COLLATE

Software Engineering

Notes

Unit 4

Syllabus

UNIT – IV

Software Testing:
Software process, Functional testing: Boundary value analysis, Equivalence class testing, Decision table testing,
Cause effect graphing, Structural testing: Path testing, Data flow and mutation testing, unit testing, integration and
system testing, Debugging, Testing Tools & Standards.
Software Maintenance:
Management of Maintenance, Maintenance Process, Maintenance Models, Reverse Engineering, Software Reengineering,
Configuration Management, Documentation.

TESTING

Testing is the process of evaluating a system or its component(s) with the concentrating to find whether it satisfies the specified requirements or not.

Testing is executing a system in order to identify any gaps, errors, or missing requirements in contrary to the actual requirements.

Software testing can be stated as the process of verifying and validating that a software or application is bug free, meets the technical requirements as guided by it's design and development and meets the user requirements effectively and efficiently with handling all the exceptional and boundary cases.

The process of software testing aims not only at finding faults in the existing software but also at finding measures to improve the software in terms of efficiency, accuracy and usability. It mainly aims at measuring specification, functionality and performance of a software program or application.

WHO DOES TESTING?

It depends on the process and the associated stakeholders of the project(s). In the IT industry, large companies have a team with responsibilities to evaluate the developed software in context of the given requirements. Moreover, developers also conduct testing which is called **Unit Testing**. In most cases, the following professionals are involved in testing a system within their respective capacities −

- Software Tester
- Software Developer
- Project Lead/Manager
- End User

Different companies have different designations for people who test the software on the basis of their experience and knowledge such as Software Tester, Software Quality Assurance Engineer, QA Analyst, etc.

It is not possible to test the software at any time during its cycle. The next two sections state when testing should be started and when to end it during the SDLC.

SOFTWARE TESTING CAN BE DIVIDED INTO TWO STEPS:

VERIFICATION & VALIDATION

These two terms are very confusing for most people, who use them interchangeably.

1. **Verification**: it refers to the set of tasks that ensure that software correctly implements a specific function.

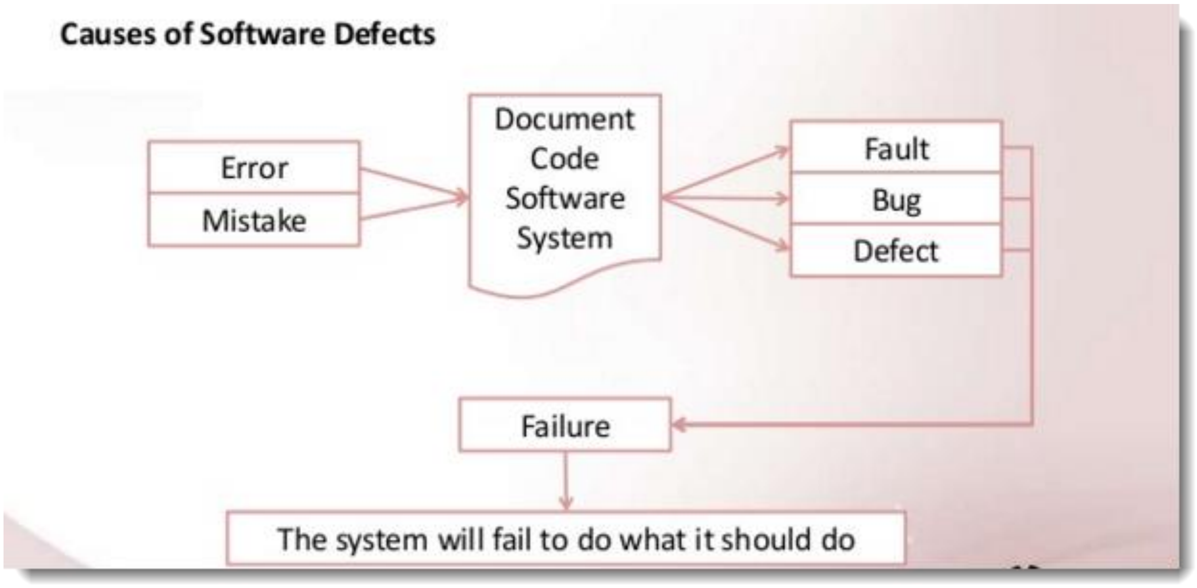
2. **Validation**: it refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

The following table highlights the differences between verification and validation.

Verification	Validation
Verification addresses the concern: "Are you building it right?"	Validation addresses the concern: "Are you building the right thing?"
Ensures that the software system meets all the functionality.	Ensures that the functionalities meet the intended behavior.
Verification takes place first and includes the checking for documentation, code, etc.	Validation occurs after verification and mainly involves the checking of the overall product.
Done by developers.	Done by testers.
It has static activities, as it includes collecting reviews, walkthroughs, and inspections to verify a software.	It has dynamic activities, as it includes executing the software against the requirements.
It is an objective process and no subjective decision should be needed to verify a software.	It is a subjective process and involves subjective decisions on how well a software works.

DIFFERENCE BETWEEN ERROR, MISTAKE, FAULT, BUG, FAILURE & DEFECT

If someone makes an **error** or ***mistake*** in using the software, this may lead directly to a problem – the software is used incorrectly and so does not behave as we expected. However, people also design and build the software and they can make mistakes during the design and build. These mistakes mean that there are ***flaws*** in the software itself. These are called ***defects*** or sometimes ***bugs*** or ***faults***.


Causes of Software Defects
Error
Mistake
Document
Code
Software
System
Fault
Bug
Defect
Failure
The system will fail to do what it should do

Error: ***Error*** is a human action that produces an incorrect result. ***It is deviation from actual and expected value.*** The mistakes made by programmer is known as an 'Error'.

Bug: ***A Bug is the result of a coding Error or Fault in the program*** which causes the program to behave in an unintended or unanticipated manner. It is an evidence of fault in the program. Bugs arise from mistakes and errors, made by people, in either a program's source code or its design.

Defect or Fault: ***A Defect is a deviation from the Requirements.*** A Software Defect is a condition in a software product which does not meet a software requirement (as stated in the requirement specifications) or end-user expectations. In other words, a defect is an error in coding or logic that causes a program to malfunction or to produce incorrect/unexpected result.

Failure: Failure is a deviation of the software from its intended purpose. It is the inability of a system or a component to perform its required functions within specified performance requirements. Failure occurs when fault executes.

DESIGN OF TEST CASES

Exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical software systems is either extremely large or infinite. Therefore, we must design an optional test suite that is of reasonable size and can uncover as many errors existing

in the system as possible. Actually, if test cases are selected randomly, many of these randomly selected test cases do not contribute to the significance of the test suite,
i.e. they do not detect any additional defects not already being detected by other test cases in the suite. Thus, the number of random test cases in a test suite is, in general, not an indication of the effectiveness of the testing. In other words, testing a system using a large collection of test cases that are selected at random does not guarantee that all (or even most) of the errors in the system will be uncovered. Consider the following example code segment which finds the greater of two integer values x and y. This code segment has a simple programming error.

```
if (x>y)
        max = x;
else
        max = x;
```

For the above code segment, the test suite, **{(x=3,y=2);(x=2,y=3)}** can detect the error, whereas a larger test suite **{(x=3,y=2);(x=4,y=3);(x=5,y=1)}** does not detect the error. So, it would be incorrect to say that a larger test suite would always detect more errors than a smaller one, unless of course the larger test suite has also been carefully designed. This implies that the test suite should be carefully designed than picked randomly. Therefore, systematic approaches should be followed to design an optimal test suite. In an optimal test suite, each test case is designed to detect different errors.

FUNCTIONAL TESTING VS. STRUCTURAL TESTING

In the black-box testing approach, test cases are designed using only the functional specification of the software, i.e. without any knowledge of the internal structure of the software. For this reason, black-box testing is known as functional testing. On the other hand, in the white-box testing approach, designing test cases requires thorough knowledge about the internal structure of software, and therefore the white-box testing is called structural testing.

BLACK-BOX TESTING

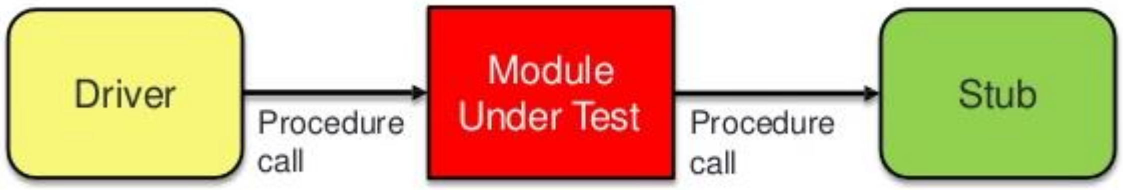
Testing in the large vs. testing in the small

Software products are normally tested first at the individual component (or unit) level. This is referred to as testing in the small. After testing all the components individually, the components are slowly integrated and tested at each level of integration (integration testing). Finally, the fully integrated system is tested (called system testing). ***Integration and system testing are known as testing in the large.***

UNIT TESTING

Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation.

Modules are required to provide the necessary environment (which either call or are called by the module under test) is usually not available until they too have been unit tested, stubs and drivers are designed to provide the complete environment for a module. The role of stub and driver modules is pictorially shown in figure. A stub procedure is a dummy procedure that has the same I/O parameters as the given procedure but has a highly simplified behavior. For example, a stub procedure may produce the expected behavior using a simple table lookup mechanism. A driver module contain the nonlocal data structures accessed by the module under test, and would also have the code to call the different functions of the module with appropriate parameter values.

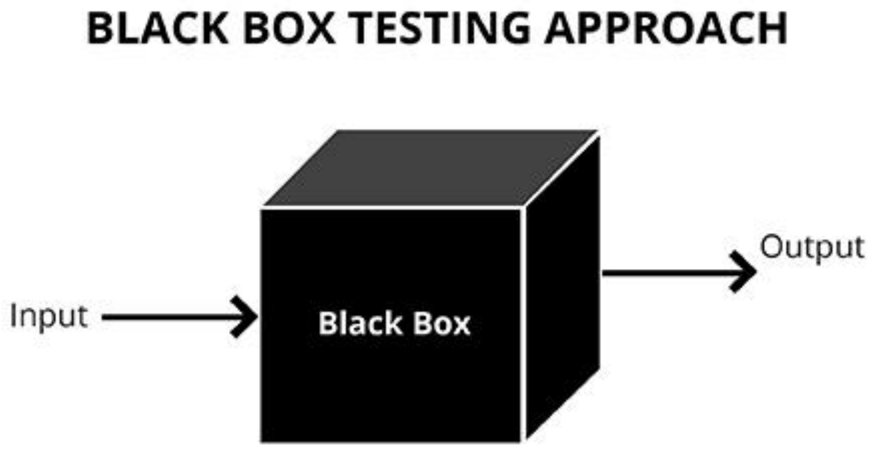


Unit testing with the help of driver and stub modules

Black Box Testing

In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches to designing black box test cases.

- Equivalence class portioning
- Boundary value analysis



Equivalence Class Partitioning

In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class. The main idea behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class. Equivalence classes for a software can be designed by examining the input data and output data. The following are some general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and

another equivalence class for invalid input values should be defined.

Example 1: For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000, there are three equivalence classes: The set of negative integers, the set of integers in the range of 0 and 5000, and the integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes and a possible test set can be: {-5,500,6000}.

Example 2: Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form y=mx + c.
The equivalence classes are the following:

- Parallel lines (m1=m2, c1≠c2)
- Intersecting lines (m1≠m2)
- Coincident lines (m1=m2, c1=c2)

Now, selecting one representative value from each equivalence class, the test suit (2, 2) (2, 5), (5, 5) (7, 7), (10, 10) (10, 10) are obtained.

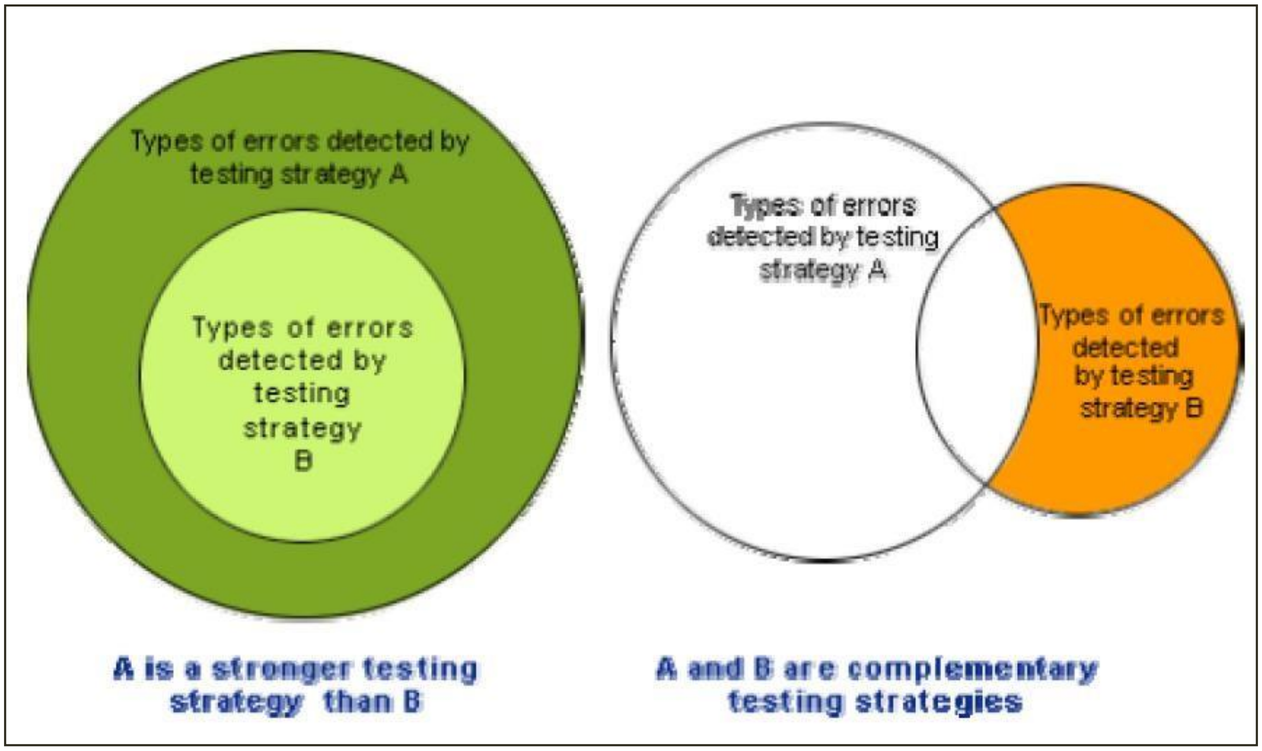
BOUNDARY VALUE ANALYSIS

A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <. Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes.

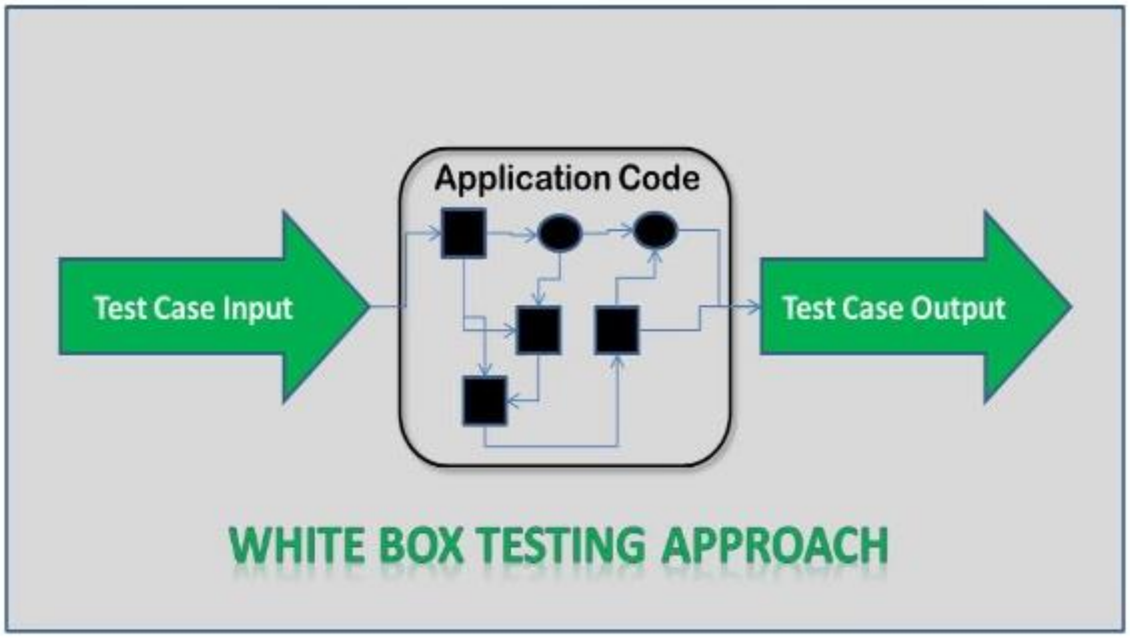
Example: For a function that computes the square root of integer values in the range of 0 and 5000, the test cases must include the following values: {0, -1,5000,5001}.

WHITE BOX TESTING

One white-box testing strategy is said to be *stronger than* another strategy, if all types of errors detected by the first testing strategy is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. When two testing strategies detect errors that are different at least with respect to some types of errors, then they are called *complementary*. The concepts of stronger and complementary testing are schematically illustrated in figure.


Types of errors detected by testing strategy A
Types of errors detected by testing strategy B
A is a stronger testing strategy than B
Types of errors detected by testing strategy A
Types of errors detected by testing strategy B
A and B are complementary testing strategies

Stronger and complementary testing strategies



COMPARISON: BLACK BOX AND WHITE BOX TESTING

BLACK BOX TESTING	**WHITE BOX TESTING**
Internal workings of an application are not required.	Knowledge of the internal workings is must.
Also known as closed box/data driven testing.	Also knwon as clear box/structural testing.
End users, testers and developers.	Normally done by testers and developers.

BLACK BOX TESTING	WHITE BOX TESTING
THis can only be done by trial and error method.	Data domains and internal boundaries can be better tested.

DIFFERENT LEVELS OF SOFTWARE TESTING

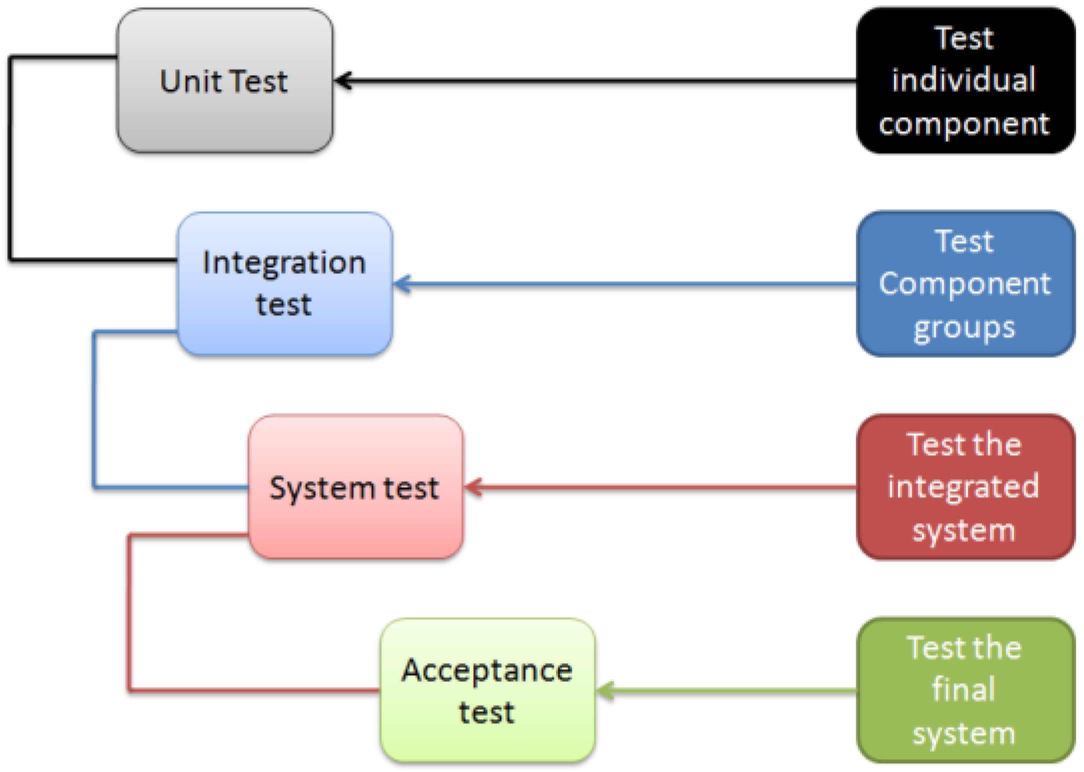
Software level testing can be majorly classified into 4 levels:

1. Unit Testing: A level of the software testing process where individual units/components of a software/system are tested. The purpose is to validate that each unit of the software performs as designed.

2. Integration Testing: A level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.

3. System Testing: A level of the software testing process where a complete, integrated system/software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.

4. Acceptance Testing: A level of the software testing process 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.



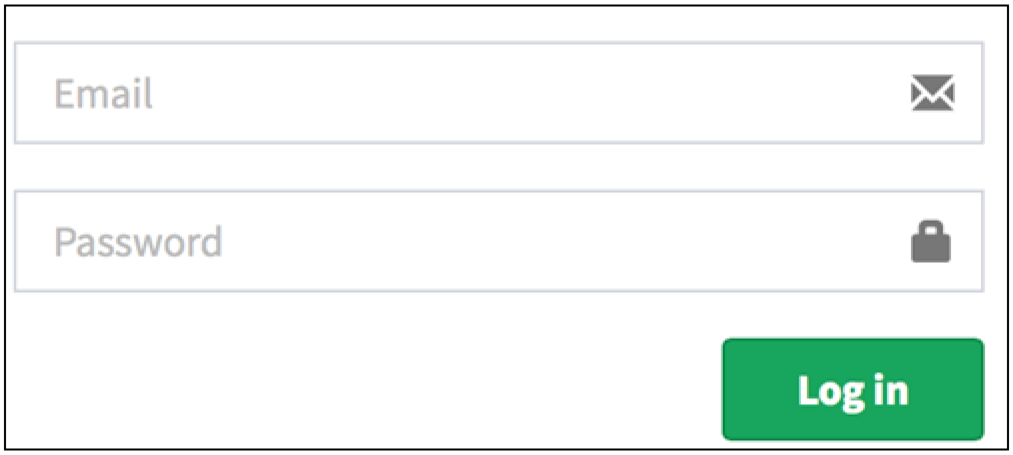
DECISION TABLE TESTING

Decision table testing is a testing technique used to test system behavior for different input combinations. This is a systematic approach where the different input combinations and their corresponding system behavior (Output) are captured in a tabular form. That is why it is also called as a Cause-Effect table where Cause and effects are captured for better test coverage.

A Decision Table is a tabular representation of inputs versus rules/cases/test conditions. Let's learn with an example.

Example 1: Decision Base Table for Login Screen

Let's create a decision table for a login screen.



The condition is simple if the user provides correct username and password the user will be redirected to the homepage. If any of the input is wrong, an error message will be displayed.

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Username (T/F)	F	T	F	T
Password (T/F)	F	F	T	T
Output (E/H)	E	E	E	H

ADVANTAGE OF DECISION TABLE TECHNIQUE:

1. Any complex business flow can be easily converted into the test scenarios & test cases using this technique.
2. Such type of table are work iteratively, means the table created at the first iteration is used as input table for next tables. Such iteration can be carried out only if the initial table is unsatisfactory.
3. Simple to understand and everyone can use this method design the test scenarios & test cases.
4. It provide complete coverage of test cases which help to reduce the rework on writing test scenarios & test cases.
5. These tables guarantee that we consider every possible combination of

condition values. This is known as its “completeness property”.

CAUSE AND EFFECT GRAPH TESTING TECHNIQUE

Cause-Effect Graph graphically shows the connection between a given outcome and all issues that manipulate the outcome. Cause Effect Graph is a black box testing technique. It is also known as Ishikawa diagram because of the way it looks, invented by Kaoru Ishikawa or fish bone diagram.

It is generally uses 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.

A “Cause” stands for a separate input condition that fetches about an internal change in the system. An “Effect” represents an output condition, a system transformation or a state resulting from a combination of causes.

THE CAUSE-EFFECT DIAGRAM CAN BE USED UNDER THESE CIRCUMSTANCES:

- To determine the current problem so that right decision can be taken very fast.
- To narrate the connections of the system with the factors affecting a particular process or effect.

- To recognize the probable root causes, the cause for a exact effect, problem, or outcome.

BENEFITS OF MAKING CAUSE-EFFECT DIAGRAM

- It finds out the areas where data is collected for additional study.
- It motivates team contribution and uses the team data of the process.
- Uses synchronize and easy to read format to diagram cause-and-effect relationships.
- Point out probable reasons of difference in a process.
- It enhances facts of the procedure by helping everyone to learn more about the factors at work and how they relate.
- It assists us to decide the root reasons of a problem or quality using a structured approach.

STEPS TO PROCEED ON CAUSE-EFFECT DIAGRAM:

Firstly: Recognize and describe the input conditions (causes) and actions (effect)

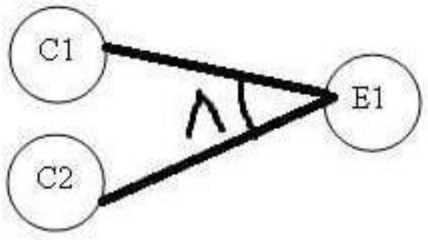
Secondly: Build up a cause-effect graph

Third: Convert cause-effect graph into a decision table

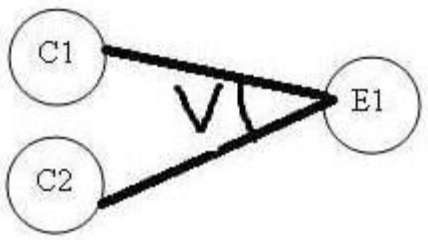
Fourth: Convert decision table rules to test cases. Each column of the decision table represents a test case

SYMBOLS USED IN CAUSE-EFFECT GRAPHS:

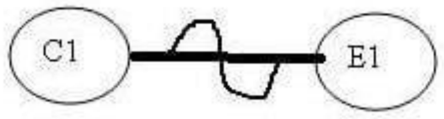
AND – For effect E1 to be true, both the causes C1 and C2 should be true



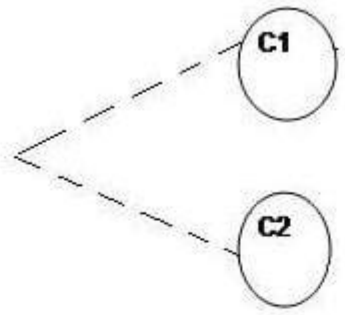
OR – For effect E1 to be true, either of causes C1 OR C2 should be true



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



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



LET’S DRAW A CAUSE AND EFFECT GRAPH BASED ON A SITUATION

***Situation*:**

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

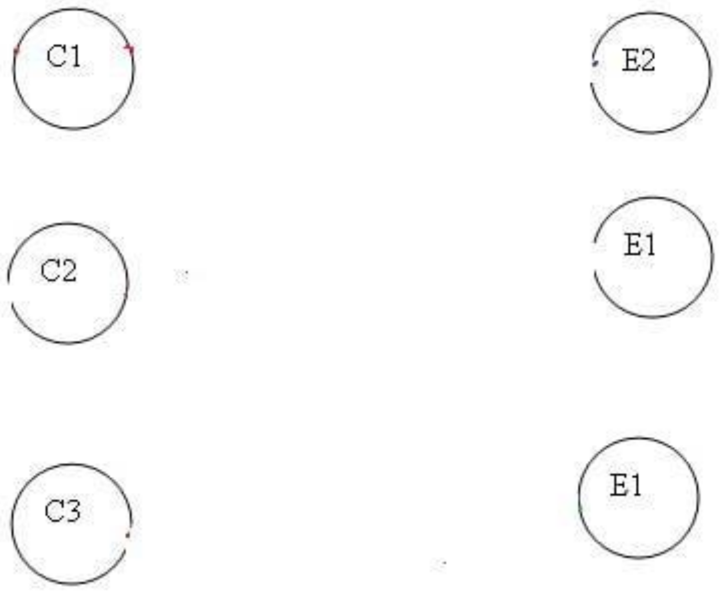
- The first character must be an “A” or a “B”.
- The second character must be a digit.
- If the first character is an “A” or “B” and the second character is a digit, the file must be updated.
- If the first character is incorrect (not an “A” or “B”), the message X must be printed.
- If the second character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

Solution:
The causes for this situation are:
C1 – First character is A
C2 – First character is B
C3 – the Second character is a digit

The effects (results) for this situation are
E1 – Update the file
E2 – Print message “X”
E3 – Print message “Y”

LET’S START!!

First, draw the causes and effects as shown below:



Key – Always go from effect to cause (left to right). That means, to get effect “E”, what causes should be true.

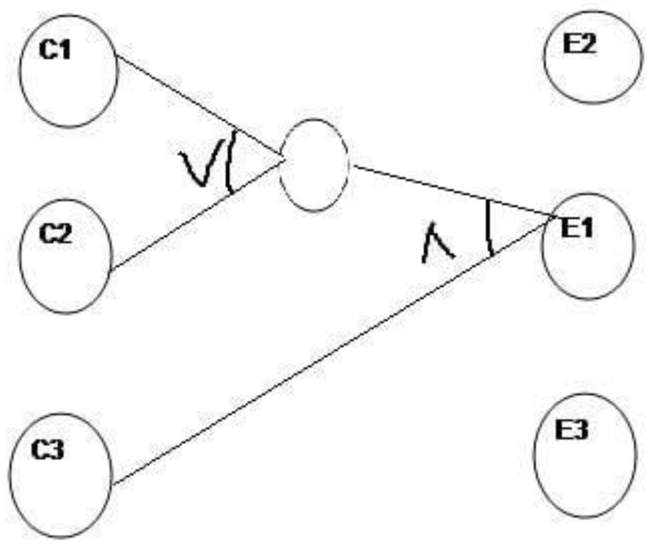
In this example, let’s start with Effect E1.

Effect E1 is to update the file. The file is updated when
- The first character is "A" and the second character is a digit
- The first character is "B" and the second character is a digit
- The first character can either be "A" or "B" and cannot be both.

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

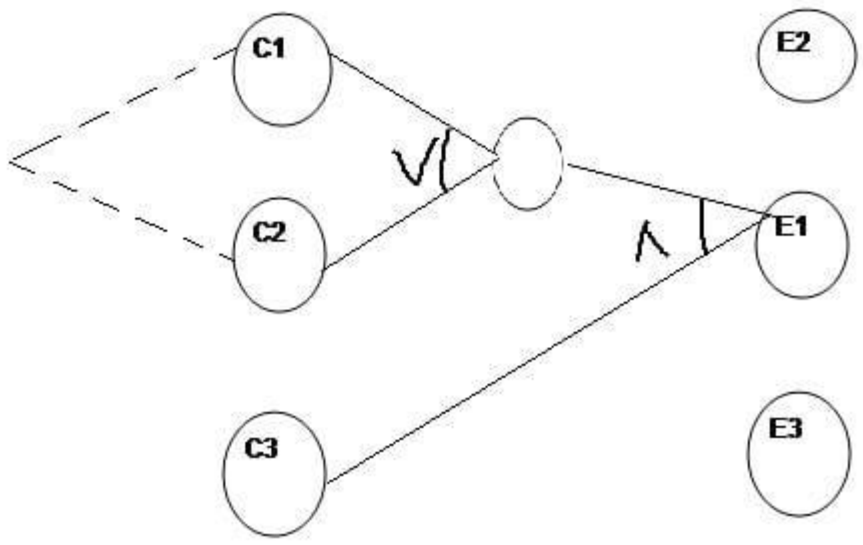
For E1 to be true – following are the causes:
- C1 and C3 should be true
- C2 and C3 should be true
- C1 and C2 cannot be true together. This means C1 and C2 are mutually exclusive.

Now let's draw this:

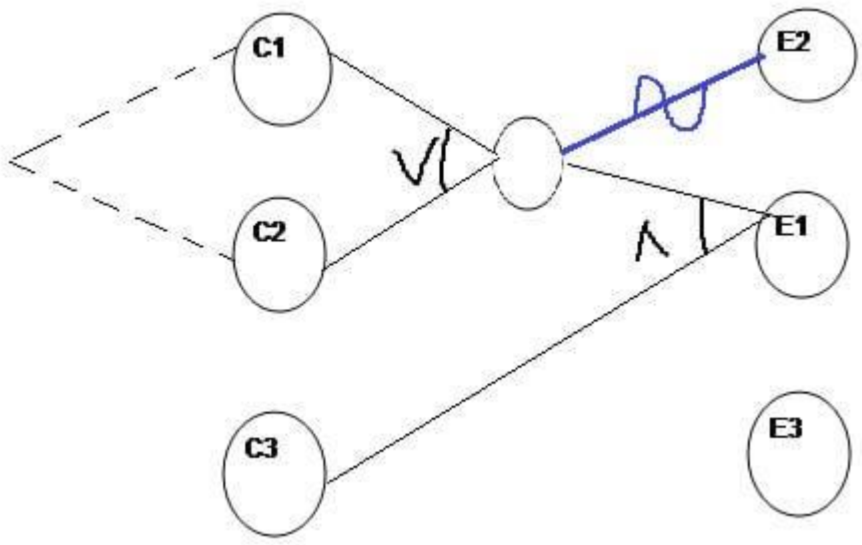


So as per the above diagram, for E1 to be true the condition is
$(C1 \lor C2) \land C3$

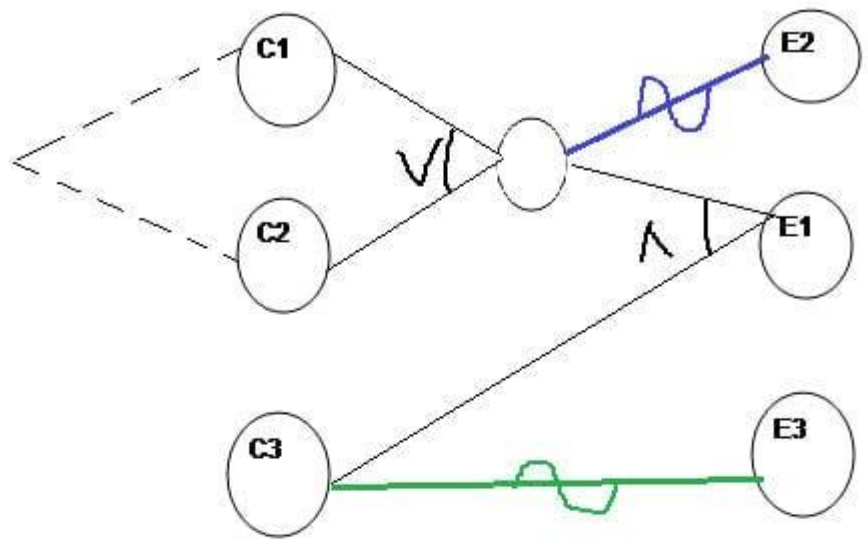
The circle in the middle is just an interpretation of the middle point to make the graph less messy.
There is a third condition where C1 and C2 are mutually exclusive. So the final graph for effect E1 to be true is shown below:



Let’s move to Effect E2:
E2 states to print message “X”. Message X will be printed when the 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)

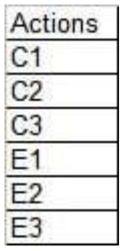


This completes the Cause and Effect graph for the above situation.

Now let's move to draw the **Decision table based on the above graph**.

WRITING DECISION TABLE BASED ON CAUSE AND EFFECT GRAPH

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



Actions
C1
C2
C3
E1
E2
E3

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

Actions	
C1	
C2	
C3	
E1	1
E2	
E3	

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

Actions		
C1	1	
C2		1
C3	1	1
E1	1	1
E2		
E3		

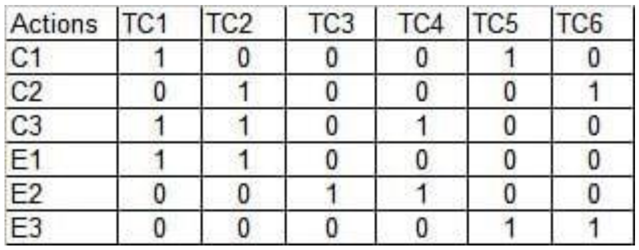
For E2 to be True, either C1 or C2 has to be false shown as

Actions				
C1	1		0	
C2		1		0
C3	1	1	0	1
E1	1	1		
E2			1	1
E3				

For E3 to be true, C3 should be false.

Actions						
C1	1		0		1	
C2		1		0		1
C3	1	1	0	1		
E1	1	1				
E2			1	1		
E3					1	1

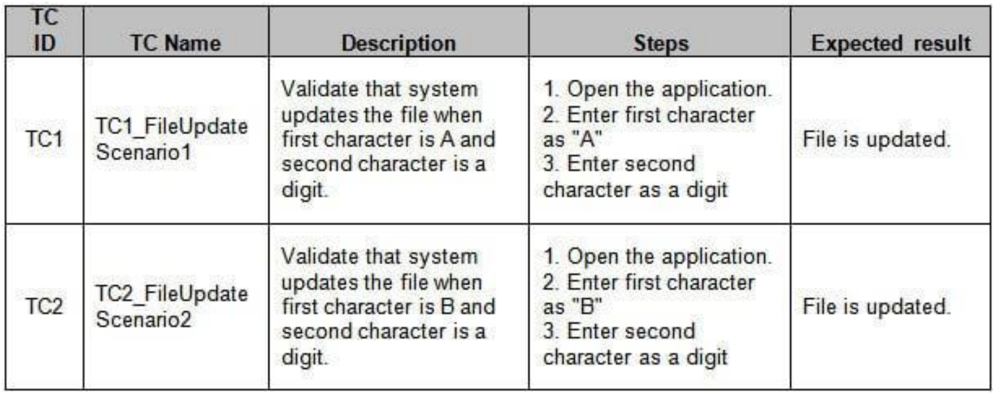
So it’s done. Let’s complete the graph by adding 0 in the blank column and including the test case identifier.



Actions	TC1	TC2	TC3	TC4	TC5	TC6
C1	1	0	0	0	1	0
C2	0	1	0	0	0	1
C3	1	1	0	1	0	0
E1	1	1	0	0	0	0
E2	0	0	1	1	0	0
E3	0	0	0	0	1	1

WRITING TEST CASES FROM THE DECISION TABLE

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



TC ID	TC Name	Description	Steps	Expected result
TC1	TC1_FileUpdate Scenario1	Validate that system updates the file when first character is A and second character is a digit.	1. Open the application. 2. Enter first character as "A" 3. Enter second character as a digit	File is updated.
TC2	TC2_FileUpdate Scenario2	Validate that system updates the file when first character is B and second character is a digit.	1. Open the application. 2. Enter first character as "B" 3. Enter second character as a digit	File is updated.

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)

Conclusion

Summarizing the steps once again:

1. Draw the circles for Causes and Graphs
2. Start from effects and move towards the cause.
3. Look for mutually exclusive causes.

This finishes the Cause and Effect graph dynamic test case writing technique. We have seen how to draw the graph and how to draw the decision table based on it. The final step of writing test cases based on decision table is comparatively easy.

STRUCTURAL TESTING

Structural testing, also known as glass box testing or white box testing is an approach where the tests are derived from the knowledge of the software's structure or internal implementation.

The other names of structural testing includes clear box testing, open box testing, logic driven testing or path driven testing.

PATH TESTING

Path Testing is a structural testing method based on the source code or algorithm and NOT based on the specifications. It can be applied at different levels of granularity.

PATH TESTING ASSUMPTIONS:

- The Specifications are Accurate
- The Data is defined and accessed properly
- There are no defects that exist in the system other than those that affect control flow

PATH TESTING TECHNIQUES:

- **Control Flow Graph (CFG) -** The Program is converted into Flow graphs by representing the code into nodes, regions and edges.
- **Decision to Decision path (D-D) -** The CFG can be broken into various Decision to Decision paths and then collapsed into individual nodes.
- **Independent (basis) paths -** Independent path is a path through a DD-path graph which cannot be reproduced from other paths by other methods.

DATA FLOW TESTING

Data flow testing is a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of variables or data objects. Dataflow Testing focuses on the points at which variables receive values and the points at which these values are used.

ADVANTAGES OF DATA FLOW TESTING:

Data Flow testing helps us to pinpoint any of the following issues:

- A variable that is declared but never used within the program.
- A variable that is used but never declared.

- A variable that is defined multiple times before it is used.
- Deallocating a variable before it is used.

MUTATION TESTING

Mutation testing is a structural testing technique, which uses the structure of the code to guide the testing process. On a very high level, it is the process of rewriting the source code in small ways in order to remove the redundancies in the source code

These ambiguities might cause failures in the software if not fixed and can easily pass through testing phase undetected.

MUTATION TESTING BENEFITS:

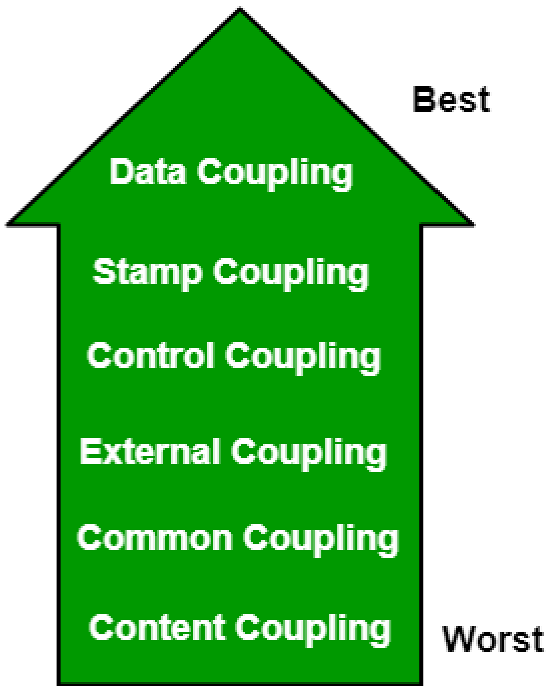
Following benefits are experienced, if mutation testing is adopted:

- It brings a whole new kind of errors to the developer's attention.
- It is the most powerful method to detect hidden defects, which might be impossible to identify using the conventional testing techniques.
- Tools such as Insure++ help us to find defects in the code using the state-of-the-art.
- Increased customer satisfaction index as the product would be less buggy.
- Debugging and Maintaining the product would be more easier than ever.

COUPLING AND COHESION

COUPLING

Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.



TYPES OF COUPLING:

Data Coupling: If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent to each other and communicating through data. Module communications don't contain tramp data. Example-customer billing system.

Stamp Coupling In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice made by the insightful designer, not a lazy programmer.

Control Coupling: If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.

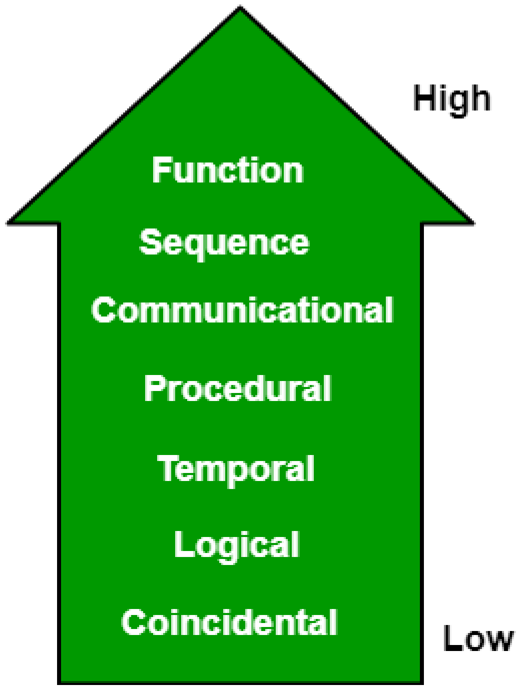
External Coupling: In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex-protocol, external file, device format, etc.

Common Coupling: The modules have shared data such as global data structures.The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses and reduced maintainability.

Content Coupling: In a content coupling, one module can modify the data of another module or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

COHESION

Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.



TYPES OF COHESION:

Functional Cohesion: Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.

Sequential Cohesion: An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.

Communicational Cohesion: Two elements operate on the same input data or contribute towards the same output data. Example- update record int the database and send it to the printer.

Procedural Cohesion: Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.

Temporal Cohesion: The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time-span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at init time.

Logical Cohesion: The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.

Coincidental Cohesion: The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component.

SOFTWARE MAINTENANCE

Necessity of Software Maintenance

Software maintenance is becoming an important activity of a large number of software organizations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product per se, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features. When the hardware platform is changed, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface. For instance, a software product may need to be maintained when the operating system changes. Thus, every software product continues to evolve after its development through maintenance efforts. Therefore it can be stated that software maintenance is needed to correct errors, enhance features, port the software to new platforms, etc.

Types of software maintenance

There are basically three types of software maintenance. These are:

- **Corrective:** Corrective maintenance of a software product is necessary to rectify the bugs observed while the system is in use.
- **Adaptive:** A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.
- **Perfective:** A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

Problems associated with software maintenance

Software maintenance work typically is much more expensive than what it should be and takes more time than required. In software organizations, maintenance work is mostly carried out using ad hoc techniques. The primary reason being that software maintenance is one of the most neglected areas of software engineering. Even though software maintenance is fast becoming an important area of work for many companies as the software products of yester years age, still software maintenance is mostly being carried out as fire-fighting operations, rather than through systematic and planned activities.

Software maintenance has a very poor image in industry. Therefore, an organization often cannot employ bright engineers to carry out maintenance work. Even though maintenance suffers from a poor image, the work involved is often more challenging than development work. During maintenance it is necessary to thoroughly understand someone else's work and then carry out the required modifications and extensions.

Another problem associated with maintenance work is that the majority of software products needing maintenance are legacy products.

Software Reverse Engineering

Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing of its functionalities. A process model for reverse engineering has been shown in fig. 24.1. A program can be reformatted using any of the several available prettyprinter programs which layout the program neatly. Many legacy software products with complex control structure and unthoughtful variable names are difficult to comprehend. Assigning meaningful variable names is important because meaningful variable names are the most helpful thing in code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible. Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.

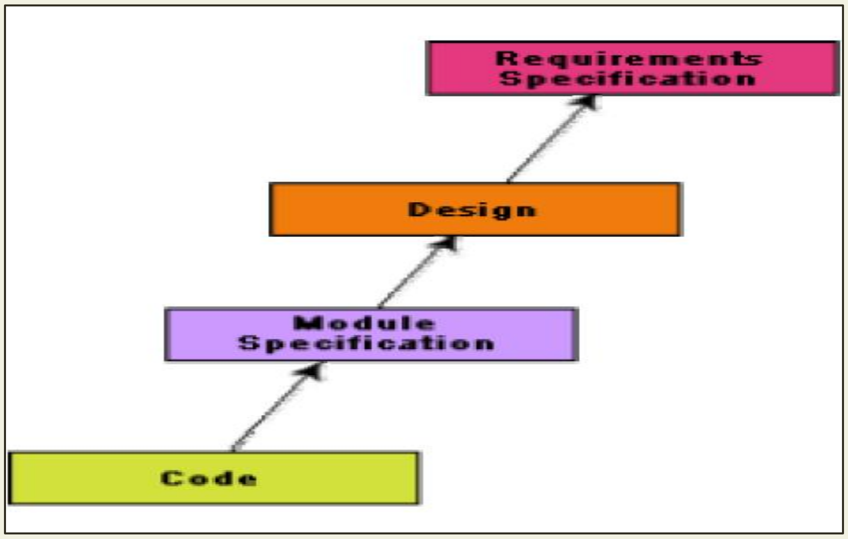


Fig. 24.1: A process model for reverse engineering

After the cosmetic changes have been carried out on a legacy software the process of extracting the code, design, and the requirements specification can begin. These activities are schematically shown in fig. 24.2. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code. The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.

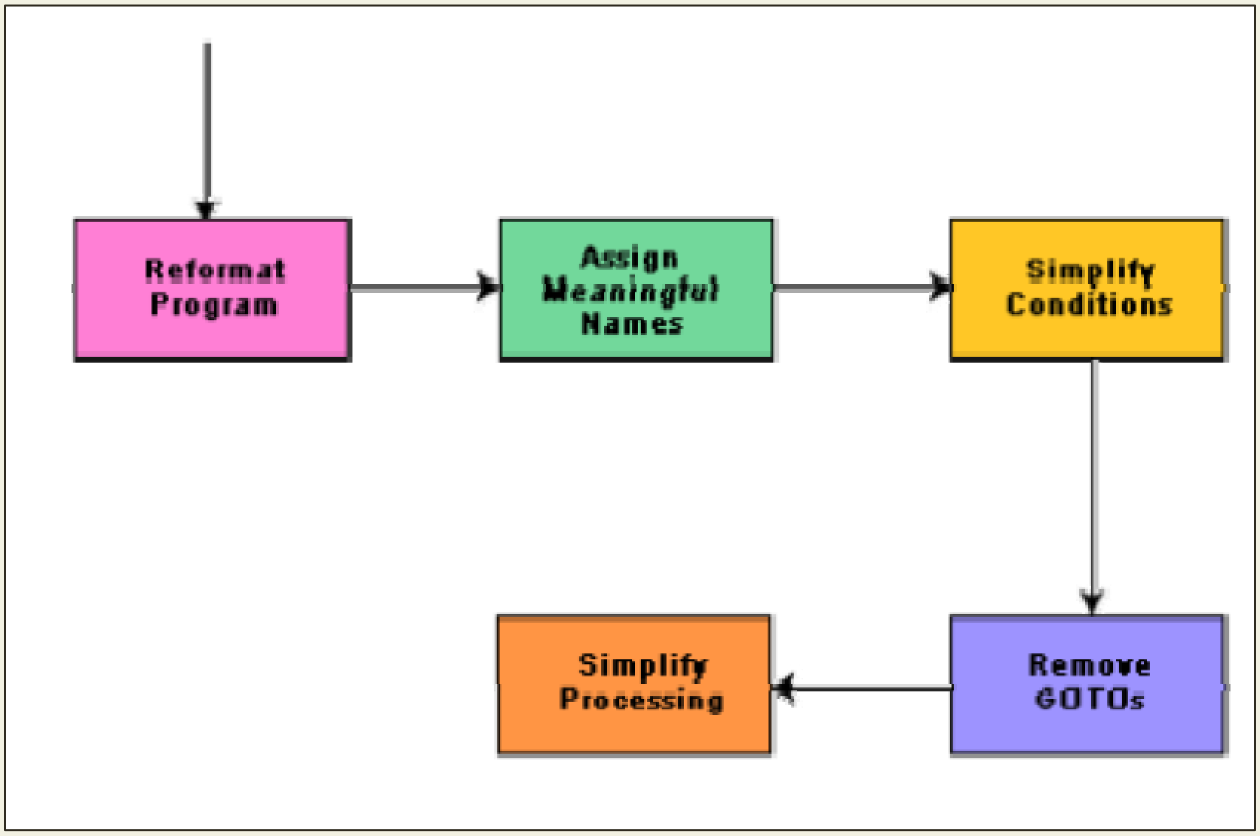


Fig. 24.2: Cosmetic changes carried out before reverse engineering

Legacy software products

It is prudent to define a legacy system as any software system that is hard to maintain. The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product. Many of the legacy systems were developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

The activities involved in a software maintenance project are not unique and depend on several factors such as:

- the extent of modification to the product required

- the resources available to the maintenance team
- the conditions of the existing product (e.g., how structured it is, how well documented it is, etc.)
- the expected project risks, etc.

When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents. But more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

SOFTWARE MAINTENANCE PROCESS MODELS

Two broad categories of process models for software maintenance can be proposed. The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later. This maintenance process is graphically presented in fig. 25.1. In this approach, the project starts by gathering the requirements for changes. The requirements are next analyzed to formulate the strategies to be adopted for code change. At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle team, especially for projects involving unstructured and inadequately documented code. The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system. Also, debugging of the reengineered system becomes easier as the program traces of both the systems can be compared to localize the bugs.

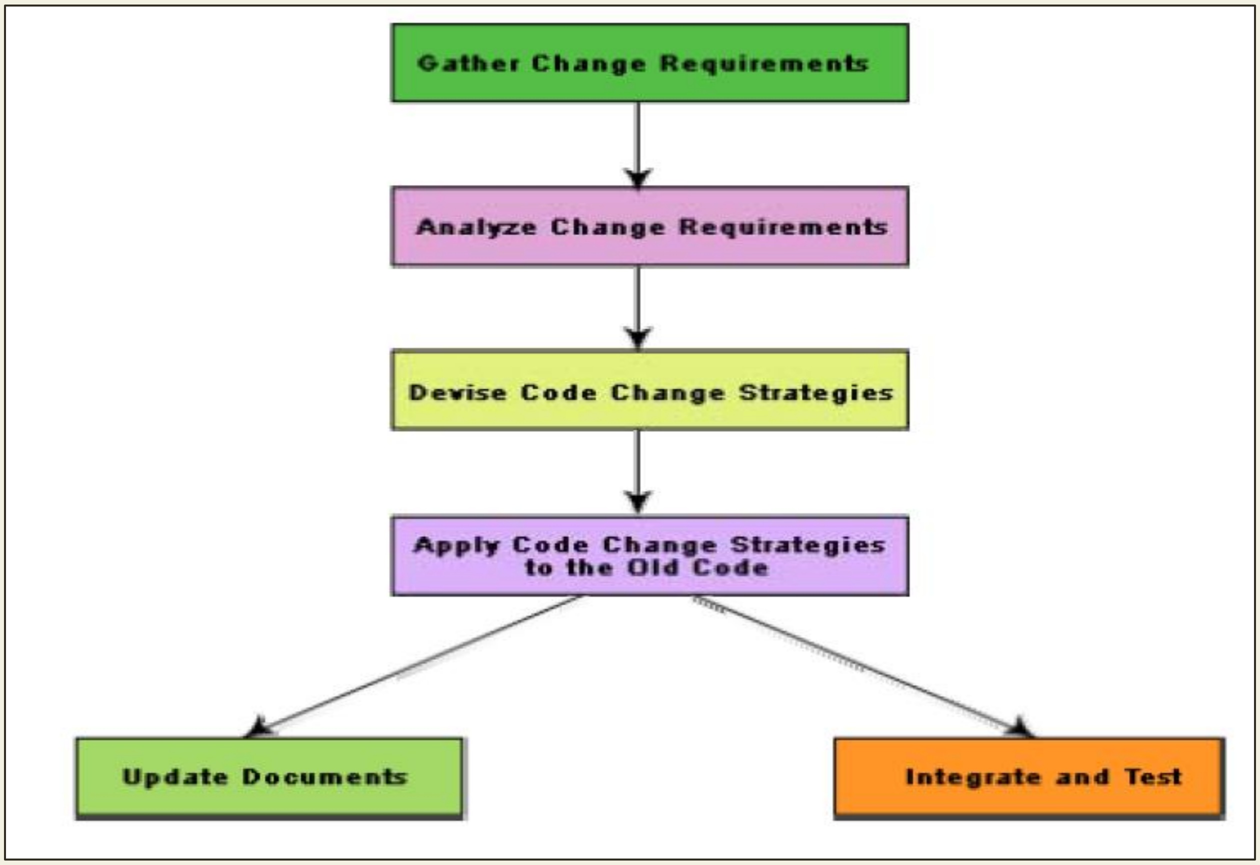

Gather Change Requirements
Analyze Change Requirements
Devise Code Change Strategies
Apply Code Change Strategies to the Old Code
Update Documents
Integrate and Test

Fig. 25.1: Maintenance process model 1

The second process model for software maintenance is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as software reengineering. This process model is depicted in fig. 25.2. The reverse engineering cycle is required for legacy products. During the reverse engineering, the old code is analyzed (abstracted) to extract the module specifications. The module specifications are then analyzed to produce the design. The design is analyzed (abstracted) to produce the original requirements specification. The change requests are then applied to this requirements specification to arrive at the new requirements specification. At the design, module specification, and coding a substantial reuse is made from the reverse engineered products. An important advantage of this approach is that it produces a more structured design compared to what the original product had, produces good documentation, and very often results in increased efficiency. The efficiency improvements are brought about by a more efficient design. However, this approach is more costly than the first approach. An empirical study indicates that process 1 is preferable when the amount of rework is no more than 15%. Besides the amount of rework, several other factors might affect the decision regarding using process model 1 over process model 2:

- Reengineering might be preferable for products which exhibit a high failure rate.
- Reengineering might also be preferable for legacy products having poor design and code structure.

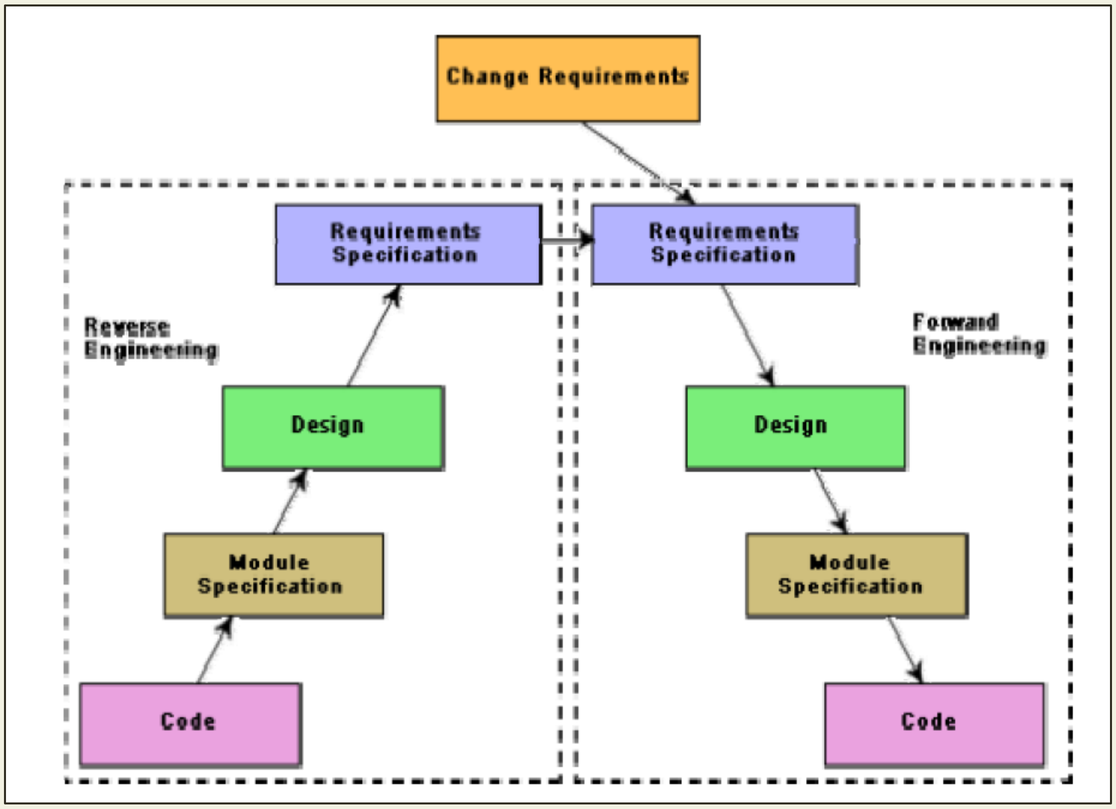


Fig. 25.2: Maintenance process model 2

Software Reengineering

Software reengineering is a combination of two consecutive processes i.e. software reverse engineering and software forward engineering as shown in the fig. 25.2.

Estimation of approximate maintenance cost

It is well known that maintenance efforts require about 60% of the total life cycle cost for a typical software product. However, maintenance costs vary widely from one application domain to another. For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost.

Boehm [1981] proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model. Boehm's maintenance cost estimation is made in terms of a quantity called the Annual Change Traffic (ACT). Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion.

$$\text{ACT} = \frac{\text{KLOC}_{\text{added}} + \text{KLOC}_{\text{deleted}}}{\text{KLOC}_{\text{total}}}$$

where, $KLOC_{added}$ is the total kilo lines of source code added during maintenance.

$KLOC_{deleted}$ is the total kilo lines of source code deleted during maintenance.

Thus, the code that is changed, should be counted in both the code added and the code deleted. The annual change traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost:

$$\text{maintenance cost} = ACT \times \text{development cost.}$$

Most maintenance cost estimation models, however, yield only approximate results because they do not take into account several factors such as experience level of the engineers, and familiarity of the engineers with the product, hardware requirements, software complexity, etc.

Content Feedback

Feel free to submit your issues, suggestions and ratings regarding this course content

Click Here

Join & Share WhatsApp Group of COLLATE

(Click the button to join group)

MSIT

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

MAIT

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

ADGITM (NIEC)

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

GTBIT

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

BVP

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

BPIT

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

HMR

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	