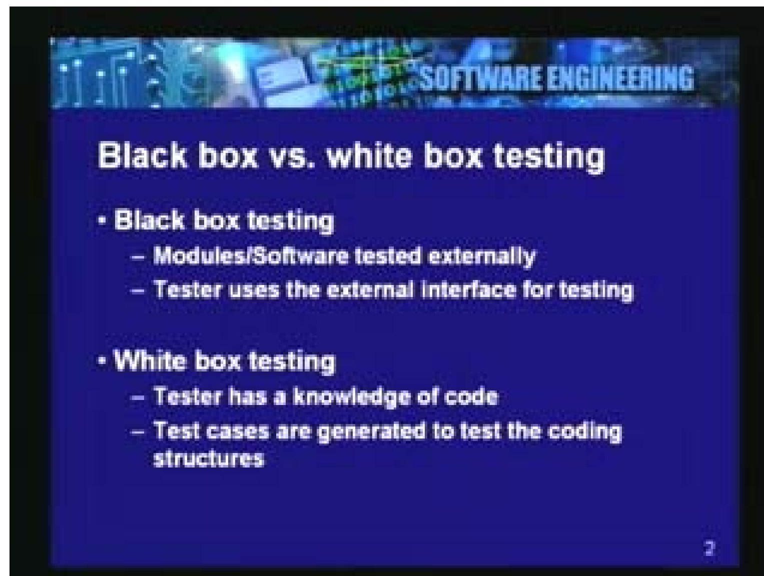
Software Engineering
Prof. Rushikesh K. Joshi
Computer Science & Engineering
Indian Institute of Technology, Bombay
Lecture - 19
Software Testing – II

In continuation with the last lecture on software testing, where we mainly discussed black box testing methods, now we will discuss some of the white box testing methods. Let us see what the techniques are and how you derive your test suites and different test cases through an analysis of the code. So let us first look at what are the different black box and white box testing techniques and it is an overview of these two different techniques with a comparison. In black box testing, basically modules are first tested externally from the point of view of externally observable behavior.

(Refer Slide Time 01:24)

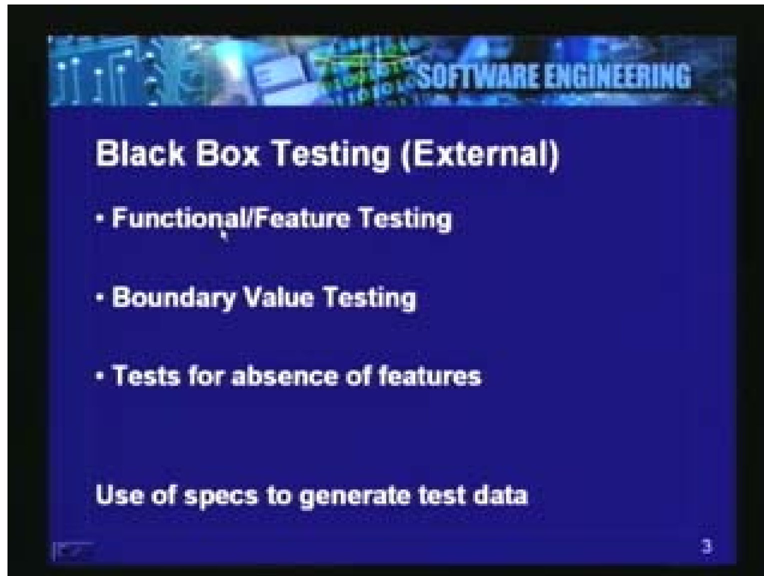


The tester uses the external interface for testing and in white box testing as oppose to the black box approach, the tester has the knowledge of the code. In a black box method you do not know what is there inside. So, given a module if you want to test the module, the tester will not look at the code that the module contains, but the specifications or the contracts that are supported by the module. So based on the contracts or externally observable behavior, the tester derives the test cases and test these modules in the black box fashion.

But in a white box technique the tester looks at the code and then derives the test cases. In a white box method, basically the test cases do not get derived from the high level specifications or the functional requirements. You are mainly looking at the code, the structure, and the usages of variables, different control parts, and data flow parts and so on.

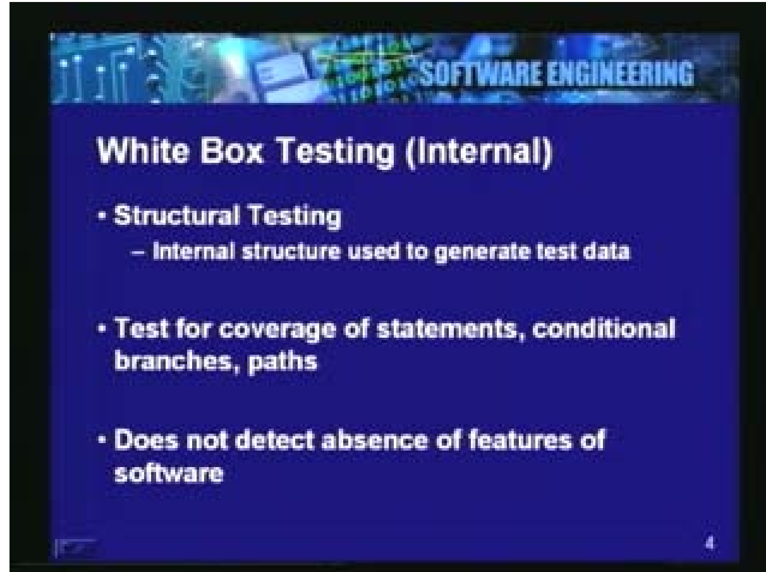
Thus tester has knowledge of code in white box testing and test cases are generated to test the coding structures. These are the main differences in black box and white box approaches. This slide summarizes black box testing techniques which are external testing techniques. For example, first one is the functional or feature testing, we also have seen boundary values testing where you mainly look at the boundaries of the different parameters that you pass to functions and find out whether the module works with the extremes and also check for the behavior beyond the extremes.

(Refer Slide Time 03:24)



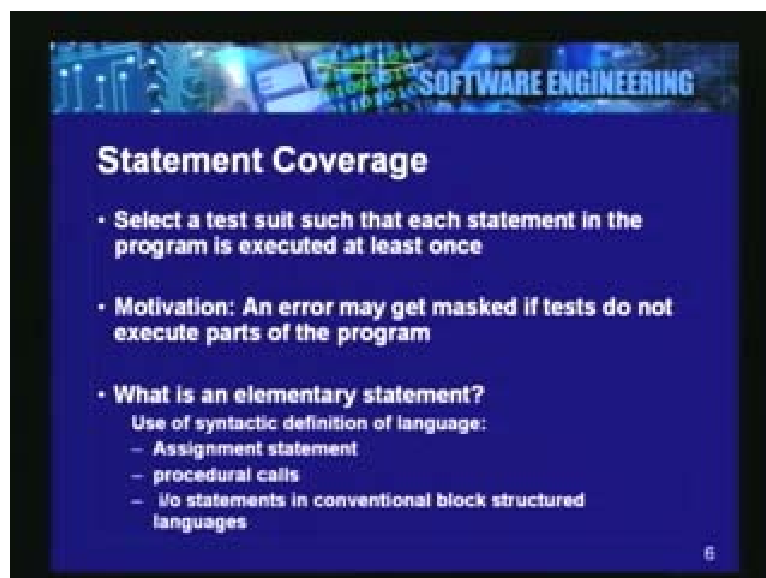
Then test for absence of features. For example whether a given feature or a use case has been supported or not can be found out through black box testing measures. And you typically use specifications to generate the test data. The white box testing is mainly internal testing. The difference is mainly from internal or external perspective. This is also called structural testing, because the internal structure is used to generate test data. Typically it covers test for statement coverage, conditional branch coverage, and different paths in the control flow of the program or the module.

(Refer Slide Time 04:22)



But this does not detect absence of features in the software. For example, if we look at a module which has a 'while' statement and which mainly covers a specific feature; we will be mainly not looking at the feature for which the 'while' statement has been programmed, but you will be mainly looking at the conditions and applying certain criteria which you are going to see and deriving these test cases. We have a few examples in later slides on how the white box testing criteria may not detect absence of features in the software. So let us continue with our coverage criteria and explore more about these white box testing techniques. Let us look at statement coverage first.

(Refer Slide Time 05:50)

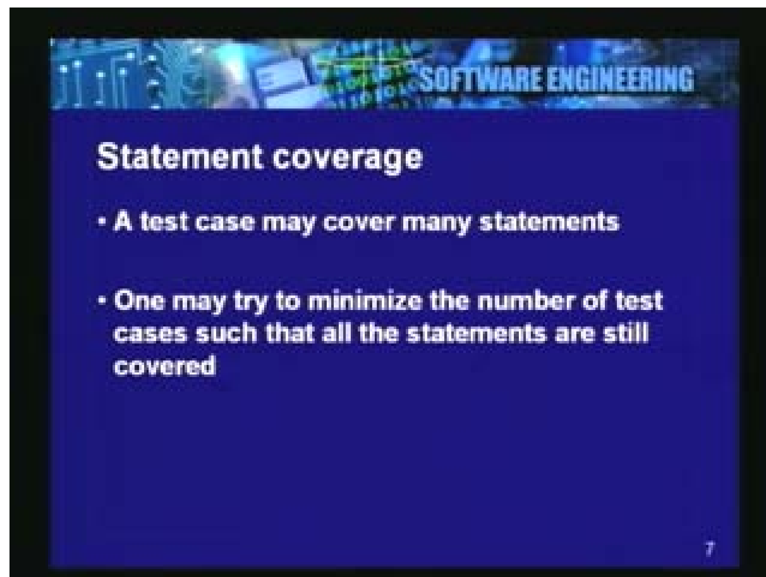


So, what statement coverage says is that, you select a test suite such that, each statement in the program is executed at least once. If you have a program of 5000 lines, your test suite that satisfies the statement coverage criteria will generate as many test cases as required to cover all these statements. If you have 5000 lines of code, you do not need 5000 test cases to cover all the 5000 lines of code. You might be able cover these statements with lesser number of test suites. For example, if you just have a single sequence without any branching, just one 'case' may be enough to cover all the statements.

What is the motivation for this particular criterion, why do you go for statement coverage criteria? This is basically from the fact that an error may get masked if test do not execute parts of the program. For example, if you have not covered all the statements, you might be actually hiding an error which may be because of those statements that have not been covered by your test suites. So a statement coverage criteria guarantees that all statements in the program are covered at least once. When we say all statements, we basically mean all elementary statements.

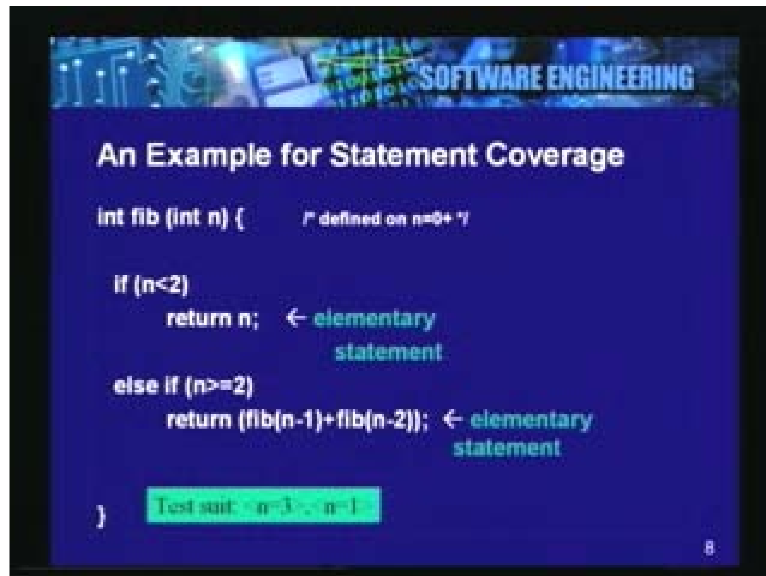
What are the elementary statements? You can find them out from syntactic definition of the language that we are using for programming. For example, an assignment statement is an elementary statement. A procedure call and I by O statement in the conventional block structured languages; for example, 'printf'by'scanf' etc, all are the elementary statements. All of them should be covered at least once. As we have seen, this slide summarizes that the test case may cover many statements and one may try to minimize the number of test cases such that, all statements are still covered.

(Refer Slide Time 08:25)



So, how do we reuse the number of test cases and try to cover as many statements as possible? It is not required that you have as many test cases as the number of statements in the program. You can cover all those elementary statements with fewer test cases. So what are the criteria, what are the techniques? Let us continue and let us find out answers to these questions. Here is an example for statement coverage. This is a function that calculates Fibonacci integer ‘n’ and it is defined on n is equal to 0 or more.

(Refer Slide Time 09:02)

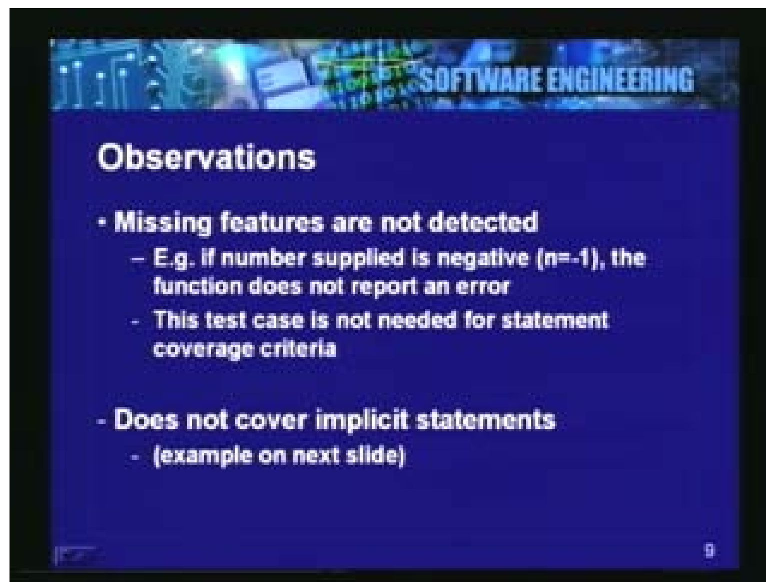


Let us look at this code. We have an ‘if statement’; if ‘n’ is less than 2, then you return ‘n’, else if ‘n’ is greater than or equal to 2, you recursively apply this Fibonacci and you calculate Fibonacci n-1 and then you add to it Fibonacci n-2. So, what are the elementary statements in this? Is this ‘if’ statement an elementary statement? This is not an elementary statement, this is the branching statement. But this ‘return n’ is an elementary statement or basic statement in this terminology of statement coverage and the next return statement in the ‘else’ part is also another elementary statement. So what the criteria says is that you must be able to cover all these elementary statements through your test suite.

Is it possible to cover these two statements with one case itself? If you look at the first condition which is n <2, that means this is for ‘n’ greater than or equal to 0. This will be satisfied by 0 and 1 values against the values of n. And if you look at this else part, this ‘return’ the number by recursive computations of Fibonacci n-1 and n-2. Here the condition is satisfied by a value of ‘n’ to be greater than or equal to 2. So you cannot get one test suite to cover these two statements. It requires at least 2 cases and two are enough. For example here the green box in the bottom of the slide gives one such test suite n is equal to 3 and n is equal to 1. This test suite contains two different test runs. You have to execute this program twice, first with a value of ‘n’ to be say 3 and then with a value of ‘n’ to be 1.

If you execute this Fibonacci function with n to be 3, this else part will be applicable and you will cover this second elementary statement. Where as if you execute this program with the value of n is equal to 1, the first elementary statement will be covered. So this gives you the statement coverage criteria and this is an example of statement coverage. Now, let us look at some of the observations. We said that missing features do not get detected. So what are the missing features? For example, in this program if number supplied is negative, the function does not report an error. So there is no feature in this function to return with an error if the number supplied is negative.

(Refer Slide Time: 12:14)



So, that feature will not get detected, because our coverage criteria are mainly from the point of view of covering the statements which are there. It is not covering what is not there. So, if you are able to cover these two elementary statements, you satisfy the statement coverage criteria. Even otherwise, black box measures will not be able to detect missing features and this is an example that we have. If the number supplied is negative, the function does not report an error. And if this is the wanted feature, but which is missing in a software, we will not be able to detect through statement coverage, because such a test case is not generated through this particular coverage technique. We are only mainly worried about covering the statements which are there and we do not cover implicit statements.

So what are those implicit statements? Example is on the next slide. So we have here another function, a flip function which takes in a Boolean variable and returns a Boolean result.

(Refer Slide Time: 13:38)

SOFTWARE ENGINEERING

Implicit statements

```
bool flip (bool var) {
 bool local;
  if (isTrue(var))
      local = false; ← elementary statement
  return local; ← elementary statement
}
```

Test suit <var=True> sufficient to cover all statements
Error that flip does not work with <var=false> is not detected

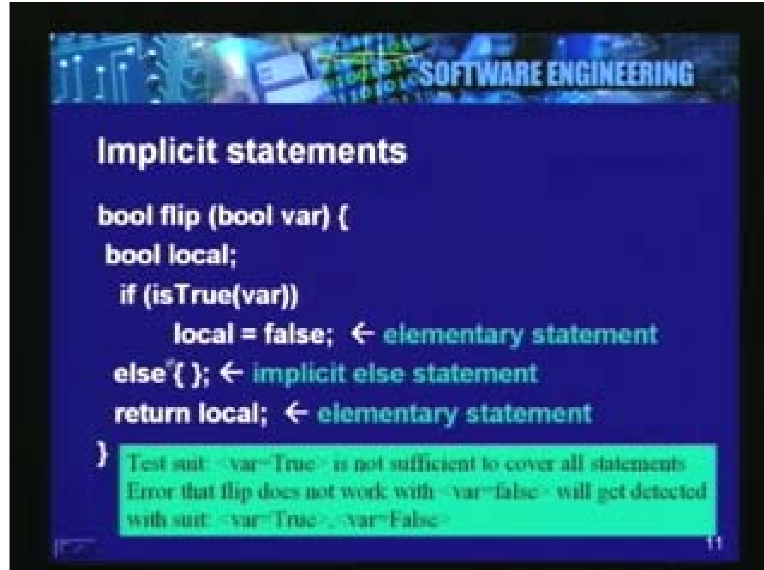
10

This function is intended to flip the variable and return the flipped Boolean value. So what is expected with flip true is false and the expected result on flip false is true? So let us look at the code that we have for this flip statement and let us see what happens when we apply statement coverage criteria. You have declared a Boolean local variable. What we are saying is that, if whatever comes in ‘var’ is true, then first assign ‘false’ to ‘local’ and return local. This is just an example. You have two elementary statements here, one assignment statement and one return statement. So, ‘local is equal to false’ is one elementary statement and ‘return local’ is another statement. In order to cover ‘local is equal to false’ and return local, you have one test suite which is enough.

If you have ‘var’ to be true, this assignment ‘local is equal to false’ will get executed. Because if Boolean value ‘var’ is true’ then this ‘is true’ predicate will evaluate to true and the local is equal to false assignment will get executed and you will be able to see the correct return value to be false. So ‘var is equal to true’ is sufficient to cover all the statements, but the error that flip does not work with ‘var is equal to false’ is not detected. Why does this not get detected? The reason is that we have overlooked the else part. So, when you have this ‘if’ statement, you also have the ‘else’ part which is implicit here. So we have to also cover that implicit part.

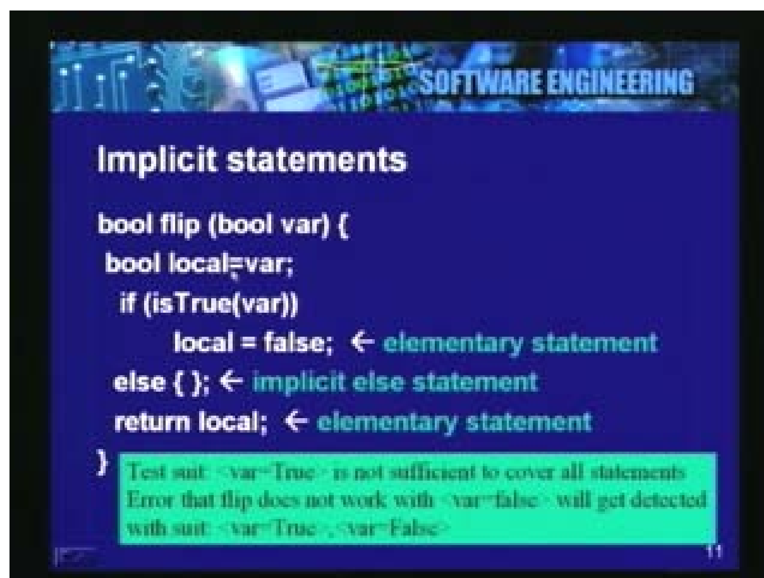
The next slide gives you the implicit else statement. So if you look at the code we have also added the implicit statement. This program and the program on the earlier slide are equivalent.

(Refer Slide Time 16:00)



In the above program, we have 'else' doing nothing but simply return local. If you have false and if you say that the value of local is assigned to one default, this may not work correctly. For example test suite 'var is equal to true' is not sufficient to cover all statements. Now if you assign 'local is equal to var' in the first statement and if you simply return local as it is, then you will not get correct value. For example, I will make this small change on this slide, say local is equal to var and let us look at this program now. If you assign the initial value of local to be var as it is and then you will be returning local as it is, if you execute this with 'var is equal to false', then we will return false.

(Refer Slide Time 16:53)



So test suite ‘var is equal to True’ is not sufficient to cover all statements. Error that flip does not work with ‘var is equal to false’ will get detected with this test suite: “var is equal to true, var is equal to false”. So, if you want to cover this statement you need to execute this ‘else’ part and in order to execute ‘else’ part, var should not be true. That gives you another input value for this parameter ‘var’ to be false. So if you have this test suite, then you are able to cover all these elementary statements. Basically you are able to cover both the assignment statement and the second else part if you use the second test suite.

In the last slide, since the program does not contain the ‘else’ part, you might overlook the ‘else’ part and might not generate the test suite to cover that statement and then that results in an insufficient statement coverage criteria, and you will not be able to detect the error. So if you also look at the implicit statement, the implicit branches and apply the statement coverage criteria, you are improving on your statement coverage and you will cover errors that get resulted due to such implicit statements which are not seen in the source code.

Now we will move on to another technique after seeing the statement coverage criteria which talks about path based testing. You have to look at what are the basic paths in the program, such that if you cover these basic paths you will be able to cover all the statements in the program. In this basic path testing method, you select test suite such that, the basic paths are covered and this guarantees that every statements gets covered. So let us look at this path based criteria and link it to statement coverage.

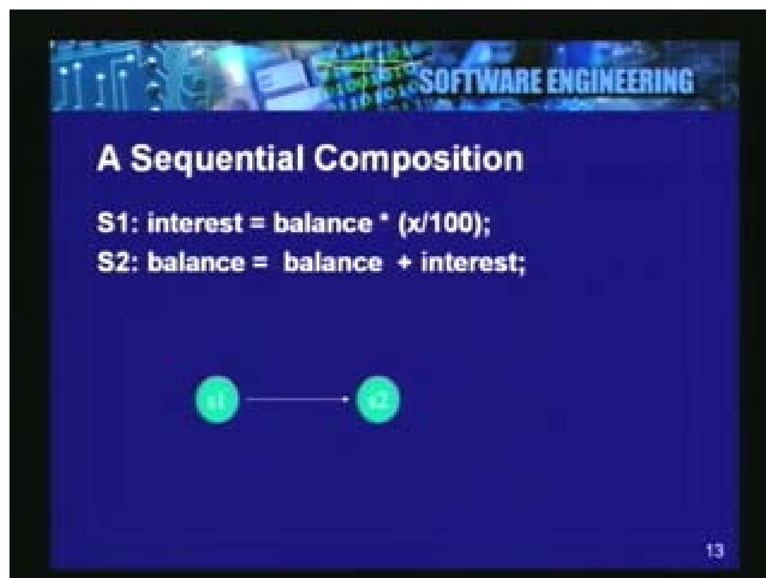
(Refer Slide Time 19:00)

Basically, have representations for elementary statements such as assignments, IO call. Then conditional statement such as ‘if then else’ statement, and also you may have the ‘case’ statement and conditional loops ‘While-do’ and ‘Repeat-until’ and sequential composition of two sequential statements.

So these are the primitives that give you variations in the control flow of the program and you can generate the control flow diagram of a given modular program. And then go through different paths and see how many paths you should test. In other words how many test cases you should generate to cover the basic paths. And then all these basic paths lead to coverage of all the statements in the program that you are testing. Let us look at some of these primitives that are used to generate these control flow graphs.

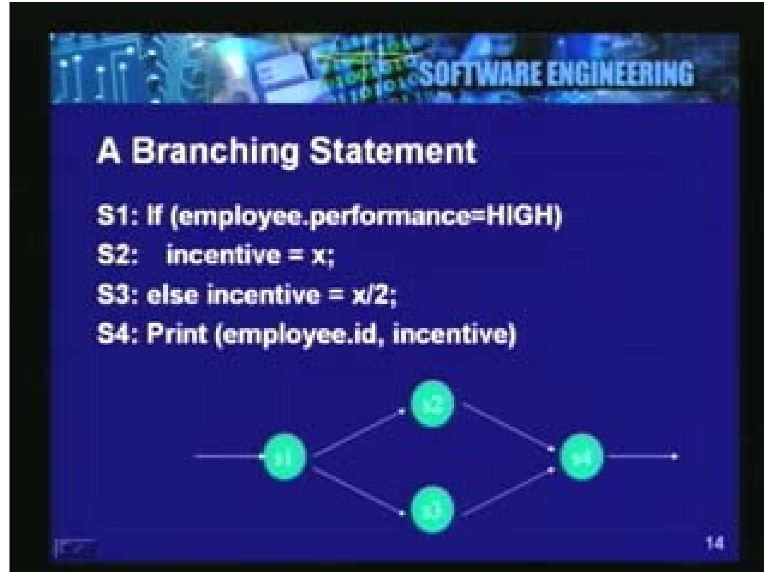
So first is sequential composition. You have two statements. Here is one example; you have S1 which calculates interest as balance into some x percentage factor and S2 as balance is equal to balance plus interest. So you add this interest into the balance.

(Refer Slide Time: 20:15)



You have S1 followed by S2, so this is your sequential composition. If you just have only the sequential statements one after the other; S1 followed by S2 followed by S3 followed by S4, then that would mean that you just need to have one test suite to cover all the statements in the program. So we have only one path and you will have one test suite here, if we have just only the sequential composition. Then next we need to look at the branching statement. For example you have a code on this slide. If employee dot performance is high, then you apply one incentive equal to some value x, else incentive is x by 2 and then print employee id and the incentive.

(Refer Slide Time 21:16)



So, you have four statements here; S1, S2, S3, S4 and this is your equivalent flow graph. You have S1 as the branching node. From S1 you can go to S2 and S3 and then back to S4. The S4 is after the 'if then else'. So S4 executes after the entire 'if then else' statement executes. It is basically a joint point or the join of two branches and S1 is the fourth point where you branch into two different paths. 'Else' part is the S3 part and the 'if' part is the S2 part. So, for this particular flow how many test cases do you require to you cover all this statements? You will require two test cases. So that one path is S1 S2 S4 and the other path that covers the nodes S1 S3 S4.

So, you have two basic paths in this program which you can test, that will result into the statement coverage criteria including all the paths that get generated due to all the possible branching at this particular condition node. Now, this is the 'while' statement. In 'while' statement you execute a condition at the beginning of the 'while' loop and while that condition is true you continue to execute the body of the 'while' loop. And at the end of the 'while' loop, the condition evaluates to false, you terminate the 'while' loop and then the program continues.

(Refer Slide Time 23:20)

SOFTWARE ENGINEERING

A While Statement

S1: while (!end_of_file (file))
S2: read a value from file, and print it;
S3: file.close();

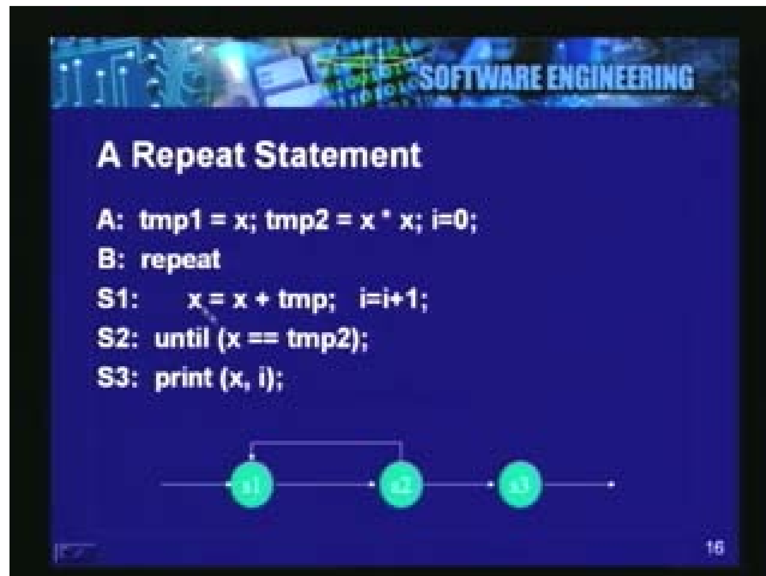
In this slide we have three statements. You are processing a file; ‘while it is not the end of file’ [while (! end_of_file (file))], you read a value from file and print it. When the end of file is reached, the statement gets terminated and you continue with program. S3 is the next statement. So, basically you can see that it is a sequential composition of S1 S2 together as ‘while’ loop and then S3 at the end of the ‘while’ loop. So, S1 is your conditional statement from where you can go to S2 and from S2 you come back to S1 to check the condition. It can be seen here from this figure that in ‘while loop’ you first check the condition.

When you open a file, you first want to check whether you have already reached the end of file. If the file has no input, you do not want to read the value from that file. So, you have to check before you execute the while loop body. So, this is the typical usage where you want to first check the condition and then continue. So from S1, we can see that S3 is the termination of the ‘while loop’ or outside the ‘while loop’. Once you reach S3, you continue with the rest of the program and S2 is your body of the while loop. In S2, we have combined two statements (read a value from file, and print it) into one itself, because adding one more statement here will not make a difference to number of paths, as this is the same path.

In same path itself, if you have these nodes you can merge them. So we come back to S1 after we read the value from the file and print it. Then check the condition again and if the end of file has not been reached, you still continue with the body of the while loop else you terminate ‘while’ loop and continue with S3. This is your ‘while’ statement and in order to cover these statements, how many test cases do you need? First is to go into the ‘while’ body and the second test is to terminate the ‘while’ loop. You require a file which has nothing in it, where end of file will be the beginning file pointer itself. And in a file which has a few items in it, which will be able to go through the loop at least once. So, one item in file is good enough to cover all these statements.

So a file with one single item and a file with nothing in it or an empty file can be used to cover these basic parts, which are two in this particular case. Now let us look at the case of the repeat statement. This is your repeat statement. We assign a value x to ‘tmp1’ (temp 1) and then x square to ‘tmp2’ and set I to 0, and then repeat x is equal to x plus tmp and also increment the I count until x becomes equal to tmp2. That means we are showing that by adding the value of x again and again, you can generate the square.

(Refer Slide Time 26:10)



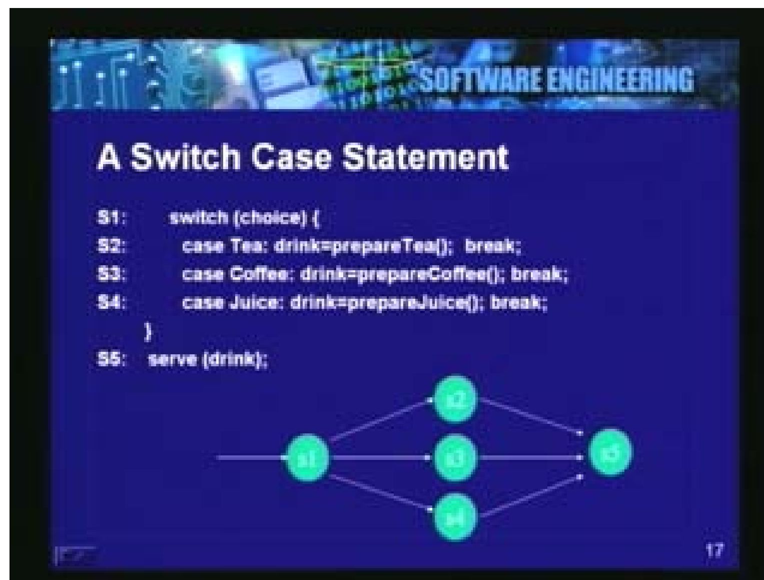
At the end, once we come out of this repeat loop, we print the value of x and I. We know that we can repeat it until this x becomes a square of itself and then at the end of the loop, we print the value of x and I. So, this is your repeat statement and if you want to cover all the statements you need to again take this one particular path and then another path from S1, S2 to S3.

We can see that this will not work for all the values. If the value is just 1 then ‘x’ will get added to itself and the value will become 2 and you may never terminate this loop. So this criteria gives you all these paths, the value of x to be 1 is good enough, but then once you get into this particular statement, you might not come out of this. At the same time, if it is stated that this loop works for this particular set of values, then you will be able to at least cover these paths. So you can see that these criteria are not perfect and they may not still detect all the possible errors. For example, boundary value analysis is not done in this and typically you are mainly looking at only the structure of the code and the available statements and the available branching, through them you are covering the test.

In this repeat statement, this is the diagram or this is the flow graph of the repeat statement. Coming back to our flow diagrams and how to compute the basic paths; we have these S1, S2 and S3: S1 is your basic statement where you are incrementing x with temp and incrementing the value of I and then you are going to S2.

You can see that S2 is the branching node from where you have two outgoing edges. Where as S1 is your elementary statement and from S2 you can come back to S1 or you can jump to S3.You continue until you get the x square, otherwise you go to S3 and continue your execution after the repeat statement. So, here we can see that there is this path from S1 S2 S3 and in order to cover this particular edge you need to choose a test case, such that you cover S1 S2 and comeback to S1 and then perhaps take S2 again and then through S3 you come out. Now, we will look at another possible statement in programming languages such as C, C++.

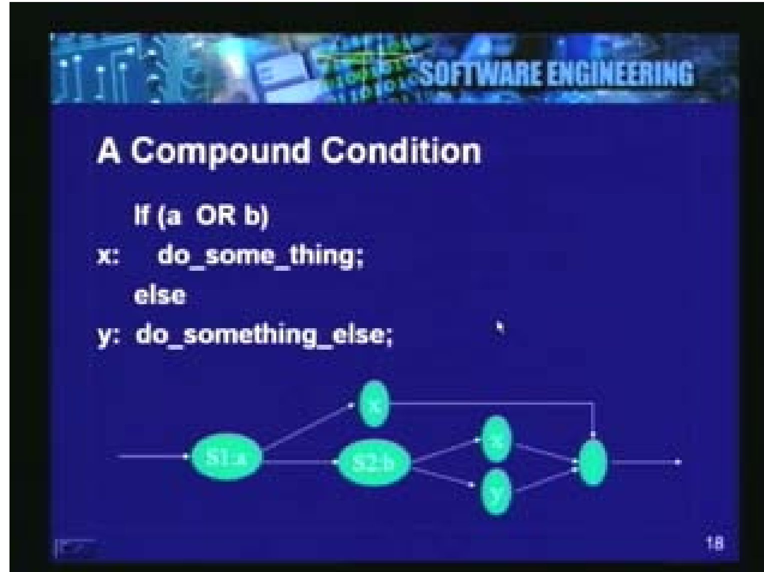
(Refer Slide Time: 29:45)



You have this ‘switch case’ statement, where you have many choices at the switch node and you can fork into many choices and then join later. For example, here you have a ‘choice’. If the ‘choice’ is tea, then you prepare tea. So, ‘drink is equal to prepare Tea ()’ and then terminate the case statement. Another case is, if the ‘choice’ is coffee, then you prepare coffee. If ‘choice’ is juice, you prepare juice and so on.

So, you have these three different cases and S1 is the branching statement or the branching predicate, from where you find out what the choice is and then branch to appropriate nodes and then you join at S5. S5 is your ‘serve (drink)’ or it is actually the termination of ‘switch’ loop and I have combined the serve drink with that, because it is in sequential composition. Hence, you need not have to point out all the nodes. So S5 can be your serve drink, which is the end of your switch statement. If you want to cover all the statements, these are the basic paths S1 S2 S5, S1 S3 S5 and S1 S4 S5. So this is flow graph for a specific switch statement. Now, we will look at a compound condition. In this example, we can see that we have a compound condition.

(Refer slide Time: 31:10)



We are looking at two values and we want to execute ‘x’, if a or b is true. So, if ‘a’ is true we will execute x or if ‘b’ is true then also we will execute x, if both of them are true then we will execute x, else otherwise something else will be done. So if ‘a’ and ‘b’ both are false, then something else will be done. This is the flow graph for this compound condition. Here the S1 will check for ‘a’ and if ‘a’ is true, we execute x and then join at the end of the conditional branching statement. The last node is the joint part of this compound conditional statement.

Now, if ‘a’ is found to be false, then S2 checks for ‘b’ and ‘b’ could be false or true. We have a branching here. We are first branching at ‘a’, if ‘a’ is true then execute x and we can continue and we do not need to check for ‘b’.If ‘a’ is not true, then the lower path is taken. In S2 we check for ‘b’, if ‘b’ is true then we execute x and then we join at the end of the conditional statement. If ‘b’ is false we execute y and then we again join at the end of the conditional statement. So in this case we have this flow graph where we have two branches. This is an example of a compound condition.

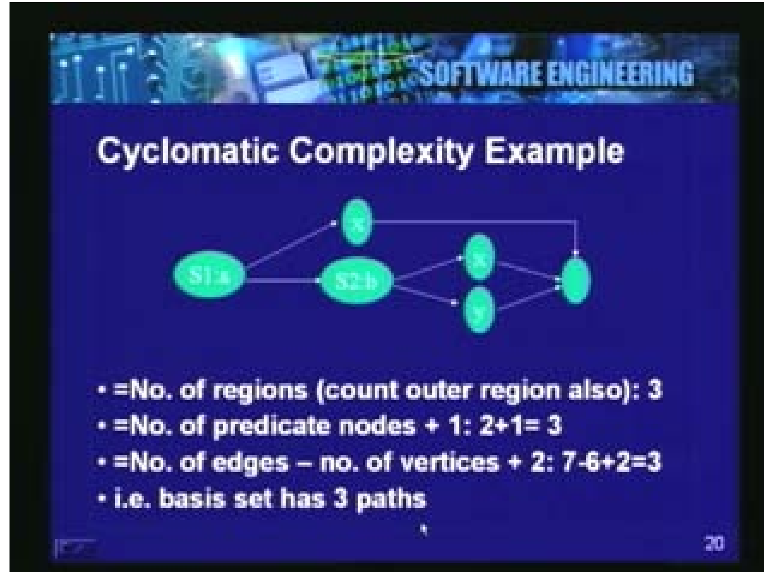
Hence, we need to carefully look at these compound conditions and formulate the flow graph. Otherwise, we will miss some of the branches, then they become implicit and you do not cover them all. This will result into errors getting undetected because of not covering all the basic paths in the control flow graph of your program. Now, we will look at measure called cyclomatic complexity and see how that gives us a clue to generating these basic paths. We have looked at all these basic statements and their flow graph.

(Refer Slide Time: 33:33)

In a big program you have to analyze the entire program and generate the flow graph of your given program and then find out how many paths in the entire flow graph should be tested so that, you will cover all these statements of the program and you can satisfy your certain coverage. Typically a program includes many branches and many statements, and you may have many such paths and you need to cover all the paths. How many cases are required? We have to reduce the number of cases, but we should also make sure that all the statements get covered. There is a measure called cyclomatic complexity and that measure actually tells us how many paths we need to generate, so that these criteria is satisfied.

So cyclomatic complexity gives number of independent paths in the basis set or the basis set for these statement coverage, which has the very basic independent paths, so that each path covers at least one extra edge in the flow graph. That means they are not redundant and it gives upper bound on number of test that must be conducted to ensure all statements get covered at least once. So, this could compute in the number of regions and in that you count the outer region also. A number of predicate nodes plus 1, or you can count it as number of edges minus number of vertices plus 2 and that is your number of regions. So, you can see that cyclomatic complexity for this compound condition which we had just seen a couple of minutes ago can be computed here to be 3.

(Refer Slide Time 35:43)

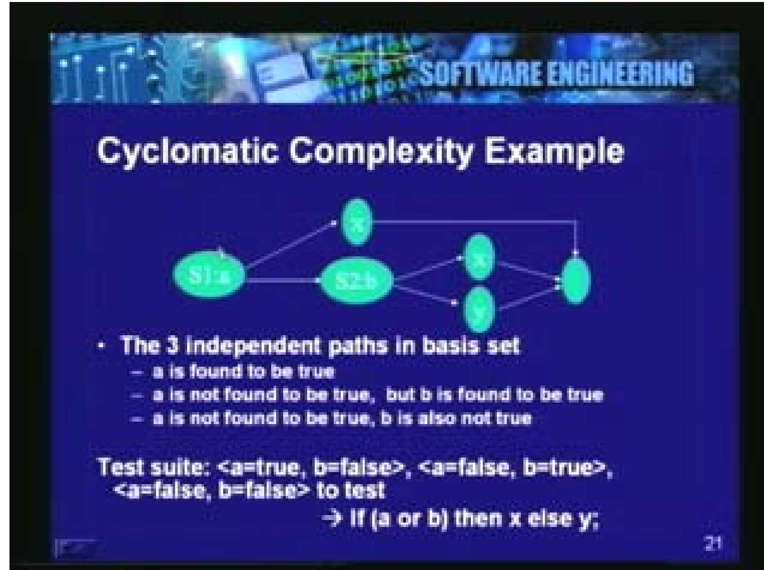


If you look at this flow graph, we have a branching if ‘a’ is true, we are going to execute ‘x’, for that we are going to take upper path. If ‘a’ is false then, we will check if ‘b’ is true, we will take the middle path and we will go by this ‘x’. Else if ‘b’ is not true, then we are going to take y, that means if ‘a’ is not true and ‘b’ is also not true, you take the bottom path and then we join here. So, we have this one region where I am rolling the mouse and then this is the middle region and you have the outer region. These three regions give you the count of number of regions that gives you the cyclomatic complexity. We can calculate the number of predicate nodes plus 1. You have one node S1 and another node S2. Hence, we have two branches and so the number of predicate node plus 1 which is 2 plus 1 equal to 3. Or the number of edges - number of vertices plus 2: So, how many edges are there? We have seven edges and 6 vertices. So, 7 minus 6 plus 2 and you get 3.

So, basis set has three parts. You take the first condition and then take one branch and cover it. Then, go to another branch, take the condition, take the branch and cover it and then for the second one cover the other branch as well and keep the earlier one as it is till the condition. So, you get the top path, the middle path and the lower path. This is your basis set and these three parts if you cover you are able to cover all the statements. This is your example for cyclomatic complexity.

And it will be interesting to see that even if you have too many statements and many paths, this number of region that you have could be small. This basis set gives you the upper bound and these many paths you need to generate. So, how do you generate these paths? You have to select the value such that these paths get covered. So you have to select appropriate values of a and b. The next slide tells you what these values are. So, if we look at this example again. Three independent paths are as follows: One is, ‘a’ is found to be true. And then ‘a’ is not found to be true, but ‘b’ is found to be true. And ‘a’ is not found to be true and ‘b’ is also not found to be true.

(Refer Slide Time: 38:14)



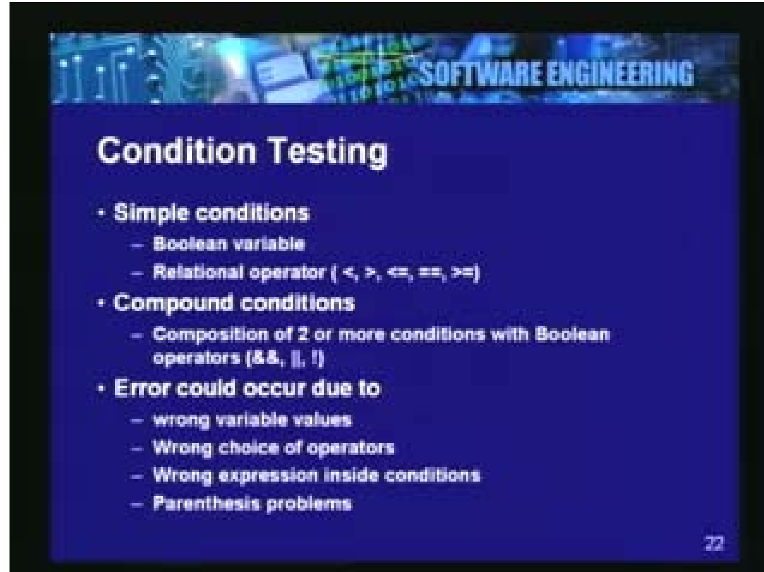
So, test suite is the following: ‘a’ is true; ‘b’ is false. We have chosen ‘b’ to be false here with ‘a’ is true, but it doesn’t matter. Then ‘a’ is false; ‘b’ true is another test. And test number 3 is, ‘a’ is false and ‘b’ is false. So these three tests can cover all the statements. These three tests give you the three basic paths in this particular example. If you have this statement which you can find in the bottom of the slide: “If (a or b) then x else y”, then you require three test cases such that you can cover all the paths in this control flow graph and all the basic paths that is independent paths in the basis set and then that results in covering all the statement in the program.

So, to this statement “If (a or b) then x else y”, you require 3 test cases which may not be apparent if you just look at the statement. You might think that you need to execute ‘then’ part and ‘else’ part only once. But if you look at this, you have three possible independent paths. This close analysis reveals the different basis path in the basis set, that is the independent path. And you need to select appropriate values such that the basis set gets covered and that results in statement coverage.

Are the independent paths in the basis set unique? They need not be unique. In this example, of course you have these three paths, but you can construct various examples or you can construct programs. In general, you can cover through different basis paths in a bigger program and I will leave it to you as an exercise. You try to write a program where there are different ways to construct the basis set or to construct the set of independent paths. So, take four paths and then see whether you can construct two examples that have exactly four independent paths in the basis set and you have two different basis sets.

Now we will look at other criteria for white box testing. In condition testing, we mainly approach from the point of view of the composition of condition. Here, we can see that we can have simple conditions or compound conditions.

(Refer Slide Time 41:00)



So, how to approach these conditions? Basically a white box testing has been covered in great detail in literature. There are many different approaches and ways to design your test suite based on white box testing. So, you could do a very exhaustive testing, but it would not be possible due to extremely very high and very large number of test cases, that is exponential number of test cases.

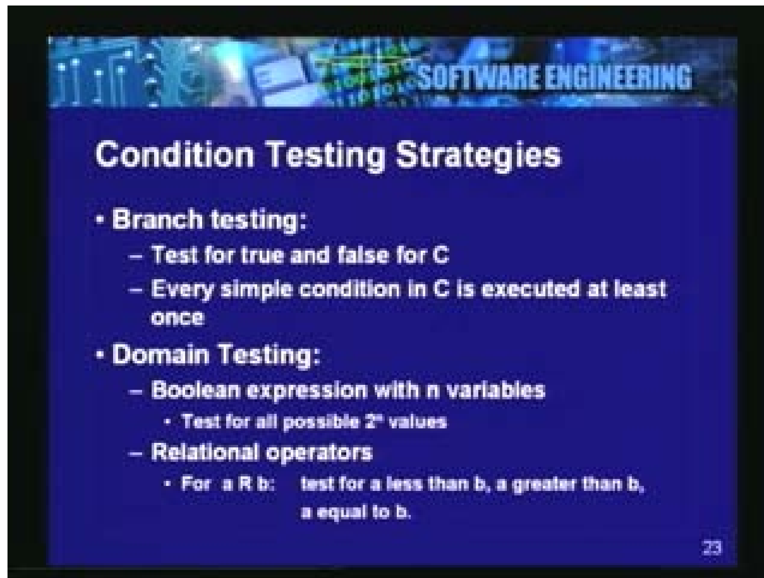
It would not be possible to cover exhaustively for all possibilities and we will have to execute literally the program for all possible cases. Say for example, if you have a loop you may not know how many times you need to execute the loop. If you have a loop from 0 to 100 for a specific value, then loop may not execute correctly (minor break in audio). So, all the exhaustive testing may be impractical, but still these coverage criteria gives us certain level of confidence and there are many such different methodologies and methods, basically approaches to white box testing which you can find in literature. We looked at various criteria for statement coverage and then we looked at the approach from cyclometric complexity point of view and now we will mention a few approaches from the point of view of conditions and composition of conditions.

So, if you look at conditions, they are mainly classified as simple conditions or compound conditions. A simple condition is typically a Boolean variable or it could also have relational operators such as 'a' **less than** 'b', 'a' **greater than or equal to, less than equal** or 'a' **not equal to** 'b' and so on. And a compound condition has a composition of two or more simple conditions, two or more compound conditions. So a compound can be complex and the composition is typically done through Boolean operators.

So, error could occur in conditions due to wrong variable values, and wrong choice of operators, wrong expressions inside the conditions and parenthesis problems. There could be a parenthesis problem where the parenthesis are not closed appropriately or closed in inappropriate places.

Condition testing looks at basically conditions and analyzes the different compositions and derives test suites based on the conditions. For example, you could have a branch testing policy which says that you test true and false for ‘C’. So you have a condition ‘C’, test true and false for ‘C’ and make sure that every simple condition in C is executed at least once.

(Refer Slide Time: 44:22)



So, these criteria could be designed for your approach from point of view of composition of conditions. Or you could take a more exhaustive approach and say that if you have Boolean expression with **n** variables and test for all possible 2 raise to **n** values. This gives you all possible values. For example if you have a, b and c. And we will just make a simple table with these three Boolean variables.

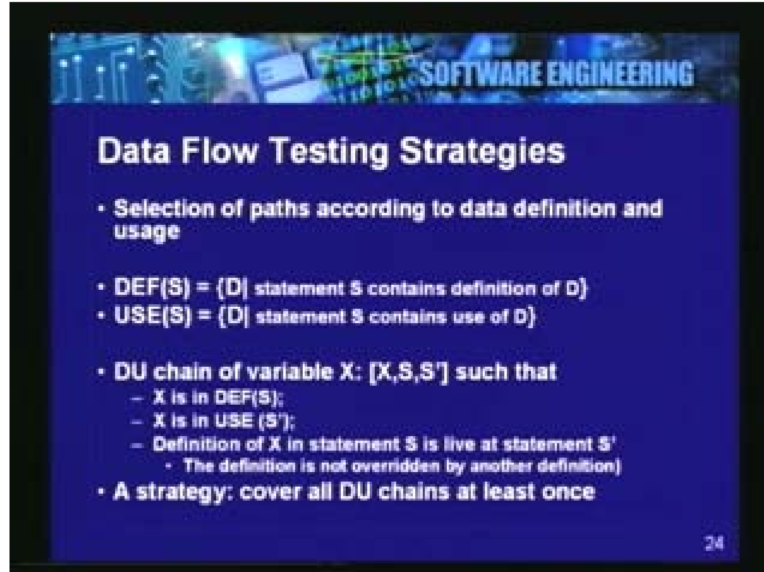
(Refer Slide Time 45:32)

So, how many values can you generate for your test suite that satisfies this criteria of domain testing, that is test for all possible 2 raised to **n** values? You have **n** here as 3 and you can generate 8 different test cases for this conditional coverage criteria.

So this is more exhaustive and you need to generate all possible values and many times it is not possible to test exhaustively for a large program. So, you have reduced your criteria and the path based criteria are quite common. If you have these Boolean expressions, you can generate these test suites with different values of all your Boolean variables and you can simply generate your test suite through a table. And if you have relational operators such as, ‘a’ or ‘b’ **less than or equal to** ‘b’ and so on, test for ‘a’ **less than** ‘b’, ‘a’ **greater than** ‘b’ and ‘a’ **equal to** ‘b’, so that you are able to cover the implicit paths as well.

Condition testing strategies look at mainly the conditions in the program, the branching, and the conditional predicate and composite conditions. And the approach is from the point of view of conditions and their composition. This is another approach to white box testing. And then there are various other approaches such as data flow testing strategies. We basically select path based on the data flows. You find out where a definition and where a data is defined and where it is used.

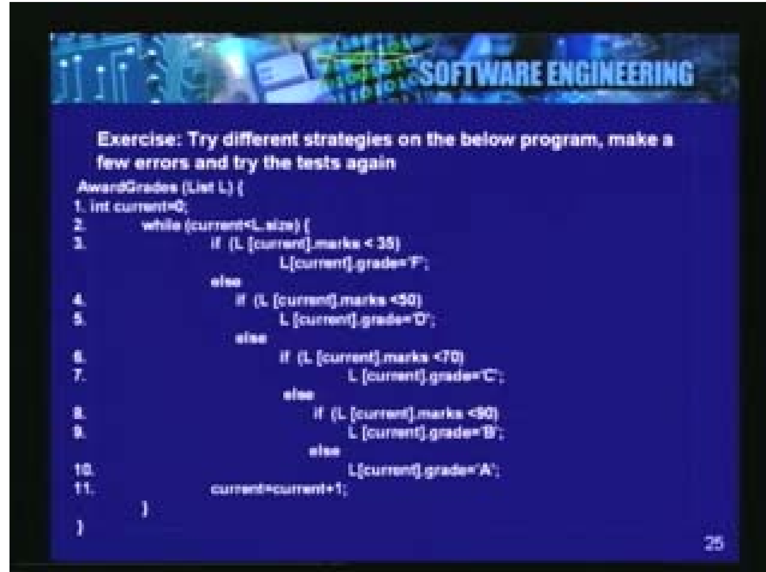
(Refer Slide Time: 46:58)



Based on that definition, DU chains should form. And you try to cover DU chains at least once, where a data is defined and where it is used. If it is defined in one statement and if it is used in two different branches, you must take that definition and generate test path such that both usages are covered. This is another approach and it is mainly from the point of view of data flow. There have been many such approaches you can find and test oracles could be built to analyze your programs and automatically generate test cases and then the validation criteria have to be satisfied. When you run your programs through these test cases which satisfy some of these criteria which we just discussed, you have to validate the output, find out whether generated output is as per the expected.

So, one can write or use tools to analyze a program and generate the tests, which also automatically test this program against the test generated. We have seen through these different examples in slides, how you can approach testing from structural point of view or white box point of view. We are mainly looking at the structure of the program, different conditions used, different branching and the looping constructs and the composition of sequential statements and program has many such. And you if you approach from the basis set or the independent paths point of view or the cyclomatic complexity, you will be able to cover all the statements. But we have also seen that it may not still detect some of the errors. Errors might still go undetected because say for example, absence of features and so on. Now we have an example program for you on this slide. You can try different strategies on below program and make a few errors and then try the test again.

(Refer Slide Time: 49:36)



Not only this program, but you could take any of your programming assignment, data structures and algorithm course or database course or any operating systems and so on. So, take any of your programming assignments which you are coding or which are coded and then apply these criteria and test the program. You probably have not been applying any of these methods formally. So, some of these techniques you should try to apply and see whether by applying these test cases will you be able to detect the bugs. Say if you find a bug you go and try to apply some of these white box and black box techniques and through that see whether you are able to detect your bugs.