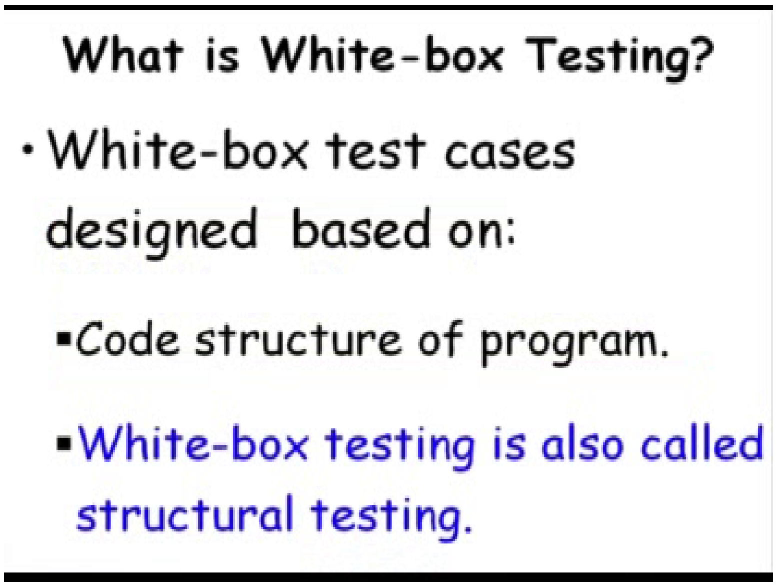
Software Testing
Prof. Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 10
White Box Testing

Welcome to this session. And we have so far looked at some Black-box testing techniques. And now we will start looking at some White-box testing techniques.

(Refer Slide Time: 00:40)

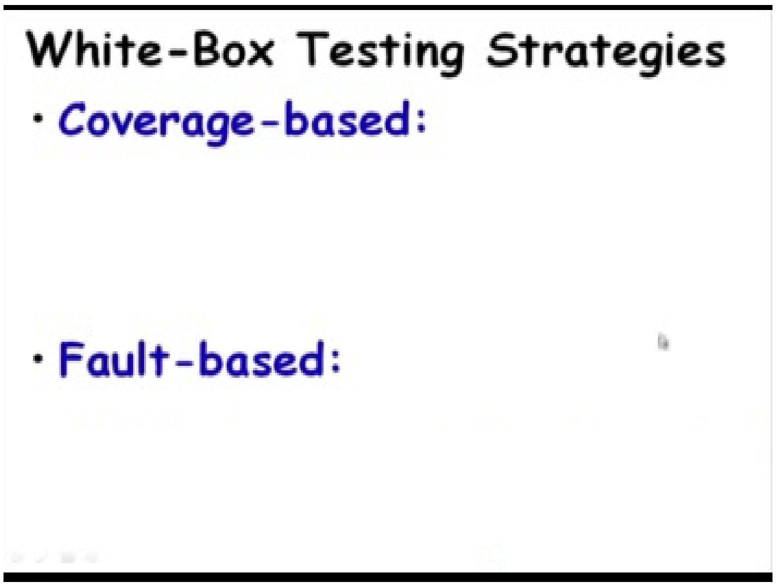


What is White-box Testing?

- White-box test cases designed based on:
  - Code structure of program.
  - White-box testing is also called structural testing.

White-box testing as we had already said involves developing the test cases by examining the code structure. And there are several white-box test strategies that are possible about a dozen of them. We will look about 6, 7 important strategies per white-box testing. These are called as structural testing because the test cases are developed by examining the structure of the code.

(Refer Slide Time: 01:23)

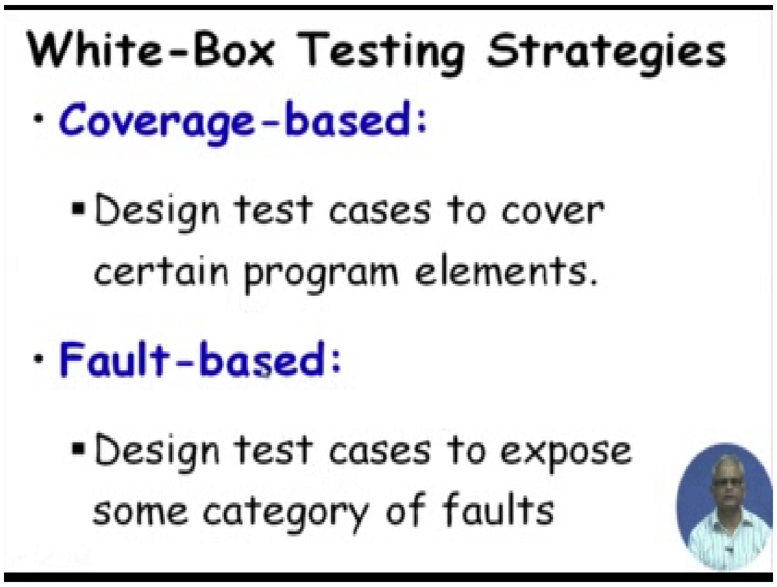

White-Box Testing Strategies

- Coverage-based:
- Fault-based:

We said that there are large numbers of white-box test strategies, but roughly we can classify them into coverage-based ones and fault-based ones. In coverage-based ones, the test cases are designed to achieve coverage of some program elements; we may try to cover statements, we may try to cover conditions, or decisions.

In decision coverage, we remember there is a decision expression involved like an if statement, while statement the test cases should ensure the decision takes both true and false, so that is the program element that is to be covered by decision testing. Whereas in as fault-based testing, we design test cases to detect some specific type of faults, for example, a fault may be that the type of the variable is wrong, int has been made into a float or float into int. So, we will use fault-based testing techniques to check whether those things have been properly detected.

(Refer Slide Time: 02:55)

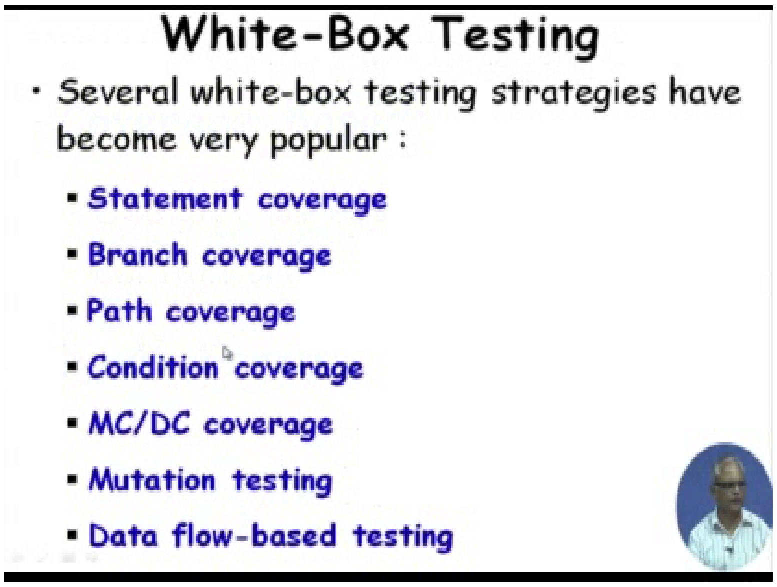


White-Box Testing Strategies

- Coverage-based:
  - Design test cases to cover certain program elements.
- Fault-based:
  - Design test cases to expose some category of faults

In a coverage-based testing, we design test cases to cover program elements; whereas in a fault-based testing we target specific type of faults and check whether those faults those type of faults has been detected.

(Refer Slide Time: 03:12)

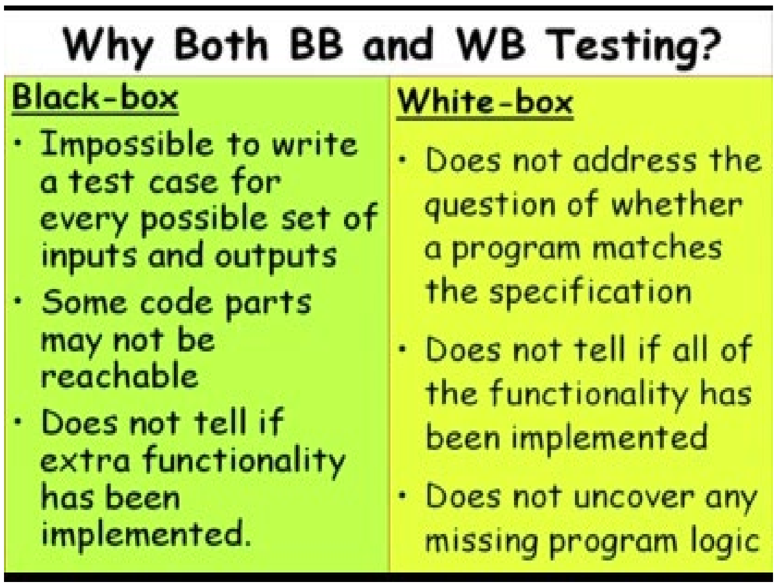


White-Box Testing

- Several white-box testing strategies have become very popular :
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Condition coverage
  - MC/DC coverage
  - Mutation testing
  - Data flow-based testing

As I was saying that there are large numbers of white-box testing techniques. This is just a sample of this - statement coverage, branch coverage, path coverage, multiple condition coverage, multiple condition decision coverage, mutation testing, and data flow testing, and so on. So, let us start looking at the simplest one, which is the statement coverage-based testing.

(Refer Slide Time: 03:35)



Why Both BB and WB Testing?

Black-box	White-box
• Impossible to write a test case for every possible set of inputs and outputs • Some code parts may not be reachable • Does not tell if extra functionality has been implemented.	• Does not address the question of whether a program matches the specification • Does not tell if all of the functionality has been implemented • Does not uncover any missing program logic

But before that lets try to have some very simple concepts about this correct. One thing is we have to be clear whether white-box testing is necessary if you are doing adequate black-box testing. Suppose, we have done a thorough black-box testing, can we say that we will skip white-box testing; or if we done a thorough white-box testing, can we say that black box testing can be skipped. If we say that no, both have to be done then we have to give a valid reason, a valid evidence that if you do not do black-box testing, you will miss out on these kind of problems.

Or if you do not do white-box testing, do only black box testing then black box testing will not be able to detect this kind of problems, I have to give examples. So what do you think is both necessary and if both necessary black-box and white-box testing then you have to give examples of the type of bugs which cannot be detected by black-box testing alone, and also you have to be the example of bugs which cannot be detected by white-box testing alone. Then we can claim that yes, both are necessary.

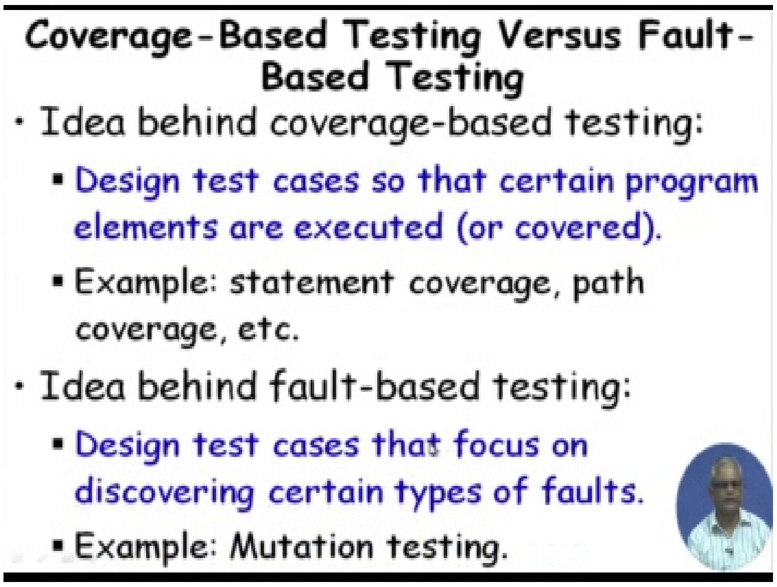
Now, let us look at that. So in black-box testing, we actually create a model of the input data, and generate test cases. But and this is based on the requirements or the description of the functionality of the software, but then we do not have no idea about the code part, so may be the programmers has written some code which is not really part of the requirement.

So only if certain input combination is given those code becomes active and that is not specified in the requirement, so we cannot test that using black-box testing. We do not know what are those input values, because input values are very large, and unless some scenario or something is written about that in the requirements, we cannot really test those input values and that is how Trojans can be implemented. That the programmers has written the code only when certain input values are given then that code works and possibly transmit some data outside and so on. So, using black-box testing, it is impossible to check whether there are Trojans in the code.

And you know what about white-box testing, can you give an example that the kind of box that white-box testing cannot detect; it can only be detected through black-box testing. In a white-box testing, we just look at the code, we do not look at the requirements document, we do not design test cases based on requirement document, only based on the code, identify the structure and then design the test cases.

But what if the code has missed some functionality that should have been there, it is there in the requirements that under this input this action should take place, but then the code has that part missing; no code is there for that. Obviously, by just looking at the code you cannot know that some functionality is missing. So both black-box and white-box testing are necessary, and this are called as complimentary testing strategies.

(Refer Slide Time: 08:04)



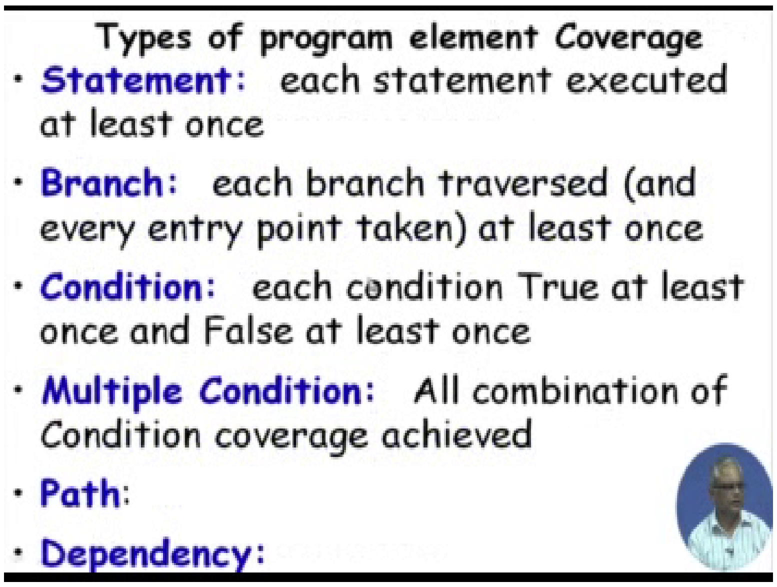
Coverage-Based Testing Versus Fault-Based Testing

- Idea behind coverage-based testing:
  - Design test cases so that certain program elements are executed (or covered).
  - Example: statement coverage, path coverage, etc.
- Idea behind fault-based testing:
  - Design test cases that focus on discovering certain types of faults.
  - Example: Mutation testing.

Now, let us look at the coverage-based testing, let us look further into the white-box testing, the coverage-based testing and the fault-based testing. In coverage-based testing, we said that we write the test cases such that some program elements are covered. And if sufficient coverage is achieved, we would have performed testing for those strategies examples are statement coverage, path coverage etcetera.

And in idea behind fault-based testing, we identify the type of faults that programmers commit normally and then check whether those faults are existing. So, one example of this is mutation testing; we look at both coverage-based testing several of them, and also we look at fault-based testing which is the mutation testing.

(Refer Slide Time: 09:03)



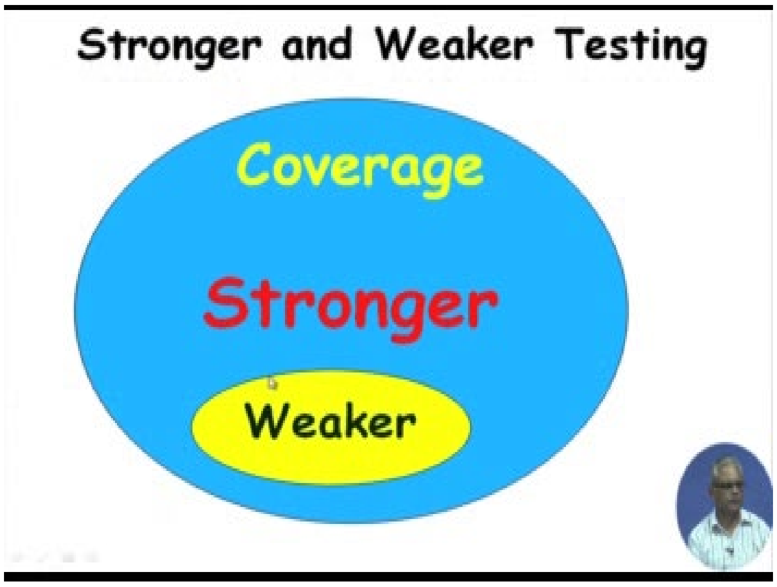
Types of program element Coverage

- **Statement:** each statement executed at least once
- **Branch:** each branch traversed (and every entry point taken) at least once
- **Condition:** each condition True at least once and False at least once
- **Multiple Condition:** All combination of Condition coverage achieved
- **Path:**
- **Dependency:**

And the coverage-based testing, we have different strategies depending on the specific program elements that we target. For example, in statement coverage, the program elements is statement; in branch coverage, which is also called as decision coverage here we check whether every branch condition, assumes true and false values. Condition coverage is also called as multiple condition coverage; so here whether the component conditions of every compound condition, so decision expression may be having compound conditions c1 and c2 or c3 etcetera, whether each of this c1, c2, c3 take true and false values.

Multiple condition, whether all possible combinations of the input conditions are considered, path coverage, dependency coverage and so on. There are different types of program elements that different coverage strategies target. These target statement, branch, condition, multiple conditions that is all possible combinations, program paths whether program paths are covered, dependency whether data dependency etcetera are covered, so we would have different strategies for white-box testing depending on the program element coverage that we attempt.

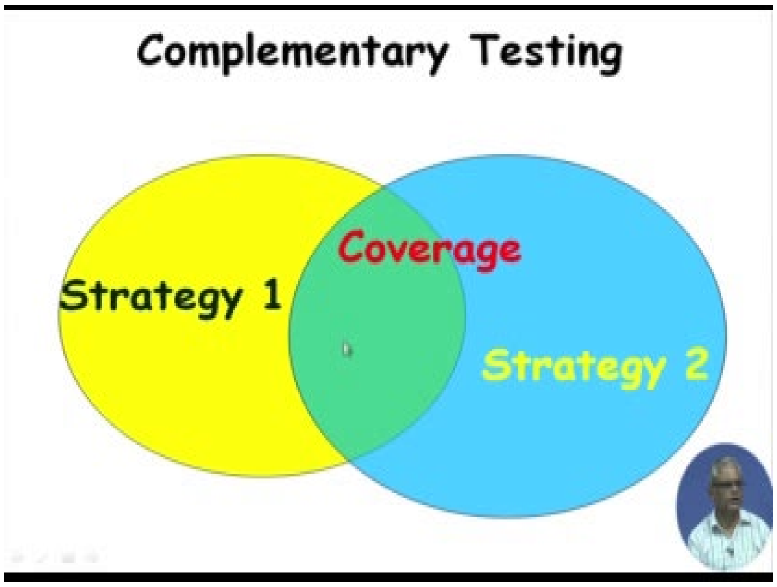
(Refer Slide Time: 10:44)



Stronger and Weaker Testing

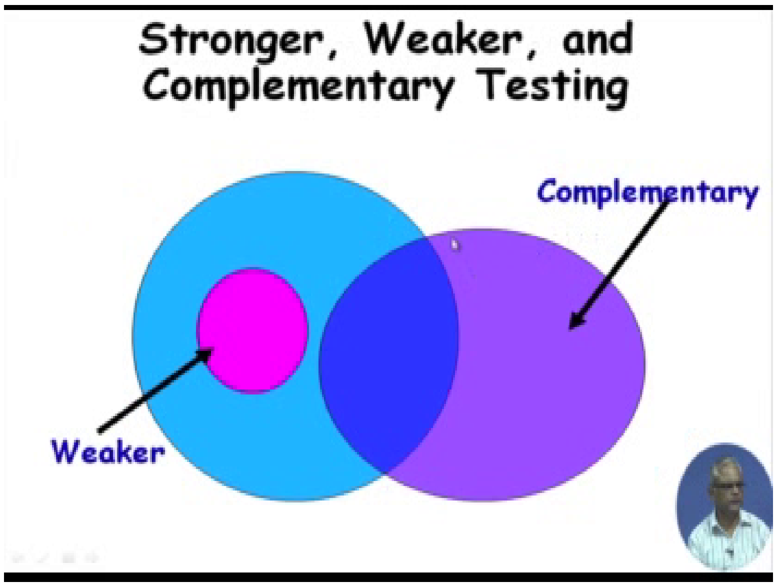
We have a notion of a stronger and weaker testing in coverage-based testing. If all the program elements that are covered by one strategy, guarantees to cover the program elements covered by another strategy then we have this stronger coverage, and the one where it covers only a subset of the program elements of their stronger, we call it as a weaker testing.

(Refer Slide Time: 11:28)


Complementary Testing

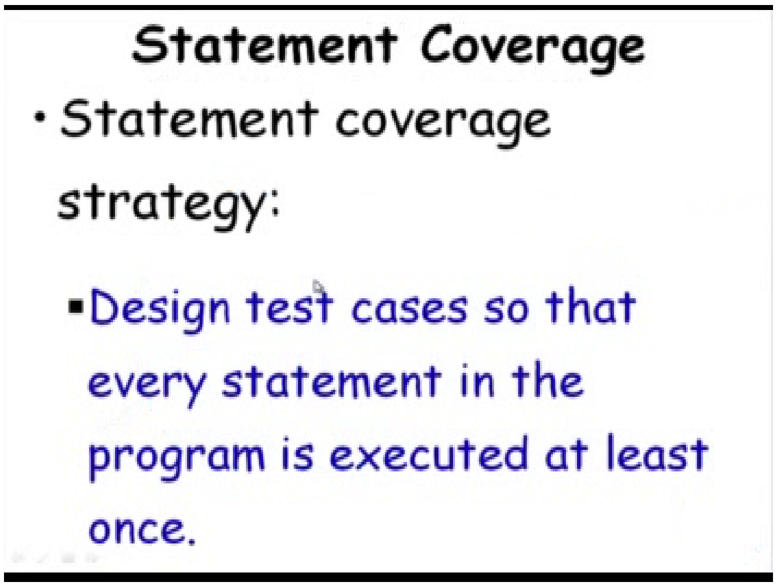
And we call complimentary testing, if two strategies, they do cover some program elements in common, but they also cover different program elements.

(Refer Slide Time: 11:40)


Stronger, Weaker, and Complementary Testing

So, this is the notion of a weaker and complimentary.

(Refer Slide Time: 11:51)

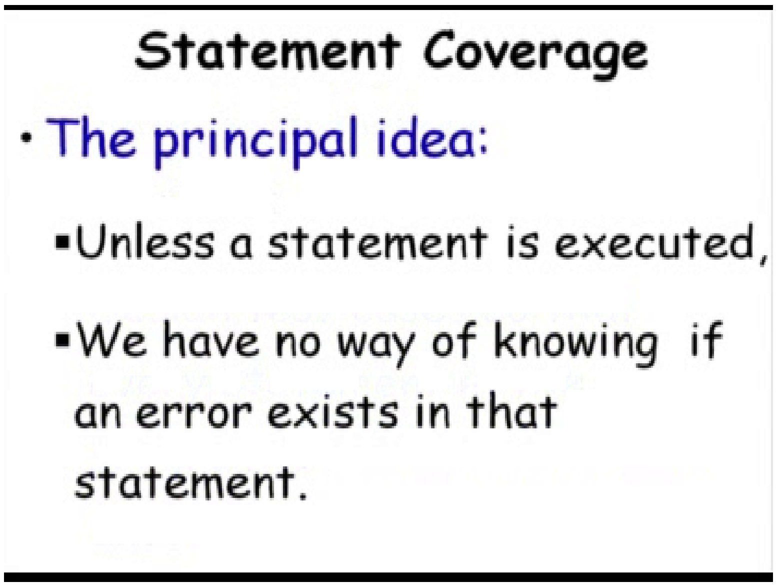

Statement Coverage

- Statement coverage strategy:
  - Design test cases so that every statement in the program is executed at least once.

Now, let us look at the simplest testing strategy which is the statement coverage testing. So, here the main motive, the main objective is to write test cases. Such that every program statement is exercised at list once. Let us look at a simple white-box testing, coverage-based testing which is the statement coverage; this is simplest testing.

The idea here in the statement coverage testing is that every statement needs to be covered by the set of test cases, because unless a statement is executed at least once you do not know if there is a problem existing in that statement, so that is the main idea here. We write test cases such that every statement is executed or covered at least once, because unless a statement is executed, you do not know if there is a problem existing in that statement.

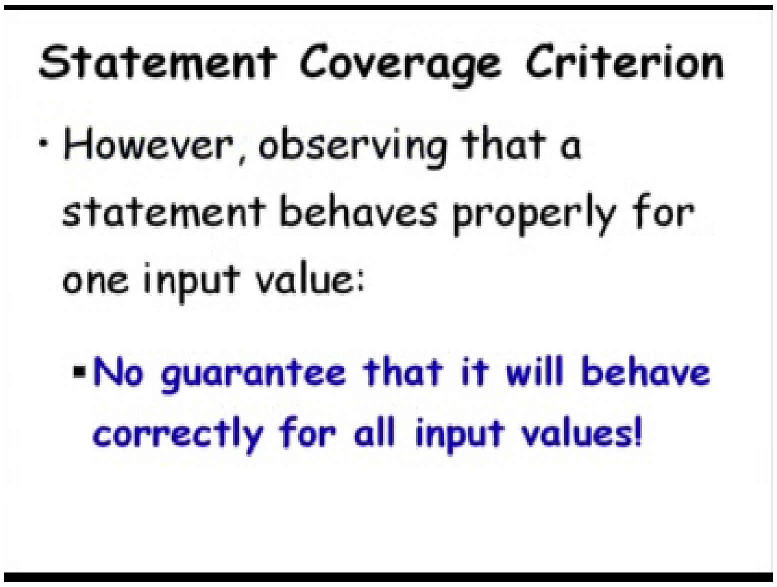
(Refer Slide Time: 13:08)


Statement Coverage

- The principal idea:
  - Unless a statement is executed,
  - We have no way of knowing if an error exists in that statement.

So, we keep on giving test cases, and observe, or measure, whether statements are getting executed. And if all the statements are executed, then we know that 100 percent statement coverage is achieved.

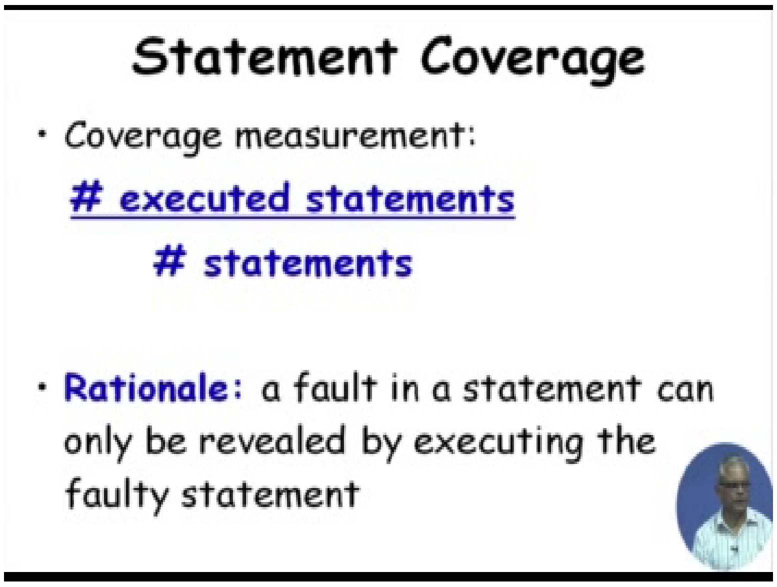
(Refer Slide Time: 13:31)


Statement Coverage Criterion

- However, observing that a statement behaves properly for one input value:
  - **No guarantee that it will behave correctly for all input values!**

So, that is what we have written here that unless statement is observed to work correctly at least for one test case, we do not know whether it will work. And also the problem with statement coverage is that just by observing a statement works fine for one input does not guarantee that; for other inputs also it will work, but at least you would have known that at least that statement works good for one of the inputs.

(Refer Slide Time: 14:18)



Statement Coverage

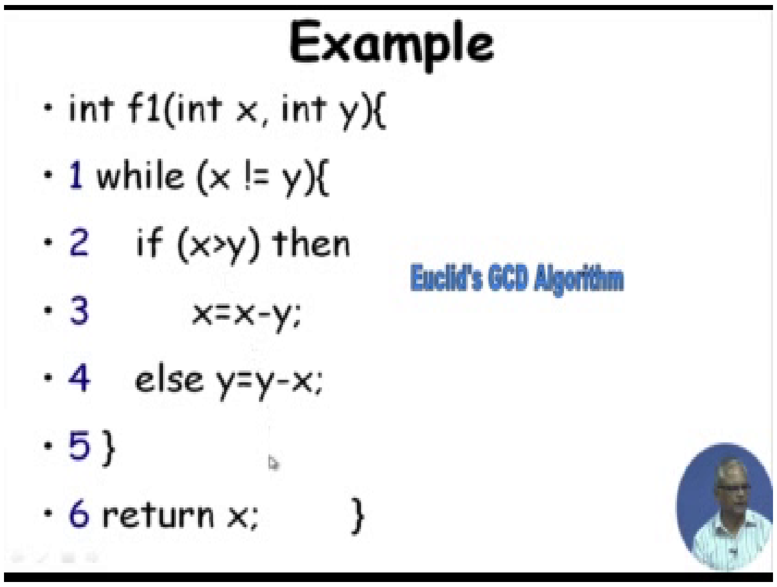
- Coverage measurement:

$$\frac{\text{\# executed statements}}{\text{\# statements}}$$

- **Rationale:** a fault in a statement can only be revealed by executing the faulty statement

Here we have this metric, percentage covered, percentage statement covered which is the total number of statements and out of which how many statements have got executed. So, you can say that 80 percent statement coverage, the 90 percent statement coverage depending on what fractions of the statements have got executed. And normally we expect 100 percent statement coverage unless we have unreachable code; we need 100 percent statement coverage.

(Refer Slide Time: 14:59)



Example

- int f1(int x, int y){
- 1 while (x != y){
- 2   if (x>y) then
- 3   x=x-y;
- 4   else y=y-x;
- 5 }
- 6 return x;   }

Euclid's GCD Algorithm

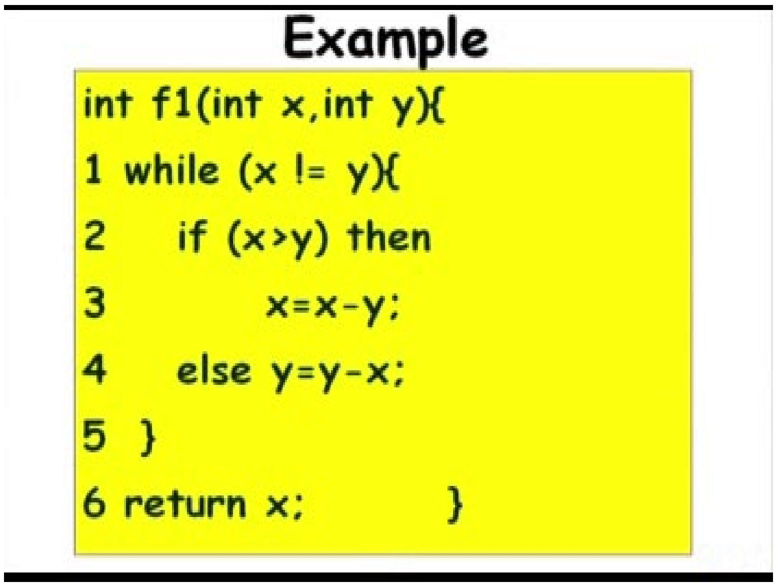
Let us look at this example, so this is famous algorithm here just code I just taken that code here. It takes two parameter x and y, while x != y. If x > y then x = x - y, else y = y - x. I think when you would have recognized this algorithm it is the famous GCD computing algorithm Euclid.

And there are 6 statements here; and we need to have the test cases which will have all the statements covered. To check whether all statements are covered, we can just write a small tool and also tools are available, open source tools like g curve available, but we can write our own tool, which can check what is the percentage of the statements that are executed.

How do we do that, we can set we can have an array with number of statements equal to the elements of the array, one of the techniques. We can have many ways to check statement coverage. And whenever a specific statement is covered is executed, we would have inserted a statement here to set the corresponding variable in the array.

So, if the third one then we would have statement just after this, set the corresponding variable in the array, but one thing is that if there are no branches, all the statements will get covered so that is how we can write in a efficient tool. That as long as it enters a block of code, it will execute all the statements there, so we need to only mark the blocks. It is possible to write a small tool ourselves which can check the percentage statement coverage, but then there are open source tools available which can report the percentage of statement executed.

(Refer Slide Time: 17:40)



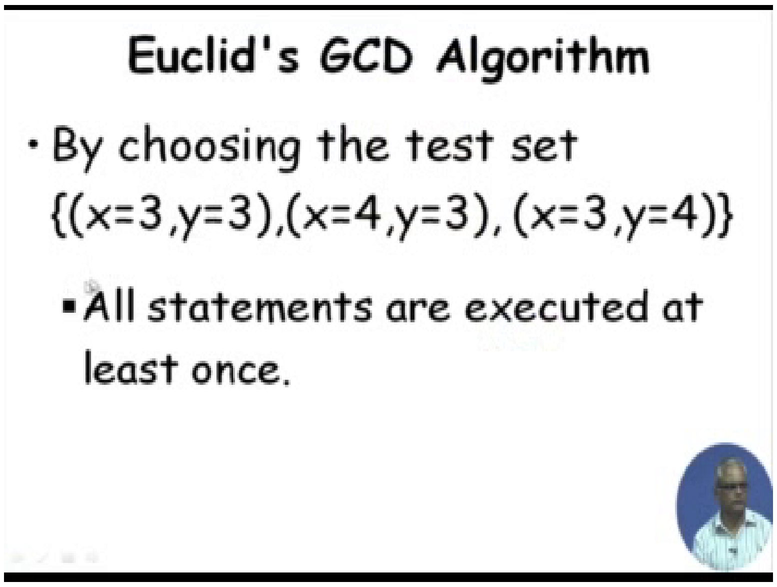
Example

```
int f1(int x,int y){
1 while (x != y){
2    if (x>y) then
3          x=x-y;
4    else y=y-x;
5 }
6 return x;        }
```

Normally we do not design test cases in our practical programs to achieve statement coverage. We just give input values randomly until we achieve 100 percent statement coverage. But then for very small programs like 4 lines, 5 lines, it is possible to find out what are the values to be given such that we would have 100 percent statement coverage. These very small program; and we know that this expression as to be true for this two enter into the loop.

And therefore, we must have x != y that is the input we have to give. And then also we have to give input such that x > y, such and that that can come here, and also have to give input such that x <= y. So, for very small programs, it is possible to design a test suite, which will cover all the statements. But as I said for larger programs, we do not really design test cases to achieve statement coverage, we give random input and keep on checking what is the coverage achieved. The open search tool or our own tool, you can check what is the coverage achieved each time we execute a test case, and we keep on giving random inputs until we get 100 percent statement coverage.

(Refer Slide Time: 19:33)

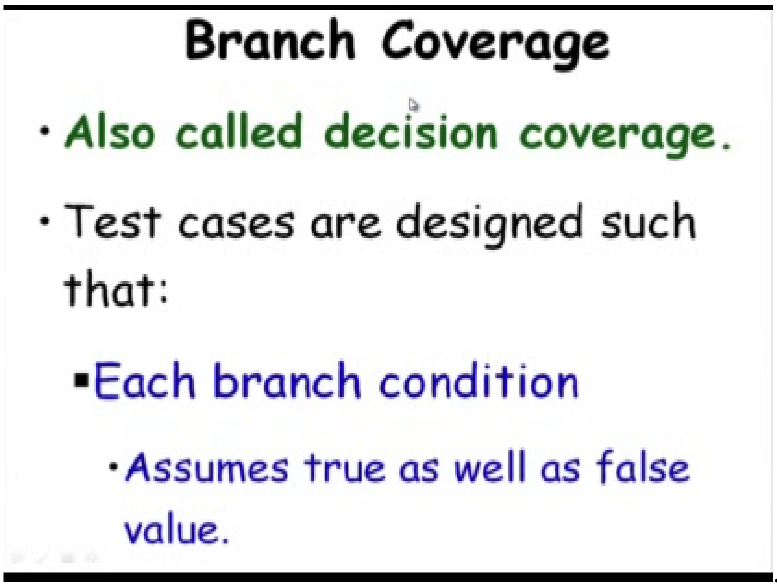


Euclid's GCD Algorithm

- By choosing the test set {(x=3,y=3),(x=4,y=3), (x=3,y=4)}
  - All statements are executed at least once.

So, if you manually design test cases, we will see that for that small program, these are the values with which we can achieve 100 percent statement coverage.

(Refer Slide Time: 19:47)



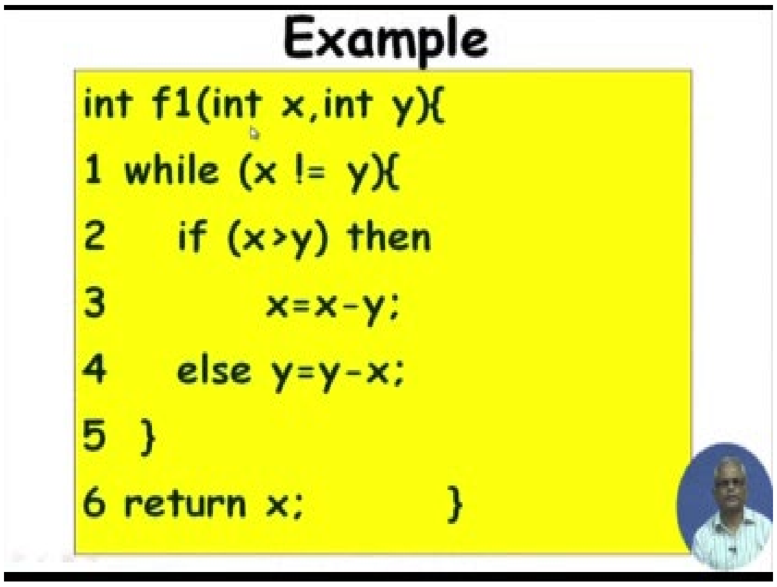
Branch Coverage

- Also called decision coverage.
- Test cases are designed such that:
  - Each branch condition
    - Assumes true as well as false value.

Statement coverage is a very simple concepts, but also it is a least effective in detecting bugs, because it is one of the weakest testing, weakest white-box testing. Let us look at stronger white-box testing. One of the stronger testing is the branch coverage or decision coverage.

In decision coverage, test cases are designed such that each branch conditions assumes true and false values. So, whenever we have a decision statement either in the form of if, while, for and so on, we check whether each of these decisions assumes both true value and false value, then we say that 100 percent branch coverage is achieved.

(Refer Slide Time: 20:51)

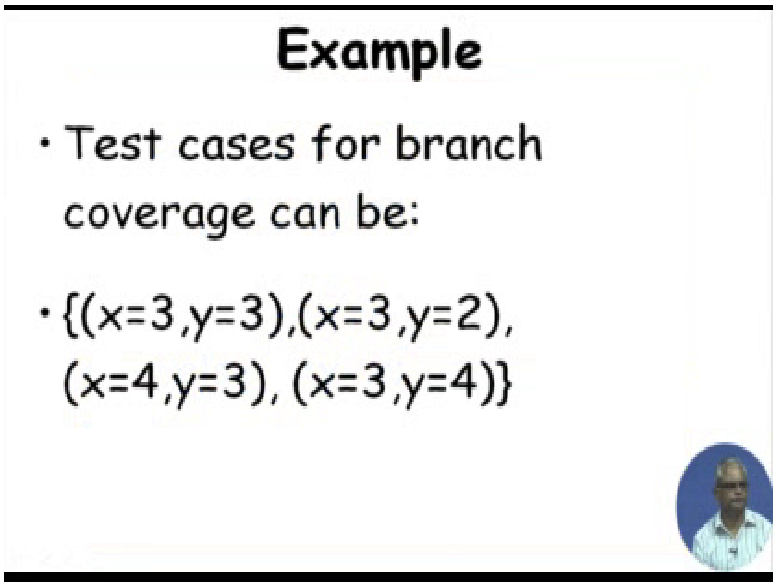


Example

```
int f1(int x,int y){
1 while (x != y){
2     if (x>y) then
3          x=x-y;
4     else y=y-x;
5 }
6 return x;        }
```

So for the same program, we would need there are branches here decisions here and while if and will check if while is both true and false, the condition here is both true and false. This condition for the ‘if’ is both true and false, we will need that the test cases would achieve for this expression to become true and false and this expression to become true and false and then will say that branch coverage is achieved.

(Refer Slide Time: 21:28)



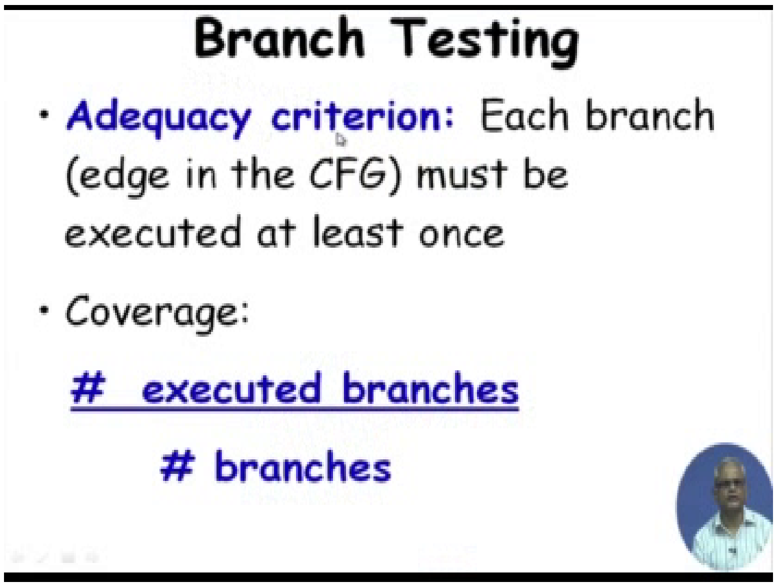
Example

- Test cases for branch coverage can be:
- {(x=3,y=3),(x=3,y=2), (x=4,y=3), (x=3,y=4)}

So, for very small programs, again we can generate or we can write test cases for achieving branch coverage. So, we can see that this set of values will set both those conditions true and false, but for larger complicated programs, it is very hard to design test cases to achieve branch coverage.

And in practice, nobody really designs test cases to achieve branch coverage they would give input values randomly, and have a coverage tool which will check the branch coverage achieved. Here again checking branch coverage, how much branch coverage achieved writing a tool is very simple, and open source tools are available, which can report what is a branch coverage that is achieved.

(Refer Slide Time: 22:25)



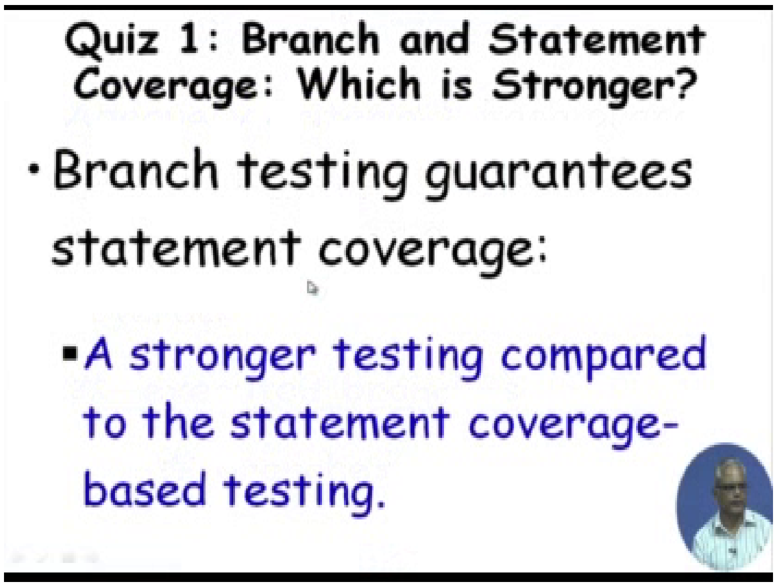
Branch Testing

- Adequacy criterion: Each branch (edge in the CFG) must be executed at least once
- Coverage:

$$\frac{\text{\# executed branches}}{\text{\# branches}}$$

So, the percent is a branch coverage achieved we can write is number of executed branches divided by the total number of branches possible. So in the program Euclid GCD, there are two branch conditions, so we would have four branch outcomes, four possible branch outcomes and depending on how many of those have taken, we would have number of executed branches. So it can be 1 by 4, 25 percent or 2 by 4 - 50 percent, or 75 percent, or 100 percent.

(Refer Slide Time: 23:10)



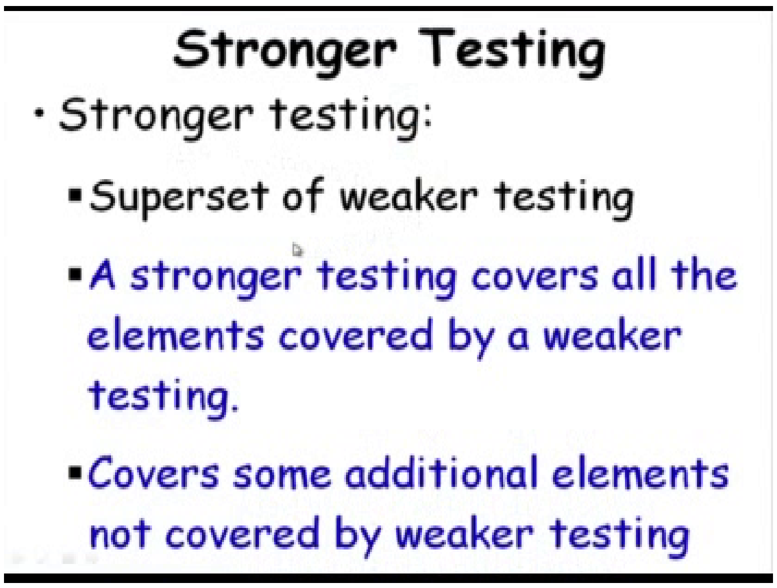
Quiz 1: Branch and Statement Coverage: Which is Stronger?

- Branch testing guarantees statement coverage:
  - A stronger testing compared to the statement coverage-based testing.

But then after discussing two white-box testing strategies - statement coverage and branch coverage, can we say which is stronger testing. Because the idea is that if you are doing a stronger testing, we do not have to do a weaker testing. A stronger testing would have covered all the program elements covered by a weaker testing. And therefore, as long as we are doing a stronger testing, let say there are several weaker testing, and we are doing one stronger test strong test, we do not have to do all those weaker test, just one strong test is good enough. So, between the branch and statement coverage which is stronger.

So if you want to show that branch coverage is stronger then we have to show that branch coverage will guarantee the statement coverage. And also we have to show that statement coverage does not achieve branch coverage, so there is at least one branch which is not covered by statement coverage; otherwise, they will be the same. So, we have to see one way, we have to show that branch coverage achieves statement coverage; and also we have to show that statement coverage does not achieve branch coverage; at least some branches are not covered by statements.

(Refer Slide Time: 24:53)



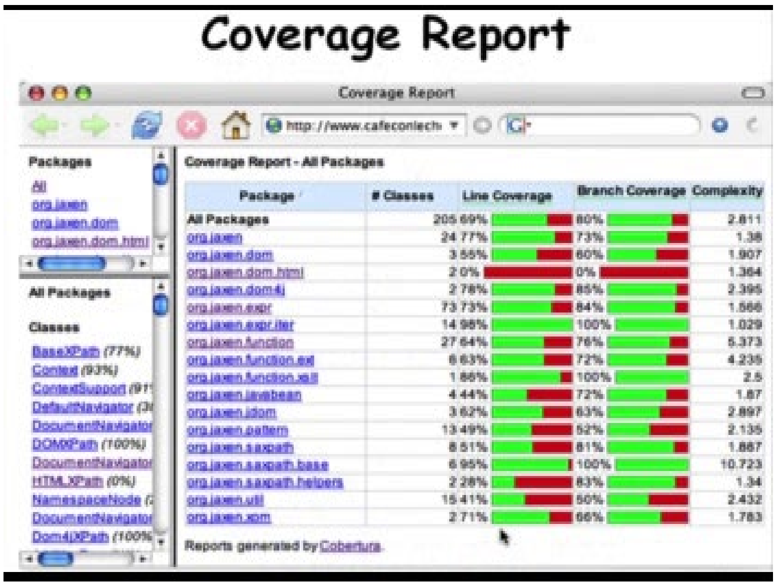
Stronger Testing

- Stronger testing:
  - Superset of weaker testing
  - A stronger testing covers all the elements covered by a weaker testing.
  - Covers some additional elements not covered by weaker testing

So how do we show that? So, we can give a argument here that in a stronger testing, we have to show that it covers all elements of the weaker testing, we can argue that every statement lies on some branch. So, if all branches are covered all statements must have been covered.

Let me just repeat this argument that every statement lies on some branch, so if all branches are covered then all statements must have been covered, so that shows that branch coverage would achieve statement coverage, but how do we show that statement coverage does not achieve branch coverage. So, for that, we can give one example at least one example, where we show that we achieve statement coverage, but that does not achieve branch coverage that would show that statement coverage does not achieve branch coverage.

(Refer Slide Time: 26:10)



Coverage Report

Coverage Report

http://www.cafeconlech

Packages
All
org.jaxen
org.jaxen.dom
org.jaxen.dom.html

All Packages

Classes
BaseXPath (77%)
Context (93%)
ContextSupport (91
DefaultNavigator (3
DocumentNavigator
DOMXPath (100%)
DocumentNavigator
HTMLXPath (0%)
NamespaceNode (
DocumentNavigator
Dom4jXPath (100%

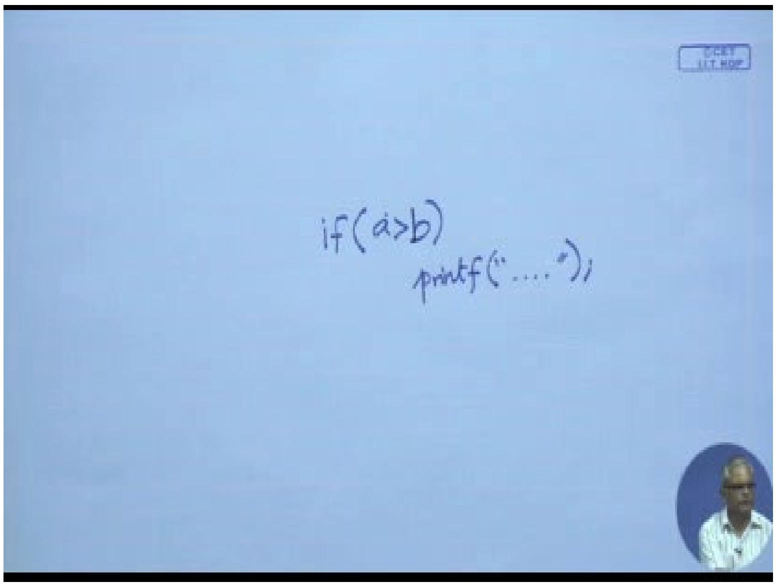
Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	205	69%	80%	2.811
org.jaxen	24	77%	73%	1.38
org.jaxen.dom	3	55%	60%	1.907
org.jaxen.dom.html	2	0%	0%	1.364
org.jaxen.dom4j	2	78%	85%	2.395
org.jaxen.expr	73	73%	84%	1.566
org.jaxen.expr.iter	14	98%	100%	1.029
org.jaxen.function	27	64%	76%	5.373
org.jaxen.function.ext	6	63%	72%	4.235
org.jaxen.function.xslt	1	86%	100%	2.5
org.jaxen.javabean	4	44%	72%	1.87
org.jaxen.jdom	3	62%	63%	2.897
org.jaxen.pattern	13	49%	52%	2.135
org.jaxen.saxpath	8	51%	81%	1.887
org.jaxen.saxpath.base	6	95%	100%	10.723
org.jaxen.saxpath.helpers	2	28%	83%	1.34
org.jaxen.util	15	41%	50%	2.432
org.jaxen.xom	2	71%	66%	1.783

Reports generated by Cobertura.

So, I do not really have a slide, the example is not written there in the slide, but just asking you to give an example where statement coverage does not give branch coverage.

(Refer Slide Time: 26:38)

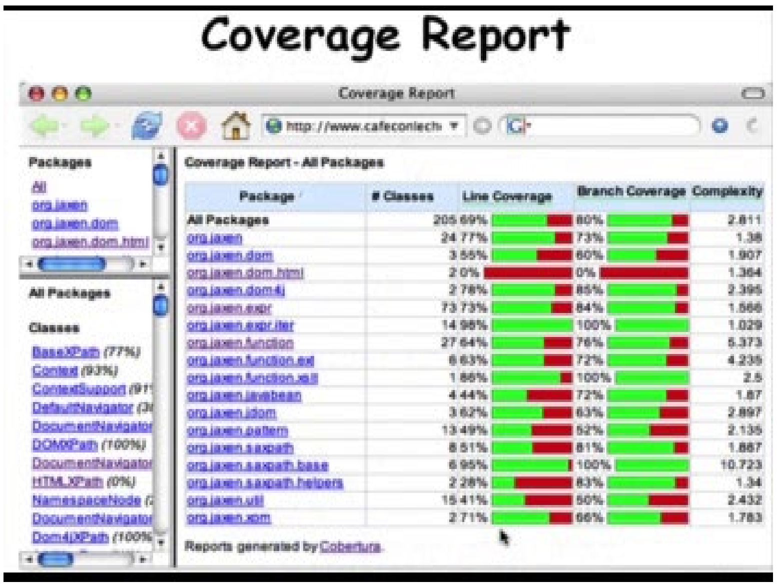


```
if(a>b)
    printf("....");
```

One simple example could be if a greater than b, printf something. So, here if a greater than b, printf is executed. Statement coverage achieved as long as a greater than b. So for any value of a, which is greater than b, we will have statement coverage. But decision coverage or branch coverage will not be achieved, because we are also needs test case which sets an equal to or less than b.

So the test case a greater than b, achieves statement coverage, but does not achieve branch cover, and this is a good enough example to show that statement coverage does not achieve branch coverage. And as I was saying that there are several tools that are available which will give open source tools are also there many of them.

(Refer Slide Time: 27:52)



Coverage Report

Coverage Report

http://www.cafeconlech

Packages
All
org.jaxen
org.jaxen.dom
org.jaxen.dom.html

All Packages

Classes
BaseXPath (77%)
Context (93%)
ContextSupport (91
DefaultNavigator (3
DocumentNavigator
DOMXPath (100%)
DocumentNavigator
HTMLXPath (0%)
NamespaceNode (
DocumentNavigator
Dom4jXPath (100%

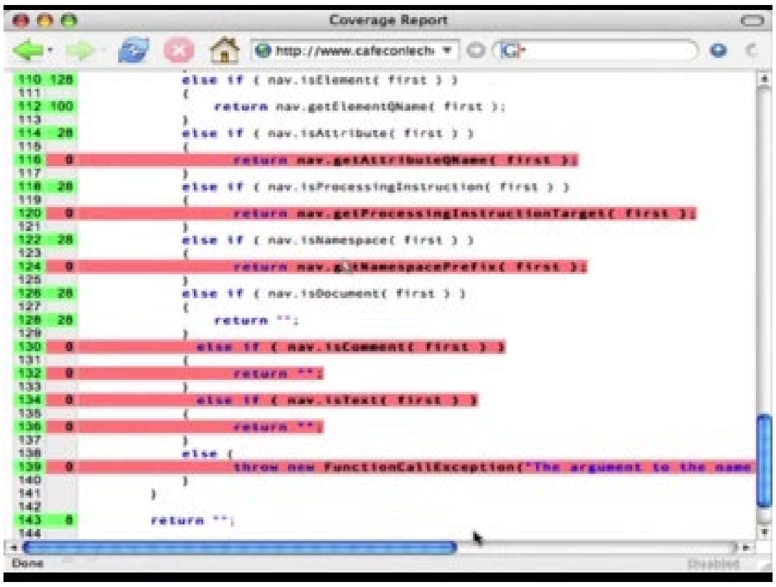
Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	205	69%	80%	2.811
org.jaxen	24	77%	73%	1.38
org.jaxen.dom	3	55%	60%	1.907
org.jaxen.dom.html	2	0%	0%	1.364
org.jaxen.dom4j	2	78%	85%	2.395
org.jaxen.expr	73	73%	84%	1.566
org.jaxen.expr.iter	14	98%	100%	1.029
org.jaxen.function	27	64%	76%	5.373
org.jaxen.function.ext	6	63%	72%	4.235
org.jaxen.function.xslt	1	86%	100%	2.5
org.jaxen.javabean	4	44%	72%	1.87
org.jaxen.jdom	3	62%	63%	2.897
org.jaxen.pattern	13	49%	52%	2.135
org.jaxen.saxpath	8	51%	81%	1.887
org.jaxen.saxpath.base	6	95%	100%	10.723
org.jaxen.saxpath.helpers	2	28%	83%	1.34
org.jaxen.util	15	41%	50%	2.432
org.jaxen.xom	2	71%	66%	1.783

Reports generated by Cobertura.

Which can give the coverage, statement coverage, branch coverage so this tool after some test cases are executed gives that for which package, how much of line coverage how much of branch coverage is achieved.

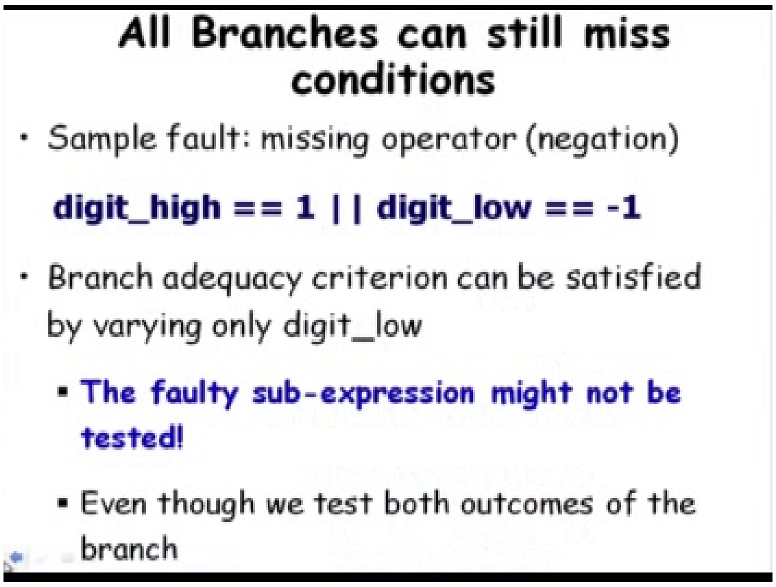
(Refer Slide Time: 28:14)



```
110 128          else if ( nav.isElement( first ) )
111              {
112 100              return nav.getElementQName( first );
113              }
114  28          else if ( nav.isAttribute( first ) )
115              {
116   0                return nav.getAttributeQName( first );
117              }
118  28          else if ( nav.isProcessingInstruction( first ) )
119              {
120   0                return nav.getProcessingInstructionTarget( first );
121              }
122  28          else if ( nav.isNamespace( first ) )
123              {
124   0                return nav.getNamespacePrefix( first );
125              }
126  28          else if ( nav.isDocument( first ) )
127              {
128  28              return "";
129              }
130   0           else if ( nav.isComment( first ) )
131              {
132   0                return "";
133              }
134   0           else if ( nav.isText( first ) )
135              {
136   0                return "";
137              }
138              else {
139   0                throw new FunctionCallException("The argument to the name
140              }
141          }
142
143   8      return "";
144
```

And not only that, it also shows which statements have not been executed. But, if we do statement, branch coverage, you do not have to do statement coverage that we already have said. But is it possible that a program where you have achieved 100 percent branch coverage still bugs can exist; and what are those bugs, can we give an example a bug which can exist even if we achieve 100 percent branch coverage. And how do we design a white-box test strategy, which will check those bugs.

(Refer Slide Time: 29:03)



All Branches can still miss conditions

- Sample fault: missing operator (negation)

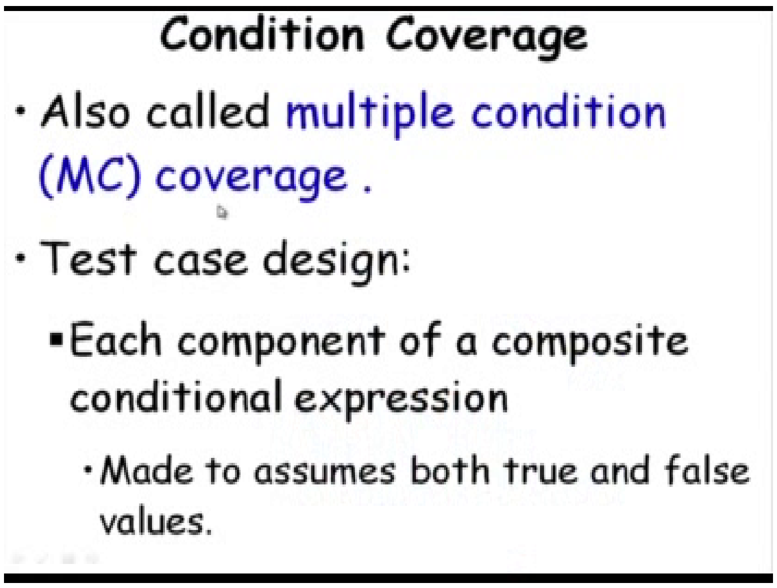
  digit_high == 1 | | digit_low == -1

- Branch adequacy criterion can be satisfied by varying only digit_low
  - **The faulty sub-expression might not be tested!**
  - Even though we test both outcomes of the branch

Let us look at one example that we have an expression here in one of the conditionals if digit high is true or digit low is false that is - 1, then do something. Now for branch coverage, if one of these parameters is true, let say digit high is true and digit low we can set it to true and false, then branch coverage will get achieved. So, using only the digit lows true and false values, when the digit high is true we will achieve, sorry when the digit high is false, the digit low is true and false value will achieve both the true and false for this branch.

There is a or expression here, so as long as this is false, true and false will make this expression - entire expression true or false. And therefore, branch coverage would be achieved, but what if we have a bug which occurs only when the digit high is true. We achieve branch coverage by keeping digit high to be false and digit low to true and false; but, what if the bug occurs only when the digit high is true. So achieving branch coverage, 100 percent branch coverage would not be able to detect this bug.

(Refer Slide Time: 31:03)

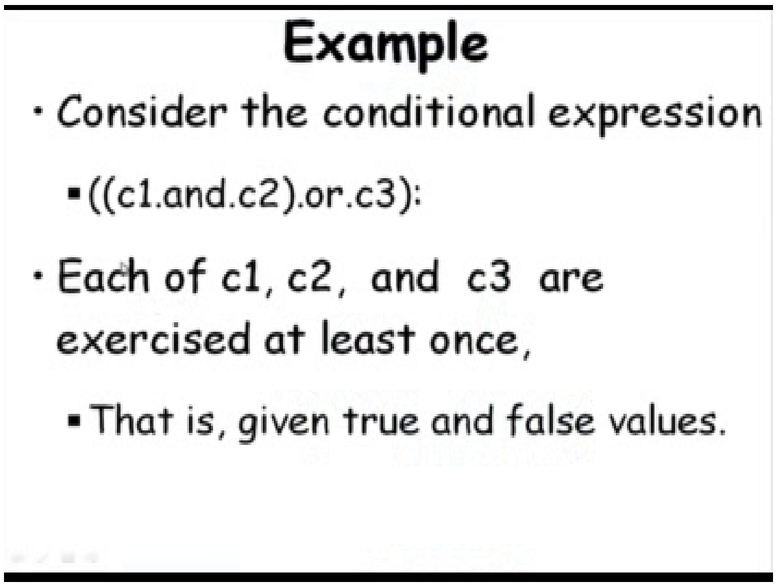


Condition Coverage

- Also called multiple condition (MC) coverage .
- Test case design:
  - Each component of a composite conditional expression
    - Made to assumes both true and false values.

So, what do we do what type of strategies, test strategies do we deploy, we have to do condition coverage, also called as multiple condition coverage. So, each component condition is takes true and false value. So, when there is only one condition in expression then branch coverage and condition coverage at the same, but when there are multiple conditions - sub conditions in a conditional expression then just achieving branch coverage may not be able to detect all bugs. We have to consider the individual component conditions - the sub conditions to be given true and false values.

(Refer Slide Time: 31:56)

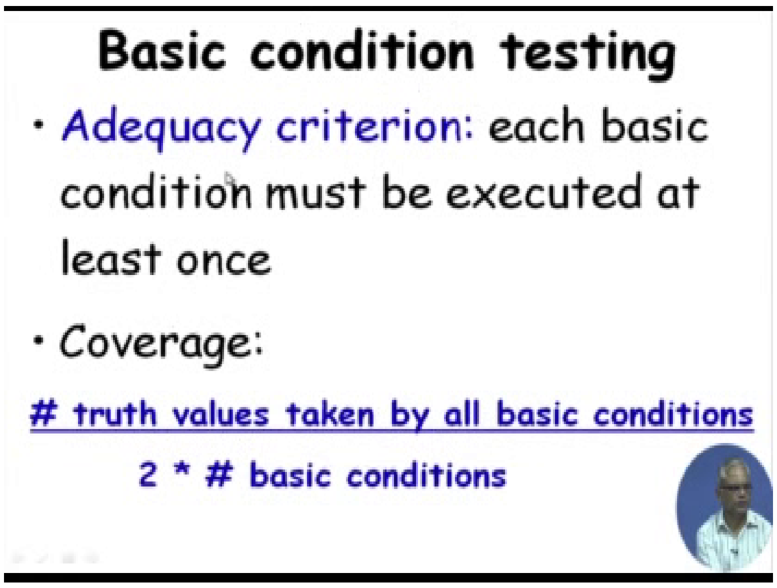


Example

- Consider the conditional expression
  - ((c1.and.c2).or.c3):
- Each of c1, c2, and c3 are exercised at least once,
  - That is, given true and false values.

Let us look at one example c 1 and c 2 or c 3. So in condition coverage, we need to have both c 1 and c 2 true and false, c 2 true and false, and c 3 true and false. And in multiple condition coverage, you have to consider all possible combinations of conditions of c 1, c 2, c 3; true, true, true; true, false, true; true, false, false; false, true, true etcetera every combination of the conditions we need to consider. And for multiple conditions, if there are n component conditions, each one taking two values true and false, we need 2^n possible test cases.

(Refer Slide Time: 32:50)



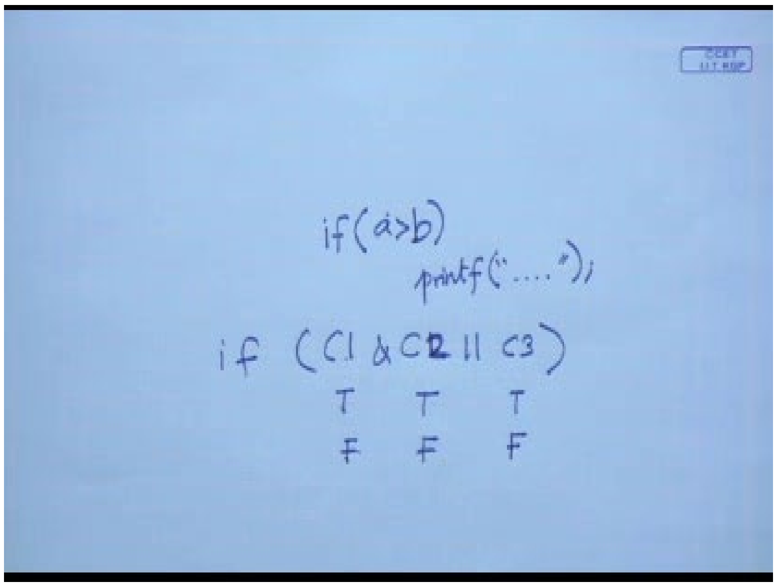
Basic condition testing

- Adequacy criterion: each basic condition must be executed at least once
- Coverage:

$$\frac{\text{\# truth values taken by all basic conditions}}{2 * \text{\# basic conditions}}$$

For basic condition testing, where we test whether each condition at least has taken true and false values, we do not take all possible combinations of conditions.

(Refer Slide Time: 33:12)

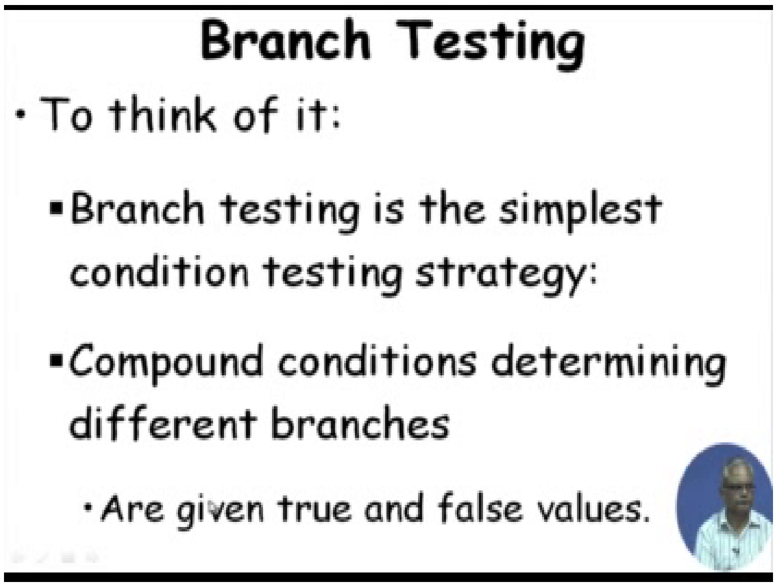


```
if(a>b)
      printf("....");
if ((c1 & c2 || c3)
     T    T    T
     F    F    F
```

So what I am saying here is that if c 1 and c 2 or c 3, if this is the condition then as long as this is true, true, true and false, false, false only two of these will achieve basic condition testing. Because each has taken true and false values or we can have a test case which makes true, false, true, and false, true, false so that will ensure that each condition takes both true and false values.

So, the number of values for this condition is 2 into n. If n basic conditions here, we would need 2 into n and the percent is covered is the truth value taken by all basic conditions divided by 2 into number of basic conditions.

(Refer Slide Time: 34:19)

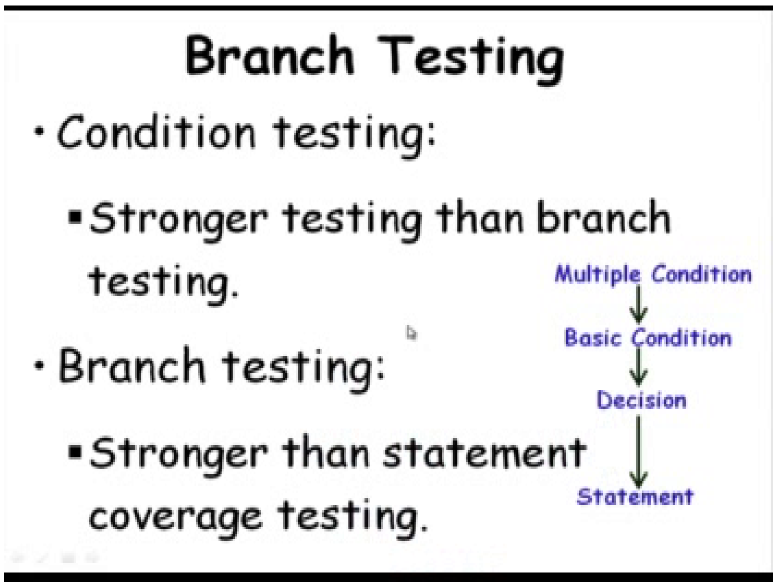


Branch Testing

- To think of it:
  - Branch testing is the simplest condition testing strategy:
  - Compound conditions determining different branches
    - Are given true and false values.

And if we relate these to branch testing, the branch testing is actually the simplest condition testing strategy, where we assume that the entire result of the condition is true and false. We do not ensure that all component conditions have taken true and false. So, the branch testing is the simplest strategy is condition strategy, and the basic condition is stronger strategy then branch testing, that would be very easy to show.

(Refer Slide Time: 35:02)



Branch Testing

- Condition testing:
  - Stronger testing than branch testing.
- Branch testing:
  - Stronger than statement coverage testing.

$$\begin{tikzcd} \text{Multiple Condition} \arrow[d] \\ \text{Basic Condition} \arrow[d] \\ \text{Decision} \arrow[d] \\ \text{Statement} \end{tikzcd}$$

So, we have this hierarchy here. Statement coverage is the weakest, and then we have this decision or branch coverage, and then we have this basic condition coverage and then the multiple condition coverage. In basic condition coverage, we just require that each component condition takes true and false values. Whereas, in multiple condition coverage, we require that all possible combinations of conditions between the component conditions have been ensured; so if you are doing multiple condition coverage, then we do not have to do any of these the basic condition coverage, decision coverage or statement coverage. Then you would say that why not do the multiple condition coverage.

Unfortunately, when the number of components condition in an expression is large to require too many test cases, such expression occurs in many applications, for example, embedded control applications. We might have a conditional expression involving 20 or 25 variables, and in that case, we need 2^{20} test cases is just too many, so how do we handle that situation

Can we achieve testing which is you must as effective as multiple condition testing and still not have an exponential number of test cases so that is the MCDC testing multiple condition decision coverage testing, where we would try to achieve as much bug detection capability as a multiple condition coverage.

And at the same time, we will try to hold down the number of test cases require to achieve 100 percent MCDC coverage. No wonder that MCDC is a very popular white-box testing strategy; it is mandated by many certifying agencies that programs have to be MCDC coverage has to be ensured for the software to be acceptable. Because without unduly increasing the number of test cases, we are able to detect almost as much bug as the multiple condition testing. So in the next session, we will look at multiple condition decision coverage that is MCDC testing and see what is involved there, how a test case is designed, and what are the basic concepts there.

Thank you.