Black Box Testing:

It is also called Behavioral/Specification-Based/Input-Output Testing. Black Box Testing is a software testing method in which testers evaluate the functionality of the software under test without looking at the internal code structure. Black box testing is the Software testing method which is used to test the software without knowing the internal structure of code or program. It is not performed early in the testing process rather it tends to be applied during later stage of testing. Attention is focus on the information domain.

It is carried out to test functionality of the program. The tester in this case, has a set of input values and respective desired results. On providing input, if the output matches with the desired results, the program is tested 'ok', and problematic otherwise. This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see

It find errors in the ff categories

1- Incorrect or missing functions
2- Interface error
3- Errors in data structures or external database access
4- Behavior or performance errors
5- Initialization or termination errors

6- 

By applying black-box techniques, we derive a set of test cases that satisfy the following criteria
– Test cases that reduce the number of additional test cases that must be designed to achieve reasonable testing (i.e minimize effort and time)
– Test cases that tell us something about the presence or absence of classes of errors

Types of Black Box Testing:

1. Functionality Testing0
2. Non-functionality Testing

Functional Testing:

In simple words, what the system actually does is functional testing. To verify that each function of the software application behaves as specified in the requirement document. Testing all the functionalities by providing appropriate input to verify whether the actual output is matching the expected output or not. It falls within the scope **of black-box testing** and the testers need not concern about the source code of the application. Checking the log in functionality- provide username and password

Non-functional Testing:

In simple words, how well the system performs is non-functionality testing. Non-functional testing refers to various aspects of the software such as performance, load, stress, scalability, security, compatibility, etc., The Main focus is to improve the user experience on how fast the system responds to a request. Loading of a webpage (Dashboard) in some seconds- customize how it can be loaded

Characteristics of Black-box testing:
– Program is treated as a black box.
– Implementation details do not matter.
– Requires an end-user perspective.
– Criteria are not precise.
– Test planning can begin early.

BLACK BOX TESTING TECHNIQUES

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

1. ***Equivalence partitioning****:* It is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data. it is the process of taking all possible test cases and placing them into classes. One test value is picked from each class while testing.

Equivalence partitioning is a testing technique where input values set into classes for testing.

- Valid Input Class = Keeps all valid inputs.
- Invalid Input Class = Keeps all Invalid inputs.

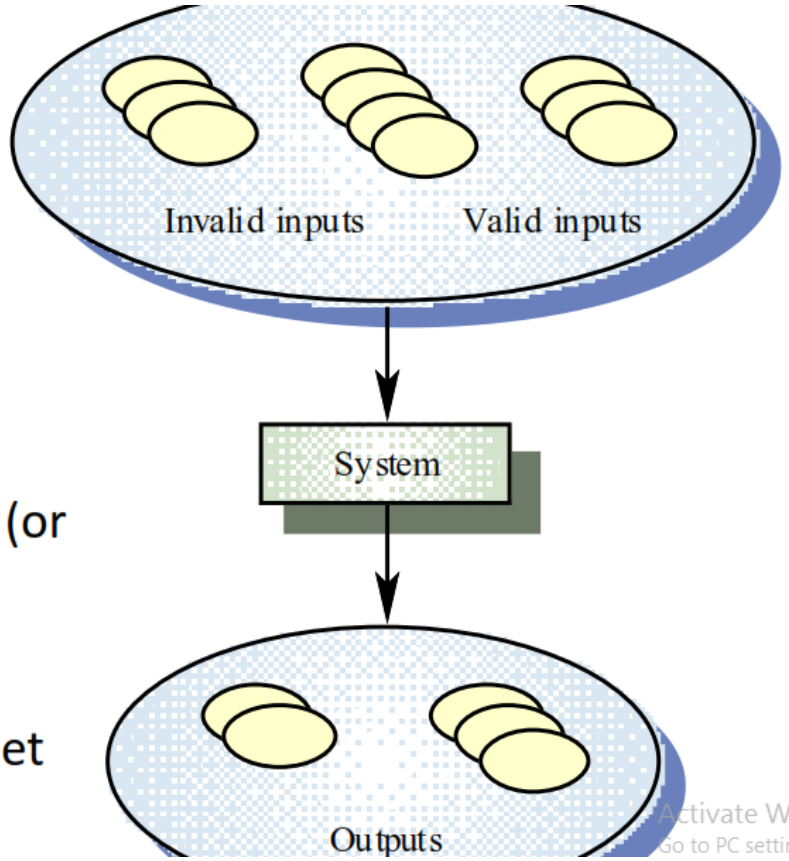
It is regarded as the black box testing that plays an important role in the Software Testing Life Cycle. The testers who perform the testing give assurance that the results are accurate and also cover very limited amount of time in doing the input scenarios.

- It divides the input data of software into different equivalence data classes.
- You can apply this technique, where there is a range in the input field.

For example, age box that can contain age in the range of 20-40, then
valid input values can be 21, 25, 30, 39, etc. and
invalid input values can be any value less than 20 or greater than 40 like 10,15,45,55, etc.

Equivalence Partitioning

- Equivalence partitioning is the process of methodically reducing the huge (or infinite) set of possible test cases into a small, but equally effective, set of test cases.



Boundary Value Analysis: It is a software test design technique that involves determination of boundaries for input values and selecting values that are at the boundaries and just inside/ outside of the boundaries as test data. Boundary value testing basically focuses on boundaries values. It evaluates whether or not a certain range of values would be acceptable by the system or not.

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.

BVA is a test case design technique that complement Equivalence pat. Rather than selecting any element of an equivalent class BVA Lead to selection of test case at the edge of the class.

For Instance: Test field accepts valid User Name and Password field to that accepts minimum 8 characters and maximum 12 characters.

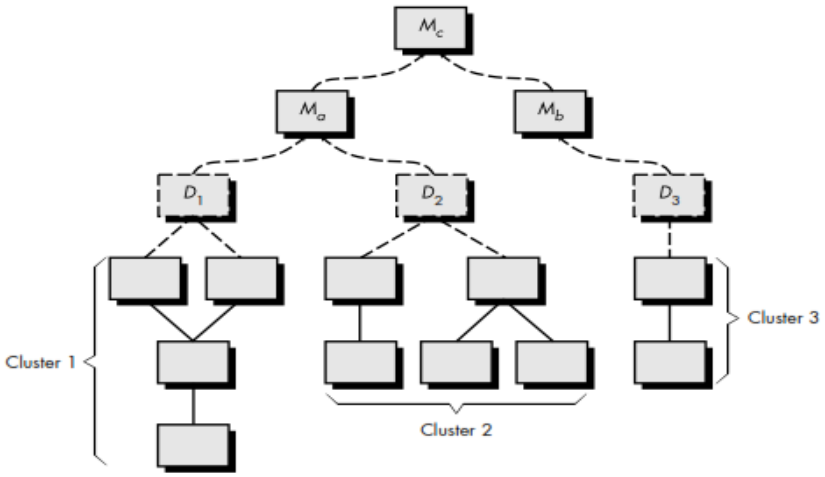
valid range 8-12, Invalid range 7 or less than 7 and *invalid range* 13 or more than 13.

- Test Cases 1: password length less than 8.
- Test Cases 2: password of length exactly 8.
- Test Cases 3: password of length between 9 and 11.
- Test Cases 4: password of length exactly 12.
- Test Cases 5: password of length more than 12.

Equivalence partitioning and boundary value analysis (BVA) are closely related and can be used together at all levels of testing.

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

FIGURE 18.7
Bottom-up integration



Decision Table Testing

A decision table demonstrates causes and their simultaneous effects in the form of a matrix. In decision table testing, there exists a unique combination in each column.

Error Guessing

This technique is mainly based on experience. Once a tester has experience working on any application its behavior and functionalities are known to him/her. This is a way through which many issues can be found out. By using this experience, it is easy for the testers to guess where most developers are prone to make mistakes. These can be handling null values, accepting the submit button without any value, uploading a file without any attachment, uploading a file with less than or more than the limit size specified, etc.

Cause Effect Graphing **Cause-effect graphing** - In both previous methods, only one input value at a time is tested. Cause (input) – Effect (output) is a testing technique where combinations of input values are tested in a systematic way.

BLACK BOX TESTING ADVANTAGES

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

BLACK BOX TESTING DISADVANTAGES

- Only a small number of possible inputs can be tested and many program paths will be left untested.
- Without clear specifications, which is the situation in many projects, test cases will be difficult to design.
- Tests can be redundant if the software designer/ developer has already run a test case.

incorrect assumptions

White-Box Testing

White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops. The "status of the program" may be examined at various points to determine if the expected or asserted status corresponds o the actual status.

Why spend time and energy worrying about(and testing) logics when we might better expend effort ensuring that program requirements have been met?"
OR
• *Why don't we spend all of our energy on black-box tests?*
The answer lies in the nature of software defects:
– Logic errors and incorrect assumptions are inversely proportional to the probability that aprogram path will be executed
– We often believe that a logical path is not likely to be executed when, in fact, it may be executed on aregular basis.
– Typographical errors are random
• Each of these reasons provides an argument for conducting white-box tests.– We often believe
.

Code Coverage Testing

Code coverage is a measure which describes the degree of which the source code of the program has been tested. It is one form of white box testing which finds the areas of the program not exercised by a set of test cases. It also creates some test cases to increase coverage and determining a quantitative measure of code coverage..

Since a product is realized in terms of program code, if we can run test cases to exercise the different parts of the code, then that part of the product realized by the code gets tested. Code coverage testing involves designing and executing test cases and finding out the percentage of code that is covered by testing. The percentage of code covered by a test is found by adopting a technique called *instrumentation* of code. There are specialized tools available to achieve instrumentation. Instrumentation rebuilds the product, linking the product with a set of libraries provided by the tool

Code Coverage Testing is made up of the following types of coverage

1. Statement coverage
2. Path coverage
3. Condition coverage

Statement coverage

The statement coverage strategy aims to design test cases so that every statement in a program is
executed at least once. The principal idea governing the statement coverage strategy is that
unless a statement is executed, it is very hard to determine if an error exists in that statement.

Unless a statement is executed,

Even if we were to achieve a very high level of statement coverage, it does not mean that the program is defect-free. First, consider a hypothetical case when we achieved 100 percent code coverage. If the program implements wrong requirements and this wrongly implemented code is "fully tested," with 100 percent code coverage, it still is a wrong program and hence the 100 percent code coverage does not mean anything.

Statement coverage

• Programs constructs can be usually classified
as:

1. Sequential control flow
2. Two way decision statements (if then else)
3. Multi way decision statements (switch)
4. Loops (while, do, repeat until, for)

```
Prints (int a, int b) {                    ----------  Print sum is a function
   int result = a+ b;
   If (result> 0)
         Print ("Positive", result)
   Else
         Print ("Negative", result)
   }
```

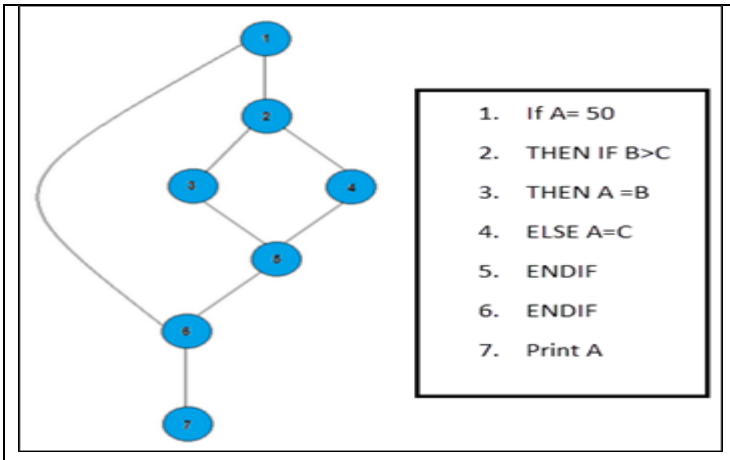
Scenario 1: If A = 3, B = 9 Number of executed statements = 5, Total number of statements = 7 Statement Coverage: **5/7 = 71%**	scenario 2, **Scenario 2:** If A No of executed statements = 6 Total no of statements = 7 $\text{Statement Coverage} = \frac{\textit{Number of executed statments}}{\textit{Total number of statments}}$ Statement Coverage: 6/7 = 85%

Path coverage

What is Path Testing?

Path testing is a structural testing method that involves using the source code of a program in order to find every possible executable path. It helps to determine all faults lying within a piece of code. This method is designed to execute all or selected path through a computer program.

Any software program includes, multiple entry and exit points. Testing each of these points is a challenging as well as time-consuming. In order to reduce the redundant tests and to achieve maximum test coverage. Basis path testing involves execution of all possible blocks in a program and achieves maximum path coverage with the least number of test cases.

	Here there are 3 paths or condition that need to be tested to get the output, • **Path 1**: 1,2,3,5,6, 7 • **Path 2**: 1,2,4,5,6, 7 • **Path 3**: 1, 6, 7

Steps for Basis Path testing

The basic steps involved in basis path testing include

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

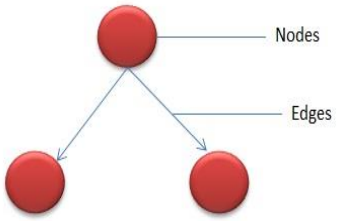
Cyclomatic Complexity in Software Testing

Cyclomatic Complexity in Software Testing is a testing metric used for measuring the complexity of a software program. It is a quantitative measure of independent paths in the source code of a software program. Cyclomatic complexity can be calculated by using control flow graphs or with respect to functions, modules, methods or classes within a software program.

Independent path is defined as a path that has at least one edge which has not been traversed before in any other paths.

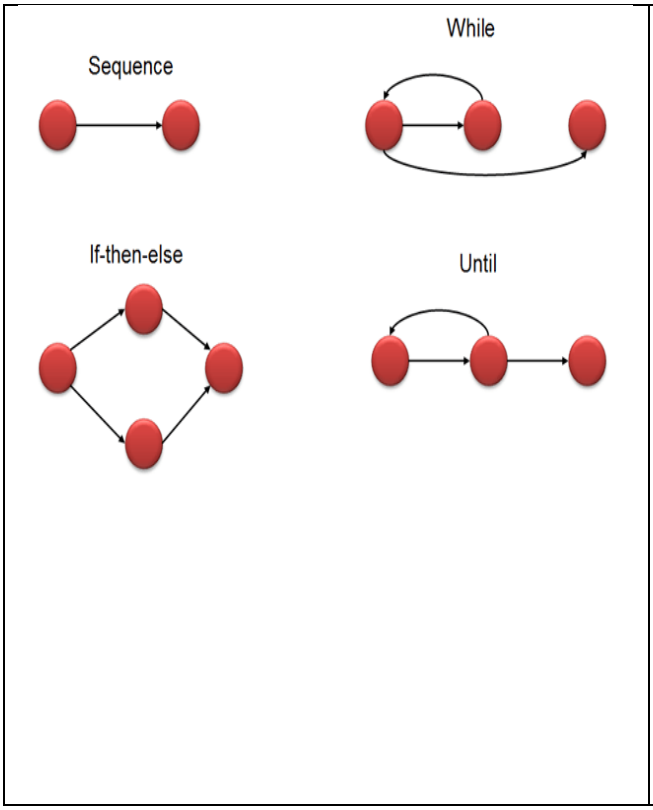
This metric was developed by Thomas J. McCabe in 1976 and it is based on a control flow representation of the program. Control flow depicts a program as a graph which consists of Nodes and Edges.

In the graph, Nodes represent processing tasks while edges represent control flow between the nodes.

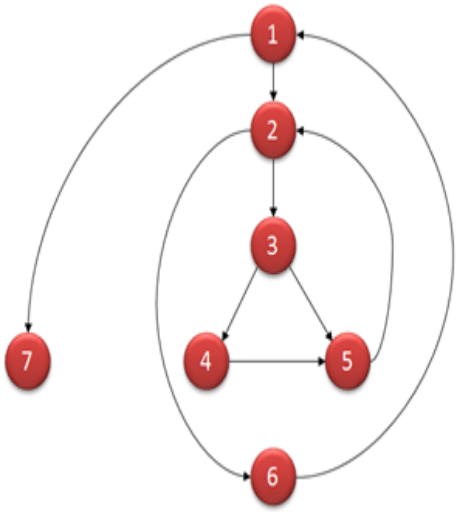


Flow graph notation for a program:

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



Flow graph for this program will be



Computing mathematically,

- V(G) = 9 – 7 + 2 = 4
- V(G) = 3 + 1 = 4 (Condition nodes are 1,2 and 3 nodes)
- Basis Set – A set of possible execution path of a program
- 1, 7
- 1, 2, 6, 1, 7
- 1, 2, 3, 4, 5, 2, 6, 1, 7

1, 2, 3, 5, 2, 6, 1, 7

How to Calculate Cyclomatic Complexity

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

V(G) = E - N + 2
Where,

E – Number of edges N – Number of Nodes

V (G) = P + 1
Where P = Number of predicate nodes (node that contains condition)

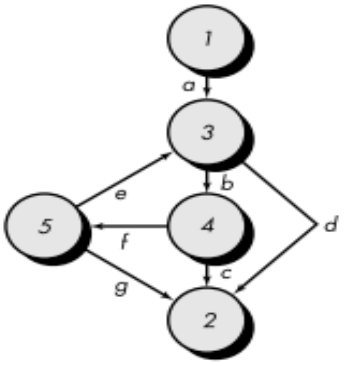
Graph Matrices

The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. To develop a software tool that assists in basis path testing, a data structure, called a *graph matrix,* can be quite useful.

A *graph matrix* is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds o an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix

Referring to the figure below, *each node on the flow graph is identified by numbers,* while *each edge is identified by letters*. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge *b.*

FIGURE 17.6
Graph matrix



Flow graph

Node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

FIGURE 17.7
Connection matrix

Node	1	2	3	4	5	Connections
1			1			1 – 1 = 0
2						
3		1		1		2 – 1 = 1
4		1			1	2 – 1 = 1
5		1	1			2 – 1 = 1

Graph matrix

3 + 1 = 4 ← Cyclomatic complexity

Condition Coverage

Condition Coverage he condition testing method focuses on testing each condition in the program. or also called expression coverage is a testing method used to test and evaluate the variables or sub-expressions in the conditional statement. The goal of condition coverage is to check individual outcomes for each logical condition. The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program

The formula to calculate Condition Coverage:

$$Condition\ Coverage = \frac{Number\ of\ Executed\ Operands}{Total\ Number\ of\ Operands}$$

Example:

```
1   IF (x < y) AND (a>b) THEN
```

For the above expression, we have 4 possible combinations

- TT FF TF FT

Consider the following input

X=3 Y=4	(x<y)	TRUE	Condition Coverage is ¼ = 25%
A=3 B=4	(a>b)	FALSE	

Debugging

Correcting the error

Testing Documentation
Testing documents are prepared at different stages -
Before Testing
Testing starts with test cases generation. Following documents are needed for reference –
 SRS document - Functional Requirements document
 Test Policy document - This describes how far testing should take place before releasing the product.
 Test Strategy document - This mentions detail aspects of test team, responsibility matrix and rights/responsibility of test manager and test engineer.
 Traceability Matrix document - This is SDLC document, which is related to requirement gathering process. As new requirements come, they are added to this matrix. These matrices help testers know the source of requirement. They can be traced forward and backward.
While Being Tested
The following documents may be required while testing is started and is being done:
 Test Case document - This document contains list of tests required to be conducted. It includes Unit test plan, Integration test plan, System test plan

and Acceptance test plan.
 Test description - This document is a detailed description of all test cases and procedures to execute them.
 Test case report - This document contains test case report as a result of the test.
 Test logs - This document contains test logs for every test case report.
After Testing
The following documents may be generated after testing :
 Test summary - This test summary is collective analysis of all test reports and logs. It summarizes and concludes if the software is ready to be launched. The software is released under version control system if it is ready to launch.

ogic errors and

oftware requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
1
1 It is important to note that quality extends to the technical attributes of the analysis, design, and
code models. Models that exhibit high quality (in the technical sense) will lead to software that
exhibits high quality from the customer's point of view.
It's interesting to note that McCall's quality factors are as valid today as they were when they were first proposed in the 1970s. Therefore, it's reasonable to assert that the factors that affect software quality do not change.
FIGURE 19.1
McCall's software

quality factors

2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality
will almost surely result.
3. There is a set of implicit requirements that often goes unmentioned (e.g., the desire for ease of use). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.
Software quality is a complex mix of factors that will vary across different applications and the customers who request them. In the sections that follow, software quality
factors are identified and the human activities required to achieve them are described

nctionality, usability, reliability, performance, and supportability. The FURPS quality
factors draw
liberally from earlier work, defining the following attributes for each of
the five major factors:
• *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
• *Usability* is assessed by considering human factors (Chapter 15), overall aesthetics, consistency,
and documentation.
• *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
• *Performance* is measured by processing speed, response time, resource consumption,
throughput, and efficiency.
Usability
Interoperability
and metrics

Reusability
x
x
x
x
x
x
x
x
x
x
x
“Any activity becomes
creative when the
doer cares about
doing it right, or
better.”
John Updike

• *Supportability* combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, maintainability—in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration, Chapter 9), the ease with which a system can be installed, and the ease with which problems can be localized.

The FURPS quality factors and attributes just described can be used to establish quality metrics for each step in the software engineering process.

9.1.3 ISO 9126 Quality Factors

The ISO 9126 standard was developed in an attempt to identify the key quality attributes
for computer software. The standard identifies six key quality attributes:

Functionality. The degree to which the software satisfies stated needs as indicated by the following subattributes: suitability,
accuracy,
interoperability,

compliance,
and security.

Reliability. The amount of time that the software is available for use as indicated by the following subattributes: maturity,
fault tolerance,
recoverability.
Usability. The degree to which the software is easy to use as indicated by the following subattributes: understandability, learnability, operability.
Efficiency. The degree to which the software makes optimal use of system resources as indicated by the following subattributes: time behavior, resource behavior.
Maintainability. The ease with which repair may be made to the software as indicated by the following subattributes: analyzability, changeability, stability, testability.
Portability. The ease with which the software can be transposed from one environment to another as indicated by the following subattributes: adaptability, installability,
conformance,
replaceability.
Like other software quality factors discussed in Sections 19.1.1 and 19.1.2, the ISO 9126 factors do not necessarily lend themselves to direct measurement. However, they do provide a worthwhile basis for indirect measures and an excellent checklist for assessing the quality of a system.