

Software Engineering - I

An Introduction to Software Construction Techniques for Industrial Strength Software

Chapter 11 – Software Verification and Validation

Software Testing

To understand the concept of software testing correctly, we need to understand a few related concepts.

Software verification and validation

Verification and validation are the processes in which we check a product against its specifications and the expectations of the users who will be using it. According to a known software engineering expert Berry Boehm, verification and validation are

Verification

- Does the product meet system specifications?
- Have you built the product right?

Validation

- Does the product meet user expectations?
- Have you built the right product?

It is possible that a software application may fulfill its specifications but it may deviate from users expectations or their desired behavior. That means, software is verified but not validated. How is it possible? It is possible because during the requirements engineering phase, user needs might not have been captured precisely or the analyst might have missed a major stakeholder in the analysis. Therefore, it is important to verify as well as validate the software product.

Defect

The second major and a very important concept is Defect. A defect is a variance from a desired product attribute. These attributes may involve system specifications well as user expectation. Anything that may cause customer dissatisfaction, is a defect. Whether these defects are in system specifications or in the software products, it is essential to point these out and fix.

Therefore software defect is that phenomenon in which software deviates from its expected behavior. This is non-compliance from the expected behavior with respect to written specifications or the stakeholder needs.

Software and Defect

Software and defects go side by side in the software development life cycle. According to a famous saying by Haliburton, Death and taxes are inevitable. According to Kernighan: Death, taxes, and bugs are the only certainties in the life of a programmer. Software and defects cannot be separated, however, it is important to learn how discovering defects at an appropriate stage improves the software quality. Therefore, software application needs to be verified as well as validated for a successful deployment.

Software Testing

With these concepts, we are in a position to define software testing. Software testing is the process of examining the software product against its requirements. Thus it is a process that involves verification of product with respect to its written requirements and

conformance of requirements with user needs. From another perspective, software testing is the process of executing software product on test data and examining its output vis-à-vis the documented behavior.

Software testing objective

- The correct approach to testing a scientific theory is not to try to verify it, but to seek to refute the theory. That is to prove that it has errors. (Popper 1965)
- The goal of testing is to expose latent defects in a software system before it is put to use.
- A software tester tries to break the system. The objective is to show the presence of a defect not the absence of it.
- Testing cannot show the absence of a defect. It only increases your confidence in the software.
- This is because exhaustive testing of software is not possible – it is simply too expansive and needs virtually infinite resources.

Successful Test

From the following sayings, a successful test can be defined

“If you think your task is to find problems then you will look harder for them than if you think your task is to verify that the program has none” – Myers 1979.

“A test is said to be successful if it discovers an error” – doctor’s analogy.

The success of a test depends upon the ability to discover a bug not in the ability to prove that the software does not have one. As, it is impossible to check all the different scenarios of a software application, however, we can apply techniques that can discover potential bugs from the application. Thus a test that helps in discovering a bug is a successful test. In software testing phase, our emphasis is on discovering all the major bugs that can be identified by running certain test scenarios. However it is important to keep in mind that testing activity has certain limitations.

Limitations of testing

With the help of the following example, we shall see how difficult it may become to discover a defect from a software application.

Example (adapted from Backhouse)

This is a function that compares two strings of characters stored in an array for equality. The tester who has to test this function has devised the following test cases in which different combinations of inputs are tried and their expected behavior is documented. From the look of this table, it seems that almost all major combinations of inputs have been documented.

Inputs and Expected Outputs

a	b	Expected result
“cat”	“dog”	False
“”	“”	True
“hen”	“hen”	True
“hen”	“heN”	False
“”	“”	False
“”	“ball”	False
“cat”	“”	False
“HEN”	“hen”	False
“rat”	“door”	False
“ ”	“ ”	True

Results of testing

The tester runs all the above-mentioned test cases and function returns the same results as expected. But it is still not correct. What can be a problem To analyze the problem, lets look at the code of this string equal routine

Code of the Function

```
bool isStringsEqual(char a[], char b[])
{
    bool result;
    if (strlen(a) != strlen(b))
        result = false;
    else {
        for (int i =0; i < strlen(a); i++)
            if (a[i] == b[i])
                result = true;
            else result = false;
    }
    return result;
}
```

Analysis of the code

It passes all the designated tests but fails for two different strings of same length ending with the same character. For example, “cut” and “rat” would results in true which is not correct.

The above-mentioned defect signifies a clear limitation of the testing process in discovering a defect which is not very frequent. However, it should be noted from this example that a tester cannot generate all possible combinations of test cases to test an application as the number of scenarios may become exhaustive.

Testing limitations

- In order to prove that a formula or hypothesis is incorrect all you have to do to show only one example in which you prove that the formula or theorem is not working.
- On the other hand, million of examples can be developed to support the hypothesis but this will not prove that it is correct.
- These examples only help you in coming up with a hypothesis but they are not proves by themselves and they only enhance your comfort level in that particular hypothesis or in this particular case, in your piece of software.
- You cannot test a program completely because:
 - the domain of the possible inputs is too large to test.
 - there are too many possible paths through the program to test.
- According to Discrete Mathematics
 - To prove that a formula or hypothesis is incorrect you have to show only one example.
 - To prove that it is correct any numbers of examples are insufficient. You have to give a formal proof of its correctness.

Test Cases and Test Data

In order to test a software application, it is necessary to generate test cases and test data which is used in the application. Test cases correspond to application functionality such that the tester writes down steps which should be followed to achieve certain functionality. Thus a test case involves

- Input and output specification plus a statement of the function under test.
- Steps to perform the function
- Expected results that the software application produces

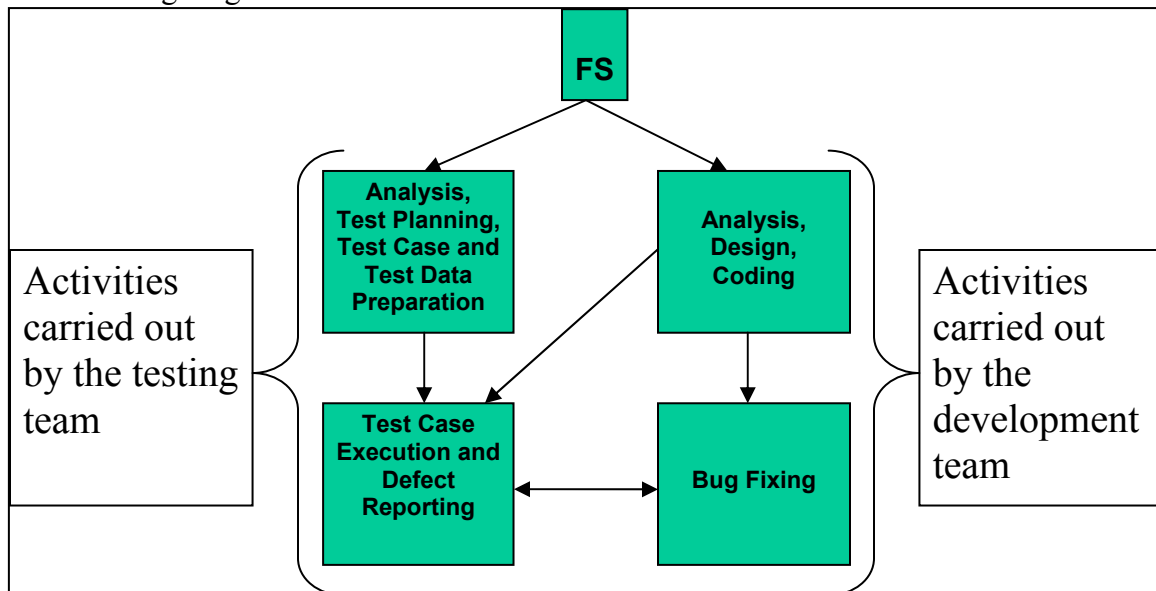
However, test data includes inputs that have been devised to test the system.

Testing vs. development

Testing is an intellectually demanding activity and has a lifecycle parallel to software development. A common misperception about testing is that it is not a challenging activity. It should be noted here that the testing demands grip over the domain and application functionality as is required from an analyst or a designer who has to develop the application from requirements. As without having an in-depth knowledge about the system and the requirements from users, a tester cannot write test cases that can verify and validate software application with respect to documented specifications and user needs. Writing test cases and generating test data are processes that demand scenario-building capabilities. These activities essentially require destructive instincts in a tester for the purpose of breaking system to discover loopholes into its functionality.

At the time when these two activities are being performed, merely initial design of the application is completed. Therefore, tester uses his/her imagination to come up with use patterns of the application that can help him/her in describing exact steps that should be executed in order to test a particular functionality. Moreover, tester needs to figure out loose points in the system from where he/she can discover defects. All these activities are highly imaginative and a tester is supposed to possess above average (if not excellent) analytical skills.

We shall explain the testing activities parallel to development activities with the help of the following diagram



Description

- Functional specification document is the starting point, base document for both testing and the development
- Right side boxes describe the development, whereas, left side boxes explain the testing process
- Development team is involved into the analysis, design and coding activities.
- Whereas, testing team too is busy in analysis of requirements, for test planning, test cases and test data generation.
- System comes into testing after development is completed.
- Test cases are executed with test data and actual results (application behavior) are compared with the expected results,
- Upon discovering defects, tester generates the bug report and sends it to the development team for fixing.
- Development team runs the scenario as described in the bug report and try to reproduce the defect.
- If the defect is reproduced in the development environment, the development team identifies the root cause, fixes it and sends the patch to the testing team along with a bug resolution report.

- Testing team incorporates the fix (checking in), runs the same test case/scenario again and verifies the fix.
- If problem does not appear again testing team closes down the defect, otherwise, it is reported again.

The Developer and Tester

Development	Testing
Development is a creative activity	Testing is a destructive activity
Objective of development is to show that the program works	Objective of testing is to show that the program does not work

Scenarios missed or misunderstood during development analysis would never be tested correctly because the corresponding test cases would either be missing or would be incorrect.

The left side column is related to the development, and the right side describes the testing. Development is a creative process as developers have to build the system, whereas, testing is a destructive activity as the goal of a tester is to break the system to discover the defects. Objective of development is to show that the program works, objective of testing is to show that program does not work. However, FS is the base document for both of these activities.

Tester analyzes FS with respect to testing the system whereas; developer analyzes FS with respect to designing and coding the system. If developer does not understand the FS correctly then he cannot implement and test it right. Thus if the same person who has developed a system, tests it, chances of carrying the same misunderstanding in testing will be very high. Therefore, an independent testing can only prove his understanding wrong. Therefore, it is highly recommended that developer should not try to test his/her own work.

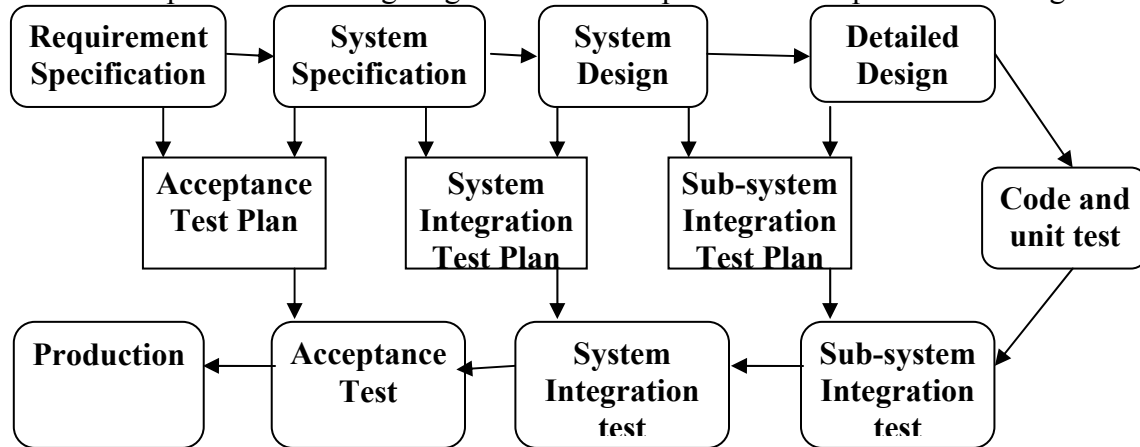
Usefulness of testing

Objective of testing is to discover and fix as many errors as possible before the software is put to use. That is before it is shipped to the client and the client runs it for acceptance. In software development organizations, a rift exists between the development and the testing teams. Often developers are found questioning about the significance or even need to have the testing resources in the project teams. Whoever doubts on the usefulness of the testing team should understand what could happen if the application is delivered to client without testing? At the best, the client may ask to fix all the defects (free of cost) he would discover during the acceptance testing. At the worst, probably he would sue the development firm for damages. However, in practice, clients are often seen complaining about the deliverables and a couple of defected deliverables are sufficient for breaking the relations next to the cancellation of contract.

Therefore, it should be well preserved among the community of developers that testers are essential rather inevitable. A good tester has a knack of smelling errors – just like auditors and it is for the good of the organization not to harm it.

Testing and software phases

With the help of the following diagram we shall explain different phases of testing



Software development process diagram

Description of testing phases

- Unit testing – testing individual components independent of other components.
- Module testing – testing a collection of dependent components – a module encapsulates related components so it can be tested independently.
- Subsystem testing – testing of collection of modules to discover interfacing problems among interacting modules.
- System testing – after integrating subsystems into a system – testing this system as a whole.
- Acceptance test – validation against user expectations. Usually it is done at the client premises.
- Alpha testing – acceptance testing for customized projects, in-house testing for products.
- Beta testing – field testing of product with potential customers who agree to use it and report problem before system is released for general use

In the following two types of testing activities are discussed.

Black box testing

In this type of testing, a component or system is treated as a black box and it is tested for the required behavior. This type of testing is not concerned with how the inputs are transformed into outputs. As the system's internal implementation details are not visible to the tester. He gives inputs using an interface that the system provides and tests the output. If the outputs match with the expected results, system is fine otherwise a defect is found.

Structural testing (white box)

As opposed to black box testing, in structural or white box testing we look inside the system and evaluate what it consists of and how is it implemented. The inner of a system consists of design, structure of code and its documentation etc. Therefore, in white box testing we analyze these internal structures of the program and devise test cases that can test these structures.

Effective testing

The objective of testing is to discover the maximum number of defects with a minimum number of resources before the system is delivered to the next stage. Now the question arises here how to increase the probability of finding a defect?

As, good testing involves much more than just running the program a few times to see whether it works or not. A good tester carries out a thorough analysis of the program to devise test cases that can be used to test the system systematically and effectively.

Problem here is how to develop a representative set of test cases that could test a complete program. That is, selection of a few test cases from a huge set of possibilities. What should be the sets of inputs that should be used to test the system effectively and efficiently?

String Equal Example

- For how many equal strings do I have to test to be in the comfortable zone?
- For how many unequal strings do I have to test to be in the comfortable zone?
- When should I say that further testing is unlikely to discover another error?

Testing types

To answer these questions, we divide a problem domain in different classes. These are called Equivalence Classes.

Equivalence Classes or Equivalence Partitioning

Two tests are considered to be equivalent if it is believed that:

- if one discovers a defect, the other probably will too, and
- if one does not discover a defect, the other probably won't either.

Equivalence classes help you in designing test cases to test the system effectively and efficiently. One should have reasons to believe that the test cases are equivalent. As for this purpose, one would need to understand the system and see in how many partitions it can be divided. These partitions should be devised such that a clear distinction should be marked. Test cases written for one partition should not yield the same results when run for the second partition as otherwise these two partitions should become one. However, finding equivalence classes is a subjective process, as two people analyzing a program will probably come up with different sets of equivalence classes.

Equivalence partitioning guidelines

- Organize your equivalence classes. Write them in some order, use some template, sequence, or group them based on their similarities or distinctions. These partitions can be hierarchical or organized in any other manner.
- Boundary conditions: determine boundary conditions. For example, adding in an empty linked list, adding after the last element, adding before the first element, etc.
- You should not forget invalid inputs that a user can give to a system. For example, widgets on a GUI, numeric instead of alphabets, etc.

Equivalence partitioning example

In the following example, we shall see how equivalence partitions can be developed for a string matching function.

String matching

Organization

For equivalence partitions, we divide the problem in two obvious categories of equal strings and the other one for unequal strings. Within these equivalent partitions, further partitioning is done. Following is the description of the equivalence partitions and their test cases

Test cases for equivalence partitions

Equal

- Two equal strings of arbitrary length
 - All lower case "cat" "cat"
 - All upper case "CAT" "CAT"
 - Mixed case "Cat" "Cat"
 - Numeric values "123" "123"
 - Two strings with blanks only " " " "
 - Numeric and character mixed "Cat1" "Cat1"
 - Strings with special characters "Cat#1" "Cat#1"
- Two NULL strings "" ""

Unequal Strings

- Two different equal strings of arbitrary length
 - Two strings with different length "cat" "mouse"
 - Two strings of same length "cat" "dog"
- Check for case sensitivity
 - Same strings with different characters capitalized "Cat" "caT"
- One string is empty

- First is NULL
- Second is NULL

“” “cat”
“cat” “”

Basis Code Structures

For structural testing it is important to know about basic coding structures. There are four basic coding structures sequence, if statement, case statement, and while loop. These four basic structures can be used to express any type of code.

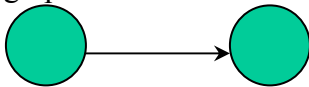
Flow graph notation

In analysis and design, you have already seen the flow graph notation. This is used to describe flow of data or control in an application. However, we do not use flow graphs to describe decisions. That is, how a branch is taken is not shown in flow graphs.

In the following, we are using flow graph notation to describe different coding structures.

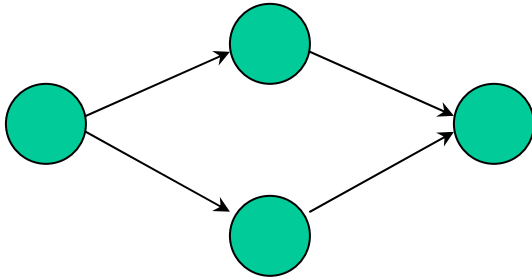
Sequence

Sequence depicts programming instructions that do not have branching or any control information. So we lump together several sequential instructions in one node of the graph.



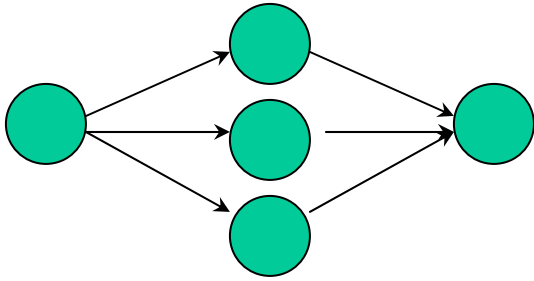
If

Second structural form is the If statement. In the following graph, the first node at the left depicts the if statement and the two nodes next to the first node correspond to the successful case (if condition is true) and unsuccessful case (if condition is false) consecutively. The control comes to the same instruction from either of these intermediate instructions.



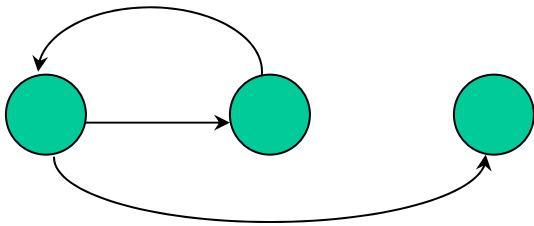
Case

In Case statement, control can take either of several branches (as opposed to only two in If statement.) First node represents the switch statement (C/C++) and nodes in middle correspond to all different cases. Program can take one branch and result into the same instruction.



While

A while loop structure consists of a loop guard instruction through which the iteration in the loop is controlled. The control keeps iterating in the loop as long as the loop guard condition is true. It branches to the last instruction when it becomes false.

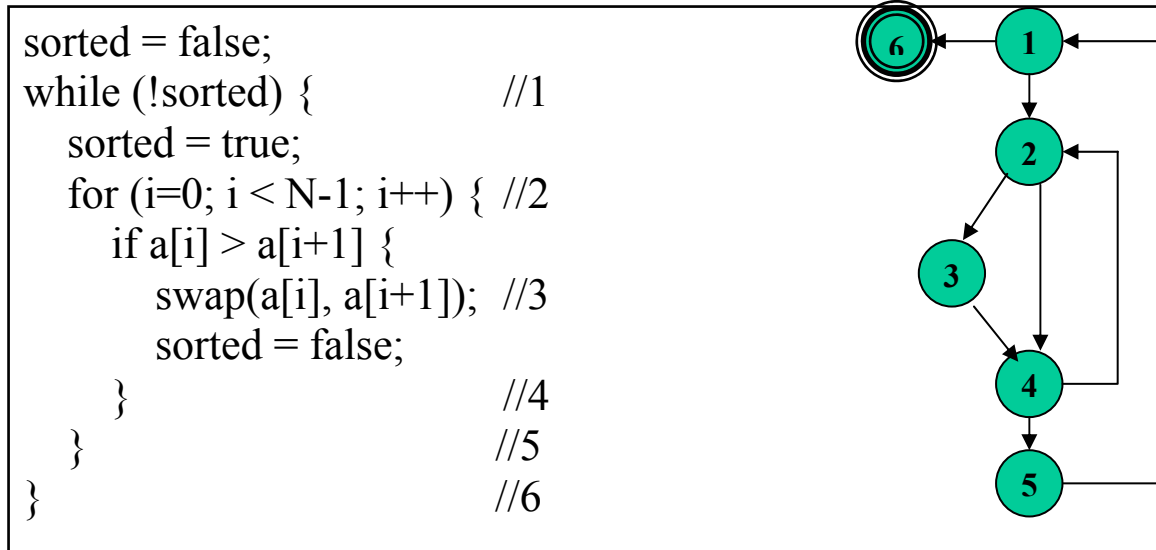


Flow graph for bubble sort

In the following example, code is given for a bubble sort function. The diagram opposite to the code is the corresponding flow graph model. It consists of six nodes. Node one is the while loop instruction that contains loop guard and node six is the ending instruction of the while loop. Node two corresponds to the for loop instructions. Node three corresponds to the swapping instruction. Nodes five and six correspond to the last instructions of the if statement and the for loop consecutively.

Point to note here is the assignment of node numbers to program instructions. As, the corresponding flow graph model would consist of nodes that correspond to instructions which are major decision points in the code. For example, when this function will be invoked, control will certainly come to the while loop instruction and at the minimum it will traverse from node one to node six. However, if control enters into the while loop even for a single iteration, it will traverse through nodes two, four and five for certain and node three too if it enters for loop and check in the if condition is true.

So all these combinations are to be tested during white box testing.



Paths

Following are possible paths from starting to the end of this code.

Path1: 1-6

Path2: 1-2-3-4-5-1-6

Path3: 1-2-4-5-1-6

Path4: 1-2-4-2-3-4-5-6-1

White box testing

As described in the section above, in white box testing we test the structure of the program. In this technique the test cases are written in a manner to cover different possibilities in code. Below are described three coverage schemes.

Coverage

- **Statement Coverage:** In this scheme, statements of the code are tested for a successful test that checks all the statements lying on the path of a successful scenario.
- **Branch Coverage:** In this scheme, all the possible branches of decision structures are tested. Therefore, sequences of statements following a decision are tested.
- **Path Coverage:** In path coverage, all possible paths of a program from input instruction to the output instruction are tested. An exhaustive list of test cases is generated and tested against the code.

White Box Testing Example

With the help of the following example, we shall see how many test cases are generated for each type of coverage schemes.

if (a == b)	//1	
c = d;	//2	
a++;	//3	

Statement coverage:

- a=1,b=1
- If a==b then statement 2, statement 3

Branch coverage

- Statement 1: two branches 1-2, 1-3
- Test case 1: if a =1, b=1 then statement 2
- Test case 2: if a=1,b=2: then statement 3

Path coverage

Same as branch testing

Paths in a program containing loops

Now we shall apply the path coverage scheme on a piece of code that contains a loop statement and see how many test cases can possibly be developed.

for (i = 0; i < N; i++) {	//1	
if (condition1)		
// do something here	//2	
else		
// do something here	//3	
// something here	//4	
}	//5	

Paths

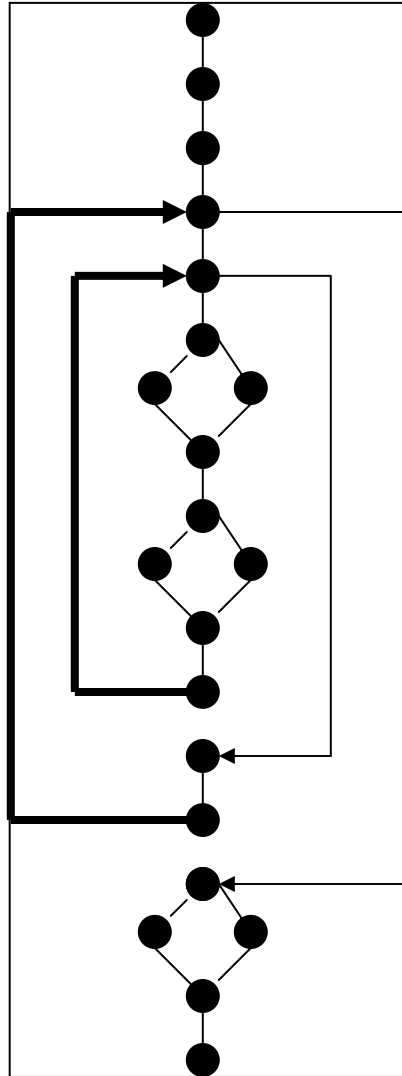
The following is an analysis of the above-mentioned code and the flow diagram. It determines the number of paths against different iterations of the loop.

- $N = 0$: If the control does not enter into the loop then only one path will be traversed. It is 1-5.
- $N=1$: Two different paths can possibly be traversed (depending on condition).
 - 1-2-4-1-5
 - 1-3-4-1-5
- $N=2$: Four possible paths can be traversed.
 - 1-2-4-1-2-4-1-5
 - 1-2-4-1-3-4-1-5
 - 1-3-4-1-2-4-1-5
 - 1-3-4-1-3-4-1-5
- Generalizing the relation between loop variable N and the number of possible paths, for the value of N , 2^N paths are possible
 - Thus if $N = 20$ it means more than 1 million paths are possible.

Flow graph of a hypothetical program

Following is a flow graph of a hypothetical program whose structure is such that it consists of two loops, one embedded in the other. Left side of the diagram signifies the loops where inner loop seem to be containing two if statements. Second loop has a branch and then program finishes.

Simple graph contains 1852 paths with each loop not iterated more than twice



Thus, the number of paths in a program that contains loops tends to infinity. It is impossible to conduct exhaustive testing of a program that may consist of infinite number of test cases. The question arises, how many test cases need to be executed in order to test all the major scenarios in the code at least once? The answer is, calculate the cyclomatic complexity of the code. This will give us a number that corresponds to the total number of test cases that need to be generated in order to test all the statements and branches in the code at least once.

Cyclomatic complexity

The concept of cyclomatic complexity is extremely useful in white box testing when analyzing the relative complexity of the program to be tested. It revolves around independent paths in a program which is any path through the program (from start to end) that introduces at least one new set of processing statements or a new condition. An independent path covers statements and branches of the code.

Cyclomatic complexity is a quantitative measure of the logical complexity of a program. It defines number of independent paths in the basis set of a program. It provides an upper bound for the number of tests that must be conducted to ensure that all statements and branches have been executed at least once.

Cyclomatic Complexity, $V(G)$, for a flow graph G is defined as:

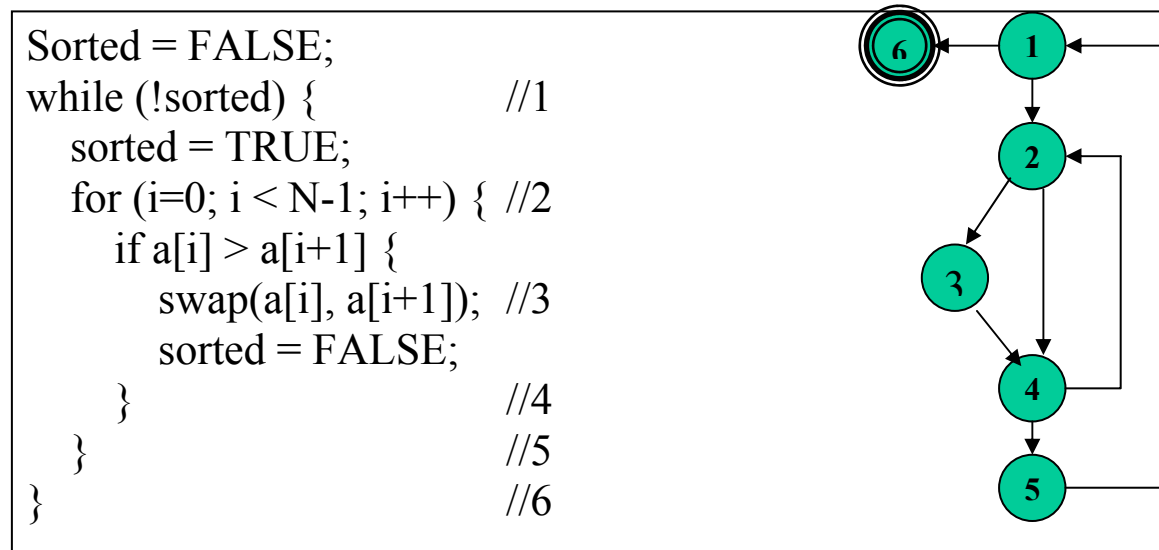
$$V(G) = E - N + 2$$

Where E is the number of edges and N is the number of nodes in the flow graph G .

Cyclomatic complexity provides us with an upper bound for the number of independent paths that comprise the basis set.

Cyclomatic Complexity of a Sort Procedure

Following is the same bubble sort program that we discussed above. This time we shall calculate its cyclomatic complexity and see how many test cases are needed to test this function.



Cyclomatic complexity

- Number of edges = 8
- Number of nodes = 6
- $C(G) = 8 - 6 + 2 = 4$

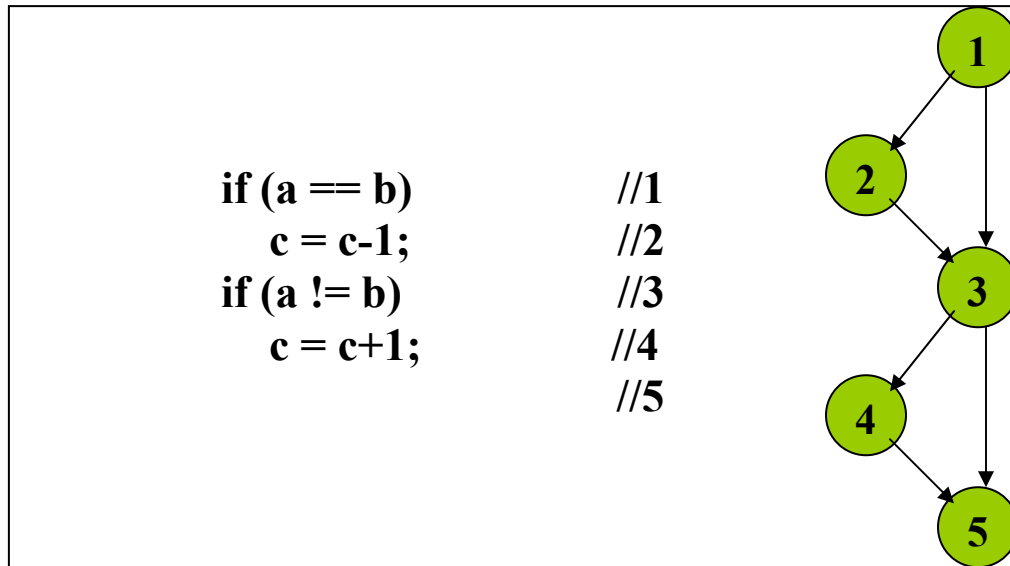
Paths to be tested

- Path1: 1-6
- Path2: 1-2-3-4-5-1-6
- Path3: 1-2-4-5-1-6
- Path4: 1-2-4-2-3-4-5-6-1

Infeasible paths

Infeasible path is a path through a program which is never traversed for any input data.

Example



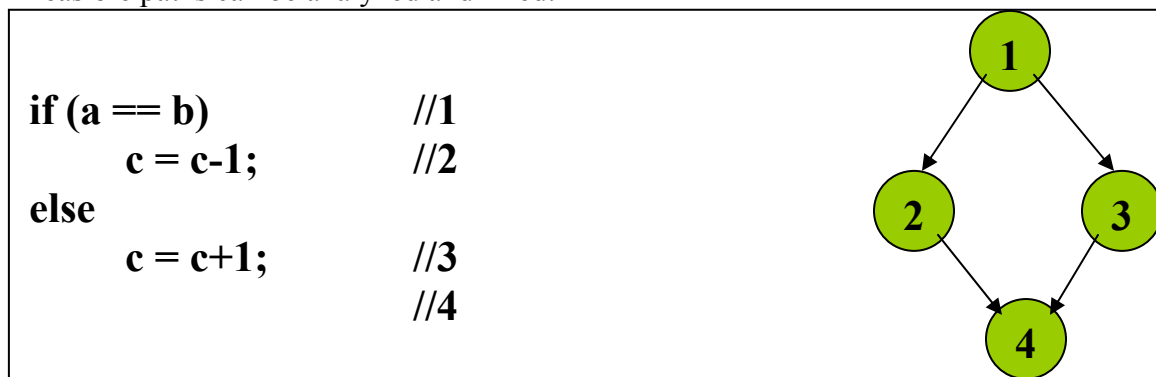
In the above-mentioned example, there are two infeasible paths that will never be traversed.

- Path1: 1-2-3-4-5
- Path2: 1-3-5

A good programming practice is such that minimize infeasible paths to zero. It will reduce the number of test cases that need to be generated in order to test the application. How can we minimize infeasible paths, by simply using else part with the if statement and avoid program statements as given above.

Modified code segment

Infeasible paths can be analyzed and fixed.



There are no infeasible paths now!

Unit testing

A software program is made up of units that include procedures, functions, classes etc. The unit testing process involves the developer in testing of these units. Unit testing is

roughly equivalent to chip-level testing for hardware in which each chip is tested thoroughly after manufacturing. Similarly, unit testing is done to each module, in isolation, to verify its behaviour. Typically the unit test will establish some sort of artificial environment and then invoke routines in the module being tested. It then checks the results returned against either some known value or against the results from previous runs of the same test (regression testing). When the modules are assembled we can use the same tests to test the system as a whole.

Software should be tested more like hardware, with

- Built-in self testing: such that each unit can be tested independently
- Internal diagnostics: diagnostics for program units should be defined.
- Test harness

The emphasis is on built in testability of the program units from the very beginning where each piece should be tested thoroughly before trying to wire them together.

Unit Testing Principles

- In unit testing, developers test their own code units (modules, classes, etc.) during implementation.
- Normal and boundary inputs against expected results are tested.
- Thus unit testing is a great way to test an API.

Quantitative Benefits

- **Repeatable:** Unit test cases can be repeated to verify that no unintended side effects have occurred due to some modification in the code.
- **Bounded:** Narrow focus simplifies finding and fixing defects.
- **Cheaper:** Find and fix defects early

Qualitative Benefits

- **Assessment-oriented:** Writing the unit test forces us to deal with design issues - cohesion, coupling.
- **Confidence-building:** We know what works at an early stage. Also easier to change when it's easy to retest.

Testing against the contract (Example)

When we write unit tests we want to write test cases that ensure a given unit honors its contract. This will tell us whether the code meets the contract and whether the contract means what we think the unit is supposed to do in the program.

Contract for square root routine

```
result = squareRoot(argument);  
assert (abs (result * result - argument) < epsilon);
```

The above contract tells us what to test:

- Pass in a negative argument and ensure that it is rejected
- Pass in an argument of zero to ensure that it is accepted (this is a boundary value)
- Pass in values between zero and the maximum expressible argument and verify that the difference between the square of the result and the original argument is less than some value epsilon.

When you design a module or even a single routine, you should design both its contract and the code to test that contract. By designing code to pass a test that fulfill its contract, you might consider boundary conditions and other issues that you wouldn't consider otherwise. The best way to fix errors is to avoid them in the first place. By building the tests *before* you implement the code you get to try out the interface before you commit to it.

Unit Testing Tips

Unit test should be conveniently located

- For small projects you can imbed the unit test for a module in the module itself
- For larger projects you should keep the tests in the package directory or a /test subdirectory of the package

By making the code accessible to developers you provide them with:

- Examples of how to use all the functionality of your module
- A means to build regression tests to validate any future changes to the code

You can use the main routine with conditional compilation to run your unit tests.

Defect removal efficiency

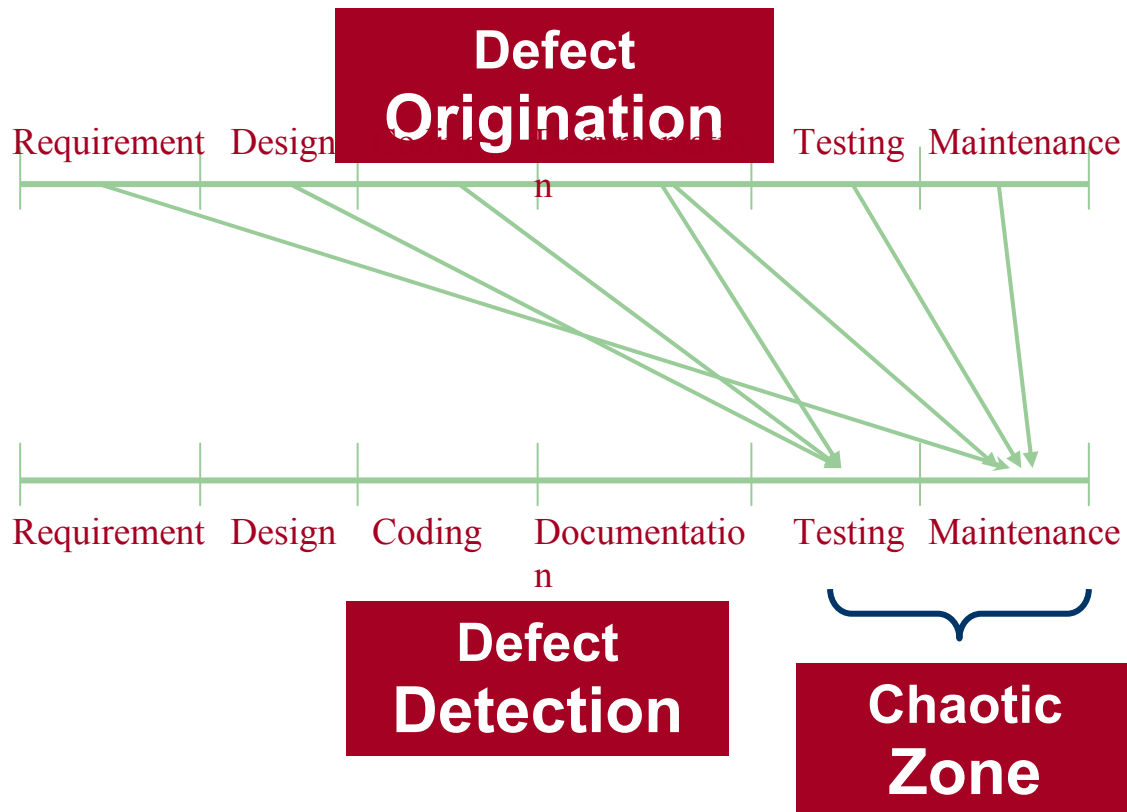
Is the ability to remove defects from the application. We shall further elaborate this idea

Design Inspection					●		●	●	●
Code Inspection				●		●		●	●
Quality Assurance			●			●	●		●
Testing		●	●	●	●	●	●	●	●
Worst	30%	37%	50%	55%	65%	75%	77%	85%	95%
Median	40%	53%	65%	70%	80%	87%	90%	97%	99%
Best	50%	60%	75%	80%	87%	93%	95%	99%	99.9%

with the help of the above-mentioned table.

The above table depicts data that was published after analyzing 1500 projects. In these projects, four different types of quality assurance mechanisms were employed. It is evident from this table that testing alone can only remove 53% of defects. However, testing and quality assurance mechanisms combine yield up to 65% efficiency. Whereas, if we combine code inspection and testing together, results are up to 75%. Similarly, design inspections and testing yield up to 80%. Moreover, combining design inspections, quality assurance and testing results are up to 95%. If all four techniques are combined, results are up to 99.9%.

Inspection and chaotic zone



In this diagram, a chaotic zone has been defined. In fact, if defects are not discovered and fixed at the appropriate stage, then at the testing and maintenance phases, these defects are piled up. Therefore, a chaotic zone is formed for the testing and the development teams as the number of defects which are piled up, destabilize the application and it becomes extremely hard to fix all these defects as some of these bugs may involve changes in requirements and design. At testing or the maintenance phases, fixing a defect in requirements or design becomes extremely expensive, as underlying code will have to be changed as well.

If we combine the results of the above two diagrams, it is evident that testing alone does not suffice. We need to employ inspection techniques and combine them with testing to increase the effectiveness of defect removal efficiency.

Defect origination

In inspections the emphasis is on early detection and fixing of defects from the program. Following are the points in a development life cycle where defects enter into the program.

- Requirements
- Design
- Coding
- User documentation

- Testing itself can cause defects due to bad fixes
- Change requests at the maintenance or initial usage time

It is important to identify defects and fix them as near to their point of origination as possible.

Inspection versus Testing

Inspections and testing are complementary and not opposing verification techniques. Both should be used during the verification and validation process. Inspections can check conformance with a specification but not conformance with the customer's real requirements. Inspections cannot check non-functional characteristics such as performance, usability, etc. Inspection does not require execution of program and they maybe used before implementation. Many different defects may be discovered in a single inspection. In testing, one defect may mask another so several executions are required. For inspections, checklists are prepared that contain information regarding defects. Reuse domain and programming knowledge of the viewers likely to help in preparing these checklists. Inspections involve people examining the source representation with the aim of discovering anomalies and defects. Inspections may be applied to any representation of the system (requirements, design, test data, etc.) Thus inspections are a very effective technique for discovering errors in a software program.

Inspection pre-conditions

A precise specification must be available before inspections. Team members must be familiar with the organization standards. In addition to it, syntactically correct code must be available to the inspectors. Inspectors should prepare a checklist that can help them during the inspection process.

Inspection checklists.

Checklist of common errors in a program should be developed and used to drive the inspection process. These error checklists are programming language dependent such that the inspector has to analyze major constructs of the programming language and develop checklists to verify code that is written using these checklists. For example, in a language of weak type checking, one can expect a number of peculiarities in code that should be verified. So the corresponding checklist can be larger. Other example of programming language dependant defects are defects in variable initialization, constant naming, loop termination, array bounds, etc.

An Inspection Checklist

Following is an example of an inspection checklist.

Exception management faults	<ul style="list-style-type: none"> • Have all possible error conditions been taken into account?
Fault Class	<ul style="list-style-type: none"> • Inspection Check
Data faults	<ul style="list-style-type: none"> • Are all program variables initialized before their values are used? • Have all constants been named? • Should the lower bound of arrays be 0, 1, or something else? • Should the upper bound of arrays be size or size -1? • If character strings are used, is a delimiter explicitly assigned?
Control faults	<ul style="list-style-type: none"> • For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for?
Input/Output faults	<ul style="list-style-type: none"> • Are all input variables used? • Are all output variables assigned a value before they are output?
Interface faults	<ul style="list-style-type: none"> • Do all function and procedure calls have correct number of parameters? • Do formal and actual parameters types match? • Are the parameters in right order? • If components access shared memory, do they have the same model of shared memory structure?
Storage management faults	<ul style="list-style-type: none"> • If a linked structure is modified, have all links been correctly assigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly de-allocated after it is no longer required?

In the checklist mentioned above, a number of fault classes have been specified and their corresponding inspection checks are described in the column at the right side. This type of checklist helps an inspector to look for specific defects in the program. These inspection checks are the outcomes of experience that the inspector has gained out of developing or testing similar programs.

Static analyzers

Static analyzers are software tools for source text processing. They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the verification and validation team. These tools are very effective as an aid to inspections. But these are supplement to but not a replacement for inspections.

Checklist for static analysis

Data faults	<ul style="list-style-type: none">• variable used before initialization• variable declared but never used• variables assigned twice but never used between assignments• possible array bound violations• undeclared variables
Control faults	<ul style="list-style-type: none">• unreachable code• unconditional branches into loops
Input/Output faults	<ul style="list-style-type: none">• variable output twice with no intervening assignment
Storage Management fault	<ul style="list-style-type: none">• unassigned pointers• pointer arithmetic