significant benefit.

1.4.5.5 Test Implementation

The test ware required for test execution is developed and/or finished during test implementation. This includes arranging the test cases into test procedures in a specific order. Therefore, test design ensures the test process, while test implementation ensures that everything is in place to run the tests.

The activities involved in test implementation are listed below:

- Prioritizing and developing test procedures and creating automated test scripts
- Creating test suites based on test procedures and automated test scripts, if any
- Arranging test suites within a test execution schedule to ensure efficient test execution
- Developing the test environment, which includes simulators, service virtualization, test harnesses, and other infrastructure items and ensuring a proper test setup
- Preparing test data and loading it into the test environment
- Verifying and updating bi-directional traceability between the test conditions, test cases, test basis, test suites, and test procedures

Tasks related to test design and implementation are often combined.

- As a part of test execution, test design and implementation may occur in exploratory testing and other types of experience-based testing.
- Test charters created during test analysis serve as the foundation for exploratory testing. As exploratory tests are designed and implemented, they are executed instantly.

1.4.5.6 Test Execution

Test suites are run based on the test execution schedule during test execution. Major activities performed during test execution are listed below:

- Tracking the versions and IDs of test tools, test objects, test items, and test software.
- Executing tests either using test execution tools or manually.
- Comparing expected and actual results.
- Establishing the likely causes of anomalies by analysing them.
- Reporting defects based on the observed failures.
- Tracking the test execution outcomes, e.g., fail, pass, blocked, etc.
- Repeating the test activities, e.g., executing a corrected test or regression testing
- Verifying and updating bi-directional traceability between the test conditions, test cases, test basis, test suites, and test procedures.

1.4.5.7 Test Completion

Test completion activities compile testware, experience, and any other pertinent data from completed test activities. When a software system is launched, a test project is cancelled or finished, a test level is finished, an iteration of an Agile project is concluded, or a maintenance release has been accomplished, test completion activities take place.

The major activities involved in test completion are listed below:

- Ensuring that the defect reports are closed.
- Tracking product backlog items or change requests for defects that are not fixed at the end of test execution.
- Concluding and archiving the test data, environment, infrastructure, and other testware for future use.
- Delivering the testware to the project and maintenance teams and any other stakeholders who could use it.
- Determining the fixes and upgrades required for future releases, projects, or iterations from the test activities.
- Improving the test process using the obtained information.

1.4.6 Test Work Products

As a part of the testing process, test work products are developed. Like there are variations in how organisations deploy the testing process, there are also variations in the types of work products developed, how these work products are managed and organised, and how these work products are named. For test work products, the ISO standard (ISO/IEC/IEEE 29119–3) serves as a guideline.

The different types of work products are listed below:

- Test Planning Work Products
- Test Monitoring and Control Work Products
- Test Analysis Work Products
- Test Design Work Products
- Test Implementation Work Products
- Test Completion Work Products

Test Planning Work Products:

Typically, one or more test plans are included in the test planning work products. The test plan outlines the exit criteria used during monitoring and control and the test basis to which other work products will be traceable.

Test Monitoring and Control Work Products:

Different types of test reports, such as test summary reports and test progress reports, are included in the test monitoring and control work products. Information about the test progress as of the date mentioned in the report and the test execution results once they are made available. Concerns regarding project management, such as resource allocation, task completion, resource usage, and effort, should be addressed by the test monitoring and control work products.

Test Analysis Work Products:

Defined and prioritised test conditions are included in the test analysis work products. Bi-directional traceability to specific test basis elements that each of these test conditions covers is enabled. Test charters are created as a part of test analysis in exploratory testing. Another outcome of test analysis is the identification and reporting of defects on a test basis.

Test Design Work Products:

To exercise the test condition defined in the test analysis, test cases and sets of test cases are included in the test design work products. Designing high-level test cases without specific input or output values is a good practise. This is because these high-level test cases document the scope of the test while also being reusable in various test cycles. These test cases are bi-directionally traceable to the test condition(s) they cover. Identification of tools and infrastructure, design of the test environment, and identification or design of the required test data are all parts of test design. But it is essential to note that the extent to which these results are recorded varies greatly.

Test Implementation Work Products:

- Test procedures and the sequencing of those test procedures
- Test suites
- Test execution schedule

Test Execution Work Products:

- Documentation of the status of individual test cases or test procedures (e.g., ready to run, pass, fail, blocked, deliberately skipped, etc.).
- Defect reports
- Documentation about which test item (or items), test object (or objects), test tools, and testware were involved in the testing.

Test Completion Work Products:

Finalized testware, test summary reports, product backlog items, change requests, and action items for the improvement of the upcoming iterations or projects are included in the test completion work products.

1.4.7 What is Traceability in Software Testing

Test conditions should be traceable, i.e., they should be linked to their sources on a test basis. Traceability can be either vertical through the layers of development documentation, i.e., from components to requirements, or horizontal through all the test documentation for a given test level, i.e., from test conditions to test cases to test scripts, e.g., system testing.

1.4.7.1 Why is Traceability Important

When the requirements of a feature or a function change, a few fields will have different ranges to be entered. Therefore, it is essential to determine to which tests these boundaries apply and change them according to the requirement. This process will become easier if the tests are linked to the requirements. With the change in requirements, previously executed tests may fail. Therefore, it is essential to determine which functionality these tests execute. The affected features or functions can be easily identified if the tests are traceable to the requirements they execute.

1.4.7.2 Traceability Between the Test Basis and Test Work Products

Throughout the test process, traceability should be established and maintained between each element of the test basis and the various test work products related to that element. This is to ensure effective test monitoring and control. Through traceability, test coverage is evaluated.

Also, it helps to:

- Examine the impact of changes.
- Audit the testing.
- Meet the IT governance criteria.
- Improve the comprehension of the test summary report and test progress report by indicating the current status of the test elements. E.g., requirements with pending tests, requirements that passed or failed the test, etc.
- Explain the technical aspects of the testing process to the stakeholders in simple terms .
- Provide information to evaluate the project progress, product quality, and product capability in relation to the business goals.

Summary

In this unit, we learned about the following concepts:

1. Impact of context on the testing process
2. Various test activities and the respective tasks within the test process
3. How work products support the test process
4. The need for maintaining traceability between the test work products and test basis

References

https://medium.com/@HugoSaxTavares/istqb-foundation-level-syllabus-526e7cc7db4c
https://www.techopedia.com/definition/16625/context-driven-testing
https://quizlet.com/591485828/istqb-14-test-process-flash-cards/
http://tryqa.com/what-is-traceability-in-software-testing/
https://www.programsbuzz.com/article/traceability-between-test-basis-and-test-work-products

UNIT 1.5: The Psychology of Testing

Unit Objectives

At the end of this unit, you will be able to:

1. Identify the psychological factors that influence the success of testing
2. Explain the difference between the mindset required for test activities and the mindset required for development activities

1.5.1 Psychological Testing in Software Testing

Human psychology greatly influences the software testing process, as humans are involved in both the software development and testing processes.

Fig 1.5.1 Psychological Testing

One of the major factors behind the scenes that greatly influences the software testing process is human psychology. As it plays a major role, it has a significant impact on the outcome. The psychology of testing makes the process smooth and hassle-free. It is basically classified into three parts. The effectiveness of communication between the testers and developers and their mindset influences the psychology of testing. Also, it fosters teamwork and mutual understanding among the members. It is difficult to find flaws in one's own work, especially while focusing on the functionality of the system.

It could be difficult to consider what could go wrong. Generally, the mindset of the developers is solution-oriented, whereas the mindset of the testers is problem-oriented, i.e., how to break them rather than how to build them. It is not necessary for the testers to have a full knowledge of how the system works; rather, they should think from an end-user perspective to determine what could go wrong. In this sense, developers' familiarity with the system's operation may blind them to potential alternate scenarios that can result in unanticipated behavior. Thus, an effective tester should focus on finding ways to break the software, but that alone cannot yield success.

1.5.2 Psychology of Testing

The three parts of the testing psychology are listed below:

- Developer's and tester's mindset
- Effective and constructive communication
- Test Independence

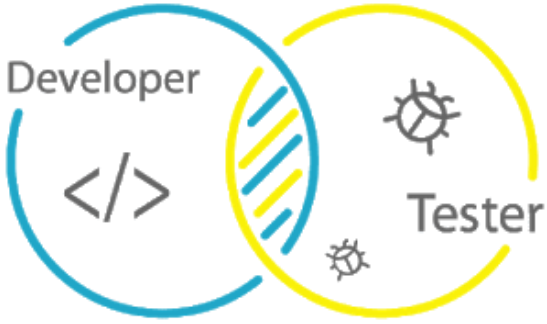


Fig 1.5.2 Developers and Testers

1.5.3 Software Development Lifecycle (Developer's Mindset)

The software development life cycle, or simply SDLC, refers to a set of processes and activities performed sequentially to develop a software product.

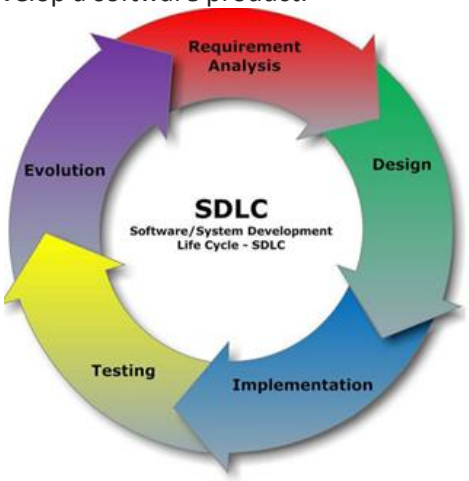


Fig 1.5.3 SDLC Process

The activities include:

Requirement Analysis: Conduct Requirement Analysis for System Level Requirements to determine SWR.

Design: Understand the architectural runway for the proposed requirements and work with SWA to check the feasibility of the architecture to be implemented for code development.

Implementation: Implement the SW Code for the SWRs, following coding standards and guidelines for specific industries.

Testing: Analyze the test reports generated from various validation phases and fix the bugs for different SW releases.

Evolution: Repeat the process for various SW releases by keeping track of proposed milestones.

Code of Conduct for Developers

Do's

- Understand the hardware for which you are developing the software, and ensure that the software works efficiently with it.
- Write clean and modular code that is easy to update and maintain.
- Follow industry standards and best practices, such as using version control, testing the code thoroughly, etc.
- Optimize your code for speed and efficiency, considering the memory usage.
- Document your code well, including comments and other information in such a way that it is easy to understand.
- Ensure that your code is reliable and robust, especially if it is being used in safety-critical applications.
- Continuously improve your skills and stay up-to-date with new technologies and development practices.
- Focus on the work and be honest.
- Create transparency in the code while developing it.
- Ensure the code is of the best quality.
- Build the code in a secure way such that it does not affect the privacy of the company.

Don'ts

- Do not create code that is vulnerable to exploits and is insecure.
- Do not create code that compromises users' privacy in ways that aren't obvious to them.
- Do not create any backdoors in the code or degrade security to keep track of or restrict what users can do. Backdoors and security flaws will inevitably be exploited by adversaries.
- Do not build the code, knowing that it will harm the company over the long term, unless it is compelling for commercial reasons.
- Do not make wrong promises on the plans for the project.
- Do not rely on trial-and-error programming. It is essential to plan and design your software before starting to write code.
- Do not neglect testing and debugging. Make sure to thoroughly test your software and address any issues that arise.
- Do not assume that hardware will always work as expected. Be prepared to handle errors and unexpected behaviour in the code.
- Do not use outdated or deprecated libraries or tools. Always use the latest stable versions and follow security updates.
- Do not over-optimize the code. Focus on making it maintainable and readable first, and then optimise it for performance if necessary.
- Do not ignore user feedback. Consider the feedback from end-users and modify the software accordingly. Don't ignore user feedback. Take into account feedback from end-users and adjust your software accordingly.

1.5.4 Software Testing Lifecycle (Tester's Mindset)

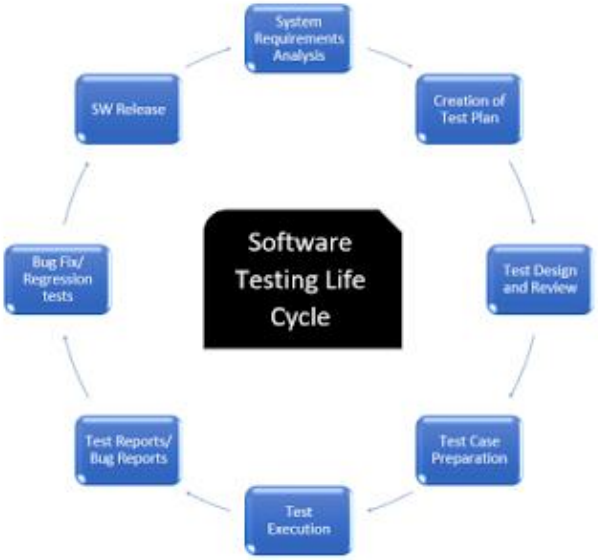


Fig 1.5.4 STLC Process

The activities include:

System Requirements Analysis: Analyze the system requirements (SYR) and create SYT (system test cases).

Creation of Test Plan: Understand the scope of the testing and create a plan for all the stakeholders that includes the due date of the deliverables, entry criteria for the tests, exit criteria for the tests, etc.

Test Design and Review: Design the tests by carefully choosing test design techniques, covering all the test cases, and reviewing them with the respective stakeholders.

Test Case Prepare: Prepare the test cases with proper traceability, including requirements, pre-conditions, test setup, etc.

Test Execution: Execute the tests.

Test Reports and Bug Reports: Understand and analyze the test reports and report them to the development team.

Bug Fix/Regression Tests: Retest a particular bug fix done by the development team and/or perform complete regression testing for the modules where the bugs were detected.

SW Release: Attach the test results for each SW release.

Code of Conduct for Testers

Do's

- Understand the requirements and design of the software being tested and develop a comprehensive test plan.
- Test the software thoroughly, covering all the use cases and edge cases.
- Use automation tools wherever possible to increase efficiency and reduce human error.
- Collaborate with the development team to ensure that issues are reported and resolved promptly.
- Document all the test results and provide clear and concise reports.
- Continuously improve your testing skills and stay up-to-date with new testing technologies and best practices.
- Advocate for the end-users and strive to ensure that the software is easy to use and meets their needs.

Don'ts

- Do not assume that the software is bug-free. Always approach testing with a critical mindset.
- Do not rely solely on manual testing. Use automation tools to increase efficiency and reduce human error.
- Do not ignore performance testing. Ensure that the software performs well under different loads and conditions.
- Do not prioritize quantity over quality. Focus on thorough testing rather than simply completing a certain number of tests.
- Do not ignore user feedback. Consider the feedback from end-users and modify the testing strategy accordingly.
- Do not skip regression testing. Ensure that the changes and updates made to the software do not introduce new bugs or issues.
- Do not ignore security testing. Ensure that the software is secure and not exploitable.
- Testers should not manipulate the Result.

1.5.5 Agile Testing Mindset

A software tester has to both identify bugs and prevent them. This can be achieved by implementing a continuous testing approach, optimising the process, and performing a proper requirements analysis. Hence, a tester's mindset should be about ensuring quality at every stage of the software development life cycle. As ensuring quality is the responsibility of the entire team in agile development, the primary goal has shifted towards the initiative and control of activities that prevent the occurrence of bugs and defects.

The Agile testing mindset is a way of approaching software testing in an Agile development environment. It involves a set of values, principles, and practises that emphasise collaboration, communication, and continuous improvement.

Key aspects of the agile testing mindset include:

1. **Customer collaboration**: Testing teams work closely with customers and stakeholders to ensure that the software meets their needs and expectations.
2. **Embracing change**: Testing teams are flexible and adaptable, responding to changes in requirements or feedback from stakeholders.
3. **Continuous improvement**: Testing teams are constantly looking for ways to improve their processes and deliver higher-quality software.
4. **Iterative approach**: Testing teams work in short cycles with frequent testing and feedback to ensure that software is delivered in a timely manner.
5. **Test automation**: Automated testing is an important part of the Agile testing mindset, allowing teams to quickly and efficiently run tests and catch issues early.
6. **Cross-functional teams**: Testing teams work closely with developers and other stakeholders to ensure that testing is integrated into the development process.
7. **Focus on quality**: Testing teams are focused on delivering high-quality software that meets customer needs and expectations.

What does the agile testing quadrants represent?

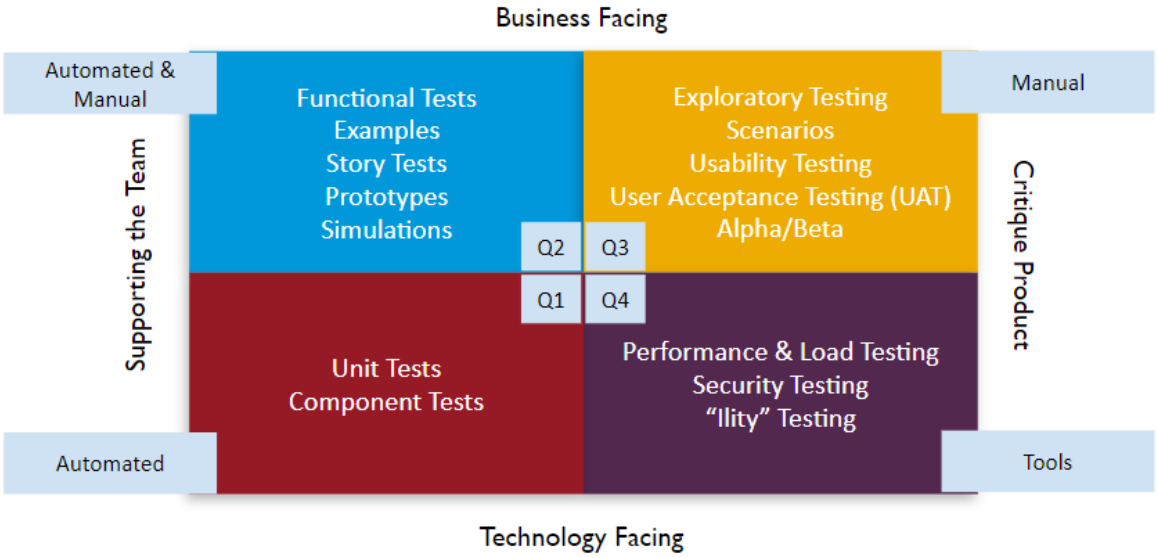


Fig 1.5.5.1 Agile Testing Quadrants

The agile testing quadrants are a development of Brian Marick's agile testing matrix. Depending on the business setting, suggestions on how to test an application are offered.

The matrix is divided into four quadrants. Each quadrant's side is associated with a distinct aspect, but the number itself does not indicate priority.:

1. Business Facing
2. Critique Product
3. Technology Facing
4. Supporting the Team

Also, the quadrants indicate if an approach is better suited for human testing, automated testing, or both. The four quadrants represent various testing purposes:

First Quadrant:

From a technology standpoint, the first quadrant proposes various methods to assist the team. This is focused on unit and component tests for testing the code.

Second Quadrant:

From a business standpoint, the second quadrant also recommends various methods for assisting the team. It includes the use of functional tests, examples, story tests, prototypes, and simulations. The aim is to concentrate the testing operations on the business rules.

Third Quadrant:

The third quadrant offers strategies that can assist the team in evaluating the product from a business standpoint. Some of the methodologies recommended include exploratory testing, scenarios, usability testing, user acceptability testing, alpha testing, and beta testing.

Fourth Quadrant:

Techniques focusing on critiquing the product from a technological standpoint are suggested in the fourth quadrant. It mentions the application of performance testing, load testing, security testing, and a few other "ility" testing categories like portability, dependability, and accessibility.

How to utilise the Agile Test Quadrant?

Let's look at the straightforward matrix initially developed by Brian Marick, which consists solely of the quadrants and their faces:

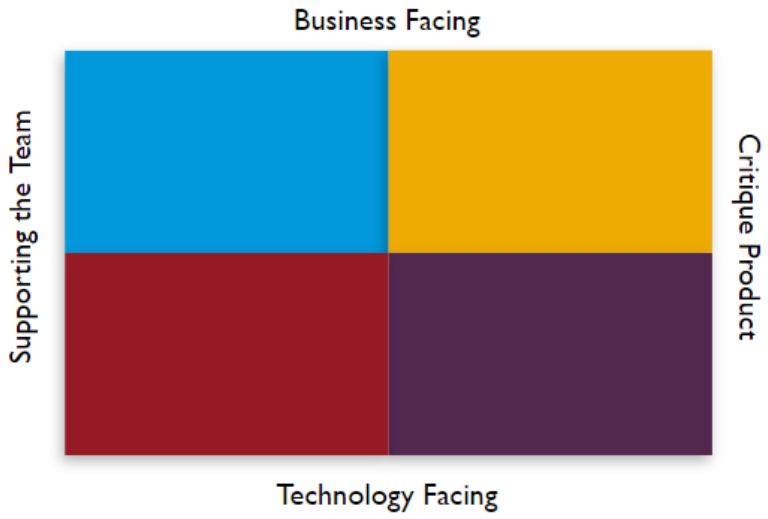


Fig 1.5.5.2 Brian Marick's Matrix

Understanding the testing environment is the first step. Before deciding which quadrant to use, it is essential to determine the answers to the following questions:

1. Should the focus be on praising the team behind the product's development or condemning it?
2. On the needs of the company or on flaws in the technology that was employed to create the product?

Based on the answers to the above questions, the type of quadrant to use should be decided. Choose and employ one of the suggested approaches in that quadrant. It is also crucial to note that the matrix does not include all of the testing methodologies that can be utilised and are suitable for use.

Example: If straightforward bugs in business rules are not being discovered, the agile testing quadrants help us to choose the right quadrants.

1. As we are working to fix business rule problems, quadrants 1 and 4 are not the right choices.
2. As simple bugs are being overlooked, the focus should be on assisting the team to locate the additional bugs before delivering the products.

Hence, quadrants 2 or 3 should be chosen. As greater support is required, quadrant 2 is the right choice.

Summary

In this unit, we learned about the following concepts:

1. The psychological factors that influence the success of testing
2. Involvement of developer's mindset and tester's mind set
3. Code of conduct for developers and testers
4. Working of agile testing quadrant

References

https://www.testdevlab.com/blog/7-psychological-factors-that-affect-testing
https://medium.com/@madhavikau/psychology-of-testing-in-software-testing-a59bdeb90ae0
https://softwaretester.careers/the-software-testers-mindset/#:~:text=The%20work%20of%20a%20software,the%20software%20development%20life%20cycle.

2. Testing Throughout the Software Development Lifecycle

Unit 2.1 - Software Development Lifecycle Models

Unit 2.2 - Test Levels

Unit 2.3 - Testing Types

Key Learning Outcomes

At the end of this module, you will be able to:

1. Use the Embedded System Software Development Life Cycle Model as per the requirement of the customer's need
2. Identify the importance of SDLC in the Embedded System Product Development
3. Differentiate various test levels in the Embedded System Development
4. Describe the various attributes of the Embedded System Software Test levels
5. Identify the different test types in the Embedded System
6. Use different test types for the various applications

UNIT 2.1: Software Development Lifecycle Models

Unit Objectives

At the end of this unit, you will be able to:

1. Explain several different software development lifecycle models, each of which requires different approaches to testing

2.1.1 Software Development Lifecycle Models

The software development lifecycle models are of three models

- Waterfall model
- V-model
- Iterative and incremental model (Agile)

2.1.2 Waterfall Model

According to the waterfall model, an embedded software project comprises multiple phases, including requirements gathering, analysis, design, coding, testing, and maintenance. These phases are executed in a sequential manner, implying that each subsequent stage cannot begin until the previous one has been completed, documented, and approved.

- Pros:
  - Step-by-step Verification &Validation
- Cons:
  - Ambiguous
  - Not dynamic
  - High risk
  - Misinterpretation of the requirements can cost the project, resources, and budget

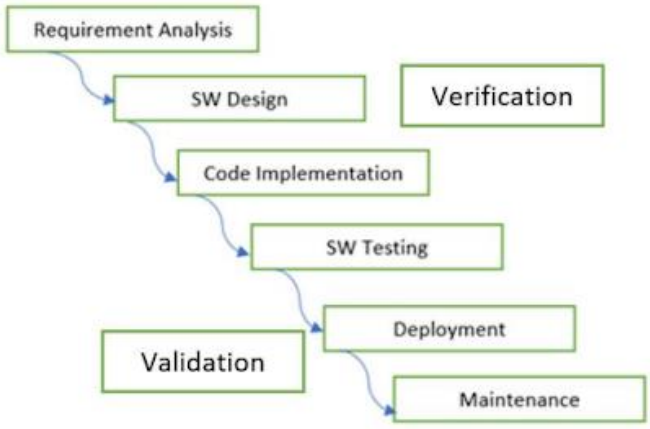


Fig 2.1.2 Waterfall Model

2.1.3 V-model

Low-level tests, such as unit tests and integration tests, are carried out on embedded products during the early stages of the development life cycle (on the left side of the V-model). These tests are usually conducted by developers themselves or in close collaboration with them. They form an integral part of the development plan and do not require a separate test plan or budget. On the other hand, high-level tests such as system tests and acceptance tests are executed towards the end of the development life cycle (on the right side of the V-model). Typically, an independent test team is responsible for conducting these tests and preparing a separate test plan and budget.

- Pros:
  - Artifacts related to validation are prepared during the verification phase (System Requirement (SYR) ↔ System Testing (SY) and Software Requirement (SWR) ↔ Software Testing (SWT)
  - Avoids flow of defects from one stage to another
  - More suitable for firmware application development
- Cons:
  - No consideration of changing requirements
  - Rigidity in the framework and not flexible
  - Hard to use for prototype developments

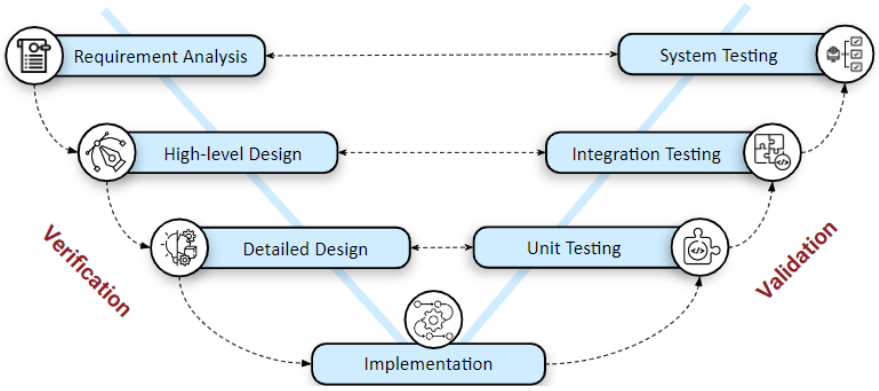


Fig 2.1.3 V-model

- Left side of the V-Model - Verification (System Requirement (SYR), Stakeholder Requirement (STR), High Level Design (HLD), Low Level Design (LLD)
- Right side of the V-model - Validation (Unit Testing (UT), Integration Testing(IT), System Testing (SYT) and User Acceptance Testing (UAT))
- Refresh your understanding of detailed activities performed in verification under the section "what do we verify?"
- Refresh your understanding of all the activities performed in validation under the section "what do we validate?"

2.1.4 Agile Model

The Agile model in embedded software development life cycle is an iterative and flexible approach to software development that focuses on delivering working software in short, frequent cycles called sprints or iterations. The Agile model emphasises close collaboration between team members and stakeholders, including hardware engineers, software engineers, testers, and customers.

- Pros:
  - Incremental and iterative approach
  - Working software for every iteration
  - Highly responsive to changing requirements
- Cons:
  - Workflow coordination is difficult
  - Dedicated availability of resources is the key challenge to overcome
  - Applicable for smaller team sizes

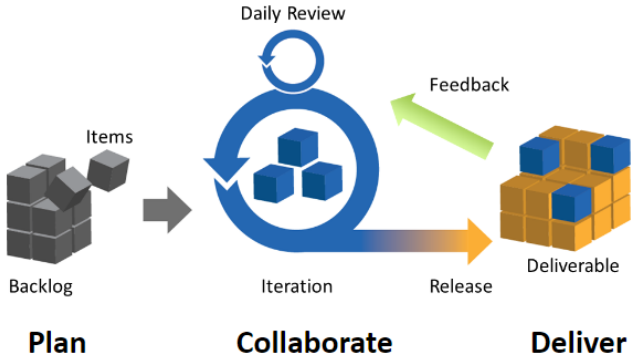


Fig 2.1.4 Agile Model

2.1.5 Test Sequence in SDLC

The following steps should be carried out during embedded software product testing:

1. Defect
2. Retest
3. Changed Items
4. Unchanged Items (Regression)

Defect:

A defect is also known as a bug or an issue. It is a flaw or a failure in a software product that prevents it from functioning as intended or performing its required functionality. Defects can occur at any stage of the software development life cycle, including the testing stage. A defect is typically identified during testing when the tester performs a set of predefined test cases, or test scenarios. This is to determine if the software product does not behave as expected or fails to meet certain requirements. Once a defect is identified, it is typically logged in a defect tracking system. This allows the development team to review, prioritize, and fix the defect.

Retest:

The purpose of retesting is to ensure that the software product is now working as expected and that the defect or issue that was previously identified has been resolved. Retesting helps to validate that the changes made by the developer to fix the defect have not introduced any new defects or issues into the system. Retesting is a critical step in the testing sequence, as it helps ensure that the software product is of high quality and meets the requirements of its users.

During the retesting phase, the tester would run the same test cases that failed previously to ensure that the defect has been fixed and that the system behaves as expected. If the test cases pass, the tester can close the defect. However, if the test cases fail, the tester will log the defect again and follow the defect testing sequence to have the developer fix the issue.

Changed Items:

Changed items refers to the components or elements of the software product that have been modified by the development team during the software development process. These changes may be the result of bug fixes, new feature implementations, or other modifications to the software product. When a software product undergoes changes, it is important to test these changes thoroughly. This is to ensure that the changes do not introduce new defects or issues into the system. Testing changed items typically involves executing a set of predefined test cases or scenarios that are designed to test the functionality of the modified components.

Unchanged Items:

Unchanged items refer to the components or elements of the software product that are not modified during the software development process. These components have already been tested and verified in previous testing cycles, and they are assumed to be working as expected. When testing unchanged items, the testing team typically executes a subset of the previously defined test cases that were used to test these components in previous testing cycles. The purpose of this testing is to ensure that the unchanged components have not been affected by changes made to other parts of the system during the current development cycle.

Summary

In this unit, we learned about the following concepts:

1. The Software Development Life Cycle model
2. Different Types of SDLC
3. The test Sequences involved in the SDLC

UNIT 2.2: Test Levels

Unit Objectives

At the end of this unit, you will be able to:

1. Compare the different test levels from the perspective of objectives, test basis, test objects, typical defects and failures, and approaches and responsibilities

2.2.1 Test Levels

Test levels are groups of test activities that are organized and managed together. Test levels are related to other activities within the software development lifecycle.

The test levels used in this syllabus are:

- Component testing
- Integration testing
- System testing
- Acceptance testing

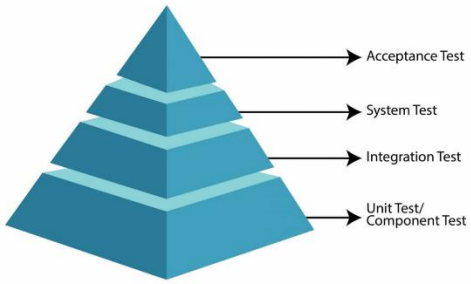


Fig 2.2.1 Test Levels

Test levels are characterized by the following attributes:

- Specific objectives
- Test basis, referenced to derive test cases
- Test object (i.e., what is being tested)
- Typical defects and failures
- Specific approaches and responsibilities

For every test level, a suitable test environment is required. In acceptance testing, for example, a production-like test environment is ideal, while in component testing the developers typically use their own development environment.

2.2.2 Component Testing

Component testing is a type of testing that focuses on verifying the individual components or modules of a software system in isolation. The purpose of component testing is to detect defects or errors in the code and ensure that each component functions correctly on its own.

Component testing is typically performed by software developers using specialized testing tools and techniques such as unit testing frameworks. During this testing phase, each component is tested independently to ensure that it meets its intended functionality and specifications. The tests are designed to cover all possible paths of execution and input/output combinations.

Once each component has been tested and verified to be working as intended, they can be integrated and tested together in the next phase of testing, known as integration testing. Overall, component testing is an important part of the software development process that helps ensure the quality and reliability of the final product.

2.2.2.1 Objectives of Component Testing

The objectives of component testing include:

- Reducing risk
- Verifying whether the functional and non-functional behaviours of the component are as designed and specified
- Building confidence in the component's quality
- Finding defects in the component
- Preventing defects from escaping to higher test levels

2.2.2.2 Test Basis of Component Testing

Examples of work products that can be used as a test basis for component testing include:

- Detailed design
- Code
- Data model
- Component specifications

2.2.2.3 Test Objects of Component Testing

Typical test objects for component testing include:

- Components, units, or modules
- Code and data structures
- Classes
- Database modules

2.2.2.4 Typical Defects and Failures of Component Testing

Examples of typical defects and failures for component testing include:

- Incorrect functionality (e.g., not as described in design specifications)
- Data flow problems
- Incorrect code and logic

Defects are typically fixed as soon as they are found, often with no formal defect management. However, when developers do report defects, this provides important information for root cause analysis and process improvement.

2.2.2.5 Specific Approaches and Responsibilities of Component Testing

Component testing is usually performed by the developer who wrote the code, but it at least requires access to the code being tested. Developers may alternate component development with finding and fixing defects. Developers will often write and execute tests after having written the code for a component. However, especially in Agile development, writing automated component test cases may precede writing application code.

For example, consider test driven development (TDD). Test driven development is highly iterative and is based on cycles of developing automated test cases, then building and integrating small pieces of code, then executing the component tests, correcting any issues, and refactoring the code. This process continues until the component has been completely built and all component tests are passed. Test driven development is an example of a test-first approach. While test driven development originated in eXtreme Programming (XP), it has spread to other forms of Agile and sequential lifecycles.

2.2.3 Integration Testing

Integration testing for embedded products refers to the process of testing the interactions between different software and hardware components of an embedded product. This is to ensure that they work together as intended. In embedded systems, software and hardware are tightly integrated. The errors or faults in one component can impact the entire system's functionality. Therefore, integration testing is critical for detecting and fixing issues before the product is released to the market.

2.2.3.1 Objectives of Integration Testing

- Reducing risk
- Verifying whether the functional and non-functional behaviors of the interfaces are as designed and specified
- Building confidence in the quality of the interfaces
- Finding defects (which may be in the interfaces themselves or within the components or systems)
- Preventing defects from escaping to higher test levels

2.2.3.2 Test Basis of Integration Testing

Examples of work products that can be used as a test basis for integration testing include:

- Software and system design
- Sequence diagrams
- Interface and communication protocol specifications
- Use cases
- Architecture at component or system level
- Workflows
- External interface definitions

2.2.3.3 Test Objects of Integration Testing

Typical test objects for integration testing include:

- Subsystems
- Databases
- Infrastructure
- Interfaces
- APIs
- Microservices

2.2.3.4 Typical Defects and Failures of Components Integration Testing

Examples of typical defects and failures for component integration testing include:

- Incorrect data, missing data, or incorrect data encoding
- Incorrect sequencing or timing of interface calls
- Interface mismatch
- Failures in communication between components
- Unhandled or improperly handled communication failures between components
- Incorrect assumptions about the meaning, units, or boundaries of the data being passed between components

2.2.3.5 Typical Defects and Failures for System Integration Testing

Examples of typical defects and failures for system integration testing include:

- Inconsistent message structures between systems
- Incorrect data, missing data, or incorrect data encoding
- Interface mismatch
- Failures in communication between systems
- Unhandled or improperly handled communication failures between systems
- Incorrect assumptions about the meaning, units, or boundaries of the data being passed between systems
- Failure to comply with mandatory security regulations

2.2.3.6 Specific Approaches and Responsibilities

Component integration tests and system integration tests should concentrate on the integration itself. For example, if integrating module A with module B, tests should focus on the communication between the modules, not the functionality of the individual modules, as that should have been covered during component testing. If integrating system X with system Y, tests should focus on the communication between the systems, not the functionality of the individual systems, as that should have been covered during system testing. Functional, non-functional, and structural test types are applicable.

2.2.4 System Testing

System testing focuses on the behavior and capabilities of a whole system or product, often considering the end-to-end tasks the system can perform and the non-functional behaviors it exhibits while performing those tasks.

2.2.4.1 Objectives of System Testing

- Reducing risk
- Verifying whether the functional and non-functional behaviors of the system are as designed and specified
- Validating that the system is complete and will work as expected
- Building confidence in the quality of the system as a whole
- Finding defects
- Preventing defects from escaping to higher test levels or production

2.2.4.2 Test Basis of System Testing

Examples of work products that can be used as a test basis for system testing include:

- System and software requirement specifications (functional and non-functional)
- Risk analysis reports
- Use cases
- Epics and user stories
- Models of system behavior
- State diagrams
- System and user manuals

2.2.4.3 Test Objects of System Testing

Typical test objects for system testing include:

- Applications
- Hardware/software systems
- Operating systems
- System under test (SUT)
- System configuration and configuration data

2.2.4.4 Typical Defects and Failures of System Testing

Examples of typical defects and failures for system testing include:

- Incorrect calculations
- Incorrect or unexpected system functional or non-functional behavior
- Incorrect control and/or data flows within the system
- Failure to carry out end-to-end functional tasks properly and completely
- Failure of the system to work properly in the production environment(s)
- Failure of the system to work as described in system and user manuals

2.2.4.5 Specific Approaches and Responsibilities

System testing should focus on the overall, end-to-end behavior of the system, both functional and non-functional. For example, a decision table may be created to verify whether functional behavior is as described in business rules. Independent testers typically carry out system testing. Defects in specifications (e.g., missing user stories, incorrectly stated business requirements, etc.) can lead to a lack of understanding of, or disagreements about, expected system behavior. Such situations can cause false positives and false negatives, which waste time and reduce defect detection effectiveness, respectively. Early involvement of testers in user story refinement or static testing activities, such as reviews, helps to reduce the incidence of such situations.

2.2.5 Acceptance Testing

Acceptance testing, like system testing, typically focuses on the behavior and capabilities of a whole system or product.

2.2.5.1 Objectives of Acceptance Testing

Objectives of acceptance testing include:

- Establishing confidence in the quality of the system as a whole
- Validating that the system is complete and will work as expected
- Verifying that functional and non-functional behaviors of the system are as specified

2.2.5.2 Test Basis of Acceptance Testing

Examples of work products that can be used as a test basis for any form of acceptance testing include:

- Business processes
- User or business requirements
- Regulations, legal contracts, and standards
- Use cases
- System requirements
- System or user documentation
- Installation procedures
- Risk analysis reports

2.2.5.3 Test Objects of Acceptance Testing

Typical test objects for any form of acceptance testing include:

- System under test
- System configuration and configuration data
- Business processes for a fully integrated system
- Recovery systems and hot sites (for business continuity and disaster recovery testing)
- Operational and maintenance processes
- Forms and reports
- Existing and converted production data

2.2.5.4 Typical Defects and Failures of Acceptance Testing

Examples of typical defects for any form of acceptance testing include:

- System workflows do not meet business or user requirements
- Business rules are not implemented correctly
- System does not satisfy contractual or regulatory requirements
- Non-functional failures such as security vulnerabilities, inadequate performance efficiency under high loads, or improper operation on a supported platform

2.2.5.5 Specific Approaches and Responsibilities of Acceptance Testing

Acceptance testing is often the responsibility of the customers, business users, product owners, or operators of a system, and other stakeholders may be involved as well.

Acceptance testing is often thought of as the last test level in a sequential development lifecycle, but it may also occur at other times, for example:

- Acceptance testing of a commercial off-the-shelf (COTS) software product may occur when it is installed or integrated.
- Acceptance testing of a new functional enhancement may occur before system testing.

Summary

In this unit, we learned about the following concepts:

1. The various stages of testing in terms of their goals in different test levels
2. The sources of information used to develop tests in different test levels
3. The items being tested in different test levels
4. Defects and failures in different test levels of SDLC

Reference

https://medium.com/@HugoSaxTavares/istqb-foundation-level-syllabus-part-2-of-6-e85155a27ccf

UNIT 2.3: Test Levels

Unit Objectives

At the end of this unit, you will be able to:

1. Compare functional, non-functional, and white-box testing
2. Recognize that functional, non-functional, and white-box tests occur at any test level
3. Compare the purposes of confirmation testing and regression testing

2.3.1 Compatibility Testing

Compatibility testing in embedded systems involves testing the compatibility of various hardware and software components within the system to ensure that they work together seamlessly. This type of testing is critical to ensure that an embedded system works as expected when it is deployed in the real world.

Embedded systems are often composed of various hardware and software components, including microprocessors, sensors, communication interfaces, firmware, and application software. These components may come from different vendors, have different specifications and interfaces, and may have been developed using different programming languages and tools.

Compatibility testing in embedded systems involves verifying that all of these components work together correctly and without any conflicts. This includes testing the compatibility of different hardware and software components, as well as different versions of the same component. It also involves testing the system's compatibility with different operating systems, platforms, and networks.

To conduct compatibility testing in embedded systems, test cases are designed to exercise different combinations of hardware and software components, as well as different operating conditions and scenarios. The testing process involves executing these test cases and analyzing the results to ensure that the system behaves as expected and meets the specified requirements.

Overall, compatibility testing is an essential part of the development process for embedded systems, as it helps to ensure that the system will work correctly and reliably in the real world.

2.3.2 Software Testing

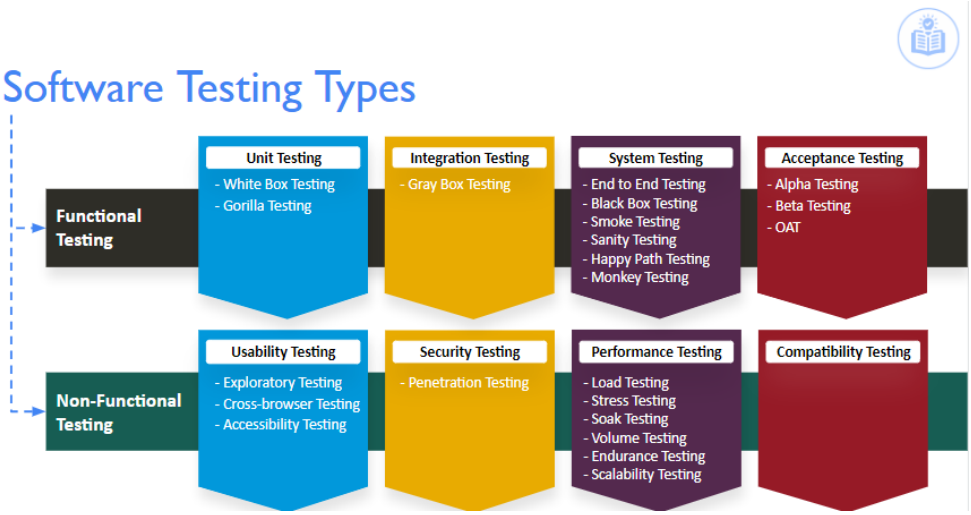


Fig 2.3.2.1 Software Testing

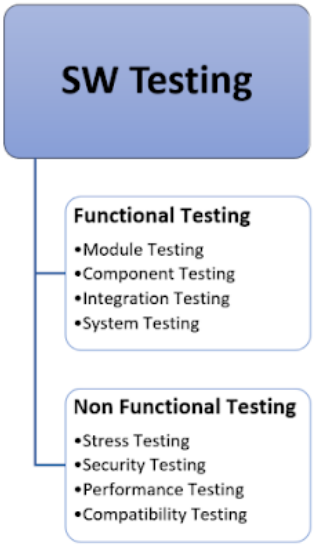


Fig 2.3.2.2 Functional and Non-functional Testing

2.3.3.1 Functional Testing

Any testing where we test the functionality of the software for the specified requirements is called functional testing.

- Component testing (A group of .c files)
- Module testing (A group of components)
- Integration testing (A group of modules to represent HLD (High-Level Design) requirement)
- System testing (Testing derived from SYR)

2.3.3.2 Non-Functional Testing

Any testing where we test the non-functional aspects of the software with respect to software components (SW Components) is non-functional testing.

- Stress Testing (How the implemented SWC reacts to incremental stress?)
- Security Testing (Conformance of SWC for security)
- Performance Testing (How the SWC behaves with various test environments?)
- Compatibility Testing (Is the implemented SWC compatible enough?)

Parameters	Functional Testing	Non-functional Testing
Execution	It is performed before non-functional testing	It is performed after the functional testing
Focus area	It is based on customer requirements	It focuses on customer's expectation
Requirement	It is easy to define functional requirements	It is difficult to define the requirements for non-functional testing
Usage	Helps to validate the behavior of the application	Helps to validate the performance of the application
Objective	Carried out to validate software actions	It is done to validate the performance of the software
Manual testing	Functional testing is easy to execute by manual testing	It's very hard to perform non-functional testing manually
Functionality	It describes what the product does	It describes how the product works
Example Test Case	Check login functionality	The dashboard should load in 2 seconds
Testing Types	Examples of functional testing types • Unit testing • Smoke testing • User acceptance • Integration testing • Regression testing • Localization • Globalization • Interoperability	Examples of non-functional testing types • Performance testing • Volume testing • Scalability • Usability testing • Load testing • Stress testing • Compliance testing • Portability testing • Disaster recovery testing

2.3.4 White Box Testing

A method of testing that may/may not test the functionality of the system by looking deep into the internal implementation/structures (design, coding, etc.) of the system. E.g., testing any embedded prototype, you designed by debugging into the software.

2.3.5 What is Regression Testing

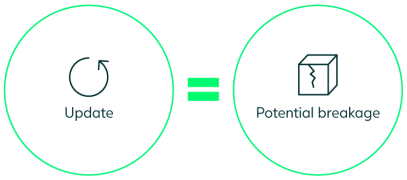


Fig. 2.3.5 Regression Testing

Regression testing is a type of software testing that is performed to ensure that changes or modifications made to a software application do not affect its existing functionality. As the name implies, regressing means returning to a former condition or state. It involves retesting previously tested software features or components to ensure that they still work correctly after any modifications, bug fixes, or enhancements are made. The primary goal of regression testing is to ensure that the software remains stable and does not break any previously working functionality, as well as to identify any new issues that may have arisen due to the changes made.

Regression testing can be performed manually or automatically, and it can be done at various stages of the software development lifecycle, such as during the initial development phase, after the system has been integrated, and after the release of a new version or patch.

Regression testing is an important part of software testing as it helps to ensure the quality of the software and reduces the risk of unexpected issues or bugs appearing in the software after release.

2.3.6 What is Retesting?

Retesting is an essential part of the software testing process because it helps ensure that the software is of high quality and free from defects. By retesting, testers can ensure that the software changes and updates have not introduced new defects and that the previously identified defects have been fixed. Regression testing, essentially, is the process of looking for flaws, whereas retesting is the process of correcting specific flaws that have already been discovered.

It is possible for these events to take place in a single testing procedure:

- Incorporation of a new feature into the software
- Verification of the existing functionality (regression testing)
- Identification of a defect in the existing functionality and rectification of the defect
- Verification of the functionality once again (with the expectation that it operates correctly)

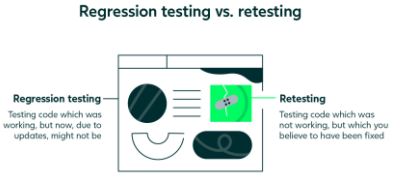


Fig. 2.3.6 Regression testing vs Retesting

2.3.7 Regression Testing Vs Retesting

Regression Testing	Retesting
Involves testing a general area of the software	Involves testing a specific feature of the software
Is about testing software that was working, but now, due to updates, might not be working	Is about testing software which you know was not working, but which you believe to have been fixed. You test it to confirm that it is now in fact fixed
Is ideal for automation as the testing suite will grow with time as the software evolves	Is not ideal for automation as the case for testing changes each time
Should always be a part of the testing process and performed each time code is changed and a software update is about to be released	Is only a part of the testing process if a defect or bug is found in the code

Summary

In this unit, we learned about the following concepts:

1. Different test levels involved in the embedded system product testing
2. Working of functional, non-functional, and black box testing on embedded products
3. The purposes of retesting as well as regression testing

References

https://www.softwaretestinghelp.com/types-of-software-testing/
https://www.guru99.com/functional-testing-vs-non-functional-testing.html
https://reqtest.com/testing-blog/white-box-testing-example/

Summary

In this module "Testing Throughout the Software Development Lifecycle", we learned about the following concepts:

- The working of different software development life cycle models (V-Model, Waterfall Model, and Agile Model)
- The functioning of various test types such as functional, non-functional, white box, confirmation, and regression testing
- The working of various test levels
- The difference between regression and confirmation testing

3. Static Techniques

Unit 3.1 - Static Testing Basics

Unit 3.2 - Review Process

Key Learning Outcomes

At the end of this module, you will be able to:

1. Explain about different static testing techniques
2. Identify errors in the code using static testing technique
3. Differentiate static and dynamic testing
4. Explain about work product review process
5. Explain about roles and responsibilities in formal view
6. Find defects in the code using work product review techniques
7. Identify the factors that influence a successful review process

UNIT 3.1: Static Testing Basics

Unit Objectives

At the end of this unit, you will be able to:

1. Recognize types of software work product that can be examined by the different static testing techniques
2. Use examples to describe the value of static testing
3. Explain the difference between static and dynamic techniques, considering objectives, types of defects to be identified, and the role of these techniques within the software lifecycle

3.1.1 Static Code Analysis

Did you know in 1997, USS Yorktown, a warship which had set sail in the Adriatic Sea for a mission, had brought down the combat machines on the network, causing the ship's propulsion system to fail, due to the failure of detecting Division by Zero Error? Drivers were reporting unintended acceleration with their XXX cars. The Company has ordered a recall of millions of units of cars produced. Among the SW problems they found were buffer overflow, invalid pointers, stack overflow, etc., which can be found using static code analysis.

The Static Code Verification/Static Code Analysis involves finding the fatal errors without executing the code. Essential in Embedded software. It is usually done on the software, which is properly compiled and, is independent of the compiler used for compilation.

Examples of work documents

- Requirement specifications
- Design document
- Source code
- Test plans
- Test cases
- Test scripts
- Help or user document
- Web page content

3.1.2 SW Testing Before Static Code Analysis?

The Initial process of software testing for finding out the run-time issues includes a manual code review. Manual code review can be an effective technique for finding run-time errors in relatively small applications (comprising 10-30k lines of code). For larger systems, however, manual review is a labour-intensive process. It requires experienced engineers to review source code samples and report dangerous or erroneous constructs, an activity that is complex, non-exhaustive, non-repeatable, and costly. Proving the absence of run-time errors is a much complex operation that is not manageable by code review.

3.1.3 Where to do Static Code Analysis in V-Model?

The Software development life cycle (V- Model in this case) requires finding the run-time issues early in the process and finding them immediately before the unit testing phase of the life cycle. This involves automating the process of finding the issues, which can be done by using static code analysis tools.

3.1.4 Bugs Identified in Static Code Analysis

Out of Bound Array Index (OBAI) usually occurs when the program tries to access the index variable which is outside the bounds of the array. Division by zero occurs when the denominator of one of the expressions in the source code is zero. Uninitialized Data variables (NIV/NIVL) are seen when there are variables that are declared and used in the code but not set to any value (Read access to uninitialized data). Illegal Dereferenced Pointers (IDP) occurs when there is a dereferencing of the NULL pointer in the source code and out of bound access of the pointer in an array. Integer Overflows (OVFL) includes finding the overflow of variables, if any, in the source code. For most arithmetic operations where an integer overflow occurs, there is an operand, which is arithmetically derived from some tainted data. Tainted data is derived from untrusted input sources like network messages, input files, or command-line options.

Effects in SW due to the above-mentioned bugs

- The Presence of the Out of Bound Array Index (OBAI) leads to the crashing of the software. The access of the index value which is outside the bounds of the size of an array will lead to the crash at runtime. Detecting this error with a high amount of precision before the unit test phase is what the static code analysis tool is expected to do.
- Division by zero usually checks if the denominator of the expression is zero. This error also leads to the catastrophic failure of the software.
- Uninitialized Data or Non-Initialized Variables/Non-Initialized Local Variables (NIV/NIVL) results in evaluating the expressions in the software with some garbage value of the variables and leading to unexpected results.
- Illegal Dereferenced Pointers (IDP) results in run-time exceptions in the software. This error compromises the security of the software. This also leads to the program crash when not detected and nullified before system testing.
- Whether or not an integer overflow (OVFL) is harmful depends on where and how the program uses the overflowed value. It is very dangerous when an overflowed value is used in some sensitive points since it may lead to other vulnerabilities.
- These sensitive points are called sinks.
- The sinks are primarily identified as memory allocation, memory access, branch statement, and other program-dependent sensitive points like overflowed value used in structure reference, which causes a shared object to be freed prematurely.

3.1.5 LDRA Tool

LDRA is an automated tool suite for testing and verifying embedded software for requirements traceability and standard compliance. The LDRA tool suite is developed by Liverpool Data Research Associates (LDRA).

3.1.5.1 Tool Capabilities

The tool suite provides multiple capabilities and services that include:

- Requirements traceability
- Test management
- Coding standards compliance
- Code quality review
- Code coverage analysis
- Data-flow and control-flow analysis
- Unit/integration/target testing
- Certification and regulatory support

3.1.5.2 Industries Served

Many objectives of various industry verticals are met by the tool suite, which includes:

- Aerospace and defense
- Medical
- Rail transportation
- Automotive
- Industrial and energy

3.1.5.3 Core Components

Core components of LDRA Test suite

S.No.	Component Name	Description
1.	LDRA Testbed	Helpful in the analysis of core static and dynamic analysis for embedded and host software. It provides compliance against different standards based on MISRA, JSF++ AV, CERT C, CWE, etc.
2.	TBvision	Helpful in achieving standards compliance, quality metrics, and code coverage analyses with help of LDRA Testbed
3.	TBrun	Helpful in running execution automatically for unit and integration testing
4.	TBmanager	Helpful in automating the whole traceability matrix between different project activities. standards objectives, requirements, design documents, source code, tests, and associated artifacts within the project verification workflow

3.1.5.4 LDRA Tool Suite

This section lists the features of the tool in a concise way.

Quality report:

The tool provides compliance against different quality metrics such as cyclometric complexity, knots metric, Halstead complexity measures, etc. This report provides a clear picture of code quality and confirms that the embedded software is maintainable and testable.

Automated code review (both static and dynamic):

No doubt, manual code review is best, but it is very difficult to perform if your code base contains thousands of lines of code. In addition, the skill of the auditor is quite critical to reviewing code. LDRA's tool suite performs automated code reviews that include both static and dynamic analysis of code against host and embedded software.

Compliance with standards:

The LDRA tool suite is very effective for checking the quality of code against standards such as MISRA and CERT. Similarly, the LDRA tool suite is quite effective in overall enhancing the quality of IT products to a level where it is difficult to compromise.

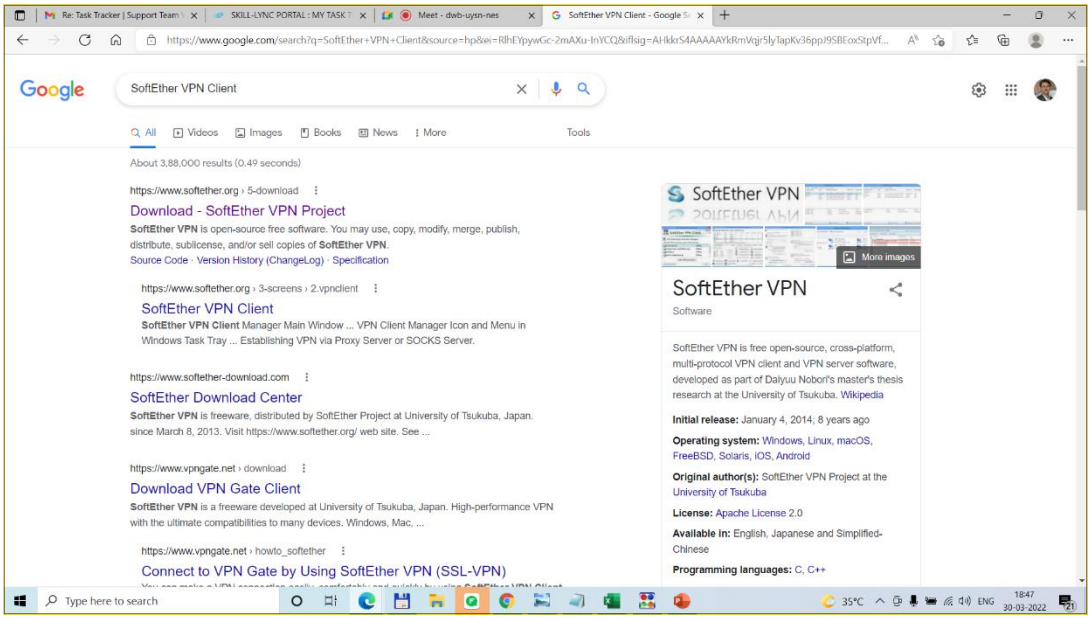
3.1.5.5 Support Different Languages

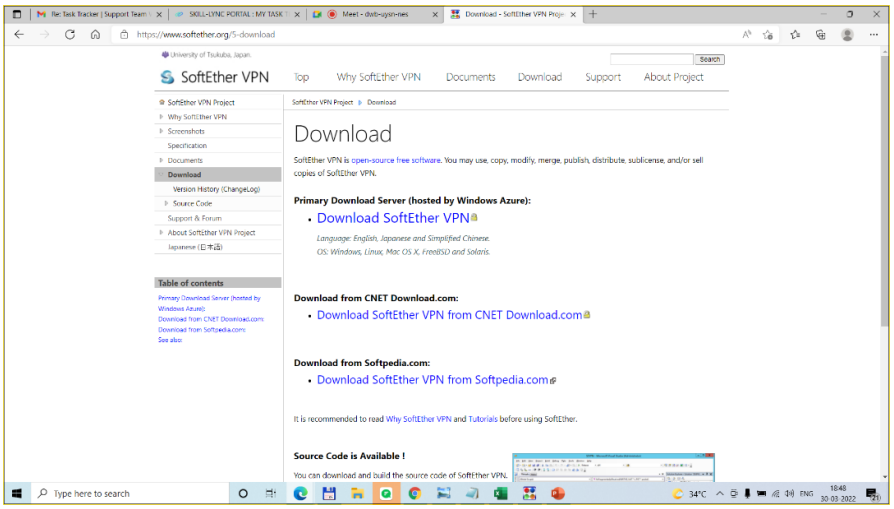
LDRA's tool site supports different programming languages that include:

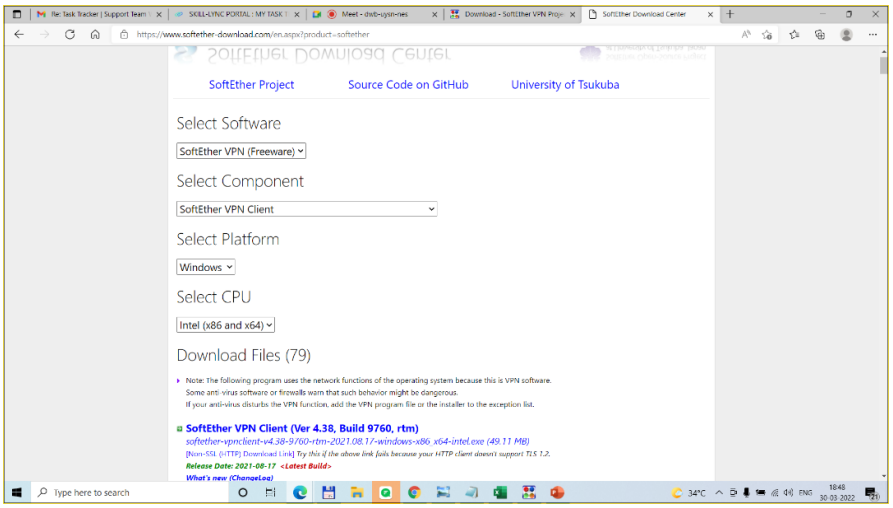
- C
- C++
- Java
- Ada95

3.1.6 LDRA Tool Setup

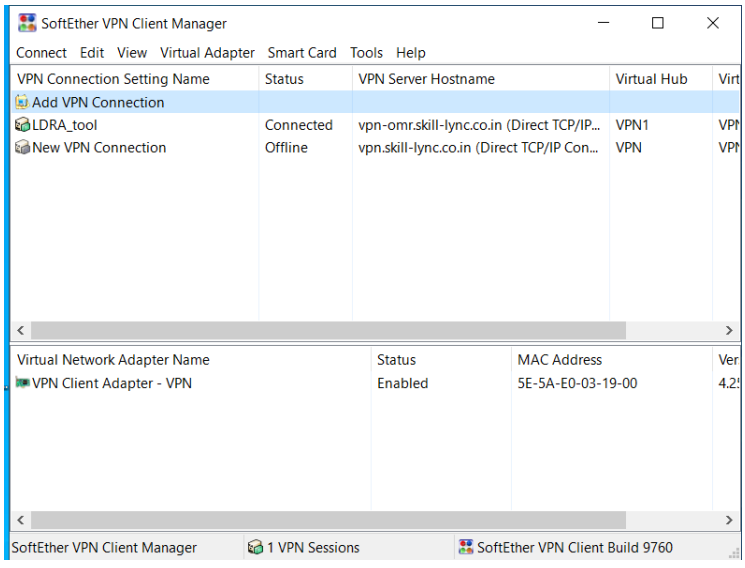
Step 1: Install VPN software.



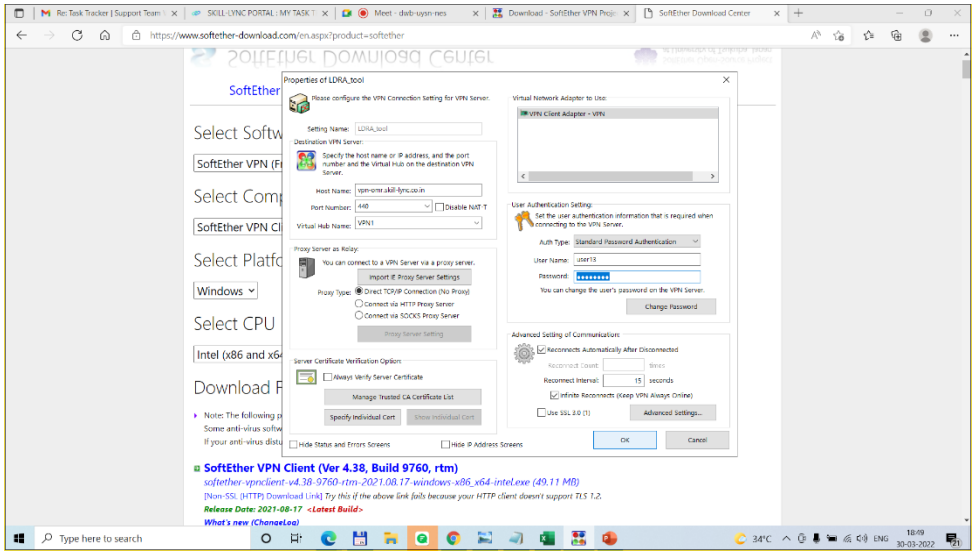




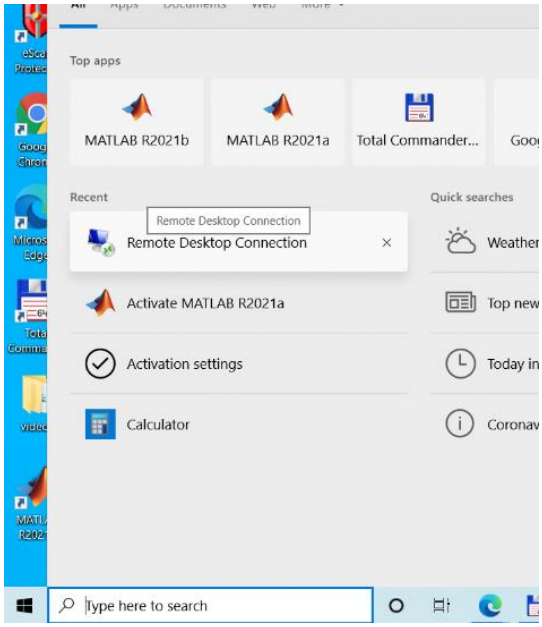
Step 2: Configure the VPN software**.**

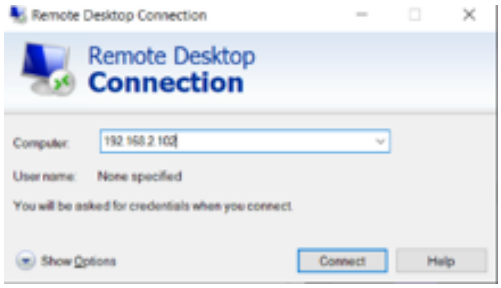


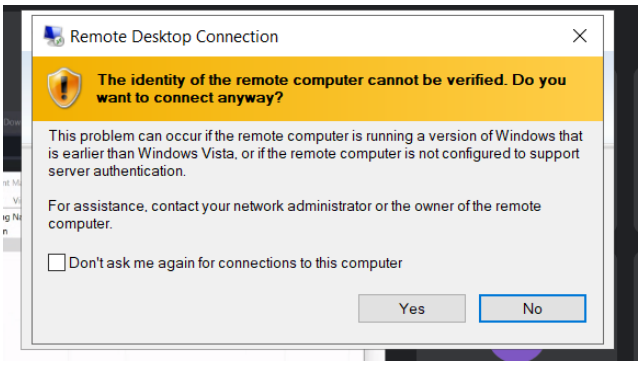
Password: Welcome@123



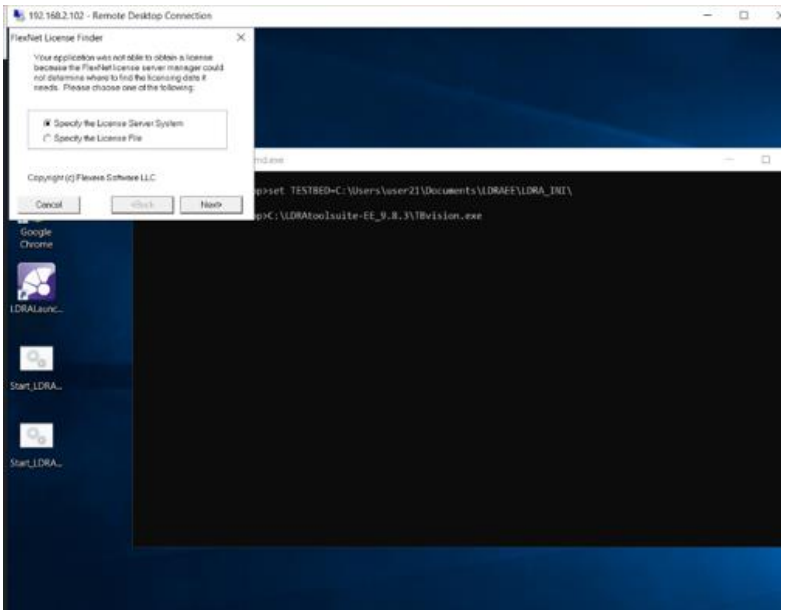
Step 3: Connect to remote desktop.

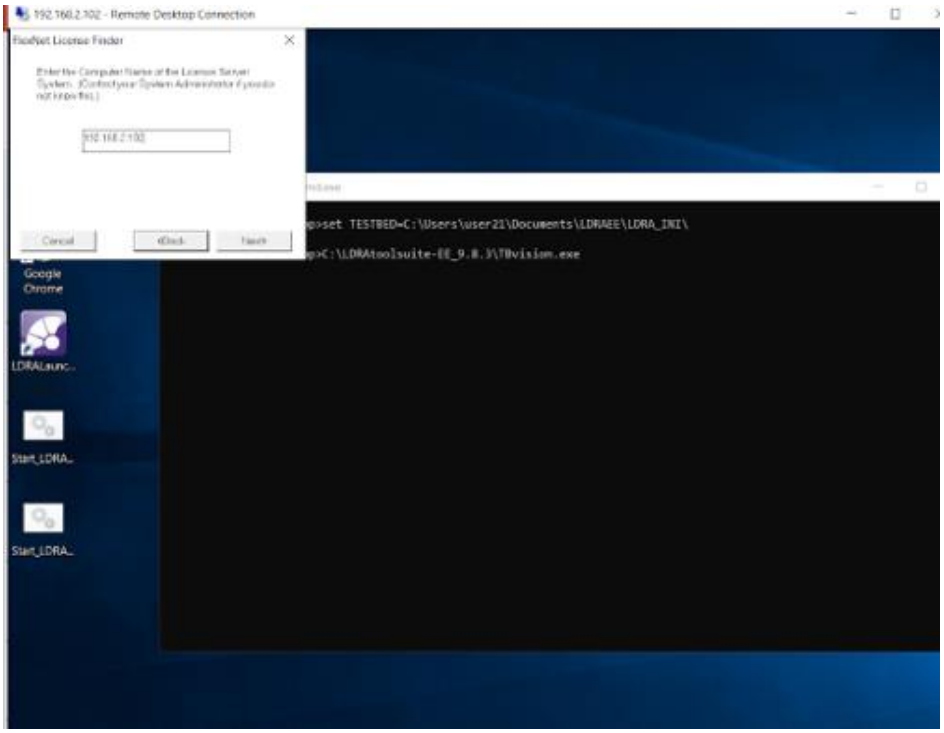


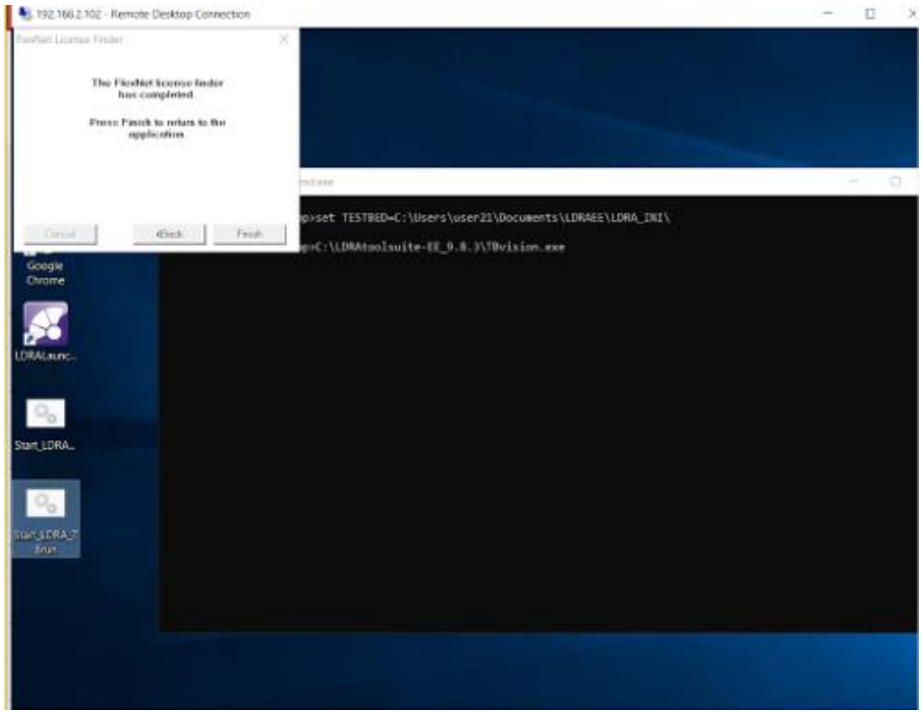




Step 4: Run LDRA tool set.

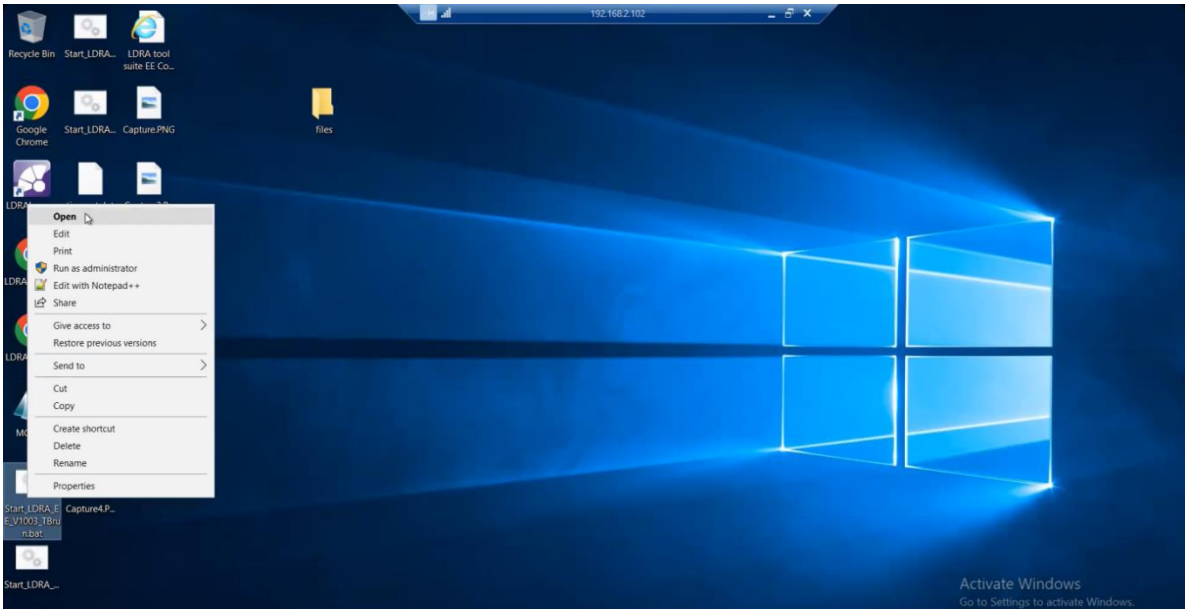


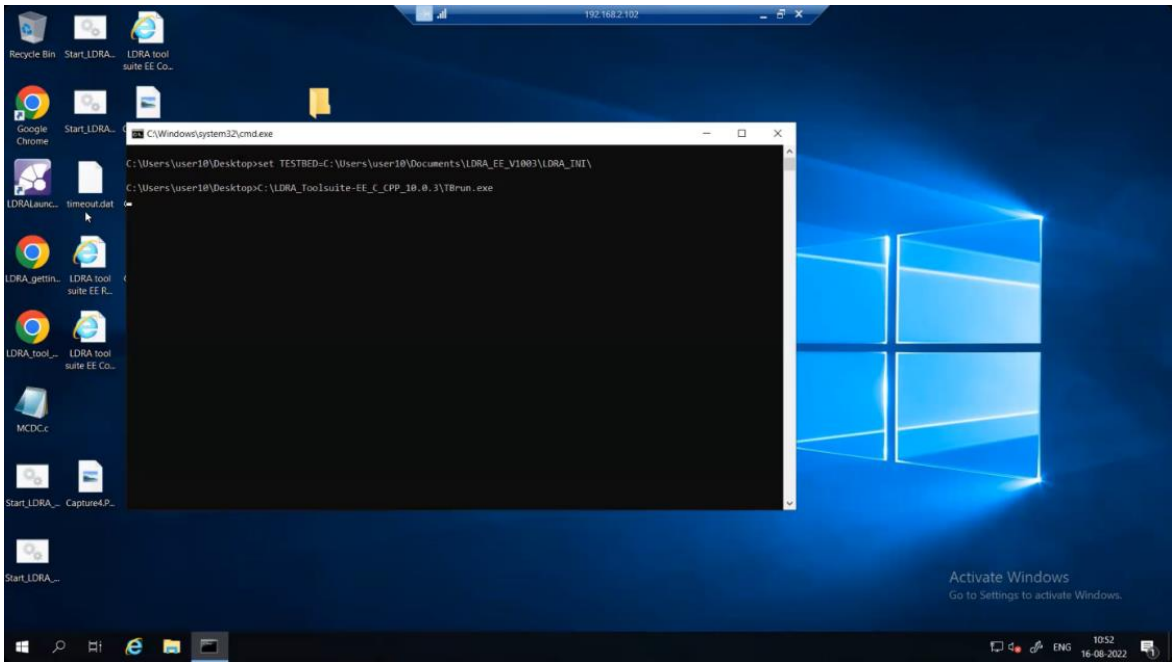

192.168.2.102 - Remote Desktop Connection
FlexNet License Finder
Cancel
<Back
Next>
set TESTBED=C:\Users\user21\Documents\LDRAEE\LDRA_INI\
C:\LDRAtoolsuite-EE_9.8.3\TBvision.exe
Google Chrome
LDRALaunc...
Start_LDRA...
Start_LDRA...


192.168.2.102 - Remote Desktop Connection
The FlexNet license finder has completed.
Press Finish to return to the application.
Cancel
<Back
Finish
set TESTBED=C:\Users\user21\Documents\LDRAEE\LDRA_INI\
C:\LDRAtoolsuite-EE_9.8.3\TBvision.exe
Google Chrome
LDRALaunc...
Start_LDRA...
Start_LDRA_3...

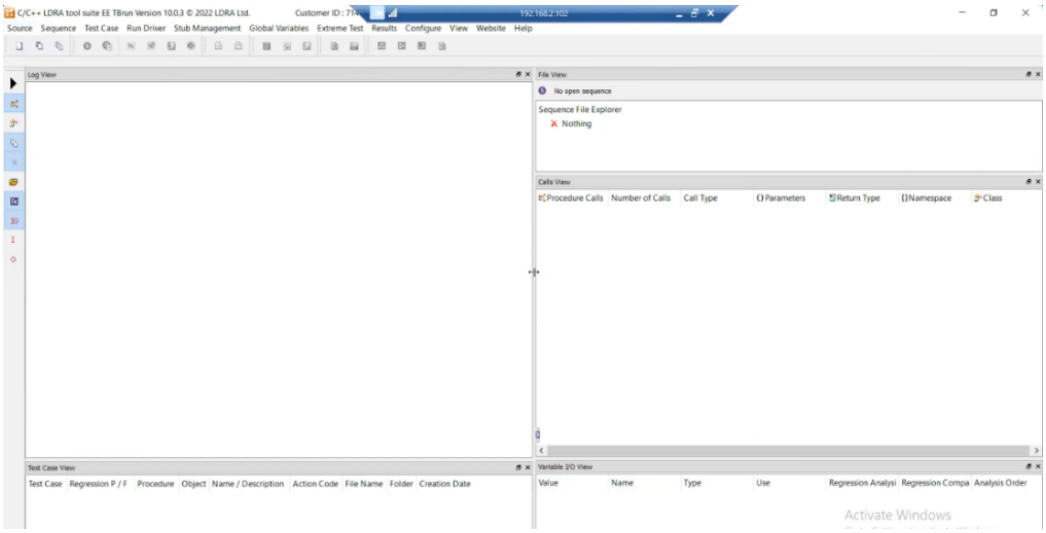
3.1.7 Static Code Analysis using LDRA Tool

Step 1: Open LDRA Tool using remote desktop connection.

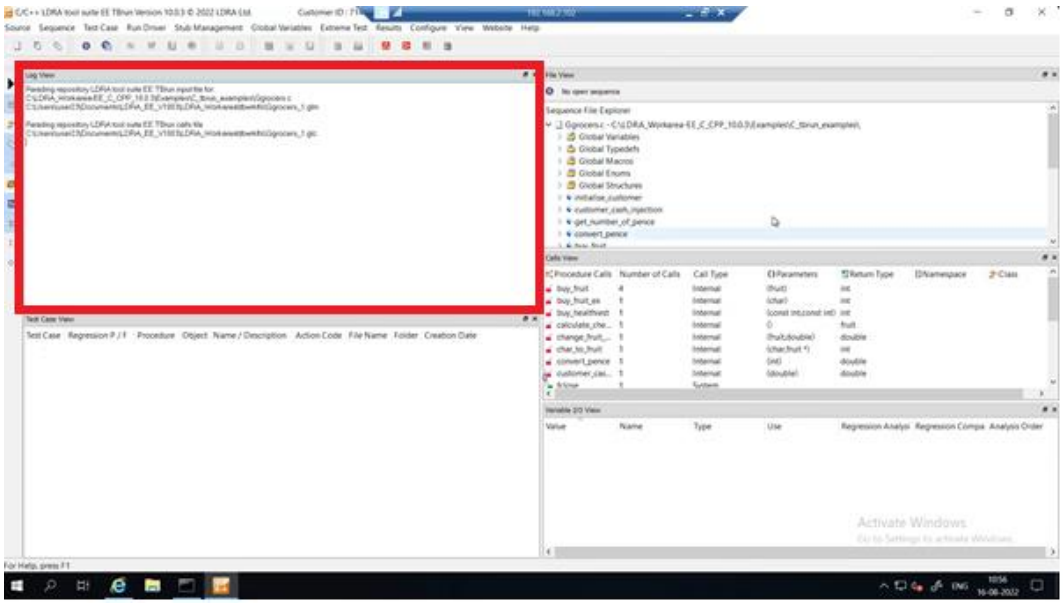


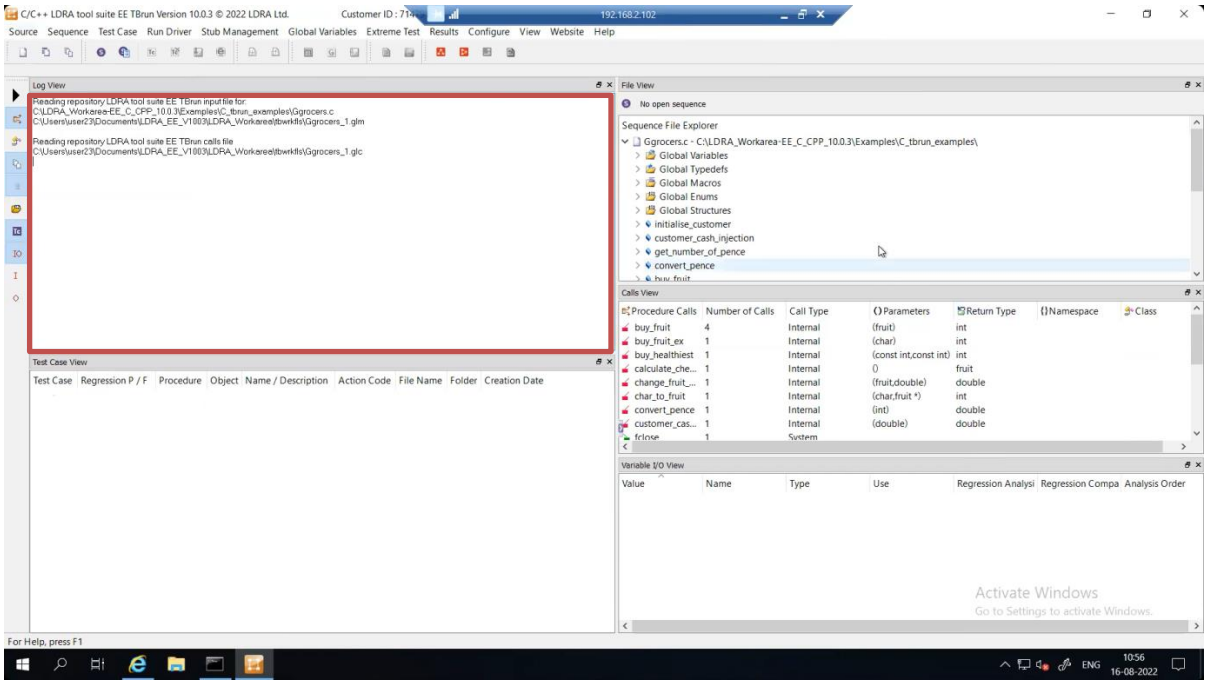


The LDRA workspace will look like this.

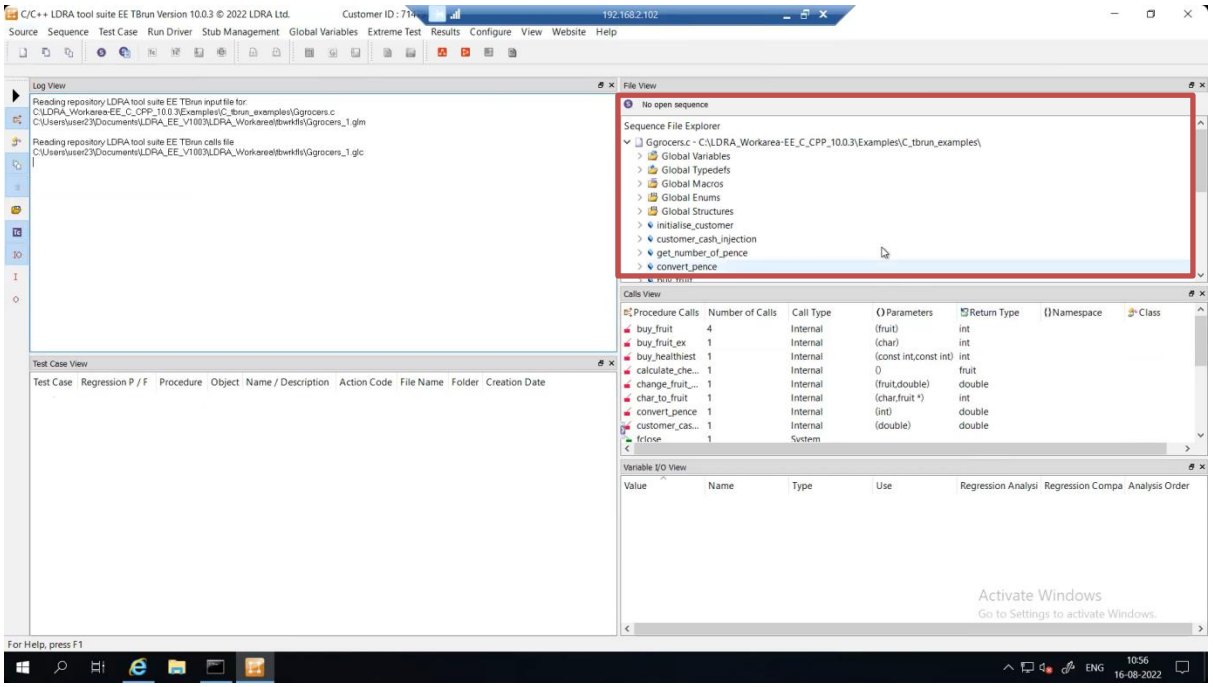


The log view gives the information regarding the test process on file.

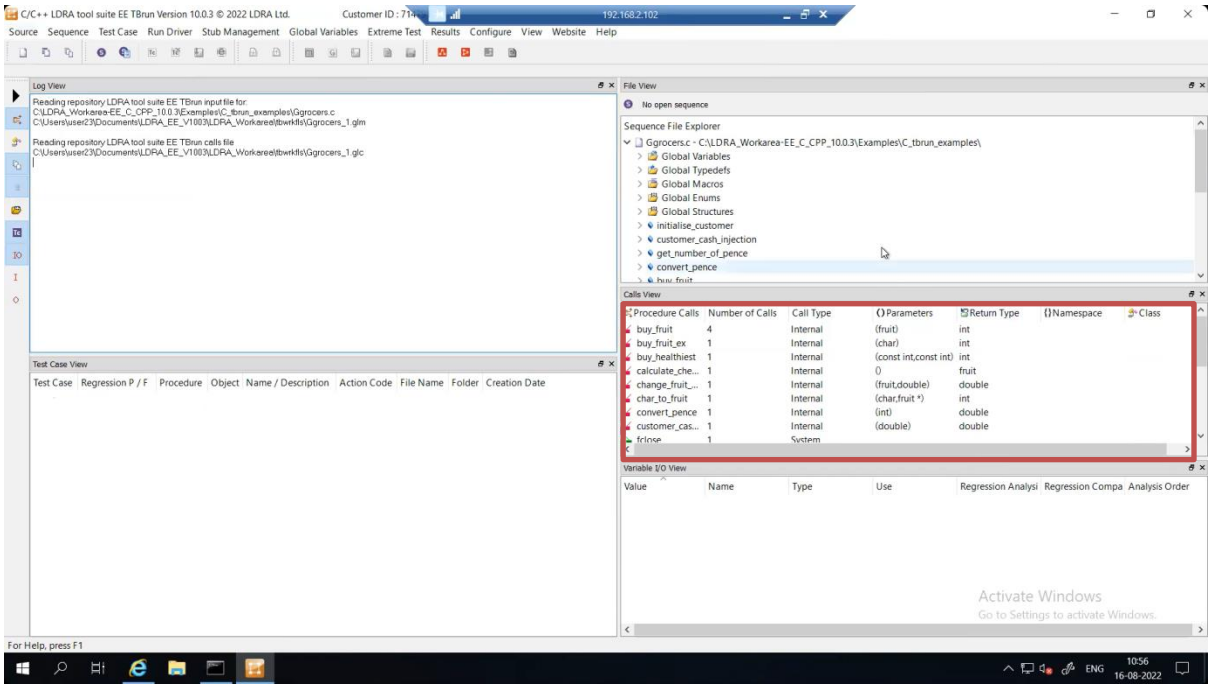




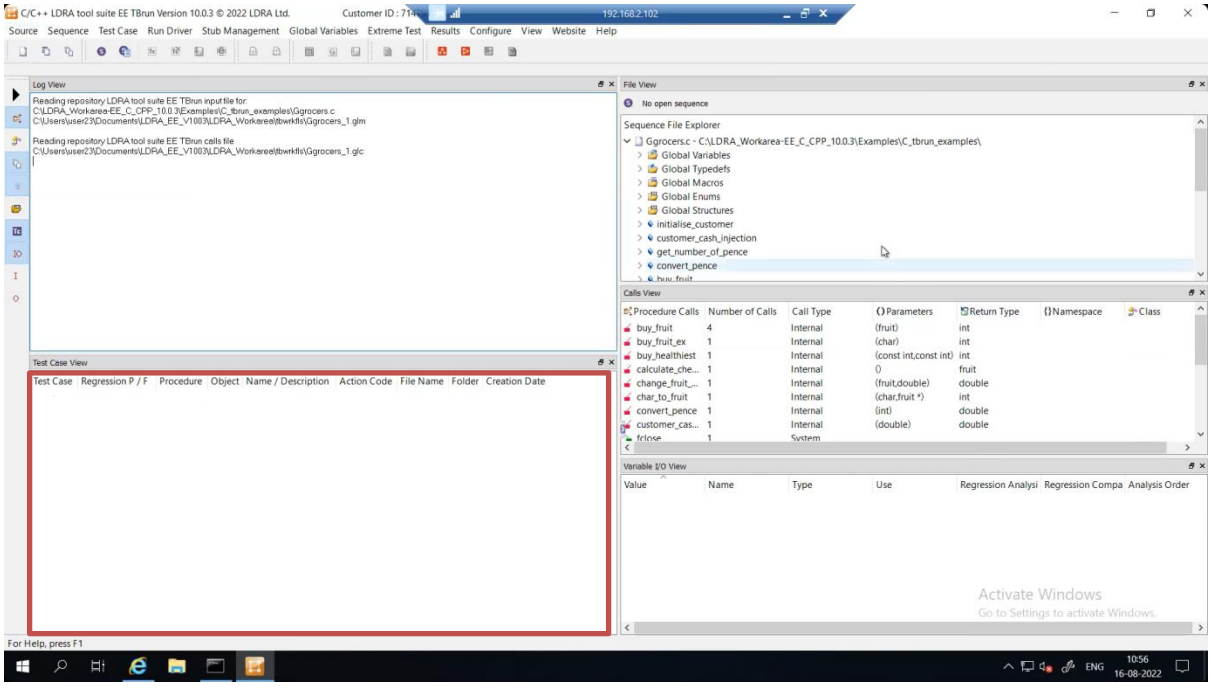
File View gives the information about the global variables, macros; Enum, Structures and all the sub function are present in the test file.



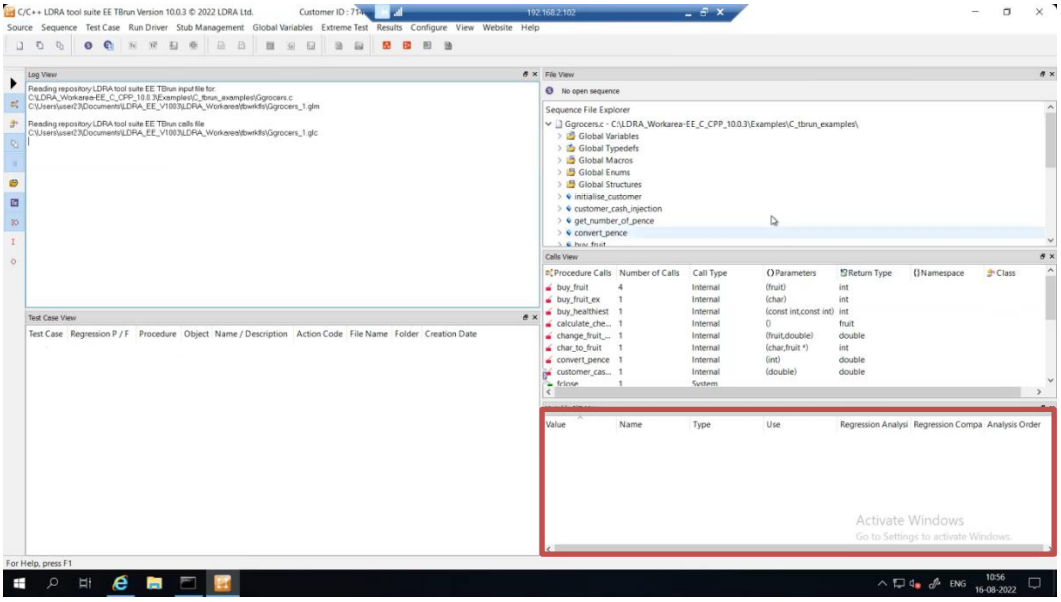
Call View gives the details of the number of function calls present in the test file.



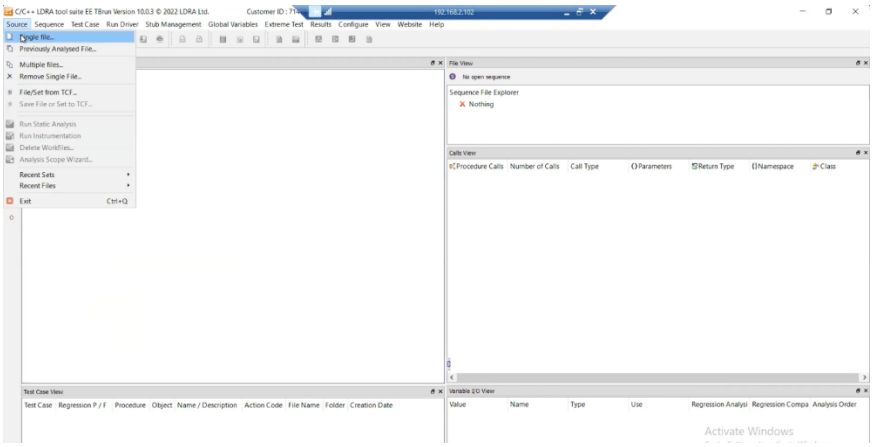
Test Case View gives the information about the number of test case and their regression status and their details.



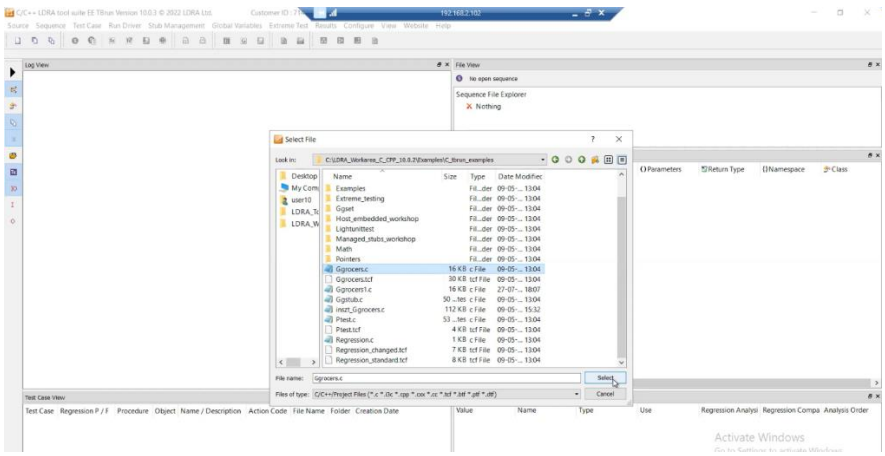
Variable I/O view allows users to enter input for variables in the function while we do manual testing during automatic testing the values for input output variables will be displayed.



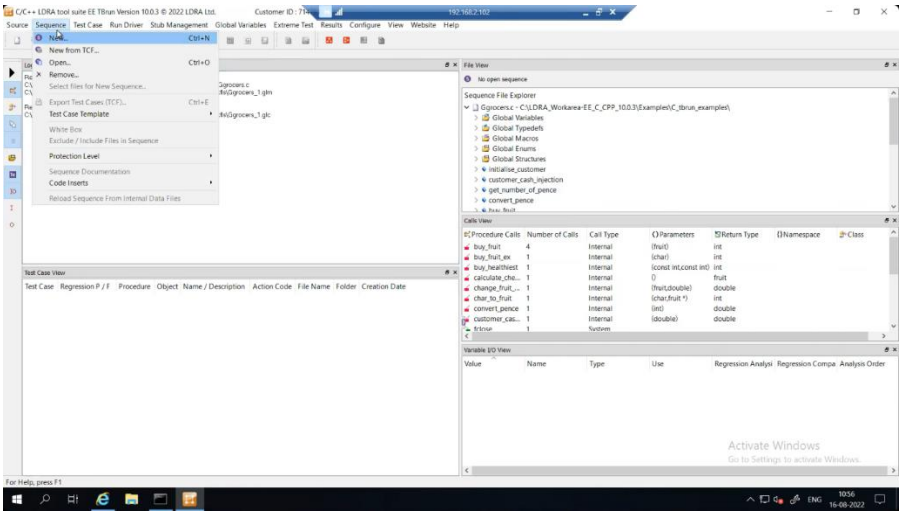
Step 2: Load the single file from source.



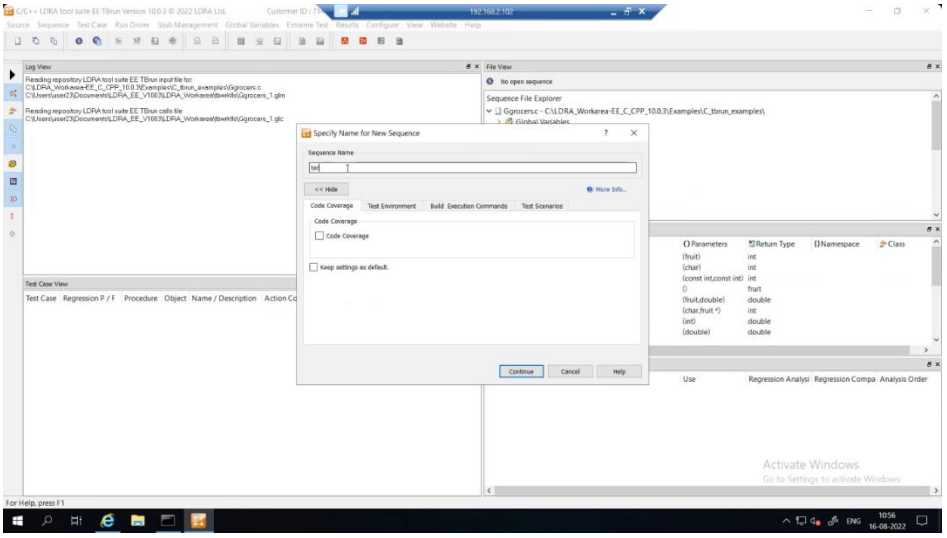
Select the .c file which does need to be tested.

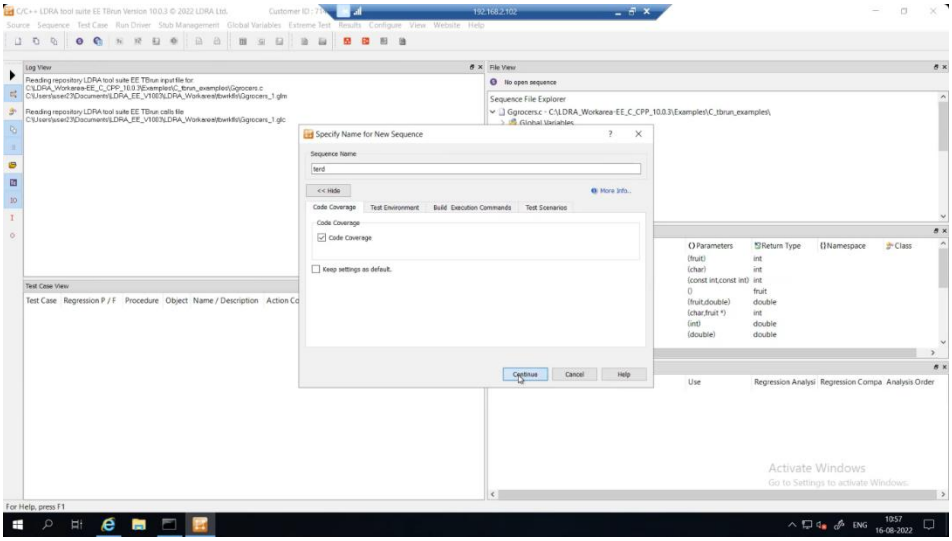


Step 3: Create a new sequence for the test.

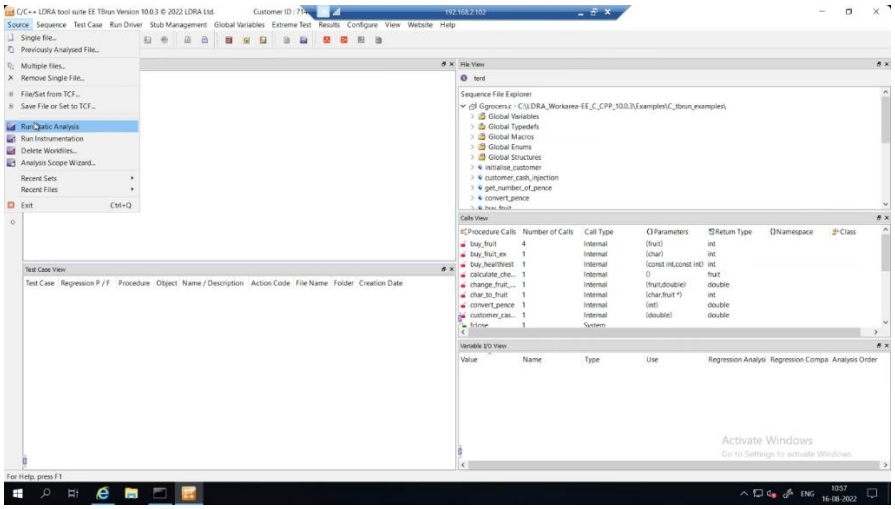


Give the name for the sequence and enable the code coverage then to click continue.

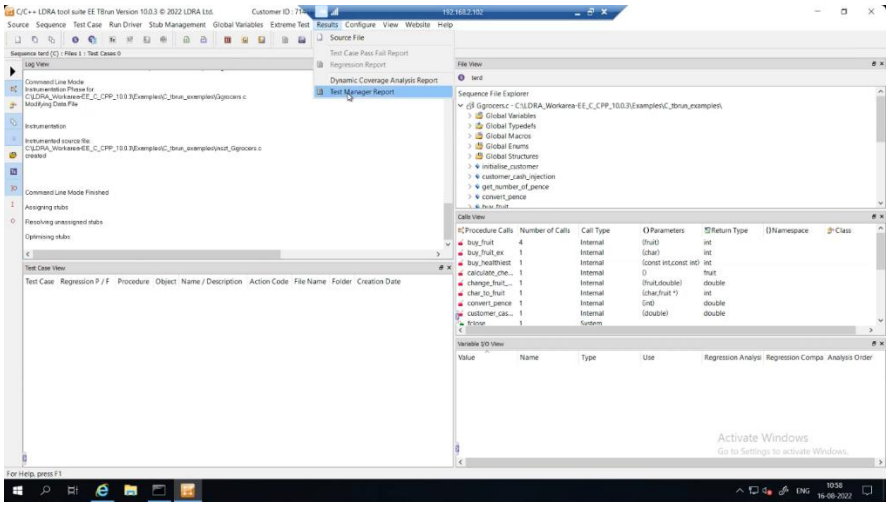




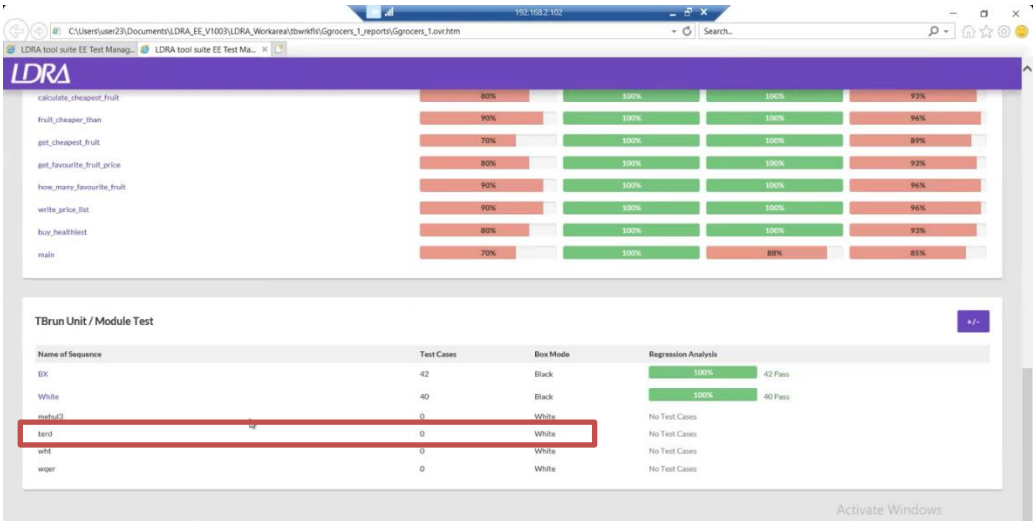
Step 4: Click Run Static Analysis from source.



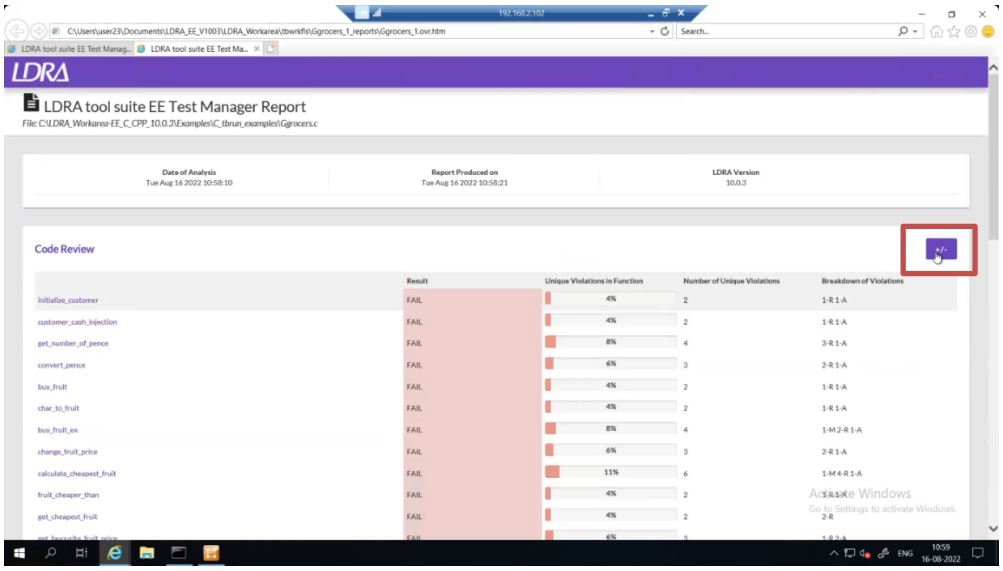
Step 5: Click on Test Manager Report from Results

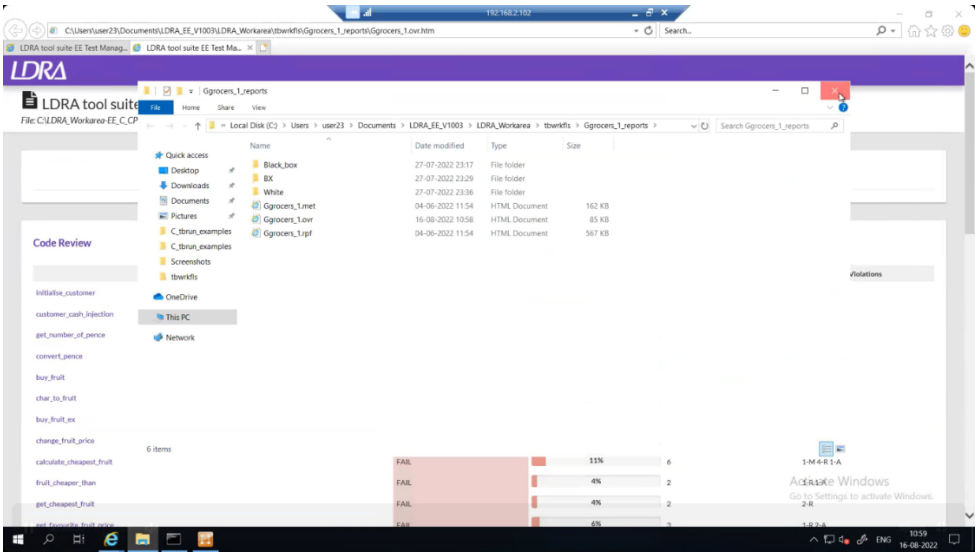


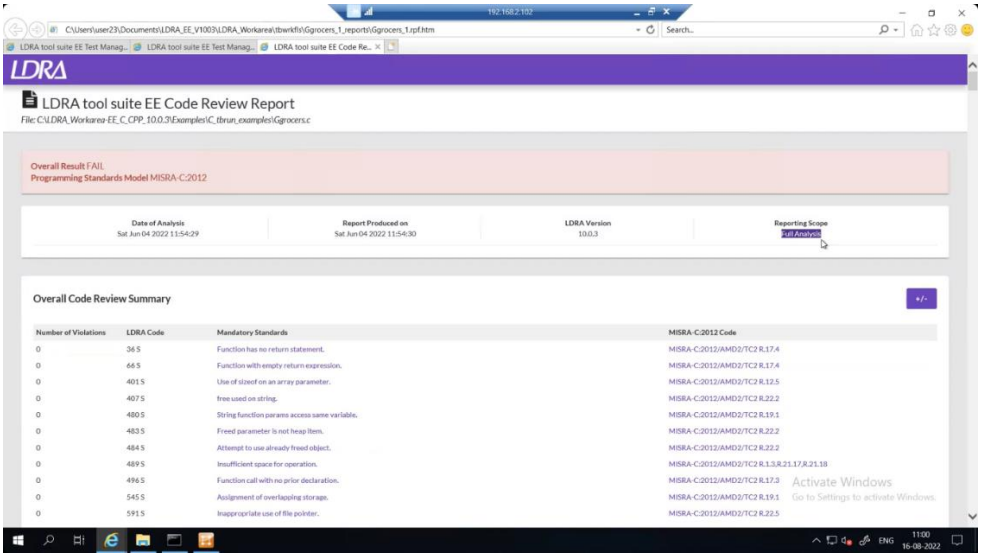
Step 6: Reports will appear as HTML file.

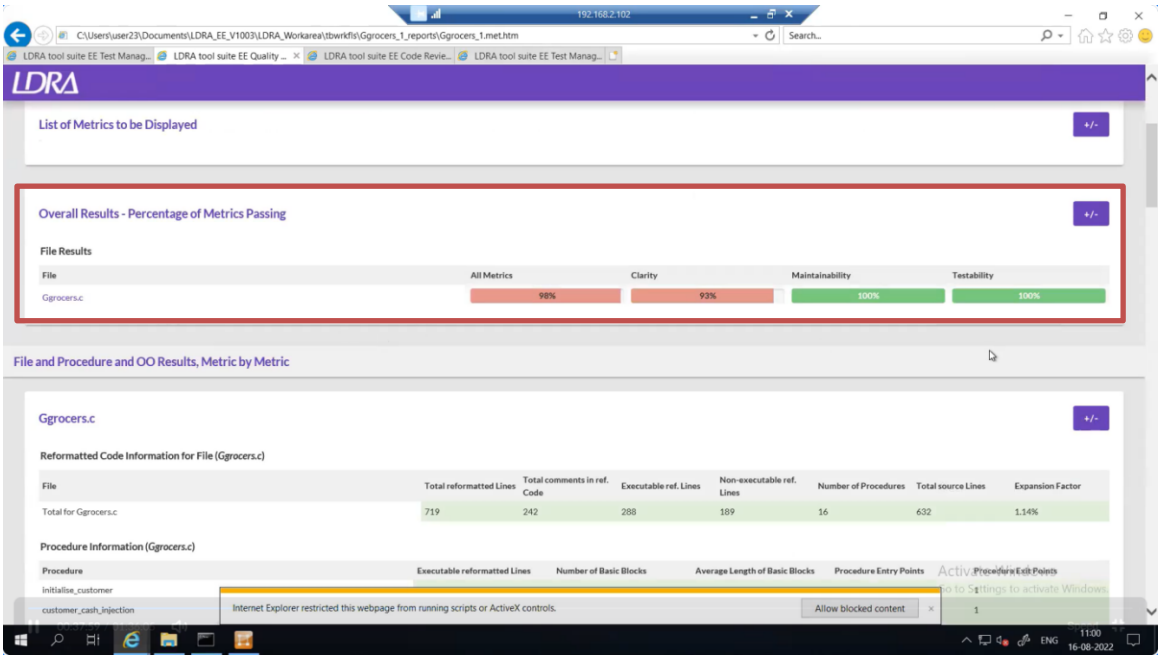


To get all report click on +/-

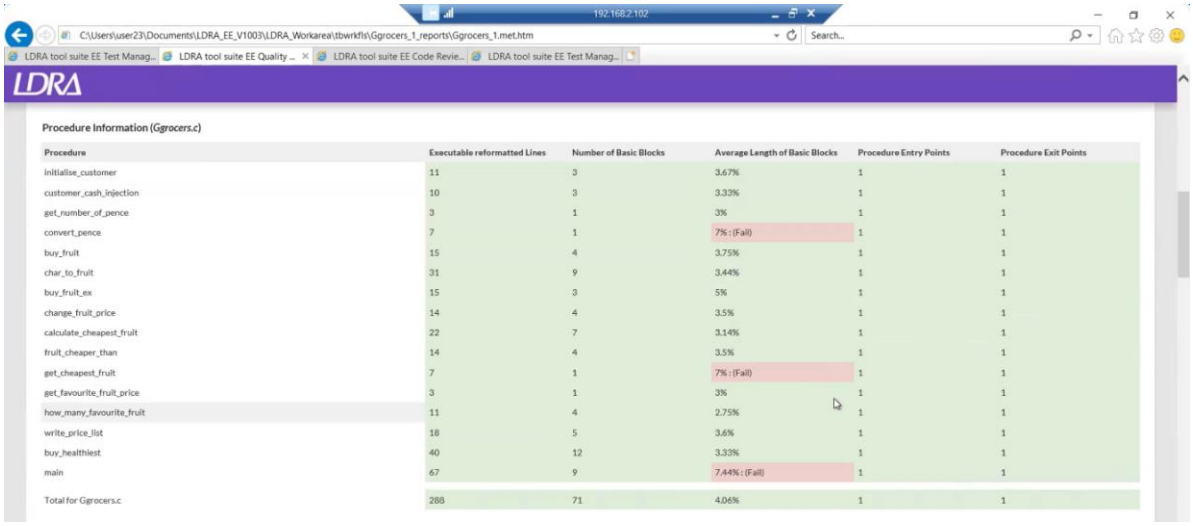








To view the details of the function, click on the particular function to see the report as given below.



3.1.8 What is Dynamic Testing?

Under Dynamic Testing, a code is executed. It checks for functional behavior of the software system, memory/CPU usage and overall performance of the system. Hence the name “Dynamic”. The main objective of this testing is to confirm that the software product works in conformance with the business requirements. This testing is also called an Execution technique or validation testing. Dynamic testing executes the software and validates the output with the expected outcome. Dynamic testing is performed at all levels of testing, and it can be either black or white box testing.

3.1.9 Difference between Static and Dynamic Testing

Static testing is about the prevention of defects whereas dynamic testing is about finding and fixing the defects. Static testing does the verification process while dynamic testing does the validation process. Static testing is performed before compilation whereas dynamic testing is performed after compilation.

Static Testing	Dynamic Testing
Testing was done without executing the program This testing does the verification process	Testing is done by executing the program Dynamic testing does the validation process
Static testing is about prevention of defects	Dynamic testing is about finding and fixing the defects
Static testing gives an assessment of code and documentation	Dynamic testing gives bugs/bottlenecks in the software system
Static testing involves a checklist and process to be followed	Dynamic testing involves test cases for execution
This testing can be performed before compilation	Dynamic testing is performed after compilation
Static testing covers the structural and statement coverage testing Cost of finding defects and fixing is less	Dynamic testing techniques are Boundary Value Analysis & Equivalence Partitioning Cost of finding and fixing defects is high
Return on investment will be high as this process involved at an early stage	Return on investment will be low as this process involves after the development phase
More reviews comments are highly recommended for good quality	More defects are highly recommended for good quality
Requires loads of meetings	Comparatively requires lesser meetings

Summary

In this unit, we learned about the following concepts:

1. Basics of static code analysis
2. Uses of static code analysis
3. Basics of dynamic testing
4. Difference between static and dynamic testing
5. LDRA tool suite for static code analysis of the source code

References

https://www.guru99.com/static-dynamic-testing.html#:~:text=Static%20testing%20is%20about%20the,testing%20is%20performed%20after%20compilation

https://allabouttesting.org/quick-review-ldra-tool-suite/

UNIT 3.2: Review Process

Unit Objectives

At the end of this unit, you will be able to:

1. Summarize the activities of the work product review process
2. Recognize the different roles and responsibilities in a formal review
3. Explain the differences between different review types: informal review, walkthrough, technical review, and inspection
4. Apply a review technique to a work product to find defects
5. Explain the factors that contribute to a successful review

3.2.1 What is Review Process?

The formality of a review process is related to factors such as the software development lifecycle model, the maturity of the development process, the complexity of the work product to be reviewed, any legal or regulatory requirements, and/or the need for an audit trail. The focus of a review depends on the agreed objectives of the review (e.g., finding defects, gaining understanding, educating participants such as testers and new team members, or discussing and deciding by consensus). The ISO standard (ISO/IEC 20246) contains more in-depth descriptions of the review process for work products, including roles and review techniques.

Review Process:

Reviews vary from informal to formal. Informal reviews are characterized by not following a defined process and not having formal, documented output. Formal reviews are characterized by team participation, documented results of the review, and documented procedures for conducting the review.

3.2.2 Work Product Review Process

The review process comprises the following main activities:

- Planning
- Initiate review
- Individual review (i.e., individual preparation)
- Issue communication and analysis
- Fixing and reporting

Planning:

Defining the scope, which includes the purpose of the review, what documents or parts of documents to review, and the quality characteristics to be evaluated. Estimating effort and timeframe. Identifying review characteristics such as the review type with roles, activities, and checklists, selecting the people to participate in the review and allocating roles, defining the entry and exit criteria for more formal review types (e.g., inspections). Checking that entry criteria are met (for more formal review types).

Initiate Review:
Distributing the work product (physically or by electronic means) and other material, such as issue log forms, checklists, and related work products, explaining the scope, objectives, process, roles, and work products to the participants, and answering any questions that participants may have about the review

Individual Review:
Reviewing all or part of the work product and noting potential defects, recommendations, and questions.

Issue Communication and Analysis:
Communicating identified potential defects (e.g., in a review meeting), analyzing potential defects, assigning ownership and status to them, evaluating and documenting quality characteristics, and evaluating the review findings against the exit criteria to make a review decision (reject; major changes are needed; accept, possibly with minor changes).

Fixing and Reporting:
Creating defect reports for those findings that require changes, fixing defects found (typically done by the author) in the work product reviewed, communicating defects to the appropriate person or team (when found in a work product related to the work product reviewed), recording the updated status of defects (in formal reviews), potentially including the agreement of the comment originator, gathering metrics (for more formal review types), checking that exit criteria are met (for more formal review types), and accepting the work product when the exit criteria are reached.

3.2.3 Roles and Responsibilities in a Formal Review

A typical formal review will include the following roles:

- Author
- Management
- Facilitator (also known as a moderator)
- Review leader
- Reviewers
- Scribe (or recorder)

Author:
The author creates the work product under review and fixes defects in the work product under review (if necessary).

Fig 3.2.3.1 Author

Management:

Management is responsible for review planning. It decides on the execution of reviews and assigns staff, budget, and time. It also monitors ongoing cost-effectiveness and executes control decisions in the event of inadequate outcomes.

Fig 3.2.3.2 Management

Facilitator:

Facilitators ensure the effective running of review meetings (when held). Mediates, if necessary, between the various points of view and is often the person upon whom the success of the review depends.

Fig 3.2.3.3 Facilitator

Review Leader:

Review leaders take overall responsibility for the review, decide who will be involved, and organize when and where it will take place.

Fig 3.2.3.4 Review Leader

Reviewers:

Reviewers may be subject matter experts, persons working on the project, stakeholders with an interest in the work product, and/or individuals with specific technical or business backgrounds. They identify potential defects in the work product under review and may represent different perspectives (e.g., tester, programmer, user, operator, business analyst, usability expert, etc.).

Fig 3.2.3.5 Reviewers

Scribe:
Compiles potential flaws discovered during the individual review activity and records new potential defects, open points, and decisions from the review meeting (when held).

Fig 3.2.3.6 Scribe

In some review types, one person may play more than one role, and the actions associated with each role may also vary based on review type. In addition, with the advent of tools to support the review process, especially the logging of defects, open points, and decisions, there is often no need for a scribe. Further, more detailed roles are possible, as described in the ISO standard (ISO/IEC 20246).

3.2.4 Review Types

Although reviews can be used for various purposes, one of the main objectives is to uncover defects. All review types can aid in defect detection, and the selected review type should be based on the needs of the project, available resources, product type and risks, business domain, and company culture, among other selection criteria. Reviews can be classified according to various attributes.

3.2.4.1 Informal Review

Examples of informal review is buddy check, pairing, and pair review. The main purpose of an informal review is to detect potential defects. Possible additional purposes include generating new ideas or solutions and quickly solving minor problems. It is not based on a formal, documented process. It may not involve a review meeting. It may be performed by a colleague of the author (buddy check) or by more people. The results may be documented. It varies in usefulness depending on the reviewers. The use of checklists is optional. It is very commonly used in agile development.

3.2.4.2 Walkthrough

The main purpose of a walkthrough is to find defects, improve the software product, consider alternative implementations, and evaluate conformance to standards and specifications. Possible additional purposes include exchanging ideas about techniques or style variations, training participants, and achieving consensus. Individual preparation before the review meeting is optional. The author of the work product usually leads a review meeting. The scribe is mandatory. The use of checklists is optional. It may take the form of scenarios, dry runs, or simulations. Potential defect logs and review reports may be produced. It may vary in practice from quite informal to very formal.

3.2.4.3 Technical Review

The primary goals are to reach agreement and identify potential flaws. Possible further purposes include evaluating quality and building confidence in the work product, generating new ideas, motivating, and enabling authors to improve future work products, and considering alternative implementations. Reviewers should be technical peers of the author and technical experts in the same or other disciplines. Individual preparation before the review meeting is required. A review meeting is optional and is ideally led by a trained facilitator (typically not the author). Scribe is mandatory, ideally not the author. The use of checklists is optional. Potential defect logs and review reports are typically produced.

3.2.4.4 Inspection

The main purposes are detecting potential defects, evaluating quality, building confidence in the work product, and preventing future similar defects through author learning and root cause analysis. Possible further purposes: motivating and enabling authors to improve future work products and the software development process; achieving consensus. It follows a defined process with formal, documented outputs based on rules and checklists. Roles and responsibilities in a formal review, which are mandatory, may include a dedicated reader (who reads the work product aloud during the review meeting). Individual preparation before the review meeting is required. Reviewers are either the author's peers or experts in other disciplines relevant to the work product. Specific entry and exit criteria are used. The scribe is mandatory. The review meeting is led by a trained facilitator (not the author). The author cannot act as the review leader, reader, or scribe. Potential defect logs and a review report are produced. Metrics are collected and used to improve the entire software development process, including the inspection process.

3.2.5 Apply a Review Technique to a Work Product to find Defects

There are a few review techniques that can be applied during the individual review (i.e., individual preparation) activity to uncover defects. These techniques can be used across the review types described above. The effectiveness of the techniques may differ depending on the type of review used. Examples of different individual review techniques for various review types are listed as ad hoc, checklist-based, scenario-based and dry-run-based, role-based, and perspective-based.

Ad hoc:

In an ad hoc review, reviewers are provided with little or no guidance on how this task should be performed. Reviewers often read the work product sequentially, identifying, and documenting issues as they encounter them. Ad hoc reviewing is a commonly used technique that requires little preparation. This technique is highly dependent on the reviewer's skills and may lead to many duplicate issues being reported by different reviewers.

Checklist-based:

A checklist-based review is a systematic technique whereby the reviewers detect issues based on checklists that are distributed at review initiation (e.g., by the facilitator). A review checklist consists of a set of questions based on potential defects, which may be derived from experience.

Checklists should be specific to the type of work product under review and should be maintained regularly to cover issue types missed in previous reviews. The main advantage of the checklist-based technique is its systematic coverage of typical defect types. Care should be taken not to simply follow the checklist in individual reviews but also to look for defects outside the checklist.

Scenarios-based and dry runs-based:

In a scenario-based review, reviewers are provided with structured guidelines on how to read through the work product. A scenario-based approach assists reviewers in conducting "dry runs" on the work product based on expected usage (if the work product is documented in a suitable format such as use cases). These scenarios provide reviewers with better guidelines on how to identify specific defect types than simple checklist entries. As with checklist-based reviews, in order not to miss other defect types (e.g., missing features), reviewers should not be constrained to the documented scenarios.

Role-based:

A role-based review is a technique in which the reviewers evaluate the work product from the perspective of individual stakeholder roles. Typical roles include specific end-user types (experienced, inexperienced, senior, child, etc.) and specific roles in the organization (user administrator, system administrator, performance tester, etc.).

Perspective-based:

In perspective-based reading, like a role-based review, reviewers take on different stakeholder viewpoints when individually reviewing. Typical stakeholder viewpoints include end users, marketing, designers, testers, and operations. Using different stakeholder viewpoints leads to more depth in individual reviewing and less duplication of issues across reviewers.

3.2.6 Factors that Contribute a Successful Review

Organizational success factors for reviews include:

Each review has clear objectives, defined during review planning, and used as measurable exit criteria. Review types are applied that are suitable to achieve the objectives and are appropriate to the type and level of software work products and participants. Any review techniques used, such as checklist-based or role-based reviewing, are suitable for effective defect identification in the work product to be reviewed. Any checklists used address the main risks and are up-to-date. Large documents are written and reviewed in small chunks so that quality control is exercised by providing authors early and frequent feedback on defects. Participants have adequate time to prepare. Reviews are scheduled with adequate notice. Management supports the review process (e.g., by incorporating adequate time for review activities in project schedules).

People related success factors for reviews include:

To meet the review objectives, the appropriate people must be involved, such as individuals with diverse skill sets or perspectives who may use the document as a work input. Testers are seen as valued reviewers who contribute to the review and learn about the work product, which enables them to prepare more effective tests and to prepare those tests earlier. Participants dedicate adequate time and attention to detail. Reviews are conducted in small chunks so that reviewers do not lose concentration during the individual review and/or the review meeting (when held). Defects found are acknowledged, appreciated, and handled objectively. The meeting is well managed, so that participants consider it a valuable use of their time. The review is conducted in an atmosphere of trust, and the outcome will not be used for the evaluation of the participants. Participants avoid body

language and behaviors that might indicate boredom, exasperation, or hostility toward other participants. Adequate training is provided, especially for more formal review types such as inspections. A culture of learning and process improvement is promoted.

Summary

In this unit, we learned about the following concepts:

1. The different review processes involved in the software testing
2. How review technique helps to identify the defects in the product
3. Roles and responsibilities of formal review process

Reference

https://medium.com/@HugoSaxTavares/istqb-foundation-level-syllabus-chapter-3-of-6-static-testing-813868d2460c

Summary

In this module "Static Testing Basics", we learned about the following concepts:

1. The different static testing techniques
2. Identify errors in the code using the static testing technique
3. Difference between static and dynamic testing
4. The work product review process
5. Roles and responsibilities in formal view
6. Finding defects in the code using product review techniques
7. Identifying factors that influence a successful review process

4. Test Techniques

Key Learning Outcomes

At the end of this module, you will be able to:

1. Explain the difference between white box and black box testing
2. Explain common pitfalls in experienced based technique
3. Explain the characteristics of testability
4. List the software testing guidelines
5. Explain about black box testing in embedded system
6. List the different techniques used in the black box testing techniques
7. Explain about various levels white box testing in embedded System
8. Explain different techniques used in the white box testing
9. Explain about various types of experienced based testing techniques
10. Explain about different test design techniques

UNIT 4.1: Categories of Test Techniques

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the characteristics, commonalities, and differences between black-box test techniques, white-box test techniques, and experience-based test techniques

4.1.1 Software Testing

Software testing determines the correctness, completeness, and quality of the software being developed. Institute of Electrical and Electronics Engineers (IEEE) defines testing as "the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results." Software testing is closely related to the term's "verification" and "validation." Verification refers to the process of ensuring that the software is developed according to its specifications. For verification, techniques like reviews, analysis, inspections, and walkthroughs are commonly used. While validation refers to the process of checking that the developed software meets the requirements specified by the user.

Verification: Is the software being developed in the right way?

Validation: Is the right software being developed?

Software testing is performed either manually or by using automated tools to make sure that the software is functioning in accordance with the user's requirements.

4.1.1.1 Advantages of Software Testing

It removes errors that prevent software from producing outputs according to user requirements and remove errors that lead to software failure. It also ensures that the software conforms to business as well as user needs. It ensures that the software is developed according to user requirements. It improves the quality of the software by removing as many possible errors from it as possible.

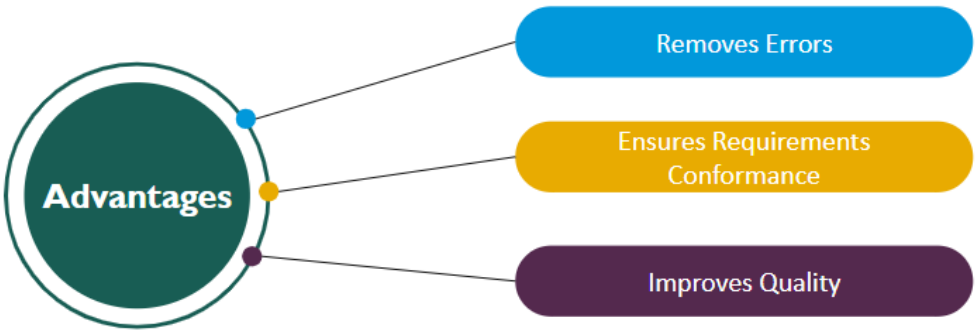


Fig 4.1.1.1 Advantages of Software Testing

4.1.1.2 Role of Testing in Various Phases of SDLC

Software Development Phase	Testing
Requirements specification	To identify the test strategy. To ensure that requirements are met. To create functional test conditions.
Design	To check the consistency of the design with the requirements. To check the sufficiency of the design. To create structural and functional test conditions.
Coding	To check the consistency of implementation with the design. To check the sufficiency of implementation. To create structural and functional test conditions for programs and units
Testing	To check the sufficiency of the test plan. To test the application programs.
Installation and maintenance	To put the tested system into operation. To make changes to the system and retest the modified system.

4.1.1.3 Reasons for Errors in Software

Programming errors: Programmers can make mistakes while developing the source code.

Unclear requirements: Either the user is not clear about the desired requirements, or the developers are unable to understand the user's requirements in a clear and concise manner.

Software complexity: The greater the complexity of the software, the more the scope of committing an error (especially by an inexperienced developer)

Changing requirements: The users usually keep on changing their requirements, and it becomes difficult to handle such changes in the later stages of the development process. Therefore, there are chances of making mistakes while incorporating these changes into the software.

Time pressures: Maintaining the schedule of software projects is difficult. When deadlines are not met, the attempt to speed up the work causes errors.

Poorly documented code: If the code is not well documented or well written, then maintaining and modifying it becomes difficult. This causes errors.

4.1.1.4 Software Testing Guidelines

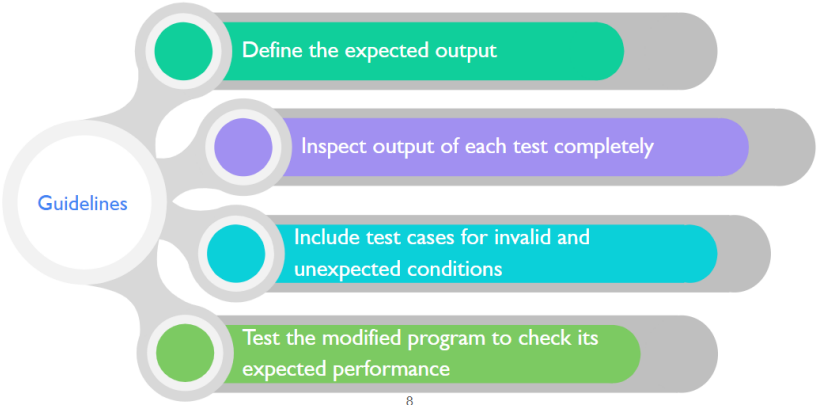


Fig 4.1.1.4 Software Testing Guidelines

Define the expected output: When programs are executed during testing, they may or may not produce the expected outputs due to different types of errors present in the software. To avoid this, it is necessary to define the expected output before software testing begins. Without knowledge of the expected results, testers may fail to detect an erroneous output.

Inspect the output of each test completely: Software testing should be performed once the software is complete to check its performance and functionality, as well as the occurrence of errors in various phases of software development.

Include test cases for invalid and unexpected conditions: Generally, software produces correct outputs when it is tested using accurate inputs. However, if unexpected input is given to the software, it may produce erroneous outputs. Hence, test cases that detect errors even when unexpected and incorrect inputs are specified should be developed.

Test the modified program to check its expected performance: Sometimes, when certain modifications are made to the software (like the addition of new functions), it is possible that the software produces unexpected outputs. Hence, it should be tested to verify that it performs in the expected manner even after modifications.

4.1.1.5 Characteristics of Testability

Easy to operate: High-quality software can be tested in a better manner. This is because if the software is designed and implemented with quality in mind, then comparatively fewer errors will be detected during the execution of tests.

Stability: Software becomes stable when changes made to it are controlled and existing tests can still be run.

Observability: Testers can easily identify whether the output generated for certain input is accurate simply by observing it.

Easy to understand: Software that is easy to understand can be tested in an efficient manner. Software can be properly understood by gathering maximum information about it. For example, to have proper knowledge of the software, its documentation can be used, which provides complete information about the software code, thereby increasing its clarity and making the testing easier.

Decomposability: By breaking software into independent modules, problems can be easily isolated, and the modules can be easily tested.

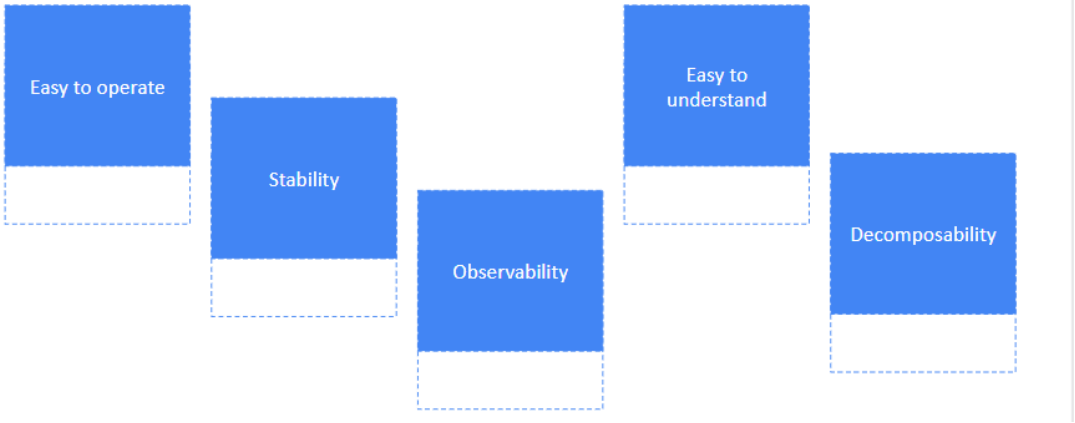


Fig 4.1.1.5 Characteristics of Testability

4.1.1.6 Characteristics of Software Test

High probability of detecting errors: To detect maximum errors, the tester should understand the software thoroughly and try to find the possible ways in which the software can fail. For example, in a program to divide two numbers, the possible way in which the program can fail is when 2 and 0 are given as inputs and 2 is to be divided by 0. In this case, a set of tests should be developed that can demonstrate an error in the division operator.

No redundancy: Resources and testing time are limited in the software development process. Thus, it is not beneficial to develop several tests that have the same intended purpose. Every test should have a distinct purpose.

Choose the most appropriate test: There can be different tests that have the same intent, but due to certain limitations, such as time and resource constraints, only a few of them are used. In such a case, the tests, which are likely to find a greater number of errors, should be considered.

Moderate: A test is considered good if it is neither too simple nor too complex. Many tests can be combined to form one test case. However, this can increase the complexity and leave many errors undetected. Hence, all tests should be performed separately.

4.1.1.7 Method of Testing

Black-box Testing: A method of testing that tests the functionality of the system without looking into the internal implementation/structures of the system. E.g., Testing the lock and unlock features of your mobile.

White-box Testing: A method of testing that may/may not test the functionality of the system by looking deeply into the internal implementation/structures (design, coding, etc.) of the system. E.g., Testing any embedded prototype, you designed by debugging the software.

4.1.1.8 Commonalities of Black Box and White Box Testing

They both help to achieve maximum bug elimination: Although one is performed at the end of the software development process and the other at every stage of the process, they both work to ensure that the application is free from bugs and performs its functions.

Detecting defects: Black box testing and white box testing detect defects in the overall behaviour of the application. Both testing types are performed by testers who interact with the software to discover if the software is working well. If any defects are identified, they both help to discover the root cause of those defects.

4.1.1.9 Difference Between Black Box and White Box Testing

Parameter	Black Box Testing	White Box Testing
Definition	It is a testing approach that is used to test the software without knowledge of the internal structure of the program or application.	It is a testing approach in which the internal structure is known to the tester.
Alias	It is also known as box testing, data-driven, or functional testing.	It is also called structural testing, clear box testing, code-based testing, or glass box testing.
Base of Testing	Testing is based on external expectations; the internal behaviour of the application is unknown.	Internal workings are known, and the tester can test accordingly.
Usage	This type of testing is ideal for higher levels of testing like system testing and acceptance testing.	Testing is best suited for lower-level testing such as unit or integration testing.
Automation	Testing and programming are dependent on each other, so it is tough to automate.	White box testing is easy to automate.
Objective	The main objective of this testing is to check the functionality of the system under test.	The main objective of White Box testing is to check the quality of the code.
Basis for test cases	After completing the requirement specification document, testing can begin.	Testing can start after preparing the detailed design document.
Tested by	Performed by the end user, developer, and tester.	Usually done by testers and developers.
Granularity	Granularity is low.	Granularity is high.
Time	It is less exhaustive and time-consuming.	Exhaustive and time-consuming method.
Code Access	Code access is not required for Black Box testing.	White box testing requires code access. Thereby, the code could be stolen if testing is outsourced.
Benefit	Well suited and efficient for large code segments.	It allows removing the extra lines of code, which can introduce hidden defects.
Techniques	Equivalence partitioning is a type of black box testing.	Statement coverage, branch coverage, and path coverage are White Box testing techniques.
	Equivalence partitioning divides input values into valid and invalid partitions and selects corresponding values from each partition of the test data.	Statement coverage validates whether every line of the code is executed at least once.
	Boundary value analysis	Branch coverage validates whether each branch is executed at least once.
	Checks boundaries for input values.	The path coverage method tests all the paths of the program.
Drawbacks	If you frequently modify the application, you must update the automation test script.	Automated test cases can become useless if the code base is rapidly changing.

4.1.2 Experienced Based Techniques

The experience-based testing technique is based on the skill and experience of the testers, experts, users, etc. It is conducted in an ad hoc manner because proper specifications are not available to test the applications. Here the tester depends on past experiences with the same technologies. According to his experience, the tester focuses on the critical areas of the software, such as the area's most frequently used by the customer or the area's most likely to fail. Experience some specification-based techniques that complement structure-based or design-based techniques. This technique is used for low-risk systems. This kind of testing is done even when there are no specifications or an inadequate specification list.

4.1.2.1 Types of Experienced Based Techniques

- Error Guessing Techniques
- Exploratory Techniques
- Checklist Based Testing
- Fault Attack Testing

Error Guessing

The tester applies his experience to guess the areas in the application that are prone to error.

Exploratory Testing

As the name implies, the tester explores the application and uses his experience to navigate through its different functionalities.

Checklist Based Testing

In this technique, we apply a tester's experience to create a checklist of different functionalities and use cases for testing.

4.1.2.2 When Should We Use Experience Based Technique?

Experience based techniques should be used when requirements and specifications are not available, when requirements are inadequate, when knowledge of the software product is limited, and when time constraints prevent following a structured approach.

4.1.2.3 Scenarios to Avoid

While we can always use this testing in conjunction with structured techniques, in some cases we cannot. At times, there are contractual requirements where we need to present test coverage and specific test matrices. Experience-based techniques cannot measure the test coverage. Hence, we should avoid them in such cases.

4.1.2.4 Common Pit Falls of Experienced Based Testing

As the name suggests, this testing technique is solely based on the experience of the tester. The quality of testing depends on the tester, and it would differ from person to person. If the tester's experience is insufficient, this approach may result in a very poorly tested application, which may lead

to bugs in the application. In some cases, the tester has experience, but the domain is new. For example, the application is in the banking domain, but the tester has worked on e-commerce applications. In such cases, experience-based testing would not work as well. Hence, it is extremely important that we use this technique when a tester has been working in the same domain or on the application for a long time.

Summary

In this unit, we learned about the following concepts:

1. Software Testing
2. Guidelines for software testing
3. Characteristics of software testing
4. Methods of testing
5. Difference between white box and black box testing
6. Experience-based Techniques

Reference

https://ecomputernotes.com/software-engineering/software-testing
https://www.orientsoftware.com/blog/black-box-and-white-box-software-testing/
https://www.guru99.com/back-box-vs-white-box-testing.html
https://www.toolsqa.com/software-testing/ISTQB/experience-based-testing-technique

UNIT 4.2: Black-box Test Techniques

Unit Objectives

At the end of this unit, you will be able to:

1. Apply equivalence partitioning to derive test cases from given requirements
2. Apply boundary value analysis to derive test cases from given requirements
3. Apply decision table testing to derive test cases from given requirements
4. Apply state transition testing to derive test cases from given requirements
5. Explain how to derive test cases from a use case

4.2.1 What is Black-box Testing?

Black box testing is a technique of software testing that examines the functionality of the software without peering into its internal structure or coding. The primary source of black box testing is a list of requirements that is stated by the customer. In this method, the tester selects a function and gives an input value to examine its functionality, then checks whether the function is giving the expected output or not. If the function produces the correct output, then it passes in testing; otherwise, it fails. The test team reports the result to the development team and then tests the next function. After completing the testing of all functions and finding severe problems, it is given back to the development team for correction.

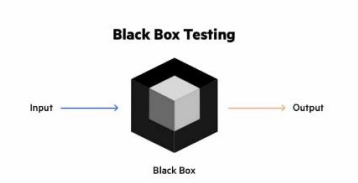
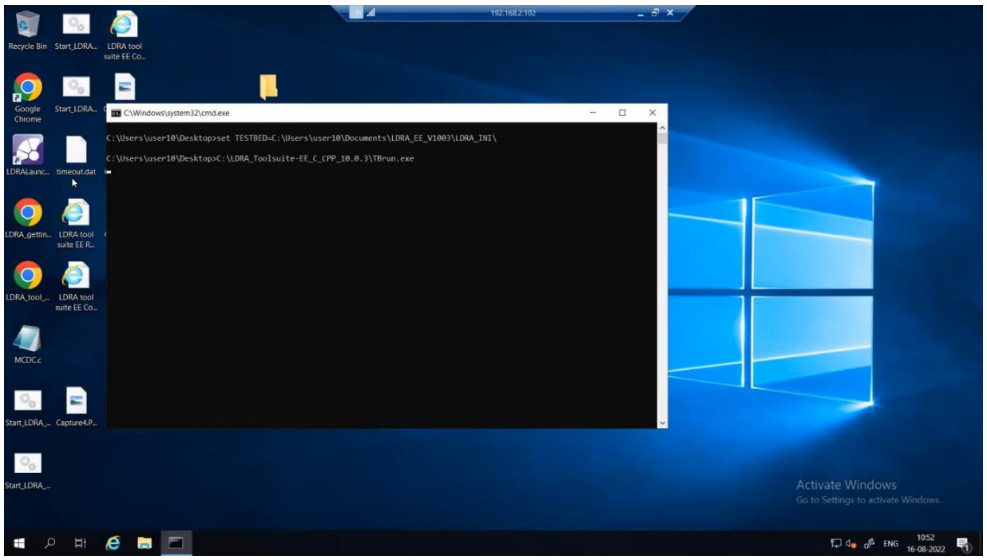


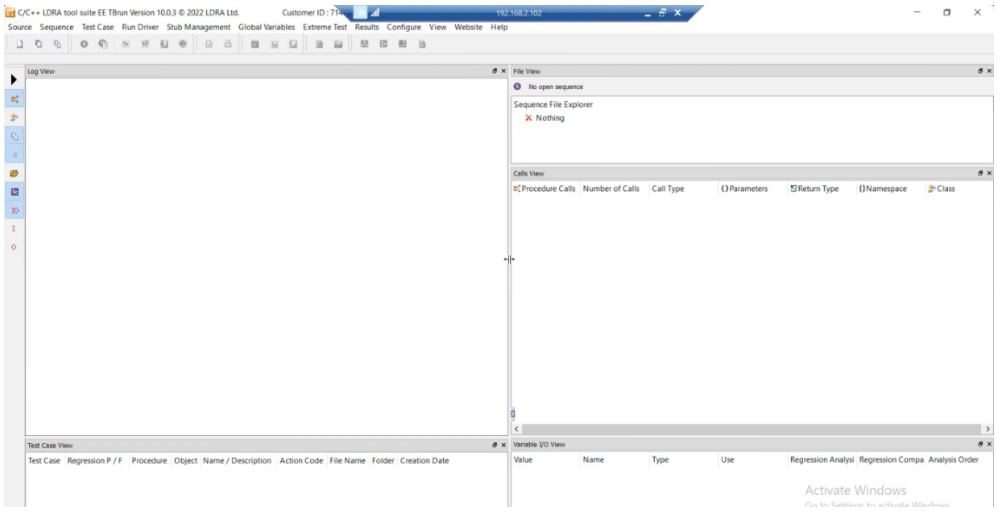
Fig. 4.2.1. Black-box Testing

S.NO	PROS	CONS
1.	Testers do not need to learn implementation details of the system	Requires prioritization, typically infeasible to test all user paths
2.	Tests can be executed by crowdsourced or outsourced testers	Difficult to calculate test coverage
3.	Low chance of false positives	If a test fails, it can be difficult to understand the root cause of the issue
4.	Tests have lower complexity, since they simply model common user behavior	Tests may be conducted at low scale or on a non-production-like environment

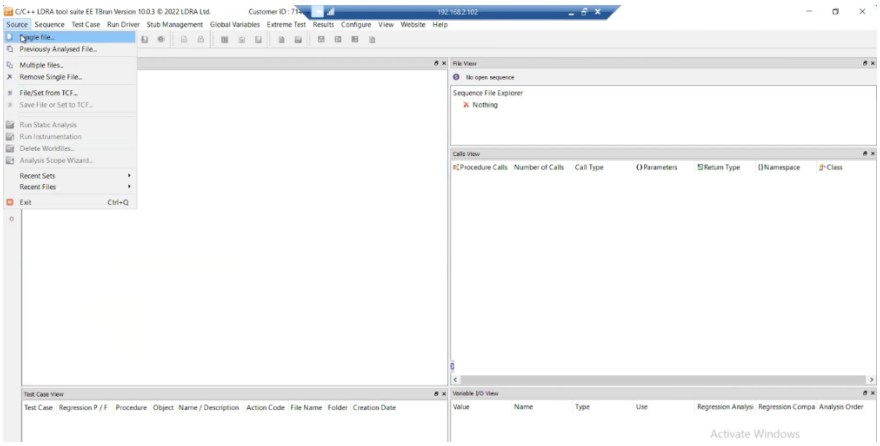
4.2.2 Black-box Testing using LDRA Tool

Step 1: Open LDRA Tool using remote desktop connection.

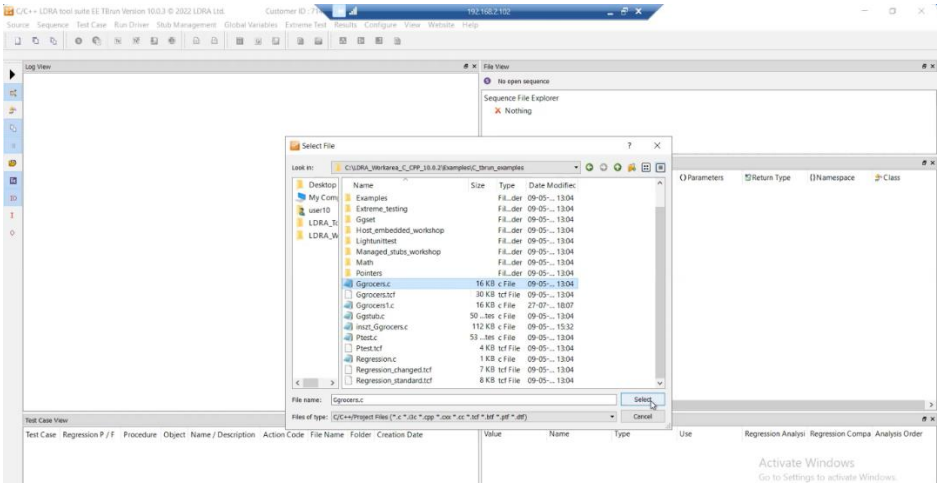




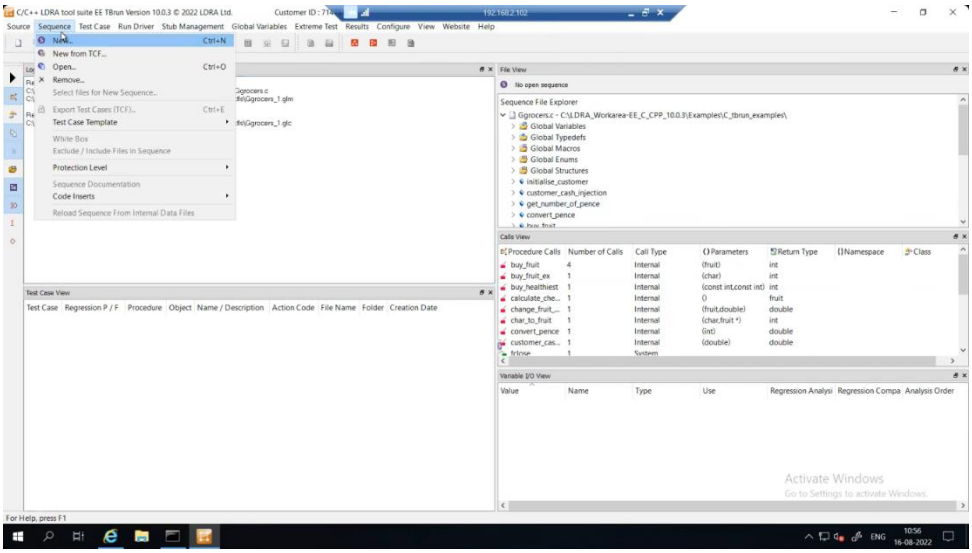
Step 2: Load the single file from source.



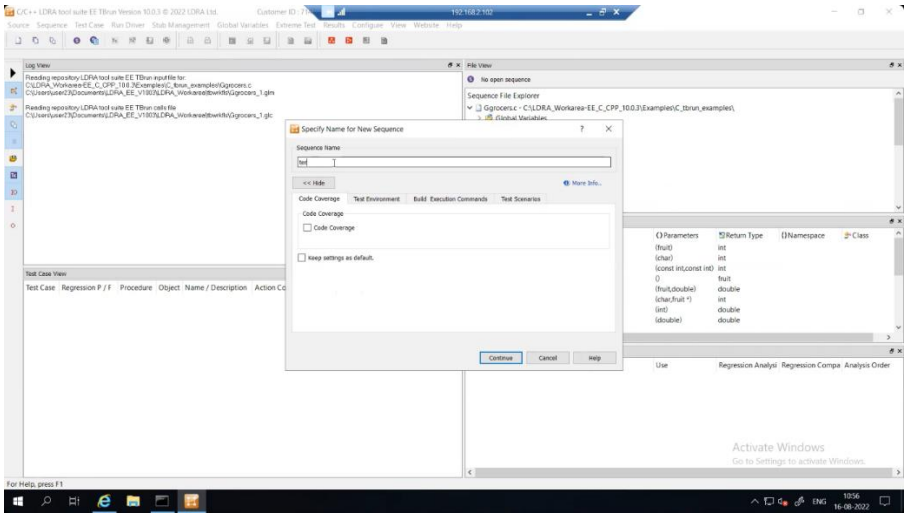
Select the .c file which does need to be tested.

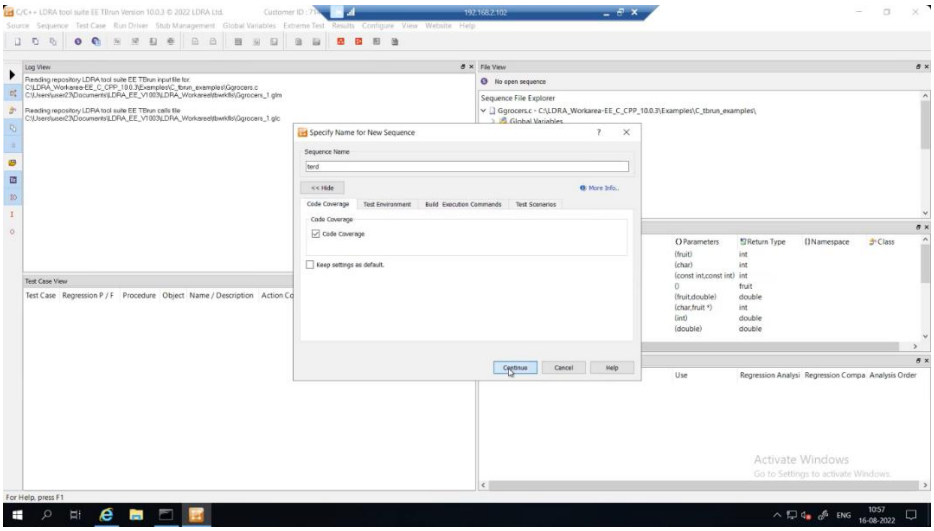


Step 3: Create a new sequence for test.

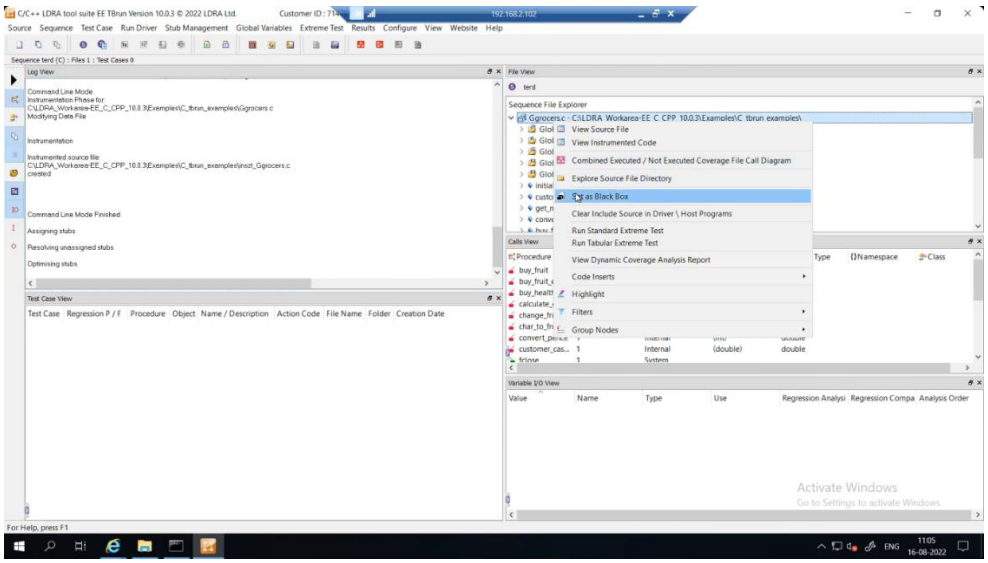


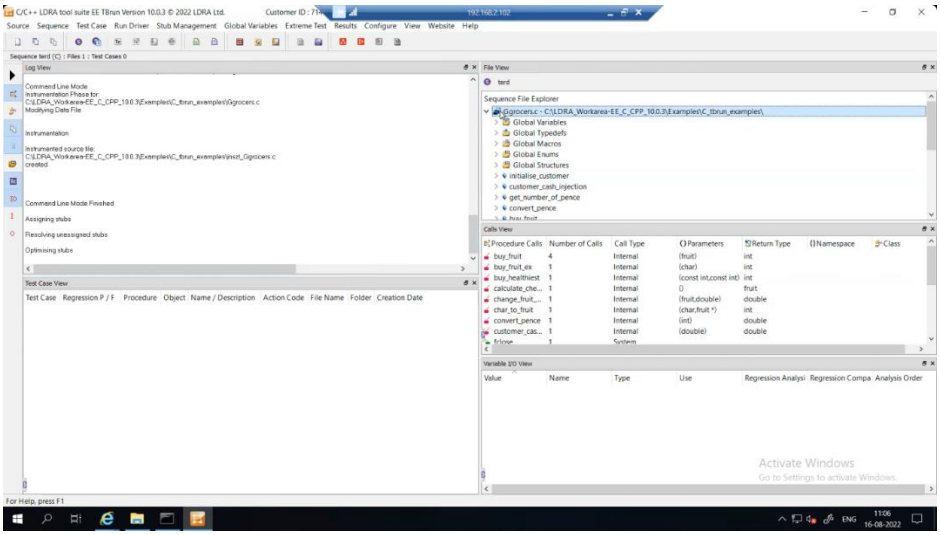
Give the name for the sequence and enable the code coverage then to click continue.



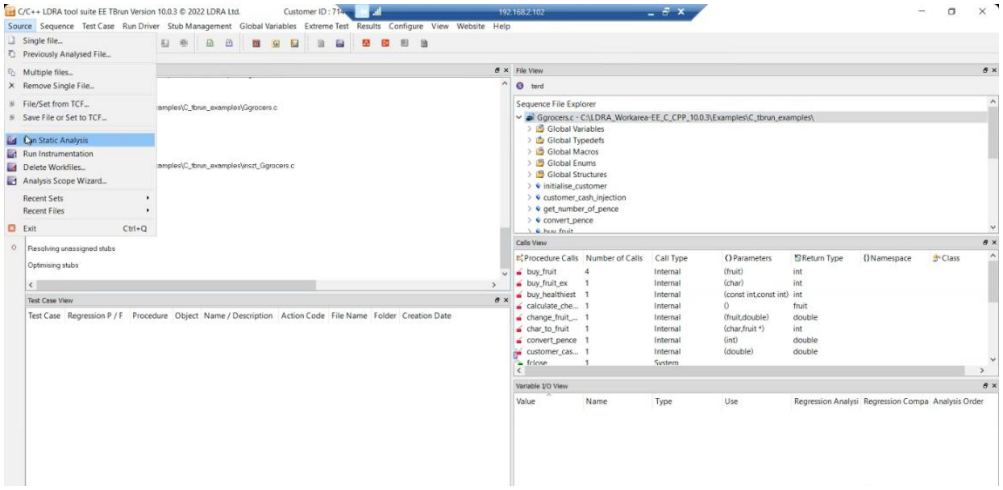


Step 4: For Black Box Test Right Click on the .c file and click on Set as Black Box

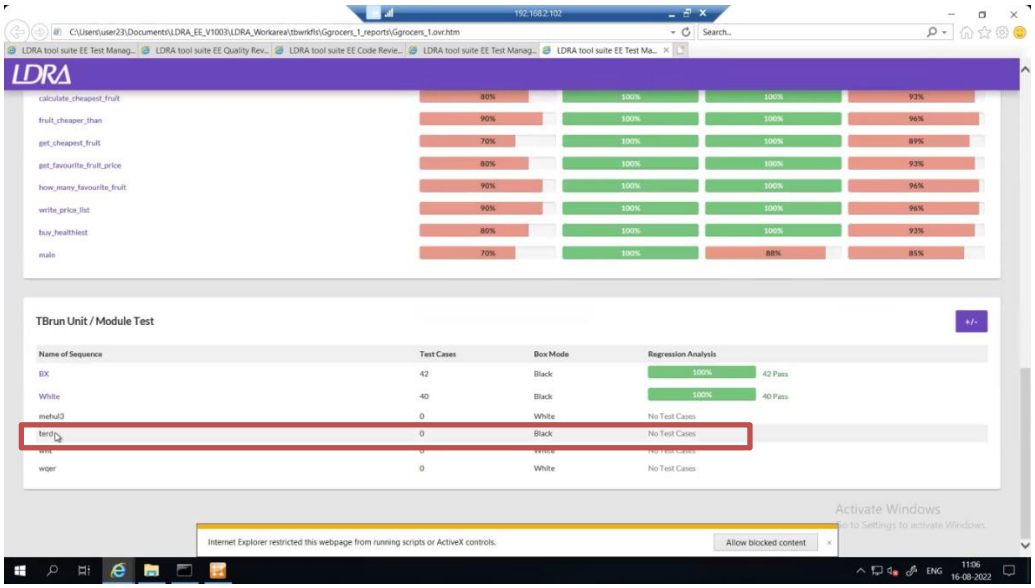




Step 5: Give Run Static Analysis from Source

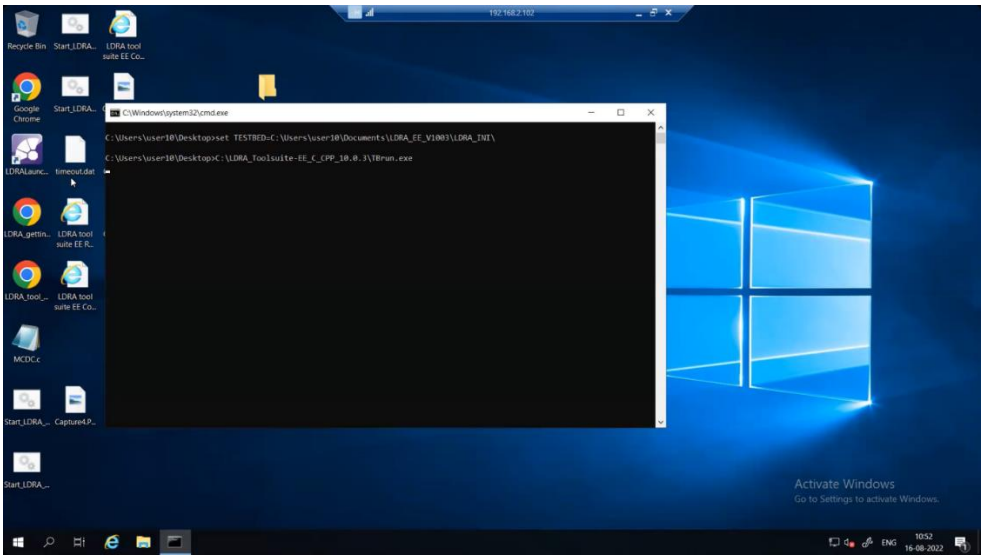


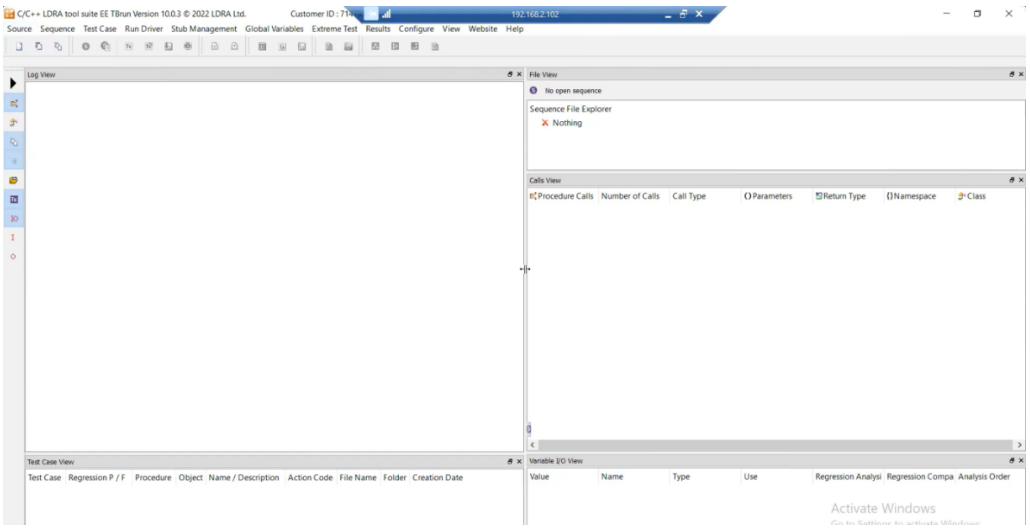
Step 6: Click on Test Manager Report from Results



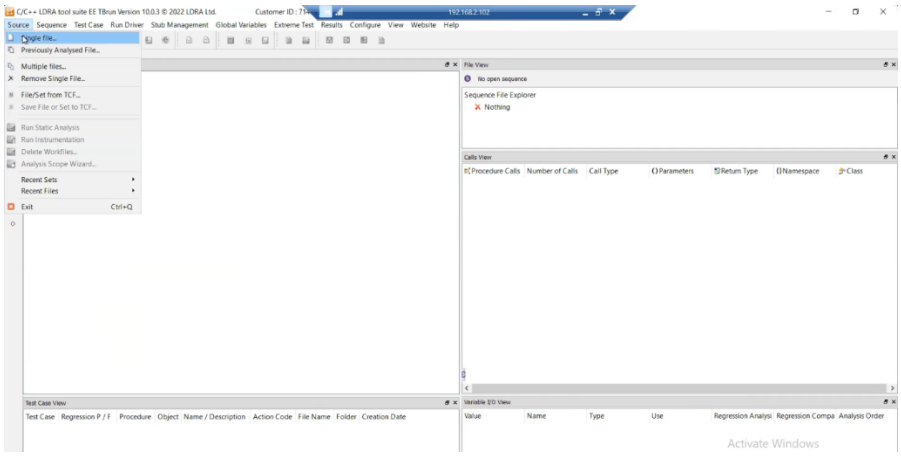
4.2.3 Creation of Test Cases for Black-box Testing

Step 1: Open LDRA Tool using remote desktop connection.

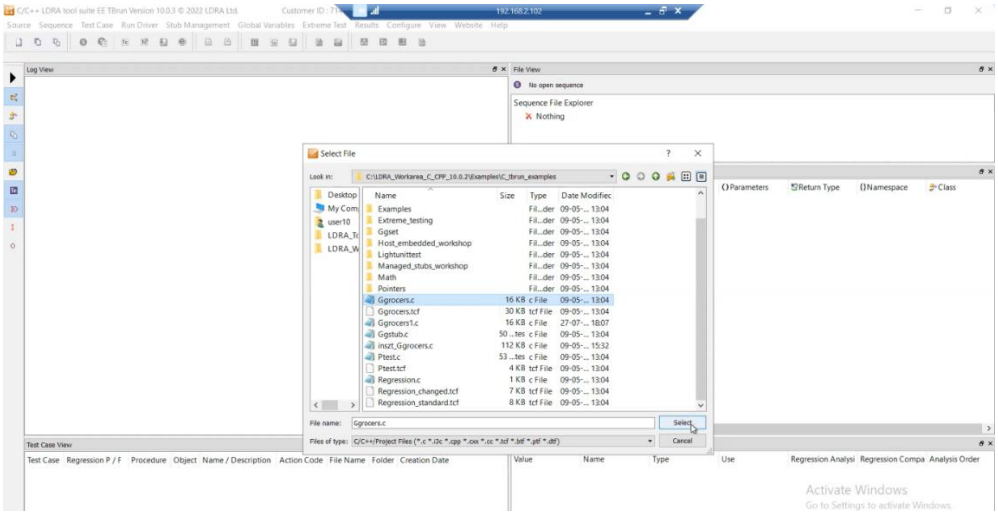




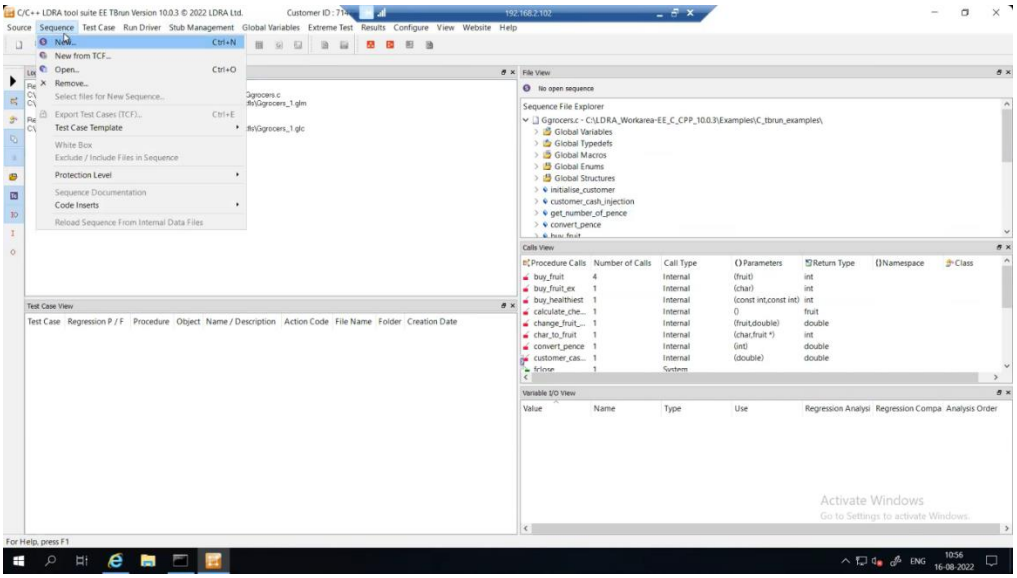
Step 2: Load the single file from source.



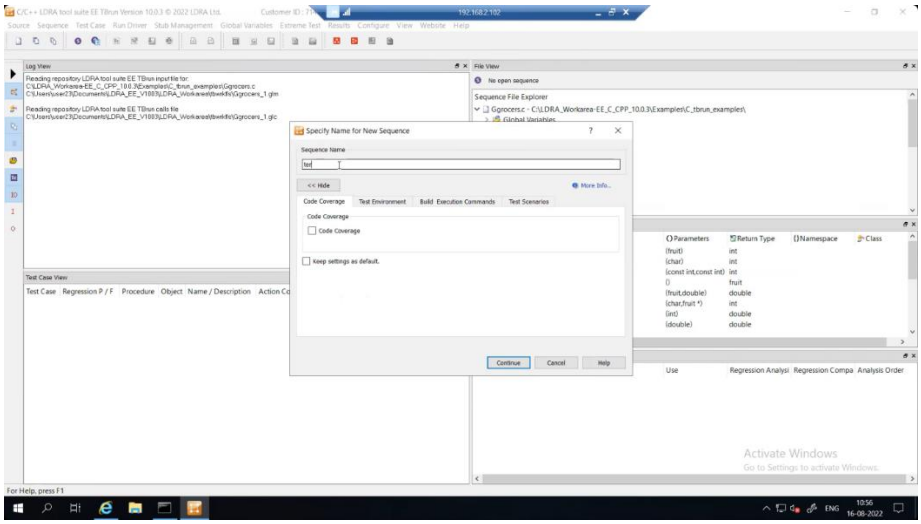
Select the .c file which does need to be tested.

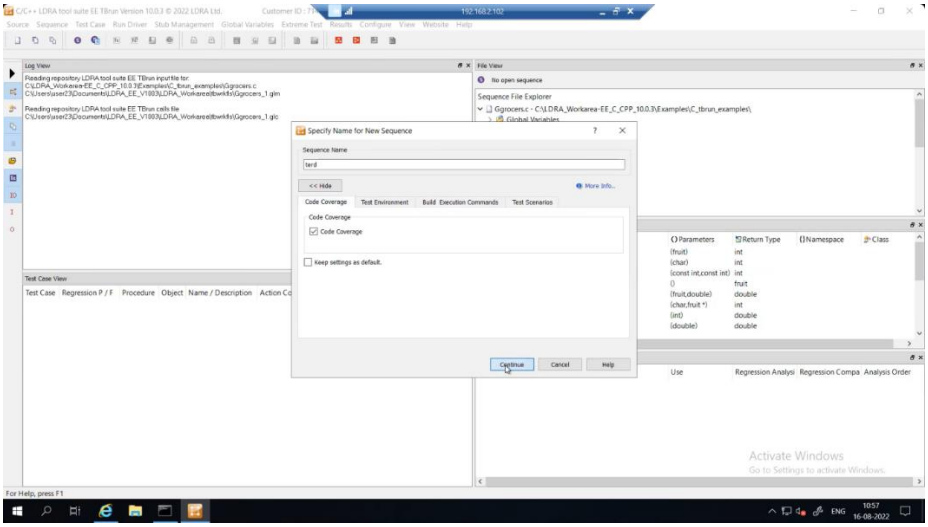


Step 3: Create a new sequence for the test.

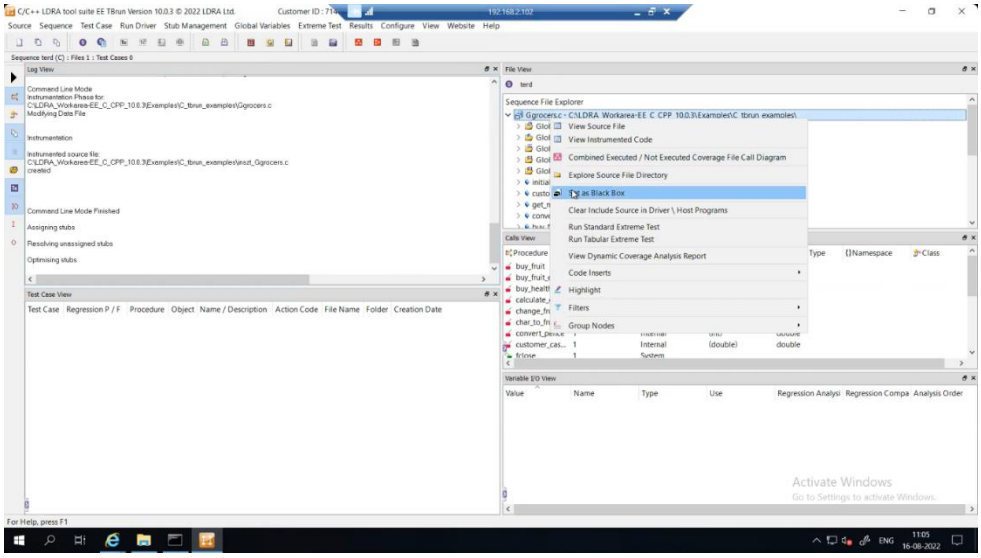


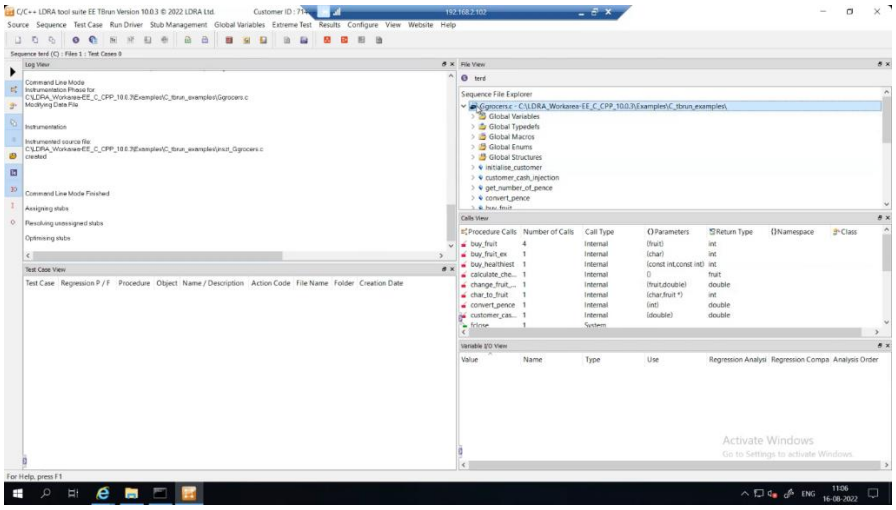
Give name for the sequence and enable the code coverage then to click continue



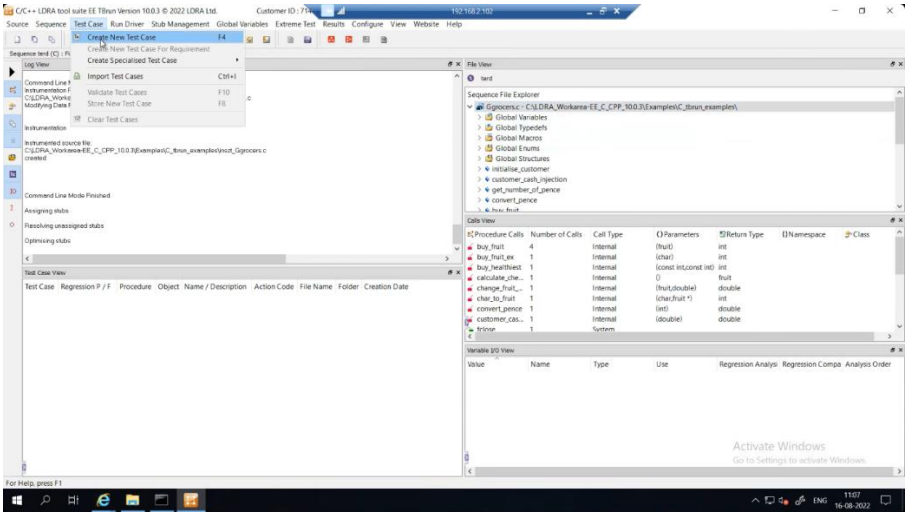


Step 4: For Black Box Test Right Click on the .c file and click on Set as Black Box

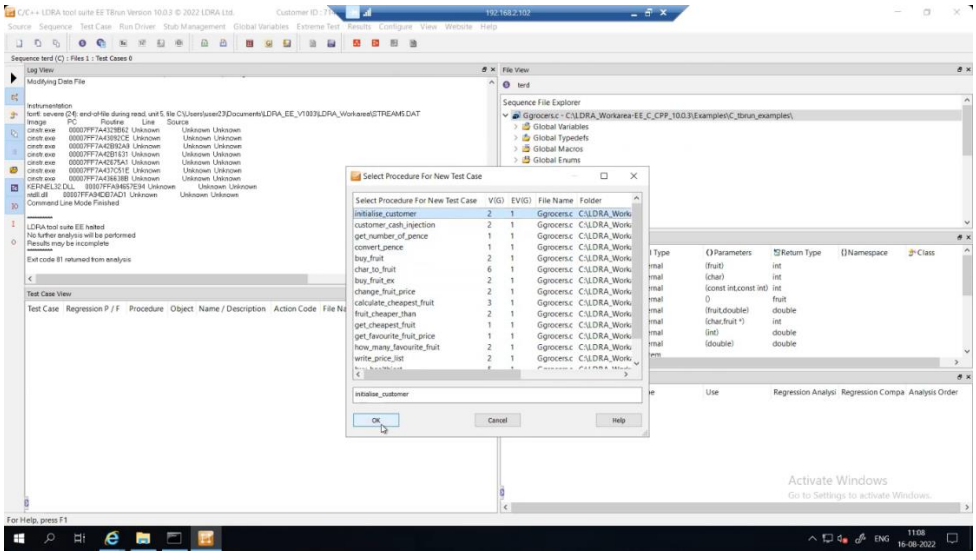




Step 5: Create new test case from Test Case



Step 6: Select the function to be tested (We have selected Initialise_Customer)



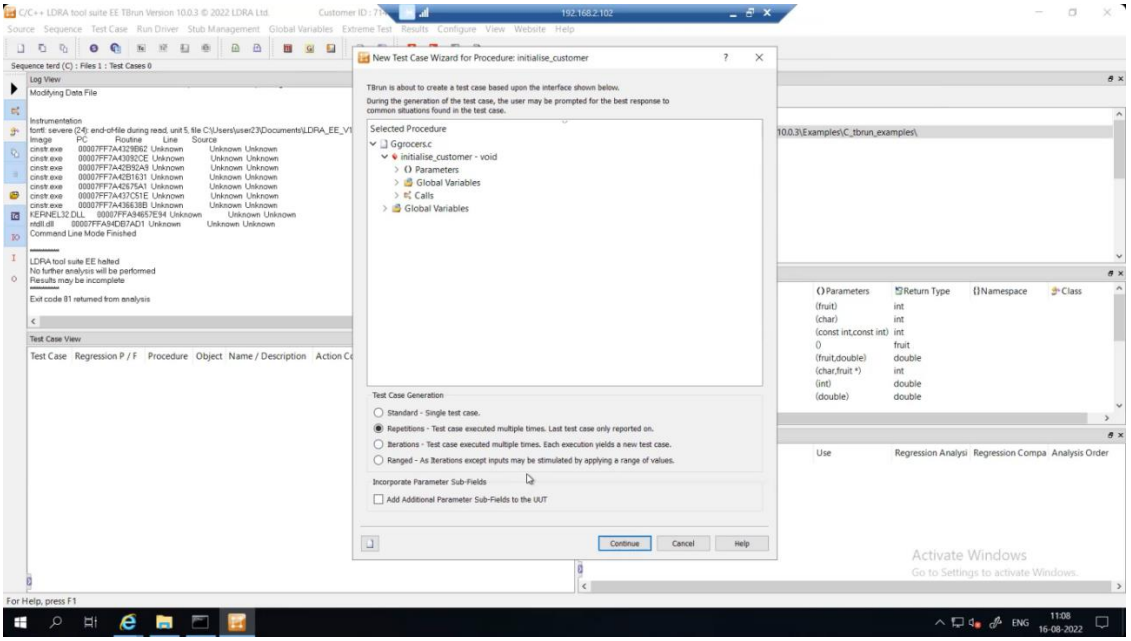
Select the Test Case Generation then click continue here we have chosen Repetition.

Standard for only one test case generation

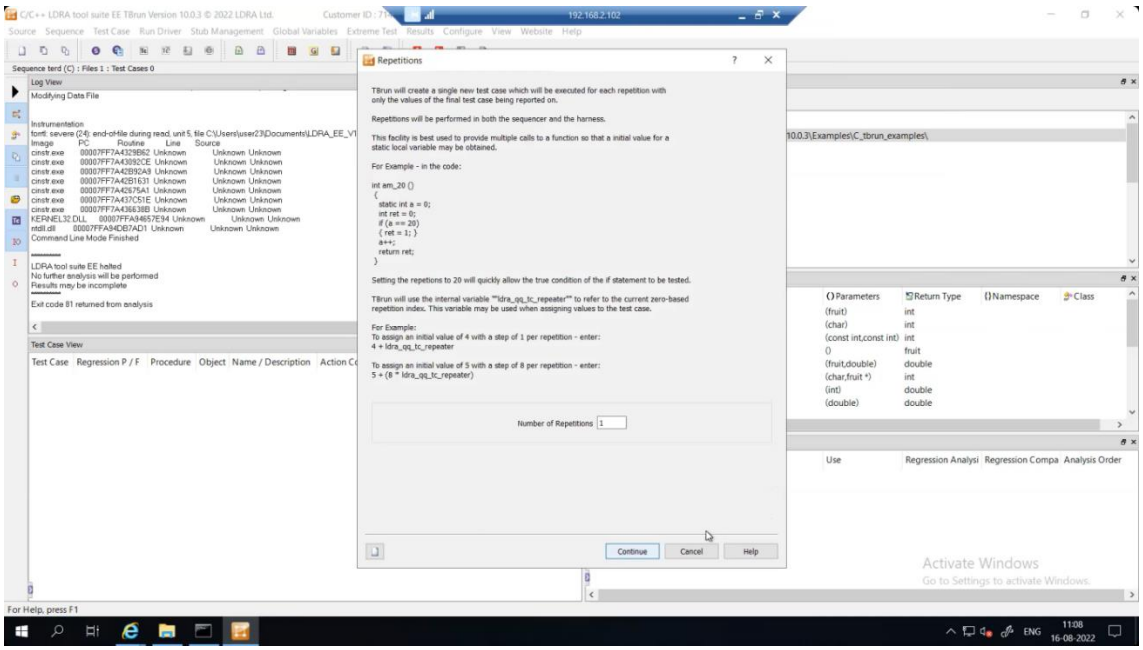
Repetition for multiple test case generation

Iteration for Numerous test case but the report will be generated for new test case only.

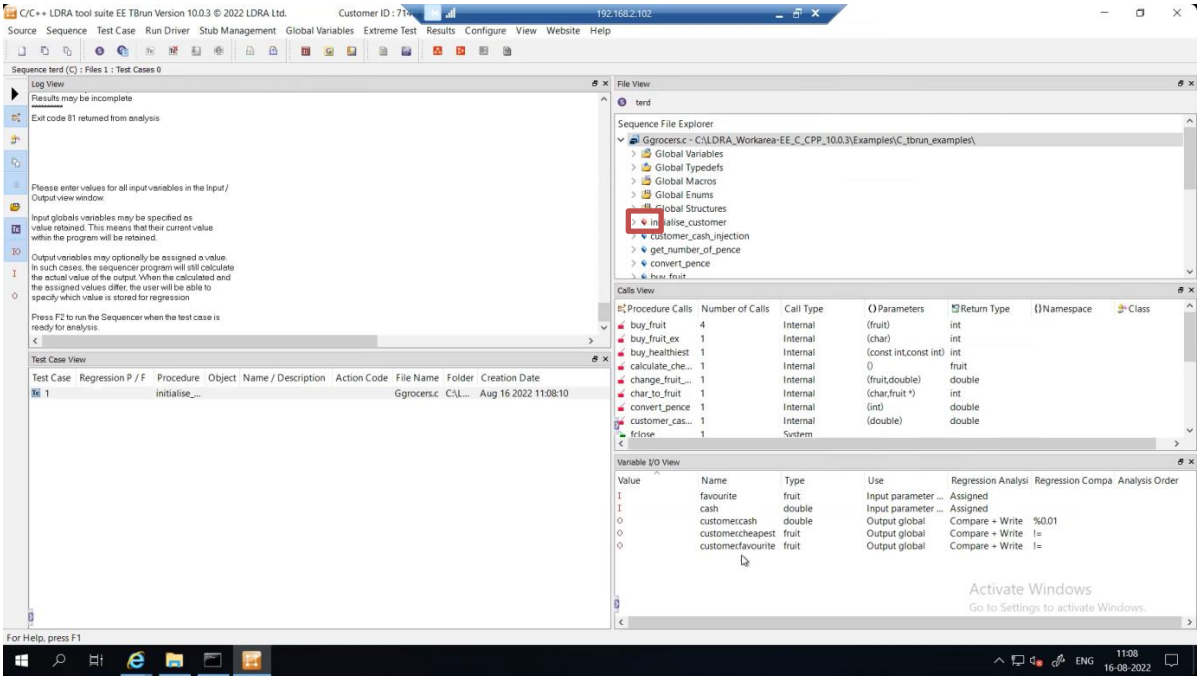
Ranged for number of test cases between the range for ex: 5 to 8.



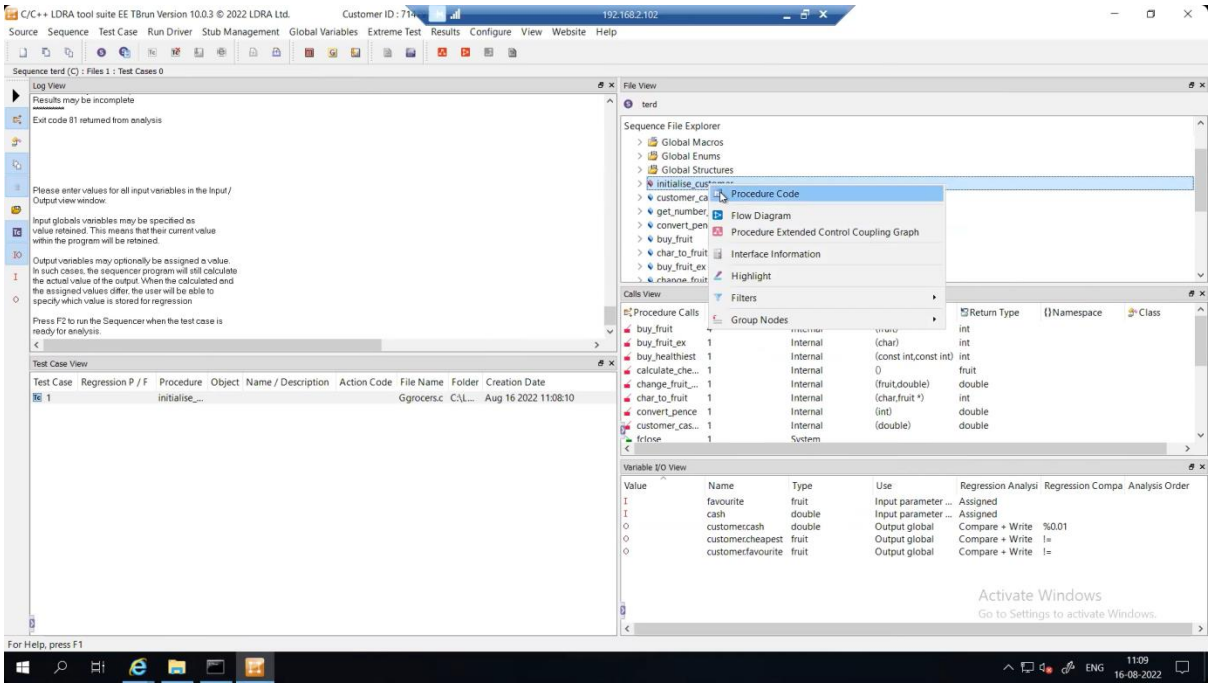
Enter the number of repetitions and click continue here we have given number as 1



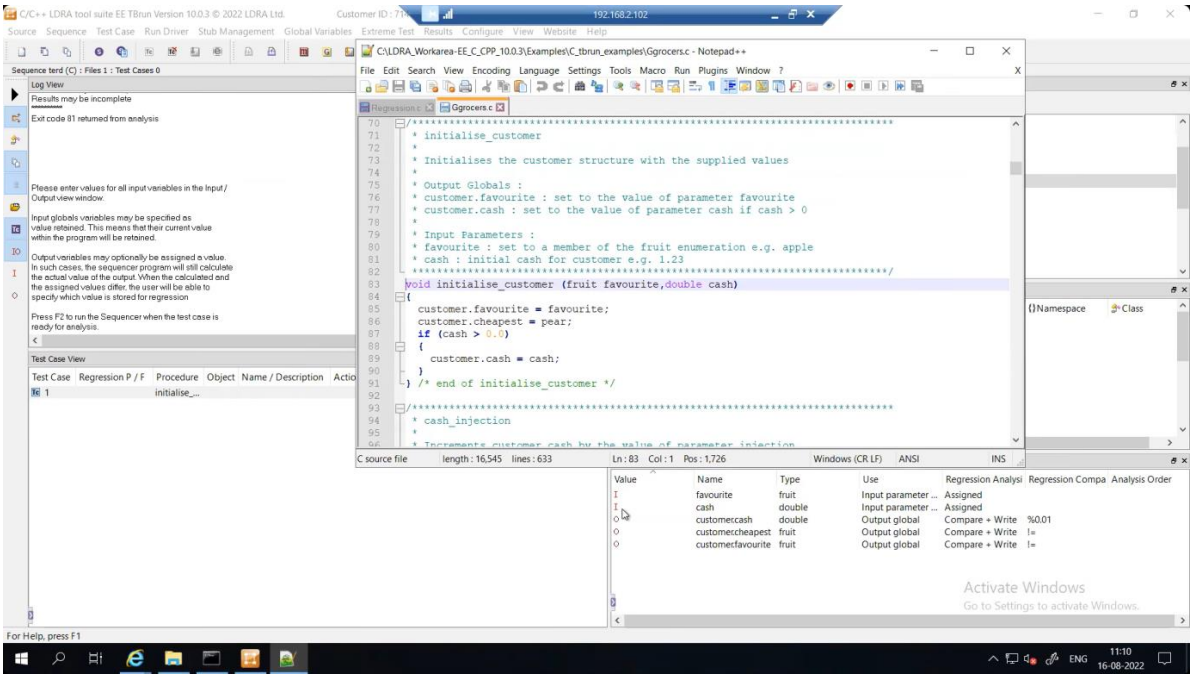
Now we will be able to see the function box color has changed from blue to red since we created a test case. Other functions which we do not generate test cases are in blue color.



Step 7: Now we need to give the input and output values for the variables in the Variable I/O view. First, we must know the code of it. To view the code right click on the function and then select procedure code.



Example: This is the code we have so for this we will give the Input and Output values



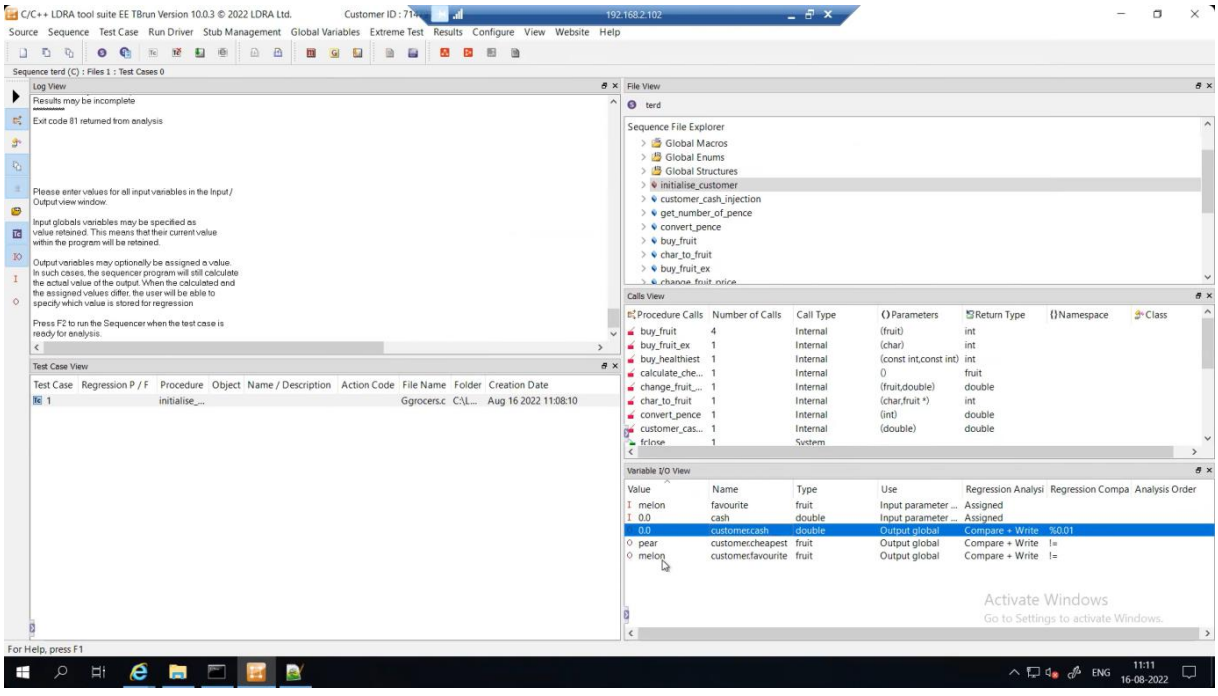
From this we will be able to understand the following

Favorite and customerfavorite is same

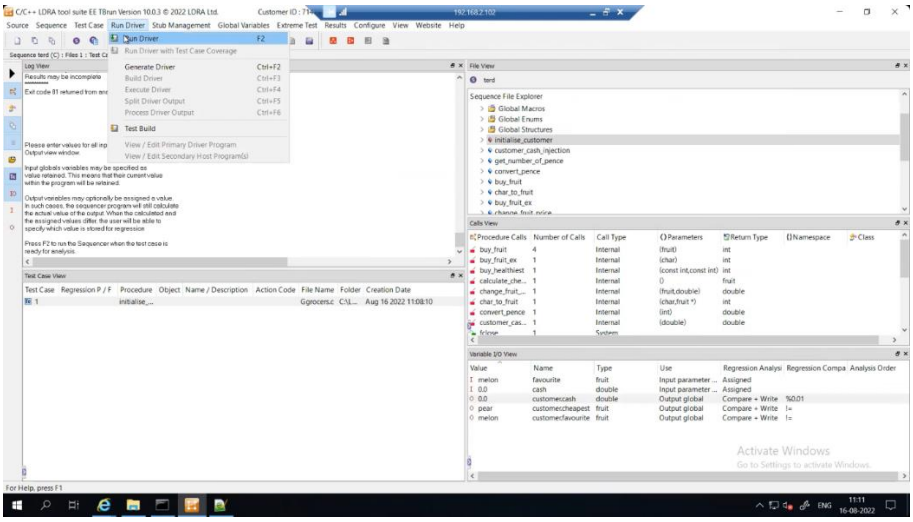
Customercheapest=peer

Cash=0.0

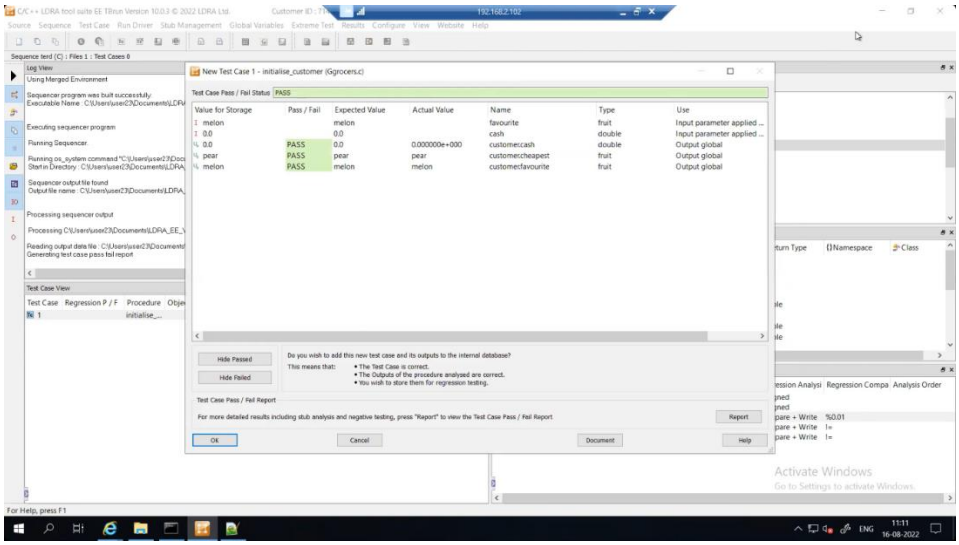
Customercash=0.0

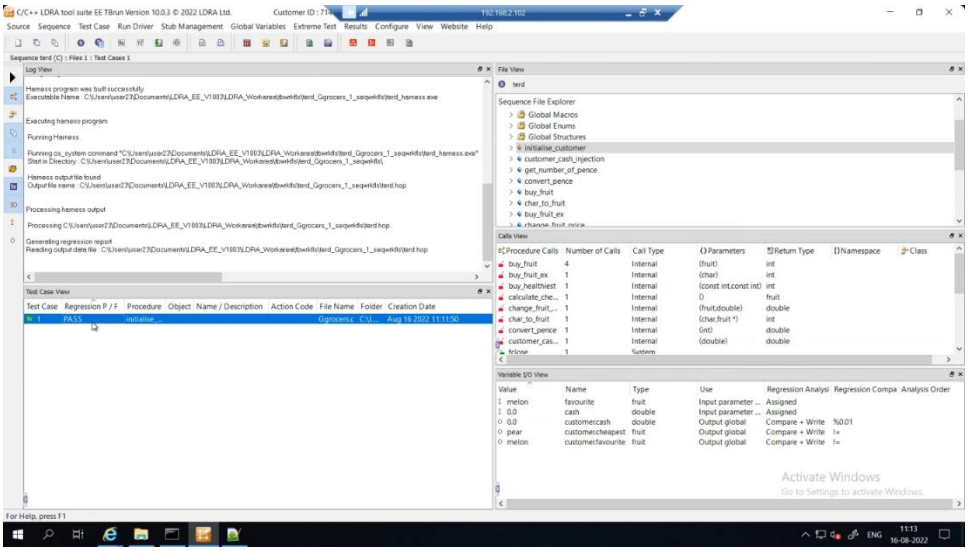


Step 8: Click on Run Driver in Run Driver or Press F2

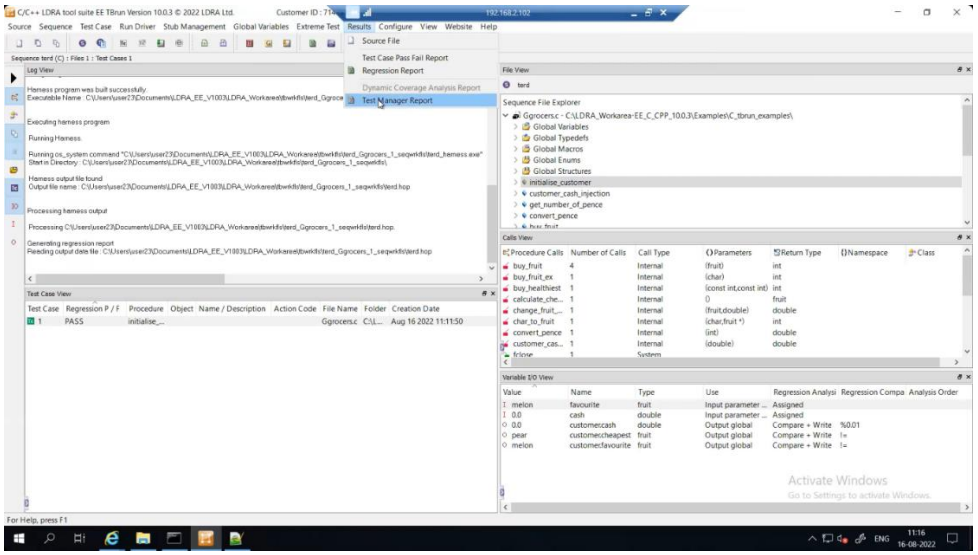


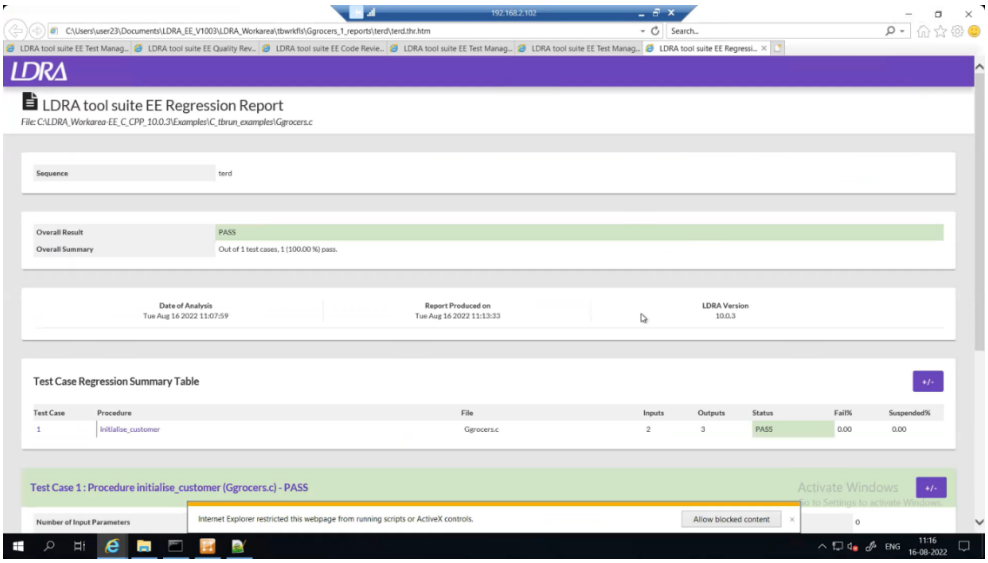
Step 9: Regression report will be generated.

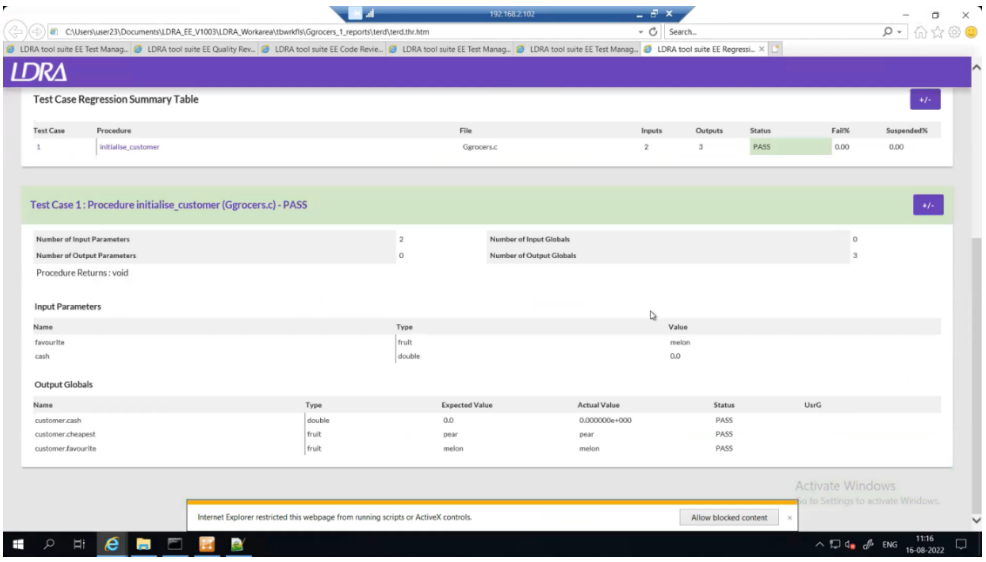




Step 10: To see the report click on Test Manager Report





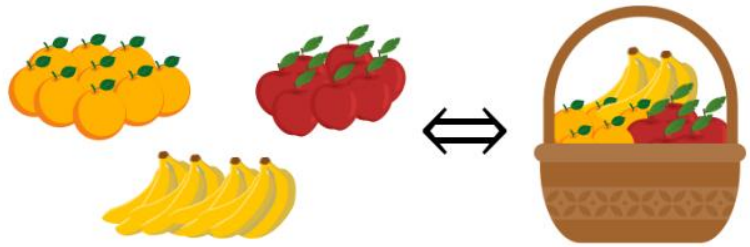


4.2.4 Equivalence Classes and Boundary Check

What is an equivalence class?

- Objects belonging to the same class

Image 4 presents a group of oranges, apples, and banana. Making an equivalence class with images 5,6,7.



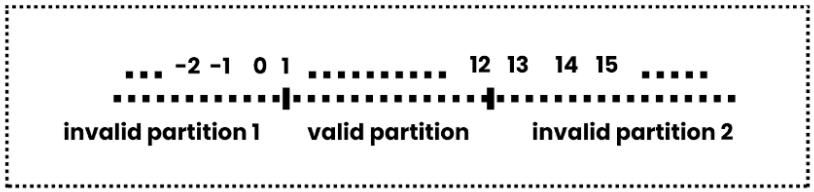


Fig 4.2.4.1 Example of Equivalence Class

The boundaries are the values on and around the beginning and end of a partition. If possible, test cases should be created to generate inputs or outputs that will fall on either side of each boundary. This would result in three cases per boundary. Procedures for determining equivalence classes and performing boundary checks

- Determine base classes for every variable/parameter
- Determine the definition range
- Determine a class of valid values
- Determine a class of invalid values

Define classes:

- For every equivalency class allocate class elements being processed (possibly) different to a new class

Select class elements:

- For every equivalency class select one representative class elements

Example:

```
unsigned char a;
if (a < 100)
{
Statements for if loop;
 }
else
{
Statements for else loop;
 }
```

What are the possible test cases to test the code above?

Solution:

Step 1: Determine base class 'a' is a variable of type unsigned char

Step 2: Define class, the range of variable 'a' is 0 to 255

Step 3: Select class elements determine the boundary classes 0 < a < 99; (if loop execution)

a = 100; (else loop execution)

101 < a < 255 (else loop execution)

Possible values of 'a' = 0, 255 (upper and lower limits)

'a' = 99, 100, 101 (equivalence class boundaries)

Real time example:

```
Unsigned int16 sensor_val_1;
Unsigned int8 sensor_val_2;
If (sensor_val_1 < 1000)
{
sensor_val_2 = 200;
}
Else
 {
sensor_val_2 = 0;
}
```

Variable	Type	Expected Value	Actual Value
sensor_val_1	Unsigned int16	999	999
sensor_val_2	Unsigned int8	200	200

Variable	Type	Expected Value	Actual Value
sensor_val_1	Unsigned int16	1000	1000
sensor_val_2	Unsigned int8	200	0

Fig 4.2.4.2 Real Time Example

4.2.5 What is a Decision Table?

The boundaries are the values on and around the beginning and end of a partition. If possible, test cases should be created to generate inputs or outputs that will fall on either side of each boundary. This would result in three cases per boundary. A decision table is a black-box test technique that visually presents combinations of inputs and outputs, where inputs are conditions or cases, and outputs are actions or effects. A full decision table contains all combinations of conditions and actions. On the contrary, an optimized one excludes impossible combinations of conditions and combinations of inputs that don't have any effect on outputs. They can also be applied when the action of the software depends on many logical decisions. What's more, these techniques can be used at any test level.

4.2.5.1 How to Create a Decision Table?

To explain how to create a decision table we will illustrate it with the login feature. What cases should we consider? A user can enter a valid email or invalid email. The user's email is in the database. A user enters a correct or incorrect password (a password that doesn't fit the account or password is invalid). A user can be blocked.

Conditions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Valid email	0	1	0	0	0	1	1	1	0	0	0	1	1	1	0	1
Email in database	0	0	1	0	0	1	0	0	1	1	0	1	1	0	1	1
User is blocked	0	0	0	1	0	0	1	0	1	0	1	1	0	1	1	1
Valid and correct password	0	0	0	0	1	0	0	1	0	1	1	0	1	1	1	1
Action	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Error message	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
Login password	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Fig. 4.2.5.1.1 Decision Table

The table above looks overly complicated, but we still must optimize it by removing cases that cannot exist and combining cases for which a change in one of the conditions is not relevant to the action of the system. We can see that some cases don't make any sense. Cases 3, 9, 10, and 15 cannot occur because, when an email is invalid, it cannot be saved in the database, so we can delete those cases. The next cases to be removed are 4, 7, 11, and 14; if the email is not saved in the database, it cannot be blocked. We can also combine cases 1 and 5 if the email is invalid (it doesn't matter if the password is). Let's see what the table looks like now, after simplifying and removing a few cases.

Conditions	1	2	3	4	5	6	7
Valid email	0	1	1	1	1	1	1
Email in database	0	0	1	0	1	1	1
User is blocked	0	0	0	0	1	0	1
Valid and correct password	-	0	0	1	0	1	1
Action	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Error message	1	1	1	1	1	0	1
Login password	0	0	0	0	0	1	0

Fig. 4.2.5.1.2 Decision Table

Of course, in decision tables, we can use yes/no or true/false, which are analogous to 0/1 values. We can use a number value (for example, temperature), words, pictograms, or whatever we need. An example of such a decision table may be a table containing information about access by users to various locations of the application, depending on the type of user.

Conditions	1	2	3	4	5
Recruiter	0	1	0	0	0
Manager	0	0	1	0	0
Admin	0	0	0	1	0
Super Admin	0	0	0	0	1
Action	**1**	**2**	**3**	**4**	**5**
Candidate data	-	view only	view only	view and block	view and block
User account	no access	no access	view only/ edit own	edit	edit
Manager account	no access	no access	view only	edit	edit
Permissions account	no access	no access	view only	edit	edit
Admin account	no access	no access	no access	view only	edit

Fig. 4.2.5.1.3 Decision Table

4.2.5.2 A Decision Table, as a Tool to Find Missing Signals

To present how a decision board can help in finding missing signals, we will use the example of simplified water heating, operating on the following principles:

The water in the tank should have a temperature between 30°C and 60°C.

The heater turns on when the temperature drops below 30°C.

The heater turns off when the temperature rises to 60°C.

The heater will also turn off when the water level in the tank drops below the minimum level.

Conditions	1	2	3	4	5	6
Temperature	< 30*C	30*C - 60*C	> 60*C	< 30*C	30*C - 60*C	> 60*C
To low water level	0	0	0	1	1	1
Action	**1**	**2**	**3**	**4**	**5**	**6**
Water heating	0	0	0	1	1	0

Fig. 4.2.5.2.1 Decision Table

At first glance, all we need to do with the table is to optimize the decision table. However, it is not recommended for us to turn on the water heating as soon as the temperature drops to 60*C, because according to our assumptions we want the temperature in the tank to be between 30 and 60*C. So we should optimize cases 1,2, and 3 - change the temperature to '-'(because the heating doesn't switch on when the water level is too low) and add one condition and one action changing the value of this condition, we'll call it 'warm water'. When the water is warm, the heating won't switch on.

Conditions	1	2	3	4	5	6
Temperature	-	< 30*C	30*C - 60*C	> 60*C	30*C - 60*C	< 30*C
To low water level	0	1	1	1	1	1
Warm water	-	0	0	-	1	1
Action	**1**	**2**	**3**	**4**	**5**	**6**
Water heating	0	1	1	0	0	0
Warm water	0	0	0	1	1	0

Fig. 4.2.5.2.2 Decision Table

4.2.5.3 How can the Decision Table Help in Software Testing?

A well-created decision table can help sort out the right response of the system, depending on the input data, as it should include all conditions. It simplifies the design of the logic and thus improves the development and testing of our product. With design tables, the information is presented in a clear, understandable way, so it's easier to find them than in the text describing the logic of the system. And finally, of course, using this technique helps find edge cases and identify missing signals in the system.

4.2.6 What is State Transition Testing?

State transition testing is a black box testing technique in which changes to input conditions cause changes in the application under test's (AUT's) state or output. State transition testing helps analyze the behavior of an application under different input conditions. Testers can provide positive and negative input values and record the system's 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. 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 a sequence of events that occur in the application under test, i.e., this will allow the tester to test the application's behavior for a sequence of input values, when the system under test has a dependency on the events or values of the past.

4.2.6.1 When to 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.

4.2.6.2 Four Parts of State Transition Diagram

1. State
2. Transition
3. Event
4. Action

Consider an ATM machine that is operational. When we try to withdraw money from it, we will follow the below steps one by one.

Step 1: Insert ATM card.

Step 2: Type in the Secret PIN.

Step 3: Error Message (If You Input the Incorrect PIN)

Step 4: Withdrawing funds (if the correct PIN was entered)

From the above example **State** is software might get 1st try, due to that **transition** will happen from one state to another state. Next the **Even**t that origin a transition (Withdrawing Money), due to that **Action** will happen either throwing error message when the wrong PIN is entered not Withdrawing money or Access Granted when the Correct PIN is entered.

4.2.6.3 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.

4.2.6.4 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, they will be logged in successfully. If the user enters an invalid password on the first or second try, the user will be asked to re-enter the password. And finally, if the user enters an incorrect password a third time, the account will be blocked.

4.2.6.5 State Transition Diagram

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.

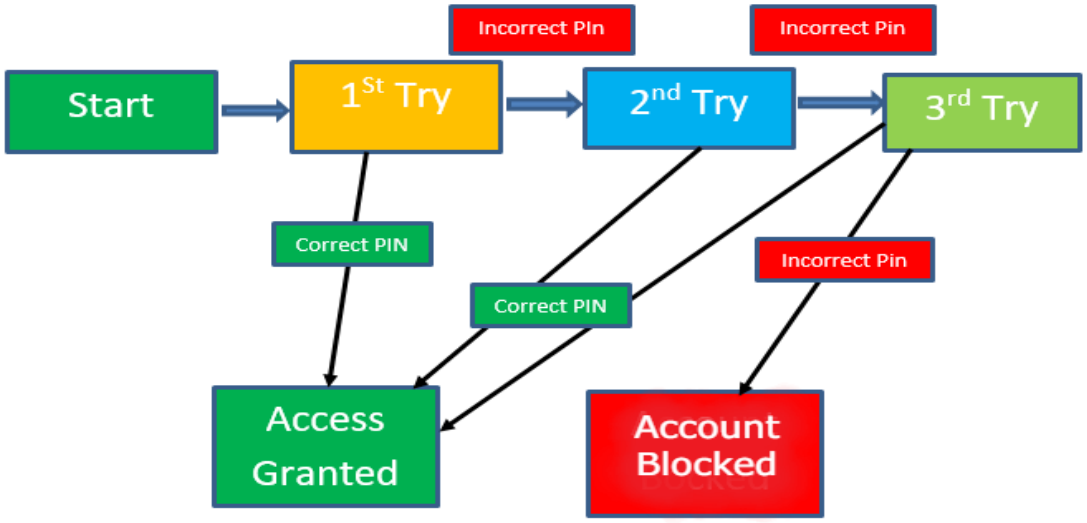


Fig. 4.2.6.5 State Transition Diagram

4.2.6.6 State Transition Table

	Correct PIN	Incorrect PIN
S1) Start	S5	S2
S2) 1st attempt	S5	S3
S3) 2nd attempt	S5	S4
S4) 3rd attempt	S5	S6
S5) Access granted	–	–
S6) Account blocked	–	–

In the table when the user enters the correct PIN, state is transitioned to S5 which is Access granted. And if the user enters a wrong password he is moved to next state. If he does the same 3rd time, he will reach the account blocked state.

4.2.6.7 Advantages and Disadvantages of State Transition Technique

Advantages	Disadvantages
This testing technique will provide a pictorial or tabular representation of system behavior which will make the tester to cover and understand the system behavior effectively.	The main disadvantage of this testing technique is that we can't rely in this technique every time. For example, if the system is not a finite system (not in sequential order), this technique cannot be used.
By using this testing, technique tester can verify that all the conditions are covered, and the results are captured	Another disadvantage is that you have to define all the possible states of a system. While this is all right for small systems, it soon breaks down into larger systems as there is an exponential progression in the number of states.

4.2.7 What is a Test Case?

It is a systematic procedure of executing a set of instructions to validate a particular SW requirement, either at the unit level, component level, module level, system level, or customer acceptance level. A systematic procedure for designing a test case considers various dependencies, ranging from the test case ID, test steps, identifying the test setup, pre-condition, test procedure, expected results, actual results, pass/fail, etc. In a real-world scenario, we will deal with more attributes to execute a test case at various levels of SW testing.

Attributes of a Test Case (24 attributes)

Test Case ID: This is an ID that denotes the linkage between the test case and SW Requirements. An ID is generated automatically and is unique.

Verification Method: This attribute specifies the verification method of the requirement. This can be a system test, integration test, system integration test (Mechanics/Hardware & Hardware/Software), system-system integration, SW test, SW integration test, SW module test, etc. It can be mandatory /optional depending on the decision of the team.

Testing Status: This attribute is used to show the coverage status of the requirement. This attribute can have the following values - Not Applicable, Not Covered, Partially Covered and Fully Covered. This attribute can be mandatory/optional depending on the team to decide.

Responsible Team: It defines which development team is responsible for the SW test. It can be mandatory/optional depending on the decision of the team.

Maturity: This attribute indicates the status of the requirement. It indicates whether the requirement is ready for review, in creation phase, reviewed and accepted by all relevant disciplines, accepted by stakeholders and invalid. It can be mandatory/optional depending on the decision of the team.

Module Name: It is used to name the module to which the requirement belongs to. This attribute can have the names of the modules used in the implementation of the project. It can be mandatory/optional depending on the decision of the team.

Pre-condition: This attribute defines the pre-condition for the SW test to execute. This attribute is recommended to be mandatory.

Test Procedure: This attribute indicates the procedure in which the test shall execute. E.g., Test inputs, actions, and sequential steps to be performed while executing the tests. This attribute is recommended to be mandatory.

Expected Result: This attribute is used to anticipate the expected result from the test we execute using the test procedure. This attribute is recommended to be mandatory.

Actual Result: This attribute is used to identify the actual result from the test we execute using the test procedure. This attribute is recommended to be mandatory.

Test Method: Test method(s) is used to create the test case and check whether the test methods are used adequately. It can be of the following values, ranging from equivalence classes, boundary values, state-based chart/state transition chart, elementary comparison test, control flow test, error guessing, etc. This attribute is recommended to be mandatory.

Test Responsible: This attribute indicates the person(s) responsible for the test. It is recommended to be mandatory.

Test Designer: This attribute indicates the name of the responsible test engineer who has designed the test for the test. It is recommended to be mandatory.

Manual Automation: This attribute determines whether the test case can be automated or not. It has the following values: Manual, and To Automate. This attribute is recommended to be mandatory.

SIL (Safety Integrity Level): Based on the highest SIL level of the linked requirements to which the test case belongs, this attribute is considered for Risk-based Test Strategy. It can list various SIL levels according to the industry. This attribute is recommended to be mandatory.

Priority: This attribute indicates the priority of the test case execution within one release cycle. This attribute is recommended to be mandatory. It can range from Low, Medium, High, etc.

Release: This attribute is used to define from which release onwards the test case is applicable. It is recommended to be mandatory.

Valid Until Release: This attribute determines the validity of test case we are writing. It basically determines until which release is the test case valid.

Customer-Demo: The test cases which should be a part of the Customer-Demo are defined under this attribute. This attribute is recommended to be mandatory.
HW Variant: This attribute is used to define for which variants this test case is applicable (based on requirements definition). This is a multi-select field. This attribute is recommended to be mandatory.
Reuse: Identify the Test Cases which are reused (or partially reused) from other projects. These test cases can be used on a "high" level testing, for example when requirements are not yet available. This attribute is recommended to be mandatory.
Test Suite: This attribute defines which test suite the test case must be executed belongs to. The definition of the test suite is done in the test strategy/test plan. This attribute is recommended to be mandatory and can have the values like Regression, Integration, etc.

4.2.7.1 Mind Map of Test Cases

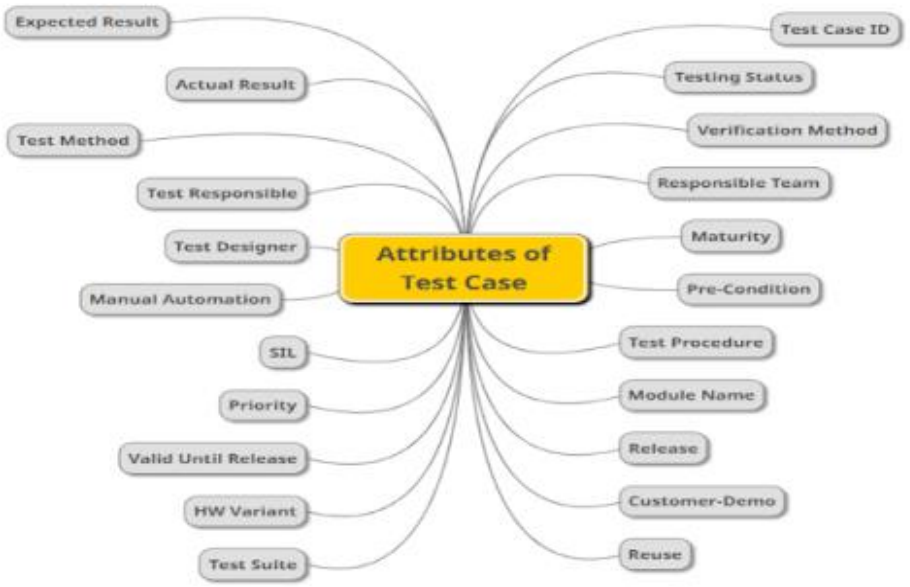


Fig. 4.2.7.1 Mind Map Test Cases

4.2.8 Test Specification

It is the blueprint for designing a test. Real-world example:

- Suppose you are given the task of serving meals to a family.
    - How many people must be fed?
    - What dishes should I serve based on the tastes of the family?
    - How many dishes should be served?
    - At what time (lunch or dinner) do you have to serve?
    - Do all the people prefer the same quantity of food?

These are some of the questions that would form the results of the test specifications, based on which we design the test cases.

4.2.8.1 Test Cases VS Test Specifications

Test Specifications	Test Cases
• Blueprint for designing test cases • Written by Test Manager/Tester and reviewed by Business Analyst/Business Manager. • More about thinking and discussing details. • Based on the testing environment, available resources and test setup, necessary changes can be made and shall be reviewed by stakeholder.	• Set of actions that determine how to test a test scenario • Written by the QA team • More about the documentation of test results • Based on the proposed changes in the test specification, test cases shall be changed

4.2.8.2 Real-case Scenario - Test Case from SW Required

SWR:

ID	SW Requirement Specification	Type	JIRA ID	Maturity	Release	Valid Until Release	Implementation Status	Verification Method	Function	SW_Component	Responsible Team	SIL	Test Severity
ID_1	If the Signal_A is 1 AND flag B is 1, then set signal_B to 1, else set signal_B to 0.	requirement	ABC-1234	Proposed	Release_A1 01	Release_A105	Implemented	SWT	Function A	SW_A	Your Team	SIL_High	High

SWR (Before Test)

ID	Test Case Title	Module Name	Test Method	Test Precondition	Test procedure	Expected Result	Actual Result	Responsible Team	Testing Status	Release	Valid Until Release	Function	SIL	Automation	Customer Demo
ID_1	To Test Signal_B is 1 when Signal_A is 1 and flag B is 1	SW_A	Elementary Comparison test	Connect the debugger Reset all the flags Reset all the signals	SET signal_A to 1 and flag B to 0 SET signal_A to 0 and flag B to 1 SET signal_A to 1 and flag B to 1	Signal_B is SET to 0 Signal_B is set to 0 Signal_B is set to 1		Your Team		Release_A1 01	Release_A10 5	Function_A	SIL_High		

SWR:

ID	SW Requirement Specification	Type	JIRA ID	Maturity	Release	Valid Until Release	Implementation Status	Verification Method	Function	SW_Component	Responsible Team	SIL	Test Severity
ID_1	If the Signal_A is 1 AND flag B is 1, then set signal_B to 1, else set signal_B to 0.	requirement	ABC-1234	Proposed	Release_A1 01	Release_A105	Implemented	SWT	Function A	SW_A	Your Team	SIL_High	High

SWR (After Test):

ID	Test Case Title	ModuleName	Test Method	Test Precondition	Test procedure	Expected Result	Actual Result	Responsible Team	Testing Status	Release	Valid Until Release	Function	SIL	Automation	Customer Demo
ID_1	To Test Signal_B is 1 when Signal_A is 1 and flag B is 1	SW_A	Elementary Comparison test	Connect the debugger Reset all the flags Reset all the signals	SET signal_A to 1 and flag B to 0 SET signal_A to 0 and flag B to 1 SET signal_A to 1 and flag B to 1	Signal_B is SET to 0 Signal_B is set to 0 Signal_B is set to 1	Signal_B is SET to 0 Signal_B is set to 0 Signal_B is set to 1	Your Team	Pass	Release_A1 01	Release_A10 5	Function_A	SIL_High	Yes	Yes

Summary

In this unit, we learned about the following concepts:

1. Balckbox Testing
2. Method to apply equivalence partitioning to derive test cases from given requirements
3. Method to apply boundary value analysis to derive test cases from given requirements
4. Method to apply decision table testing to derive test cases from given requirements
5. Method to apply state transition testing to derive test cases from given requirements
6. What is a test case?
7. Methods to derive test cases from a use case
8. Various attributes of test case specifications
9. Overview of test specifications

References

https://www.imperva.com/learn/application-security/black-box-testing/
https://www.merixstudio.com/blog/decision-table-software-testing/
https://www.guru99.com/state-transition-testing.html

UNIT 4.3: White-box Test Techniques

Unit Objectives

At the end of this unit, you will be able to:

1. Explain statement coverage
2. Explain decision coverage
3. Explain the value of statement and decision coverage

4.3.1 Introduction

White box testing is an approach that allows testers to inspect and verify the inner workings of a software system—its code, infrastructure, and integrations with external systems. White box testing is an essential part of automated build processes in a modern Continuous Integration and Continuous Delivery (CI/CD) development pipeline. White box testing is often referenced in the context of Static Application Security Testing (SAST), an approach that checks source code or binaries automatically and provides feedback on bugs and possible vulnerabilities. White box testing provides inputs and examines outputs, considering the inner workings of the code.

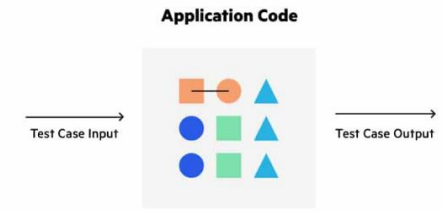


Fig. 4.3.1 White-box Techniques

S. No	PROS	CONS
1	Code optimization by revealing hidden errors	A complex and expensive procedure that requires the adroitness of a seasoned professional, expertise in programming, and understanding of the internal structure of a code.
2	Transparency of the internal coding structure is helpful in determining the type of input data needed to test an application effectively.	An updated test script is required when the implementation is changing too often.
3	covers all possible code paths, allowing a software engineering team to perform thorough application testing.	Exhaustive testing becomes even more complex using the white box testing method if the application is large.
4	It enables programmers to introspect because developers can carefully describe any new implementation.	Some conditions might not be tested as it is not realistic to test every single one.
5	Test cases can be easily automated.	The necessity to create a full range of inputs to test each path and condition makes the white box testing method time-consuming.

4.3.2 Statement Coverage (C0)

Statement coverage is a white-box testing technique that ensures all executable statements in the code are run and tested at least once. For example, if there are several conditions in a block of code, each of which is used for a certain range of inputs, the test should execute each range of inputs to ensure all lines of code are executed. Statement coverage helps uncover unused statements, unused branches, missing statements that are referenced by part of the code, and dead code left over from previous versions. Each statement in the SW code shall be executed at least once. In the example shown in image 2, when the stimulus or input is x = 5 and y = 0, the expected result is always x = 25 and y = 30, irrespective of the placement of the statement y = x + 5. Identify the error in Image 2 for both the correct and incorrect program references. Understand that C0 coverage is 100%, but we could not find the defect in the program.

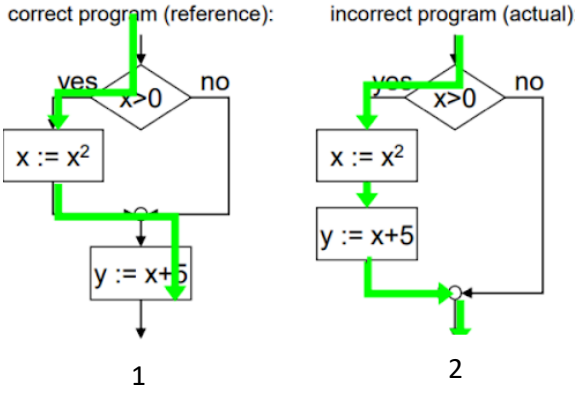


Fig. 4.3.2 Statement Coverage

Stimulus/ Input: x=5, y =0

Expected result: x=25, y=30

4.3.3 Decision Coverage(C1)

Each branch in the Control Flow Graph (CFG) of the SW code shall be executed. In the example shown in image 3, when the stimulus or input is x = 5 and y = 0, the expected result is always x = 25 and y = 30, irrespective of the placement of the statement y = x + 5. When the stimulus or input is x = -1 and y = 0, the expected result is x = -1 and y = 4, but the actual result is x = -1 and y = 0. Understand that C1 coverage is 100%, and we could find the defect in the program.

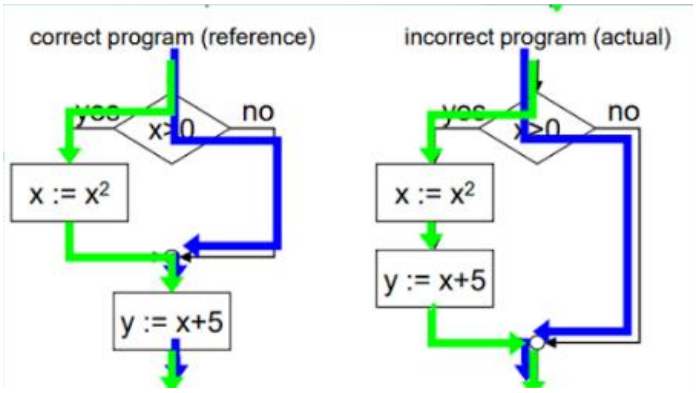


Fig. 4.3.3 Decision Coverage

Stimulus/ Input: x=5, y =0

Expected result: x=25, y=30

4.3.3.1 Value Statement and Decision Coverage (OR)

Modified Condition Decision Coverage (MCDC)

Objective: All atomic conditions and their combinations in loops and decisions are executed, so that every atomic condition changes its logical value once

Atomic Condition: A condition where the variable modifies according to a value

4.3.4 Modified Condition Decision Coverage (MCDC)

Example:

- Sensor_value_1 < 20
- Temperature >= 30
- Velocity <= 100
- Sensor_val_2 > 30
- Sensor_val_3 == 50

Note: '==' is a comparison operator comparing value of sensor_val_3 to 50

Tip: If the number of atomic conditions is 'N' then the number of test cases derived from MCDC is 'N+1'.

Deriving test cases in MCDC (we call them Pairs of MCDC)

Tip: A && B

- Let A, B be two atomic conditions
- Let Numeric value '0' = False Numeric Value '1' = True
- Total #Variables = 2, #Testcases from MCDC =3

Tip: A || B

- Let A, B be two atomic conditions
- Let Numeric value '0' = False and Numeric Value '1' = True
- Total #Variables = 2, #Testcases from MCDC =3

A	B	A&&B	A	B	A||B
0	1	0	0	0	0
1	0	0	0	1	1
1	1	1	1	0	1

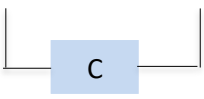
Fig. 4.3.4 Modified Condition Decision Coverage

Deriving test cases in MCDC (we call them Pairs of MCDC)

Let us consider the following example

Pres_val >100 AND fall_v_sensor >= 10 AND

sensor_val_3 == 100



Recreating the above example, A AND B AND C => A && B && C. From the tip in the earlier slides, 3. Atomic conditions and 4 test cases can be derived for MCDC. Let us now see how we will derive MCDC test cases for A && B && C (Since we have logical operator &&, we follow the rule of A&&B).

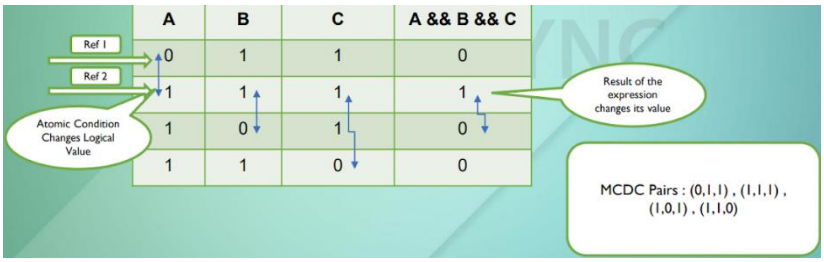


Fig. 4.3.4 Modified Condition Decision Coverage

Let us now see how we will derive MCDC test cases for A || B && C (Since we have logical operator && and ||, we follow the rule of A&&B and A||B)

A	B	C	A || B && C

Fig. 4.3.4 Modified Condition Decision Coverage Table

Will Be using Wacom Tablet for filling the table on live

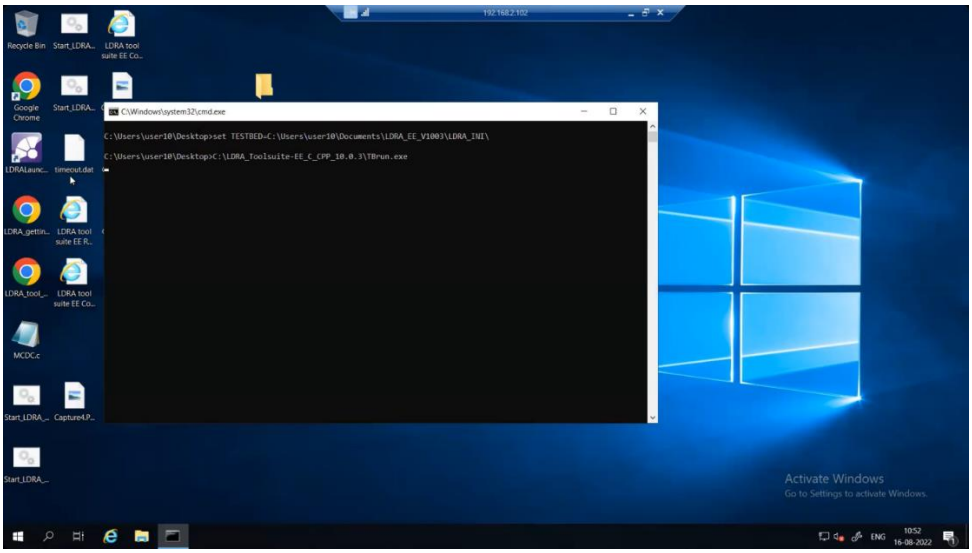
Let us now see how we will derive MCDC testcases for more complex (A && B) || (C && D) (Since we have logical operator && and || , we follow the rule of A&&B and A||B)

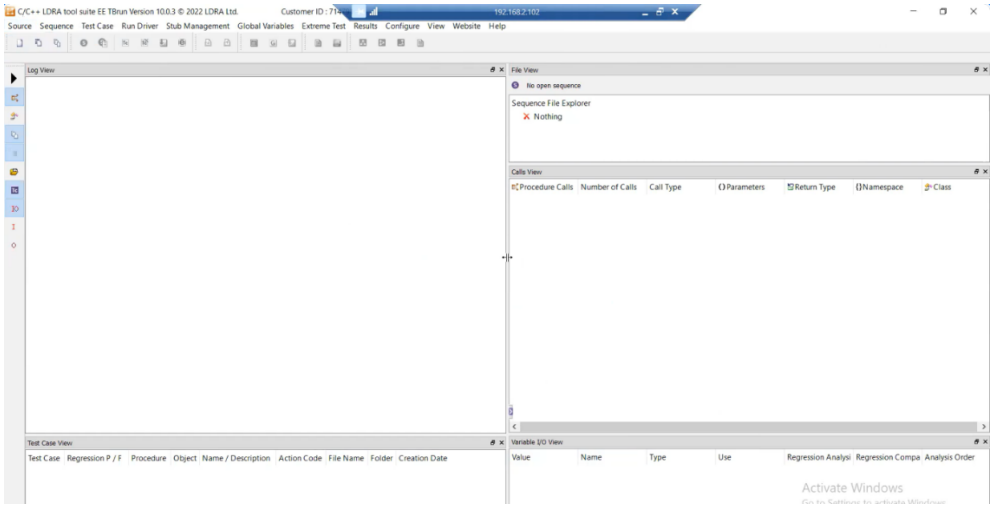
A	B	C	D	(A && B) || (C && D)

Fig. 4.3.4 Modified Condition Decision Coverage Table

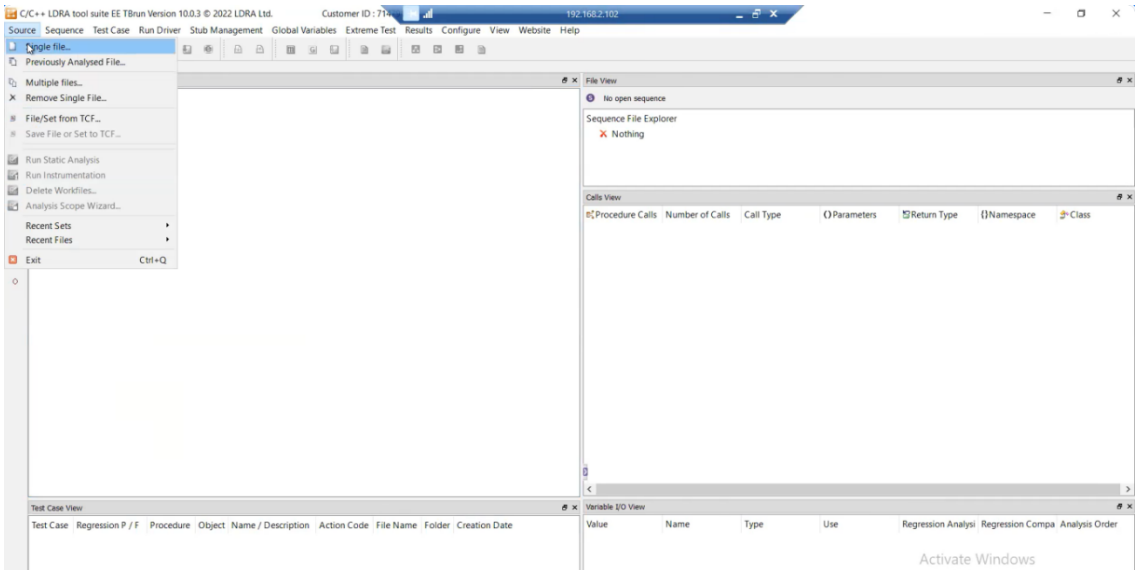
4.3.5 White-box Testing using LDRA Tool

Step 1: Open LDRA Tool using remote desktop connection.

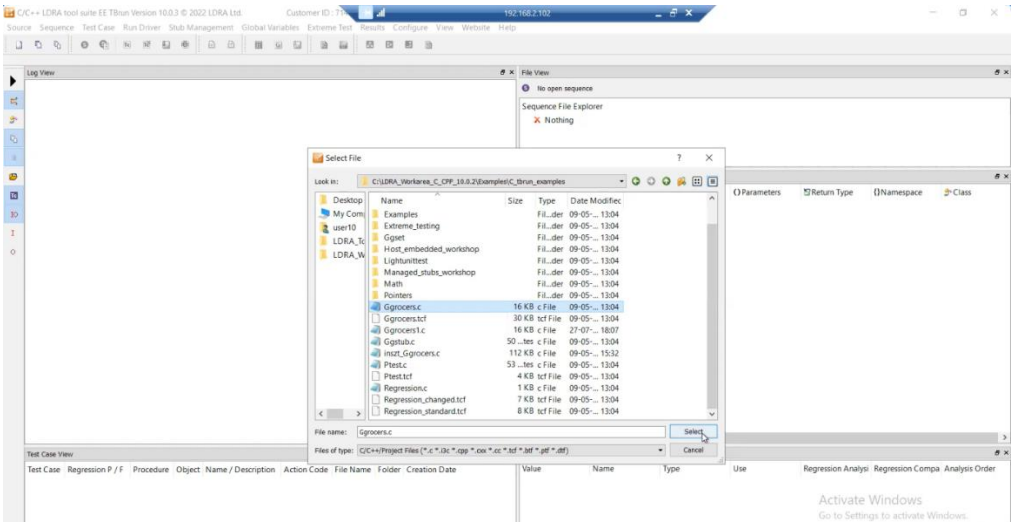


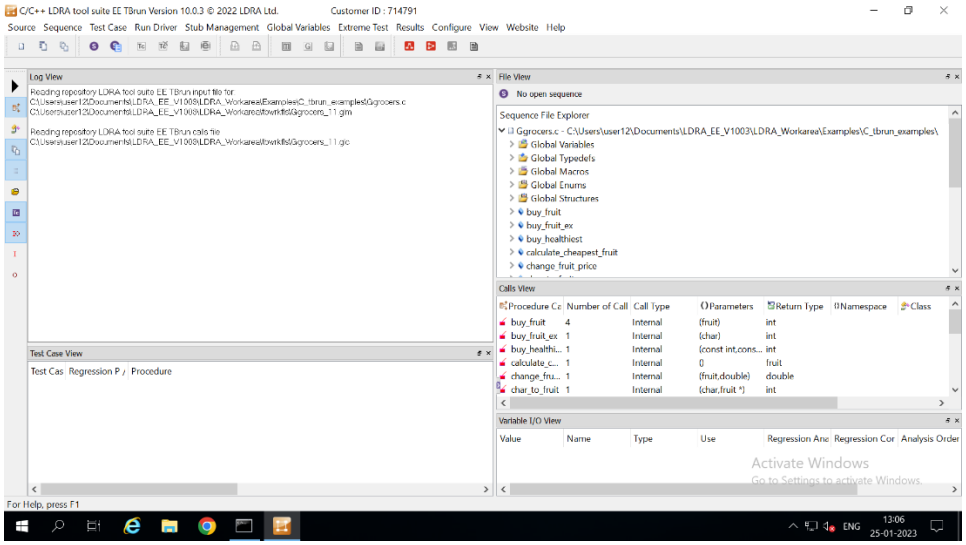


Step 2: Load the single file from source.

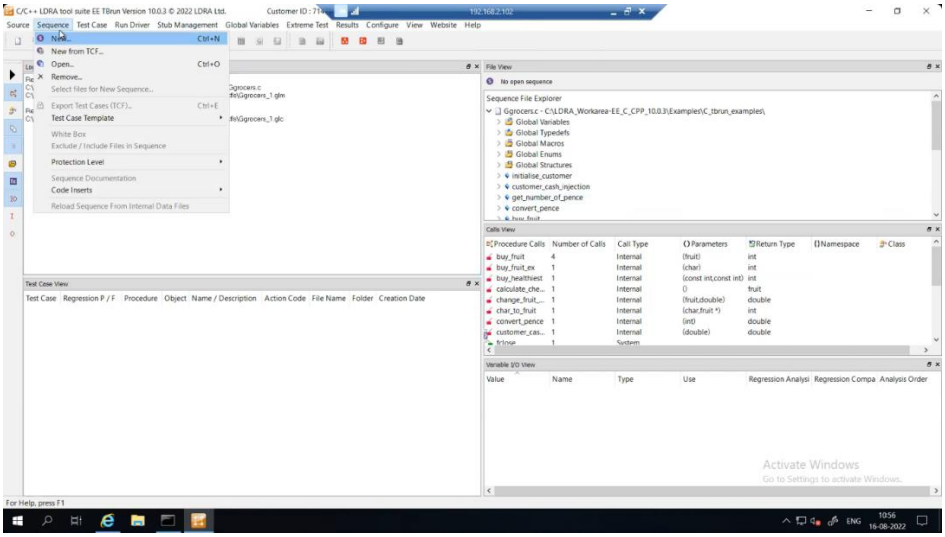


Select the .c file which does need to be tested.

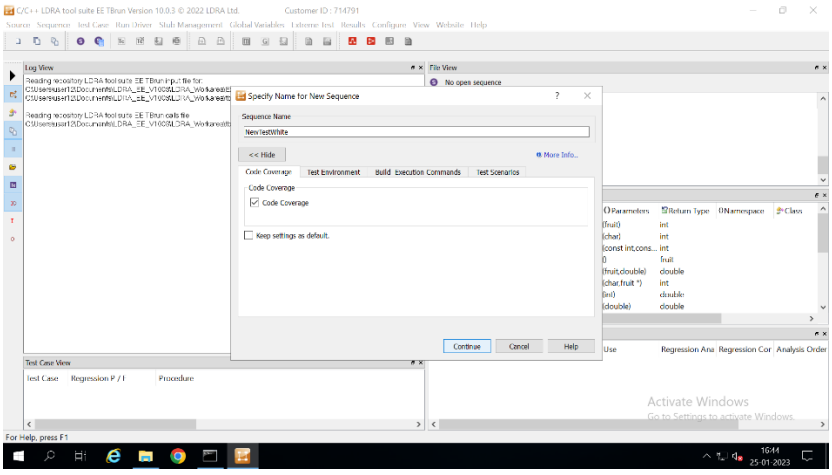




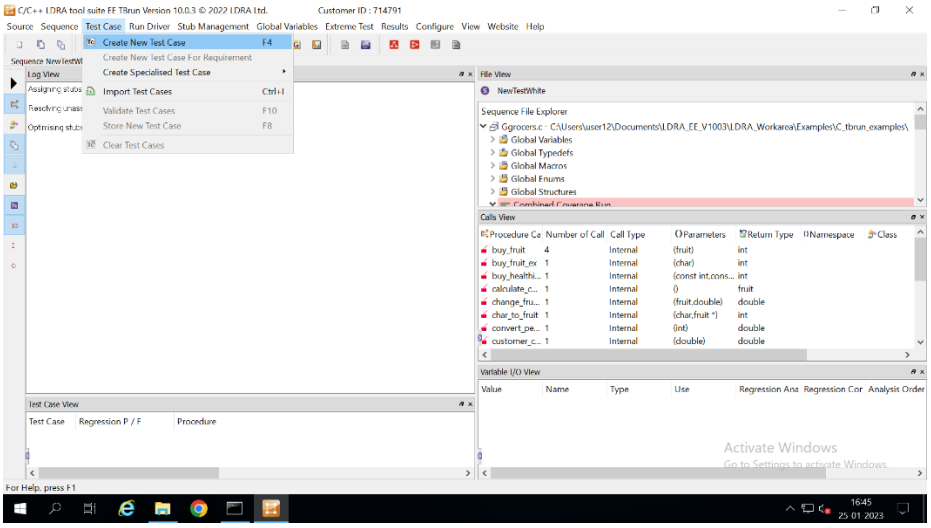
Step 3: Create a new sequence for test.



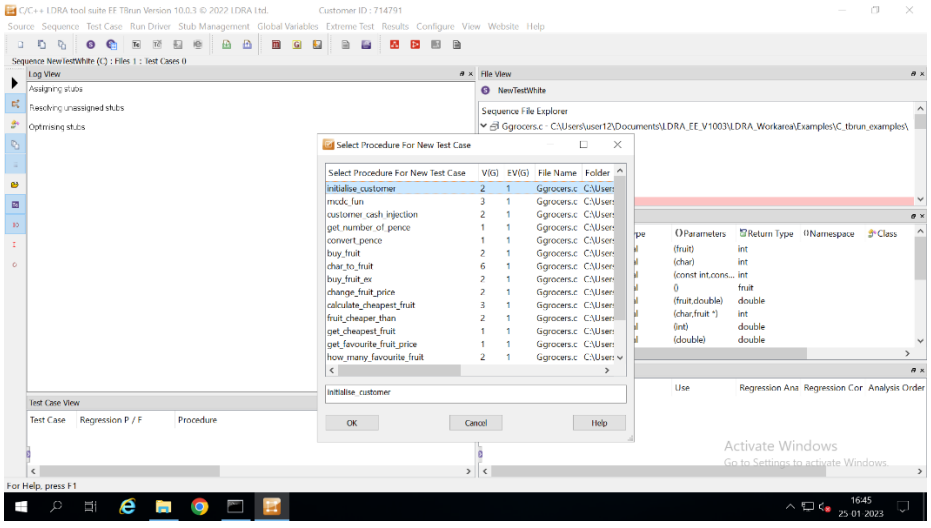
Give name for the sequence and enable the code coverage then to click continue



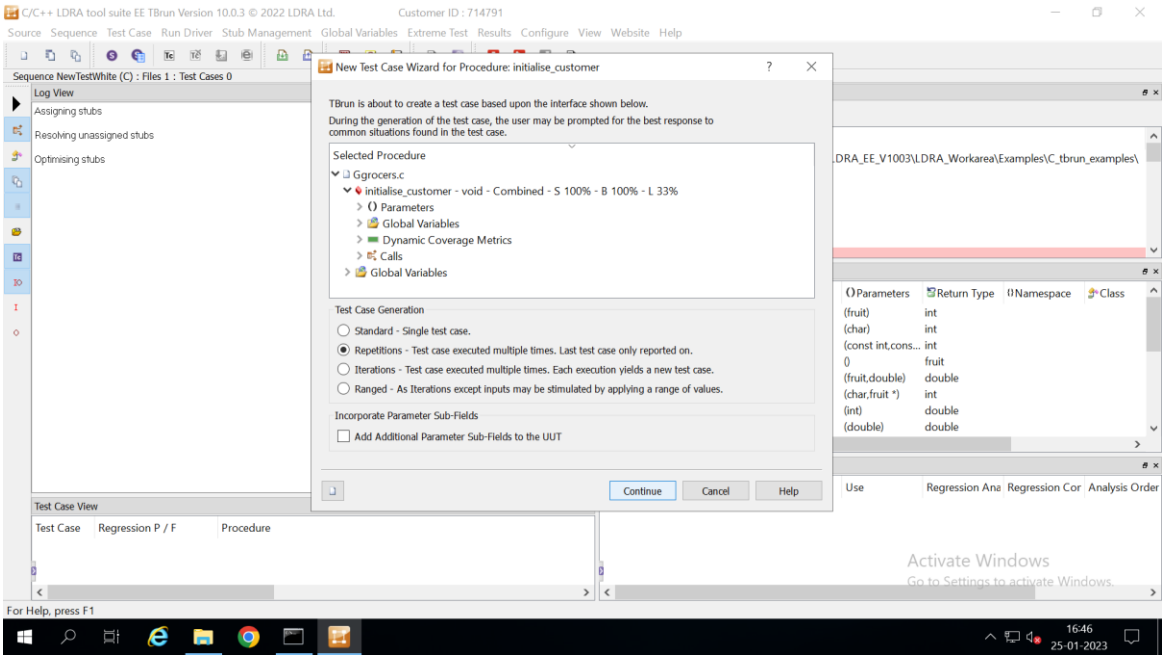
Step 4: Create new test case from Test Case



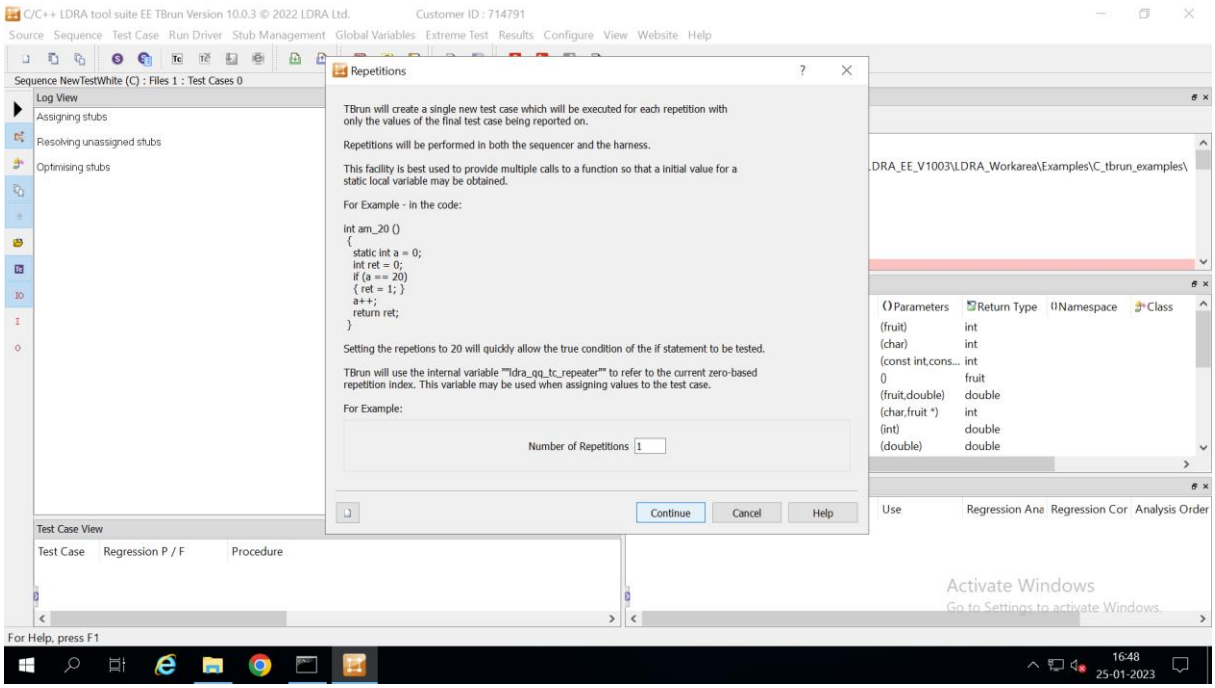
Step 5: Select the function to be tested (We have selected Initialise_Customer)



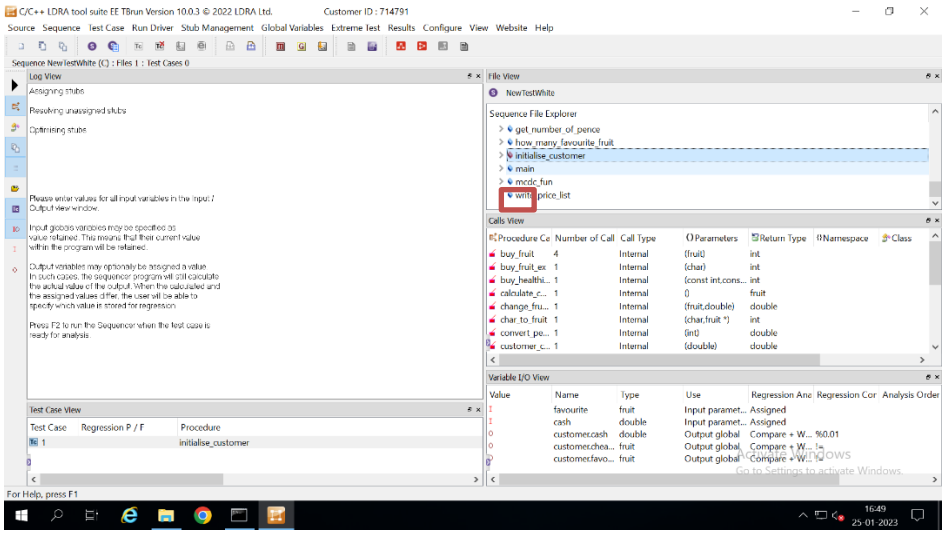
Select the Test Case Generation then click continue here we have chosen Repetition.
Standard for only one test case generation
Repetition for multiple test case generation
Iteration for Numerous test case but the report will be generated for new test case only.
Ranged for number of test cases between the range for ex:5 to 8.



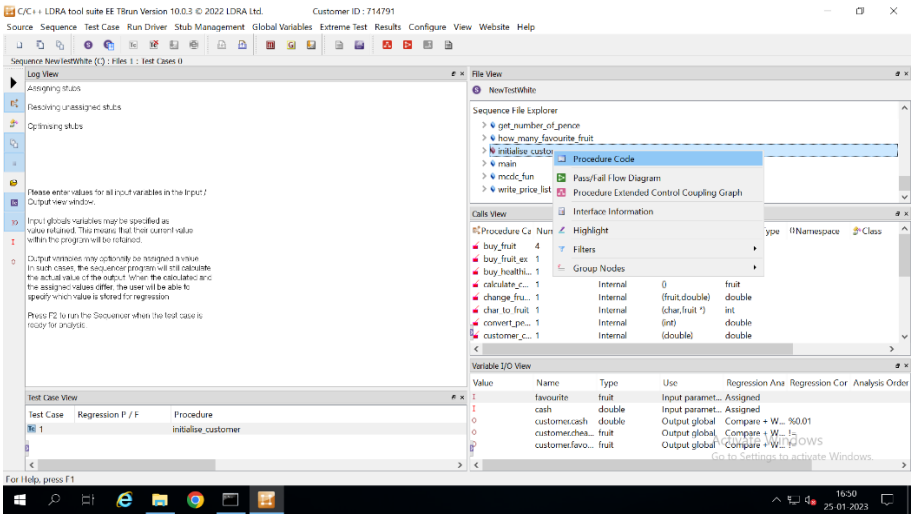
Enter the number of repetitions and click continue here we have given number as 1.

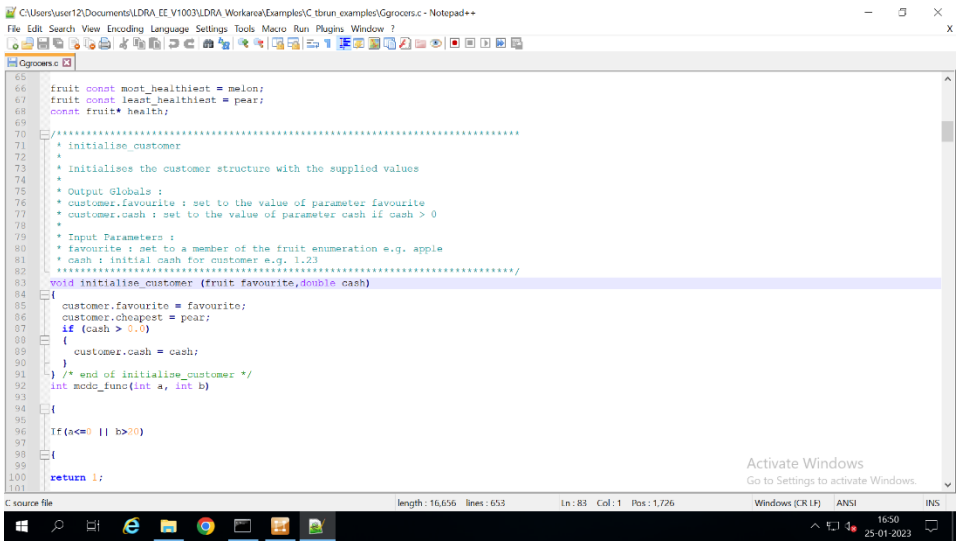


Now we will be able to see the function box color has changed from blue to red since we created a test case. Other functions which we not generated test cases are in blue color.

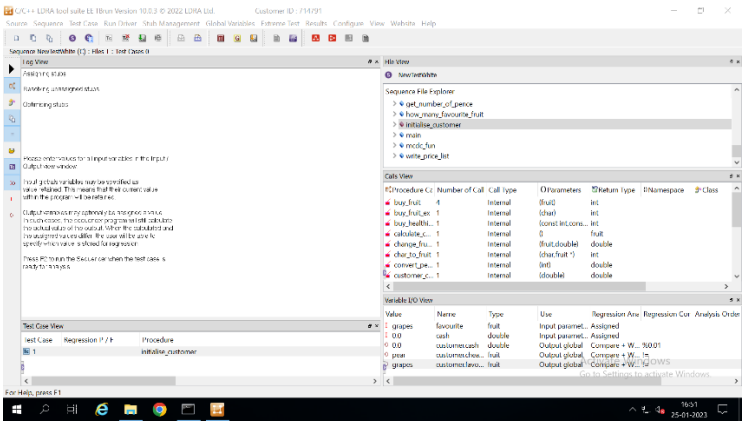


Step 6: Now we need to give the input and output values for the variables in the Variable I/O view. First, we must know the code of it. To view the code right click on the function and then select procedure code.

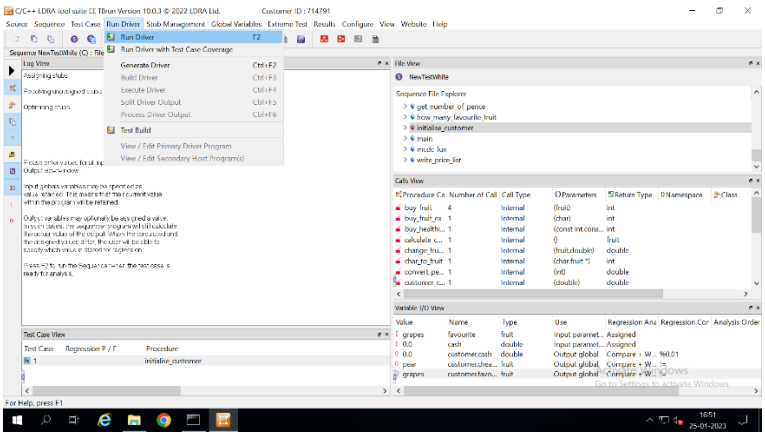




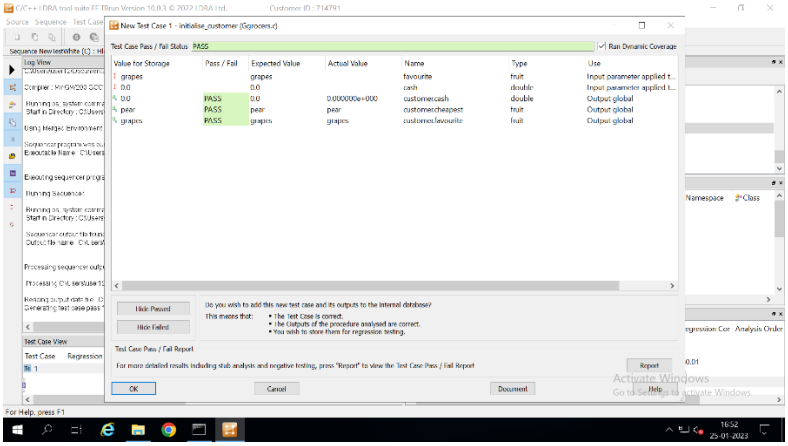
From this we will be able to understand the following
Favourite and customerfavouite is same
Customercheapest=pear
Cash=0.0
Customercash=0.0
Ex: This is the code we have so for this we will give the Input and Output values

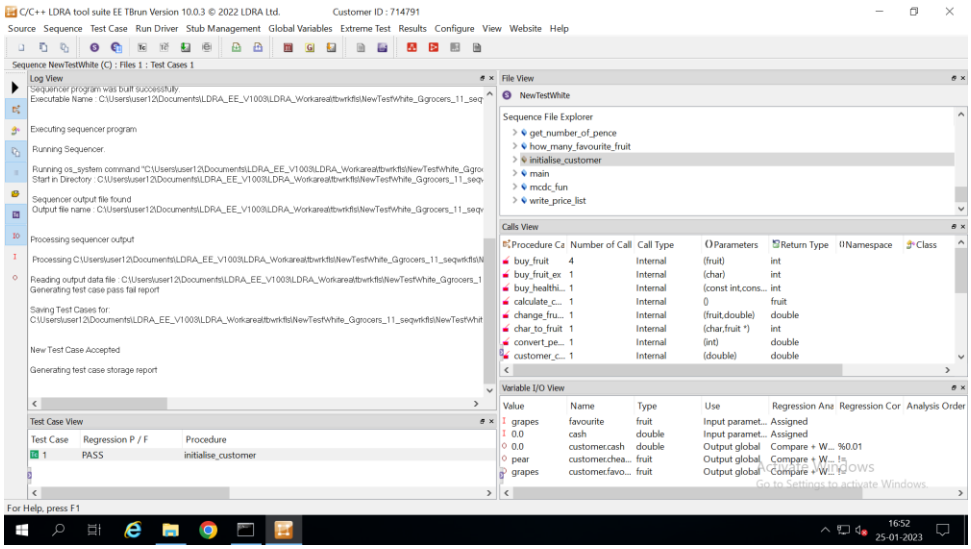


Step 7: Click on Run Driver in Run Driver or Press F2

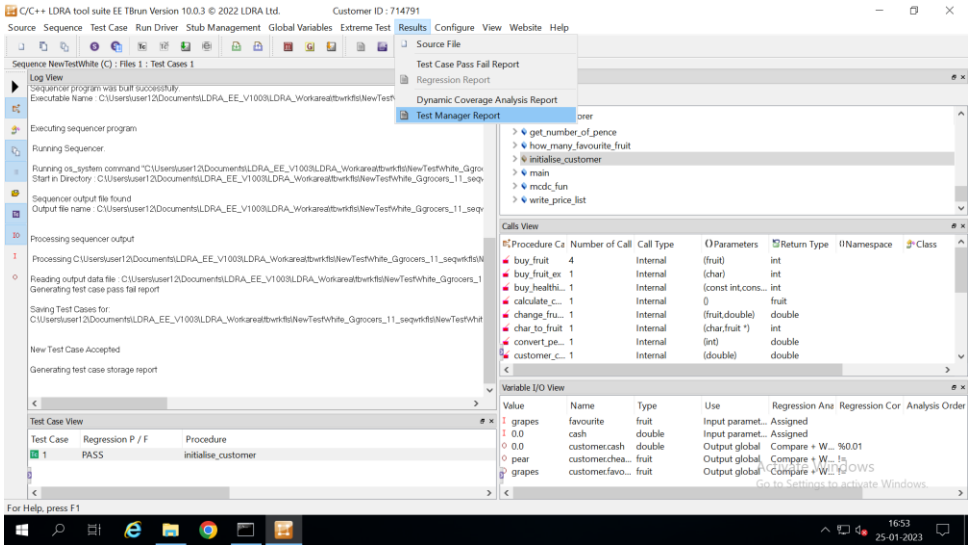


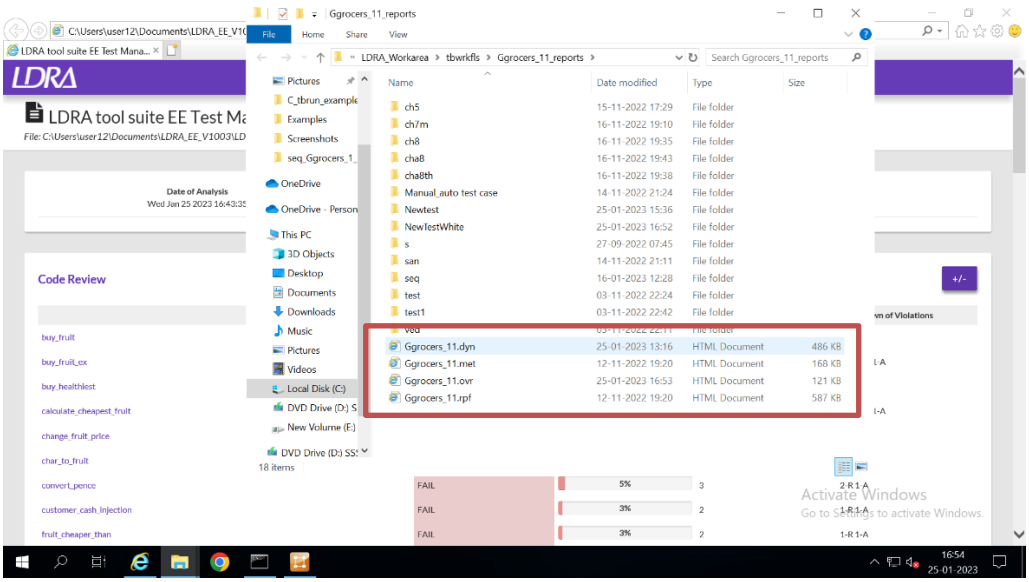
Step 8: Regression report will be generated.



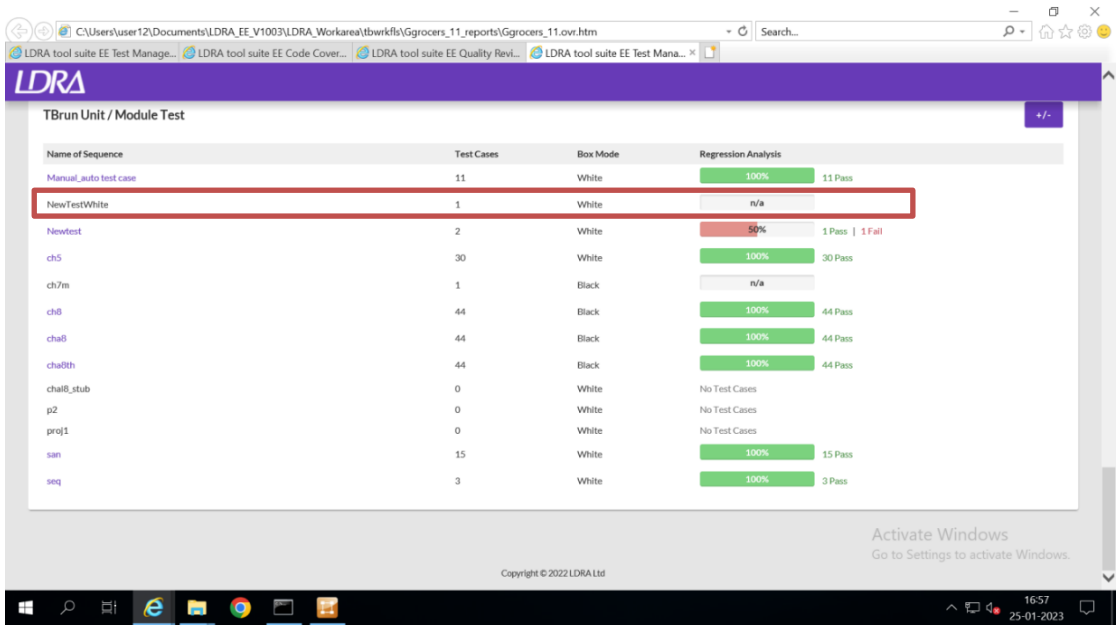


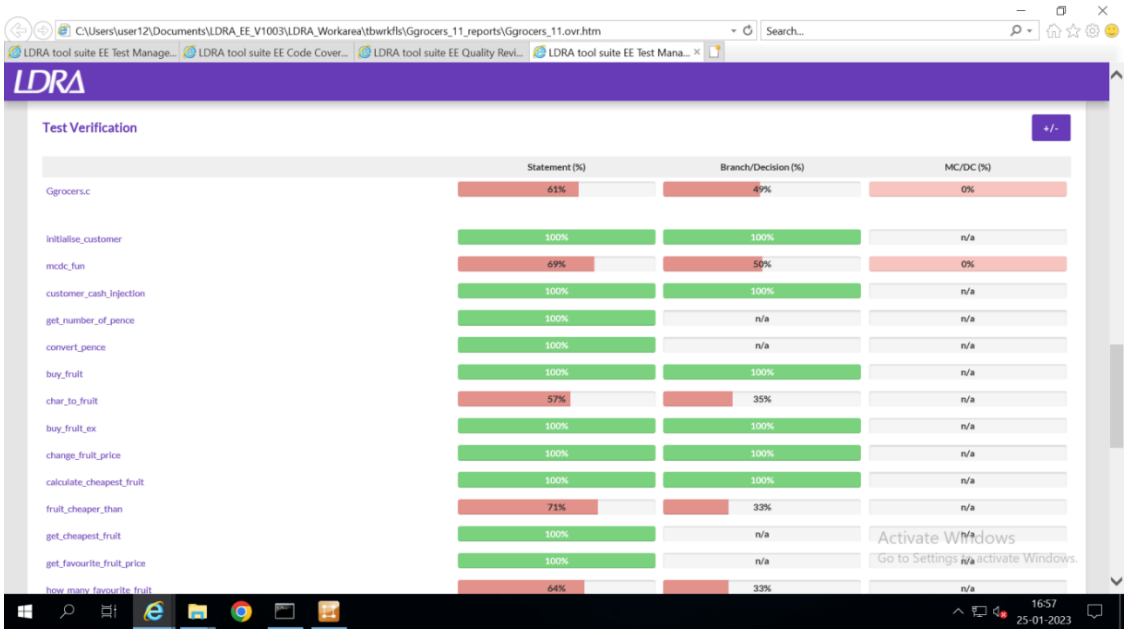
Step 9: To see the report click on Test Manager Report





Since White Box Testing focuses on internal functionality, the Regression Analysis is not applicable.





4.3.6 PICT (Pairwise Independent Combinational Testing) Tool

You can effectively create test configurations and test cases for software systems with the assistance of the Pairwise Independent Combinatorial Testing tool (PICT). In a fraction of the time needed for manual test case design, you can produce tests using PICT that are more efficient than manually generated tests. To obtain thorough combinatorial coverage of your parameters, you should employ the test cases generated by PICT, which are a condensed set of parameter value options.

Note: Conditional constraints let you define restrictions on the functionality of the domain.

As a command-line utility, PICT operates. Create a model file that includes a description of the interface's (or a set of configurations' or data's) specifications. To obtain thorough combinatorial coverage of your parameters, you should employ the test cases generated by PICT, which are a condensed set of parameter value options.

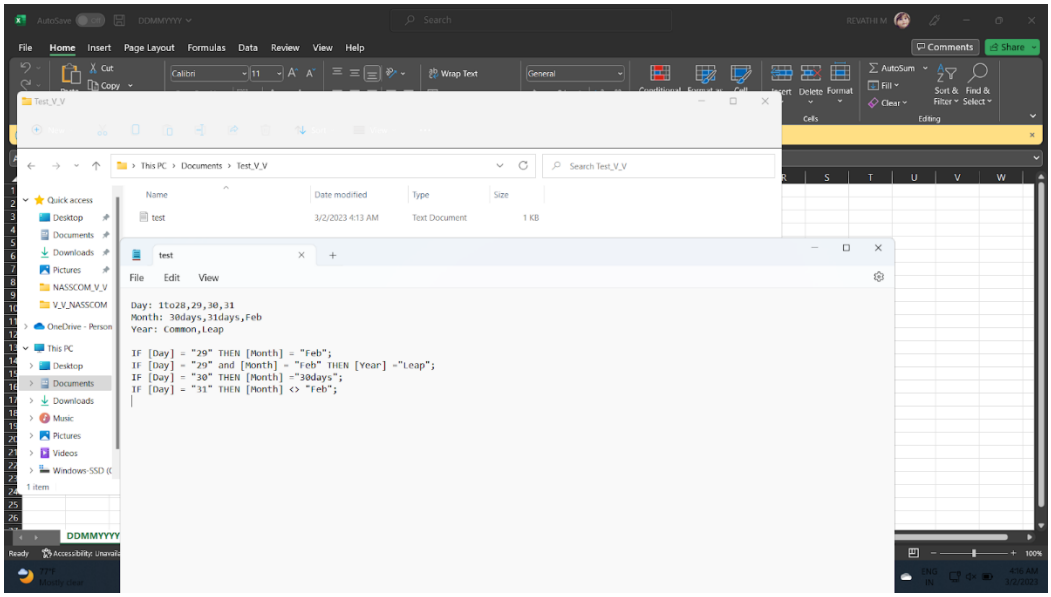
Installation procedure:

1. Double-click the.msi file after unzipping the attached file.
2. Follow the easy instructions to install.
3. To learn more about the features and usage after installation, browse the help file at C:/Program Files/PICT/PICTHelp.htm.

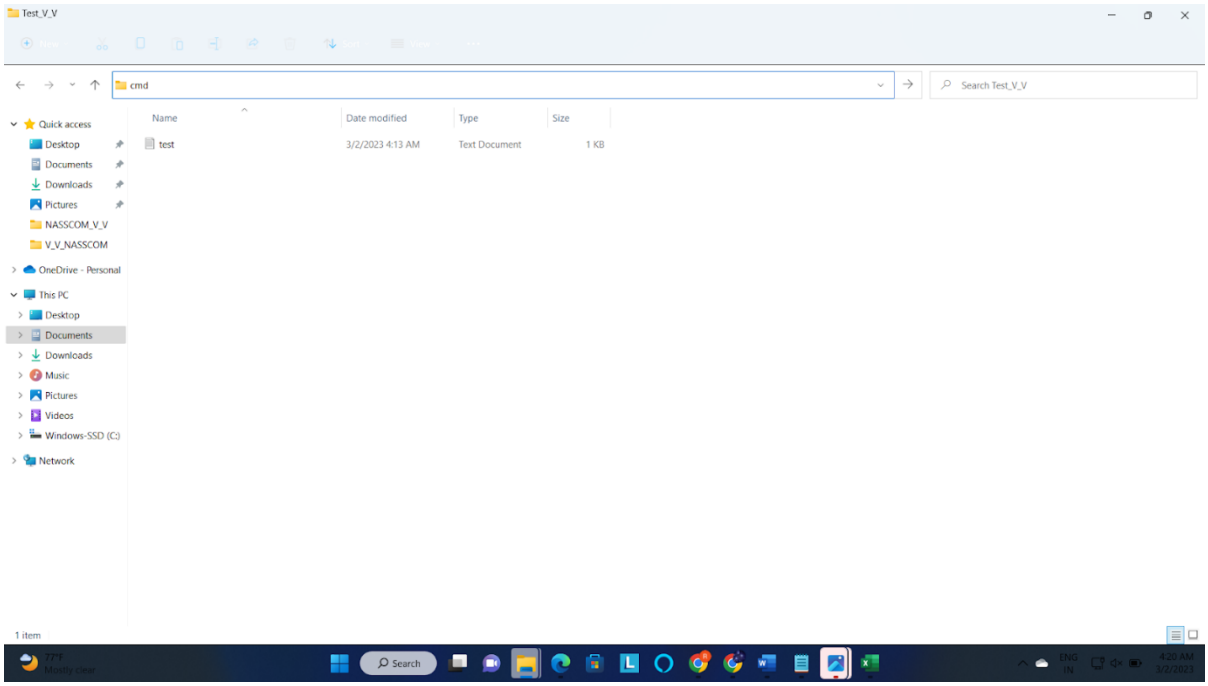
Note:

- Installation files can be downloaded from the URL https://balasegu.weebly.com/pict.html.
- To generate test cases for the constraints on online, use the following URL: https://pairwise.yuuniworks.com/

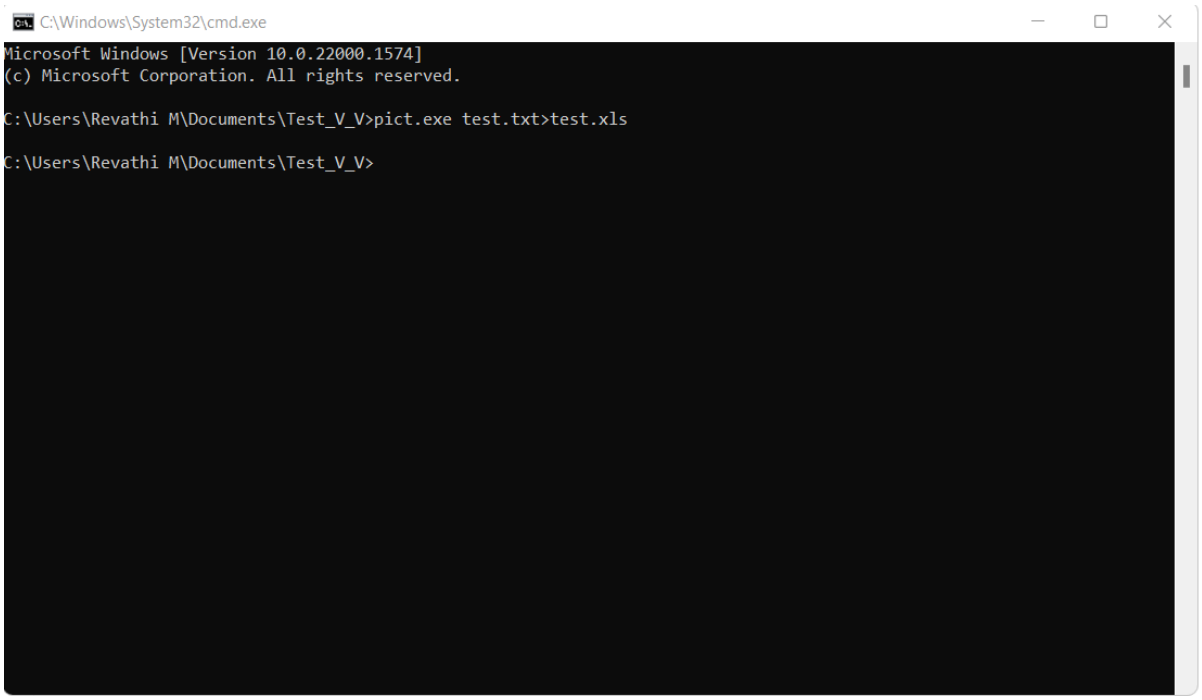
Step 1: Open the folder that contains the test file.



Step 2: Enter the command Prompt from the address bar of the current folder where the testing file is located.

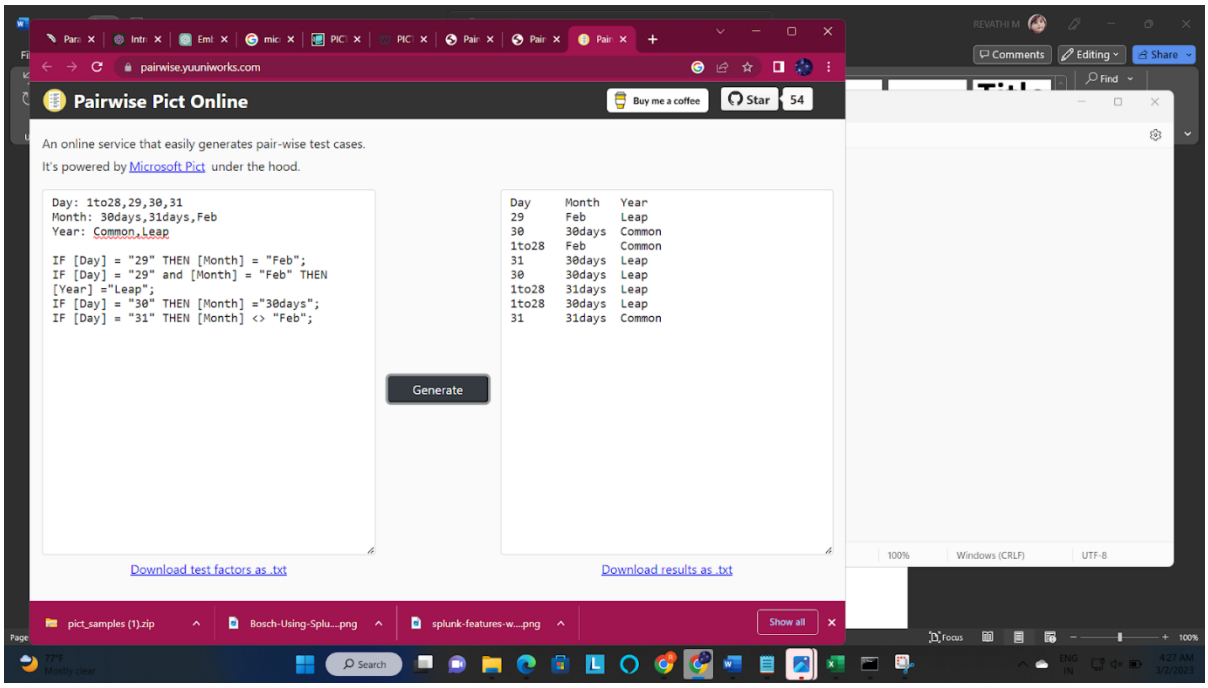


Step 3: Enter pict.exe test.txt(File to be tested)>test.xls(Test case file)



PICT Online:

Paste the code to be tested in the right side and click on generate. The test cases for the given test are shown on the right side.



Summary

In this unit, we learned about the following concepts:

1. White box testing
2. Different test coverage levels
3. Overview of statement coverage - C0
4. Overview of decision coverage - C1
5. The value of statement and decision coverage (Modified Coverage and Decision coverage MCDC) - C2
6. Usage of LDRA tool
7. Usage of PICT

References

https://pairwise.yuuniworks.com/
https://balasegu.weebly.com/pict.html

UNIT 4.4: Experience Based Test Techniques

Unit Objectives

At the end of this unit, you will be able to:

1. Explain error guessing
2. Explain exploratory testing
3. Explain checklist-based testing

4.4.1 Error Guessing

Error guessing is a software testing technique for guessing the error that can prevail in the code. It is an experience-based testing technique where the test analyst uses his or her experience to guess the problematic areas of the application. This technique necessarily requires skilled and experienced testers. It is a type of black-box testing technique and can be viewed as an unstructured approach to software testing.

Characteristics

- Black-Box
- Non-formal test design technique

Procedure

- Nominate experienced test engineers (domain and methodology)
- Specific documentation evaluation

What error should, I guess?

- Determine all the failure test cases for a given scenario.
- If pressure, velocity, and disk temperatures needed to be checked:
    - Test cases can be derived from the fact that increasing brake pressure reduces wheel velocity while increasing wheel temperature.
- For pressure controllers
    - Pressure, volume, and current tests where expected O/P is not known:
        - Test open/close valve timings for 1 ms and 9 ms and see if the O/P is within range or limits.
        - Test analogue valve operations and compare the results to those of digital valve operations.

4.4.2 Exploratory Testing

Exploratory testing is an approach to software testing that is often described as simultaneous learning, test design, and execution. It focuses on the discovery and relies on the guidance of the individual tester to uncover defects that are not easily covered in the scope of other tests. Testers and QA managers are encouraged to include exploratory testing as part of a comprehensive test coverage strategy. Exploratory testing speeds up documentation, facilitates unit testing, and helps create an instant feedback loop.

4.4.2.1 Why Should You Use Exploratory Testing?

When testing mission-critical applications, exploratory testing ensures you don't miss edge cases that lead to critical quality failures. Plus, use exploratory testing to aid the unit test process; document the steps and use that information to test extensively during later sprints. It is especially useful to find new test scenarios to enhance the test coverage.

4.4.2.2 Why You Should Not Use Exploratory Testing?

With any type of testing that is regulated, or compliance-based, scripted testing is the best way to go. In compliance-based testing, where certain checklists and mandates need to be followed for legal reasons, it is advised to stick to scripted testing. One example of this is accessibility testing, where several laws govern the testing protocol and there are defined standards that need to be passed. Exploratory testing complements QA teams' existing test strategy. It consists of a series of undocumented testing sessions to uncover previously unknown issues and bugs. When combined with automated testing and other testing practices, it increases test coverage, discovers edge cases, potentially adds new features, and overall improves the software product. With no structural rigidity, it encourages experimentation, creativity, and discovery within the teams.

4.4.2.3 Importance of Exploratory Testing CI/CD

The goal is to complement traditional testing to find million-dollar defects that are generally hidden behind the defined workflow. Exploratory testing opens testing to all key stakeholders, not just trained testers. Using an exploratory testing tool, one can capture screenshots, record voice memos, and annotate feedback during sessions. This enables faster and more efficient reviews by people beyond the traditional software tester.

4.4.3 Checklist - Based Testing

A test design technique based on experience in which an experienced tester uses a high-level list of items to be noted, checked, or remembered, or a set of rules or criteria against which a product must be verified.

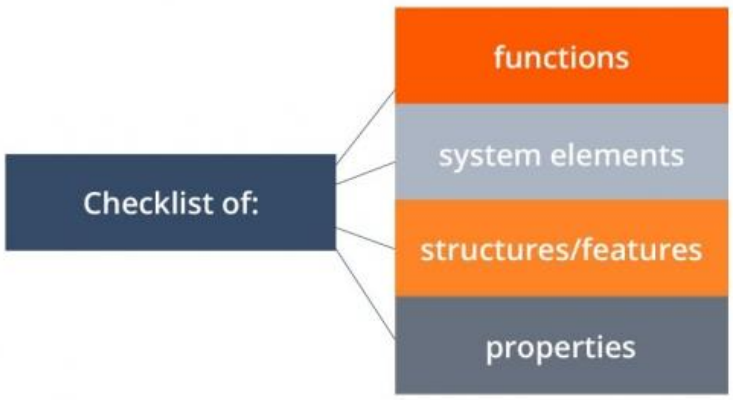


Fig 4.4.3 Commonly used Checklist

4.4.3.1 Checklist - Based Testing

Functional (black-box) checklists contain checks for the dominant functions of the complete system or for the definite functions of the lower levels.

System elements checklists (white boxes) examine sub-systems and modules of higher levels as well as special data items at secondary levels.

Structure/feature checklists path through various aspects, such as the list of customers and producers for definite sources or units sharing some average data, etc.

Properties checklists test fixed values such as definite specification units, code systems, etc.

4.4.3.2 Advantages

- Flexibility: This kind of checking can be used in all testing types.
- Easy to create: It is not difficult to create, use, and maintain a checklist.
- Analyzing the results: Checklists are easy to follow and examine.
- Team integration: The checklist can provide ready-made guidance and help new testing personnel integrate into their work.
- Controlling deadlines: This type of testing aids in controlling test completion and avoiding missing critical bugs before the deadline.

4.4.3.3 Disadvantages

- Different interpretations: QA engineers with various backgrounds can accomplish identical tasks using different approaches.
- "Holes" in coverage: It is difficult to capture all functional or structural components, especially those at higher levels.
- Item overlap: In trying to cover a large scope of material, there may be a duplication of the same information. This can lead to excessive testing.
- Reporting problems: Complex system components, functions, and their interactions are difficult or even impossible to illustrate, applying checklists.

Summary

In this unit, we learned about the following concepts:

1. Error guessing
2. When error Guessing can be done?
3. Exploratory testing
4. Checklist-based testing

UNIT 4.5: Test Design Techniques for Embedded Systems

Unit Objectives

At the end of this unit, you will be able to:

1. State transition testing
2. Control Flow test
3. Elementary comparison test
4. Classification-tree method
5. Evolutionary algorithms
6. Statistical usage testing
7. Rare event testing
8. Mutation analysis

4.5.1 Introduction

Ways in which we design the tests based on the understanding of requirements of the software.

Test Design Specifications (TDS) are an important part of the testing process for embedded software. A TDS is a document that outlines the specific tests that will be run on the software, including the test cases, test conditions, and expected results. When designing TDS for embedded software testing, it is important to consider the unique characteristics of the embedded system. These may include factors such as memory limitations, real-time constraints, and hardware dependencies.

To create effective TDS for embedded software testing, consider the following:

- **Identify the specific functionality to be tested**. This includes both functional requirements (what the software is supposed to do) and non-functional requirements (how the software is supposed to perform).
- **Determine the test environment**. This includes both the physical environment (where the embedded system is located) and the software environment (what other software is running on the system).
- **Define the test cases**. This includes the specific steps to be taken to execute each test case, as well as the expected results.
- **Consider the hardware dependencies**. The TDS should take into account any hardware dependencies, including the specific hardware components used in the embedded system.
- **Plan for error handling**. The TDS should outline how errors and exceptions will be handled, including any error messages that will be displayed to users.
- **Consider the testing tools and techniques**. The TDS should identify the specific tools and techniques that will be used to test the software, including any automation tools that may be used.

Overall, a well-designed TDS is critical for ensuring that embedded software is thoroughly tested and meets all necessary requirements. It helps to ensure that testing is comprehensive, efficient, and effective, ultimately leading to a higher quality and more reliable product.

Different Test Design Techniques:

- State transition testing
- Control flow test
- Elementary comparison test
- Classification-tree method
- Evolutionary algorithms
- Statistical usage testing
- Rare event testing
- Mutation analysis

4.5.2 State Transition Technique

Many embedded systems, or parts of embedded systems, show state-based behavior. In designing these systems, state-based modelling is used. Models composed during this process serve as the basis for test design. The purpose of the state-based test design technique is to verify the relationships between events, actions, activities, states, and state transitions. By using this technique, one can determine if a system's state-based behavior meets the specifications set for this system. The behavior of a system can be classified into the following three types:

Simple Behavior: The system always responds in the exact same way to a certain input, independent of the system's history.

Continuous Behavior: The current state of the system depends on its history in such a way that it is not possible to identify a separate state.

State-based Behavior: The current state of the system is dependent on the system's history and can clearly be distinguished from other system states. State-based behavior can be represented by using tables, activity charts, or state charts of these. State charts are the most common method of modelling state-based behavior and are often described using UML (the Unified Modelling Language).

4.5.2.1 Fault Categories

States

- States without incoming transitions
- Missing initial states
- An additional state
- A missing state
- A corrupt state

Guards

- A guard must point to a transition and not to a state
- Guards on completion-event transitions
- Guards on initial
- Overlapping guards
- Guard is false but a transition still takes place
- Incorrect implementation of the guard

Transitions

- Transitions must have an accepting and a resultant state
- Conflicting transitions
- Missing or incorrect transitions
- Missing or incorrect actions

Events

- Missing event
- Hidden paths
- A reaction takes place to an undefined event (also known as a “trap door”) (implementation fault).

Miscellaneous

- The use of synchronization in an orthogonal region

Characteristics

- Black-Box and White-Box

Procedure

- Establish State-Event-Table
- Establish State-Transition-Tree
- Determine legal test cases
- Determine illegal test cases
- Determine “Guard” test cases

All test paths in State-Transition-Tree

- One test path is a sequence of test steps
- Every test step needs:
    - Event (test stimulus)
    - Guard (precondition for test stimulus)
    - Action (system reaction)
    - State (resulting state)

4.5.3 UML Diagram Showing State-Change

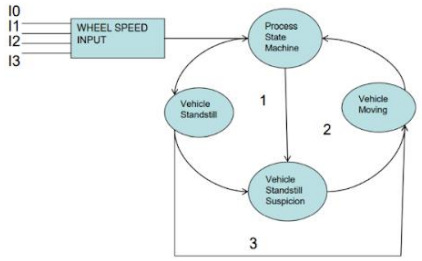


Fig 4.5.3 State Change Diagram

4.5.4 Control Flow Test (CFT)

Decision point A: (1, 2), (1, 3), (1, 4), (6, 2), (6, 3), (6, 4)

Decision point B: (2, 5), (3, 5), (4, 5), (2, 6), (3, 6), (4, 6)

All branch combinations with test degree 2: (1, 2), (1, 3), (1, 4), (2, 5), (2, 6), (3, 5), (3, 6), (4, 5), (4, 6), (6, 2), (6, 3), (6, 4)

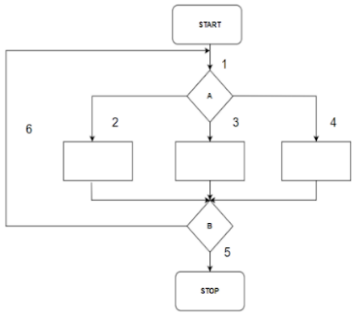


Fig 4.5.4 Control Flow Test

Specify test paths

- Test path = executable sequence of branches. All test paths will start with 1 and end with 5

Combination: (1, 2), (2, 5) → (1, 2, 5)

Combination: (1, 3), (3, 5) → (1, 3, 5)

Combination: (1, 2), (2, 6), (6, 2), (2, 5) → (1, 2, 6, 2, 5)

Combination: (1, 4), (4, 5) → (1, 4, 5)

Specify test paths

Combination: (1, 3), (3, 6), (6, 4), (4, 6), (6, 3), (3, 5) → (1, 3, 6, 4, 6, 3, 5)

Cancelling Redundant Test paths

(1, 2), (1, 3), (1, 4), (2, 5), (2, 6),(3, 5),

(3, 6), (4, 5), (4, 6), (6,2), (6, 3), (6, 4)

Test Paths Derived:

- Path 1: (1, 2, 5)
- Path 2: (1, 3, 5)
- Path 3: (1, 4, 5)
- Path 4: (1, 2, 6, 2, 5)
- Path 5: (1, 3, 6, 4, 6, 3, 5)

Specify test cases

- Choose physical values for the parameters
- The chosen values shall lead the system into the defined path
- Specify the expected result

4.5.4.1 Limitation of CFT

Control flow test has two severe disadvantages. The first is that the number of paths is exponential to the number of branches. For example, a function containing 10 if-statements (n) has (2n)1024 paths to test. Adding just one more if-statement doubles the count to 2048. The second disadvantage is that many paths are impossible to exercise due to relationships of data. For example, consider the following C/C++ code fragment:

```
if (success)
{
statement1;
statement2;
}
if (success)
{
statement3;
 }
```

Control flow graph considers this fragment to contain 4 paths. In fact, only two are feasible: success=false and success=true

4.5.5 Elementary Comparison Test (ECT)

It consists of the following steps:

- Analyze the functional specification
- Identify the test situation (conditions)
- Specify logical test cases
- Concretize the logical test cases (Use MCDC)
- Specify concrete test cases
- Specify the expected result

MCDC is the concept we use for ECT (Can be used for both Black box and white box testing)

4.5.6 Classification-tree Method (CTM)

It consists of the following steps:

- Identifying aspects of the test object
- Portioning input domain according to aspects
- Specifying logical test cases
- Constructing physical test cases
- Defining test actions

- Defining checkpoints
- Determining the start situation
- Establishing the test script

Example Adaptive Cruise Control

- System keeps vehicle on adjusted velocity (0-30 KMPH)
- System considers vehicle environment (0-90 KMPH) (0-40 KMPH – IR 'ON'))
- System avoids collision with vehicle driving ahead (Yes/ NO) (If Yes, consider distance up to 25m; If No, consider velocity difference (0 -30 KMPH)

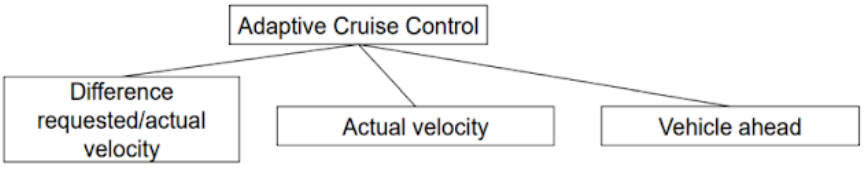


Fig 4.5.6 Specifying Test Objects

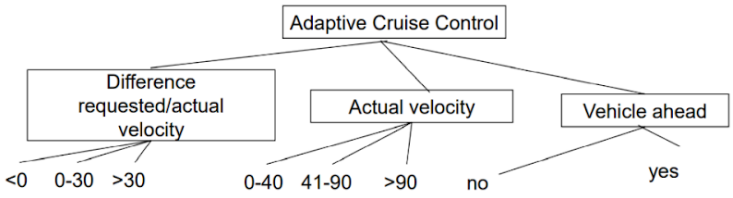


Fig 4.5.6 Equivalence Classes with Input Ranges

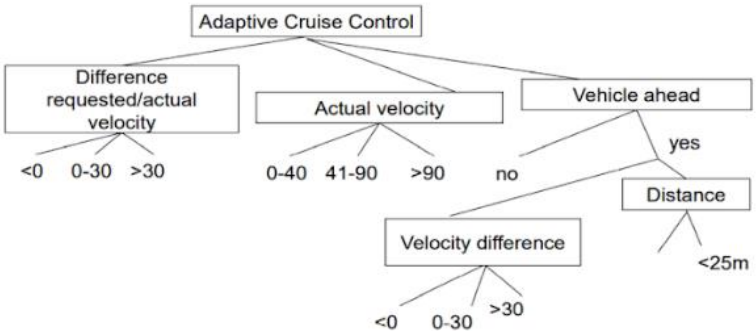


Fig 4.5.6 Equivalence Classes with Input Ranges (More Refined)

Deriving test cases (green dots), 4 test cases numbered 1-4

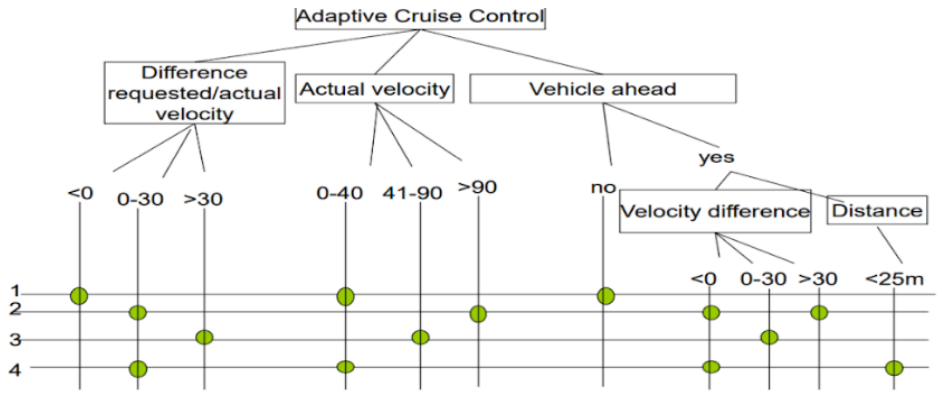


Fig 4.5.6 Deriving Test Cases

4.5.7 Evolutionary Algorithm

Evolutionary algorithms (also known as genetic algorithms) are search techniques and procedures based on the evolution theory of Darwin. Survival of the fittest is the driving force behind these algorithms. Evolutionary algorithms are used for optimization problems, or problems that can be translated into optimization problems. Whereas test design techniques focus on individual test cases, evolutionary algorithms deal with "populations" and the "fitness" of individual test cases. Evolutionary algorithms can, for instance, be used to find test cases for which the system violates its timing constraints or to create a set of test cases that cover a certain part of the code. Some application areas (Wegener and Groditmann (1998) and Wegener and Mueller (2001)) for evolutionary algorithms are:

- Safety testing.
- Structural testing.
- Mutation testing.
- Robustness testing.
- Temporal behavior testing.

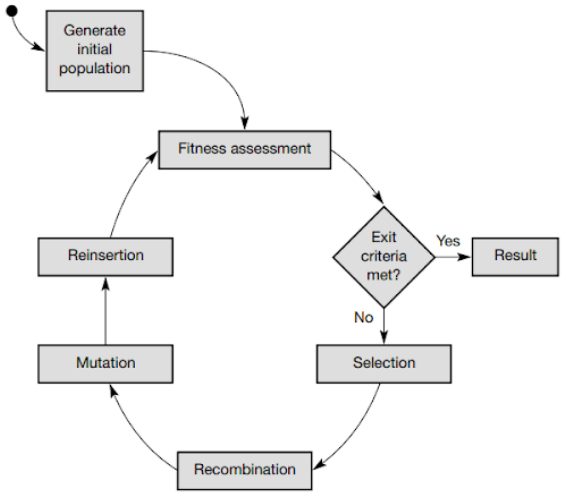


Fig 4.5.7 Evolutionary Algorithm

4.5.8 Statistical Test Analysis

The objective of a real-life test is to detect defects concerning the operational usage of a system. The test should therefore simulate the real situation as much as possible. Operational profiling is one method for describing a system's actual usage. An operational profile is a quantitative characterization of how a system will be used. Musa (1998) introduced the concept of operational profiles, and based on his work, Cretu (1997) and Woit (1994) developed some enhancements to the technique. Statistical usage testing uses operational profiles to produce a statistically relevant set of test cases.

Operational Profiles: Operational profiles deal with the use of a system. The user could be a human being or another system that triggers the system under test.

User Profile: A user is a person or a subsystem that uses the system. A user profile is a description of the usage of the system and the proportion of the customer group's usage it represents. Instead of

the proportion, it is possible to make an estimation of the relative importance of this profile. System mode profile A system mode is a set of functions that belong to each other.

Functional Profile: Each system mode is divided into functions. A list of functions is prepared, and the probability of occurrence is determined for each. Functions should be defined so that each represents a substantially different task. For every functional profile, an operational profile is created. The operational profile describes the usage of the functional profile.

4.5.9 Rare Event Testing

Statistical usage testing "ignores" events that happen very infrequently. It is often very important that these events are handled correctly by the system under test. These events, called rare events, are often related to safety aspects or to the main functionality of the system. Rare events do not deal with system failures. Rare event testing is like statistical usage testing executed close to the release date. If the main functionality of the system is to deal with these rare events (e.g., an ABS braking system for braking under slippery conditions), this handling of rare events must be tested much earlier on in the process.

The first steps of specifying rare event test cases are like those for statistical usage testing.

Customer Profile: The customer in this case is the one purchasing the system. The purpose of this step is to identify groups of customers who purchase the system for different reasons or for different usage. It only makes sense to identify these different groups if this influences rare events or rare event frequencies.

User Profile: A user is a person or a (sub)system that uses the system. The user can potentially trigger rare events.

System Mode: Profile Identifies the sets of functions that belong to each other.

Functional Profile: Each system mode is divided into functions. It is not useful to break down a system mode profile further if it is already known that there are no rare events related to it.

Operational Profile: For every functional profile, an operational profile is created. This is only done for a functional profile where rare events can occur. An operational profile is a matrix of histories of the systems and events. Every combination of a history and an event has a probability. Those combinations with a very small probability. Those combinations with a very small probability are of real importance here. The operational profile is the basis for the specification of the rare event test cases. If the operational profiles are already specified for the statistical usage test, then it is possible to reuse these profiles. Be aware that in those operational profiles the rare events may have been omitted (given a probability of zero) because they happen too rarely for statistical usage testing. It is best to re-evaluate the events that received a probability of zero.

All the history–event combinations with a very low probability are selected. Every combination is a test case. A test script for such a test case contains the following issues precondition, description of the rare event, and result state.

Thoroughness: Every test case is a combination of a historical state and a rare event. To increase the thoroughness of the test, a cascade of rare events can be tested. This is only possible if a rare event leads to a historical state of the system where another rare event is possible.

4.5.10 Mutation Analysis

With mutation analysis, faults caused on purpose must be detected. Fault seeding means introducing faults into a system under test. The testers of the system are just not aware of what the faults are. After execution of the test set, the fault detection rate is measured. This rate is equal to the number of faults found divided by the number of faults introduced. Mutation analysis at the module or class level is done in a quite different manner. The faults, called mutations, are introduced in a line of code; each mutation is syntactically correct. Only one mutation is introduced per module or class. Once a mutation is introduced, the program is compiled; this program is called a mutant. This mutant is tested with the test set, and after a set of mutants has been tested, the fault detection rate is measured. The rate is equal to the number of mutants found divided by the total number of mutants introduced. The basic concept of this method is that the fault detection rate of the test set is an indicator of the quality of the test. A test set with a high fault detection rate can give a reliable measure of the quality of the system under test or a good indication of the number of undetected faults in the system. Hence, it can be indicative of the confidence one can have in the system. The benefits and usefulness of this method are open to debate. The method makes no direct contribution to the quality of the system under test and merely gives an indication of the quality of the test set. If formal test design techniques are used, it is questionable if it makes any sense to use mutation analysis.

Summary

In this unit, we learned about the following concepts:

- Working of State transition testing
- Method to do control flow test
- Working of elementary comparison test
- Creation of test case using classification-tree method
- Evolutionary algorithms
- Statistical usage testing
- Method of rare event testing
- Working of mutation analysis

Summary

In this module "Test Techniques", we learned about the following concepts:

1. Classification of different test techniques
2. White box testing
3. Black box testing
4. Experience-based techniques
5. Difference between white-box and black-box testing
6. Various experience-based test design techniques
7. Use of the LDRA tool for Black Box and White Box Testing
8. Usage of the PICT tool for test case generation for given test factors

5. Testing Infrastructure

Unit 5.1 - Embedded Software Test Environments

Unit 5.2 - Tools Categorization of Test Tools

Unit 5.3 - Test Automation

Key Learning Outcomes

At the end of this module, you will be able to:

1. Explain about Embedded system software test environment
2. Explain about prototype tools which we use in Embedded System Software Testing
3. Explain about different tools which we use in the Embedded System Software Testing
4. Explain about python operators
5. Perform file operations using python
6. Explain about Boolean values and logical operators
7. Explain different data types used in python script
8. Explain about basic input and output operations in python

UNIT 5.1: Embedded Software Test Environments

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the simulation model
2. Explain the prototyping model and its tools
3. Explain the pre-production environment
4. Explain the post-production environment

5.1.1 Simulation

A simulation is a model that mimics the operation of an existing or proposed system, providing evidence or aiding in decision-making by being able to test different scenarios or process changes. This can be coupled with virtual reality technologies for a more immersive experience.

5.1.2 Prototype

The prototyping model is a system development method in which a prototype is built, tested, and reworked as necessary until an acceptable outcome is achieved from which the complete system or product can be developed.

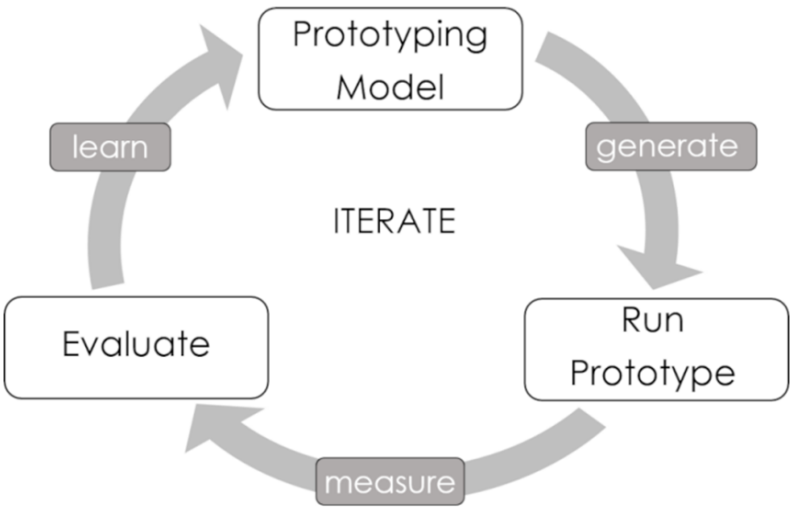


Fig. 5.1.2 Prototyping

5.1.2.1 Prototype Testing

Prototype testing is a way to evaluate the viability of a design in terms of how it can achieve the desired functionalities. It typically involves sharing a clickable prototype with multiple stakeholders so that various functions can be tested comprehensively.

Pre-development benefits of Prototype Testing

- It helps to evaluate the 'first impression' about the software.
- It allows designers to correct processes and other flaws early on.
- It validates the design process behind the functionality.
- It reduces the overall cost by capturing errors at an early stage.

Post-development benefits of Prototype Testing

- Major errors in the final software application will have been eliminated or minimized due to this testing.
- Any missing features or functions can be spotted and integrated into the design.
- It can help enhance the user experience by allowing the designers to perfect the flow before the product goes into development.

5.1.2.2 How to Test a Prototype?

There is a fixed framework for how a prototype should be tested, so it checks all the required boxes. In fact, the boxes themselves are the first step in preparing an app design for the prototype test.

1. Detail the specifies of your objective for testing

Why are you testing the app design or prototype in the first place? That should be the first question you ask yourself, because it will help you define the scope and objectives of conducting the test. You can break this down into functional and experiential goals, each of which is tested by the targeted user. This will give you valuable information about what to change and how to change it when the results of the test come out.

2. Define the tester demographic

You need to identify the ideal persona before testing your design prototype. Now that you have your objectives defined, this part should be easy. It involves picking the ideal user demographic for your application and then qualifying potential testers against this persona. For example, what age group will be using the app, what is the ideal monthly income profile or educational profile of the end-user, and so on. You will need to ask questions like, 'How familiar are they with this general category of product?', 'Are my testers already familiar with the OS platform we are targeting?', etc. Once you have defined the user demographic, you can move to the next stage.

3. Break your testing into specific stages

Also known as staging the prototype, this is critical component of the testing phase because it gives users a specific set of functions to test out. For instance, you may want to test a paper prototype by using information architects to validate the flow of some features. Alternatively, you may want to test a Hi-Fi prototype to see what type of user experience it leads to. This allows you to be more thorough in your testing because you are approaching it from a dual 'feeling and function' point of view.

4. Define the tasks for the testers

Testers often need very specific instructions in terms of what goals they need to accomplish on the prototype. The 'just check it out and tell me what you think' approach does not work here. They need to know what features to test and what actions to execute within the scope of that feature. On the UX side, you can ask them to fill out a questionnaire about how they felt while using the app. Doing both these things will give you more comprehensive results that are useful. These results should be able to guide your nest steps to improve the experience or refine a particular feature.

5. Clinical testing versus real-world testing

Knowing the difference between how an app will be used under specific test conditions and out in the real world can give you valuable insight into actual usability aspects. In a formal app prototype test, a special space is usually created for users so they can test the app without any distractions. Doing A/B testing in this manner will give you tremendous insights into how an app behaves under different conditions. For instance, how does an interruption in the mobile network affect a particular feature of the app? Are the error messages appropriate? Does the app hang and force the user to quit it? These are critical pieces of information that you normally wouldn't get under ideal test conditions.

6. Data gathering and analysis

How are you collecting data regarding the usability of the prototype? Are you using screen capture software to track finger contact and other gestures? Are you simply making a recording of the user testing the app? These questions will help you outline your data capture mechanisms, organize data, and analyze it so it yields actionable information to improve the user experience and correct flaws in the app before it goes into development. This brings us to the final part of planning the test.

5.1.2.3 Prototyping Tools

Tool Name	Suited for	Compatibility	Free Trial or Free Version	Price
Figma	Building interactive prototypes, with the help of some user-friendly tools.	Cloud, SaaS, Web, Mac/Windows desktop, Android/iOS mobile, iPad.	A free version is available.	Starts at $12 per editor per month
InVision	Creating Hi-Fi interactive prototypes, within minutes.	Cloud, SaaS, Web, Android/iOS mobile, iPad.	A free version is available.	Starts at $4 per user per month.
Adobe XD	Developing wireframes as well as Hi-Fi interactive prototypes, with the help of some advanced features.	Cloud, SaaS, Web, Mac/Windows desktop, iOS/Android mobile.	A free trial is available for 7 days.	Starts at $9.99 per user per month
Sketch	Building interactive prototypes, while having real-time collaboration with your team and others.	Cloud, SaaS, Web, Mac desktop, iPhone.	A free trail is offered for 30 days.	Starts at $9 per editor per month.
Framer	Creating Hi-Fi interactive prototypes, at affordable prices.	Cloud, SaaS, Web.	A free version as well as a free trail is available.	Starts at $15 per month

5.1.3 Pre-Production

Pre-production environments are where your team builds and tests software for the digital service. Your pre-production environments will most likely include

- One or more development environments where your developers can build and experiment with new software.
- An integration environment where you can combine all the code from your development environments and see if it works as intended.
- A staging environment where you can do most of your testing in an environment that closely mimics the live site (or production environment).

5.1.3.1 Primary Pre-Production Environments

Pre-production environments are usually built before applications go into production. The two primary environments are development and test environments.

Development environment - The development environment provides a layer on which software engineers can build and test their code. These tests are usually limited and focus mostly on unit-style tests.

Test environment - Test environments are usually where QA engineers will run a long list of test cases to make use the code acts as expected.

5.1.3.2 Maintaining your Pre-Production Environment

Pre-production environments are not static. They don't remain pristine with multiple developers constantly testing, editing, and updating them. This can lead your various environments to un-sync and cause problems in the future.

Automation

Your pre-production processes should have automatic checks to ensure that each layer is staying in sync as required. For instance, implementing automated processes that scan your databases and code bases for changes in the files or data.

Code repository practices

This is one of the simplest ways to ensure that both pre-production and active production environments are staying in sync. Your team should not be editing code in production or test environments. This allows developers to edit code in the development environment and see, at least theoretically, how it will interact once live.

Version control

Tracking the many interactions of a single project allows you to revisit previous various versions and ensures that your code base doesn't change unless it is meant to.

5.1.4 Post-Production

By definition, "post" means after, and "production release" refers to deployment to live or production environments.

5.1.4.1 Objectives of Post-Production

The objective is to verify the release in production and live environments.

What tasks and activities are included in the post-release verification phase?

Post-production release tasks and activities generally include:

- Post-production release verification
- Report verification results
- Reporting any issues found on production
- Post-release verification data cleanup
- Post-release monitoring (if applicable)

Who creates the post-production release test plan?

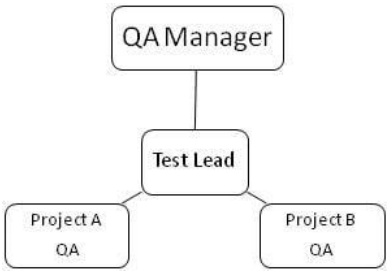


Fig 5.1.4.1 Post-production Test Plan

Who approves the post-production release test plan?

This will vary across companies and depend on the organization's structure. Again, considering the same organisational structure as shown in the previous question, the post-production release test plan should be reviewed and approved by the test lead or QA manager.

When do we create the post-production release verification plan?

The post-production release test plan can be created anytime during the software development cycle after the requirements, development scope, and impact areas are identified and locked. It is usually easier for the QA to create the post-production release test plan midway through the sprint. That ensures that there is enough time for review and approval. It is good practise to include this test plan with any formal QA sign-off documents before the project enters the deployment and release phases.

What else do we need to know about the post-production release verification process?

- It is important to set clear expectations regarding the scope and purpose of post-release verification.
- Stakeholders (internal and external) should be made aware of the following:
    - The team cannot test everything in production.
    - The team cannot squeeze days' worth of testing into the few hours set aside for post-release verification.
- Therefore, the testing in production would be essentially based on the approved post-production release test plan.

5.1.4.2 Limitations

Due care should be taken while deciding the extent of post-production release testing. There are limitations to what and how much we can test in production. Production environments have live client data that needs to be handled very carefully. Additional planning should be done for changes that involve data migration, updating, deletion, etc.

Example #1): For an eSurvey company, if testing involves answering and submitting the survey, QA will need to send a request to delete the test survey after verification so as to not impact the client survey collection data and their statistics.

An Example:

Project Overview:

The following changes need to be made to a social media application, specifically to the sign-up process.

- Last name field validation needs to be removed. It was previously implemented as 'Last name should have a minimum of 4 characters" (improvement for an existing field).
- Implement a toggle button next to the email address so that the user can configure the privacy settings for the email address that will be displayed on their profile (new feature request).
- The user should be able to choose their avatar (new feature request).
- Reduce API calls during the sign-up process to improve application performance (improvement)

5.1.4.3 Post-Production Release Verification Plan

S.No.	Description	Expected Result	Status	Comments
1	Go to Livesiteurl	Website homepage should load successfully	Pass	
2	Click on Sign up as new user	User should be redirected to the registration/sign up page	Pass	
3	Fill in the required fields and click on Register button Note: -Enter last name as 'Lee' -Toggle the privacy button to Do not Display -Chose an Avatar	• User should be redirected to their Profile page after successful registration. • User phone number should not be shown • User selected Avatar should show	Partial Pass	Avatar is not rendering properly and is showing as broken image. Reported in JIRA as BUG-1088
4	Monitoring - Verify if the application performance has improved after this release	Reduction of API calls during Sign up process should improve application performance	Ongoing	Action is on Dev Lead and Dev Ops team to monitor application for 24 hours
5	Post release cleanup	Delete the test account created	Done	

Summary

In this unit, we learned about the following concepts:

1. Simulation models
2. How prototyping helps to design a product and launch it?
3. Various prototyping tools
4. The various prototyping models and their tools
5. The pre-production environment
6. The post-production environment and its limitations
7. Objectives of post production
8. Various post production with their release

Reference

https://www.twi-global.com/technical-knowledge/faqs/faq-what-is-simulation#:~:text=A%20simulation%20is%20a%20model,for%20a%20more%20immersive%20experience

UNIT 5.2: Tools Categorization of Test Tools

Unit Objectives

At the end of this unit, you will be able to:

1. Explain the categorization of test tools

5.2.1 Embedded Software Testing Tools

Embedded testing concepts have much in common with application software testing. However, the comparison of application validation and embedded system testing methods reveals some important differences between the two methodologies. Embedded developers often have access to hardware-based embedded software testing tools that are generally not used in app development and the testing of applications. Embedded systems often have unique characteristics that should be reflected in the test plan. These differences tend to give embedded system testing its own distinctive flavor. Below is a list of the top embedded software testing tools that help companies improve their software development procedures.

5.2.1.1 Tessy

Fig 5.2.1.1 Tessy

Tessy by Hitex Development Tools is a big player in the embedded software testing and development tools market. Tessy can test code written in C and C++ in the embedded environment. It is often used for version verification with multiple standards. Tessy is used by test engineers to configure and execute automated tests and easily generate test reports. Tessy includes the Classification Tree Method used for test specifications. The support of HTML, Word, and Excel test documentation also comes in handy.

5.2.1.2 Egg Plant

Fig 5.2.1.2 Test Plant

EggPlant tools created by TestPlant are designed to work with non-standard software, e.g., embedded software, that can't be installed on the test system. EggPlant covers a wide range of tasks, including functional and performance testing. EggPlant tools work perfectly on the stack with other tools, which is very useful when trying to adapt them for teams that already have a determined toolset. EggPlant can also be used to run tests in the system without installing any code on the server. Not being dependent on the underlying code when running with nonstandard technology is eggPlant's strong suit. In other words, EggPlant can run on virtually any system. The tool also allows for manual direction during test execution. The "Ask and Answer" commands let testers interact with the automated test execution, add input, and designate other behaviors dynamically. By taking advantage of manual interaction capabilities, testers can use eggPlant where most other automated tools are useless due to the extreme complexity of functionality. EggPlant's dynamic testing capabilities make automated testing more creative and spontaneous.

5.2.1.3 Parasoft

Fig 5.2.1.3 Parasoft

Parasoft's DTP is a great automation tool for teams that use continuous development strategies in the IDE or on the target for testing embedded software. The tool integrates into any embedded testing environment and provides automatic reports for statistical and historical purposes at the component level. Parasoft DTP includes tools to perform static analysis, code reviews, code coverage analysis, and even traceability. The tool captures results based on open-source testing frameworks (Google Test, CppUTest, etc.). Parasoft follows the general trend of providing preconfigured industry-specific and regulatory-specific settings and standards for code compliance purposes. Parasoft templates are fully customizable to meet the end user's needs. Having a customized tool helps build on traceability and other tracking or monitoring features required for compliance verification for safety-critical systems.

5.2.1.4 Klocwork

Fig 5.2.1.4 Klockwork

KlocWork Insight offers popular static testing tools for embedded software development. It automatically identifies bugs in the code and holes in security systems. Security issues in applications leave many holes for attacks that may result in terrible consequences, from minor overloading of the application to a complete shutdown. This tool assists with the integration and testing of the application at the component level. KlocWork Insight includes built-in systems for the following standards: CWE, CWE/SANS Top 25, CERT, OWASP, DISA STIG, and MISRA.

5.2.1.5 Vector Software

Fig 5.2.1.5 Vector Software

Vector Software provides embedded software test tools specifically designed for unit and integration testing. The "toolchain" method is used in Vector's software. It supports a total cross-development environment that includes:

- Cross-compiler
- Target board
- Real-time OS
- Debug emulator

Vector's tool includes configurable target integrations for each toolchain an embedded software code system needs. Vector Software is a big player on the market for automated software testing tools, especially when it comes to safety-critical embedded applications. VectorCast automation and management capabilities include a wide range of tests, including unit-level, integration-level, and system-level testing. Vector Software's tools are fully certified for use in building and testing regulated applications. It's a clear advantage for companies that work with medical devices or on aviation software projects, where security standard compliance is essential.

Summary

In this unit, we learned about the following concepts:

1. Various categorization of test tools and their role in embedded software testing.

UNIT 5.3: Test Automation

Unit Objectives

At the end of this unit, you will be able to:

1. Understand the basics of Python
2. Explain the data types, variables, basic input-output operations, and basic operators
3. Explain the Boolean values and logical operations
4. Determine the functions in Python and file handling

5.3.1 Introduction to Python

Python is a high-level, general-purpose, and very popular programming language. Python programming language (the most recent version is Python 3) is used in web development and machine learning applications, as well as all cutting-edge software technology. The Python programming language is very well suited for beginners as well as for experienced programmers with other programming languages like C++ and Java. Python is a dynamic, interpreted (bytecode-compiled) language. There are no type declarations of variables, parameters, functions, or methods in the source code. This makes the code short and flexible, and you lose the compile-time type checking of the source code. Python tracks the types of all values at runtime and flags code that does not make sense as it runs.

Facts about Python Programming Language

Python is currently the most widely used multi-purpose, high-level programming language. Python allows programming in object-oriented and procedural paradigms. Python programs are generally smaller than programs in other programming languages like Java. Programmers have to type relatively less, and the indentation requirement of the language makes them readable all the time.

The Python language is being used by almost all tech-giant companies like Google, Amazon, Facebook, Instagram, Dropbox, Uber, etc. The biggest strength of Python is its huge collection of standard libraries, which can be used for the following:

- Machine Learning
- GUI applications (like Kivy, Tkinter, PyQt, etc.)
- Web frameworks like Django (used by YouTube, Instagram, and Dropbox)
- Image processing (like OpenCV or Pillow)
- Web scraping (like Scrapy, BeautifulSoup, and Selenium)
- Test frameworks
- Multimedia
- Scientific computing
- Text processing, and many more.

5.3.2 Data Types

Python is a loosely typed language. Therefore, there is no need to define the data type of variables. There is no need to declare the variables before using them.

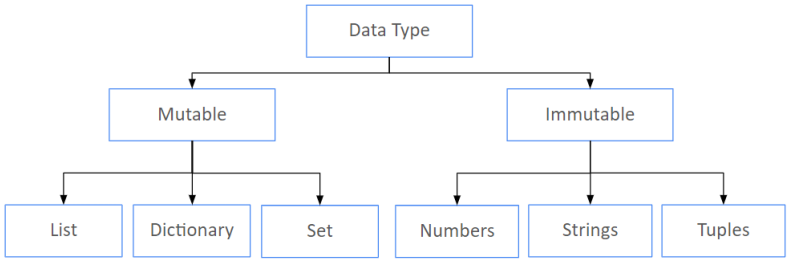


Fig 5.3.2 Data Types

5.3.3 Data Types: Immutable

Immutable objects in Python can be defined as objects that do not change their values and attributes over time. These objects become permanent once created and initialized, and they form a critical part of the data structures used in Python. Python is used for numbers, tuples, strings, frozen sets, and user-defined classes, with some exceptions. They cannot be changed, and their values and nature remain constant once initialized, thus the name "immutable."

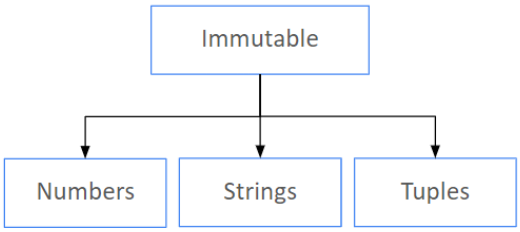


Fig 5.3.3 Immutable Data Types

5.3.3.1 Data Types: Numbers

Numeric data types are numbers stored in database columns. These data types are typically grouped by exact numeric types; values where the precision and scale need to be preserved the exact numeric types are integer, bigint, decimal, numeric, number, and money.

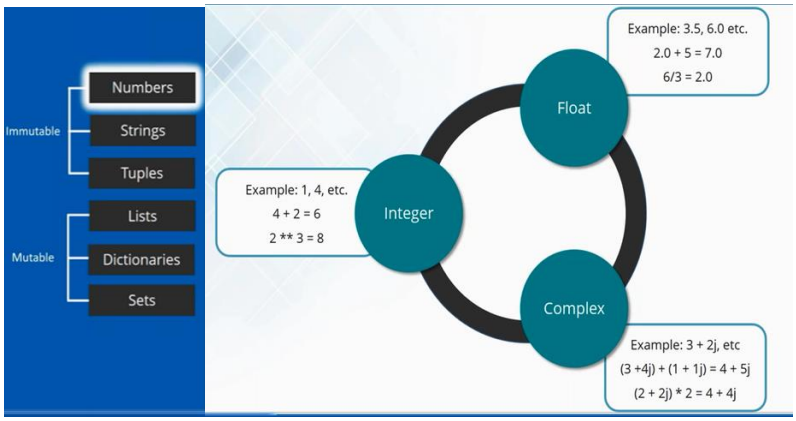


Fig 5.3.3.1 Numbers

5.3.3.2 Data Types: String

Strings in Python are identified as a contiguous set of characters represented in quotation marks. Python allows either a pair of single or double quotes. The slice operator ([] and [:]) can be used to take subsets of strings, with indexes starting at 0 at the beginning of the string and working their way to -1 at the end. For both positive and negative number indexing, the slice operator will take one data point before that indexing.

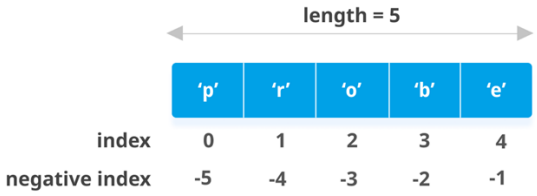


Fig 5.3.3.2 String

String Slicing Example

```
1  x='hello'
2  print(x[1:4])
3  print(x[2:-1])
4  print(x[:-1])
5  print(x[1:])
```

one value before to index value

And the corresponding output is,

```
C:\Users\kr\PycharmProjects\
ell
ll
hell
ello
```

5.3.3.2.1 Data Types: String Operation

String operation	Explanation	Example
+	Adds two strings together	x = "hello " + "world"
*	Replicates a string	x = " " * 20
upper	Converts a string to uppercase	x.upper()
lower	Converts a string to lowercase	x.lower()
title	Capitalizes the first letter of each word in a string	x.title()
find, index	Searches for the target in a string	x.find(y) x.index(y)
rfind, rindex	Searches for the target in a string, from the end of the string	x.rfind(y) x.rindex(y)
startswith, endswith	Checks the beginning or end of a string for a match	x.startswith(y) x.endswith(y)
replace	Replaces the target with a new string	x.replace(y, z)
strip, rstrip, lstrip	Removes whitespace or other characters from the ends of a string	x.strip()
encode	Converts a Unicode string to a bytes object	x.encode("utf_8")

Fig 5.3.3.2.1 String Operation

String Operation	Explanation	Syntax
Capitalize	Capitalizes first letter of string	str.capitalize
Title	first characters of all the words are capitalized.	str.title()
Count	number of occurrences of substring sub in the range [start, end].	str.count(sub, start=0, end=len(string))
Endswith	if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.	str. endswith(suffix, beg=0, end=len(string))
Find	Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.	str. find(str, beg=0, end=len(string))
Index	Same as find() but raises an exception if str not found.	str.index(str, beg=0, end=len(string))
Whether the string is alphanumeric, alphabets or digits	Returns true if string has alphanumeric, alphabets and digits respectively	isalnum() isalpha() isdigit()

str.capitalize(): The Python String capitalize() method returns a copy of the string that only capitalizes the first character.

str.count(sub, start=0, end=len(string)): Counts the number of times str appears in the string or a substring of the string if the starting and ending indexes are given.

str.endswith(suffix[, start[, end]): The endswith() Python string method returns True if the string ends with the specified suffix, False otherwise, with the matching optionally limited by the start and end indices.

str.find(str, beg=0 end=len(string)): If starting index beg and ending index end are provided, this function will determine if str occurs in a string or a substring of a string, returning index if found and -1 otherwise.

str.index(str, beg=0, end=len(string)): Same as find(), but raises an exception if str is not found.

str.isalnum(): Python string method **isalnum()** checks whether the string consists of alphanumeric characters.

str.isalpha(): Python string method **isalpha()** checks whether the string consists of alphabetic characters only.

str.isdigit(): Python string method **isdigit()** checks whether the string consists of only digits.

str.islower(): Python string method **islower()** checks whether all the case-based characters (letters) of the string are lowercase.

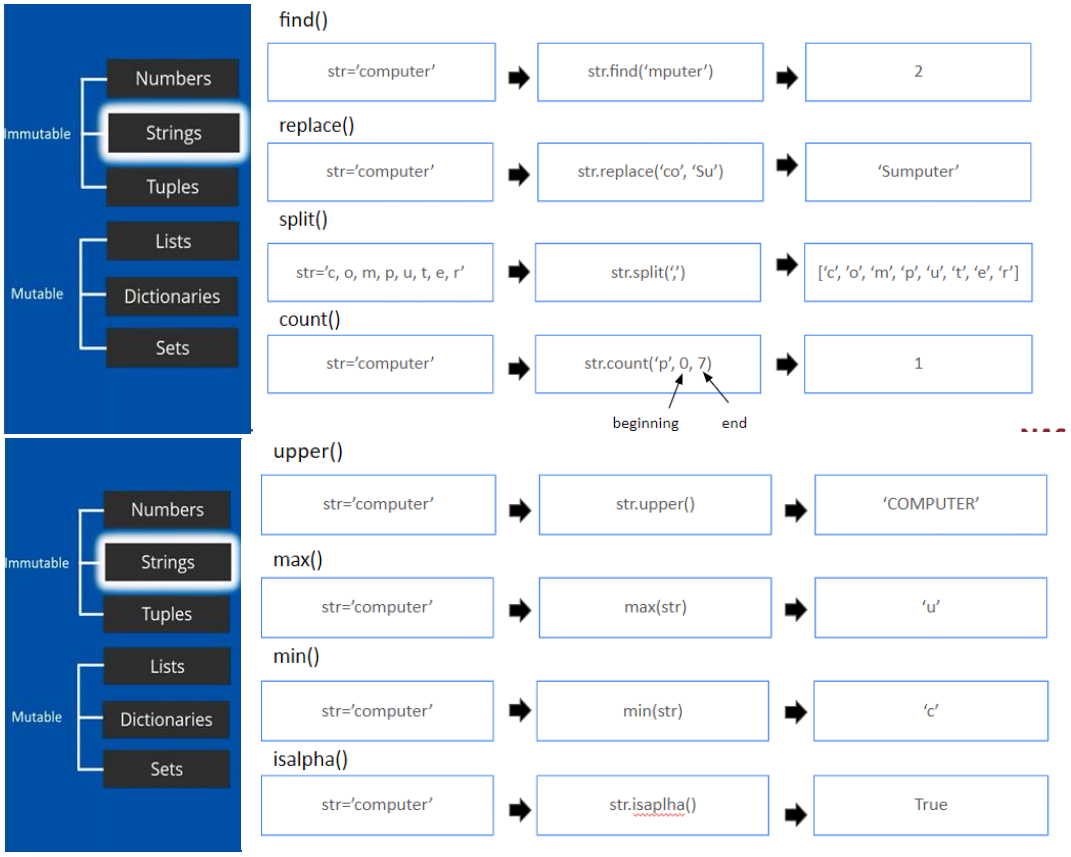


Fig 5.3.3.2.1 Strings-sequence Operations

5.3.3.2.2 Strings Example

str = 'Hello World!'

print (str) # Prints complete string

print (str[0]) # Prints first character of the string

print (str[2:5]) # Prints characters starting from 3rd to 5th

print (str[2:]) # Prints string starting from 3rd character

print (str * 2) # Prints string two times

print (str + "TEST") # Prints concatenated string

5.3.3.3 Data Types: Tuple

Tuples can be a collection of various data types, and unlike simpler data types, conventional methods of getting the type of each element of a tuple are not possible. A tuple type is an array with a predefined length and predefined types in each index position in the array.

```
myTuple = ('name', 2.4, 5, 'Python')
```

Fig 5.3.3.3 Tuple

#Empty Tuple is created by
x =()
x=tuple()

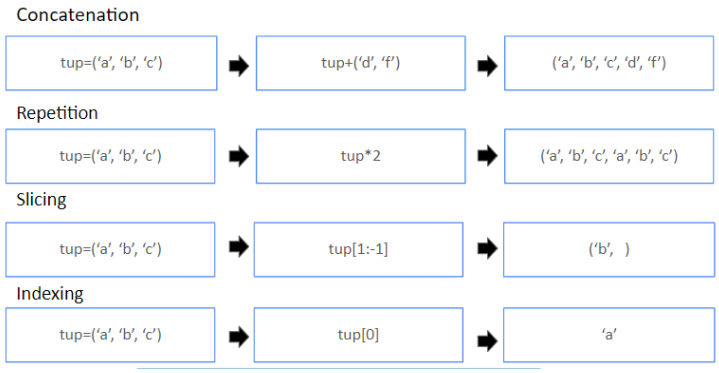


Fig 5.3.3.3 Tuple Sequence Operation

- Indexing works for both positive & negative values

5.3.3.3.1 Tuple Example

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')
print (tuple) # Prints complete tuple
print (tinytuple * 2) # Prints tuple two times
print (tuple + tinytuple) # Prints concatenated tuple
print (tuple[0]) # Prints first element of the tuple
print (tuple[1:3]) # Prints elements starting from 2nd till 3rd
print (tuple[2:]) # Prints elements starting from 3rd element
```

Why Tuple is not Updated-an Illustration?

```
x=(123,'john')
print(x)
print(id(x))
print(x*2)
print(id(x))
print(x)
x=x*2
print(x)
print(id(x))
```

Output:

```
(123, 'john')
36127568
(123, 'john', 123, 'john')
36127568
(123, 'john')
(123, 'john', 123, 'john')
31323360
```

5.3.4 Data Types: Mutable

Mutable means something is changeable or can change. In Python, "mutable" is the ability of objects to change their values. These are often the objects that store a collection of data.

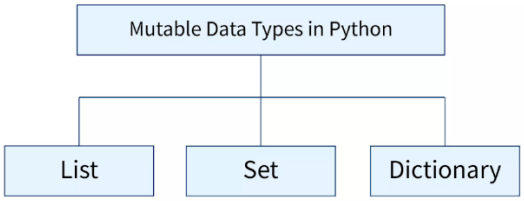


Fig 5.3.4 Mutable Data Types

5.3.4.1 Data Types: List

A list is a sequence of mutable Python objects like integers, floating-point numbers, and string literals. A list can be modified and updated. A list can be defined in square brackets.

```
myList = ('name', 2.4, 5, 'Python')
```

Fig 5.3.4.1 List

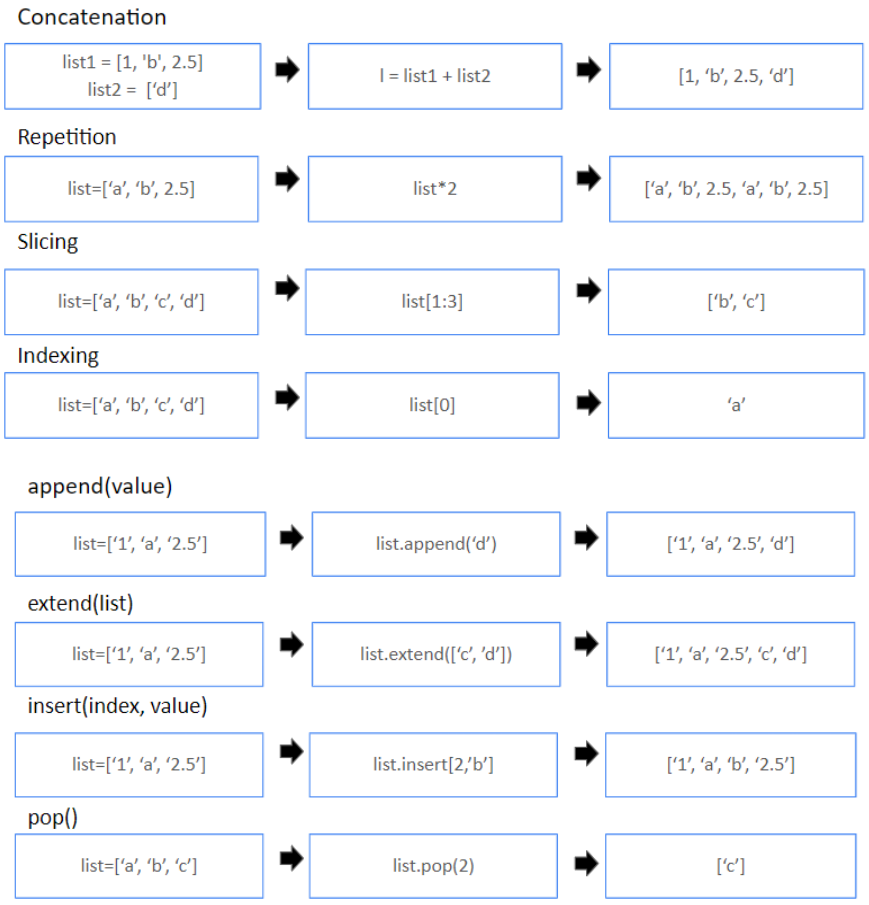


Fig 5.3.4.1 List Sequence operation

Note:

- In the append method, only one argument can be given.
- When using concatenation, repetition, slicing, and indexing methods, we can't update and modify the list.
- When using methods like append, insert, pop, and extend, we can update or modify the list.

Examples of lists:

list = ['abcd', 786 , 2.23, 'john', 70.2]
tinylist = [123, 'john']
print (list) # Prints complete list
print (tinylist * 2) # Prints list two times
print (list + tinylist) # Prints concatenated lists
print (list[0]) # Prints first element of the list
print (list[1:3]) # Prints elements starting from 2nd till 3rd
print (list[2:]) # Prints elements starting from 3rd element

5.3.4.1.1 Difference between Tuple and List

```
list1=['abc','def','ghi','jkl',34,78,90.65]
list2=['yhju']
print(list2[1:2])
print(list1+['xyz'])#concat operation adds 'xyz' to the list 'list1'
print(list1)# element xyz is not updated in the list name 'list1'
print(list1[1:2])
print(list1[3])
print(list1*3)
list=[10,20,30]
print(list.append(40))

x=[1,2,3,'po']
y=x.append(4) #append operation adds [4  ] to the list 'x'
print(x) # elements [4  ] is updated in the list name 'x'(shows that list is mutable python object)
print(y) # the statement x.append([4  ]) returns no value
```

Output:

```
[]
['abc', 'def', 'ghi', 'jkl', 34, 78, 90.65, 'xyz']
['abc', 'def', 'ghi', 'jkl', 34, 78, 90.65]
['def']
jkl
['abc', 'def', 'ghi', 'jkl', 34, 78, 90.65, 'abc', 'def', 'ghi', 'jkl', 34, 78, 90.65, 'abc', 'def', 'ghi', 'jkl', 34, 78, 90.65]
None
[1, 2, 3, 'po', 4]
None
```

List is not Updated with Concatenation

List is Updated with append

```
tup=('abc','def','ghi','jkl',34,78,90.65)
tup1=('yhju',)
print(tup1[1:2])
print(tup+('xyz',))#concatenate element 'xyz' with tuple name 'tup'
print(tup)            #element xyz is not updated in the tuple name 'tup'
print(tup[1:2])
print(tup[3])
print(tup*3)
```

Output:

```
()
('abc', 'def', 'ghi', 'jkl', 34, 78, 90.65, 'xyz')
('abc', 'def', 'ghi', 'jkl', 34, 78, 90.65)
('def',)
jkl
('abc', 'def', 'ghi', 'jkl', 34, 78, 90.65, 'abc', 'def', 'ghi', 'jkl', 34, 78, 90.65, 'abc', 'def', 'ghi', 'jkl', 34, 78, 90.65)
```

Tuple is not updated

5.3.4.2 Data Types: Dictionary

In a dictionary, values are stored and fetched by key. A dictionary key can be an integer or a string datatype. Values: can be any arbitrary Python datatype (such as list, number, string, tuple, dictionary, etc.).

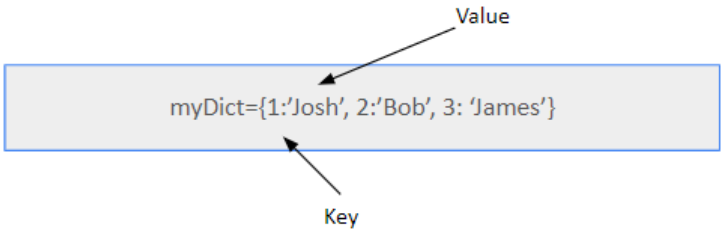


Fig 5.3.4.2 Dictionary

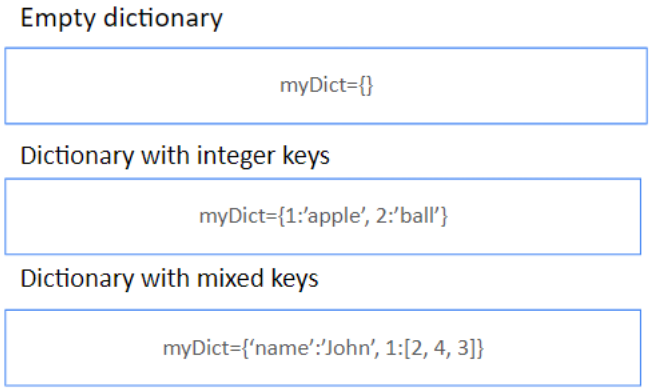


Fig 5.3.4.2 Dictionary Example

5.3.4.2.1 Dictionary Examples

dict = {}
dict['one'] = "This is one"
dict[2] = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
print (dict['one']) # Prints value for 'one' key
print (dict[2]) # Prints value for 2 key
print (tinydict) # Prints complete dictionary
print (tinydict.keys()) # Prints all the keys
print (tinydict.values()) # Prints all the values

```
dict1={1:'a',2:'b'}
z=dict1.update({3:'h'})
print(dict1)   #here dictionary is updated
print(z)
```

Output:

```
{1: 'a', 2: 'b', 3: 'h'}
None
```

```
myDict={}
print(myDict)
myDict = {1: "735269","name" : "Rani Niveda", "age":25,"list":[10,20,30],tuple:(100,200,300)}
print(len(myDict))
print(myDict[1])
print(myDict)
print(myDict["list"])
print(myDict[tuple])
print(myDict.keys())
print(myDict.values())

print(myDict.items())
print(myDict.get(2))
print(myDict)
#print(myDict.get("list"))
myDict.update({"Lang":"Python"})
print(myDict)
print(myDict.pop("age"))
myDict.pop("Lang")
print(myDict)
```

Output:

```
{}
5
735269
{1: '735269', 'name': 'Rani Niveda', 'age': 25, 'list': [10, 20, 30], <class 'tuple'>: (100, 200, 300)}
[10, 20, 30]
(100, 200, 300)
dict_keys([1, 'name', 'age', 'list', <class 'tuple'>])
dict_values(['735269', 'Rani Niveda', 25, [10, 20, 30], (100, 200, 300)])
dict_items([(1, '735269'), ('name', 'Rani Niveda'), ('age', 25), ('list', [10, 20, 30]), (<class 'tuple'>, (100, 200, 300))])
None
{1: '735269', 'name': 'Rani Niveda', 'age': 25, 'list': [10, 20, 30], <class 'tuple'>: (100, 200, 300)}
{1: '735269', 'name': 'Rani Niveda', 'age': 25, 'list': [10, 20, 30], <class 'tuple'>: (100, 200, 300), 'Lang': 'Python'}
25
{1: '735269', 'name': 'Rani Niveda', 'list': [10, 20, 30], <class 'tuple'>: (100, 200, 300)}
```

5.3.4.3 Difference between .get() function and myDict[1] method (accessing dict.)

1) If key is not found in the dictionary, then accessing dict. mydict [2] is found. Then it raises a key error.

```
mydict={}
print(mydict)
mydict={1:'735269','name':'Rani Niveda','age':25,'list':[10,20,30],tuple:(100,200,300)}
print(len(mydict))
print(mydict[2])
```

Output:

```
Traceback (most recent call last):
{}
5
  File "C:/Users/karthick/PycharmProjects/jbkkn/bn.py", line 5, in <module>
    print(mydict[2])
KeyError: 2
```

If key is not found in the dictionary, we can use the "get" function to find it. Then it allows you to provide None value.

```
mydict={}
print(mydict)
mydict={1:'735269','name':'Rani Niveda','age':25,'list':[10,20,30],'tuple':(100,200,300)}
```

print(len(mydict))
print(mydict.get(2))
print(mydict.get(2,4))
Output:
{}
5
None
4 (returns default value 2nd parameter)

Updating Dict. with same keys:

```
n = {'name': 'john','code':6734, 'dept': 'sales'}
n.update({'code':7878})
print(n)
```

updating dict with same keys

Output:

```
C:\Users\kr\PycharmProjects\untitled4\venv\Scripts\python.exe
{'name': 'john', 'code': 7878, 'dept': 'sales'}
```

5.3.4.4 Data Types: Set

A set is a mutable object, but the elements in the set are immutable Python objects like numbers, strings, and tuples.

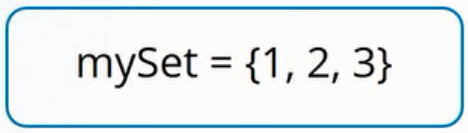


Fig 5.3.4.3 Set

#Empty set is created by:
x= set()

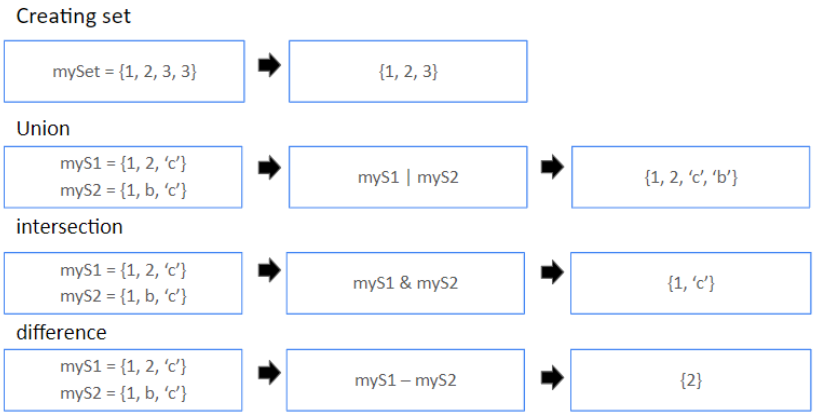


Fig 5.3.4.3 Set

5.3.4.4.1 Set Example

```
myset={1,2,3,3}
print(myset)
myS1={1,2,'c'}
myS2={1,'b','c'}
print(myS1 | myS2)
print(myS1 & myS2)
print(myS1 - myS2)
print(myS2 - myS1)
```

Output:

```
set([1, 2, 3])
set([1, 2, 'b', 'c'])
set([1, 'c'])
set([2])
set(['b'])
```

5.3.5 Difference Between Mutable and Immutable Objects

Mutable Objects	Immutable Objects
A mutable object can be changed after it is created	An immutable object cannot be changed after it is created
Examples: List, set, dictionary	Examples: tuples, int, float, bool, frozenset
Mutable objects are not considered as thread-safe in nature	Immutable objects are regarded as thread-safe in nature
Mutable Objects are slower to access, as compared to immutable objects	Immutable objects are faster to access when compared to mutable objects
Mutable objects are useful when we need to change the size or contents of our object	Immutable objects are best suitable when we are sure that we don't need to change them at any point in time
Changing mutable objects is a cheaper operation in terms of space and time	Changing immutable objects is an expensive operation since it involves creating a new copy for any changes made

5.3.6 Python Variables

A variable is nothing, but a programming element used to define, store, and perform operations on the input data. A Python variable is a symbolic name that is a reference or pointer to an object. Once an object is assigned to a variable, you can refer to the object by that name. But the data itself is still contained within the object. Python variables are of four different types: integer, long integer, float, and string. Integers are used to define numeric values. Long integers are used for defining integers with longer lengths than a normal integer. Floats are used for defining decimal values. Strings are used for defining characters. A variable name must begin with a letter of the alphabet or an underscore.

Example:

abc=100 #valid syntax
_abc=100 #valid syntax
3a=10 #invalid syntax
@abc=10 #invalid syntax

The first character can be followed by letters, numbers, or underscores.

Example:

a100=100 #valid
_a984_=100 #valid
a9967$=100 #invalid
xyz-2=100 #invalid

Python variable names are case sensitive.

Example:

a100 is different from A100.
a100=100
A100=200

Reserved words cannot be used as variable names.

Example: break, class, try, continue, while, if

break=10 //invalid
class=5 //invalid
try=100 //invalid

5.3.7 Input and Output Functions

In Python, using the input() function, we take input from a user, and using the print() function, we display output on the screen. Using the input() function, users can give any information to the application in string or number format.

Python 3 includes the following two built-in functions for handling user and system input:

- input(prompt): To accept input from a user.
- print(): To display output on the console/screen.

In Python 2, we can use the following two functions:

- input([prompt])
- raw_input([prompt])

5.3.7.1 Print() Function

In Python, the print() function is used to display results on the screen. The syntax for print() is as follows:

Example:

print (“string to be displayed as output ”)
print (variable)
print (“String to be displayed as output ”, variable)
print (“String1 ”, variable, “String 2”, variable, “String 3” ……)

Example:

>>>print (“Welcome to Python Programming”) Welcome to Python Programming
>>>x = 5
>>>y = 6
>>>z = x + y
>>>print (z)
11
>>> print (“The sum = ”, z)
The sum = 11
>>> print (“The sum of ”, x, “ and ”, y, “ is ”, z)
The sum of 5 and 6 is 11

The print() function evaluates the expression before printing it on the monitor. The print() symbol displays the entire statement specified by the print() symbol. To print more than one item, a comma (,) is used as a separator in print().

5.3.7.2 Input() Function

In Python, the input() function is used to accept data as input at run time. The syntax for the input() function is: Variable = input ("prompt string"). In syntax, a "prompt" string is a statement or message to the user indicating what input is acceptable. If a prompt string is used, it is displayed on the monitor, and the user can provide the expected data from the input device. The input() function takes whatever is typed from the keyboard and stores the entered data in the given variable. If the prompt string is not given in input(), no message is displayed on the screen, and thus the user will not know what is to be typed as input.

Example 1:

input( ) with prompt string
>>>city=input (“Enter Your City: ”)
Enter Your City: YYYYY
>>>print (“I am from “, city)
I am from YYYYY

Example 2:
input( ) without prompt string
>>> city=input()
XXXXXX
>>> print ("I am from", city)
I am from XXXXXX

5.3.8 Operator

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

Example:
>>> 2+3
5
Here, + is the operator that performs addition.
2 and 3 are the operands and 5 is the output of the operation.

5.3.9 Python Operator

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

5.3.9.1 Arithmetic Operators

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) –	9//2 = 4 and 9.0//2.0 = 4.0, 11//3 = 4, 11.0//3 = 4.0

Example of Arithmetic Operator:

```
a = 21, b = 10
print ("Line 1 - Value of c is ", a + b)
c = a - b
print ("Line 2 - Value of c is ", c)
c = a * b
print ("Line 3 - Value of c is ", c)
c = a / b
print ("Line 4 - Value of c is ", c)
c = a % b
print ("Line 5 - Value of c is ", c)
a = 2
b = 3
c = a**b
print ("Line 6 - Value of c is ", c)
a = 10
b = 5
c = a//b
print ("Line 7 - Value of c is ", c)
```

5.3.9.2 Comparison (Relational) Operators

These operators compare the values on either side of them to determine their relationship. They are also called relational operators.

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Example of Comparison (Relational) Operator:

```
a = 21
b = 10
if ( a == b ):
print ("Line 1 - a is equal to b")
else:
print ("Line 1 - a is not equal to b")
if ( a != b ):
print ("Line 2 - a is not equal to b")
else:
print ("Line 2 - a is equal to b")
if ( a < b ):
print ("Line 3 - a is less than b" )
else:
print ("Line 3 - a is not less than b")
if ( a > b ):
print ("Line 4 - a is greater than b")
else:
print ("Line 4 - a is not greater than b")
if ( a <= b ):
print ("Line 5 - a is either less than or equal to b")
else:
print ("Line 5 - a is neither less than nor equal to b")
if ( b >= a ):
print ("Line 6 - b is either greater than or equal to b")
else:
print ("Line 6 - b is neither greater than nor equal to b")
```

5.3.9.3 Assignment Operators

Assignment operators are used in Python to assign values to variables. a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left. There are various compound operators in Python, like a += 5 that adds to the variable and later assigns the same. It is the same as a = a + 5. Assume variable a holds 10 and variable b holds 20, then

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a

-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Example of Assignment Operator:

```
a = 3
b = 2
c = a + b
print ("Line 1 - Value of c is ", c)
c += a
print ("Line 2 - Value of c is ", c )
c *= a
print ("Line 3 - Value of c is ", c )
c /= a
print ("Line 4 - Value of c is ", c )
c %= a
print ("Line 5 - Value of c is ", c)
c **= a
print ("Line 6 - Value of c is ", c)
c //= a
print ("Line 7 - Value of c is ", c)
```

5.3.9.4 Python Logical Operators

Operator	Description	Example
and Logical AND	If both the operands are true, then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

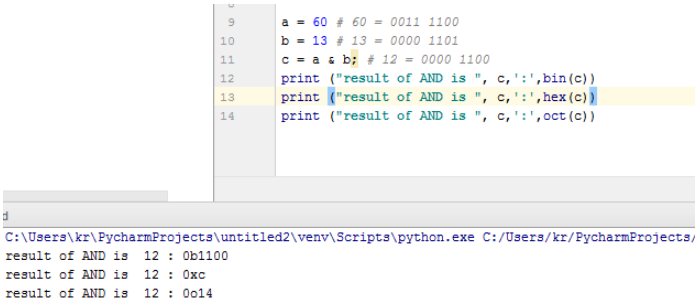
Example of Logical Operator:

```
x=True
y=false
print(x and y)
print(x or y)
print(not y)
```

5.3.9.5 Bitwise Operators

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) = 12 (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Example of Bitwise Operator:



```
a = 60 # 60 = 0011 1100
b = 13 # 13 = 0000 1101
c = a & b; # 12 = 0000 1100
print ("result of AND is ", c,':',bin(c))
print ("result of AND is ", c,':',hex(c))
print ("result of AND is ", c,':',oct(c))

C:\Users\kr\PycharmProjects\untitled2\venv\Scripts\python.exe C:/Users/kr/PycharmProjects/
result of AND is  12 : 0b1100
result of AND is  12 : 0xc
result of AND is  12 : 0o14
```

5.3.9.6 Python's Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples.

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not find a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

5.3.9.7 Python's Identity Operators

Identity operators compare the memory locations of two objects.

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

5.3.10 Python Operator Precedence

S.No.	Operator & Description
1	****** Exponentiation (raise to the power)
2	**~ + -** Complement, unary plus and minus (method names for the last two are +@ and -@)
3	*** / % //** Multiply, divide, modulo and floor division
4	**+ -** Addition and subtraction
5	**>> <<** Right and left bitwise shift
6	**&** Bitwise 'AND'
7	**^ |** Bitwise exclusive `OR' and regular `OR'

8	**<= < > >=** Comparison operators
9	**<> == !=** Equality operator
10	**= %= /= //= -= += *= **=** Assignment operators
11	**is** and **is not** are Identity operators
12	**in** and **not in** are Membership operators
13	**not, or, and** are Logical operators

5.3.11 Python Boolean Values

In programming, you often need to know if an expression is True or False. You can evaluate any expression in Python and get one of two answers: True or False. When you compare two values, the expression is evaluated, and Python returns the Boolean answer.

Program:

```
a = 200
b = 33

if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Output:

```
b is not greater than a
```

The bool() function evaluates any value and returns True or False in response.

Program:

```
print(bool("Hello"))
print(bool(15))
```

Output:

```
True
True
```

5.3.11.1 'not' Boolean Logical Operation

The only Boolean operator with one argument is not. It takes one argument and returns the opposite result: False for True and True for False. Here it is in a truth table:

A	not A
True	False
False	True

A	not A	Identity	Yes	No
True	False	True	True	False
False	True	False	True	False

Identity: Since this operator simply returns its input, you could just delete it from your code with no effect.

Yes: This is a short-circuit operator since it doesn't depend on its argument. You could just replace it with True and get the same result.

No: This is another short-circuit operator since it doesn't depend on its argument. You could just replace it with False and get the same result.

5.3.11.2 'and' Boolean Logical Operation

The **and** operator takes two arguments. It evaluates to False unless both inputs are True.

A	B	A and B
True	True	True
False	True	False
True	False	False
False	False	False

5.3.11.3 'or' Boolean Logical Operation

The value of the **or** operator is True unless both of its inputs are False.

A	B	A or B
True	True	True
False	True	True
True	False	True
False	False	False

5.3.11.4 Other Operator

Equality and Inequality: The most common comparison operators are the equality operator (==) and the inequality operator (!=). It's almost impossible to write any meaningful amount of Python code without using at least one of those operators.

is operator: The **is** operator checks for object identity. In other words, x is y evaluates to True only when x and y evaluate to the same object. The is not operator is the inverse of the is operator.

in operator: The **in** operator checks for membership. An object can define what it considers members.

5.3.12 Functions

- Function – Function call and definition
- Types of Function
- Built in Functions
- User defined Functions
- Types of Arguments
- Lambda Function
- Diff between lambda and user defined func
- Recursive functions

5.3.12.1 Defining Math Functions

In mathematics, a function was originally the idealization of how a varying quantity depends on another quantity. If the function is called f, this relationship is denoted by y = f (x) (read f of x), where x is the function's argument or input, and y is the function's value, output, or image of x by f.

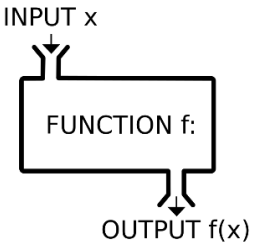


Fig 5.3.12.1 Functions

5.3.12.2 Function Definition

A function is a block of organized, reusable sets of instructions that is used to perform some related actions.

Why do we use functions?

- Reusability of code minimizes redundancy
- Procedural decomposition makes things organized

5.3.12.3 Types of Functions

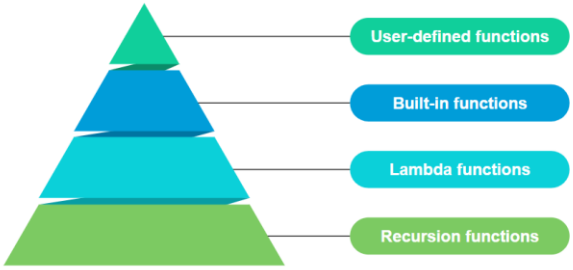


Fig 5.3.12.3.1 Types of Functions

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Fig 5.3.12.3.2 Built-in Functions

5.3.12.4 Defining a Function

Here are some simple rules for defining a function in Python. The keyword def (the start of the function header) is followed by the function name and parentheses (()). Arguments should be placed within these parentheses. You can also define parameters inside these parentheses. A colon (:) to mark the end of the function header. The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as returning None.

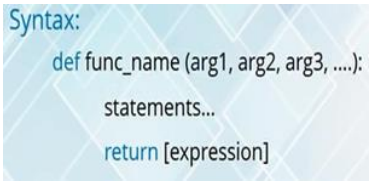


Fig 5.3.12.4 Syntax of Function

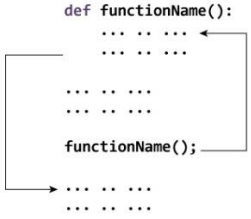


Fig 5.3.12.4 Defining Function

5.3.12.5 Calling a Function

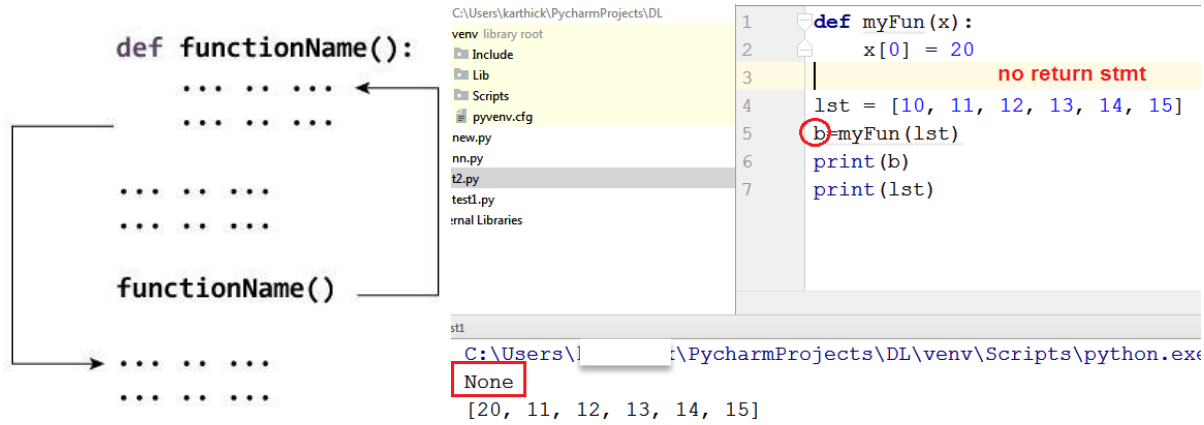


Fig 5.3.12.5 Calling a Function

Note:

1) If we specify return=0, then the output from the function is 0.

2) If there is no return statement in the function define, the output will be None.

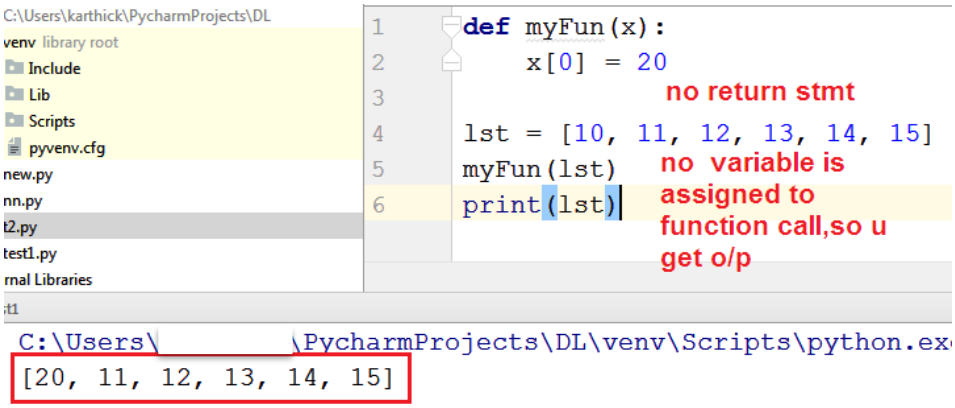


Fig 5.3.12.5 Calling a Function

5.3.12.6 Function Example

Observe how to pass a list as a parameter into the function and do some list methods within the function and see how it returns the values.

```
def myFun(x):
 x[0] = 20
return x
lst = [10, 11, 12, 13, 14, 15]
myFun(lst)
print(lst)
```

Output:

[20, 11, 12, 13, 14, 15]

5.3.13 Pass by Reference

All parameters (arguments) in the Python language are passed by reference. It means that if you change what a parameter refers to within a function, the change also reflects in the calling function. Pass by reference operates on the same logic as pass by address or reference. When a parameter is passed by reference, both the caller and the callee use the same variable for the parameter. If the callee modifies the parameter variable, the effect is visible in the caller's variable. The function can modify the argument, i.e., the value of the variable in the caller's scope can be changed.

Example of Pass by Reference:

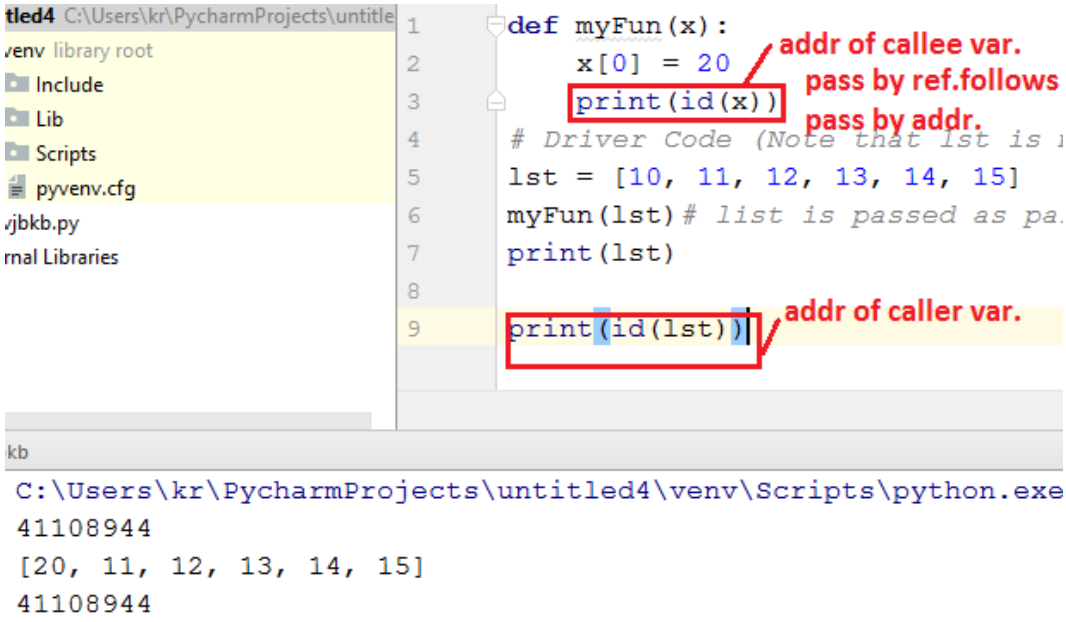
```
def changeme( mylist ):
    print ("Values inside the function before change: ", mylist)
    mylist[2]=50
    print ("Values inside the function after change: ", mylist)
    return
# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

addr. of this var(local var.) should be same for mylist in fn.call(as per defn pass by reference refer to pass by memory address)

Therefore, this would produce the following result

```
('Values inside the function before change: ', [10, 20, 30])
('Values inside the function after change: ', [10, 20, 50])
('Values outside the function: ', [10, 20, 50])
```

Example of Pass by Reference (wrt Address):



5.3.14 Pass by Value

When a parameter is passed by value, the caller and callee have two independent variables with the same value. If the callee modifies the parameter variable, the effect is not visible to the caller**.** In call-by-value, the argument expression is evaluated, and the result of this evaluation is bound to the corresponding variable in the function. So, if the expression is a variable, its value will be assigned (copied) to the corresponding parameter. This ensures that the variable in the caller's scope will be unchanged when the function returns**.**

Example of Pass by Value:

```
def new(a):
   a='2'+a
   print(a)
   print(id(a))
a='hello'
new(a)
print(a)
print(id(a))
```

Output:

```
2hello
35424128
hello
6111264
```

5.3.15 Types of Function Arguments

You can call a function by using the following types of formal arguments:

- Required/Positional arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

5.3.15.1 Required Arguments

Required arguments are the arguments passed to a function in the correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

Example:

```
def add(x,y,z): //remove y,z for correct result
   sum=0
   for i in x:
      sum=sum+i
      print(sum)
lst=[10,20,30]
add(lst)
```

Output:

```
print(sum)
    6 lst=[10,20,30]
----> 7 add(lst)
TypeError: add() missing 2 required positional arguments: 'y' and 'z'
```

5.3.15.2 Keyword Arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name. This allows you to place arguments out of order because the Python interpreter can use the keywords provided to match the values with parameters. Keyword arguments can often be used to make function calls more explicit.

Example:

```
def printinfo( name, age ):
  print ("Name: ", name)
  print ("Age ", age)
  return;
printinfo( age=50, name="miki" )
```

Output:

```
('Name: ', 'miki')
('Age ', 50)
```

5.3.15.3 Default Arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

Example:

```
def printinfo( name, age = 35 ):
  print ("Name: ", name)
  print ("Age ", age)
  return;
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

Output:

```
('Name: ', 'miki')
('Age ', 50)
('Name: ', 'miki')
('Age ', 35)
```

5.3.15.4 Variable-Length Arguments

Why do we need variable-length arguments?

Example:

```
def adder(x,y,z):
   print('sum',x+y+z)
adder(5,10,15,20,25)
```

When we run the above program, the output will be,

```
Traceback (most recent call last):
  File "C:/Users/name/PycharmProjects/jbkkn/nn.py", line 3, in <module>
    adder(5,10,15,20,25)
TypeError: adder() takes exactly 3 arguments (5 given)
```

In the above program, we passed 5 inputs to the adder() function instead of 3 arguments due to which we got TypeError.

In Python, we can pass a variable number of arguments to a function using special symbols. There are two special symbols:

- *args (Non-keyword arguments)
- **kwargs (Keyword arguments)

We use *args and **kwargs as arguments when we are unsure about the number of arguments to pass in the functions.

Python *args:

Python has *args, which allow us to pass a variable number of non-keyword arguments to functions. In the function, we should use an asterisk * before the parameter name to pass variable-length arguments. The arguments are passed as a tuple, and these passed arguments make up a tuple inside the function with the same name as the parameter excluding the asterisk *.

Python **kwargs:

Python passes variable-length non-keyword arguments to functions using *args, but we cannot use this to pass keyword arguments. For this problem, Python has a solution called **kwargs, which allows us to pass the variable-length of keyword arguments to the function. In the function, we use the double asterisk ** before the parameter name to denote this type of argument. The arguments are passed as a dictionary, and these arguments make a dictionary inside the function with the same name as the parameter, excluding double asterisks (**).

You may need to process a function for more arguments than you specified while defining the function.

Syntax:

```
def functionname(*var_name ):
   statements
return [expression]
```

An asterisk (*) is placed before the variable name to pass a variable number of arguments to a function. It is used to pass all non-keyworded, variable-length arguments.

Example:

```
def myFun(*argv):
    for arg in argv:
        print (arg)
myFun('Hello', 'Welcome', 'to', 'Python')
```

Output:

```
Hello
Welcome
to
Python
```

Using *args to pass the variable length non-keyword arguments to the function

```
def printinfo(*var):
   sum=0
   for var in var:
      sum=sum+var
   print('sum:',sum)
# Now you can call printinfo function
printinfo( 10,50 )
printinfo( 70, 60, 50 )
```

Output:

```
('sum:', 60)
('sum:', 180)
```

Using **kwargs to pass the variable keyword arguments to the function

```
def intro (**data):
    for key,value in data.items():
        print('{} is {}'.format(key,value))
intro(Firstname='sita',Lastname='sharma',Age=22,Phone=1234567890)
intro(Firstname='John',Lastname='wood',Email='Johnwood@gmail.com',Country='India')
```

Output:

```
Lastname is sharma
Age is 22
Firstname is sita
Phone is 1234567890
Lastname is wood
Email is Johnwood@gmail.com
Firstname is John
Country is India
```

5.3.16 Anonymous/Lambda Functions

In Python, "anonymous function" means that a function is without a function name. As we already know, the def keyword is used to define normal functions, and the lambda keyword is used to create anonymous functions. These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions. Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions, and they cannot access global variables. They are one-line functions.

Syntax:

X=lambda arg1,arg2,.....argn: expression

Example:

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;
# Now you can call sum as a function
print ("Value of total: ", sum( 10, 20 ))
print ("Value of total: ", sum( 20, 20 ))
```

Output:

```
('Value of total: ', 30)
('Value of total: ', 40)
```

5.3.16.1 Difference between Lambda Vs User-defined Functions in Python

Lambda Functions	User-Defined Functions
lambda as a keyword	def as a keyword
One-line functions	Multi-line functions (i.e., function contain multiple statements)
No function name	Function name
Can’t access global variables	Access local and global variables
Return just one value in the form of an expression	Return values as data

5.3.17 Recursion Functions

In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of constructs are termed "recursive functions." The following image shows the working of a recursive function called recurse.

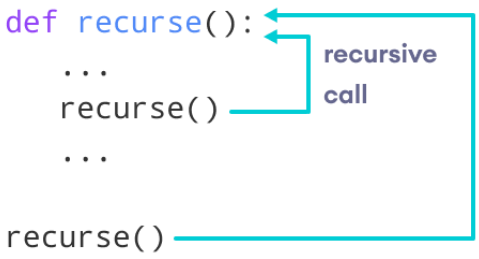


Fig 5.3.17 Recurse

```
def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
num = 3
print("The factorial of", num, "is", factorial(num))
```

5.3.18 Modules

Simply, a module is a file consisting of Python code. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use.

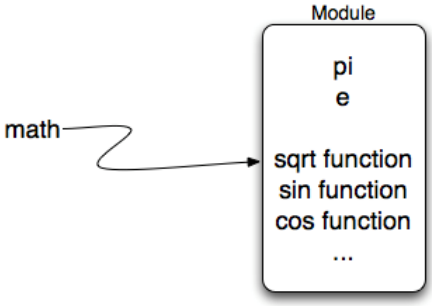


Fig 5.3.18 Modules

5.3.18.1 Built-in Modules

```
help> modules

Please wait a moment while I gather a list of all available modules...

__future__          atexit              html                sched
__main__            audioop             http                scrolledlist
_abc                autocomplete        hyperparser         search
_ast                autocomplete_w      idle                searchbase
_asyncio            autoexpand          idle_test           searchengine
_bisect             base64              idlelib             secrets
_blake2             bdb                 imaplib             select
_bootlocale         binascii            imghdr              selectors
_bz2                binhex              imp                 setuptools
_codecs             bisect              importlib           shelve
_codecs_cn          browser             inspect             shlex
_codecs_hk          builtins            io                  shutil
_codecs_iso2022     bz2                 iomenu              signal
_codecs_jp          cProfile            ipaddress           site
_codecs_kr          calendar            itertools           smtpd
_codecs_tw          calltip             json                smtplib
_collections        calltip_w           keyword             sndhdr
_collections_abc    cgi                 lib2to3             socket
_compat_pickle      cgitb               linecache           socketserver
_compression        chunk               locale              sqlite3
_contextvars        cmath               logging             squeezer
_csv                cmd                 lzma                sre_compile
_ctypes             code                macosx              sre_constants
_ctypes_test        codecontext         macpath             sre_parse
```

Fig 5.3.18.1 Built-in Modules

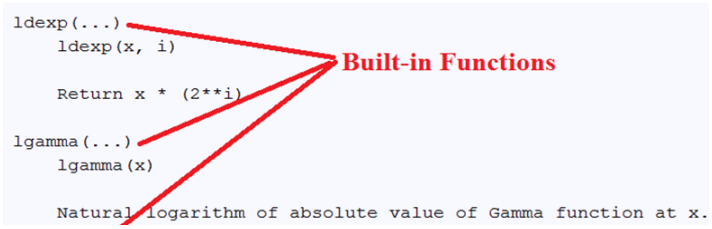
The import Statement:

Syntax:

import module1name

You can use any Python source file as a module by executing an import statement in some other Python source file. When the interpreter encounters an import statement, it imports the module if it is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. If a built-in module is found, its built-in initialization code is executed and finished. If no matching file is found, ImportError is raised. If a file is found, it is called, yielding an executable code block. If a syntax error occurs, SyntaxError is raised.

And the corresponding Built-in functions



import math

help(math)

And the corresponding Built-in functions

asinh(...)

asinh(x)

Return the inverse hyperbolic sine of x.

cos(...)

cos(x)

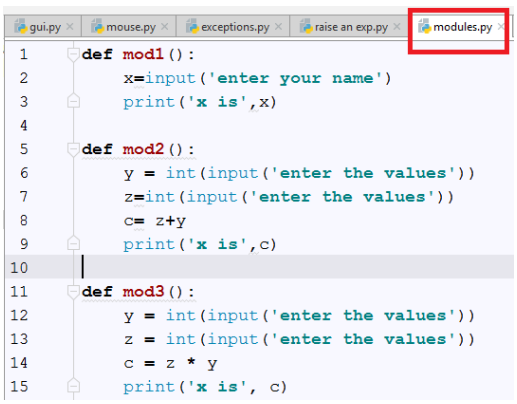
Return the cosine of x (measured in radians).

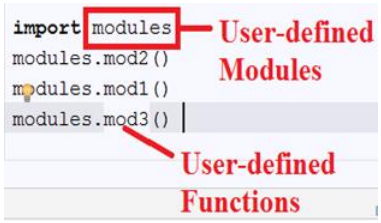
cosh(...)

cosh(x)

Return the hyperbolic cosine of x.

5.3.18.2 User-defined Modules





And the output is,

```
enter the values2
enter the values3
x is 5
enter your namengnb
x is ngnb
enter the values4
enter the values7
x is 28
```

5.3.18.3 The From Import Statement

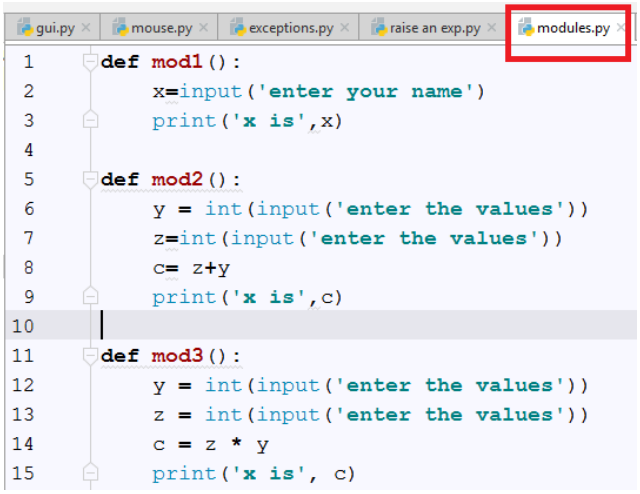
Python's "from" statement lets you import specific attributes from a module into the current namespace. This statement does not import the entire module into the current namespace.

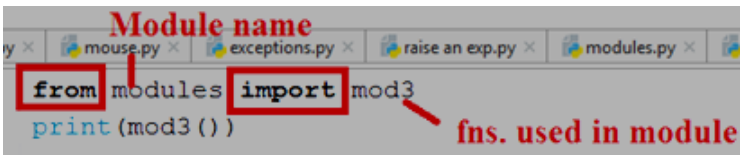
Syntax:

from module_name import name1,name2, ... nameN

Example:

First, create the .py file





And the output is

```
enter the values2
enter the values3
x is 6
None
```

5.3.18.4 The From Import * Statement

It is also possible to import all the names from a module into the current namespace by using the following import statement: All the functions and constants can be imported using *.

Syntax:

from module_name import *

Example:

from math import*

Print(pi)

Print(factorial(6))

And the output is,

```
3.141592653589793
720
```

5.3.18.5 Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences:

- The current directory
- If the module is not found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path.(entire system)

The module search path is stored in the system module sys as the sys.path variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation dependent default.

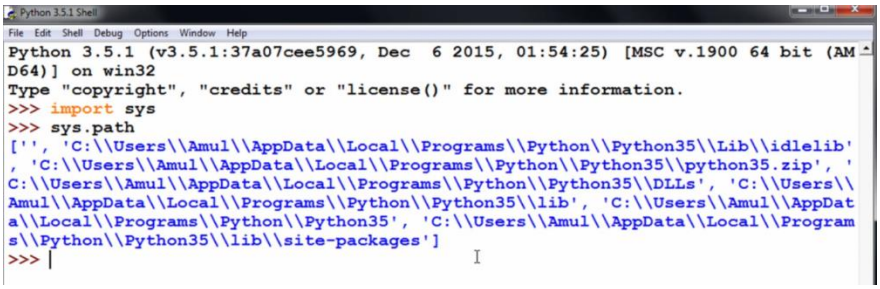
Example:

import mathematics

Print(mathematics.pi)

And the output is

```
Traceback (most recent call last):
  File "C:/Users/          .NIELITCHN/PycharmProject
    import mathematics
ModuleNotFoundError: No module named 'mathematics'
```

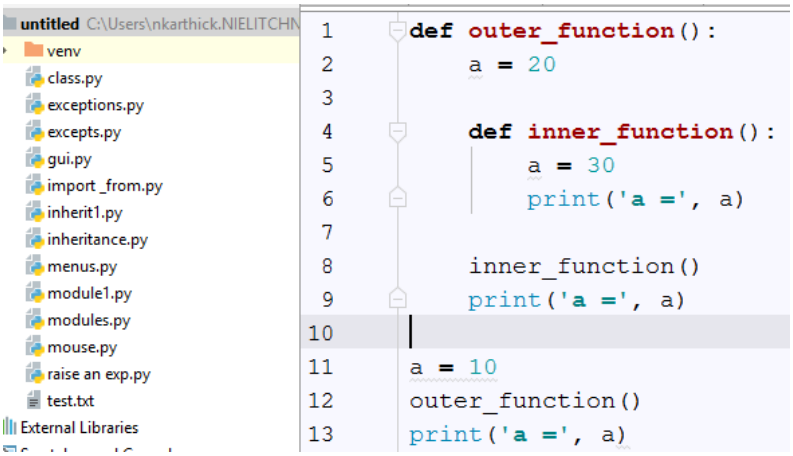


5.3.19 Python Variable Scope

Scope is the portion of the program from where a namespace can be accessed directly. At any given moment, there are at least three nested scopes.

- Scope of the current function, which has local names
- Scope of the module, which has global names
- Outermost scope, which has built-in names (i.e., like keywords).

Example:



And the output is
('a=', 30)
('a=', 20)
('a=', 10)

5.3.20 File I/O

5.3.20.1 Printing to the Screen

```
kjbjbk.py
print ("Python is really a great language,", "isn't it?")
```

This produces the following result on your standard screen:

```
kjbjbk
C:\Users\EPDPT5\PycharmProjects\hnvhfvhf\venv\Scripts\python.exe
Python is really a great language, isn't it?
```

5.3.20.2 Reading Keyboard Input

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are:

- raw_input
- Input

Difference between raw_input() and input()

In Python 2:

input(): Interprets and evaluates the input, which means that if the user enters an integer, an integer will be returned; if the user enters a string, a string is returned.

raw_input(): raw_input() takes exactly what the user typed and passes it back as a string**.** It doesn't interpret the user's input. Even if an integer value of 10 is entered or a list is entered, its type will be string only.

In Python 3:

raw_input() was renamed to input(), so now the input() function returns an exact string.

Old input() was removed.

5.3.20.3 Opening and Closing Files

By default, Python provides the basic functions and methods necessary to manipulate files. You can do most of the file manipulation using a file object.

The open function

Before you can read or write a file, you must open it using Python's built-in **open() function**. This function creates a file object, which would be utilized to call other support methods associated with it.

Syntax:

file object = open(file_name, access_mode,buffering)

Parameters detail:

file_name: The file_name argument is a string value that contains the name of the file that you want to access. We can **create a file with any file format(.txt,.doc,.docx,.xls,.xlsx,.ppt,.pptx)**

access_mode: The access_mode determines the mode in which the file must be opened, i.e., read, write, append, etc. This is an optional parameter, and the default file access mode is read (r).

buffering: If the buffering value is set to 0, no buffering will take place. If the buffering value is 1, line buffering will be performed while accessing a file. If you specify the buffering value as an integer greater than 1, then the buffering action will be performed with the indicated buffer size. If negative, the buffer size is the system default (default behavior).

A buffer stores a chunk of data from the operating system's file stream until it is consumed, at which point more data is brought into the buffer. The reason it is good practice to use buffers is that interacting with the raw stream might have high latency, i.e., considerable time is taken to fetch data from it and to write to it. Let's take an example. Let's say you want to read 100 characters from a file every 2 minutes over a network. Instead of trying to read from the raw file stream every 2 minutes, it is better to load a portion of the file into a buffer in memory and then consume it when the time is right. Then, the next portion of the file will be loaded into the buffer, and so on. Note that the size of the buffer will depend on the rate at which the data is consumed. For the example above, 100 characters are required after 2 minutes. So, anything less than 100 will result in an increase in latency; 100 itself will do just fine; and anything more than a hundred will be great.

A list of the different modes of opening a file:

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer will be at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Closing Files:

The close() method:

The close() method of a file object closes the file object, after which no more writing can be done. It is good practice to use the close() method to close a file.

Syntax:

fileObject.close()

5.3.20.4 Reading and Writing Files

The write() method:

The write() method writes any string to an open file. It is important to note that Python strings can have binary data and not just text. The write() method does not add a newline character ('\n') to the end of the string.

Syntax:

fileObject.write(string)

Here, passed parameter is the content to be written into the opened file.

The read() method:

The read() method reads a string from an open file. It is important to note that Python strings can have binary data and not just text.

Syntax:

fileObject.read([count])

Here, the passed parameter "count" is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file, and if the count is missing, then it tries to read as much as possible, which may be until the end of the file.

5.3.20.5 File Positions

The **tell() method** tells you the current position within the file.

The **seek(offset,from) method** changes the current file position. The offset argument indicates the number of bytes to be moved. The *from* argument specifies the reference position from which the bytes are to be moved.

5.3.20.6 Renaming and Deleting Files

Python's "os" module provides methods that help you perform **file-processing operations**, such as renaming and deleting files. To use this module, you need to import it first, and then you can call any related functions.

The rename() method:

The rename() method takes two arguments: the current filename and the new filename.

Syntax:

os.rename("current_file_name"," new_file_name")

The delete() method:

You can use the delete() method to delete files by supplying the name of the file to be deleted as the argument.

Syntax:

os.remove("file_name")

Example:

```
import os
# Delete file test2.txt
os.remove("text2.txt")
```

5.3.21 Splunk

Splunk is a powerful tool that can help support software testing in a variety of ways, from log analysis to performance monitoring to test automation. By leveraging Splunk's capabilities, testers can improve the effectiveness and efficiency of their testing efforts and ultimately deliver higher-quality software to end-users.

Splunk is a software platform for searching, analysing, and visualising machine-generated data that has been collected from the websites, applications, sensors, devices, etc. that make up your IT infrastructure and company.

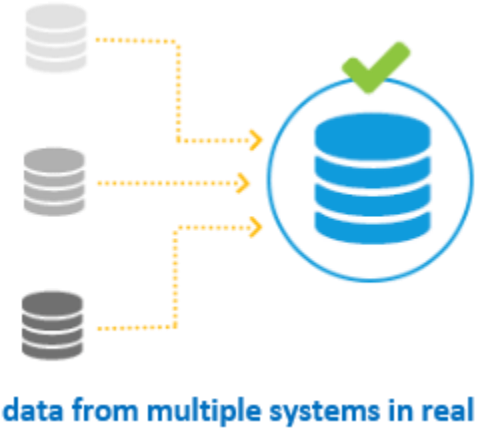


Fig 5.3.21.1 Splunk

Benefits of Splunk:

You can enter your data in any format, such as .csv, .json, or another format.

You may set up Splunk to notify users of alerts and events as soon as a machine's status changes.

You can foresee the resources required to scale up the infrastructure with accuracy.

Operational intelligence knowledge objects can be produced.

Functions of Splunk:

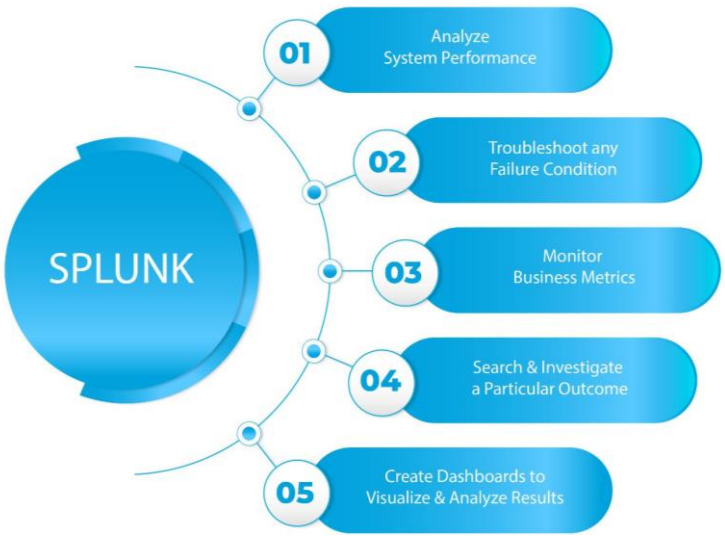


Fig 5.3.21.2 Functions of Splunk

Log analysis: Splunk can be used to collect, index, and analyze log data generated during testing. This can help testers identify patterns, errors, and issues that may not be visible through manual testing.

Performance testing: Splunk can be used to monitor and analyze system performance data during load testing, stress testing, and other types of performance testing. This can help identify performance bottlenecks and optimize system performance.

Test automation: Splunk can be integrated with testing frameworks and tools to automate test execution and generate reports. This can help streamline the testing process and improve efficiency.

Monitoring: Splunk can be used to monitor the system and applications during testing to identify issues and errors in real-time. This can help testers quickly identify and resolve issues during testing.

Reporting: Splunk can be used to generate reports on testing metrics, such as test execution time, test coverage, and test results. This can help testers and other stakeholders track progress and identify areas for improvement.

An example for how Splunk is used for health monitoring in a XXX company:

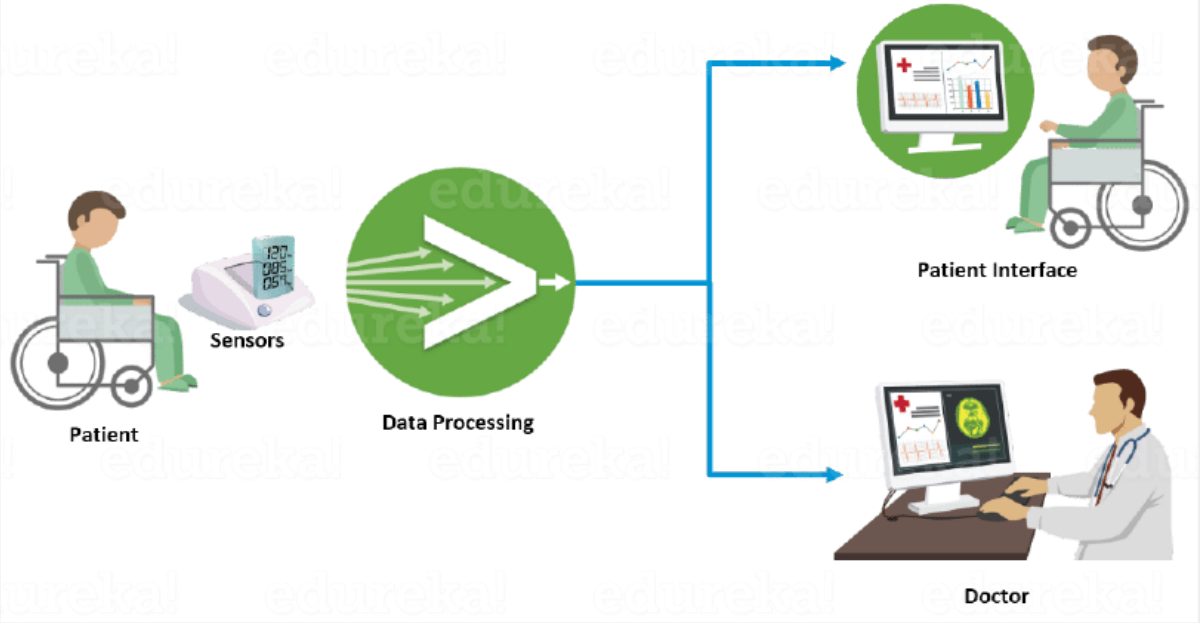


Fig 5.3.21.3 Splunk in health monitoring

Here, we have discussed how XXX company used Splunk for data analytics. Using IoT devices, they gathered healthcare information from patients who were stationed far away (sensors). Using the patient interface, Splunk would process this data and notify the doctor and the patient of any unusual activities. Thanks to Splunk, they were able to do the following:

- Real-time reporting of health conditions
- Analyzing patterns in the patient's medical history by digging further
- When the patient's health deteriorates, alarms or alerts are sent to both the doctor and the patient

5.3.22 HiL Testing

What is HiL Testing?

Hardware-in-the-loop (HIL) testing is a type of testing method used in the development and validation of complex control systems. It involves testing the Electronic Control Unit (ECU) or the controller of a system by interfacing it with a simulated or real hardware component, which could be a sensor, actuator or any other physical device.

The HIL testing process involves running real-time simulations on a computer to create an environment in which the ECU can interact with the hardware component as if it were in a real-world scenario. The signals sent and received by the ECU are monitored and analyzed, and the results are used to verify the system's behavior and functionality.

This approach is useful because it allows engineers to test and validate the performance of the system in a controlled environment before it is deployed in the real world. It also enables them to identify and correct any issues or bugs in the system, reducing the risk of failures and improving the overall quality of the system.

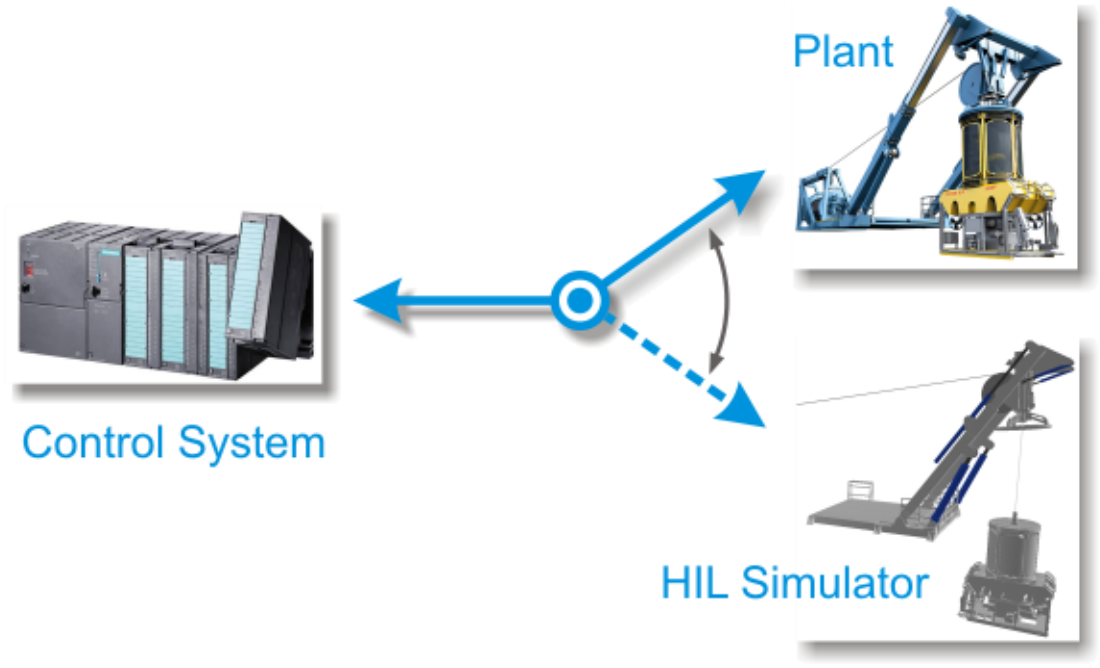


Fig 5.3.22 HiL Testing

Example for HiL Testing

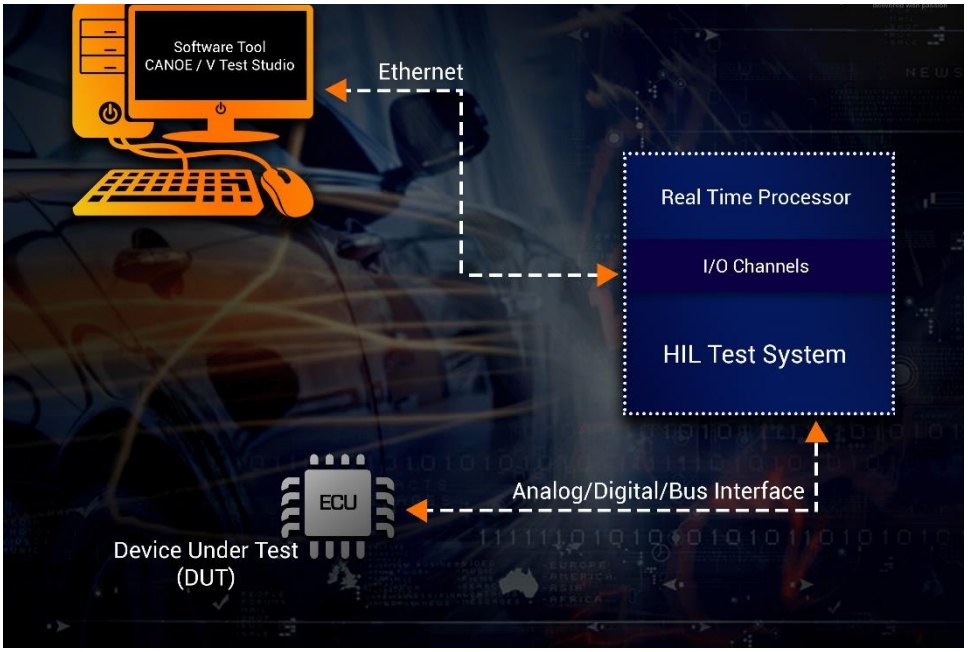
Suppose you are developing a new Electronic Stability Control (ESC) system for a car. The ESC system relies on data from various sensors, such as the wheel speed sensors and the steering angle sensor, to detect and correct any loss of control of the vehicle.

To test the ESC system using HIL, you would connect the ECU of the ESC system to a simulated vehicle model running on a computer. The simulated vehicle model would include virtual sensors that generate signals similar to the real sensors installed in a car. For example, the virtual wheel speed sensors would simulate the rotation of the wheels, and the virtual steering angle sensor would simulate the movement of the steering wheel.

You would then run various test scenarios, such as sudden turns or emergency braking, to verify that the ESC system responds correctly to different driving conditions. The simulated signals generated by the virtual sensors would be fed into the ECU, and the output signals from the ECU would be used to control the simulated brake actuators and throttle actuators. The system's behavior and performance would be analyzed and evaluated based on the results obtained from these tests.

Through this HIL testing process, any issues or bugs in the ESC system can be identified and corrected before it is installed in a real car, improving the overall safety and performance of the vehicle

HiL Testing Architecture



HiL Test System

A HIL test system is at the heart of HIL testing. The simulation part of the testing that we mentioned earlier, needs an environment that can trick the ECU into believing that it is connected to a real vehicle.

The test system provides this environment. And for that purpose, it requires a gamut of software and hardware components. Let's discuss them!

Real-time Processing Unit: The processing unit is at the core of a HIL Test System. It performs the most intensive task of executing the components of the test system. Complex automotive controllers have a large number of I/O channels and bus communication to be handled. From logging of the data and I/O communication to test stimulus generation, it is the processing unit that keeps the tasks going.Why a real-time processor, you might ask? Well, the test system replaces actual ECUs and their I/O signals. For the test to be executed perfectly, an accurate simulation of the vehicle electronics has to be ensured. And hence, a real-time processing unit.

I/O Interfaces: The ECU under test needs to be connected to the HIL Test system for the signals to interact. The I/O interfaces are responsible for establishing this connection. These interfaces can be digital or analog signals.Using the I/O interfaces the test engineers can:

- Generate stimulus signals
- Manage the communication of the sensors and the actuators between DUT and Test System
- Manage current/voltage and optical I/O channels

The importance of I/O interfaces also lies in the fact that it has to make sure that the DUT must always behave in a way that it is controlling the actual hardware. Test System providers may provide both dedicated as well as configurable I/O interfaces.

Software Interface: Software components are essential to the HIL Test System as these are required for writing test cases, creating required stimulations and generating test reports.Typical software components that aid the HIL Test system are:

Test Case Scripting: The tool is required to create test cases using scripting language. Various scripting languages are used depending on the HIL Test system brand. For example, Vector VT System uses VTest Studio for writing test cases in CAPL script. On the other hand, DSpace uses Python for the same.

ECU simulation: A software tool for simulating other ECUs in a vehicle is of the prime importance in an automotive HIL testing setup. The ECU under test might have to interact with other ECUs during its operation. And therefore, during the HIL testing all such ECUs need to be simulated.

Device Under Test (DUT): Device under test (DUT) is the control unit, the functionality of which is being validated. It can be a powertrain ECU, a Battery Management System ECU or any other automotive control unit. For testing purpose, DUT is connected to the HIL Test System.

Popular HIL Test Systems that are deployed in Hardware-in-Loop testing

Vector VT system: VT System is a popular modular test system that is designed to access the ECU's inputs and outputs for validation. The system works together with tools like CANoe and VTest Studio. Based on CAN Bus protocol, VT System also supports CAN, LIN and other communication protocols.

National Instrument Lab View: Lab View from NI, is a HIL test framework that helps the engineers create flexible test applications using a graphical programming approach. A test system designed to integrate a vast range of hardware, LabVIEW serves diverse industries to develop smart machines and industrial equipment.

DSpace System: The Test System comprises the complete tool-chain (software and hardware) required for a simulated test environment. The biggest advantage of using DSpace for automotive software development is that it is designed to support the tests methods and coverages mandated by ISO 26262 standard.

HIL Testing in Action: Testing a Motor Control System

Motor Controller ECU has some highly safety-critical tasks at hand viz. power steering, EV powertrain and so on. State-of-the art HIL Testing, thus becomes mandatory.

Generally, the motor controller ECUs need to be tested for

- PWM measurements
- Checking whether Encoders and resolvers are working fine at higher clock rate
- Evaluating the functioning of the Position sensor

A HIL Test system simulates the power electronics, electric motor and the vehicle environment at the signal, electrical and mechanical levels.

And for this, the HIL Test system needs access to

- ECU connections in order to control them
- Power supply
- Communication Bus (CAN)
- I/O channels

The HIL Test System does the following tasks:

- Simulates other ECUs with the help of CANoe or equivalent tool (depending on the type of HIL Test system deployed)
- Generates the required sensor input such as PWM signals, required voltage
- Simulates the actuator (electric motor) with electronic load and measures the output from the motor controller ECU such as PWM duty cycle
- Measures the voltage and current supplied

The Test Cases are written in software tools such as VTest Studio using scripts (CAPL, Python). The test reports are generated after the required tests are executed.

References

https://www.embitel.com/blog/embedded-blog/how-to-perform-hardware-in-loop-testing-for-an-automotive

Summary

In this unit, we learned about the following concepts:

1. The basics of Python
2. Various data types, variables, basic input-output operations, and basic operators
3. The Boolean values and the various logical operations used in the Python
4. Various functions in Python and file handling operations in Python
5. Uses of Splunk in implementing automation in testing
6. What is HiL Testing?

Summary

In this module "Test Infrastructure", we learned about the following concepts:

1. Various embedded software test environments like simulation, prototype, pre-production, and post-production
2. Various test tools available in the market and their categorization (Tessy, Parasoft, Egg Plant, KlocWork, and Vector software)
3. Importance of Python in implementation test automation for embedded products
4. Various datatypes used in Python
5. Various logical operators in Python
6. Various file handling mechanisms using Python
7. How does Splunk help with test automation?

Summary

In this course "Verification and validation", we learned about the following concepts:

1. Importance of testing
2. Tester mindset
3. Developers mindset
4. Different levels of testing and their significance in the SDLC
5. Various levels of testing
6. Various types of testing
7. Efficient testing with test design techniques
8. How does automation enable testers?