Unit – 4- Test design Techniques

-Identifying test conditions & designing test cases
-Categories of test design techniques
-Specification based or black box techniques
-Structure based or white box techniques
-Experienced based techniques

Identifying test conditions & designing test cases

- **What is Test analysis? or How to identify the test conditions?**

Test analysis: identifying test conditions

- Test analysis is the process of looking at something that can be used to derive test information. This basis for the tests is called the **test basis**.

- **Test basis** is the information we need in order to start the test analysis and create our own test cases. Basically it's a **documentation on which test cases are based**, such as requirements, design specifications, product risk analysis, architecture and interfaces.

- We can use **the test basis documents to understand what the system should do once built**. The test basis includes whatever the tests are based on.

- Sometimes **tests can be based on experienced user's knowledge of the system** which may not be documented.

- From testing perspective we look at the test basis in order to see what could be tested. These are the test conditions.

- A **test condition** is simply something that we could test. While **identifying the test conditions we want to identify as many conditions as we can** and then we select about which one to take forward and combine into test cases. We could call them **test possibilities.**

- **What is traceability in Software testing?**

- Test conditions should be able to be linked back to their sources in the test basis, this is known as **traceability.**

- Traceability can be **horizontal** through all the test documentation for a given test level (e.g. from test conditions through test cases to test scripts) or

- it can be **vertical** through the layers of development documentation (e.g. from requirements to system).

Why is traceability important? So, let's have a look on the following examples:

- The requirements for a given function or feature have changed. **Some of the fields now have different ranges that can be entered. Which tests were looking at those boundaries?** They now need to be changed. **How many tests will actually be affected by this change in the requirements?** These questions can be answered easily if the requirements can easily be traced to the tests.

Example2:

- A set of tests that has run OK in the past has now started creating serious problems.
  - **What functionality do these tests actually exercise?**
- Traceability between **the tests and the requirement** being tested enables the functions or features affected to be identified more easily.

  Before delivering a new release, we want to know whether or not we have tested all of the specified requirements in the requirements specification. We have the list of the tests that have passed – **Was every requirement tested?**

Test design: specifying test cases

- **What is Test design? or How to specify test cases?**

  Test design: specifying test cases

- Basically test design is the act of creating and writing test suites for testing a software.

- **Test analysis and identifying test conditions** gives us a **generic idea for testing** which covers quite a large range of possibilities. But when we come to make **a test case we need to be very specific.**

- In fact now **we need the exact and detailed specific input**. But just **having some values to input to the system is not a test,** if you don't know what the system is supposed to do with the inputs, you will not be able to tell that whether your test has passed or failed.

- Test cases can be documented as described in the **IEEE 829 Standard for Test Documentation.**

- One of the **most important aspects of a test is that it checks that the system does what it is supposed to do.**

- Copeland says ‘At its core, testing is the process of comparing **“what is” with “what ought to be” .**

- If we simply put in some inputs and think that was fun, I guess the system is probably OK because it didn’t crash, but are we actually testing it? We don’t think so.

- You have observed that the system does what the system does but this is not a test. Boris Beizer refers to this as **'kiddie testing'** [Beizer, 1990].

- We may not know what the right answer is in detail every time, and we can still get some benefit from this approach at times, but it isn't really testing. **To know what the system *should* do, we need to have a source of information about the correct behavior of the system –** this is called an **'oracle'** or **a test oracle.**

What is Test implementation? or How to specifying test procedures or scripts?

- The document that describes **the steps to be taken in running a set of tests and specifies the executable order of the tests** is called a **test procedure** in IEEE 829, and is also known as a **test script.**

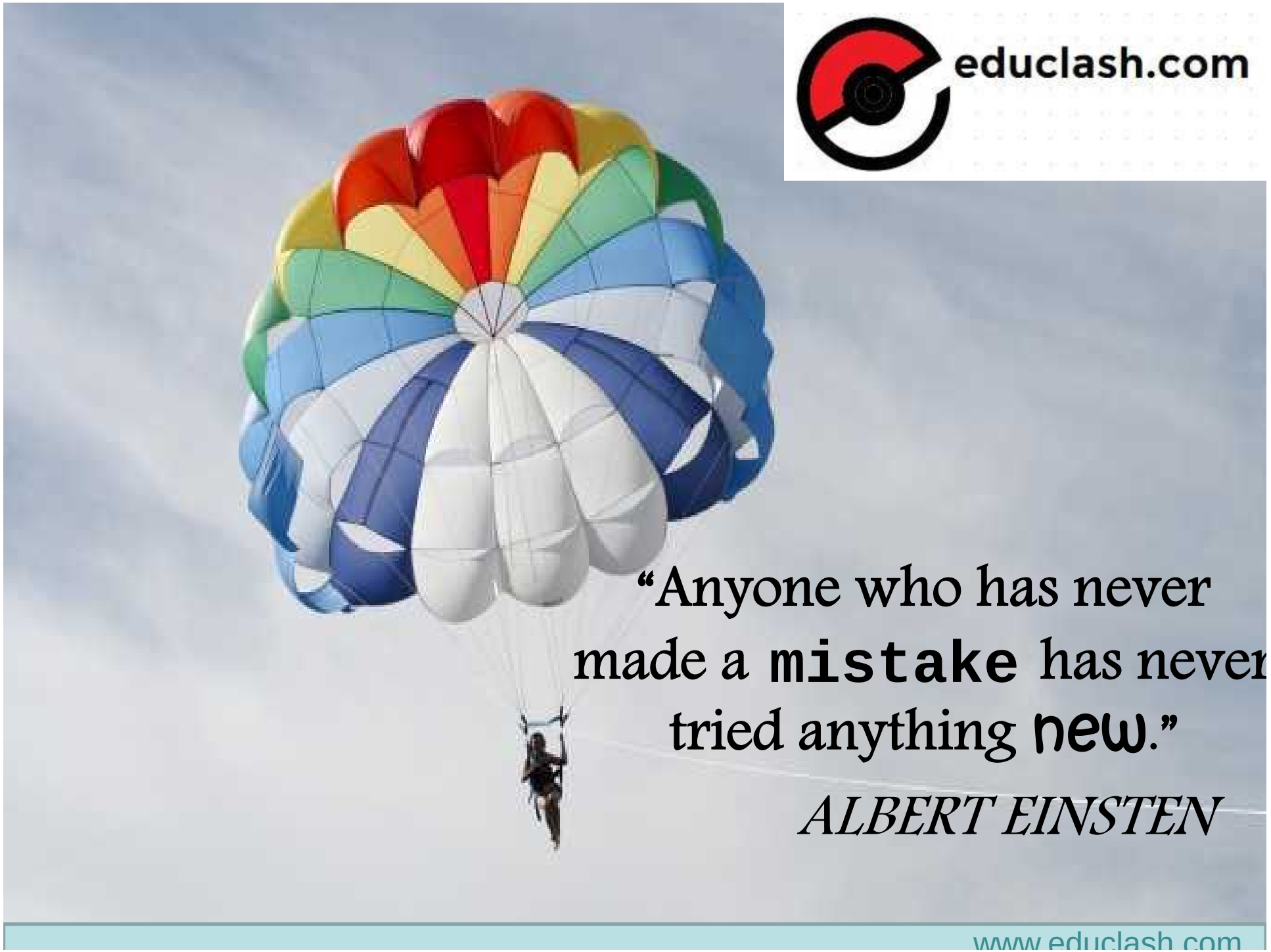
- When **Test Procedure Specification is prepared** then it is implemented and is called **Test implementation**.

- **Test script is also used to describe the instructions to a test execution tool.** An automation script is written in a programming language that the tool can understand. (This is an automated test procedure.).

- The **tests that are intended to be run manually rather than using a test execution tool** can be called as **manual test script**.

- The **test procedures, or test scripts, are then formed into a test execution schedule** that specifies which procedures are to be run first – a kind of **superscript.**

- Writing the test procedure is another opportunity to prioritize the tests, to ensure that the best testing is done in the time available. A good thumb rule is '**Find the scary stuff first**'. However the definition of what is 'scary' depends on the business, system or project and depends up on the risk of the project.


educlash.com
"Anyone who has never made a mistake has never tried anything new."
ALBERT EINSTEN
www.educlash.com

Categories of Test design technique

◆ What is test design technique?

- A test design technique basically helps us **to select a good set of tests from the total number of all possible tests** for a given system.

- There are many different types of software testing technique, each with its own strengths and weaknesses.

- Each individual technique is good at finding particular types of defect and relatively poor at finding other types.

What are the categories of test design techniques?

- **For example**, a technique that explores the upper and lower limits of a single input range is more likely to find ***boundary value defects*** than defects associated with combinations of inputs.

- Similarly, testing performed at **different stages in the SDLC** will find different types of defects; **component testing** is more likely to *find coding logic defects than system design defects.*

- Referring to the figure in last Unit, each testing technique falls into one of a number of different categories. Broadly speaking there are two main categories:
  - **Static technique**
  - **Dynamic technique**

Dynamic techniques are subdivided into three more categories:

- **Specification-based** (black-box, also known as behavioral techniques),
- **Structure-based** (white-box or structural techniques) and
- **Experience- based.**

➢ Specification-based techniques include both functional and nonfunctional techniques (i.e. quality characteristics).

What is Static testing technique? (Revision)

- Static testing is the testing of the software work products manually, or with a set of tools, but they are **not executed**.

- It starts early in the Life cycle and so it is done during the verification process.

- **It does not need computer as the testing of program is done without executing the program.** For example: reviewing walk through, inspection, etc.

- Most static testing techniques can be used **to 'test' any form of document including source code, design documents and models, functional specifications and requirement specifications.**

What is Dynamic testing technique?

- This testing technique needs computer for testing.
- It is done during Validation process.
- The software is tested by executing it on computer. Ex: Unit testing, integration testing, system testing.

What is black-box, Specification-based, also known as behavioral testing techniques?

- Specification-based testing technique is also known as **'black-box'** or input/output driven testing techniques because they view the software as a black-box with inputs and outputs.

- The testers have no knowledge of how the system or component is structured inside the box. ***In black-box testing the tester is concentrating on what the software does, not how it does it.***

- The definition mentions both functional and non-functional testing. ***Functional testing*** is concerned with ***whether the system does its features or functions.*** Non-functional testing is concerned with examining how well the system does. ***Non-functional testing like performance, usability, portability, maintainability, etc***.

- **Specification-based techniques** are appropriate at all levels of testing (component testing through to acceptance testing) where a specification exists. **For example,** when performing system or acceptance testing, the requirements specification or functional specification may form the basis of the tests.

Black box testing – Steps overview

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.

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.

Tools used for Black Box Testing:

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

Black box testing strategy:

Following are the prominent test strategy amongst the many used in Black Box Testing

- **Equivalence Class Testing:** It is used to minimize the number of possible test cases to an optimum level while maintains reasonable test coverage.

- **Boundary Value Testing:** Boundary value testing is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not.It is very useful in reducing the number of test cases. It is mostly suitable for the systems where input is within certain ranges.

- **Decision Table Testing**: A decision table puts causes and their effects in a matrix. There is unique combination in each column.

- There are four specification-based or black-box technique:
  - **Equivalence partitioning**
  - **Boundary value analysis**
  - **Decision tables**
  - **State transition testing**

Equivalence partitioning

- **Concepts**: Equivalence partitioning is a method for deriving test cases. In this method, classes of input conditions called equivalence classes are identified such that each member of the class causes the same kind of processing and output to occur.

- In this method, the tester identifies various equivalence classes for partitioning. A class is a set of input conditions that are is likely to be handled the same way by the system. If the system were to handle one case in the class erroneously, it would handle all cases erroneously.

What is Equivalence partitioning in Software testing?

- Equivalence partitioning (EP) is a specification-based or black-box technique.

- It can be applied at any level of testing and is often a good technique to use first.

- The idea behind this technique **is to divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same** (i.e. the system should handle them equivalently), hence 'equivalence partitioning'. **Equivalence partitions** are also known **as equivalence classes** – the two terms mean exactly the same thing.

- In equivalence-partitioning technique, **we need to test only one condition from each partition**. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software.

- **If one condition in a partition works**, we assume **all of the conditions in that partition will work,** and so there is little point in testing any of these others.

- Similarly, if one of the conditions in a partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition.

educlash.com

"Building a **career** is like mining for gold. You have to keep chipping away until you *discover* what you're meant to do."

DAN SCHAWBEL

DESIGNING TEST CASES USING EQUIVALENCE PARTITIONING

To use equivalence partitioning, you will need to perform two steps

1. Identify the equivalence classes
2. Design test cases

STEP 1: IDENTIFY EQUIVALENCE CLASSES

Take each input condition described in the specification and derive at least two equivalence classes for it. One class represents the set of cases which satisfy the condition (the valid class) and one represents cases which do not (the invalid class).

Following are some guidelines for identifying equivalence classes:

If the requirements state that a numeric value is input to the system and must be within a range of values, identify one valid class inputs which are within the valid range and two invalid equivalence classes inputs which are too low and inputs which are too high.

For example, if an item in inventory (numeric field) can have a quantity of +1 to +999, identify the following classes:

1. **One valid class**: (QTY is greater than or equal to +1 and is less than or equal to +999). This is written as (+1 < = QTY < = 999)
2. **The invalid class** (QTY is less than 1), also written as (QTY < 1) i.e. 0, -1, -2, … so on
3. **The invalid class** (QTY is greater than 999), also written as (QTY >999) i.e. 1000, 1001, 1002, 1003… so on.

Invalid class	Valid Class	Invalid class
0	1	1000
-1	2	1001
-2	3	1002
-3	4	1003
-4… So on	5… up to 999	1004.. So on

What is Boundary value analysis in software testing?

- Boundary value analysis (BVA) is based on testing at the boundaries between partitions.

- Here we have both valid boundaries (in the valid partitions) and invalid boundaries (in the invalid partitions).

- As an example, **consider a printer that has an input option of the number of copies to be made, from 1 to 99.**

- To apply boundary value analysis, we will take the **minimum and maximum (boundary) values from the valid partition (1 and 99 in this case)** together with the first or last value respectively in each of the invalid partitions adjacent to the valid partition (0 and 100 in this case).

- In this example we would have three equivalence partitioning tests (one from each of the three partitions) and four boundary value tests.

While testing why it is important to do both EP and BV ?

- Technically, **because every boundary is in some partition,** if you did only boundary value analysis you would also have tested every equivalence partition.

- However**,** **this approach may cause problems if that value fails – was it only the boundary value that failed or did the whole partition fail?**

- Also by testing only boundaries we would probably not give the users much confidence as we are using extreme values rather than normal values.

- The boundaries may be more difficult (and therefore more costly) to set up as well.

- **For example**, in the printer copies example described earlier we identified the following boundary values:

- Suppose we test only the valid boundary values 1 and 99 and nothing in between. If both tests pass, this seems to indicate that all the values in between should also work.

- However, suppose that one page prints correctly, but 99 pages do not. Now we don't know whether any set of more than one page works, so the first thing we would do would be to test for say 10 pages, i.e. a value from the equivalence partition.

- We recommend that you test the partitions separately from boundaries – this means choosing partition values that are NOT boundary values.

- However, if you use the three-value boundary value approach, then you would have valid boundary values of 1, 2, 98 and 99, so having a separate equivalence value in addition to the extra two boundary values would not give much additional benefit.

- But notice that one equivalence value, e.g. 10, replaces both of the extra two boundary values (2 and 98). This is why equivalence partitioning with two-value boundary value analysis is more efficient than three-value boundary value analysis.

DECISION TABLE-BASED TESTING

Decision Table Based Testing

- **Decision tables** are a precise yet compact way to model complicated logic
- **A decision table is a table with various conditions and their corresponding actions.**
- **It is divided into four parts, condition stub, action stub, condition entry, and action entry**
- **Decision table is based on logical relationships just as the truth table.**
- **It is a tool that helps us look at the "complete" combination of conditions**

Decision Tables - Usage

- Decision tables make it easy to observe that all possible conditions are accounted for.
- Decision tables can be used for:
  - Specifying complex program logic
  - Generating test cases (Also known as *logic-based testing)*
- *Logic-based testing* is considered as:
  - Structural testing when applied to structure (i.e. control flowgraph of an implementation).
  - functional testing when applied to a specification.

Decision Tables - Structure

Conditions - *(Condition stub)*	Condition Alternatives – *(Condition Entry)*
Actions – *(Action Stub)*	Action Entries

- Each condition corresponds to a variable, relation

- Possible values for conditions are listed among the condition alternatives
  - Boolean values (True / False) – Limited Entry Decision Tables
  - Several values – Extended Entry Decision Tables

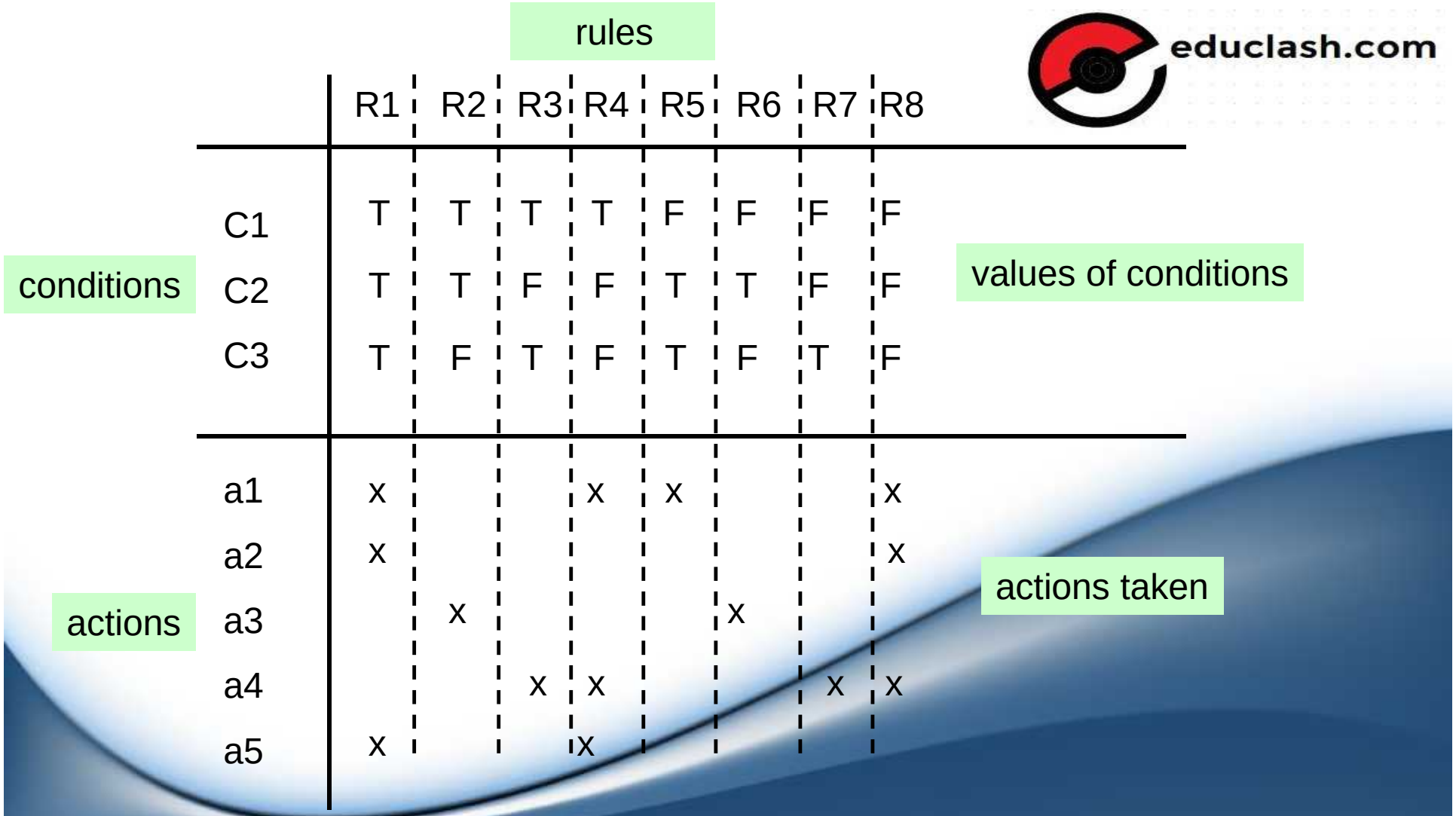
- Each action is a procedure or operation to perform

- The entries specify whether (or in what order) the action is to be performed

- A *condition stub* is declared as a statement of a condition.
- A *condition entry* provides a value assigned to the condition stub.
- Similarly, an *action* (or *decision*) composes two elements: an *action stub* and an *action entry*.

Read a Decision Table by columns of rules : R1 says when all conditions are T, then actions a1, a2, and a5 occur

Components of a Decision Table



Conditions in Decision Table

- **The conditions in the decision table may take on any number of values. When it is binary, then the decision table conditions are just like a truth table set of conditions. (Note that the conditions do not have to be binary --- table gets "big" then.)**

- **The decision table allows the iteration of all the combinations of values of the condition, thus it provides a "completeness check."**

- **The conditions in the decision table may be interpreted as the inputs, and the actions may be thought of as outputs. OR conditions needs to be thought as inputs, and actions can be processing**

Decision Table - Example

Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognized	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Printer Troubleshooting

What is Decision table in software testing?

- The techniques of equivalence partitioning and boundary value analysis are often applied to specific situations or inputs.

- However, if different combinations of inputs result in different actions being taken, this can be more difficult to show using equivalence partitioning and boundary value analysis, which tend to be more focused on the user interface.

- The other two specification-based software testing techniques, decision tables and state transition testing are more focused on business logic or business rules.

- A **decision table** is a good way to deal with combinations of things (e.g. inputs).

- This technique is sometimes also referred to as a **'cause-effect' table**.

- The reason for this is that there is an associated logic diagramming technique called **'cause-effect graphing'** which was sometimes used to help derive the decision table

- Decision tables provide a **systematic way of stating complex business rules,** which is useful for **developers as well as for testers.**

- Decision tables **can be used in test design** whether or not they are used in specifications, as they **help testers explore the effects of combinations of different inputs and other software states that must correctly implement business rules.**

- **Testing combinations can be a challenge**, as the number of **combinations can often be huge.** Testing all combinations may be impractical if not impossible.

Decision Table Development Methodology

1. Determine conditions and values
2. Determine maximum number of rules
3. Determine actions
4. Encode possible rules
5. Encode the appropriate actions for each rule
6. Verify the policy
7. Simplify the rules (reduce if possible the number of columns)

How to Use decision tables for test designing

How to Use decision tables for test designing?

- The **first task is to identify a suitable function or subsystem** which reacts according to a combination of inputs or events.

- The **system should not contain too many inputs otherwise the number of combinations will become unmanageable**.

- It is **better to deal with large numbers of conditions** by dividing them into subsets and dealing with the subsets one at a time

- Once you have **identified the aspects that need to be combined, then you put them into a table listing all the combinations of True and False for each of the aspects.**

Guidelines and Observations

- Decision Table testing is most appropriate for programs where
  - There is a lot of decision making
  - There are important logical relationships & calculations
  - There are cause and effect relationships between input and output
  - There is complex computation logic (high cyclomatic complexity)
- Decision tables do not scale up very well
- Decision tables can be iteratively refined

Advantages/Disadvantages of Decision Table

- **Advantages:**
  - **Allow us to start with a "complete" view.**
  - **Allow us to look at and consider "dependence," "impossible," and "not relevant" situations and eliminate some test cases.**
  - **Allow us to detect potential error in our specifications**

- **Disadvantages:**
  - **Need to decide (or know) what conditions are relevant for testing**
  - **Scaling up can be massive: 2^n for n conditions.**


educlash.com
"Whatever you can do or dream you can do, begin it. Boldness has genius, power and magic in it."
GOETHE
www.educlash.com

State transition testing

What is State transition testing in software testing?

- **State transition testing is used where some aspect of the system can be described in what is called a 'finite state machine'.**

- This 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.

Advantages of the state transition technique

- **The model can be as detailed or as abstract as you need it to be.**
- Important part of the system can **be modeled in detail** i.e. **requires more testing.**
- Where the system is **less important (requires less testing), the model can use a single state to** signify what would otherwise be a series of different states.

A **state transition model has four basic parts:**

- *The **states** that the software may occupy (open/closed or funded/insufficient funds);*
- *The **transitions** from one state to another (not all transitions are allowed);*
- *The **events** that cause a transition (closing a file or withdrawing money);*
- *The **actions** that result from a transition (an error message or being given your cash).*

- Hence we can see that **in any given state, one event can cause only one action,** but that the **same event – from a different state – may cause a different action and a different end state.**

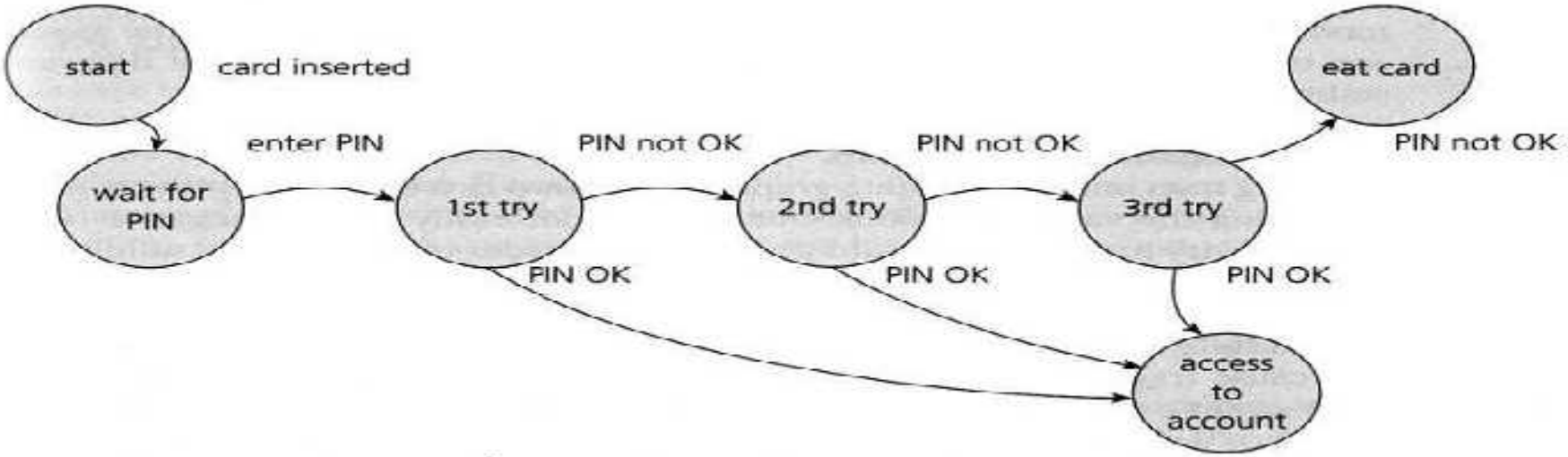


FIGURE 4.2 State diagram for PIN entry

We will look first at test cases that execute valid state transitions.

- Figure 4.2 , shows an example of entering a Personal Identity number (PIN) to a bank account.

- **The states** are shown as **circles**, **the transitions as lines** with arrows and the **events as the text near the transitions**.

- (We have not shown the actions explicitly on this diagram, but they would be a message to the customer saying things such as 'Please enter your PIN'.)

- The state diagram ***shows seven states*** but only four possible events (Card inserted, Enter PIN, PIN OK and PIN not OK).

- We have not specified all of the possible transitions here – there would also be a **time-out from 'wait for PIN' and from the three tries which would go back to the start state** after the time had elapsed and would probably eject the card.

- We have not specified all the possible events either – there would be **a 'cancel' option from 'wait for PIN' and from** the three tries, which would **also go back to the start state and eject the card.**

In deriving test cases, we may start with a typical scenario.

- **First test case** here would be the normal situation, where the correct PIN is entered the first time.

- **A second test** (to visit every state) would be to enter an incorrect PIN each time, so that the system eats the card.

- **A third test** we can do where the PIN was incorrect the first time but OK the second time, and another test where the PIN was correct on the third try. These tests are probably less important than the first two.

- Note that a transition does not need to change to a different state (although all of the transitions shown above do go to a different state). So there could be a transition from ‘access account’ which just goes back to ‘access account’ for an action such as ‘request balance’.

- Test conditions can be derived from the state graph in various ways.
- Each **state transition can be noted as a test condition**. However this state diagram, even though it is incomplete, still gives us information on which to design some useful tests and to explain the state transition technique.
- We would be able to identify the coverage of a set of tests in terms of transitions.
- State transition testing is regarded as a **black-box technique.**

Use case testing

- Use case testing is a technique that helps us identify test cases that exercise the whole system on a transaction by transaction basis from start to finish.

- A use case is a description of a particular use of the system by an actor (a user of the system).

- Each use case describes about interactions the actor has with the system in order to achieve a specific task (or, at least, produce something of value to the user).

- Actors are generally people but they may also be other systems.
- **Use cases are a sequence of steps that describe the interactions between the actor and the system.**
- **Use cases are defined in terms of the actor, not the system,** describing what the actor does and what the actor sees rather than what inputs the system expects and what is the system's outputs.
- **They often use the language and terms of the business rather than technical terms, especially when the actor is a business user.**
- They serve as the **foundation for developing test cases** mostly at the system and acceptance testing levels.

- **Use cases can uncover integration defects, that is, defects caused by the incorrect interaction between different components.** Used in this way, the actor may be something that the system interfaces to such as a communication link or sub-system.

- **Use cases describe the process flows through a system based on its most likely use. This makes the test cases derived from use cases particularly good for finding defects in the real-world use of the system** (i.e. the defects that the users are most likely to come across when first using the system).

- Each use case usually has a mainstream (or most likely) scenario and sometimes additional alternative branches (covering, for example, special cases or exceptional conditions).

- Each **use case must specify any preconditions that need to be met for the use case to work.**

- **Use cases must also specify post conditions that are observable results and a description of the final state of the system after the use case has been executed successfully.**

- The ATM PIN example is shown below in Figure 4.3. We show successful and unsuccessful scenarios.

- In this diagram we can see the interactions between the A (actor – in this case it is a human being) and S (system).

- From step 1 to step 5 that is success scenario it shows that the card and pin both got validated and allows Actor to access the account.

- But in extensions there can be three other cases that is 2a, 4a, 4b which is shown in the diagram below.

- For use case testing, we would have a **test of the success scenario and one testing for each extension.**

- In this example, we may give extension 4b a higher priority than 4a from a security point of view.

- System requirements can also be specified as a set of use cases. This approach can make it easier to involve the users in the requirements gathering and definition process.

	Step	Description
Main Success Scenario **A: Actor** **S: System**	1	A: Inserts card
	2	S: Validates card and asks for PIN
	3	A: Enters PIN
	4	S: Validates PIN
	5	S: Allows access to account
Extensions	2a	Card not valid S: Display message and reject card
	4a	PIN not valid S: Display message and ask for re-try (twice)
	4b	PIN invalid 3 times S: Eat card and exit

FIGURE 4.3 Partial use case for PIN entry

White-box
or
Structure-based
or
structural testing techniques

White Box Testing

- ✓ Testing the Internal program logic
- ✓ White box testing is also called as Structural testing.
- ✓ User does require the knowledge of software code.

Purpose

- ➢ Testing all loops
- ➢ Testing conditional statements
- ➢ Testing data structures
- ➢ Testing Logic Errors
- ➢ Testing Incorrect assumptions
- ➢ Structure = 1 Entry + 1 Exit with certain Constraints, Conditions and Loops.
- ➢ Logic Errors and incorrect assumptions most are likely to be made while coding for “special cases”. Need to ensure these execution paths are tested.

What is white-box or Structure-based or structural testing techniques?

- Structure-based testing techniques **use the internal structure of the software to derive test cases.**

- **Structure-based testing** technique is also known as **'white-box'** or 'glass-box' testing technique because **here the testers require knowledge of how the software is implemented, how it works.**

- In white-box testing the **tester is concentrating on how the software does it**. For example, a structural technique may be concerned with exercising loops in the software.

- **Different test cases may be derived to exercise the loop once, twice, and many times.** This may be done regardless of the functionality of the software.

- Structure-based techniques can also be **used at all levels of testing. Developers use structure-based techniques in component testing and component integration testing,** especially where there is good tool support for code coverage.

- Structure-based techniques are also **used in system and acceptance testing, but the structures are different.**

Approaches / Methods / Techniques for White Box Testing

➢ Basic Path Testing (Cyclomatic Complexity(Mc Cabe Method)

- Measures the logical complexity of a procedural design.
- Provides flow-graph notation to identify independent paths of processing
- Once paths are identified - tests can be developed for - loops, conditions
- Process guarantees that every statement will get executed at least once.

➢ **Structure Testing:**

- **Condition Testing** - All logical conditions contained in the program module should be tested.

- **Data Flow Testing-** Selects test paths according to the location of definitions and use of variables.

- **Loop Testing:**
  - Simple Loops
  - Nested Loops
  - Concatenated Loops
  - Unstructured Loops

What is Experience- based testing technique?

- In **experience-based techniques,** people's knowledge, skills and background are of prime importance to the test conditions and test cases.

- The **experience of both technical and business people is required, as they bring different perspectives to the test analysis and design process.** Because of the previous experience with similar systems, they may have an idea as what could go wrong, which is very useful for testing.

- **Experience-based techniques go together with specification-based and structure-based techniques,** and are also used when there is no specification, or if the specification is inadequate or out of date.

- This may be the only type of technique used for low-risk systems, but this approach may be particularly useful under extreme time pressure – in fact this is one of the factors leading to exploratory testing.

There are two types of experience based techniques :

- **Error Guessing**
- **Exploratory Testing**

What is Exploratory testing in software testing?

- As its name implies, exploratory testing is about **exploring, finding out about the software, what it does, what it doesn't do, what works and what doesn't work.**

- The **tester is constantly making decisions about what to test next and where to spend the (limited) time.** This is an approach that is **most useful** when there are **no or poor specifications and when time is severely limited.**

- **Exploratory testing is a hands-on approach in which testers are involved in minimum planning and maximum test execution.**

- The planning involves the **creation of a test document,** a short declaration of the scope & a short (1 to 2 hour) time-boxed test **effort, the objectives and possible approaches to be used.**

- **The test design and test execution activities are performed in parallel typically without formally documenting the test conditions, test cases or test scripts**. This does not mean that other, more formal testing techniques will not be used.

- For example, **the tester may decide to use boundary value analysis but will think through and test the most important boundary values without necessarily writing them down.** Some notes will be written during the exploratory-testing session, so that a report can be produced afterwards.

- Test logging is undertaken as test execution is performed, documenting the key aspects of what is tested, any defects found and any thoughts about possible further testing.

- It can also serve to complement other, more formal testing, helping to establish greater confidence in the software.

- In this way, exploratory testing can be used as a check on the formal test process by helping to ensure that the most serious defects have been found.

What is Error guessing in software testing?

- The Error guessing is a **technique where the experienced and good testers are encouraged to think of situations in which the software may not be able to cope.**

- Some people seem to be **naturally good at testing and others are good testers because they have a lot of experience** either as a tester or working with a particular system and so are able to find out its weaknesses.

- It also **saves a lot of time because of the assumptions and guessing made by the experienced testers** to find out the defects which otherwise won't be able to find.

- The **success of error guessing is very much dependent on the skill of the tester,** as good testers know where the defects are most likely to be.

- This is why an error guessing approach, used after more formal techniques have been applied to some extent, can be very effective.

- In **using more formal techniques**, the tester is likely to gain a better understanding of the system, **what it does and how it works.** With this better understanding, he or she is likely to be **better at guessing ways in which the system may not work properly.**

- **Typical conditions to try include division by zero, blank (or no) input, empty files and the wrong kind of data (e.g. alphabetic characters where numeric are required).**

- **A structured approach to the error-guessing technique is to list possible defects or failures and to design tests that attempt to produce them**. These defect and failure lists can be built based on the tester's own experience or that of other people, available defect and failure data, and from common knowledge about why software fails.

"Opportunities don't often come along. So, when they do, you have to *grab* them."

AUDREY HEPBURN

Structure-based technique in software testing

What is Structure-based technique in sw testing?or Using structure-based techniques to measure coverage and design tests

- **Structure-based techniques serve two purposes:**
  - **test coverage measurement**
  - **structural test case design**

- They are often used first to **assess the amount of testing performed by tests derived from specification-based techniques**, i.e. to assess coverage.

- They are then **used to design additional tests with the aim of increasing the test coverage**.

- Structure-based test design techniques **are a good way of generating additional test cases that are different from existing tests.**

- They can help ensure more breadth of testing, in the sense that test cases that achieve 100% **coverage** in any measure will be exercising all parts of the software from the point of view of the items being covered.

What is test coverage in software testing? It's advantages and disadvantages

- **Test coverage measures the amount of testing performed by a set of test.** Wherever we can **count** things and **can tell whether or not each of those things has been tested by some test, then we can measure coverage and is known as test coverage.**

- The **basic coverage measure is where the 'coverage item' is whatever we have been able to count** and see whether a test has exercised or used this item.

$$\text{Coverage} = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}} \times 100\%$$

- There is danger of misinterpretation in using a coverage measure. Because, 100% coverage does *not* mean 100% tested.

- **Coverage techniques** measure only **one dimension of a multi-dimensional concept.** Two different test cases may achieve exactly the same coverage but the input data of one may find an error that the input data of the other doesn't.

Benefit of code coverage measurement:

- It creates **additional test cases to increase coverage**

- It helps in **finding areas of a program not exercised by a set of test cases**

- It helps in **determining a quantitative measure of code coverage, which indirectly measure the quality of the application or product.**

Drawback of code coverage measurement:

- One drawback of code coverage measurement is that **it measures coverage of what has been written,** i.e. the code itself; **it cannot say anything about the software that has *not* been written.**

- **If a specified function has not been implemented** or a function was omitted from the specification, then structure-based techniques cannot say anything about them it only looks at a structure which is already there.

Where to apply this test coverage in software testing?

- The answer of the question is that test coverage can be used in any level of the testing.
- Test coverage can be measured based on a number of different structural elements in a system or component.
- Coverage can be measured at component testing level, integration-testing level or at system- or acceptance-testing levels.
- For example, at system or acceptance level, the coverage items may be requirements, menu options, screens, or typical business transactions. At integration level, we could measure coverage of interfaces or specific interactions that have been tested.

We can also measure coverage for each of the specification-based techniques or black-box testing:

- **EP: percentage of equivalence partitions exercised** (we could measure valid and invalid partition coverage separately if this makes sense);

- **BVA: percentage of boundaries exercised (**we could also separate valid and invalid boundaries if we wished);

- **Decision tables: percentage of business rules or decision table columns tested;**

<u>State transition testing</u>: there are a number of possible coverage measures:

- Percentage of states visited
- Percentage of (valid) transitions exercised
- Percentage of invalid transitions exercised (from the state table).

Why to measure code coverage?

- To know whether we have enough testing in place
- To maintain the test quality over the life cycle of a project
- To know how well our tests actually tested our code

How we can measure the coverage?

- Coverage measurement of code is best done **by using tools** and there are a number of such tools in the market. These tools help in:

- **Increasing the quality and productivity of testing**.

- They increase quality by ensuring that more **structural aspects are tested, so defects on those structural paths can be found.**

- They **increase productivity and efficiency by highlighting** tests that may be redundant, i.e. testing the same structure with different data (as there is possibility of finding the defects by testing the same structure with different data).

What are the types of coverage?

1) Statement coverage

2) Decision coverage

3) Condition coverage

What is Statement coverage? Advantages and Disadvantages

- The statement coverage is also known as line coverage or segment coverage.
- The statement coverage **covers only the true conditions.**
- Through statement coverage we can identify the statements executed and where the code is not executed.
- In this process each and every line of code needs to be checked and executed

Advantage of statement coverage:

- It verifies what the written code is expected to do and not to do
- It measures the quality of code written
- It checks the flow of different paths in the program and it also ensure that whether those path are tested or not.

Disadvantage of statement coverage:

- It cannot test the false conditions.
- It does not report that whether the loop reaches its termination condition.
- It does not understand the logical operators.

The statement coverage can be calculated as shown below:

$$\text{Statement coverage} = \frac{\text{Number of statements exercised}}{\text{Total number of statements}} \times 100\%$$

- To understand the statement coverage in a better way let us take an example which is basically a pseudo-code.

- It is not any specific programming language, but should be readable and understandable to you, even if you have not done any programming yourself. Consider code sample 4.1 :

```
READ X
READ Y
I F X>Y THEN Z = 0
ENDIF
```

Code sample 4.1

- To achieve 100% statement coverage of this code segment just one test case is required, one which ensures that variable X contains a value that is greater than the value of variable Y, for example, X = 12 and Y = 10. Here we are doing **structural test *design* first,** since we **are choosing our input values in order ensure statement coverage.**

- Now, let's take another example where we will measure the **coverage first**.

  – In order to simplify the example, we will regard each line as a statement. A statement may be on a single line, or it may be spread over several lines.

  – One line may contain more than one statement, just one statement, or only part of a statement. Some statements can contain other statements inside them.

– In code sample 4.2, we have two read statements, **one assignment statement,** and then one IF statement on three lines, but the IF statement contains another statement (print) as part of it.

```
1 READ X
2 READ Y
3 Z =X + 2*Y
4 IF Z> 50 THEN
5 PRINT LARGE Z
6 ENDIF
```

Code sample 4.2

- Although it isn't completely correct, we have numbered each line and will regard each line as a statement. Let's analyze the coverage of a set of tests on our six-statement program:

- **TEST SET 1**
  Test 1_1: **X= 2, Y = 3**

  Test 1_2: **X =0, Y = 25**

  Test 1_3: **X =47, Y = 1**

Which statements have we covered?

- In Test 1_1, the value of Z will be 8, so we will cover the statements on lines 1 to 4 and line 6.
- In Test 1_2, the value of Z will be 50, so we will cover exactly the same statements as Test 1_1.
- In Test 1_3, the value of Z will be 49, so again we will cover the same statements.

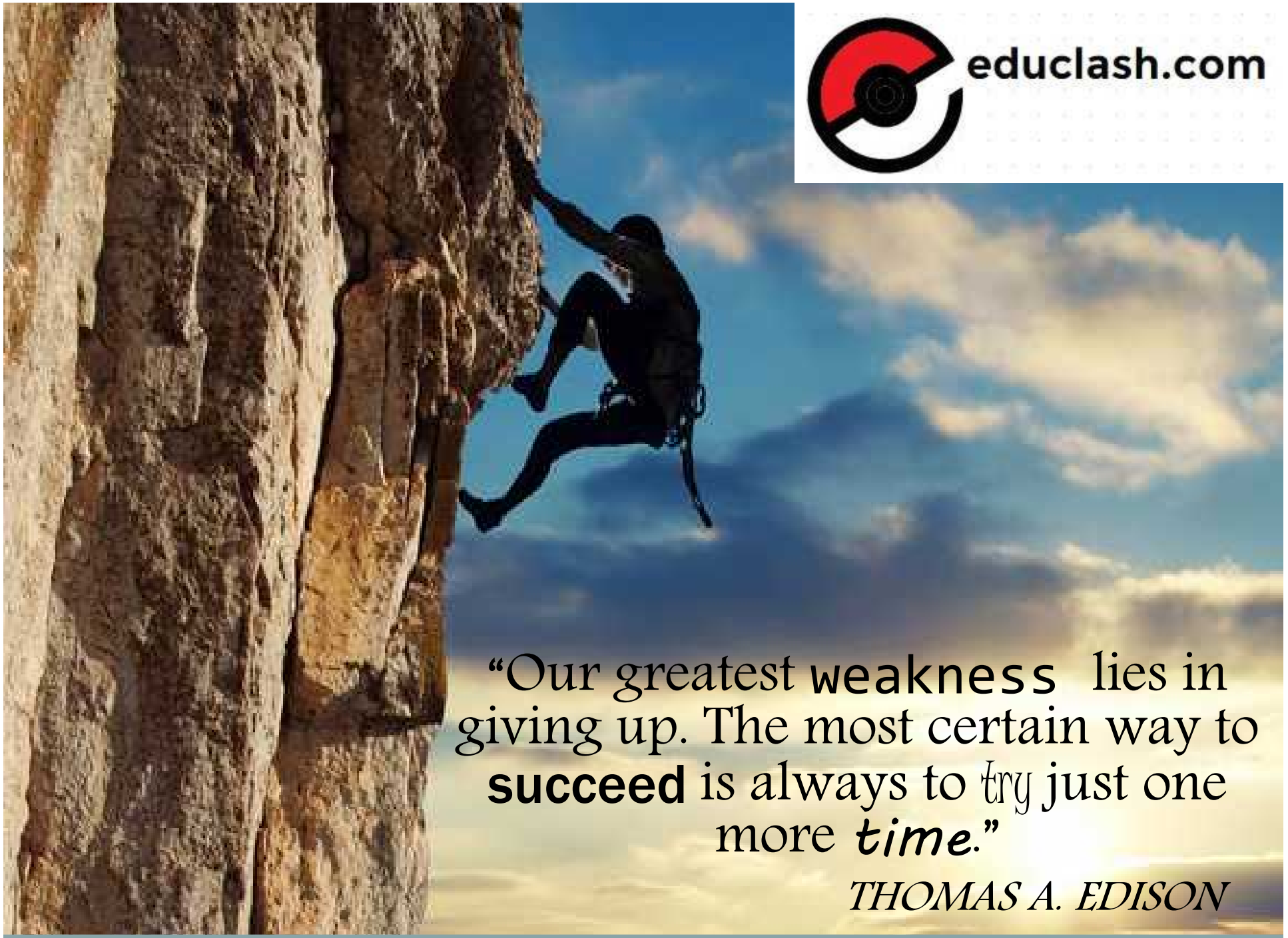
- Since **we have covered five out of six statements,** we have **83% statement coverage** **(**with three tests). What test would we need in order to cover statement 5, the one statement that we haven't exercised yet? How about this one:

- **Test 1_4: X = 20, Y = 25**

- **This time the value of Z is 70, so we will print 'Z' and we have exercised all six of the statements.**

- **So now statement coverage = 100%.**

- Notice that we measured coverage first, and then designed a test to cover the statement that we had not yet covered.


educlash.com
"Our greatest weakness lies in giving up. The most certain way to succeed is always to try just one more time."
THOMAS A. EDISON
www.educlash.com

What is Decision coverage? Advantages and disadvantages

- Decision coverage also known as **branch coverage or all-edges coverage.**

- It **covers both the true and false conditions** unlikely the statement coverage.

- A branch is the outcome of a decision, **so branch coverage simply measures which decision outcomes have been tested.** This sounds great because it takes a more in-depth view of the source code than simple statement coverage

- **A decision is an IF statement, a loop control statement** (e.g. DO-WHILE or REPEAT-UNTIL), or a CASE statement, where there are two or more outcomes from the statement. With an IF statement, the exit can either be TRUE or FALSE, depending on the value of the logical condition that comes after IF.

Advantages of decision coverage:

- To validate that all the branches in the code are reached
- To ensure that no branches lead to any abnormality of the program's operation
- It eliminate problems that occur with statement coverage testing

Disadvantages of decision coverage:

- This metric ignores branches within boolean expressions which occur due to short-circuit operators.

The decision coverage can be calculated as given below:

$$\text{Decision coverage} = \frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} \times 100\%$$

- In the previous section we saw that just one test case was required to achieve 100% statement coverage. However, decision coverage requires each decision to have had both a True and False outcome.

- Therefore, to achieve 100% decision coverage, a second test case is necessary where A is less than or equal to B which ensures that the decision statement 'IF A > B' has a False outcome. So one test is sufficient for 100% statement coverage, but two tests are needed for 100% decision coverage. It is really very important to note that **100% decision coverage guarantees 100% statement coverage, but *not* the other way around.**

```
1 READ A
2 READ B
3 C = A – 2 *B
4 IFC <0THEN
5 PRINT “C negative”
6 ENDIF
```

Code sample 4.3

- Let’s suppose that we already have the following test, which gives us 100% statement coverage for code sample 4.3.
- TEST SET 2 Test 2_1: A = 20, B = 15
- The value of C is -10, so the condition ‘C < 0′ is True, so we will print ‘C negative’ and we have executed the True outcome from that decision statement. But we have not executed the False outcome of the decision statement. What other test would we need to exercise the False outcome and to achieve 100% decision coverage?

- Before we answer that question, let's have a look at another way to represent this code. Sometimes the decision structure is easier to see in a control flow diagram (see Figure 4.4).

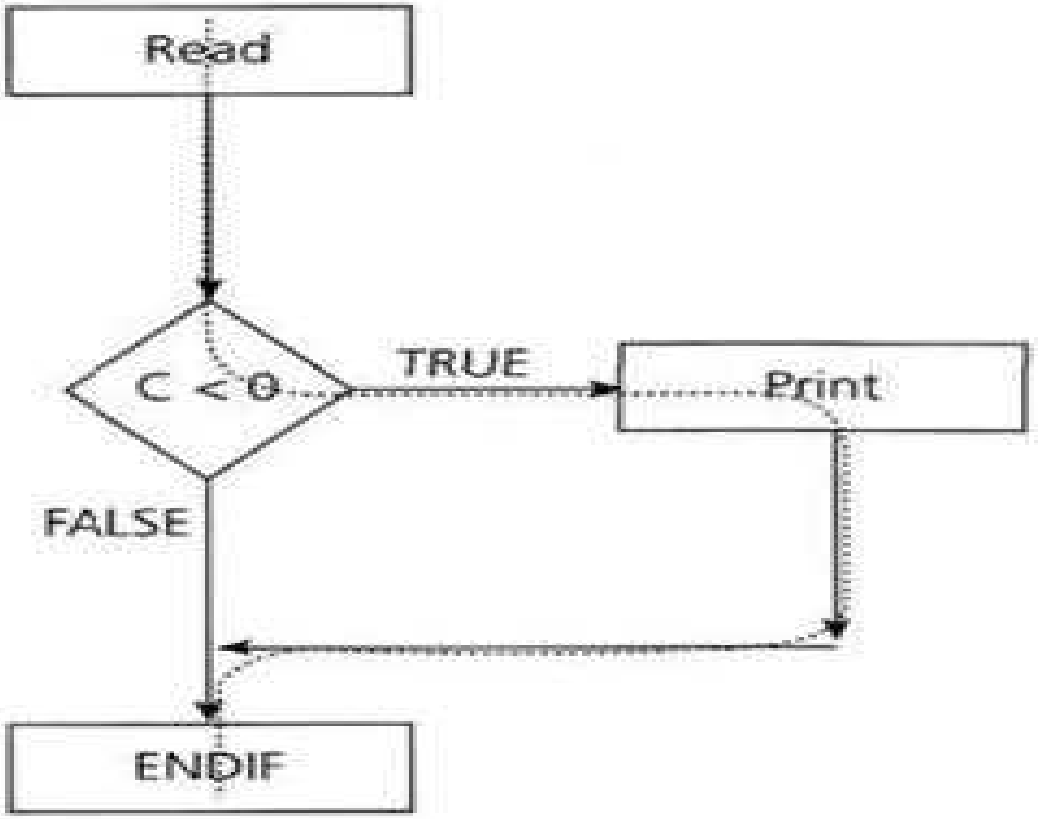


FIGURE 4.4 Control flow diagram for code sample 4.3

- The dotted line shows where Test 2_1 has gone and clearly shows that we haven't yet had a test that takes the False exit from the IF statement.
  Let's modify our existing test set by adding another test:

- TEST SET 2
  Test 2_1: A = 20, B = 15
  Test 2_2: A = 10, B = 2

- This now covers both of the decision outcomes, True (with Test 2_1) and False (with Test 2_2).

What is Condition coverage?

- This is **closely related to decision coverage** but has ***better sensitivity to the control flow.***
- However, full condition coverage **does not guarantee full decision coverage.**
- Condition coverage **reports the true or false outcome of each condition.**
- Condition coverage **measures the conditions independently of each other.**

How to choose that which testing technique is best?

OR

List the factors that influence the selection of the appropriate test design technique for a particular kind of problem, such as the type of system, risk, Customer requirements, models for use case modeling, requirements models or tester knowledge.

- How to choose which testing technique is best, decision will be based on a number of factors, both **internal and external.**

  The **internal factors** that influence the decisions about which technique to use are:

- **Models used in developing the system-** Since testing techniques are based on models used to develop that system, will to some extent govern which testing techniques can be used. For example, if the specification contains a state transition diagram, state transition testing would be a good technique to use.

- **Testers knowledge and their experience -** How much testers know about the system and about testing techniques will clearly influence their choice of testing techniques. This knowledge will in itself be influenced by their experience of testing and of the system under test.

- **Similar type of defects -** Knowledge of the similar kind of defects will be very helpful in choosing testing techniques (since each technique is good at finding a particular type of defect). This knowledge could be gained through experience of testing a previous version of the system and previous levels of testing on the current version.

- **Test objective -** If the test objective is simply to gain confidence that the software will cope with typical operational tasks then use cases would be a sensible approach. If the objective is for very thorough testing then more rigorous and detailed techniques (including structure-based techniques) should be chosen.

- **Documentation –** Whether or not documentation (e.g. a requirements specification) exists and whether or not it is up to date will affect the choice of testing techniques. The content and style of the documentation will also influence the choice of techniques (for example, if decision tables or state graphs have been used then the associated test techniques should be used).

- **Life cycle model used -** A sequential life cycle model will lend itself to the use of more formal techniques whereas an iterative life cycle model may be better suited to using an exploratory testing approach.

The **external factors** that influence the decisions about which technique to use are:

- **Risk assessment -** The greater the risk (e.g. safety-critical systems), the greater the need for more thorough and more formal testing. Commercial risk may be influenced by quality issues (so more thorough testing would be appropriate) or by time-to-market issues (so exploratory testing would be a more appropriate choice).

- **Customer and contractual requirements -** Sometimes contracts specify particular testing techniques to use (most commonly statement or branch coverage).

- **Type of system used -** The type of system (e.g. embedded, graphical, financial, etc.) will influence the choice of techniques. For example, a financial application involving many calculations would benefit from boundary value analysis.

- **Regulatory requirements -** Some industries have regulatory standards or guidelines that govern the testing techniques used. For example, the aircraft industry requires the use of equivalence partitioning, boundary value analysis and state transition testing for high integrity systems together with statement, decision or modified condition decision coverage depending on the level of software integrity required.

- **Time and budget of the project -** Ultimately how much time there is available will always affect the choice of testing techniques. When more time is available we can afford to select more techniques and when time is severely limited we will be limited to those that we know have a good chance of helping us find just the most important defects.

END OF UNIT IV