

**WELCOME
TO
EIGHTH LECTURE
OF**

Software Testing Techniques and Strategies

COURSE CODE :SE-481

LEVELS OF TESTING

2. What is Integration Testing?

- ✓ Once all the individual units are created and tested, we start combining those “Unit Tested” modules and start doing the integrated testing. So the meaning of
- ✓ Integration testing is quite straightforward- Integrate/combine the unit tested module one by one and test the behaviour as a combined unit
- ✓ A typical software project consists of multiple software modules, coded by different programmers.
- ✓ Integration Testing focuses on checking data communication amongst these modules.
- ✓ Hence it is also termed as 'I & T' (Integration and Testing), 'String Testing' and sometimes 'Thread Testing'.

Definition by ISTQB

- ✓ **Integration testing:** Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems.
- ✓ **Component integration testing:** Testing performed to expose defects in the interfaces and interaction between integrated components.
- ✓ **System integration testing:** Testing the integration of systems and packages; testing interfaces to external organizations (e.g. Electronic Data Interchange, Internet).

LEVELS OF TESTING

2.Integration Testing?

Why do Integration Testing?

Although each software module is unit tested, defects still exist for various reasons like

- A Module in general is designed by an individual software developer whose understanding and programming logic may differ from other programmers.

integration Testing becomes necessary to verify the software modules work in unity

- At the time of module development, there are wide chances of change in requirements by the clients. These new requirements may not be unit tested and hence system integration Testing becomes necessary.
- Interfaces of the software modules with the database could be erroneous
- External Hardware interfaces, if any, could be erroneous
- Inadequate exception handling

LEVELS OF TESTING

2.Integration Testing?

When To Start Integration Testing:

After Unit Testing

Method

- Any of Black Box Testing, White Box Testing, and Gray Box Testing methods can be used.

Goal Of Integration Testing:

- The main function or goal of Integration testing is to
- To test the interfaces between the units/modules.
- To check the combinational behavior, and validate whether the requirements are implemented correctly or not. priority is to be given for the integrating links rather than the unit functions which are already tested.
- Here we should understand that Integration testing does not happen at the end of the cycle, rather it is conducted simultaneously with the development. So in most of the times, all the modules are not actually available to test and here is what the challenge comes to test something which does not exist!

LEVELS OF TESTING

Integration Test Case:

Integration Test Case differs from other test cases in the sense it focuses mainly on the interfaces & flow of data/information between the modules. Here priority is to be given for the integrating links rather than the unit functions which are already tested.

Sample Integration Test Cases for the following scenario: Application has 3 modules say

- ✓ 'Login Page',
 - ✓ 'Mail box' and
 - ✓ 'Delete mails' and each of them are integrated logically.
-
- Here do not concentrate much on the Login Page testing as it's already been done in Unit Testing.
 - But check how it's linked to the Mail Box Page.
 - Similarly Mail Box: Check its integration to the Delete Mails Module.

Test Case ID	Test Case Objective	Test Case Description	Expected Result
1	Check the interface link between the Login and Mailbox module	Enter login credentials and click on the Login button	To be directed to the Mail Box
2	Check the interface link between the Mailbox and Delete Mails Module	From Mail box select the an email and click delete button	Selected email should appear in the Deleted/Trash folder

LEVELS OF TESTING

2. Integration Testing?

Approaches/Methodologies/Strategies of Integration Testing:

The Software Industry uses variety of strategies to execute Integration testing

I. Big Bang Approach :

II. Incremental Approach: which is further divided into following

- a. Top Down Approach
- b. Bottom Up Approach
- c. Sandwich Approach –

Combination of Top Down and Bottom Up

Below are the different strategies, the way they are executed and their limitations as well advantages.

LEVELS OF TESTING

2.Integration Testing?

I. Big Bang Approach:

Here all component are integrated together at once, and then tested.

Advantages:

- Convenient for small systems.

Disadvantages:

- Fault Localization is difficult.
- Given the sheer number of interfaces that need to be tested in this approach, some interfaces links to be tested could be missed easily.
- Since the integration testing can commence only after "all" the modules are designed, testing team will have less time for execution in the testing phase.

Since all modules are tested at once, high risk critical modules are not isolated and tested on priority. Peripheral modules which deal with user interfaces are also not isolated and tested on priority

LEVELS OF TESTING

2. Integration Testing?

II. Incremental Approach

:

In this approach, testing is done by joining two or more modules that are logically related. Then the other related modules are added and tested for the proper functioning. Process continues until all of the modules are joined and tested successfully.

This process is carried out by using dummy programs called Stubs and Drivers. Stubs and Drivers do not implement the entire programming logic of the software module but just simulate data communication with the calling module.

Stub: Is called by the Module under Test.

Driver: Calls the Module to be tested.

Incremental Approach in turn is carried out by two different Methods:

- Bottom Up
- Top Down

LEVELS OF TESTING

2. Integration Testing?

II. Incremental Approach

:

In this approach, testing is done by joining two or more modules that are logically related. Then the other related modules are added and tested for the proper functioning. Process continues until all of the modules are joined and tested successfully.

This process is carried out by using dummy programs called Stubs and Drivers. Stubs and Drivers do not implement the entire programming logic of the software module but just simulate data communication with the calling module.

Stub: Is called by the Module under Test.

Driver: Calls the Module to be tested.

Incremental Approach in turn is carried out by two different Methods:

- Bottom Up
- Top Down

LEVELS OF TESTING

Integration Testing?

II. Incremental Approach

a. Bottom up Integration

In the bottom up strategy, each module at lower levels is tested with higher modules until all modules are tested. It takes help of Drivers for testing Diagrammatic Representation:



Advantages:

Fault localization is easier.

No time is wasted waiting for all modules to be developed unlike Bigbang Approach

Disadvantages:

- Critical modules (at the top level of software architecture) which control the flow of application are tested last and may be prone to defects.
- Early prototype is not possible

LEVELS OF TESTING

Integration Testing?

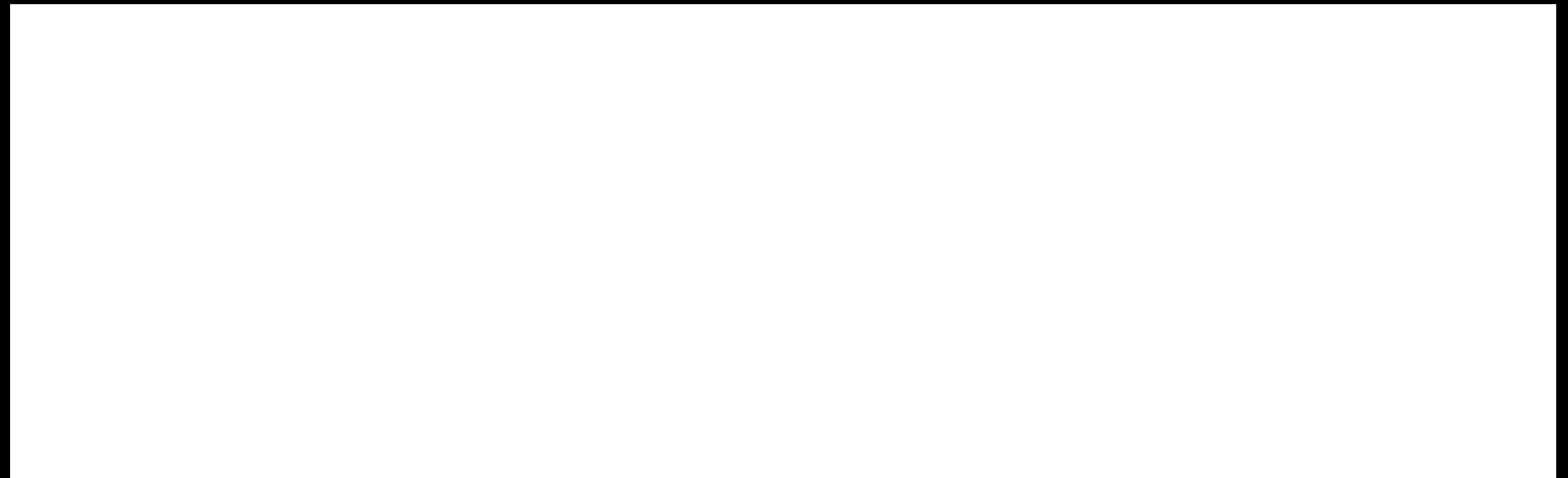
II. Incremental Approach

B . Top down Integration:

In Top to down approach, testing takes place from top to down following the control flow of the software system.

Takes help of stubs for testing.

Diagrammatic Representation:



Advantages:

- Fault Localization is easier.
- Possibility to obtain an early prototype.
- Critical Modules are tested on priority; major design flaws could be found and fixed first.

Disadvantages:

- Needs many Stubs.
- Modules at lower level are tested inadequately.

LEVELS OF TESTING

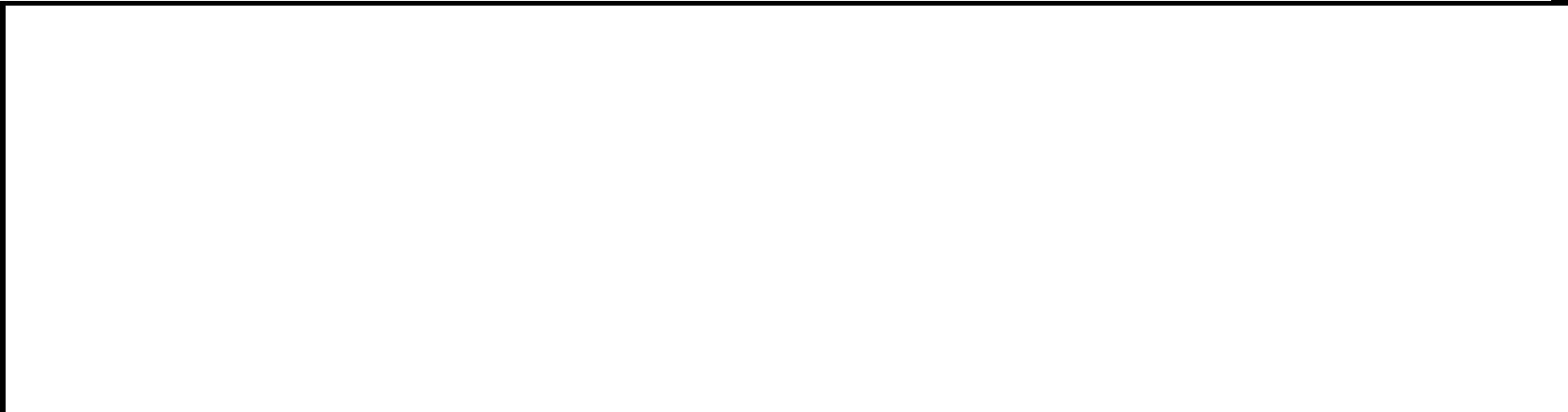
Integration Testing

II. Incremental Approach

c. Sandwich/Hybrid

Sandwich Testing is the combination of bottom-up approach and top-down approach, so it uses the advantage of both bottom up approach and top down approach .

Its is also known as Mixed Integration testing or Hybrid Testing



LEVELS OF TESTING

2. Integration Testing

C. Sandwich/Hybrid

Sandwich Testing is the combination of bottom-up approach and top-down approach, so it uses the advantage of both bottom up approach and top down approach

Sandwich testing is basically viewed as 3 layers:

2. A target layer is the middle.
 3. A layer above the target called as the top layer.
 4. A layer below the target called as the bottom layer.
- ✓ In sandwich testing, testing is mainly focused for main target layer.
 - ✓ This testing is selected on the basis of system characteristics and structure's code.
 - ✓ It tries to minimize the number of hubs and drivers when there are more than 3 Layers
 - ✓ Bottom-up testing starts from middle layer and goes upward to the top layer where as
 - ✓ Top-down testing starts from middle layer and goes downward.
 - ✓ Here, top modules are tested with lower modules at the same time lower modules are integrated with top modules and tested. This strategy makes use of stubs as well as drivers.
 - ✓ With the successful test coverage of the entire software application from both the directions, only the middle one or the target layer is left for the performance of the final set of tests.
 - ✓
 - ✓ Initially it uses the stubs and drivers where stubs simulate the behavior of missing component. It is also known as the Hybrid **Integration Testing**.
 - ✓ Sandwich Testing also can not be used for those systems which have a lot of interdependence between different modules and it allows parallel testing.

LEVELS OF TESTING

2. Integration Testing

II. Incremental Approach

C. Sandwich/Hybrid

Advantage of Sandwich Testing:

- Sandwich approach is useful for very large projects having several subprojects. When development follows a spiral model and the module itself is as large as a system, then one can use sandwich testing.
- It allows parallel testing.
- Sandwich testing is time saving approach.
- sandwich testing performs more coverage with same stubs.
- Both Top-down and Bottom-up approach starts at a time as per development schedule. Units are tested and brought together to make a system .

Disadvantage of Sandwich Testing:

- It needs more resources and big teams perform both bottom-up and top-down methods of testing at a time or one after the other.
- Sandwich Testing is very costly.
- Sandwich Testing can not be used for such systems which have a lot of interdependence between different modules.
- Sometimes In sandwich testing the need of stubs and drivers is very high.

LEVELS OF TESTING

2.Integration Testing

Entry and Exit Criteria.

Entry and Exit Criteria to Integration testing phase in any software development model

Entry Criteria:

- Unit Tested Components/Modules
- All High prioritized bugs fixed and closed during unit testing .
- Required Test Environment to be set up for Integration testing

Exit Criteria:

- Successful Testing of Integrated Application.
- All Modules coded completely and integrated successfully
- All High prioritized integration bugs fixed and closed
- Integration tests Plan, test case, scenarios to be signed off and documented
- Technical documents to be submitted followed by release Notes.

LEVELS OF TESTING

2.Integration Testing?

Best Practices/ Guidelines for Integration Testing

- First, determine the Integration Test Strategy that could be adopted and later prepare the test cases and test data accordingly.
- Study the Architecture design of the Application and identify the Critical Modules. These need to be tested on priority.
- Obtain the interface designs from the Architectural team and create test cases to verify all of the interfaces in detail. Interface to database/external hardware/software application must be tested in detail.
- After the test cases, it's the test data which plays the critical role.
- Always have the mock data prepared, prior to executing. Do not select test data while executing the test cases.

LEVELS OF TESTING

3. What is System Testing?

- ✓ System testing is the type of testing to check the behaviour of a complete and fully integrated software product based on the software requirements specification (SRS) document.
- ✓ The main focus of this testing is to evaluate Business / Functional / End-user requirements.
- ✓ This is black box type of testing where external working of the software is evaluated with the help of requirement documents & it is totally based on Users point of view.
- ✓ For this type of testing do not required the knowledge of internal design or structure or code.
- ✓ This testing is to be carried out only after System Integration Testing is completed where both Functional & Non-Functional requirements are verified.
- ✓ In the integration testing testers are concentrated on finding bugs/defects on integrated modules.
- ✓ But in the Software System Testing testers are concentrated on finding bugs/defects based on software application behavior, software design and expectation of end user.

LEVELS OF TESTING

3. System Testing

- ✓ System Testing Is the Most critical and important phase of Software Testing
- ✓ Testing the fully integrated applications including external peripherals in order to check how components interact with one another and with the system as a whole.
- ✓ This is also called End to End testing scenario..
- ✓ Verify through testing of every input in the application to check for desired outputs.
- ✓ Testing of the user's experience with the application.
- ✓ In Software Development Life Cycle the System Testing is perform as the first level of testing where the System is tested as a whole.
- ✓ In this step of testing check if system meets functional requirement or not.
- ✓ System Testing enables you to test, validate and verify both the Application Architecture and Business requirements.
- ✓ Application/System is tested in an environment that particularly resembles the effective production environment where the application/software will be lastly deployed.
- ✓ Generally, a separate and dedicated team is responsible for system testing. And,
- ✓ System Testing is performed on staging server which is similar to production server. So this means you are testing software application as good as production environment

*A **staging server** is a type of **server** that is used to test a software, website or service in a **production**-similar environment before being set live. It is part of staging environment or **staging site**, where it serves as a temporary hosting and testing **server** for any new software or websites.*

LEVELS OF TESTING

How to Start System Testing?

In Software System Testing following steps needs to be executed

Entry Criteria for System Testing:

- ✓ • Unit testing should be finished.
- ✓ • Integration of modules should be fully integrated.
- ✓ • As per the specification document software development is completed.
- ✓ • Testing environment is available for testing (similar to Staging environment)

LEVELS OF TESTING

3. System Testing

Step 1) First & important step is System Test Plan:

All points to be cover in System Test plan may vary from organization to organization as well as based on project plan, test strategy & main test plan.

Nevertheless, here is list of standard point to be considered while creating System Test Plan:

- Goals & Objective
- Scope
- Critical areas Area to focus
- Test Deliverable
- Testing Strategy for System testing
- Testing Schedule
- Entry and exit criteria
- Suspension & resumption criteria for system testing
- Test Environment
- Roles and Responsibilities

LEVELS OF TESTING

3. System Testing

Step 2) Second step is Test Cases:

Here you should consider different type of testing like Functional testing, Regression testing, Smoke testing, Sanity testing, Ad-hoc testing, Exploratory testing, Usability testing, GUI software testing, Compatibility testing, Performance testing, Load testing, Stress testing, Volume testing, Error handling testing, Scalability testing, Security testing, Capacity testing, Installation testing, Recovery testing, Reliability testing, Accessibility testing etc

While writing test case you need to check that test cases are covering all functional, non-functional, technical & UI requirements



Step 3) Creation of test data which used for System testing.

Step 4) Automated test case execution.

Step 5) Execution of normal test case & update test case if using any test management tool (if any).

Step 6) Bug Reporting, Bug verification & Regression testing.

Step 7) Repeat testing life cycle (if required).

LEVELS OF TESTING

3. System Testing

Roles & Responsibilities

- ✓ •Who the tester works for - This is a major factor in determining the types of system testing a tester will use. Methods used by large companies are different than that used by medium and small companies.
- ✓ •Time available for testing - Ultimately, all 50 testing types could be used. Time is often what limits us to using only the types that are most relevant for the software project.
- ✓ •Resources available to the tester - Of course some testers will not have necessary resources to conduct a testing type. For example if you are a tester working for a large software development firm, you are likely to have expensive automated testing software not available to others.
- ✓ •Software Tester's Education - There is a certain learning curve for each type of software testing available. To use some of the software involved, a tester has to learn how to use it.
- ✓ •Testing Budget - Money becomes a factor not just for smaller companies and individual software developers but large companies as well

LEVELS OF TESTING

Different Types of System Testing

There are more than 50 types of System Testing. Below listed the types of system testing a large software development company would typically use

1. **Usability Testing** - Usability Testing mainly focuses on the user's ease to use the application, flexibility in handling controls and ability of the system to meet its objectives
2. **Load Testing** - Load Testing is necessary to know that a software solution will perform under real-life loads.
3. **Regression Testing** - Regression Testing involves testing done to make sure none of the changes made over the course of the development process have caused new bugs. It also makes sure no old bugs appear from the addition of new software modules over time.
4. **Recovery Testing** - Recovery testing is done to demonstrate a software solution is reliable, trustworthy and can successfully recoup from possible crashes.
5. **Migration Testing** - Migration testing is done to ensure that the software can be moved from older system infrastructures to current system infrastructures without any issues.
6. **Functional Testing** - Also known as functional completeness testing, Testing involves trying to think of any possible missing functions. Testers might make a list of additional functionalities that a product could have to improve it during functional testing.
7. **Hardware/Software Testing** - IBM refers to Hardware/Software testing as "HW/SW Testing". This is when the tester focuses his/her attention on the interactions between the hardware and software during system testing.

LEVELS OF TESTING

4. Acceptance Testing OR User Acceptance Testing

- ✓ After the system test has corrected all or most defects, the system will be delivered to the user or customer for Acceptance Testing or User Acceptance Testing (UAT)
- ✓ It is a level of software testing where a system is tested for acceptability.
- ✓ The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

Definition by ISTQB :

- **acceptance testing**: Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.

There are various forms of acceptance testing:

- User acceptance Testing
- Business acceptance Testing
- Contract Acceptance Testing
- Regulations/Compliance Acceptance Testing (RAT)
- Alpha Testing
- Beta Testing

LEVELS OF TESTING

4. Acceptance Testing OR User Acceptance Testing

When is it performed?

- ✓ Acceptance Testing is the fourth and last level of software testing performed after System Testing and
- ✓ before making the system available for actual use.

Who performs it?

- ✓ Acceptance testing is basically done by the
- ✓ User or Customers' customers
- ✓ stakeholders
- ✓ tester (rarely), management,
- ✓ Sales, Support teams performs acceptance testing depending on the type of test carried out

Strategy /Approach Acceptance Testing OR User Acceptance Testing

- I. Internal Acceptance Testing
- II. External Acceptance Testing

LEVELS OF TESTING

4. Strategy /Approach Acceptance Testing OR User Acceptance Testing

- I. Internal Acceptance Testing** (Also known as Alpha Testing) is performed by members of the organization that developed the software but who are not directly involved in the project (Development or Testing). Usually, it is the members of Product Management, Sales and/or Customer Support.
- II. External Acceptance Testing** is performed by people who are not employees of the organization that developed the software, consist of two types
 - a. Customer Acceptance Testing** is performed by the customers of the organization. They are the ones who asked the organization to develop the software. [This is in the case of the software not being owned by the organization that developed it.] .
 - b. User Acceptance Testing** (Also known as Beta Testing) is performed by the end users of the software. They can be the customers themselves or the customers' customers.

LEVELS OF TESTING

4. Strategy /Approach Acceptance Testing OR User Acceptance Testing

- I. Internal Acceptance Testing** (Also known as Alpha Testing) is performed by members of the organization that developed the software but who are not directly involved in the project (Development or Testing). Usually, it is the members of Product Management, Sales and/or Customer Support.
- II. External Acceptance Testing** is performed by people who are not employees of the organization that developed the software, consist of two types
 - a. Customer Acceptance Testing** is performed by the customers of the organization. They are the ones who asked the organization to develop the software. [This is in the case of the software not being owned by the organization that developed it.] .
 - b. User Acceptance Testing** (Also known as Beta Testing) is performed by the end users of the software. They can be the customers themselves or the customers' customers.

LEVELS OF TESTING

4. Acceptance Testing OR User Acceptance Testing

Alpha & Beta Testing

Alpha Testing normally takes place in the development environment and is usually done by internal staff. Long before the product is even released to external testers or customers. Also potential user groups might conduct Alpha Tests, but the important thing here is that it takes place in the development environment.

- Based on the feedback – collected from the alpha testers – development teams then fix certain issues and improve the usability of the product.

Beta Testing, also known as “field testing”, takes place in the customer’s environment and involves some extensive testing by a group of customers who use the system in their environment. These beta testers then provide feedback, which in turn leads to improvements of the product.

- Alpha and Beta Testing are done before the software is released to all customers

LEVELS OF TESTING

4. Acceptance Testing OR User Acceptance Testing

Analogy

During the process of manufacturing a ballpoint pen,

- the cap,
- the body,
- the tail and
- clip,
- the ink cartridge

- ✓ ALL the units of ballpoint are produced separately, and unit tested separately.
- ✓ When two or more units are ready, they are assembled and Integration Testing is performed.
- ✓ When the complete pen is integrated, System Testing is performed.
- ✓ Once System Testing is complete, Acceptance Testing is performed so as to confirm that the ballpoint pen is ready to be made available to the end-users.
- ✓ Method Usually, Black Box Testing method is used in Acceptance Testing.
- ✓ Testing does not normally follow a strict procedure and is not scripted but is rather ad-hoc.

Types of Acceptance Testing OR User Acceptance Testing

1) User Acceptance Testing (UAT) i

Specific requirements which are quite often used by the end-users are primarily picked for the testing purpose. This is also termed as End-User Testing.

The term “User” here signifies the end-users to whom the Product/application is intended and hence, testing is performed from the end-users perspective and from their point of view.

#2) Business Acceptance Testing (BAT)

This is to assess whether the Product meets the business goals and purposes or not.

BAT mainly focuses on business benefits (finances) which are quite challenging due to the changing market conditions/advancing technologies so that the current implementation may have to undergo changes which result in extra budgets. Even the Product passing the technical requirements may fail BAT due to these reasons.

#3) Contract Acceptance Testing (CAT)

This is a contract which specifies that once the Product goes live, within a predetermined period, the acceptance test must be performed and it should pass all the acceptance use cases.

Contract signed here is termed as Service Level Agreement (SLA), which includes the terms where the payment will be made only if the Product services are in-line with all the requirements, which means the contract is fulfilled.

Sometimes, this contract may happen before the Product goes live. Either the ways, a contract should be well defined in terms of the period of testing, areas of testing, conditions on issues encountered at later stages, payments, etc.

#4) Regulations/Compliance Acceptance Testing (RAT)

This is to assess whether the Product violates the rules and regulations that are defined by the government of the country where it is being released. This may be unintentional but will impact negatively on the business.

Usually, the developed Product/application that is intended to be released all over the world, has to undergo RAT, as different countries/regions have different rules and regulations defined by its governing bodies.

If any of the rules and regulations are violated for any country, then that country or the specific region in that country will not be allowed to use the Product and is considered as a Failure. Vendors of the Product will be directly responsible if the Product is released even though there is a violation.

#5) Operational Acceptance Testing (OAT)

This is to assess the operational readiness of the Product and is a non-functional testing. It mainly includes testing of recovery, compatibility, maintainability, technical support availability, reliability, fail-over, localization etc.

OAT mainly assures the stability of the Product before releasing it to the production.

LEVELS OF TESTING

4. Acceptance Testing OR User Acceptance Testing

Acceptance Criteria: Acceptance criteria are defined on the basis of the following attributes

- Functional Correctness and Completeness
- Data Integrity
- Data Conversion
- Usability
- Performance
- Timeliness
- Confidentiality and Availability
- Installability and Upgradability
- Scalability
- Documentation

LEVELS OF TESTING

4. Acceptance Testing OR User Acceptance Testing

Exit Criteria::

- The User Acceptance test: focuses mainly on the functionality thereby validating the fitness-for-use of the system by the business user. The user acceptance test is performed by the users and application managers.
- **The Operational Acceptance test:** also known as Production acceptance test validates whether the system meets the requirements for operation. In most of the organization the operational acceptance test is performed by the system administration before the system is released. The operational acceptance test may include testing of backup/restore, disaster recovery, maintenance tasks and periodic check of security vulnerabilities.
- **Contract Acceptance Testing** means that a developed software is tested against certain criteria and specifications which are predefined and agreed upon in a contract. The project team defines the relevant criteria and specifications for acceptance at the same time when the team agrees on the contract itself. Acceptance should be formally defined when the contract is agreed.
- **Compliance/Regulation acceptance testing:** It is also known as regulation acceptance testing is performed against the regulations which must be adhered to, such as governmental, legal or safety regulations.

LEVELS OF TESTING

The following diagram explains the fitment of acceptance testing in the software development life cycle.



LEVELS OF TESTING

SUMMARY

But apart from that before you take a look at differences first of all take a brief look at above than go for differences

System Testing	Integration Testing
1. In system testing we test the complete system as a whole to check whether the system is properly working or not means as per the requirements or not	1. In integration testing we test the modules to see whether they are integrating properly or not by combining the modules and tested as a group.
2. In system testing testers always have to concentrate on both functional and non-functional testing like performance, load, stress, security, recovery testing and so on.	2. In integration testing testers have to concentrate on functional testing means main focus on how two modules are combined and tested as a group.
3. For performing this testing system must be integrated tested.	3. For performing this testing system must be unit tested before.
4. It starts from the requirements specifications.	4. It starts from the interface specification.
5. System Testing does not test the visibility of code.	5. Integration Testing test the visibility of the integration structure.
7 In System Testing Tester pays attention to the system functionality.	7. In Integration Testing Tester pays attention
8. It is always only the kind of Black Box Testing.	8. It is a kind of both White Box Testing and Black Box Testing.

LEVELS OF TESTING

Alpha Testing	Beta Testing (Field Testing)
1. It is always performed by the developers/ Independent Testing Team at the software development site.	1. It is always performed by the customers / Independent Testing team at their own site
2. Alpha Testing is not open to the market and public	2. Beta Testing is always open to the market and public (not all).
3. Alpha Testing is not known by any other different name	3. Beta Testing is also known by the name Field Testing means it is also known as field testing
4. It is always performed in Virtual Environment.	4. It is performed in Real Time Environment.
5. It is the form of Acceptance Testing.	5. It is also the form of Acceptance Testing.
6. It comes under the category of both White Box Testing and Black Box Testing.	6. It is only a kind of Black Box Testing.
7. It is considered as the User Acceptance Testing (UAT) which is done at developer's area.	7. It is also considered as the User Acceptance Testing (UAT) which is done at Customers or users area.
8. It is always performed at the developer's premises in the absence of the users.	8. It is always performed at the user's premises in the absence of the development team

TO
NINETH LECTURE
OF

Software Testing Techniques and Strategies

COURSE CODE :SE-481

LEVELS OF TESTING

SUMMARY

But apart from that before you take a look at differences first of all take a brief look at above than go for differences

System Testing	Integration Testing
1. In system testing we test the complete system as a whole to check whether the system is properly working or not means as per the requirements or not	1. In integration testing we test the modules to see whether they are integrating properly or not by combining the modules and tested as a group.
2. In system testing testers always have to concentrate on both functional and non-functional testing like performance, load, stress, security, recovery testing and so on.	2. In integration testing testers have to concentrate on functional testing means main focus on how two modules are combined and tested as a group.
3. For performing this testing system must be integrated tested.	3. For performing this testing system must be unit tested before.
4. It starts from the requirements specifications.	4. It starts from the interface specification.
5. System Testing does not test the visibility of code.	5. Integration Testing test the visibility of the integration structure.
7 In System Testing Tester pays attention to the system functionality.	7. In Integration Testing Tester pays attention
8. It is always only the kind of Black Box Testing.	8. It is a kind of both White Box Testing and Black Box Testing.

LEVELS OF TESTING

Alpha Testing	Beta Testing (Field Testing)
1. It is always performed by the developers/ Independent Testing Team at the software development site.	1. It is always performed by the customers / Independent Testing team at their own site
2. Alpha Testing is not open to the market and public	2. Beta Testing is always open to the market and public (not all).
3. Alpha Testing is not known by any other different name	3. Beta Testing is also known by the name Field Testing means it is also known as field testing
4. It is always performed in Virtual Environment.	4. It is performed in Real Time Environment.
5. It is the form of Acceptance Testing.	5. It is also the form of Acceptance Testing.
6. It comes under the category of both White Box Testing and Black Box Testing.	6. It is only a kind of Black Box Testing.
7. It is considered as the User Acceptance Testing (UAT) which is done at developer's area.	7. It is also considered as the User Acceptance Testing (UAT) which is done at Customers or users area.
8. It is always performed at the developer's premises in the absence of the users.	8. It is always performed at the user's premises in the absence of the development team

Regression Testing

- ✓ Regression Testing is a type of testing that is done to verify that a code change in the software does not impact the existing functionality of the product.
- ✓ This is to make sure the product works fine with new functionality, bug fixes or any change in the existing feature.
- ✓ Previously executed test cases are re-executed in order to verify the impact of change.
- ✓ Regression Testing is a Software Testing type in which test cases are re-executed in order to check whether the previous functionality of the application is working fine and the new changes have not introduced any new bugs.
- ✓ This test can be performed on a new build when there is a significant change in the original functionality that too even in a single bug fix.
- ✓
- ✓ Regression means retesting the unchanged parts of the application.

REGRESSION TESTING

- ✓ Regression test should be a test, and not merely a check
- ✓ In the simplest terms, regression testing is a form of software testing that confirms or denies the functionality of software components after changes to a system
- ✓ A defect that emerges after a change to the software is known as a regression bug
- ✓ In the event that a change is made to a software component and a regression bug is discovered and fixed, more regression testing is required to confirm that resolving this issue does not in turn cause other issues

RETESTING VS REGRESSUION TESTING”

- **Regression testing**
- **Retesting** is done to make sure that the tests cases which failed in last execution are passed after the defects are fixed.
- **Regression** testing is not carried out for specific defect fixes.
- **Retesting** is carried out based on the defect fixes.

They can therefore occur in one and the same testing process, where:

- ✓ Change in requirements and code is modified according to the requirement
- ✓ You update your software with a new feature
- ✓ You detect a bug in your existing functionality
- ✓ You fix the bug.
- ✓ You test the existing functionality

Regression	Retesting
Involves testing a general area of the software.	Involves testing a specific feature of the software.
Is about testing software which 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 defect or bug is found in the code.

How is Regression Testing Best Implemented

There are typically three different methods for approaching regression testing on a project:



Regression Testing

Retest All: This method of regression testing simply re-tests the entirety of the software, from top to bottom. This is one of the methods for Regression Testing in which all the tests in the existing test bucket or suite should be re-executed. This is very expensive as it requires huge time and resources. In many cases, the majority of these tests are performed by automated tools, but often that is neither feasible nor necessary.

Regression Test Selection: Rather than a full re-test process, this method allows the team to choose a representative selection of tests that will approximate a full testing of the test suite, but require far less time or cost to do so.

- ✓ Instead of re-executing the entire test suite, it is better to select part of test suite to be run
- ✓ Test cases selected can be categorized as
- ✓ 1) Reusable Test Cases 2) Obsolete Test Cases 3). New test case

Test Case Prioritization: With a set of limited test cases, it is ideal to prioritize those tests. Try to prioritize tests which could impact both current and future builds of the software. Prioritize the test cases depending on business impact, critical & frequently used functionalities. Selection of test cases based on priority will greatly reduce the regression test suite.

Hybrid Regression Testing:

The hybrid technique is a combination of Regression test selection and Test case Prioritization. Rather than selecting the entire test suite, select only the test cases which are re-executed depending on their priority.

Selecting test cases for Regression Testing:

It was found from industry data that good number of the defects reported by customers were due to last minute bug fixes creating side effects and hence selecting the Test Case for regression testing is an art and not that easy.

Effective Regression Tests can be done by selecting following test case -

- Test cases which have frequent defects

- Functionalities which are more visible to the users

- Test cases which verify core features of the product

- Test cases of Functionalities which has undergone more and recent changes

- All Integration Test Cases

- All Complex Test Cases

- Boundary value test cases

- Sample of Successful test cases

- Sample of Failure test cases

Regression Test Overview

are required to execute again and again and running the same test cases again and again manually is time-consuming and tedious one too.

For Example, Consider a product X, in which one of the functionality is to trigger confirmation, acceptance, and dispatched emails when Confirm, Accept and Dispatch buttons are clicked.

Some issue occurs in the confirmation email and in order to fix the same, some code changes are done. In this case, not only the Confirmation emails need to be tested but Acceptance and Dispatched emails also needs to be tested to ensure that the change in the code has not affected them.

It is a testing method which is used to test the product for modifications or for any updates being done. It verifies that any modification in a product does not affect the existing modules of the product.

Verifying that the bugs are fixed and the newly added features have not created any problem in the previous working version of the software.

Testers perform Functional Testing when a new build is available for verification. The intent of this test is to verify the changes made in the existing functionality and the newly added functionality as well.

When this test is done, the tester should verify whether the existing functionality is working as expected and the new changes have not introduced any defect in functionality that was working before this change.

Regression test should be a part of the Release Cycle and must be considered in the test estimation.

What is Adhoc Testing?

- ✓ Adhoc testing is an informal testing type with an aim to break the system.
- ✓ Ad hoc Testing does not follow any structured way of testing and it is randomly done on any part of application.
- ✓ This testing is usually an unplanned activity.
- ✓ It does not follow any test design techniques to create test cases.
- ✓ In fact it does not create test cases altogether!
- ✓ This testing is primarily performed if the knowledge of testers in the system under test is very high.
- ✓ Testers randomly test the application without any test cases or any business requirement document.
- ✓ Main aim of this testing is to find defects by random checking. Adhoc testing can be achieved with the testing technique called **Error Guessing**. Error guessing can be done by the people having enough experience on the system to "guess" the most likely source of errors.

Adhoc Testing Disadvantages

- ✓ One of the main disadvantages of ad hoc testing is that the actual testing process is not documented since it does not follow a particular test case. This makes it more difficult for the tests to regenerate an error. Because, in order to get that error, the tester will need to remember the exact steps he/she took to get there, which is not always possible.
- ✓ Since this testing aims at finding defects through random approach, without any documentation, Defects will not be mapped to test cases. Hence, sometimes, it is very difficult to reproduce the defects as there are no test steps or requirements mapped to it.
- ✓ The **test** scenarios executed during the **ad-hoc testing** are not documented so the tester has to keep all the scenarios in their mind which he/she might not be able to recollect in future
- ✓ Ad hoc testing will be effective only if the tester is knowledgeable and experienced of the System Under Test. . If the tester does not have prior knowledge about the functionality of the application under test, ad hoc testing will not be useful and won't be able to identify any errors.

Occasionally, as a result of invalid test cases that are developed by the tester, invalid errors are reported. This can become an issue in the following error fixing processes

- ✓ Ad hoc testing also does not guarantee that all errors will be found. The success of ad hoc testing relies on the skill and knowledge of the tester. Since there are no previously created or documented test cases, the amount of time, effort and resources that go into these tests remains unspecified. Finding one error could take anything from a few minutes to a few hours or longer.

Advantages Of Ad Hoc Testing

- ✓ One of the main advantages of ad hoc testing is that it is able to identify any errors that would usually go unnoticed during formal testing methods. This can save a lot of time as it requires none of the planning that structured testing does.
- ✓ Another advantage is that testers get to explore the application freely, according to their own knowledge and understanding of the application.
- ✓ They can then execute various tests as they go along, helping identify errors throughout the process.
- ✓ Thirdly, testers and developers of the application can easily test the app themselves, as it does not require test cases. This allows the developers to create more efficient and bug-free code easily.
- ✓ Ad hoc testing can also be combined with other testing techniques and executed thereafter to produce more effective and informative results overall

When And When Not To Conduct Ad hoc testing

- ✓ Ad hoc testing is commonly conducted when there is a lack of time to perform longer and more exhaustive testing processes.
- ✓ The more thorough testing method includes preparing test requirements documents, test cases, and test case designs.
- ✓ The ideal time to conduct ad hoc testing is after the completion of all formal testing techniques.
- ✓ However, ad hoc testing can also be conducted in the middle of the software development, after the complete development of the software, or after a few modules have already been developed.

When Not To Conduct Ad hoc testing

- ✓ It is important to take note of the few scenarios when ad hoc testing is not recommended. A few conditions when ad hoc testing should not be conducted include:
 - When Beta testing is being conducted
 - In test cases which already have existing errors

Types of Adhoc testing

Buddy Testing

This type of adhoc testing is conducted with a minimum of two people. It takes place after unit testing of a module has been conducted and completed. This type of testing can also be considered a combination of both system and unit testing. Two buddies mutually work on identifying defects in the same module. Mostly one buddy will be from development team and another person will be from testing team. Buddy testing helps the testers develop better test cases and development team can also make design changes early. This testing usually happens after Unit Testing completion.

Pair testing

Similar to ‘buddy testing’ in some ways, ‘pair testing’ involves a pair of testers working together on the modules for testing. The two testers will share ideas, knowledge, and opinions over the same machine in order to identify defects or errors.

This method of testing involves using testers who are paired according to their expertise and knowledge levels, allowing for different insights to any problem they identify. The two testers will share the same setup, also sharing the work of testing and documenting all observations between them. This method of testing also allows for one tester to execute the tests, while the other can take notes on the findings. One person can execute the tests and another person can take notes on the findings. Roles of the persons can be a tester and scribe during testing.

Comparison Buddy and Pair Testing: Buddy testing is combination of unit and System Testing together with developers and testers but Pair testing is done only with the testers with different knowledge levels. (Experienced and non-experienced to share their ideas and views)

Monkey Testing

Randomly test the product or application without test cases with a goal to break the system.

Due to the random nature of the testing, this method has earned the name ‘monkey testing’. Monkey testing is most commonly done in the unit testing level. Here, testers randomly test the application or product without test cases. The tester’s main goal is to analyze the data or tests in completely random ways, ensuring that the system is able to withstand any crash.

Testers provide the software with random inputs and observe their corresponding outputs. Based on the output data, they can determine any errors, inconsistencies or system crashes better.

Best practices to follow Adhoc testing

Following best practices can ensure effective Adhoc Testing

□ Good business knowledge

Testers should have good knowledge of the business and clear understanding of the requirements- Detailed knowledge of the end to end business process will help find defects easily. Experienced testers find more defects as they are better at error guessing.

□ Test Key Modules

Key business modules should be identified and targeted for ad-hoc testing. Business critical modules should be tested first to gain confidence on the quality of the system.

□ Record Defects

All defects need to be recorded or written in a notepad. Defects must be assigned to developers for fixing. For each valid defect, corresponding test cases must be written & must be added to planned test cases.

These Defect findings should be made as lesson learned and these should be reflected in our next system while we are planning for test cases. **Conclusion:**

The advantage of Ad-hoc testing is to check for the completeness of testing and find more defects than planned testing. The defect catching test cases are added as additional test cases to the planned test cases.

Ad-hoc Testing saves lot of time as it doesn't require elaborate test planning, documentation and Test Case design.

Dynamic Testing

• What is Black Box Testing?

- Black box testing is a software testing techniques in which functionality of the software under test (SUT) is tested without looking at the internal code structure, implementation details and knowledge of internal paths of the software. This type of testing is based entirely on the software requirements and specifications.
- In Black Box Testing we just focus on inputs and output of the software system without bothering about internal knowledge of the software program.

Definition by ISTQB

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

• .

Black Box Testing **Dynamic Testing Techniques**

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

Levels Applicable To

- Black Box testing method is applicable to the following levels of software testing:
 - ✓ **Integration Testing**
 - ✓ **System Testing**
 - ✓ **Acceptance Testing**
- The higher the level, and hence the bigger and more complex the box, the more black box testing method comes into use.

Dynamic Testing Techniques

WORKING PROCESS OF BLACK BOX TESTING TECHNIQUE

Below are the steps which explain the working process of Black Box Testing

Step 1 Input: Requirement and functional specification of the system are examined. High level design documents and application block source code are also examined. Tester chooses valid input and rejects the invalid inputs.

Step 2 Processing Unit: Do not concern with the internal working of the system. In processing unit tester constructs test cases with the selected input and execute them.

Tester also performs Invalid and Valid inputs

Partitions Output load testing, stress testing, security review and globalization testing. If any defect is detected it will be fixed and re-tested.

Step 3 Output: After all these testing, tester gets desired output and prepares final report

Dynamic Testing Techniques

Black Box Testing –Perform by the Tester

- Here are the generic steps followed to carry out any type of Black Box Testing.
- Initially requirements and specifications of the system are examined.
- Tester chooses valid inputs (positive test scenario) to check whether SUT processes them correctly. Also some invalid inputs (negative test scenario) are chosen to verify that the SUT is able to detect them.
- Tester determines expected outputs for all those inputs.
- Software tester constructs test cases with the selected inputs.
- The test cases are executed.
- Software tester compares the actual outputs with the expected outputs.
- Defects if any are fixed and re-tested.

Dynamic Testing Techniques

Black Box Testing Advantages

- Tests are done from a user's point of view and will help in exposing discrepancies in the specifications.
- Tester need not know programming languages or how the software has been implemented.
- Tests can be conducted by anyone independent from the developers, allowing for an objective perspective and the avoidance of developer-bias.
- Test cases can be designed as soon as the specifications are complete.

Black Box Testing Disadvantages

- It is difficult to identify all possible inputs
- Only a small number of possible inputs can be tested and possibility of many program paths will be left untested.
- Chances of having unidentified paths during this testing
- Without clear specifications, which are the situation in many projects, test cases will be difficult to design.

Dynamic Testing Techniques

Tools used for Black Box Testing:

- Tools used for Black box testing largely depends on the type of black box testing you are doing.
- For Functional/ Regression Tests you can use - QTP, Selenium
- For Non-Functional Tests, you can use - Loadrunner, Jmeter
-

Types of Black Box Testing

- There are many types of Black Box Testing but following are the prominent ones -
 - **Functional testing** - This black box testing type is related to functional requirements of a system; it is done by software testers.
 - **Non-functional testing** - This type of black box testing is not related to testing of a specific functionality, but non-functional requirements such as performance, scalability, usability.
- **Regression testing** - Regression Testing is done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code

Dynamic Testing Techniques

I. Black Box Testing Techniques

- Following are some techniques that can be used for designing black box tests.
- **Equivalence partitioning:** It is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.
- **Boundary Value Analysis:** It is a software test design technique that involves determination of boundaries for input values and selecting values that are at the boundaries and just inside/ outside of the boundaries as test data.
- **Cause Effect Graphing:** It is a software test design technique that involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph, and generating test cases accordingly
- **Decision Table Testing:** A decision table puts causes and their effects in a matrix. There is unique combination in each column.
- **Error Guessing:**
 - This is purely based on previous experience and judgment of tester. Error Guessing is the art of guessing where errors can be hidden. For this technique there are no specific tools, writing the test cases that cover all the application paths.
- **State Base Testing**

Dynamic Testing Techniques

BLACK BOX TESTING - I

i. EQUIVALENCE CLASS PARTITIONING:

- Equivalence Class Partitioning is a test case design techniques in black box testing. Equivalence partitioning is a Test Case Design Technique to divide the input data of software into different equivalence data classes.
- Test cases are designed for equivalence data class. The equivalence partitions are frequently derived from the requirements specification for input data that influence the processing of the test object.
- A use of this method reduces the time necessary for testing software using less and effective test cases
- Equivalence Partitioning = Equivalence Class Partitioning = ECP
- It can be used at any level of software for testing and is preferably a good technique to use first. In this technique, only one condition to be tested from each partition. Because we assume that, all the conditions in one partition behave in the same manner by the software. In a partition, if one condition works other will definitely work. Likewise we assume that, if one of the conditions does not work then none of the conditions in that partition will work.

Dynamic Testing Techniques

BLACK BOX TESTING

i. EQUIVALENCE CLASS PARTITIONING:

- Equivalence partitioning is a testing technique where input values set into classes for testing.
- **Valid Input Class = Keeps all valid inputs.**
- **Invalid Input Class = Keeps all Invalid inputs**
- **How is this partitioning performed while testing:**
 - 1. If an input condition specifies a range, one valid and one two invalid classes are defined.
 - 2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
 - 3.If an input condition specifies a member of a set, one valid and one invalid equivalence class is defined.
 - 4. If an input condition is Boolean, one valid and one invalid class is defined.

Dynamic Testing

BLACK BOX TESTING Techniques

1. EQUIVALENCE CLASS PARTITIONING EXAMPLE:

- Let's consider the behavior of tickets in the Flight reservation application, while booking a new flight.
- Ticket values 1 to 10 are considered valid & ticket is booked. While value 11 to 99 is considered invalid for reservation and error message will appear, "Only ten tickets may be ordered at one time".



Here is the test condition

1. Any Number greater than 10 entered in the reservation column (let say 11) is considered invalid.
2. Any Number less than 1 that is 0 or below, then it is considered invalid.
3. Numbers 1 to 10 are considered valid
4. Any 3 Digit Number say -100 is invalid We cannot test all the possible values because if done, number of test cases will be more than 100. To address this problem, we use equivalence partitioning hypothesis where we divide the possible values of tickets into groups or sets as shown below where the system behavior can be considered the same

Dynamic Testing

Techniques

BLACK BOX TESTING-1

The divided sets are called Equivalence Partitions or Equivalence Classes. Then we pick only one value from each partition for testing. The hypothesis behind this technique is **that if one condition/value in a partition passes all others will also pass.** Likewise, **if one condition in a partition fails, all other conditions in that partition will fail.**

Dynamic Testing Techniques

BLACK BOX TESTING - I

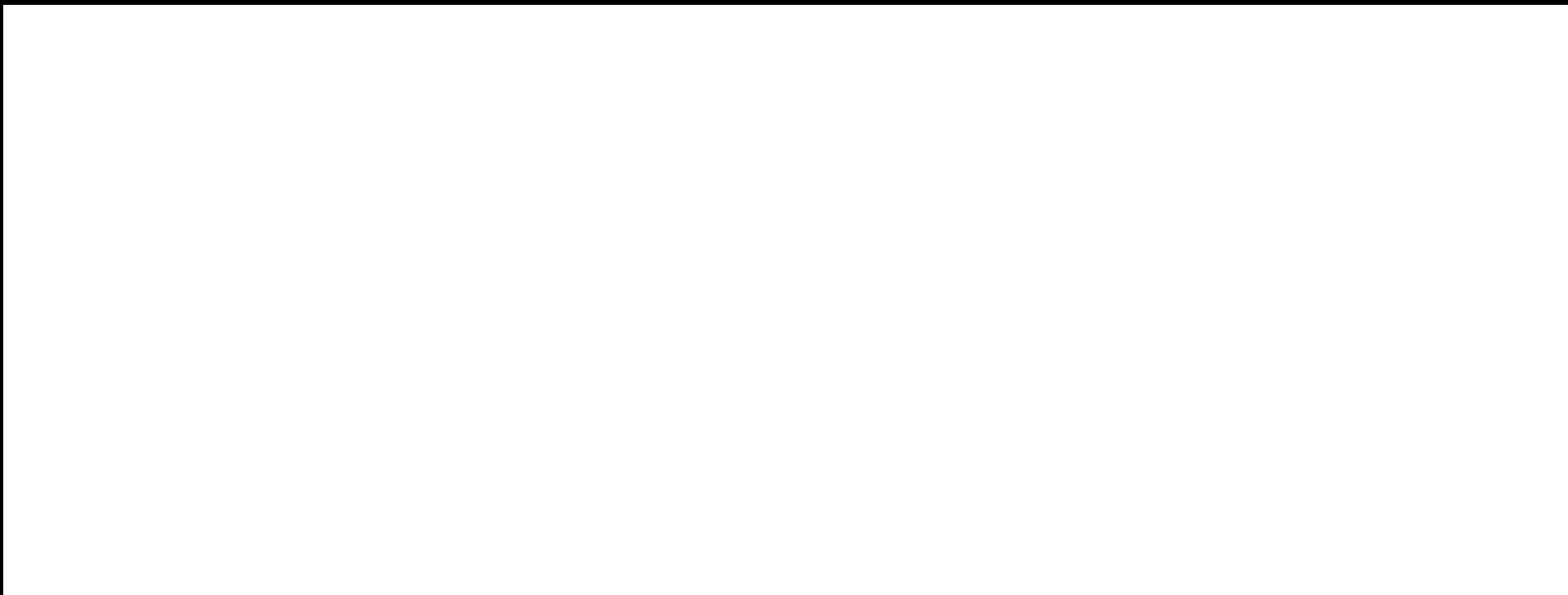
1. EQUIVALENCE CLASS PARTITIONING:

- Example 2
- § A text field permits only numeric characters
- § Length must be 6-10 characters long



At the time of testing, test 4 and 12 as invalid values and 7 as valid one.

It is easy to test input ranges 6–10 but harder to test input ranges 2-600. Testing will be easy in the case of lesser test cases but you should be very careful. Assuming, valid input is 7. That means, you believe that the developer coded the correct valid range (6-10).



Dynamic Testing Techniques

BLACK BOX TESTING - I

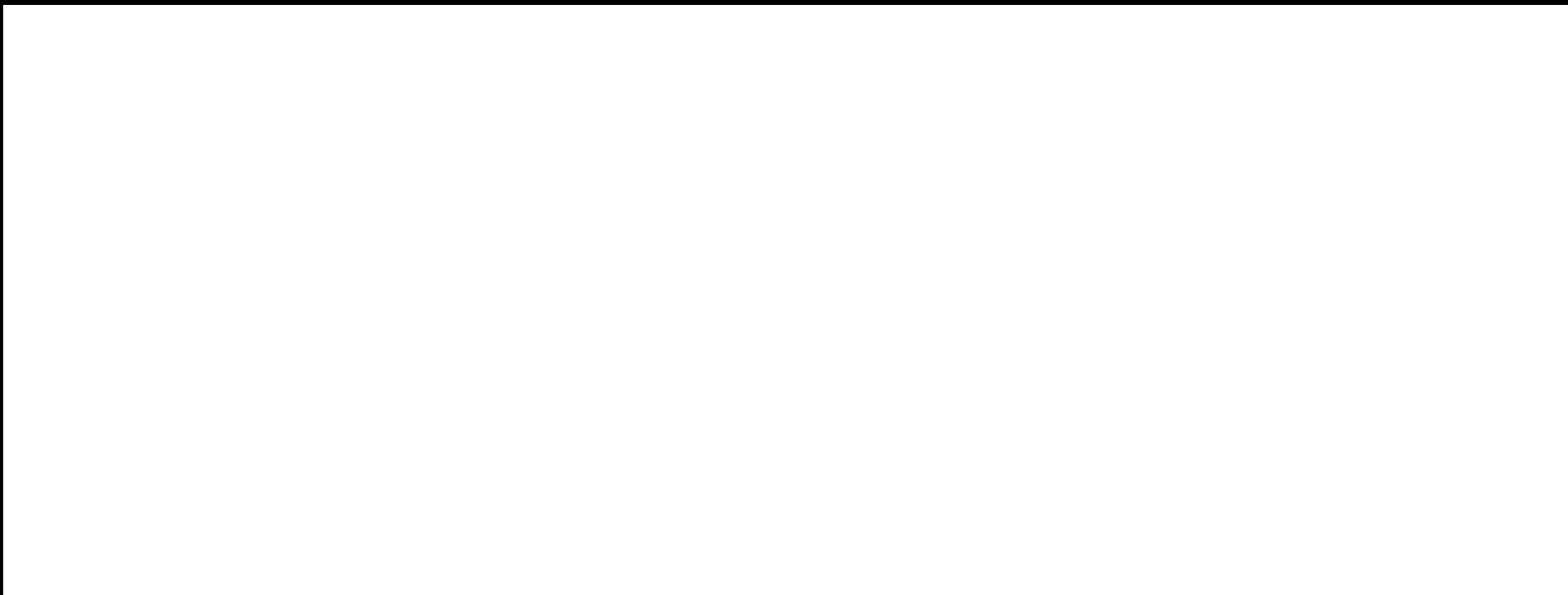
1. EQUIVALENCE CLASS PARTITIONING:

- Example 2
- § A text field permits only numeric characters
- § Length must be 6-10 characters long



At the time of testing, test 4 and 12 as invalid values and 7 as valid one.

It is easy to test input ranges 6–10 but harder to test input ranges 2-600. Testing will be easy in the case of lesser test cases but you should be very careful. Assuming, valid input is 7. That means, you believe that the developer coded the correct valid range (6-10).



THANKYOU



TO
TENTH LECTURE
OF

Software Testing Techniques and Strategies

COURSE CODE :SE-481

Dynamic Testing

• What is Black Box Testing?

- Black box testing is a software testing techniques in which functionality of the software under test (SUT) is tested without looking at the internal code structure, implementation details and knowledge of internal paths of the software. This type of testing is based entirely on the software requirements and specifications.
- In Black Box Testing we just focus on inputs and output of the software system without bothering about internal knowledge of the software program.

Definition by ISTQB

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

• .

Dynamic Testing Techniques

i. Black Box Testing Techniques

- Following are some techniques that can be used for designing black box tests.
- **Equivalence partitioning:** It is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.
- **Boundary Value Analysis:** It is a software test design technique that involves determination of boundaries for input values and selecting values that are at the boundaries and just inside/outside of the boundaries as test data.
- **Cause Effect Graphing:** It is a software test design technique that involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph, and generating test cases accordingly
- **Decision Table Testing:** A decision table puts causes and their effects in a matrix. There is unique combination in each column.
- **Error Guessing:**
 - This is purely based on previous experience and judgment of tester. Error Guessing is the art of guessing where errors can be hidden. For this technique there are no specific tools, writing the test cases that cover all the application paths.
- **State base Testing:**

Dynamic Testing Techniques

Black Box Testing

- **TEST DESIGN TECHNIQUES – II Boundary Value Analysis**

- In software testing, the **Boundary Value Analysis (BVA)** is a black box test design technique based on test cases. This technique is applied to see if there are any bugs at the boundary of the input domain. Thus, with this method, there is no need of looking for these errors at the center of this input.
- BVA helps in testing the value of boundary between both valid and invalid boundary partitions. With this technique, the boundary values are tested by the creation of test cases for a particular input field.
- In Boundary Value Analysis, you test boundaries between equivalence partitions
- **Boundary value analysis** is a test case design techniques in **black box testing**.
- Normally Boundary value analysis is part of **stress and negative testing**.
- In our earlier example instead of checking, one value for each partitions you will check the values at the partitions like 0, 1, 10, and 11 and so on. As you may observe, you test values at **both valid and invalid boundaries**.

Dynamic Testing Techniques

Black Box

- **Testing**
• **Boundary Value Analysis** is also called **range checking**



The extreme ends or boundary partitions might depict the values of lower-upper, start-end, maximum-minimum, inside-outside etc.

In general, the BVA technique comes under the Stress and Negative Testing.

This technique is an easy, quick and brilliant way to catch any input errors that might occur to interrupt the functionality of a program.

So, to save their time and to cut the testing procedure short, the experts delivering software testing and quality management services rely on the Boundary Value Analysis method.

For testing of data related to boundaries and ranges, the method is considered as a very suitable one.

Dynamic Testing Techniques

BLACK BOX TESTING

- **EXAMPLE OF BVA:**
- **For Example:** Using Boundary Value Analysis technique tester creates test cases for required input field. For example; an Address text box which allows maximum 500 characters. So, writing test cases for each character will be very difficult so that will choose boundary value analysis.

Dynamic Testing Techniques

Black Box Testing

ii. Boundary Value Analysis

- **An Example of Boundary Value Analysis:**
- Consider the testing of a software program that takes the integers ranging between the values of -100 to +100. In such a case, three sets of the valid equivalent partitions are taken, which are – the negative range from -100 to -1, zero (0), and the positive range from 1 to 100.
- Each of these ranges has the minimum and maximum boundary values. The Negative range has a lower value of -100 and the upper value of -1. The Positive range has a lower value of 1 and the upper value of 100
- While testing these values, one must see that when the boundary values for each partition are selected, some of the values overlap. So, the overlapping values are bound to appear in the test conditions when these boundaries are checked.
- These overlapping values must be dismissed so that the redundant test cases can be eliminated.
- So, the test cases for the input box that accepts the integers between -100 and +100 through BVA are:
- Test cases with the data same as the input boundaries of input domain: -100 and +100 in our case.
- Test data having values just below the extreme edges of input domain: -101 and 99
- Test data having values just above the extreme edges of input domain: -99 and 101
- This is a very basic example to understand the BVA testing technique!
- With this technique, it is quite easy to test a small set of data in place of testing the whole lot of data sets. This is why, in software testing and quality management services, this method of testing is adopted more often.

Black Box Testing

ii. Boundary Value Analysis

- **Examples 1. Input Box should accept the Number 1 to 10**

- Here we will see the Boundary Value Test Cases

- **INVALID VALID INVALID**

- **0 1---- 10 11 - - - so on**

- Test Scenario Description Expected Outcome

- Boundary Value = 0 System should NOT accept

- **Boundary Value = 1** System should accept

- Boundary Value = 2 System should accept

- Boundary Value = 9 System should accept


- **Boundary Value = 10** System should accept

- Boundary Value = 11 System should NOT accept

Dynamic Testing Techniques

Black Box Testing

ii. Boundary Value Analysis

- **Example 2.** Suppose you have very important tool at office, accept valid User Name and Password field to work on that tool, and accept minimum 8 characters and maximum 12 characters. Valid range 8--12, Invalid range 7 or less than 7 and invalid range 13 or more than 13.
- 
- Write Test Cases for Valid partition value, Invalid partition value and exact boundary value. □
 - Test Cases 1: Consider password length less than 8 i.e 7.
 - **Test Cases 2: Consider password of length exactly 8.**
 - Test Cases 3: Consider password of length above 9 and 11.
 - **Test Cases 4: Consider password of length exactly 12.**
 - Test Cases 5: Consider password of length more than 12 i.e 13.

Dynamic Testing Techniques

Black Box Testing

ii. Boundary Value Analysis

- Example 3 Test cases for the application whose input box accepts numbers between 1-1000. Valid range 1-1000, Invalid range 0 and Invalid range 1001 or more.



Write Test Cases for Valid partition value, Invalid partition value and exact boundary value.

- Test Cases 1: Consider test data exactly as the input boundaries of input domain i.e. values 1 and 1000.
- Test Cases 2: Consider test data with values just below the extreme edges of input domains i.e. values 0 and 999.
- Test Cases 3: Consider test data with values just above the extreme edges of input domain i.e. values 2 and 1001

Black Box Testing

- **Example : Equivalence**

- Following password field accepts minimum 6 characters and maximum 10 characters
- That means results for values in partitions 0-5, 6-10, 11-onwards should be equivalent



<i>Test Scenario #</i>	<i>Test Scenario Description</i>	<i>Expected Outcome</i>
1	Enter 0 to 5 characters in password field	System should not accept
2	Enter 6 to 10 characters in password field	System should accept
3	Enter 11 to onwards character in password field	System should not accept

ii. Boundary Value Analysis

- **Evaluation of Boundary Value Analysis as a Software Testing Technique**
 - **Boundary Value Analysis Advantages:**
 - The BVA technique of testing is quite easy to use and remember because of the uniformity of identified tests and the automated nature of this technique.
 - One can easily control the expenses made on the testing by controlling the number of identified test cases. This can be done with respect to the demand of the software that needs to be tested.
 - BVA is the best approach in cases where the functionality of a software is based on numerous variables representing physical quantities.
 - The technique is best at revealing any potential UI or user input troubles in the software.
 - The procedure and guidelines are crystal clear and easy when it comes to determining the test cases through BVA.
 - The test cases generated through BVA are very small.
 - **Boundary Value Analysis Disadvantages:**
 - This technique sometimes fails to test all the potential input values. And so, the results are unsure.
 - The dependencies with BVA are not tested between two inputs.
 - This technique doesn't fit well when it comes to Boolean Variables.
 - It only works well with independent variables that depict quantity.

Dynamic Testing Techniques

BLACK BOX TESTING

ii. Boundary Value Analysis

- In BVA, the software testers have found a fairly simple and correct testing method. This technique can be one of the most important testing techniques if used with care and correctness.
- However, the technique is a little limiting when there are some issues with variable dependency or when a foresight is needed for the functionality of a system..
- But even today, many of the companies and experts delivering the software testing services, have adopted the BVA methods of testing.

BLACK BOX TESTING

• iii. What is DECISION TABLE

- Decision table testing is black box test design technique to determine the test scenarios for complex business logic.
- A decision table is an excellent tool to use in both testing and requirements management.
- It is **a structured exercise to formulate requirements when dealing with complex business rules**.
- They can also be applied when the action of the software depends on many logical decisions.
- Decision tables are used to model complicated logic.
- They can make it easy to see that all possible combinations of conditions and when conditions are missed, it is easy to see this.
- **A decision table 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, optimized one excludes impossible combinations of conditions and combinations of inputs that don't have any effect on outputs.
- The decision table is a software testing technique which is used for testing the system behavior for different input combinations. This is a systematic approach where the different input combinations and their corresponding system behavior are captured in a tabular form.
- A decision table is a good way to deal with different combinations of inputs with their associated outputs and also called cause-effect table.
- We can apply Equivalence Partitioning and Boundary Value Analysis techniques to only specific conditions or inputs. Although, if we have dissimilar inputs that result in different actions being taken or secondly we have a business rule to test that there are different combination of inputs which result in different actions. We use decision table to test these kinds of rules or logic.

BLACK BOX TESTING

iii. DECISION TABLE

- **Why use decision tables?** A decision table is an outstanding technique used for testing and requirements management. Some of the reasons why the decision table is important include:
- Testing with all combination might be impractical.
- Decision tables are very much helpful in test design technique.
- It provides a regular way of stating complex business rules which benefits the developers as well as the testers.
- It is a structured exercise to prepare requirements when dealing with complex business rules.
- It is also used in model complicated logic
- Powerful visualization
- Compact and structured presentation
- Preventing errors is easier
- Avoid incompleteness and inconsistency
- Modular knowledge
- Group related rules into single table
- Combine tables to achieve decision

BLACK BOX TESTING

iii. DECISION TABLE

Method

- A Decision Table is usually divided into four quadrants as in the table below.

Condition being tested	Condition Statement	Condition Entries
Possible action to take	Action Statement	Action Entries

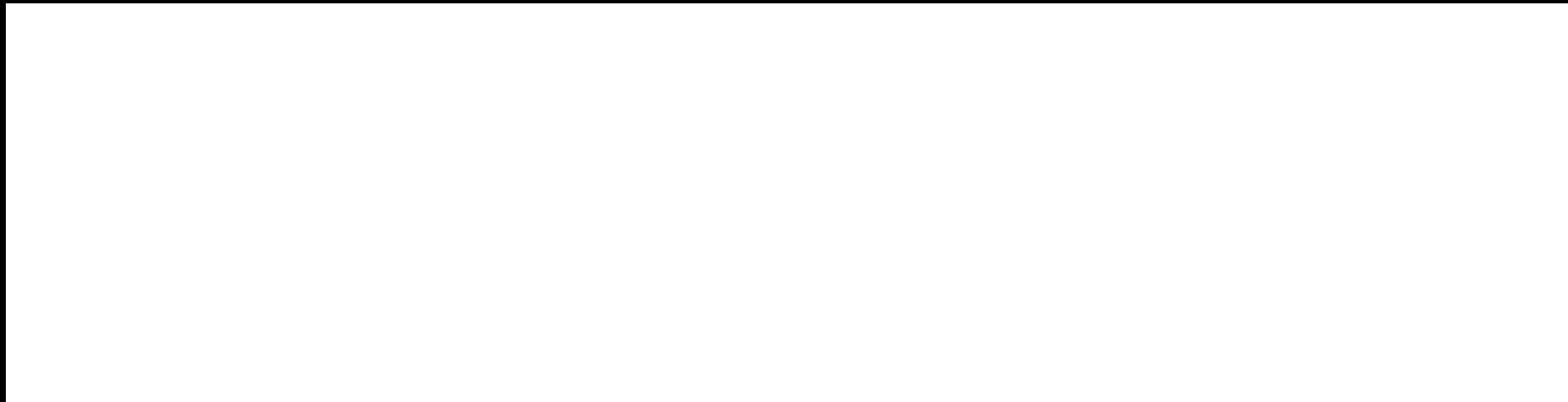
- The upper half of the table lists the conditions being tested while the lower half lists the possible actions to be taken.
- Each column represents a certain type of condition or rule
- 1. Rules are complete (i.e., every combination of decision table values including default combinations are inherent in the decision table)
- 2. Redundancy & Inconsistency The rules are consistent (i.e, there is not two actions for the same combinations of conditions).
- When using “don’t care” entries a level of care must be taken, using these entries can cause redundancy and inconsistency within a decision table.
- Using rule counts to check the completeness of the decision table can help to eliminate redundant rules within the table. An example of a decision table with a redundant rule

BLACK BOX TESTING

iii. DECISION TABLE

- Example 1: Login Screen

- 1.

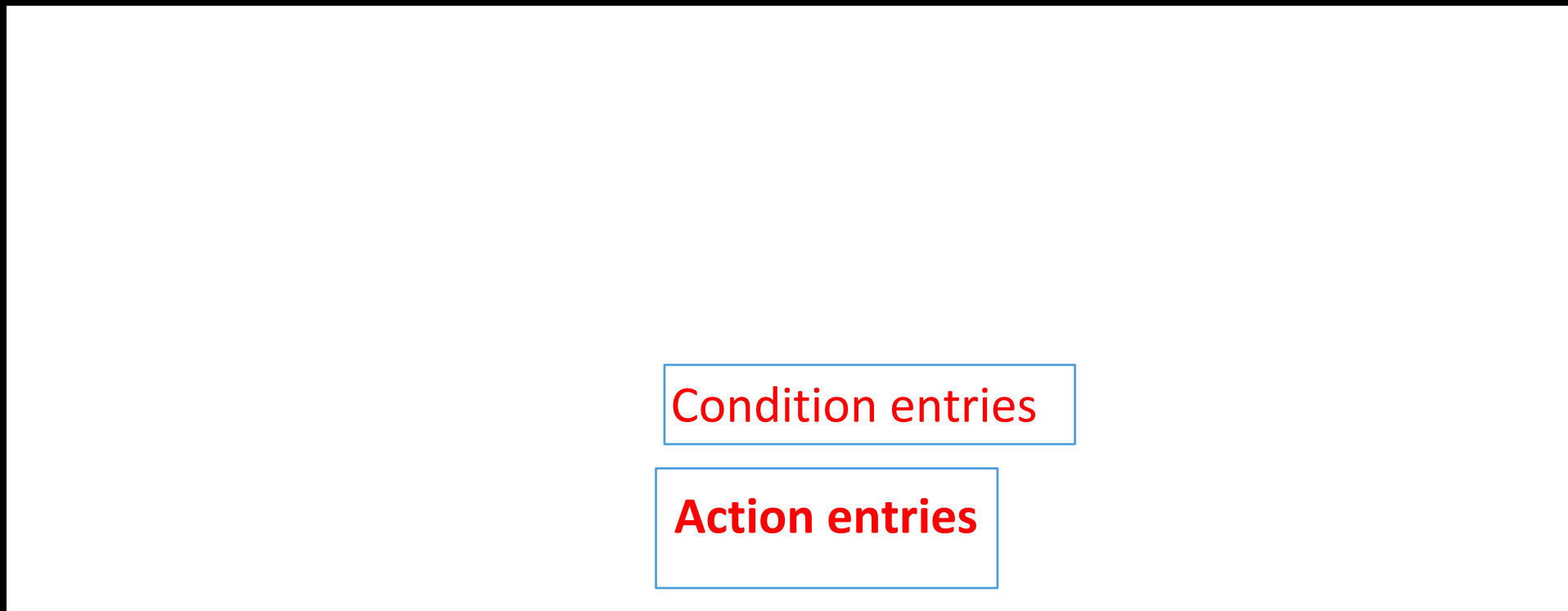


BLACK BOX TESTING

iii. DECISION TABLE

- Example 1: Login Screen

- 1.



BLACK BOX TESTING

iii. DECISION TABLE

BLACK BOX TESTING

iii. DECISION TABLE

BLACK BOX TESTING

iii. DECISION TABLE



- Note:
- Calculate how many columns are needed in the table. The number of columns depends on the number of conditions and the number of alternatives for each condition. If there are two conditions and each condition can be either true or false, you need 4 columns.

BLACK BOX TESTING

iii. DECISION TABLE



- Note:
- Calculate how many columns are needed in the table. The number of columns depends on the number of conditions and the number of alternatives for each condition. If there are two conditions and each condition can be either true or false, you need 4 columns.
- Mathematically,
- Determine maximum number of rules
- Number of rules = $\text{values}(\text{cond1}) * \text{values}(\text{cond2})$
- **Rules = $2 * 2 = 4$**

If there are three conditions there will be 8 columns and so on.

- Determine maximum number of rules
- Number of rules = $\text{values}(\text{cond1}) * \text{values}(\text{cond2}) * \text{values}(\text{cond3})$
- **Rules = $2 * 2 * 2 = 8$**

BLACK BOX TESTING

iii. DECISION TABLE

Example 2: Decision Base Table for Upload Screen

- Now consider a dialogue box which will ask the user to upload photo with certain conditions like
 - ✓ You can upload only '.jpg' format image
 - ✓ file size less than 32kb
 - ✓ resolution 137*177.
- **If any of the conditions fails the system will throw corresponding error message stating the issue and if all conditions are met photo will be updated successfully**

BLACK BOX TESTING

iii. DECISION TABLE

Example 2: Decision Base Table for Upload Screen

- Now consider a dialogue box which will ask the user to upload photo with certain conditions like
- You can upload only '.jpg' format image
- file size less than 32kb
- resolution 137*177.
- If any of the conditions fails the system will throw corresponding error message stating the issue and if all conditions are met photo will be updated successfully
- **No. of possible columns=Three input conditions with possible two possible outcomes=**

BLACK BOX TESTING

iii. DECISION TABLE

Example 2: Decision Base Table for Upload Screen

- Now consider a dialogue box which will ask the user to upload photo with certain conditions like
 - ✓ You can upload only '.jpg' format image
 - ✓ file size less than 32kb
 - ✓ resolution 137*177.
- If any of the conditions fails the system will throw corresponding error message stating the issue and if all conditions are met photo will be updated successfully
- **No. of possible columns=Three input conditions with possible two possible outcomes= $C1 * C2 * C3 = 2 * 2 * 2 = 8$**

BLACK BOX TESTING

Example 2: Decision Base Table for Upload Screen

Now consider a dialogue box which will ask the user to upload photo with certain conditions like

- ✓ You can upload only '.jpg' format image
- ✓ file size less than 32kb
- ✓ resolution 137*177.

If any of the conditions fails the system will throw corresponding error message stating the issue and if all conditions are met photo will be updated successfully

No. of possible columns=Three input conditions with possible two possible outcomes= $C1 * C2 * C3$
= $2 * 2 * 2 = 8$

Conditions	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Format	conditions							
Size								
resolution								
Output								
ACTION								

BLACK BOX TESTING

Example 2: Decision Base Table for Upload Screen

Now consider a dialogue box which will ask the user to upload photo with certain conditions like

- ✓ You can upload only '.jpg' format image
- ✓ file size less than 32kb
- ✓ resolution 137*177.

If any of the conditions fails the system will throw corresponding error message stating the issue and if all conditions are met photo will be updated successfully

No. of possible columns=Three input conditions with possible two possible outcomes=

$$C1 * C2 * C3 = 2 * 2 * 2 = 8$$

Conditions	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Format	.jpg	.jpg	.jpg	.jpg	Not .jpg	Not .jpg	Not .jpg	Not .jpg
Size	Less than 32kb	Less than 32kb	>= 32kb	>= 32kb	Less than 32kb	Less than 32kb	>= 32kb	>= 32kb
resolution	137*177	Not 137*177	137*177	Not 137*177	137*177	Not 137*177	137*177	Not 137*177
Output								

BLACK BOX TESTING

Example 2: Decision Base Table for Upload Screen

Now consider a dialogue box which will ask the user to upload photo with certain conditions like

- ✓ You can upload only '.jpg' format image
- ✓ file size less than 32kb
- ✓ resolution 137*177.

If any of the conditions fails the system will throw corresponding error message stating the issue and if all conditions are met photo will be updated successfully

No. of possible columns=Three input conditions with possible two possible outcomes=

$$C1 * C2 * C3 = 2 * 2 * 2 = 8$$

Conditions	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Format	.jpg	.jpg	.jpg	.jpg	Not .jpg	Not .jpg	Not .jpg	Not .jpg
Size	Less than 32kb	Less than 32kb	>= 32kb	>= 32kb	Less than 32kb	Less than 32kb	>= 32kb	>= 32kb
resolution	137*177	Not 137*177	137*177	Not 137*177	137*177	Not 137*177	137*177	Not 137*177
Output	Photo uploaded	Error message resolution mismatch	Error message size mismatch	Error message size and resolution mismatch	Error message for format mismatch	Error message for format and resolution mismatch	Error message for format and size mismatch	Error message for format, size, and resolution mismatch

BLACK BOX TESTING

iii. DECISION TABLE

- **Example 3:**

Let's take another example. If you are a new customer and you want to open a credit card account then there are three conditions first you will get a 15% discount on all your purchases today, second if you are an existing customer and you hold a loyalty card, you get a 10% discount and third if you have a coupon, you can get 20% off today (but it can't be used with the 'new customer' discount). Discount amounts are added, if applicable.

iii. DECISION TABLE

there are three conditions first you will get a 15% discount on all your purchases today, second if you are an existing customer and you hold a loyalty card, you get a 10% discount and third if you have a coupon, you can get 20% off today (but it can't be used with the 'new customer' discount). Discount amounts are added, if applicable.

New customers with coupon

New customers without a coupon

Existing customers with a loyalty card and no coupon

Existing customers without a loyalty card and no coupon

Existing customers with a loyalty card and Coupon

Existing customers without loyalty and with a coupon

- To create a decision table, you will have to find conditions and actions.

Conditions:

- Customer (New customer , Existing Customer with coupon
- Coupon
- Loyalty card

Actions/Outcomes

- 10%
- 15%
- 20%
- No Discount

BLACK BOX TESTING

iii. DECISION TABLE

- To create a decision table, you will have to find conditions and actions.

Conditions:

- Customer (New customer , Existing Customer)
- Coupon
- Loyalty card

No. of possible columns=Three input conditions with possible two possible outcomes=

$$C1 * C2 * C3 = 2 * 2 * 2 = 8$$

Actions/Outcomes

- 10%
- 15%
- 20%
- No Discount

BLACK BOX TESTING

hold a loyalty card, you get a 10% discount and third if you have a coupon, you can get 20% off today (but it can't be used with the 'new customer' discount). Discount amounts are added, if applicable.

Customer (New customer	<input type="checkbox"/> N.C,	Existing Customer	<input type="checkbox"/> E.C
Coupon-	<input type="checkbox"/> True	No Coupon	<input type="checkbox"/> False
Loyalty card	-- <input type="checkbox"/> True		

[illegible]

BLACK BOX TESTING

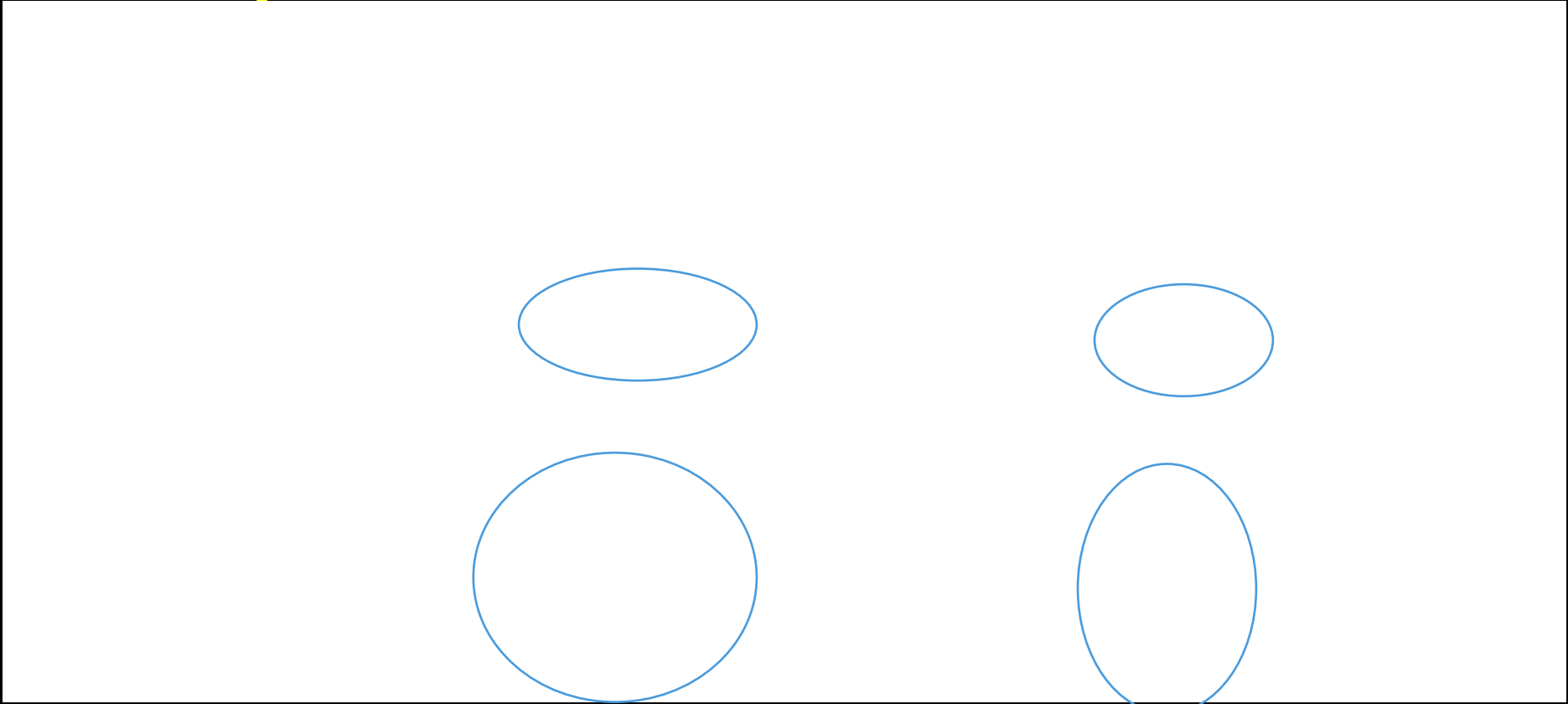
iii. DECISION TABLE

- **Example :4** Company X sells merchandise to wholesale and retail outlets. Wholesale customers receive a two percent discount on all orders. The company also encourages both wholesale and retail customers to pay cash on delivery by offering a two percent discount for this method of payment. Another two percent discount is given on orders of 50 or more units. Each column represents a certain type of order

BLACK BOX TESTING

- Simplify the decision table and write down the test cases

Example 1:



Conditions	Case 1	Case 2	Case 3
Email	F	T	.
Password	-	F	T
Actions			
Expected results	Error Please Enter email	Error Please Enter email	Login Processed

BLACK BOX TESTING

Simplifying Decision table

If you are a new customer and you want to open a credit card account then there are three conditions first you will get a 15% discount on all your purchases today, second if you are an existing customer and you hold a loyalty card, you get a 10% discount and third if you have a coupon, you can get 20% off today (but it can't be used with the 'new customer' discount). Discount amounts are added, if applicable.

Example 2

[illegible]

BLACK BOX TESTING

Simplifying Decision table

If you are a new customer and you want to open a credit card account then there are three conditions first you will get a 15% discount on all your purchases today, second if you are an existing customer and you hold a loyalty card, you get a 10% discount and third if you have a coupon, you can get 20% off today (but it can't be used with the 'new customer' discount). Discount amounts are added, if applicable.

Example 2:

Conditions	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
Customer	N.C	N.C	E.C	E.C	E.C	E.C
Card Holder	F	F	T	T	F	F
	T	F	T	F	T	F
Coupon						
Actions						
10%			*	*		
15%		*				
20%	*		*		*	
No Discount						NO DISCOUNT

BLACK BOX TESTING

Simplified Decision table

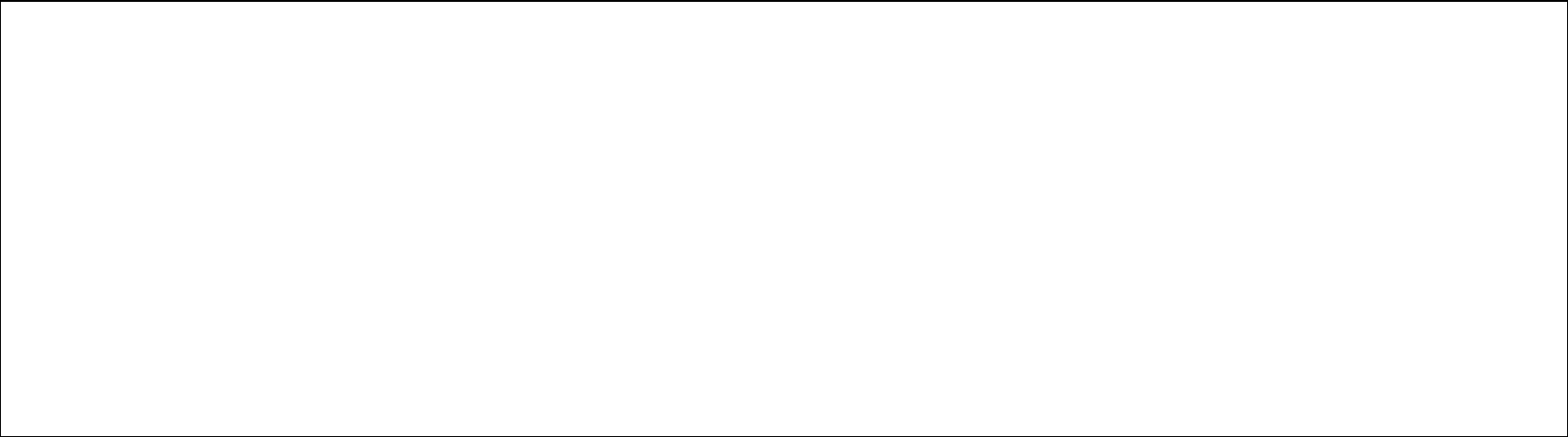
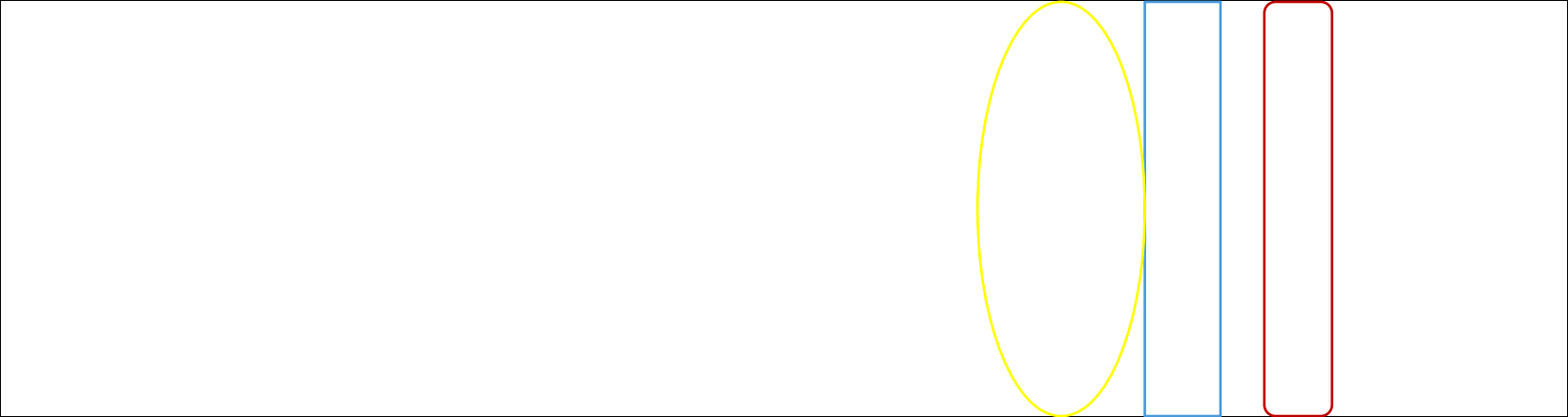
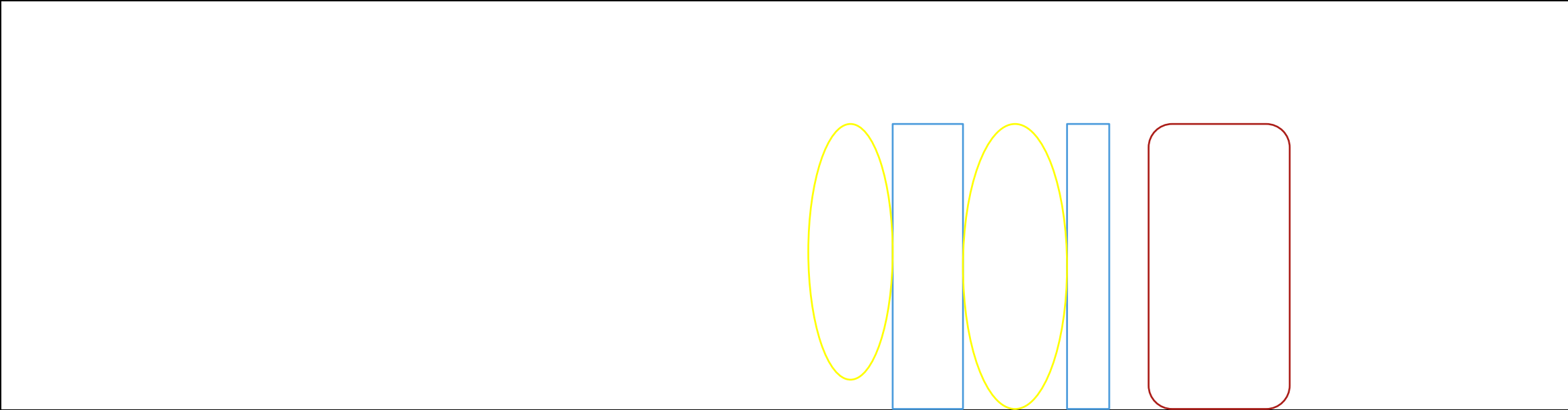
If you are a new customer and you want to open a credit card account then there are three conditions first you will get a 15% discount on all your purchases today, second if you are an existing customer and you hold a loyalty card, you get a 10% discount and third if you have a coupon, you can get 20% off today (but it can't be used with the 'new customer' discount). Discount amounts are added, if applicable.

Example 2:

Conditions	Case 1	Case 2	Case 3	Case 4	Case 5
Customer	-	N.C	E.C	E.C	E.C
Card Holder	F	F	T	T	F
	T	F	T	F	F
Coupon					
Actions					
10%			*	*	
15%		*			
20%	*		*		
No Discount					NO DICOUNT

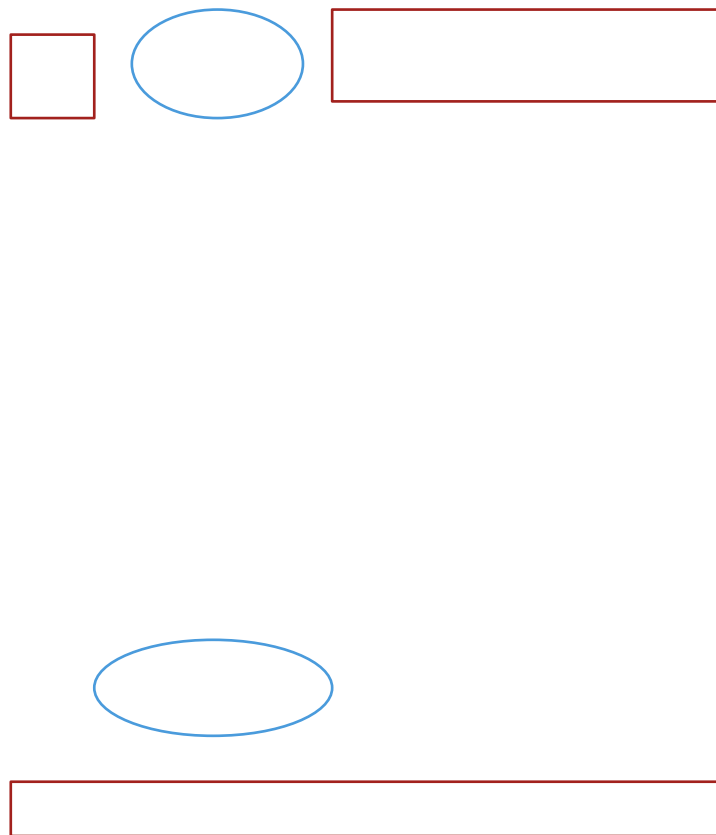
BLACK BOX TESTING

Example 3:



BLACK BOX TESTING

Example 4 :Simplify Decision Table:



BLACK BOX TESTING

iii. DECISION TABLE

To Construct a Decision Table:

- 1) Draw boxes for the top and bottom left quadrants.
 - 2) List the conditions in the top, left quadrant. When possible, phrase the conditions as questions that can be answered with a Y/T for yes/true and an N/F for no.
- ✓ This type of Decision Table is known as a limited entry table.
 - ✓ When a Decision Table requires more than two values for a condition, it is known as an extended entry table.
- 3) List the possible actions in the bottom, left quadrant.
 - 4) Count the possible values for each condition and multiply these together to determine how many unique combinations of conditions are present.

Draw one column in the top and bottom right quadrants for each combination.
- For example, if there are two conditions and the first condition has two possible values while the second has three possible values, draw six ($2 * 3$) columns.
- 5) Enter all possible combinations of values in the columns in the top, right quadrant of the table.
 - 6) For each column (each unique combination of conditions), mark an X in the bottom, right quadrant in the appropriate action row. The X marks the intersection between the required action and each unique combination of condition values.

BLACK BOX TESTING

iii. DECISION TABLE

- Ensure that the Table is Complete

To complete Step 5 above, ensuring that no combinations are missed, follow these steps:

1) Start with the last condition and alternate its possible values across the row. Note how often the pattern repeats itself. For a condition with two possible values, the pattern repeats itself every two columns. If three values are possible, the pattern repeats itself every three columns.

For example, for a table with three conditions each with values Y or N, there are eight ($2 * 2 * 2$) columns. Complete the third (last) row by entering Y/T once across the row followed by one N/F. The pattern repeats itself every two columns.

The third row would look like:

Y N Y N Y N Y N

2) Move to the condition in the row above the last condition. Cover each pattern group with a value for this condition. Note how often the new pattern repeats itself.

For example, complete the second row by entering Y across the row two times followed by two entries of N. This pattern repeats itself every four columns. The second row would look like:

Y Y N N Y Y N N

3) Repeat step 2 until all rows are complete.

BLACK BOX TESTING

iii. DECISION TABLE

- To complete the first row, enter Y across the row four times followed by four entries of N. The first row would look like:
- Y Y Y Y N N N N

- **Flag Error Conditions**

- If a specific combination of conditions is invalid, then define one action to flag the error or invalid condition.

How to Simplify the Decision Table

- If two or more combinations result in the same action, then the table can be simplified. Consider the following example:
- Condition 1 Y Y
- Condition 2 Y Y
- Condition 3 Y N
- Action 1 X X
- The same action occurs whether condition 3 is true or false. As a result, one column can be eliminated from the table as follows:
- Condition 1 Y
- Condition 2 Y
- Condition 3 —
- Action 1 X

•

BLACK BOX TESTING

iii. DECISION TABLE

Decision Table Methodology

1.	Identify Conditions & Values	Find the data attribute each condition tests and all of the attribute's values.
2.	Identify Possible Actions	Determine each independent action to be taken for the decision or policy.
3.	Compute Max Number of Rules	Multiply the number of values for each condition Data attribute by each other.
4.	Enter All Possible Rules	Fill in the values of the condition data attributes in each numbered rule column.
5.	Define Actions for each Rule	For each rule, mark the appropriate actions with an X in the decision table.
6.	Verify the Policy	Review completed decision table with end-users.
7.	Simplify the Table	Eliminate and/or consolidate rules to reduce the Number of columns.

BLACK BOX TESTING

• Advantages of Decision Table testing

- When the system behavior is different for different input and not same for a range of inputs, both equivalent partitioning, and boundary value analysis won't help, but decision table can be used.
- The representation is simple so that it can be easily interpreted and is used for development and business as well.
- This table will help to make effective combinations and can ensure a better coverage for testing
- Any complex business conditions can be easily turned into decision tables
- In a case we are going for 100% coverage typically when the input combinations are low, this technique can ensure the coverage.

Disadvantages of Decision Table testing

- The main disadvantage is that when the number of input increases the table will become more complex.

BLACK BOX TESTING

Why is Decision Table Testing is important?.

- This testing technique becomes important when it is required to test different **combinations**. It also helps in better test coverage for complex business logic.
- This technique can make sure of good coverage of different combinations, and the representation is simple so that it is easy to interpret and use.
- This table can be used as the reference for the requirement and for the functionality development since it is easy to understand and cover all the combinations.
- The **significance of this technique becomes immediately clear as the number of inputs increases. Number of possible Combinations is given by 2^n , where n is the number of Inputs. For $n = 10$, which is very common in the web based testing, having big input forms, the number of combinations will be 1024. Obviously, you cannot test all but you will choose a rich sub-set of the possible combinations using decision based testing technique**

How can the decision table help in software testing?

- A well-created decision table can help to sort out the right response of the system, depending on the input data
- it should include all conditions. It simplifies designing the logic and improves the development and testing of our product.
- With design tables, the information are 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, creating using this technique helps to find edge cases and to identify missing signals in the system

THANKYOU



TO
11TH LECTURE
OF

Software Testing Techniques and Strategies

COURSE CODE :SE-481

BLACK BOX TESTING

iv Cause-Effect Graph-Black Box Software

- ✓ ☐ Cause and effect graph is a test case design technique.
- ✓ It is a black box testing technique.
- ✓ ☐ It is generally used for hardware testing but now adapted to software testing, usually tests external behavior of a system.
- ✓ ☐ It is a testing technique that aids in choosing test cases that logically relate Causes (inputs) to Effects (outputs) to produce test cases.
- ✓ ☐ Cause-Effect Graphing is a technique which starts with set of requirements and determines the minimum possible test cases for maximum test coverage which reduces test execution time and ultimately cost.
- ✓ ☐ The Cause-Effect graph technique restates the requirements specification in terms of logical relationship between the input and output conditions. Since it is logical, it is obvious to use Boolean operators like AND, OR and NOT

BLACK BOX TESTING

iv Cause-Effect Graph

- STEPS
- Step 1: For a module, identify the input conditions (causes) and actions (effect).
- Step 2: Develop a cause-effect graph.
- Step 3: Transform cause-effect graph into a decision table.
- Step 4: Convert decision table rules to test cases. Each column of the decision table represents a test case.

BLACK BOX TESTING

iv Cause-Effect Graph



Basic symbols used in Cause-effect graphs are as under:

Consider each node as having the value 0 or 1 where 0 represents the 'absent state (FALSE)' and 1 represents the 'present state(TRUE)'. Then the identity function states that if c_1 is 1, e_1 is 1 or we can say if c_0 is 0, e_0 is 0.

- The **NOT** function states that if C_1 is 1, e_1 is 0 and vice-versa. Similarly,
- **OR** function states that if C_1 or C_2 or C_3 is 1, e_1 is 1 else e_1 is 0.
- The **AND** function states that if both C_1 and C_2 are 1, e_1 is 1; else e_1 is 0.
- The AND and OR functions are allowed to have any number of input

BLACK BOX TESTING

iv Cause-Effect Graph



In addition to the basic relationships, under the assumption of a, b being cause nodes and e, f being effect nodes, the constraints explained below and shown in Figure

- Exclusive Or (E):** At most one of the variables a and b can be true.
- Inclusive Or (I):** At least one of the variables a and b must be true.
- Only-one (O):** One and only one of the variables a and b must be true.
- Required (R):** In order to variable a to be true, variable b must be true.
- Masking (M):** When the effect e is true, effect f must be false.

BLACK BOX TESTING

iv Cause-Effect Graph

BLACK BOX TESTING

A Simple Cause-Effect Graphing Example 1

The starting point for the Cause-Effect Graph is the requirements document. The requirements describe “what” the system is intended to do.

The requirements can describe real time systems, events, data driven systems, state transition diagrams, object oriented systems, graphical user interface standards, etc.

Any type of logic can be modeled using a Cause-Effect diagram. Each cause (or input) in the requirements is expressed in the cause-effect graph as a condition, which is either true or false. Each effect (or output) is expressed as a condition, which is either true or false.

I have a requirement that says: “If A OR B, then C.”

The following rules hold for this requirement:

- If A is true and B is true, then C is true.
- If A is true and B is false, then C is true.
- If A is false and B is true, then C is true.
- If A is false and B is false, then C is false.

The cause-effect graph that represents this requirement is provided in Figure 1.

The cause-effect graph shows the relationship between the causes and effects.



BLACK BOX TESTING

IV CAUSE-EFFECT GRAPH

- In Figure 1, A, B and C are called nodes. Nodes A and B are the **causes**, while Node C is an **effect**. Each node can have a true or false condition. The lines, called vectors, connect the cause nodes A and B to the effect node C.
- All requirements are translated into nodes and relationships on the cause-effect graph. There are only four possible relationships among nodes, and they are indicated by the following symbols MENTION ABOVE:



BLACK BOX TESTING

iv Cause-Effect Graph

- Example:2
- Let's draw a cause and effect graph based on a situation:

Situation:

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

- The first character must be an “A” or a “B”.
- The second character must be a digit.
- If the first character is an “A” or “B” and the second character is a digit, the file must be updated.
- If the first character is incorrect (not an “A” or “B”), the message X must be printed.
- If the second character is incorrect (not a digit), the message Y must be printed.

BLACK BOX TESTING

iv Cause-Effect Graph

- The causes (Inputs) for this situation are:
- C1 – First character is A
- C2 – First character is B
- C3 – Second character is a digit

The effects (results) for this situation are

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

LET’S START!! First draw the causes and effects as shown below



BLACK BOX TESTING

iv Cause-Effect Graph

- Key – Always go from effect to cause (left to right). That means, to get effect “E” ,what causes should be true. In this example, let’s start with Effect E1. Effect E1 is to update the file. The file is updated when – First character is “A” and second character is a digit – First character is “B” and second character is a digit – First character can either be “A” or “B” and cannot be both. Now let’s put these 3 points in symbolic form:
- For E1 to be true – following are the causes: – C1 and C3 should be true – C2 and C3 should be true – C1 and C2 cannot be true together.

The causes (Inputs) for this situation are:

C1 – First character is A

C2 – First character is B

C3 – Second character is a digit

The effects (results) for this situation are

E1 – Update the file

E2 – Print message “X”

E3 – Print message “Y”

LET’S START!! First draw the causes and effects as shown below

BLACK BOX TESTING

iv Cause-Effect Graph

- Key – Always go from effect to cause (left to right). That means, to get effect “E”, what causes should be true. In this example, let’s start with Effect E1. Effect E1 is to update the file. The file is updated when – First character is “A” and second character is a digit – First character is “B” and second character is a digit – First character can either be “A” or “B” and cannot be both. Now let’s put these 3 points in symbolic form:
- For E1 to be true – following are the causes: – C1 and C3 should be true – C2 and C3 should be true – C1 and C2 cannot be true together..

The causes (Inputs) for this situation are:

C1 – First character is A

C2 – First character is B

C3 – Second character is a digit

The effects (results) for this situation are

E1 – Update the file

E2 – Print message “X”

E3 – Print message “Y”

LET’S START!! First draw the causes and effects as shown below



BLACK BOX TESTING

iv Cause-Effect Graph

- So as per the above diagram, for E1 to be true the condition is $(C1 \vee C2) \wedge C3$. The circle in the middle is just an interpretation of the middle point to make the graph less messy. There is a third condition where C1 and C2 are mutually exclusive. So the final graph for effect E1 to be true is shown below:



Let's move to Effect E2:

E2 states to print message “X”. Message X will be printed when First character is neither A nor B.

Which means Effect E2 will hold true when either C1 OR C2 is invalid. So the graph for Effect E2 is shown as (In blue line)

BLACK BOX TESTING

iv Cause-Effect Graph

Let's move to Effect E2:

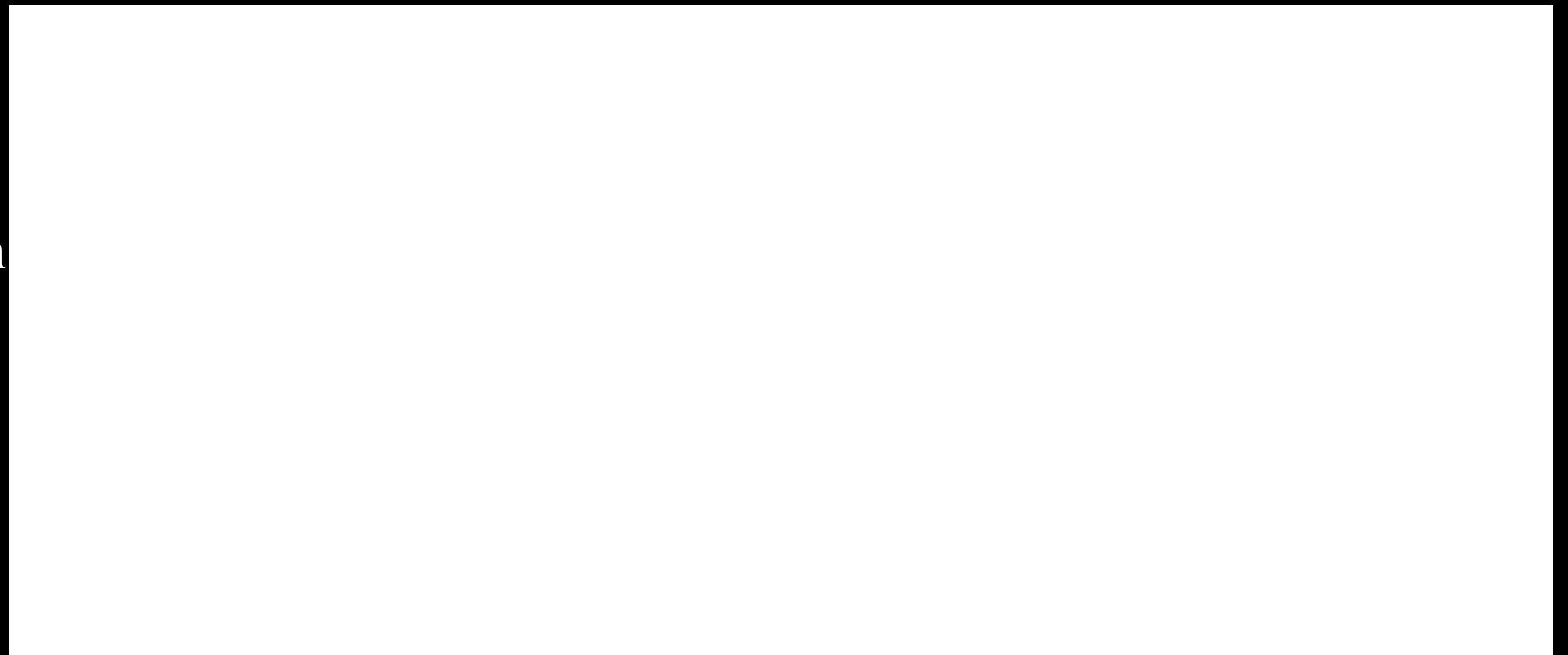
E2 states to print message "X". Message X will be printed when First character is neither A nor B.

Which means Effect E2 will hold true when either C1 OR C2 is invalid. So the graph for Effect E2 is shown as (In blue line)



- **For Effect E3:**

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



- This completes the Cause and Effect graph for the above situation.
- Now let's move to draw the **Decision table based on the above graph.**

BLACK BOX TESTING

iv Cause-Effect Graph

- Writing Decision table based on Cause and Effect graph First write down the Causes and Effects in a single column shown below
- Key is the same. Go from bottom to top which means traverse from effect to cause. Start with Effect E1. For E1 to be true, the condition is: $(C1 \ C2) \ C3$. Here we are representing True as 1 and False as 0

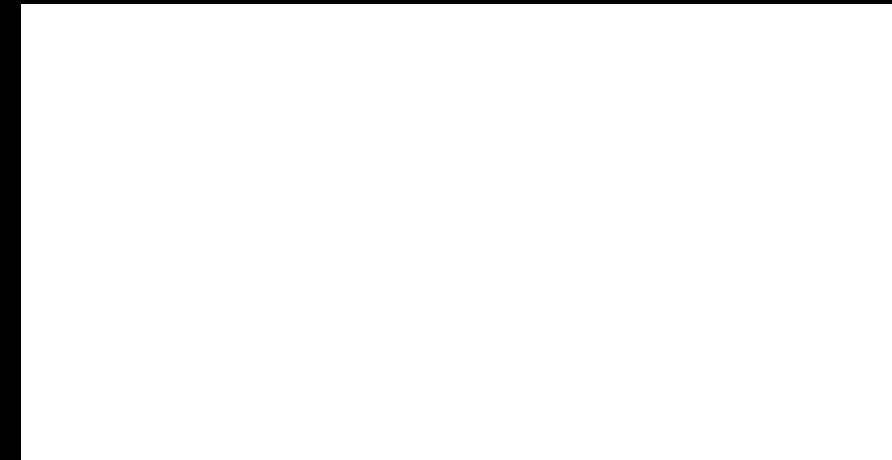
First put Effect E1 as True in the next column as

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

BLACK BOX TESTING

iv Cause-Effect Graph

- For E2 to be True, either C1 or C2 has to be false shown as



For E3 to be true, C3 should be false



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



BLACK BOX TESTING

iv Cause-Effect Graph

- Writing Test cases from the decision table I am writing a sample test case for test case 1 (TC1) and Test Case 2 (TC2).



- In a similar fashion, you can create other test cases. (A test case contains many other attributes like preconditions, test data, severity, priority, build, version, release, environment etc. I assume all these attributes to be included when you write the test cases in actual situation The columns in the decision table are converted into

BLACK BOX TESTING

iv Cause-Effect Graph

- **EXAMPLE: 3** In the following example. Withdrawing money at an Automated Teller Machine (ATM) shall illustrate how to prepare a Cause Effect Graph.

The following conditions must be fulfilled in order to get money from the machine,

- ☐ The bankcard is valid
- ☐ The PIN must be correctly entered
- ☐ The maximum number of PIN Inputs is three
- ☐ There is money in the machine, and in the account

The following actions/Effects are possible at the machine:

- ☐ Reject Card ,
- ☐ Ask for another PIN input ,
- ☐ “Held” the card.
- ☐ Ask for an alternate dollar amount
- ☐ Pay the requested amount of Money
- **Step 1:** For a module, identify the input conditions (causes) and actions (effect).
- **Step 2:** Develop a cause-effect graph.

EXAMPLE: 3

C1 The bankcard is valid

C2 The PIN must be correctly entered

C3 The maximum number of PIN Inputs is three

C4 There is money in the machine, and in the account

The following actions are possible at the machine:

E1 Reject Card , $C1=0 \rightarrow E1=1$

E2 Ask for another PIN input , $(C1=1 \wedge C2=0 \wedge C3=0)$ True cond for
E2

E3 “Held” the card. $(C1=1 \wedge C2=0 \wedge C3=1) \square E3=1$

E4 Ask for an alternate dollar amount

E5 Pay the requested amount of Money
 $(C1=1 \wedge C2=1 \wedge C4=1) \square E5=1$

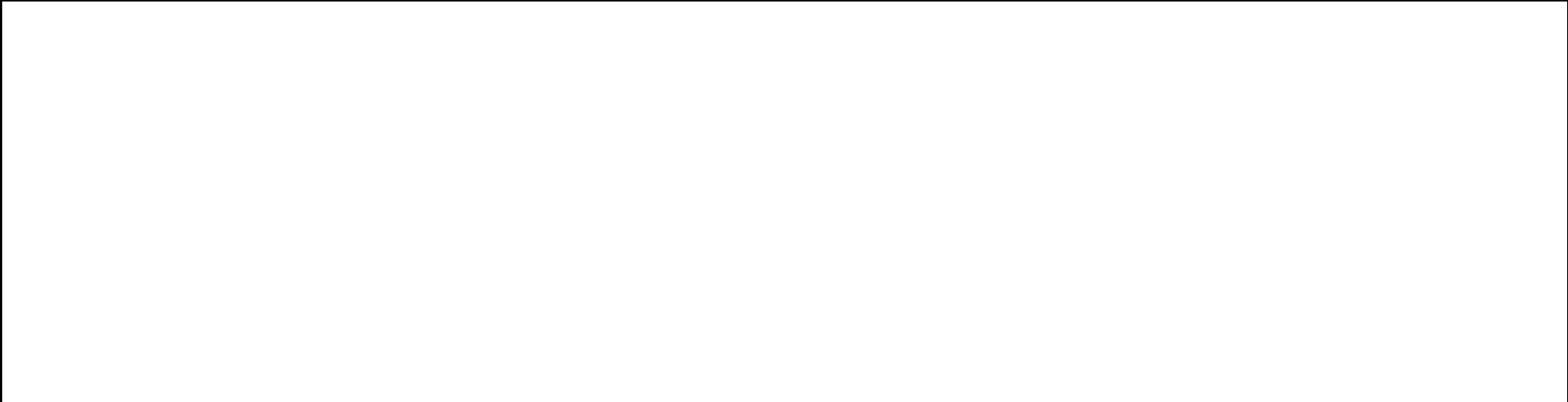
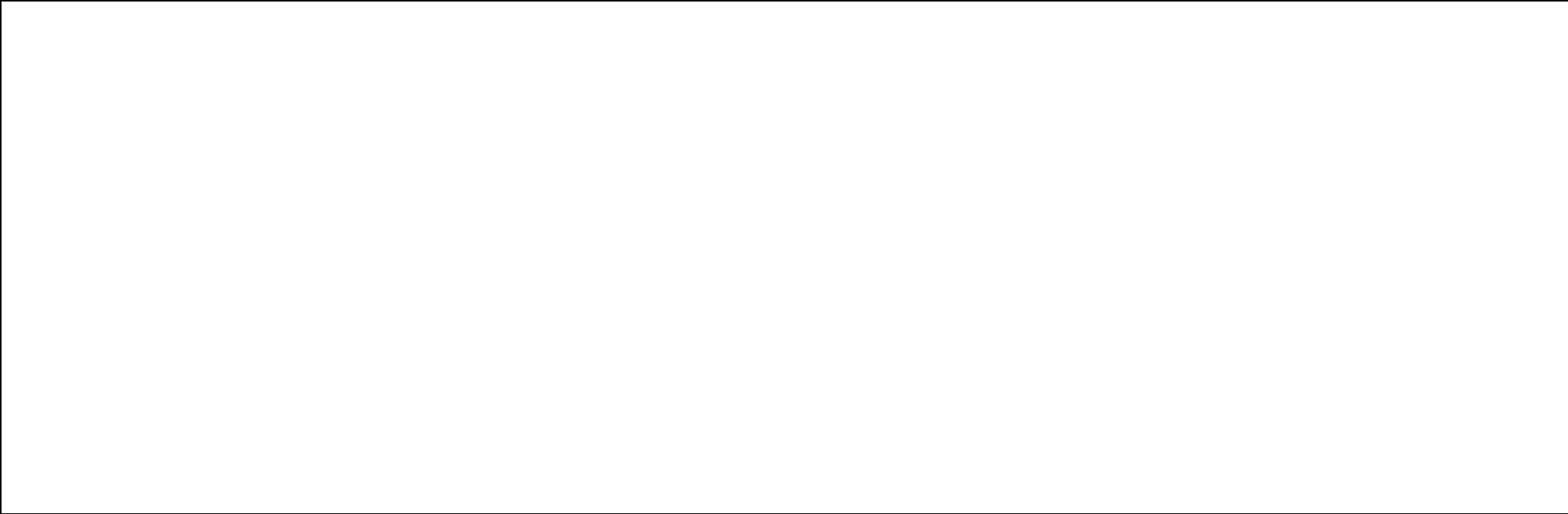
Step 3: Transform cause-effect graph into a decision table

Step 4: Convert decision table rules to test cases. Each column of the decision table represents a test case Write down the test cases according to decision table.

BLACK BOX TESTING

Example:4

iv Cause-Effect Graph



BLACK BOX TESTING

v. State Transition

- State transition testing is a black box testing technique and is used where some aspect of the system can be described in what is called a “finite state machine”. This simply means that the system can be in a (finite) number of different states, and the transitions from one state to another are determined by the rules of the “machine”.
- This is the model on which the system and the tests are based. Any system where you get a different output for the same input, depending on what has happened before, is a finite state system.
- State Transition testing, a black box testing technique, in which outputs are triggered by changes to the input conditions or changes to 'state' of the system. In other words, tests are designed to execute valid and invalid state transitions.

BLACK BOX TESTING

v. State Transition



BLACK BOX TESTING

v. State Transition

- **State Transition Testing Technique** is helpful where you need to test different system transitions.
- **State Transition Testing Example:** If you request to withdraw
- \$100 from a bank ATM, you may be given cash. Later you may make exactly the same request but be refused the money (because your balance is insufficient).
- This later refusal is because the state of your bank account had changed from having sufficient funds to cover the withdrawal to having insufficient funds. The transaction that caused your account to change its state was probably the earlier withdrawal.



BLACK BOX TESTING

v. State Transition

When to use?

- ✓ This can be used when a tester is testing the application for a finite set of input values.
- ✓ When the system under test has a dependency on the events/values in the past.
- ✓ When we have sequence of events that occur and associated conditions that apply to those events
- ✓ It is used for real time systems with various states and transitions involved

BLACK BOX TESTING

v. State Transition

Four Parts of State Transition Model:

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

- **States:** The states that the software may occupy (open/closed or funded/insufficient funds);



- **Transition** from one state to another: The transitions from one state to another (not all transitions are allowed);



- 3) **Events** that causes a transition like closing a file or withdrawing money



4. **Actions** that results from a transition (an error message or being given the cash.)



BLACK BOX TESTING

v. State Transition

Deriving Test cases:

- Understand the various state and transition and mark each valid and invalid state
- Defining a sequence of an event that leads to an allowed test ending state
- Each one of those visited state and traversed transition should be noted down
- Steps 2 and 3 should be repeated until all states have been visited and all transitions traversed
- For test cases to have a good coverage, actual input values and the actual output values have to be generated

State Transition Diagram and State Transition Table

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

BLACK BOX TESTING

v. State Transition

State Transition Diagram and State Transition Table

- There are two main ways to represent or design state transition,
 1. State transition diagram ,
 2. 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.

BLACK BOX TESTING

v. State Transition

Example 1:

A System's transition is represented as shown in the below diagram:



BLACK BOX TESTING

v. State Transition

Example 1:

A System's transition is represented as shown in the below diagram:



The tests are derived from the above state and transition and below are the possible scenarios that need to be tested

Tests	Test 1	Test 2	Test 3
Start State	Off	On	On
Input /Event	Switch ON	Switch Off	Switch off
Output /Action	Light ON	Light Off	Fault
Finish State	ON	OFF	On

Tests	Input ON	Input OFF	Input on
Start State	Off	On	On
Finish State	ON	OFF	Fault

BLACK BOX TESTING

v. State Transition



BLACK BOX TESTING

v. State Transition



BLACK BOX TESTING

BLACK BOX TESTING

v. State Transition

Example 3:

- Let's consider an ATM system function where if the user enters the invalid password three times the account will be locked.
- In this system, if the user enters a valid password in any of the first three attempts the user will be logged in successfully. If the user enters the invalid password in the first or second try, the user will be asked to re-enter the password. And finally, if the user enters incorrect password 3rd time, the account will be blocked.

BLACK BOX TESTING

v. State Transition

1. In this system, if the user enters a valid password in any of the first three attempts the user will be logged in successfully.
2. If the user enters the invalid password in the first or second try, the user will be asked to re-enter the password.
And finally,
3. if the user enters incorrect password 3rd time, the account will be blocked.



BLACK BOX TESTING

v. State Transition

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.

State	Correct PIN	Incorrect PIN	
S1) Start	1 st attempt (s2)	S2	
S2) 1 st attempt	S5	S3	
S3) 2 nd attempt	S5	S4	
S4) 3 rd attempt			
S5) Access Granted	-	-	
S6) Account blocked	-	-	

State	Correct PIN	Incorrect PIN	
S1) Start	1 st attempt (s2)	S2	
S2) 1 st attempt	S5	S3	
S3) 2 nd attempt	S5	S4	
S4) 3 rd attempt	S5	S6	
S5) Access Granted	-	-	
S6) Account blocked	-	-	

BLACK BOX TESTING

v. State Transition

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.

State	Correct PIN	Incorrect PIN	
S1) Start	1 st attempt (s2)	S2	
S2) 1 st attempt	S5	S3	
S3) 2 nd attempt	S5	S4	
S4) 3 rd attempt	S5	S6	
S5) Access Granted	-	-	
S6) Account blocked	-	-	

BLACK BOX TESTING

v. State Transition

Advantages:

- Allows testers to familiarize with the software design and enables them to design tests effectively.
- It also enables testers to cover the unplanned or invalid states.

Disadvantages:

- 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.
- 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.

BLACK BOX TESTING

v. State Transition

Summary:

- ✓ State Transition testing is defined as the testing technique in which changes in input conditions cause's state changes in the Application under Test.
- ✓ State Transition Testing Technique is helpful where you need to test different system transitions.
- ✓ 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.
- ✓ In state transition table all the states are listed on the left side, and the events are described on the top.
- ✓ This main advantage of this testing technique is that it will provide a pictorial or tabular representation of system behavior which will make the tester to cover and understand the system behavior efficiently.
- ✓ The main disadvantage of this testing technique is that we can't rely in this technique every time.

THANKYOU



TO
12TH LECTURE
OF

Software Testing Techniques and Strategies

COURSE CODE :SE-481

WHITE BOX TESTING:

- ✓ If we go by the definition, “White box testing” (also known as **clear**, **glass** box or **structural** testing) is a testing technique which evaluates the code and the internal structure of a program.
- ✓ The term "**white box**" was used because of the see-through box concept. The clear box or white box name symbolizes the ability to see through the software's outer shell (or "box") into its inner workings. Likewise, the "black box" in "**Black Box Testing**" symbolizes not being able to see the inner workings of the software so that only the end-user experience can be tested
- ✓ White box testing involves looking at the structure of the code.
- ✓ When you know the internal structure of a product, tests can be conducted to ensure that the internal operations performed according to the specification. And all internal components have been adequately exercised.

WHITE BOX TESTING:

Definition by ISTQB:

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

Example:

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

WHITE BOX TESTING:

Why we perform WBT?

a. To ensure:

- ✓ That all independent paths within a module have been exercised at least once.
- ✓ All logical decisions verified on their true and false values.
- ✓ All loops executed at their boundaries and within their operational bounds
internal data structures validity

b. To discover the following types of bugs:

- ✓ Logical error tend to creep into our work when we design and implement functions, conditions or controls that are out of the program
- ✓ The design errors due to difference between logical flow of the program and the actual implementation
- ✓ Typographical errors and syntax checking

Steps to Perform WBT

Step #1 –

Understand the functionality of an application through its source code. Which means that a tester must be well versed with the programming language and the other tools as well techniques used to develop the software.

Step #2– Create the tests and execute them.

- When we discuss the concept of testing, “coverage” is considered to be the most important factor.

Performed by:

- White box testing technique is used by both the **developers** as well as **testers**. It helps them to understand which line of code is actually executed and which is not. This may indicate that there is either a missing logic or a typo, which eventually can lead to some negative consequences.

Levels Applicable To

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

- Unit Testing: For testing paths within a unit.
- Integration Testing: For testing paths between units.
- System Testing: For testing paths between subsystems.
- **However, it is mainly applied to Unit Testing/ Integration Testing.**

Steps to Perform WBT

Step #1 –

Understand the functionality of an application through its source code. Which means that a tester must be well versed with the programming language and the other tools as well techniques used to develop the software.

Step #2– Create the tests and execute them.

- When we discuss the concept of testing, “coverage” is considered to be the most important factor

Performed by:

- White box testing technique is used by both the **developers** as well as **testers**. It helps them to understand which line of code is actually executed and which is not. This may indicate that there is either a missing logic or a typo, which eventually can lead to some negative consequences.

Levels Applicable To

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

- Unit Testing: For testing paths within a unit.
- Integration Testing: For testing paths between units.
- System Testing: For testing paths between subsystems.
- **However, it is mainly applied to Unit Testing/Integration Testing.**

ADVANTAGES OF WHITE BOX

TESTING:

- ✓ Testing can be commenced at an earlier stage.
- ✓ One need not wait for the GUI to be available.
- ✓ Testing is more thorough, with the possibility of covering most paths.
- ✓ Code optimization by finding hidden errors.
- ✓ White box tests cases can be easily automated.

DISADVANTAGES OF WHITE BOX TESTING:

- ✓ White box testing can be quite complex and expensive
- ✓ Since tests can be very complex, highly skilled resources are required, with a thorough knowledge of programming and implementation.
- ✓ White-box testing is time-consuming, bigger programming applications take the time to test fully
- ✓ Developers who usually execute white box test cases detest it. The white box testing by developers is not detailed can lead to production errors.
- ✓ White box testing requires professional resources, with a detailed understanding of programming and implementation
- ✓ Test script maintenance can be a burden if the implementation changes too frequently.
- ✓ Since this method of testing is closely tied to the application being tested, tools to cater to every kind of implementation/platform may not be readily available.

WHITE BOX TESTING:



White Box Testing Techniques:

White Box Testing is coverage of the specification in the code:

i. Statement Coverage/Line Coverage –

- ✓ Write enough test cases to execute every statement at least once.

Statement Testing = (Number of Statements Exercised / Total Number of Statements) x 100 %

ii. Branch Coverage /Decision Coverage-

- ✓ This technique is running a series of tests to ensure that all branches are tested at least once.

Branch Testing = (Number of decisions outcomes tested / Total Number of decision Outcomes) x 100 %

iii. Condition Coverage: is also known as Predicate Coverage

- ✓ Condition coverage is seen for Boolean expression, condition coverage ensures whether all the Boolean expressions have been evaluated to both TRUE and FALSE.

iv. Path Coverage -

- ✓ This technique corresponds to testing all possible paths which means that each statement and branch is covered.

Path Coverage = (Number paths exercised / Total Number of paths in the program) x 100 %

WHITE BOX TESTING:

i. LINE COVERAGE OR STATEMENT COVERAGE:

- ✓ In a programming language, a statement is nothing but the line of code or instruction for the computer to understand and act accordingly. A statement becomes an executable statement when it gets compiled and converted into the object code and performs the action when the program is in a running mode.
- ✓ Hence “**Statement Coverage**”, as the name itself suggests, it is the method of validating whether each and every line of the code is executed at least once.
- ✓ Statement coverage is also known as line coverage.
- ✓ Ensure that each code statement is executed once.
 - **Relatively weak criterion**
 - **Weakest white-box criterion**
 - The formula to calculate statement coverage is:

Statement Coverage = (Number of statements exercised/Total number of statements in Program)*100

- Studies in the software industry have shown that black-box testing may actually achieve only 60% to 75% statement coverage, this leaves around 25% to 40% of the statements untested.

WHITE BOX TESTING:

i. LINE COVERAGE OR STATEMENT COVERAGE:

- Consider the below simple pseudocode:

Test case 1

i. INPUT A & B

A=40

ii. C = A + B

B=70

iii. IF C>100

C=110

iv. PRINT “ITS DONE”

- ✓ For **Statement Coverage** –We would only need one test case to check all the lines of the code.

That means:

- ✓ If I consider TestCase_01 to be (A=40 and B=70), then all the lines of code will be executed

Statement Testing = (Number of Statements Exercised / Total Number of Statements) x 100 %

- ✓ Statement Coverage=(4/4)*100=100%

WHITE BOX TESTING:

i. LINE COVERAGE OR STATEMENT COVERAGE:

Now the question arises:

Is that sufficient?

✓ What if I consider my Test case as $A=35$ and $B=45$?

$C=90$

✓ Because Statement coverage will only cover the true side, for the pseudo code, only one test case would NOT be sufficient to test it. As a tester, we have to consider the negative cases as well.

✓ Hence for maximum coverage, we need to consider “**Branch Coverage**”, which will evaluate the “FALSE” conditions.

✓ In the real world, you may add appropriate statements when the condition fails

- i. INPUT A & B
- ii. $C = A + B$
- iii. IF $C > 100$
- iv. PRINT “ITS DONE”

WHITE BOX TESTING:

i. LINE COVERAGE OR STATEMENT COVERAGE:

Example 2:

- i. Print Sum (int a int b)
- ii. Int result = a+b;
- iii. If (result>0)
- iv. Print color (“red”, result);
- v. Else if (result<0)
- vi. Print color (“blue”, result);
- vii. }

Test data 1

contains value of A=3 , B=2

Test data 2

contain value A= -1 ,B=0

Statement Testing = (Number of Statements Exercised / Total Number of Statements) x 100 %

Tet data 1= Statement Coverage=(5/7)*100=71%

Test data 2= Statement Coverage=(7/7)*100=100%

WHITE BOX TESTING:

Control Flow Graph:

A control flow graph (CFG) is the graphical representation of control flow or computation during the execution of programs or applications. Control flow graphs are mostly used in static analysis as well as compiler applications, as they can accurately represent the flow inside of a program unit.

Flow graph notation for a program:

Flow Graph notation for a program is define as several nodes connected through the edges. Below are flow diagrams for statements like if-else, While, until and normal sequence of flow.

Nodes represent entries, exits, decisions and each statement of code.
Edges represent non-branching and branching links between nodes.

WHITE BOX TESTING:

i. LINE COVERAGE OR STATEMENT COVERAGE:

100% Statement Coverage Tip:

- ✓ You need to find out the shortest number of paths following which all the nodes will be covered
- ✓ You can say you have achieved 100% statement coverage if your test cases executes every statement in the code at-least once. ..

WHITE BOX TESTING:

i. LINE COVERAGE OR STATEMENT COVERAGE:

- **Example 3:**

Let's solve a problem on statement coverage so that you know how to solve such question when asked in exam.

How many test case are required to achieve 100% statement coverage?

Switch PC on

Start "outlook"

IF outlook appears THEN

Send an email

Close outlook

WHITE BOX TESTING:

i. LINE COVERAGE OR STATEMENT COVERAGE:

Example 3:

- i. Switch PC on
- ii. Start "outlook"
- iii. IF outlook appears THEN
- iv. Send an email
- v. Close outlook



WHITE BOX TESTING:

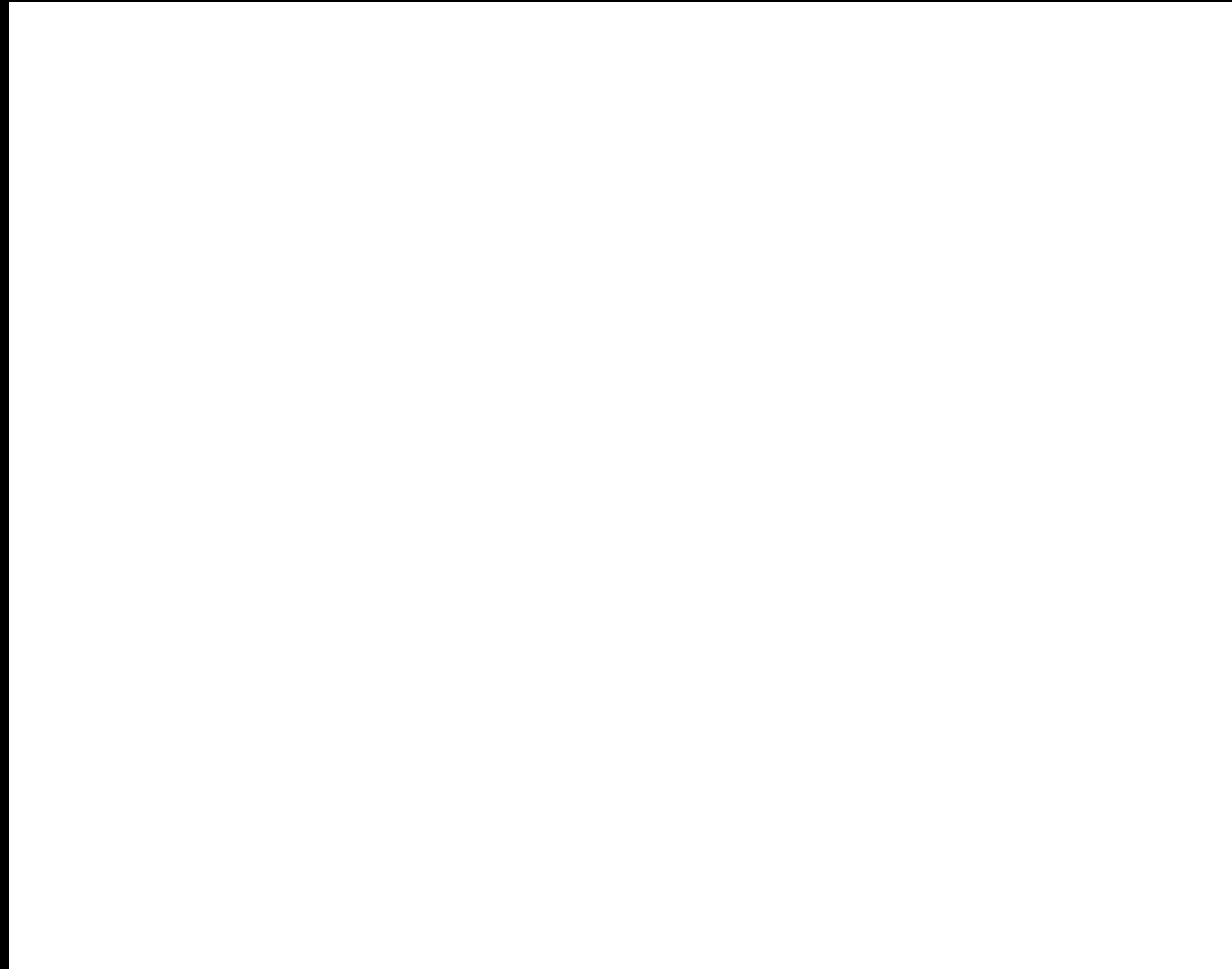
i. LINE COVERAGE OR STATEMENT COVERAGE:

Example 3:

- | | |
|------------------------------|----------|
| i. Switch PC on | Node: |
| A | |
| ii. Start "outlook" | Node :B |
| iii. IF outlook appears THEN | C |
| iv. Send an email | Node: D |
| v. Close outlook | Node : E |

Solution:

- ✓ *As you seen in diagram below in one path we can cover all statements.*
- ✓ *So 1 test case is enough to achieve 100% statement coverage*
- ✓ *Path: A->B->C->D->E*



THANKYOU



TO
13TH LECTURE
OF

Software Testing Techniques and Strategies

COURSE CODE :SE-481

WHITE BOX TESTING:

- ✓ If we go by the definition, “White box testing” (also known as **clear**, **glass** box or **structural** testing) is a testing technique which evaluates the code and the internal structure of a program.
- ✓ The term "**white box**" was used because of the see-through box concept. The clear box or white box name symbolizes the ability to see through the software's outer shell (or "box") into its inner workings. Likewise, the "black box" in "**Black Box Testing**" symbolizes not being able to see the inner workings of the software so that only the end-user experience can be tested
- ✓ White box testing involves looking at the structure of the code.
- ✓ When you know the internal structure of a product, tests can be conducted to ensure that the internal operations performed according to the specification. And all internal components have been adequately exercised.

WHITE BOX TESTING:



White Box Testing Techniques:

White Box Testing is coverage of the specification in the code:

i. Statement Coverage/Line Coverage –

- ✓ Write enough test cases to execute every statement at least once.

Statement Testing = (Number of Statements Exercised / Total Number of Statements) x 100 %

ii. Branch Coverage /Decision Coverage-

- ✓ This technique is running a series of tests to ensure that all branches are tested at least once.

Branch Testing = (Number of decisions outcomes tested / Total Number of decision Outcomes) x 100 %

iii. Condition Coverage: is also known as Predicate Coverage

- ✓ Condition coverage is seen for Boolean expression, condition coverage ensures whether all the Boolean expressions have been evaluated to both TRUE and FALSE.

iv. Path Coverage -

- ✓ This technique corresponds to testing all possible paths which means that each statement and branch is covered.

Path Coverage = (Number paths exercised / Total Number of paths in the program) x 100 %

WHITE BOX TESTING:

Control Flow Graph:

A control flow graph (CFG) is the graphical representation of control flow or computation during the execution of programs or applications. Control flow graphs are mostly used in static analysis as well as compiler applications, as they can accurately represent the flow inside of a program unit.

Flow graph notation for a program:

Flow Graph notation for a program is define as several nodes connected through the edges. Below are flow diagrams for statements like if-else, While, until and normal sequence of flow.

Nodes represent entries, exits, decisions and each statement of code.
Edges represent non-branching and branching links between nodes.

WHITE BOX TESTING:

ii. Branch/Decision Coverage:

- ✓ Branch coverage is also known as Decision coverage. It covers both the true and false conditions unlike statement coverage.
- ✓ We can say we have achieved 100% decision coverage if all the edges in the flow charts are covered by your test case.
- ✓ Branch coverage is a testing method, which aims to ensure that each one of the possible branch from each decision point is executed at least once and thereby ensuring that all reachable code is executed.
- ✓ It helps in validating all the branches in the code making sure that no branch leads to abnormal behavior of the application

To Calculate Branch Coverage

- **Branch Testing = (Number of decisions outcomes tested / Total Number of decision Outcomes) x 100 %**
- **One has to find out the minimum number of paths which will ensure that all the edges are covered**

WHITE BOX TESTING:

ii. Branch/Decision Coverage:

- **Example 1**
- **How many test case are required to achieve 100% decision coverage?**

Switch PC on
Start "outlook"
IF outlook appears THEN
Send an email
Close outlook

WHITE BOX TESTING:

ii. Branch/Decision Coverage:

Example 1:

- | | |
|------------------------------|----------|
| i. Switch PC on | Node: |
| A | |
| ii. Start "outlook" | Node :B |
| iii. IF outlook appears THEN | C |
| iv. Send an email | Node: D |
| v. Close outlook | Node : E |

Solution:

- ✓ *As you seen in diagram below in two paths we can cover all branches.*
- ✓ **We need 2 test cases to achieve 100% decision coverage. 2 paths will ensure covering of all the edges as shown is figure :**
- ✓ ***Path 2: A->B->C->D->E-O***
- ✓ ***Path 1: A->B->C-O***

WHITE BOX TESTING:

Tips

- ✓ Cover all **NODES** for 100% statement coverage. OR
- ✓ Cover all **edges** for 100% Decision/Branch coverage. OR
- ✓ You are often asked minimum number of test cases required to cover 100% statement and branch coverage in exams

WHITE BOX TESTING:

ii. Branch/Decision Coverage:

Example 2: How many test cases are required for 100% decision coverage?

- Read P
Read Q
- IF P+Q > 100 THEN
- Print “Large” ENDIF
- If P > 50 THEN
Print “P Large”
ENDIF

- Branch Testing =

(Number of decisions outcomes tested / Total Number of decision Outcomes) x 100%

WHITE BOX TESTING:

ii. Branch/Decision Coverage:

• Example 2

```
1. Read P
   Read Q
2. IF P+Q > 100 THEN
3. Print "Large"   ENDIF
4. If P > 50 THEN
5. Print "P Large"
   ENDIF
```

Solution:

- Draw Control flow graph
- Identify the path to find out the test cases that cover all the edges
- Write down the test cases

In this case there is no single path which will ensure coverage of all the edges at once.
The aim is to cover all possible true/false decisions.

(1) 1A-2B-E-4F

(2) 1A-2C-3D-E-4G-5H

Hence Branch Coverage is 2.

With 2 paths we can cover all the edges. So 2 test cases are required for 100% decision coverage.

WHITE BOX TESTING:

ii. Branch/Decision Coverage:

- **Branch coverage problems**

- ✓ Short-circuit evaluation means that many predicates might not be evaluated
- ✓ A compound predicate is treated as a single statement. If n clauses, 2^n combinations, but only 2 are tested
- ✓ Only a subset of all entry-exit paths is test.

WHITE BOX TESTING:

iii. CONDITION COVERAGE OR PREDICATE COVERAGE

- ☐ Condition coverage is also known as Predicate Coverage ☐
- Condition coverage is seen for Boolean expression,
- Condition coverage ensures whether all the Boolean expressions have been evaluated to both TRUE and FALSE.
- **Example 1:**

WHITE BOX TESTING:

- **Example 1:**
- **IF (“X && Y”)**
- **In order to suffice valid condition coverage for this pseudo-code following tests will be sufficient.**
- **TEST 1: X=TRUE, Y=TRUE , TEST 2: X=FALSE, Y=TRUE**

WHITE BOX TESTING:



WHITE BOX TESTING:

iv. Path Coverage:

- ✓ A path represents the flow of execution from the start of a method to its exit.
- ✓ A method with N decisions has 2^N possible paths, and if the method contains a loop, it may have an infinite number of paths.
- ✓ Fortunately, you can use a metric called **cyclomatic complexity** to reduce the number of paths you need to test.
- ✓ The objective is to ensure that each condition dependent/independent path through the program is executed at least once
- ✓ As shown in the illustration below, 4 test cases are required for 100% path coverage.

Path Coverage = (Number paths exercised / Total Number of paths in the program) x 100

WHITE BOX TESTING:

iv. Path Coverage:

- **Path Coverage = (Number paths exercised / Total Number of paths in the program) x 100**
- **Same example we consider it in branch coverage**
- **Read P
Read Q**
- **IF P+Q > 100 THEN**
- **Print “Large”**
- **ENDIF**
If P > 50 THEN
Print “P Large”
ENDIF

WHITE BOX TESTING:

iv. Path Coverage:

Path Coverage = (Number paths exercised / Total Number of paths in the program) x 100

```
1. Read P
   Read Q
2. IF P+Q > 100 THEN
3. Print "Large"   ENDIF
4. If P > 50 THEN
5. Print "P Large"
   ENDIF
```

WHITE BOX TESTING:

iv. Path Coverage:

Path Coverage = (Number paths exercised / Total Number of paths in the program) x 100

```

1. Read P
   Read Q
2. IF  $P+Q > 100$  THEN
3. Print "Large"   ENDIF
4. If  $P > 50$  THEN
5. Print "P Large"
   ENDIF

```

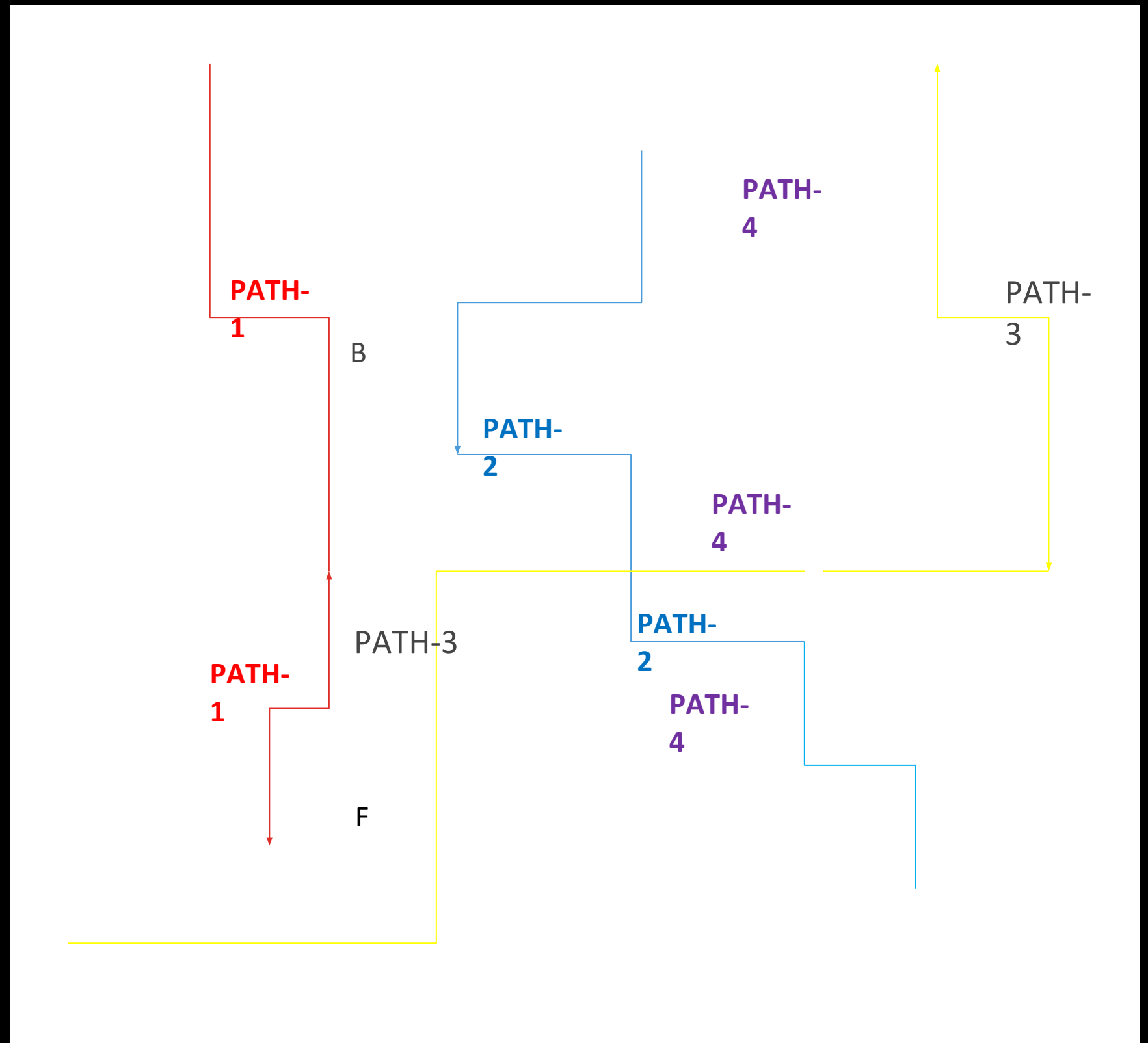
SOLUTION:

1A-2B-E-4F

1A-2B-E-4G-5H

1A-2C-3D-E-4F

1A-2C-3D-E-4G-5H



WHITE BOX TESTING:

BASIS PATH COVERAGE

- i. **Basis path testing** identifies independent paths in source code through which software execution flows.
- ii. The main objective of this **testing** method is to ensure that every **path** is covered and executed.
- iii. It is also designed to reduce the occurrence of redundant tests

METHOD

- i. An independent program path is one that traverses at least one new edge in the flow graph. In program terms, this means exercising one or more new conditions. Both the true and false branches of all conditions must be executed.
- ii. The method analyzes the control flow graph of a program to find a set of **linearly independent paths** of execution.
- iii. The method normally uses **McCabe' Cyclomatic Complexity** to determine the number of linearly independent paths and then generates test cases for each path thus obtained.
- iv. Basis path testing guarantees complete branch coverage (all edges of the control flow graph), but achieves that without covering all possible paths of the control flow graph.
- v. Basis path testing has been widely used and studied.

WHITE BOX TESTING:

Steps for Basis Path testing

The basic steps involved in basis path testing include

- i. Draw a control graph (to determine different program paths)
- ii. Calculate Cyclomatic complexity (metrics to determine the number of independent paths)
- iii. Find a basis set of paths
- iv. Generate test cases to exercise each path

• Benefits of Basis Path testing

- It helps to reduce the redundant tests
- It focuses attention on program logic
- It helps facilitates analytical versus arbitrary case design
- Test cases which exercise basis set will execute every statement in program at least once

• Conclusion:

- Basis path testing helps to determine all faults lying within a piece of code.

WHITE BOX TESTING:

Remember

- i. Branch coverage and Decision coverage are one and the same
- ii. 100% LCSAJ(Linear Code Sequence and Jump.) coverage implies 100% Branch/Decision coverage
LCSAJ stands for Linear Code Sequence and Jump, a white box testing technique to identify the code coverage, which begins at the start of the program or branch and ends at the end of the program or the branch.
- iii. 100% Branch/Decision coverage implies 100% Statement coverage
- iv. 100% Path coverage implies 100% Branch/Decision coverage
- v. 100% Path coverage implies 100% Statement coverage

product / process.

- **Software Metric**

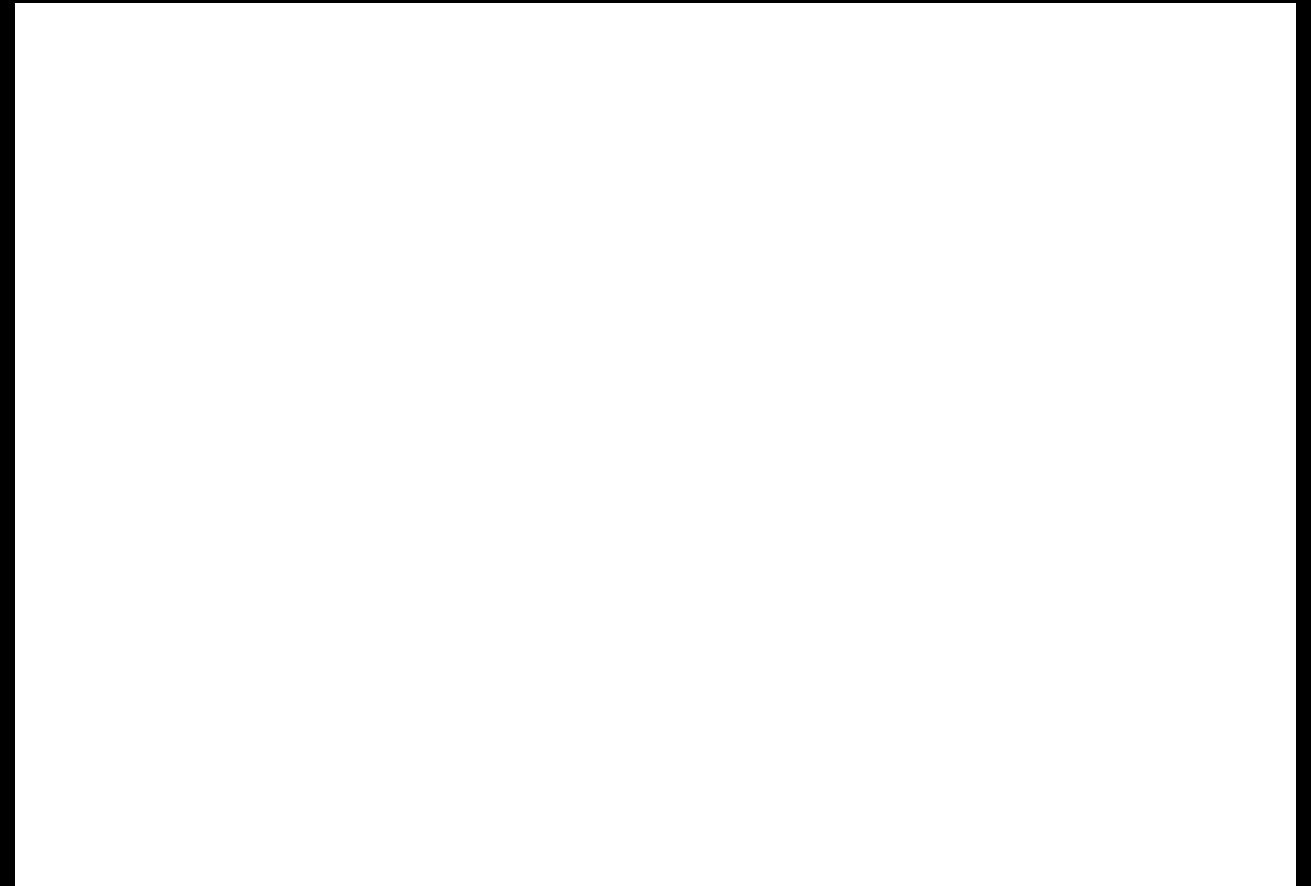
- Software metric is defined as a quantitative measure of an attribute a software system possesses with respect to Cost, Quality, Size and Schedule.
- **Example-**
- Measure - No. of Errors
- Metrics - No. of Errors found per person

What is Cyclomatic Complexity?

- This metric was developed by Thomas J. McCabe in 1976 and it is based on a control flow representation of the program. Control flow depicts a program as a graph which consists of Nodes and Edges.
- In the graph, Nodes represent processing tasks while edges represent control flow between the nodes.
- Cyclomatic complexity is a software metric used to measure the complexity of a program. This metric, measures **independent paths** through program source code.
- Independent path is defined as a path that has at least one edge which has not been traversed before in any other paths.
- Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program.

Uses of Cyclomatic Complexity:

- Cyclomatic Complexity can prove to be very helpful in the following way:
- Helps developers and testers to determine independent path executions
- Developers can assure that all the paths have been tested at least once
- Helps us to focus more on the uncovered paths .Improve code coverage
- Evaluate the risk associated with the application or program
- Using these metrics early in the cycle reduces more risk of the program



Steps to be followed Cyclomatic Complexity :

The following steps should be followed for computing Cyclomatic complexity and test cases design.

- ✓ **Step 1** - Construction of graph with nodes and edges from the code
 - ✓ **Step 2** - Complexity Calculation
 - ✓ **Step 3** - Identification of independent paths Cyclomatic
 - ✓ **Step 4** - Design of Test Cases
-
- Once the basic set is formed, TEST CASES should be written to execute all the paths.

Cyclomatic Complexity

How to Calculate Cyclomatic Complexity: Three methods to calculate the complexity of code:

1. Cyclomatic Complexity's represented by $V(G)$ using Flow Graph to calculate complexity and no. of Independent Path

Mathematically, it is set of independent paths through the graph diagram. The complexity of the program can be defined as –

- **1. $V(G) = E - N + 2$**

Where, E - Number of edges,

N - Number of Nodes

- **2. $V(G) = P + 1$**

Where P = Number of predicate nodes (node that contains condition)

- **3. No. regions in flow graph :(Close cycles)**

WHITE BOX TESTING:

Basis Path testing

Example 1: Computing mathematically,

Cyclomatic Complexity

1. $V(G) = E - N + 2$

2. $V(G) = P + 1$

Predicate Node

- A predicate is A logical function evaluated at a decision point.

In the following example, 1,2 and 3 are predicates (decision) nodes.

3. **No. of Regions: =**

Cyclomatic Complexity

Example 1: Computing mathematically,

1. $V(G) = E - N + 2$

$$V(G) = 9 - 7 + 2 = 4$$

2. $V(G) = P + 1$

$$3 + 1 = 4$$

(Condition nodes are 1, 2 and 3 nodes)

3. **No. of Regions: = 4**

Cyclomatic Complexity is 4

Region 4

Region 3

Region 1

Region 2

WHITE BOX TESTING:

Cyclomatic Complexity

Example 1: Computing mathematically,

1. $V(G) = E - N + 2$

$$V(G) = 9 - 7 + 2 = 4$$

2. $V(G) = P + 1$

3+1=4(Condition nodes are 1,2 and 3 nodes)

3. **No. of Regions: =4**

Step 3: Basis Path Coverage

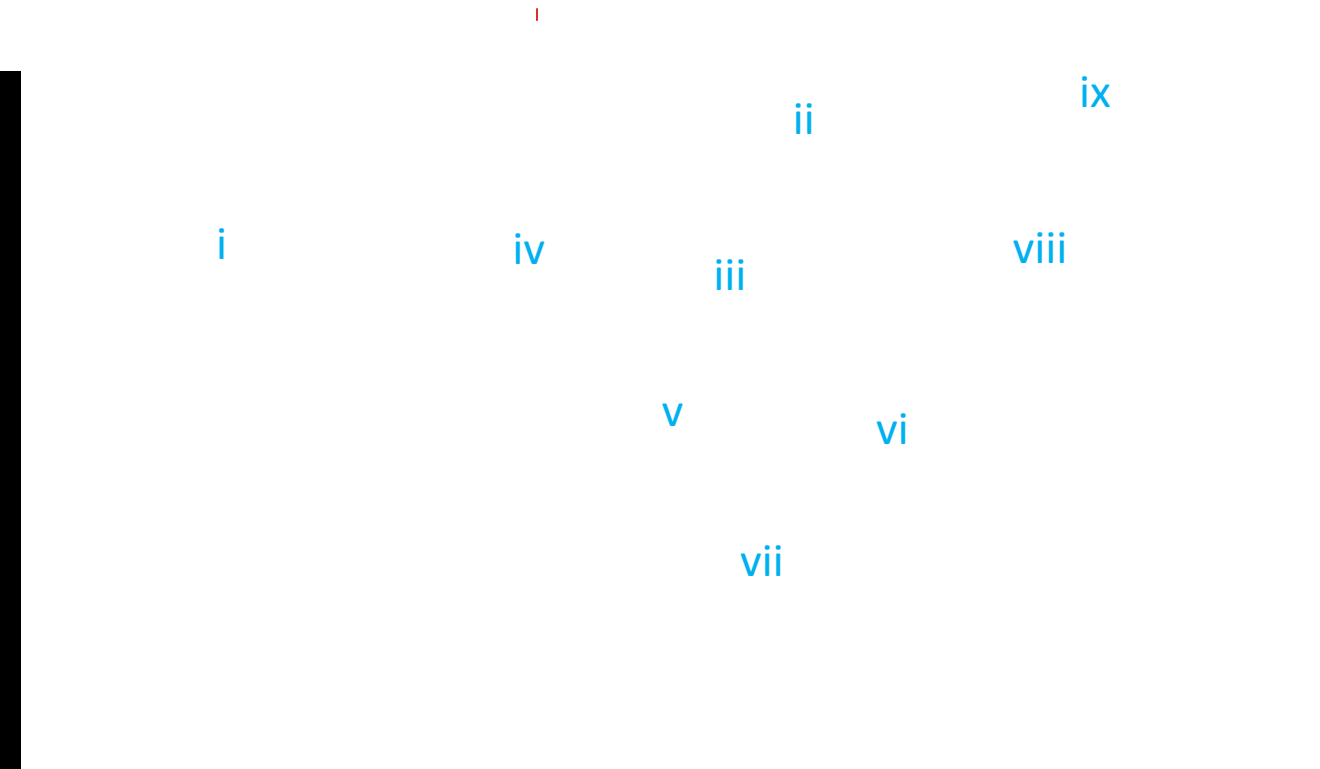
Basis Set - A set of possible execution path of a program are 4:

PATH 1 □ 1, 7

PATH 2 □ 1, 2, 6, 1, 7

PATH 3 □ 1, 2, 3, 4, 5, 2, 6, 1, 7

PATH 4 □ 1, 2, 3, 5, 2, 6, 1, 7



Cyclomatic Complexity



Properties of Cyclomatic complexity:

Following are the properties of Cyclomatic complexity:

- i. $V(G)$ is the maximum number of independent paths in the graph
- ii. $V(G) \geq 1$
- iii. G will have one path if $V(G) = 1$
- iv. Minimize complexity to 10

Complexity Number	Meaning
1-10	Structured and well written code High Testability Cost and Effort is less
10-20	Complex Code Medium Testability Cost and effort is Medium
20-40	Very complex Code Low Testability Cost and Effort are high
>40	Not at all testable Very high Cost and Effort

Control Flow Graph:



WHITE BOX TESTING

Example 2 : Draw the Control Flow Graph Calculate complexity, find out the independent path and design test cases



WHITE BOX TESTING

Basis Path

Example 2 : Draw the Control Flow Graph Calculate complexity, find out the independent path and design test cases

Step 1: Control Flow Graph

WHITE BOX TESTING

Basis Path

Example 2 : Draw the Control Flow Graph Calculate complexity, find out the independent path and design test cases

- **Step 2: Cyclomatic complexity**

- $V(G)=E-N+2$

$$8-7+2=3$$

- $V(G)=P+1$

$$2+1=3$$

- No.Closed region=3

Step 3: Independent Path

PATH 1=1,6,7

PATH 2= 1,2,3,5,6,7

PATH 3 = 1,2,4,5,6,7,

WHITE BOX TESTING

Basis Path

Example 3: Calculate complexity, find out the independent path and design test cases



WHITE BOX TESTING

Basis Path



WHITE BOX TESTING

Basis Path

Example 4: Draw the Control Flow Graph Calculate complexity, find out the independent path and design test cases



WHITE BOX TESTING

Basis Path

Example 4: Draw the Control Flow Graph Calculate complexity, find out the independent path and design test cases

Step 2: Cyclomatic complexity

$$V(G)=E-N+2$$

$$14-12+2=4$$

$$V(G)=P+1$$

$$3+1=4$$

$$\text{No.Closed region}=4$$

Step 3: Independent Path

PATH 1 □ 1,2,3,5,9,11,12

PATH 2 □ 1,2,3,5,6,7,5,9,11,12

PATH 3 □ 1,2,3,4,5,9,11,12

PATH 4 □ 1,2,3,5,9,10,12

WHITE BOX TESTING

Basis Path

Example 4: Calculate complexity, find out the independent path and design test cases

Step 2: Cyclomatic complexity

$$V(G)=E-N+2$$

$$16-14+2=4$$

$$V(G)=P+1$$

$$3+1=4$$

$$\text{No. Closed region}=4$$

Step 3: Independent Path

1, 2, 3, 4, 5, 14

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14

1, 2, 3, 4, 5, 6, 7, 11, 12, 5,

1, 2, 3, 4, 5, 6, 7, 11, 13, 5, ...

WHITE BOX TESTING

Basis Path

Example 5: Calculate complexity, find out the independent path and design test cases

Step 2: Cyclomatic complexity

$$V(G)=E-N+2$$

$$17-13+2=6$$

$$V(G)=P+1$$

$$5+1=6$$

$$\text{No. Closed region}=6$$

Step 3: Independent Path

1, 2, 3, 5, 7, 11 13

1, 2, 3, 5, 7, 11, 2, 3, 5, 7, 11, 13

1, 2, 3, 5, 8, 11, 13

1, 2, 4, 6, 9, 12, 13,

1, 2, 4, 6, 9, 12, 4, 6, 9, 12, 13

1, 2, 4, 6, 10, 12, 13

WHITE BOX TESTING

Basis Path

Example 6: Draw the Control Flow Graph Calculate complexity, find out the independent path .



Conclusion:

Cyclomatic Complexity is software metric useful for structured or White Box Testing. It is mainly used to evaluate complexity of a program. If the decision points are more, then complexity of the program is more. If program has high complexity number, then probability of error is high with increased time for maintenance and trouble shoot

-

THANKYOU



TO
14TH LECTURE
OF

Software Testing Techniques and Strategies

COURSE CODE :SE-481

WHITE BOX TESTING:

- Cost of Quality:
- **DEFINITION:**
- Cost of Quality (COQ) is a measure that quantifies the **cost of control/conformance** and the cost of failure of **control/non-conformance**. In other words, it sums up the costs related to prevention and detection of defects and the costs due to occurrences of defects.
- Definition by ISTQB: cost of quality: The total costs incurred on quality activities and issues and often split into prevention costs, appraisal costs, internal failure costs and external failure costs.
- Definition by QAI: Money spent beyond expected production costs (labor, materials, and equipment) to ensure that the product the customer receives is a quality (defect free) product. The Cost of Quality includes prevention, appraisal, and correction or repair costs.

EXPLANATION:

- **Cost of Control (Also known as Cost of Conformance)**

- i. **Prevention Cost**

- ✓ The cost arises from efforts to prevent defects.
 - ✓ Example: Quality Assurance costs o Appraisal Cost
 - ✓ The cost arises from efforts to detect defects.
Example: Quality Control costs

- **Cost of Failure of Control (Also known as Cost of Non-Conformance)**

- i. **Internal Failure Cost**

- ✓ The cost arises from defects identified internally and efforts to correct them.
 - ✓ Example: Cost of Rework (Fixing of internal defects and retesting)

- ii. **External Failure Cost**

- ✓ The cost arises from defects identified by the client or end-users and efforts to correct them.
 - ✓ Example: Cost of Rework (Fixing of external defects and retesting) and any other costs due to external defects (Product service/liability/recall, etc):

WHITE BOX TESTING:

FORMULA :

Cost of Quality (COQ) = Cost of Control + Cost of Failure of Control

where

Cost of Control = Prevention Cost + Appraisal Cost

and

Cost of Failure of Control = Internal Failure Cost + External Failure Cost



Cost of production=Cost of development+ Cost of Quality

{ (requirement+design+coding) +(Cost of Control +Cost of failure control)}
{ (requirement+design+coding) +(Prevention Cost + Appraisal Cost +Internal Failure Cost + External Failure Cost

WHITE BOX TESTING:

Conclusion:

- In its simplest form, COQ can be calculated in terms of effort (hours/days).
- A better approach will be to calculate COQ in terms of money (converting the effort into money and adding any other tangible costs like test environment setup).
- The best approach will be to calculate COQ as a percentage of total cost. This allows for comparison of COQ across projects or companies.
- To ensure impartiality, it is advised that the Cost of Quality of a project/product be calculated and reported by a person external to the core project/product team (Say, someone from the Accounts Department).
- It is desirable to keep the Cost of Quality as low as possible. However, this requires a fine balancing of costs between Cost of Control and Cost of Failure of Control. In general, a higher Cost of Control results in a lower Cost of Failure of Control. But, the law of diminishing returns holds true here as well

Problem reporting, tracking and analyses

SOFTWARE DEFECTS MANAGEMENT

Defect or Bug

- When a tester executes the test cases, he might come across the test result which is contradictory to expected result. This variation in the test result is referred as a defect. These defects or variation are referred by different names in a different organization like issues, problem, bug or incidents.

Debugging:

- Identify and remove errors from program (code) is known as debugging.

Bug reports

- Reports detailing bugs in software are known as bug reports.

Why Bugs Occur in the Software?

Plentiful studies have been on very little projects to extremely large ones and the analysis shows various causes of bugs in the software, few of them are listed below:

- ✓ One of the extreme causes is the specification. In several cases, specifications are the largest producer of bugs. Either specifications are not written, specifications are not thorough enough, constantly changing or not communicated well to the development team.
- ✓ Another bigger reason is that software is always created by human beings. They know numerous things but are not expert and might make mistakes.
- ✓ Further, there are deadlines to deliver the project on time. So increasing pressure and workload conduct in no time to check, compromise on quality and incomplete systems. So this leads to occurrence of Bugs.

Problem reporting, tracking and analysis

SOFTWARE DEFECTS MANAGEMENT

Types of Program Error

There are three basic categories of program errors:

- i. Syntax Errors
 - ii. Run-time Errors
 - iii. Logic Errors
- In the Syntax and Run-time errors, when an error occurs, the computer displays an 'Error Message', which describes the error, and its cause. Unfortunately, error messages are often difficult to interpret, and are sometimes misleading. In the **logical errors**, the program will not show an error message.

Syntax Errors

error is caused by the failure of the programmer to use the correct grammatical rules of the language.

- Syntax errors are detected, and displayed, by the compiler as it attempts to translate your program, i.e the Source code into the Object code. If a program has a syntax error it cannot be translated, and the program will not be executed.
- The compiler tries to highlight syntax errors where there seems to be a problem, however, it is not perfect and sometimes the compiler will indicate the next line of code as having the problem rather than the line of code where the problem actually exists.

Run-Time Errors

- Run-time errors are detected by the computer and displayed during execution of a program.
- They will halt the program when they occur but they often do not show up for some time.
- A run illegal operation, eg:-time error occurs when the user directs the computer to perform an
 - Dividing a number by zero
 - Assigning a variable to the wrong type of variable
 - Using a variable in a program before assigning a value to it.
- When a run-time error occurs, the computer stops executing your program, and displays a diagnostic message that indicates the line where the error occurred.

Logic Errors

- These are the hardest errors to find as they do not halt the program. They arise from faulty thinking on behalf of the programmer. They can be very troublesome.
- These are mistakes in a program's logic.
- Programs with logic errors will often compile, execute, and output results. However, at least some of the time the output will be incorrect. Error messages will generally not appear if a logic error occurs, this makes logic errors very difficult to locate and correct.

SOFTWARE DEFECTS MANAGEMENT

User Interface or Cosmetic Bug:

These types of errors are the result of incorrect formatting.

The software functionality is least affected by these kind of errors.

For example: Spelling mistake (typographical error),

incorrect color in any specific field,

incorrect width / height / length of any field etc.

Cosmetic errors with the logo of a company or incorrect name of the company in the software could lead to huge monetary losses.

Such errors least affect the software functionality but they could lead to copyright / patent / legal issues

Defects/Bug Severity and Priority

There are two key things in defects/bug of the software testing.

1. Severity
2. Priority

SOFTWARE DEFECTS MANAGEMENT

Defect Severity:

Severity of a defect is related to how severe a bug is. Or It is defined as the degree of impact that a defect has on the functionality of the product/ system. For example: If an application or web page crashes when a remote link is clicked, in this case clicking the remote link by an user is rare but the impact of application crashing is severe. So the severity is high but priority is low. Defect severity can be categorized into four types as follows:

- ❑ **Critical:** The defect that results in the termination of the complete system or one or more component of the system and causes extensive corruption of the data. The failed function is unusable and there is no acceptable alternative method to achieve the required results then the severity will be stated as critical.
- ❑ **Major:** The defect that results in the termination of the complete system or one or more component of the system and causes extensive corruption of the data. The failed function is unusable but there exists an acceptable alternative method to achieve the required results then the severity will be stated as major.
- ❑ **Moderate:** The defect that does not result in the termination, but causes the system to produce incorrect, incomplete or inconsistent results then the severity will be stated as moderate.
- ❑ **Cosmetic/ Low:** The defect that is related to the enhancement of the system where the changes are related to the look and feel of the application then the severity is stated as cosmetic.

SOFTWARE DEFECTS MANAGEMENT

Defect Priority:

- ✓ Priority of a defect is related to how quickly a bug should be fixed and deployed to live servers

Or

- ✓ Priority defines the order in which we should resolve a defect. Should we fix it now, or can it wait?
- ✓ This priority status is set by the tester to the developer mentioning the time frame to fix the defect.
- ✓ If high priority is mentioned then the developer has to fix it at the earliest.
- ✓ The priority status is set based on the customer requirements.
- ✓ For example: If the company name is misspelled in the home page of the website, then the priority is high and severity is low but the developer has to fix it.
 - Priority can be of following types:

SOFTWARE DEFECTS MANAGEMENT

Defect Priority: Priority can be of following types:

Low: The defect is an irritant which should be repaired, but repair can be deferred until after more serious defect has been fixed.

Medium: The defect should be resolved in the normal course of development activities. It can wait until a new build or version is created

High: The defect must be resolved as soon as possible because the defect is affecting the application or the product severely. The system cannot be used until the repair has been done.

Few very important scenarios related to the severity and priority which are asked during the interview:

High Priority & High Severity: An error which occurs on the basic functionality of the application and will not allow the user to use the system. (Eg. A site maintaining the student details, on saving record if it, doesn't allow to save the record then this is high priority and high severity bug)

High Priority & Low Severity: The spelling mistakes that happens on the cover page or heading or title of an application.

High Severity & Low Priority: An error which occurs on the functionality of the application (for which there is no workaround) and will not allow the user to use the system but on click of link which is rarely used by the end user.

Low Priority and Low Severity: Any cosmetic or spelling issues which is within a paragraph or in the report (Not on cover page, heading, title).

DEFECT/BUG LIFE CYCLE OR A BUG LIFE CYCLE IN SOFTWARE TESTING

- ✓ Defect life cycle is a cycle which a defect goes through during its lifetime.
- ✓ It starts when defect is found and ends when a defect is closed, after ensuring it's not reproduced.
- ✓ Defect life cycle is related to the bug found during testing.

DEFECT/BUG LIFE CYCLE OR A BUG LIFECYCLE IN SOFTWARE TESTING

New: When a defect is logged and posted for the first time. Its state is given as new.

Assigned: After the tester has posted the bug, the lead of the tester approves that the bug is genuine and he assigns the bug to corresponding developer and the developer team. Its state given as assigned.

Open: At this state the developer has started analyzing and working on the defect fix.

Fixed: When developer makes necessary code changes and verifies the changes then he/she can make bug status as 'Fixed' and the bug is passed to testing team.

Pending retest: After fixing the defect the developer has given that particular code for retesting to the tester. Here the testing is pending on the testers end. Hence its status is pending retest.

Retest: At this stage the tester do the retesting of the changed code which developer has given to him to check whether the defect got fixed or not.

Verified: The tester tests the bug again after it got fixed by the developer. If the bug is not present in the software, he approves that the bug is fixed and changes the status to "verified".

DEFECT/BUG LIFE CYCLE OR A BUG LIFECYCLE IN SOFTWARE TESTING

the tester changes the status to “reopened”. The bug goes through the life cycle once again.

Closed: Once the bug is fixed, it is tested by the tester. If the tester feels that the bug no longer exists in the software, he changes the status of the bug to “closed”. This state means that the bug is fixed, tested and approved.

Duplicate: If the bug is repeated twice or the two bugs mention the same concept of the bug, then one bug status is changed to “duplicate”.

Rejected: If the developer feels that the bug is not genuine, he rejects the bug. Then the state of the bug is changed to “rejected”.

Deferred: The bug, changed to deferred state means the bug is expected to be fixed in next releases. The reasons for changing the bug to this state have many factors. Some of them are priority of the bug may be low, lack of time for the release or the bug may not have major effect on the software.

Not a bug: The state given as “Not a bug” if there is no change in the functionality of the Application. For an example: If customer asks for some change in the look and field of the application like change of color of some text then it is not a bug but just some change in the looks of the application.

DEFECT/BUG LIFE CYCLE OR A BUG LIFECYCLE IN SOFTWARE TESTING

1. Bug ID – Every bug or defect has it's unique identification number
2. Bug Description – This includes the abstract of the issue.
3. Product Version – This includes the product version of the application in which the defect is found.
4. Detail Steps – This includes the detailed steps of the issue with the screenshots attached so that developers can recreate it.
5. Date Raised – This includes the Date when the bug is reported
6. Reported By – This includes the details of the tester who reported the bug like Name and ID
7. Status – This field includes the Status of the defect like New, Assigned, Open, Retest, Verification, Closed, Failed, Deferred, etc.
8. Fixed by – This field includes the details of the developer who fixed it like Name and ID
9. Date Closed – This includes the Date when the bug is closed
0. Severity – Based on the severity (Critical, Major or Minor) it tells us about impact of the defect or bug in the software application
1. Priority – Based on the Priority set (High/Medium/Low) the order of fixing the defect can be made.

[illegible]

Defect Metrics

Important Defect Metrics

- ✓ Back the above scenario. The developer and test teams have reviews the defects reported. Here is the result of that discussion

Q. How to measure and evaluate the quality of the test execution?

- ✓ This is a question which every Test Manager wants to know.
- ✓ There are 2 parameters which you can consider as following

Defect Metrics

Important Defect Metrics

- ✓ Back the above scenario. The developer and test teams have reviews the defects reported. Here is the result of that discussion

Q. How to measure and evaluate the quality of the test execution?

- ✓ This is a question which every Test Manager wants to know.
- ✓ There are 2 parameters which you can consider as following

Defect Metrics

In the given scenario, you can calculate the **defection rejection ratio (DRR)** is $20/84 = 0.238$ (23.8 %).

Another example, supposed the Bank website has total 64 defects, but your testing team only detect 44 defects i.e. they missed 20 defects.

Therefore, you can calculate the defect leakage ratio (DLR) is $20/64 = 0.312$ (31.2 %).

Defect Metrics

Conclusion

- The smaller value of DRR and DLR is, the better quality of test execution is.

Q. What is the ratio range which is acceptable?

This range could be defined and accepted base in the project target or you may refer the metrics of similar projects.

- In this project, the recommended value of acceptable ratio is 5 ~ 10%. It means the quality of test execution is low. You should find countermeasure to reduce these ratios such as
 - ❖ Improve the testing skills of member.
 - ❖ Spend more time for testing execution, especially for reviewing the test execution results.

Defect Density

Defect Density is the number of defects confirmed in software/module during a specific period of operation or development divided by the size of the software/module. It enables the one, to decide if a piece of software is ready to be released.

Defect density is counted per thousand lines of code also known as KLOC. Formula to measure Defect Density:

Defect Density = Defect count/ size of the release

Example 1:

Suppose 10 bugs are found in 1 KLOC Therefore DD is 10/KLOC

For better understanding, consider the following example.

Example 2:

Suppose you have a software product which has been integrated with the 4 modules and you found the following bugs in each of the modules.

Module 1 = 20 bugs

Module 2 = 30 bugs

Module 3 = 50 bugs

Module 4 = 60 bugs

And the total line of code for each module is

Defect Density

Example 2:

And the total line of code for each module is

Module 1 = 1200 LOC

Module 2 = 3023 LOC

Module 3 = 5034 LOC

Module 4 = 6032 LOC

Suppose you have a software product which has been integrated with the 4 modules and you found the following bugs in each of the modules.

Module 1 = 20 bugs

Module 2 = 30 bugs

Module 3 = 50 bugs

Module 4 = 60 bugs

Total bugs = 20+30+50+60 = 160

Total Size of module = 15289 LOC

Then, we calculate defect density as : **Defect Density = Defect count/ size of the release**

$\text{Defect Density} = 160/15289 = 0.01046504 \text{ defects/loc} = 10.465 \text{ defects/Kloc}$

We can use defect density to calculate the following:

1. We can predict the remaining defect in the software product by using the defect density.
2. We can determine whether our testing is sufficient before the release.
3. We can ensure a database of standard defect densities.

Difference between Defect Priority and Defect Severity

Defect Priority	Defect Severity
Priority defines the order in which the developer should resolve a defect	It is defined as the degree of impact that a defect has on the operation of the product
Priority is categorized into three types	Severity are categorized into four types
1. Low	1. Critical
2. Medium	2. Major
3. High	3. Moderate 4. Cosmetic
Priority is associated with scheduling	Severity is associated with functionality or standards
Priority indicates how soon the bug should be fixed	Severity indicates the seriousness of the defect on the product functionality
Priority of defects is decided in consultation with the manager/client	QA engineer determines the severity level of the defect
Priority is driven by business value	Severity is driven by functionality
Its value is subjective and can change over a period of time depending on the change in the project situation	Its value is objective and less likely to change
High priority and low severity status indicates, defect have to be fixed on immediate bases but does not affect the application	High severity and low priority status indicates defect have to be fixed but not on immediate bases
Priority status is based on the customer requirements	Severity status is based on the technical aspect of the product
During UAT the development team fix defects based on priority	During SIT, the development team will fix defects based on the severity and then priority

THANKYOU

