

1. In the previous lesson we discussed black box testing, which is the testing performed based on a description of the software, but without considering any of software's internal details. In this lesson, we will discuss the counterpart of black box testing, which is called unsurprisingly white box testing or structural testing. And just like we did for black box testing, we will cover the main characteristics of white box testing and this casts the most commonly used white box testing techniques. We will conclude the lesson with the discussion of the main strengths and limitations of white box testing. So that you will know, when to use it? How to use it? And what to expect from it?

# SOFTWARE TESTING

## WHITE-BOX TESTING

2. In the last lesson, we talked about black-box testing or functional testing, which is the kind of testing that you perform when you just look at the description of the software. Today we're going to cover white-box testing, which is the kind of testing that we perform when we open up the box. We look inside the program, and we actually test it based on its code. And there is one basic assumption behind the idea of white-box testing, which is a very intuitive one, and the assumption is that executing the faulty statement is a necessary condition for revealing a fault. In other words, if there is a bug in the program there is no way we're going to be able to find this bug or this fault, if we don't execute the statement that contains it. Which makes a lot of sense. As we did for black-box testing, we're going to start by summarizing what are the main advantages of white-box testing. The main advantage is that it's based on the code, and as such, the quality of white-box testing can be measured objectively. And what I mean here by objectively is that if you think about black-box testing in many cases, there were subjective decisions, there were had to be made in order to define tests in a black-box fashion. In the case of white-box testing, because everything is based on the code, we don't have to make such subjective decisions. And similarly, because white-box testing is based on the code, it can be measured automatically. So we can build tools and actually there are tools, and there's plenty of tools, that can be measured, the level of white-box testing can be achieved in a fully automated way. And we're going to see some of them in the course of the class. Another advantage of white-box testing is that it can be used to compare test suites. So if you have two alternative sets of tests that you can use to assess the quality of your software, white-box testing techniques can tell you which one of these two test suites is likely to be more effective in testing your code. And finally, white-box testing allows for covering the coded behavior of the software. What that means is that if there is some mistake in the code and is not obvious by looking at the specification of the code, white box testing might be able to catch it, because it tries to exercise the code. There's many different kinds of white box testing, there are control flow based techniques, data flow based techniques, and fault based techniques. And for each one of these family of techniques there are many variations. So this field is very, very broad. [In this lesson we will talk about white-box testing by focusing mainly on control-flow based testing techniques.](#)

# WHITE-BOX TESTING



## Basic assumption

Executing the faulty statement is a necessary condition for revealing a fault

## Advantages

- Based on the code
  - ⇒ can be measured objectively
  - ⇒ can be measured automatically
- Can be used to compare test suites
- Allows for covering the coded behavior

## Different kinds:

- control-flow based
- data-flow based
- fault based



# COVERAGE CRITERIA

Defined in terms of  
test requirements

Result in  
test specifications  
test cases

3. So let's start our lesson on white docs testing by considering again the program PrintSum. If you remember, this is the same program that we used when we were talking about black box testing. It's the program that takes two integers, A and B, and produces, as a result, the sum of the two. And when we were looking at this problem in the context of black box testing, we did not look at implementation. But that's exactly what we're going to do now. So we're going to open the box and look at how the code is implemented. And as you can see, the programmer was kind of creative. Because instead of just adding the two numbers and printing them, he or she also decided to print them in a specific color depending on whether they were positive numbers or negative numbers. So positive results are printed in red and negative results are printed in blue. And as you can see by looking at the code we can see some interesting cases that we might want to test. For instance you can see that there are two decisions made here. So we might decide that this is an interesting case and therefore we want to test it. Similarly we might look at this other case and we might also decide that this is another interesting case and therefore we want to test this one as well. So let's discuss this in a slightly more formal way by introducing the concept of coverage criteria which are really the essence of why box testing. First of all coverage criteria are defined in terms of test requirements where test requirements are the elements, the entities in the code that we need to exercise. That we need to execute in order to satisfy the criteria. And we'll see plenty of examples of that. And normally, when I apply a coverage criterion, my result is a set of test specifications. And we already saw test specifications. Those are basically descriptions, specifications, of how the tests should be in order to satisfy the requirements. And they also result in actual test cases, which are instantiations of the test specifications. And again this is exactly analogous to what we saw when we were talking about the black box testing. So let's see what this means by going back to our example. A minute ago, we looked at the print sum code and we identified two interesting cases for our code. And those are exactly our test requirements. So we have a first test requirement here, which is the execution of this particular statement and a second requirement here and this one corresponds to the execution of this other statement. So for this example there are two things that we need to do in order to satisfy our coverage requirements. Execute this statement and execute this statement.

## LET'S CONSIDER PROGRAM printSum AGAIN

```
1. printSum (int a, int b) {  
2.   int result = a + b;  
3.   if (result > 0)   
4.     printcol("red", result);  
5.   else if (result < 0)   
6.     printcol("blue", result);  
7. }
```

## printSum : TEST REQUIREMENTS

```
1. printSum (int a, int b) {  
2.   int result = a + b;  
3.   if (result > 0)  
4.     printcol("red", result);  
5.   else if (result < 0)  
6.     printcol("blue", result);  
7. }
```

req #1

req #2

4. Now let's see if we are all on the same page on the concept of testing specifications, and we're going to do that through a quiz. What I would like for you to do is to tell me, what are some possible test specifications that will satisfy the requirements that we just saw? And I want you to express this specification in terms of constraints on the inputs. Which means, what constraints should the input satisfy in order for this statement to be executed, and this statement to be executed, and you can write your answer here in these two slots.



## printSum : TEST SPECIFICATIONS

```
1. printSum (int a, int b) {  
2.   int result = a+b;  
3.   if (result > 0)  
4.     printcol("red", result);  
5.   else if (result < 0)  
6.     printcol("blue", result);  
7. }
```

$[a+b > 0]$

$[a+b < 0]$

5. To satisfy our first requirements, we need to find an input that causes the execution of this statement. Because this statement is executed only when result is greater than zero, our test requirement is that a plus b must be greater than 0. When this is satisfied, this statement is executed therefore any test case that implements this specification will cause the execution of this statement. Similarly if we want to cover this statement which is our second requirement we need to have a result that is less than 0. Again, the result is equal to a plus b, so all we need to do is find the test case such that a plus b is less than 0. So this is pretty straight forward.

6. So, now that we have our test specifications. Test specification number one that says that a plus b must be greater than zero. And test specification number two, for which a plus b must be less than zero. I'd like to do another small quiz and ask you to write some test cases that will implement these specifications. And I want you to write the test cases in this format. I want you to specify for each one of the test cases, what is the value of a that you need to use. What is the value of b that you need to use. And since a test case, I like to remind you, is not just a set of inputs. But is the set of inputs plus expected output? I also want you to specify, what is the expected output for these test cases? And in particular, since we have two characteristics of the output, one is the color of the output and the other one is the actual value. I want you to specify for each test case, what is the expected output color and what is the expected value?





## printSum: TEST CASES

```
1. printSum (int a, int b) {  
2.   int result = a+b;  
3.   if (result > 0)  
4.     printcol("red", result);  
5.   else if (result < 0)  
6.     printcol("blue", result);  
7. }
```

TEST SPEC #1

$a+b > 0$

TEST SPEC #2

$a+b < 0$

#1 ((a=[3], b=[9]), (output color=[red], output value=[12]))  
#2 ((a=[-5], b=[-8]), (output color=[blue], output value=[-13]))

7. In this case like in many other cases, there's not just a single right answer because you can build many test cases that will satisfy this test specification. So for example, we could pick value 3 for a and value 9 for b. Those satisfy the specification because a plus b is equal to 12 and therefore is greater than 0, and therefore this is a test case that implements this test specification. And in terms of results, what we expect to see is in the case of a result greater than 0, the caller should be red, and the upper value should be 12. And obviously for this test specification, we just need to pick two inputs such that the sum of the two inputs is less than 0. So for example, we could pick minus 5 and minus 8. The output color in this case is going to be blue, and the output value is going to be minus 13. So, what we just saw is basically how we can go from a piece of code to a set of requirements, which are the interesting aspects of the code that we want to exercise. How we can satisfy the requirements by finding the right test specifications, and then how we can initiate the test specifications into actual test cases. And this is what we will do in general when doing white books testing. And we'll do things likely different way, depending on the specific criteria that we are considering.

8. Now that we saw this overview of white-box testing, I'd like to start talking about specific coverage criterion. And I'm going to start with the first one, which is Statement Coverage. This criterion is going to be characterized by two aspects, the first one is which are the Test requirements for the criteria and the second one is how we measure Coverage for that criteria. In the case of statement coverage, these test requirements are all the statements in the program. So this is the basic, the first the, the simplest coverage criteria in the white-box arena. Let me remind you the assumption that we made at the beginning. White-box testing is based on the assumption that if there isn't a faulty element in the code, we need to exercise it. We need to execute it, in order to find the fault. And that's exactly what statement coverage does. If there is a statement that is faulty in the code, we need to exercise it, in order to find the fault. And therefore, a good measure of how well we exercise the code, is the ratio of the number of executed statements. So all the statements that my test cases executed, to the total number of statements in the program. The higher this number, the better I exercise my code. And we

can also look at coverage criterion in terms of questions. So what is the questions they were trying to answer when we look at a specific set of test cases and we assess the statement coverage that they achieved. And the question is whether each statement in the program has been executed. So, statement coverage is satisfied when all the statements in the program have been executed. And we can satisfy to different degrees and the degrees to which it's satisfied is measured by this value. So now let's go ahead and measure statement coverage on our printSum example. What I'm going to show down here is this progress bar in which we show the amount of coverage, the percentage of coverage achieved. So what this means is that the, if I get to this point I've covered 25% of the statements in the code. And my goal is to get up here to cover all the statements in the code. We have two test cases for this code. The first one that we just saw, consists of the inputs a equal to 3 and b equal to 9, and the second one has the inputs a is equal to minus 5 and b is equal to minus 8. So now let's see what happens when we run this test case. When we run this test case, I'm going to show you by highlighting in the code the parts that we cover when we start executing the code. We cover the first statement, then we always execute the second statement, which computes the result, we continue the execution, we get to the if statement. If the result is greater than zero, in this case our result is 12 because we are working with the inputs 3 and 9, and therefore we execute the true part of the if, we execute the statement. And at this point, we just jump to the end. Because we do not execute the else part of the statement, since we have executed a true one, and therefore, we cover this final statement. So at the end of the execution of this test case, we cover one, two, three, four, five statement out of seven which is roughly speaking 71%. So we can mark in here that we more or less got to 71% of coverage for this code. Now let's look at what happens when we execute test case number two. In this case again, we execute the first statement, the second statement, the third statement. In this case though, the first statement, when it evaluates the value of result, it sees that the result is not greater than zero because our inputs are minus five and minus eight. Therefore, you will execute line number five. And because the result is less than zero, you will also execute line number six. So, at this point, all of the statements in our code are executed and therefore, we achieved a 100% statement coverage, which was our goal. Before looking at other kinds of coverage, let's see how our statement coverage is used in practice. First of all, statement coverage is the most used kind of coverage criterion in industry. Normally for company that uses statement coverage, the typical coverage target is 80-90%, which mean the outcome of the test should be such that 80-90% of the statements are exercised at the end of testing. So at this point, you might be wondering, why don't we just shoot for 100%? Why don't we try to cover all of the code? We just saw that we could do it. And so I'm going to ask you the same question.

## STATEMENT COVERAGE

Test  
requirements

statements in the program

Coverage  
measure

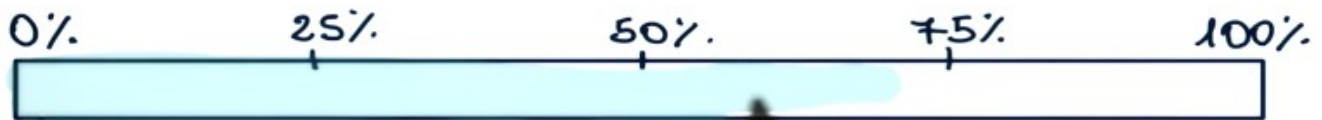
$$\frac{\text{number of executed statements}}{\text{total number of statements}}$$

## printSum: STATEMENT COVERAGE

```
1. printSum (int a, int b) {  
2.   int result = a + b;  
3.   if (result > 0)  
4.     printcol("red", result);  
5.   else if (result < 0)  
6.     printcol("blue", result);  
7. }
```

TC #1  
a = 3  
b = 9

TC #2  
a = -5  
b = -8

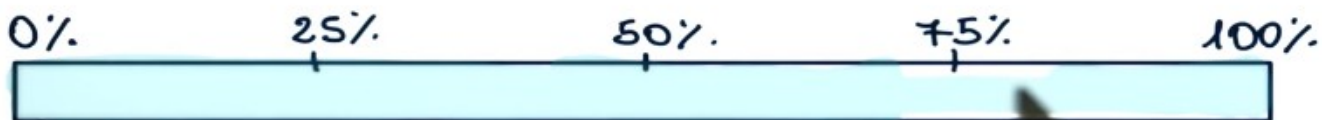


## printSum: STATEMENT COVERAGE

```
1. printSum (int a, int b) {  
2.   int result = a + b;  
3.   if (result > 0)  
4.     printcol("red", result);  
5.   else if (result < 0)  
6.     printcol("blue", result);  
7. }
```

TC #1  
a = 3  
b = 9

TC #2  
a = -5  
b = -8





# STATEMENT COVERAGE IN PRACTICE

Most used in industry

"Typical coverage" target is 80-90%.

9. so, I would like to hear from you. Why do you think that we don't aim normally at 100 % college but, slightly less than that and I want you to put your answer right here.

# STATEMENT COVERAGE IN PRACTICE

Most used in industry

"Typical coverage" target is 80-90%.



Why don't we aim at 100%.

[

]

10. To get the answer to this question, I'm going to ask you to be a little patient, and to wait until the end of the lesson. Because there are a couple of more topics that I want to cover, before I actually get in to this. Nevertheless, I wanted to ask you right away, because I wanted you to think about this, before you see the rest of the lesson.

11. Let's look at the code for PrintSum in a slightly different way by making something explicit. If we go through the code, we can see that the, the code does something in that case, if the result greater then zero, does something else if the result is not greater than zero but is less than zero, and otherwise in the case in which neither of these two conditions is true. Nothing really happens. So [we're going to make that explicit](#), we're going to say here, otherwise do nothing, which is exactly our problem, [\(tao: when result equals zero\) the code does nothing, in this case where it should do something](#). So now, let's look again in our test cases, let's consider the first one, and I'm going to go a little faster in this case, because we already saw what happens If we execute the first test case, we get to this point, we execute this statement, and then we just jump to the end, as we saw. Now we, if we execute the second test case, we do the same, we get to the else statement, the condition for the if is true, and therefore we execute this statement. And [we never reached this point for \(tao: when result equals zero\)](#) either of the test cases. So

how can we express this? In order to do that, I'm going to introduce a very useful concept. The concept of control flow graphs. The control flow graphs is just a representation for the code that is very convenient when we run our reason about the code and its structure. And it's a fairly simple one that represents statement with nodes and the flow of control within the code with edges. So here's an example of control flow graph for this code. There is the entry point of the code right here, then our statement in which we assign the result of A plus B to variable result. Our if statement and as you can see the if statement it's got two branches coming out of it, because based on the outcome of this predicate we will go one way or the other. In fact normally what we do, we will label this edges accordingly. So for example, here we will say that this is the label to be executed when the predicate is true. And this is the label that is executed when the predicate is false. Now, at this point, similar thing, statement five which corresponds with this one, we have another if statement and if that statement is true, then we get to this point and if it's false, we get to this point. So as you can see, this graph represents my code, in a much more intuitive way, because I can see right away where the control flows, while I execute the code. So we're going to use this representation to introduce further coverage criteria.

## printSum: STATEMENT COVERAGE

```
1. printSum(int a, int b) {  
2.   int result = a + b;  
3.   if (result > 0)  
4.     printcol("red", result);  
5.   else if (result < 0)  
6.     printcol("blue", result);  
    [else do nothing]  
7. }
```

TC #1

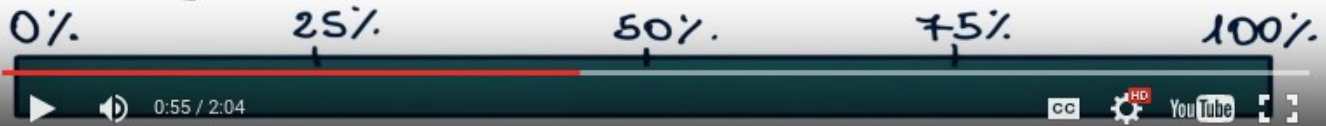
a = 3

b = 9

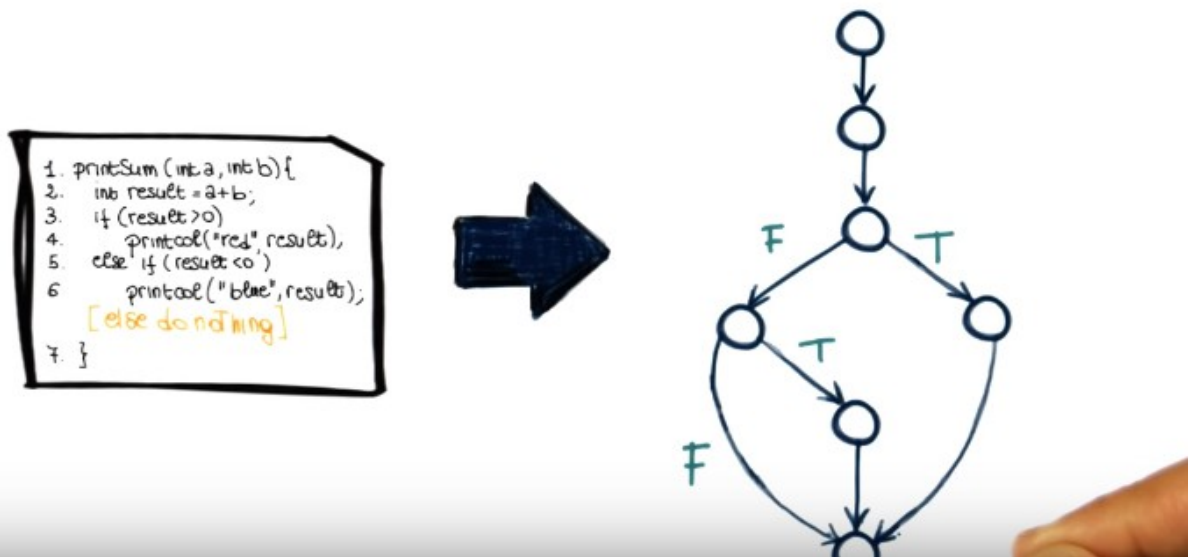
TC #2

a = -5

b = -8



# DIGRESSION: CONTROL FLOW GRAPHS



(上圖中的下面那個 FT 是對於 else if 的那個 if)

12. So now that we know what the CFG is, let's see how we can get into a count, that else part, that is missing in our example code by introducing a new kind of coverage. Branch coverage. As usual, I'm going to describe branch coverage in terms of test requirements and coverage measure. So starting from test requirements. The test requirement for branch coverage are the branches in the program. In other words, the goal of branch coverage is to execute all of the branches in the program. The coverage measure is defined accordingly as the number of branches executed by my test cases over the total number of branches in the program. And let me remind you that branches are the outgoing edges from a decision point. Therefore, an if statement, a switch statement, a while statement. Any node in the CFG that has got more than one outgoing edge. Those edges are called branches. So let's look at that using our example. So now we're looking back at our printSum example. And in addition to the code, I also want to represent the CFG for the program. So let's start by looking at how many branches we have in our code. Which means how many test requirements we have. And in this case there are two decision points. The first one that corresponds to the first if, and the second one that corresponds to the second if. So we have one, two, three, and four branches. So now, let's bring back our current set of test cases. We had two test cases. The one's that, with which we achieved a 100% statement coverage. And let's see what happens in terms of branch coverage when we run these test cases. I start from the first one, when we execute it, we follow the code, we get to this decision point because the predicate in the if statement is true. We follow the true branch, therefore we get here and then, we exit from the program. So, in this case, we covered one of the branches, which means that we got to 25% coverage. Now when we run the second test case, again we follow this path. We get to this, the first if and in this case the predicate of the if is false. Therefore, we go this way. We reach the second predicate, the second if. The result is true, so we follow the true branch and therefore, we cover these additional two branches. So at this point, we are at 75% branch coverage. So what happens is that we're missing this branch. For now, the inputs that we consider, this branch is executed. Therefore, we need to add an additional test case. And that this case that we need, is one for which this predicate is false and this predicate is false. The simplest possibility in this case is the test case for which A is equal to 0 and B is equal to 0. If we execute this test case, our execution again followed this path, follows the fourth branch here. And in

this case, because result is not less than zero either, will follow this branch as well. And therefore, we will reach our 100% branch coverage. And this covered the problem. Something that I would like to clarify before we move to the next topic, is that **100% coverage does not provide any guarantee of finding the problems in the code**. All we saw so far is the fact that **by testing more thoroughly we have more chances of finding a problem in the code**. But it doesn't matter which kind of coverage we utilize, and how much coverage we achieve. There's always a chance that we might miss something. And I will get back to this later on in the lesson. I just mentioned the fact that **we tested more thoroughly when we went from statement coverage to branch coverage**. What does that mean exactly? To explain that, I'm going to introduce the concept of test criteria subsumption. **One test criteria subsumes(包含) another criteria when all the tests suites that satisfy that criteria will also satisfy the other one**. So let me show you that with statement and branch coverage. If we identify a test width that achieves 100% branch coverage, the same test width will also achieve, necessarily, 100% statement coverage. That's what happened for our example, and also what happens in general, because **branch coverage is a stronger criteria than statement coverage**. **There is no way to cover all the branches without covering all the statements**. It is not true that any test results satisfies statement coverage will also satisfy branch coverage. And, in fact, we just saw a counter example. When we look at the printSum code. We had a test where there was achieving 100% statement coverage and was not achieving 100% branch coverage. Therefore, in this case we have a substantial relation in this direction. **Branch coverage, subsumes statement coverage**. What it also means is that normally, or **in general, it is more expensive to achieve branch coverage than achieve statement coverage**, because achieving branch coverage requires the generation of a larger number of test cases. So what this relation means is that branch coverage is stronger than statement coverage but also more expensive.

我的總結: A coverage subsumes B coverage = A coverage is a stronger criteria than B coverage = we test more thoroughly using A coverage than B coverage = A 測試得要比 B 多些

## BRANCH COVERAGE

Test  
requirements

branches in the program

Coverage  
measure

$$\frac{\text{number of executed branches}}{\text{total number of branches}}$$

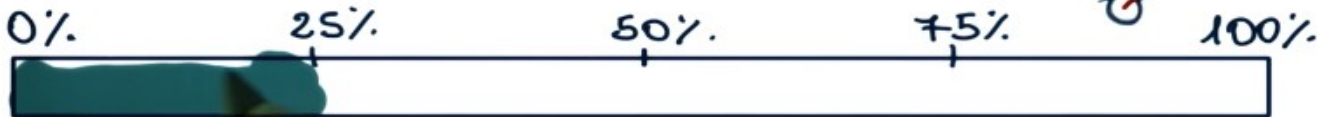
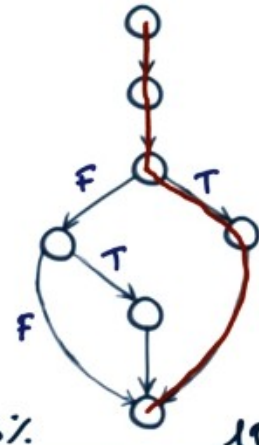


## printSum: BRANCH COVERAGE

```
1. printSum (int a, int b) {  
2.   int result = a+b;  
3.   if (result > 0)  
4.     printcol("red", result);  
5.   else if (result < 0)  
6.     printcol("blue", result);  
   [else do nothing]  
7. }
```

a=3  
b=9

a=-5  
b=-8



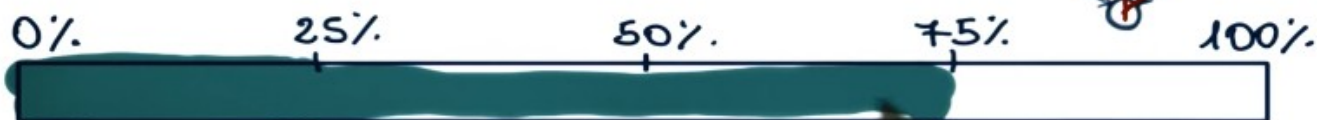
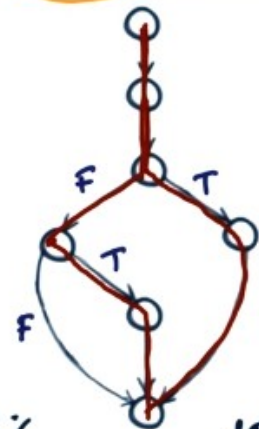
(上圖中總共有四個 branch, 即那四個 FTFT)

## printSum: BRANCH COVERAGE

```
1. printSum (int a, int b) {  
2.   int result = a+b;  
3.   if (result > 0)  
4.     printcol("red", result);  
5.   else if (result < 0)  
6.     printcol("blue", result);  
   [else do nothing]  
7. }
```

a=3  
b=9

a=-5  
b=-8



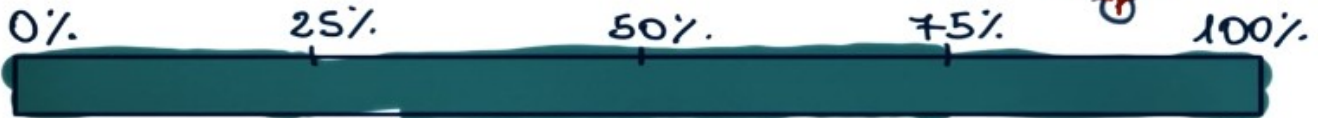
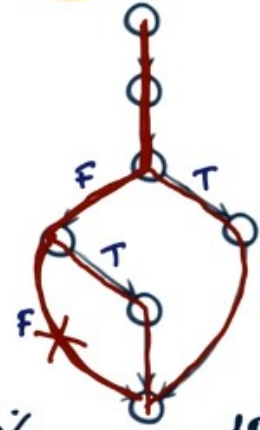
## printSum: BRANCH COVERAGE

```
1. printSum (int a, int b) {  
2.   int result = a+b;  
3.   if (result > 0)  
4.     printcol("red", result);  
5.   else if (result < 0)  
6.     printcol("blue", result);  
7.   [else do nothing]  
8. }
```

a=3  
b=9

a=0  
b=0

a=-5  
b=-8



## TEST CRITERIA SUBSUMPTION

BRANCH  
COVERAGE



STATEMENT  
COVERAGE

13. What I'm going to do next is to introduce a few additional coverage criteria using a slightly more complex example, but still a pretty simple one. What I'm showing here is a program that reads two real numbers, x and y. And then, if x is equal to 0 or y is greater than 0, it computes y as y divided by x.

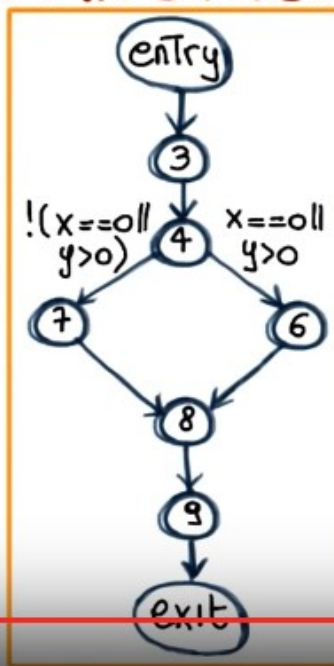
Otherwise, it computes  $x$  as  $y$  plus 2, then it writes the value of  $x$ , the value of  $y$ , and it terminates. Let's also introduce a CFG for the program. As you can see, the CFG represents the statements in the code and their control flow. And in this case, I made explicit over the branches what are the conditions under which those branches are taken to make it simpler to look at the example. Let's assume that we have two tests for this code that are shown here. For the first one, the inputs are 5 and 5. For the second one, 5 and minus 5. If we consider branch coverage for this code and we consider the two test cases, for the first one this condition is true. Because  $x$  is not equal to 0 but  $y$  is greater than 0. And therefore, we will follow this tree branch. For the second one, the condition is false. Because  $x$  is not equal to 0 and  $y$  is not greater than 0. Therefore, the negation of it is true and we will follow this branch. In other words, these two test cases achieve 100% branch coverage on this code. If we look at the code though, we can see that there is the possibility of making this code fail. Consider this statement, *if  $x$  is equal to 0, we could have a division by 0. However, these two test cases, despite the fact that they achieved 100% branch coverage, will not rebuild this problem. So how can we be more thorough?* I'll let you think about it for a second, so think about how can we test more thoroughly, in a more complete way, this code. So, in a way that goes beyond branch coverage. *And the answer is that we can make each condition true and false.* Instead of just considering the whole predicate here. And that's exactly what is required by the next criteria that we're going to consider which is condition coverage. We're going to define it as usual in terms of test requirements and coverage measure. In this case, the test requirements for condition coverage are the individual conditions in the program. So *we want each condition in the program to be both true and false first time execution.* So the way in which we can measure that is by measuring the number of conditions that were both true and false when we executed our tests over the total number of conditions. And that gives us the percentage of coverage that we achieved for condition coverage. Again, if you want to look at this criteria in the form of a question. The question would be, has each boolean sub-expression, which means every condition in every predicate, evaluated both to true and false when we run our tests.

## LET'S CONSIDER ANOTHER EXAMPLE

```

1. void main(){
2.   float x,y;
3.   read(x);
4.   read(y);
5.   if ((x==0) || (y>0))
6.     y = y/x;
7.   else x = y+2;
8.   write(x);
9.   write(y);
10. }

```



Tests:  $(x=5, y=5)$   
 $(x=5, y=-5)$

Branch coverage: 100%

How can we be more thorough?

We can make each condition T and F

# CONDITION COVERAGE

Test  
requirements

individual conditions in the program

Coverage  
measure

$$\frac{\text{number of conditions that are both T and F}}{\text{total number of conditions}}$$

14. So now that we introduced an additional criterion, let's have a quiz to see whether everybody understood the concept of subsumption. We know that branch coverage subsumes statement coverage, but what about condition coverage? Does it subsume branch coverage? Is it the case that all of the test which satisfy condition coverage will also satisfy branch coverage? Think about it, and answer either yes or no.

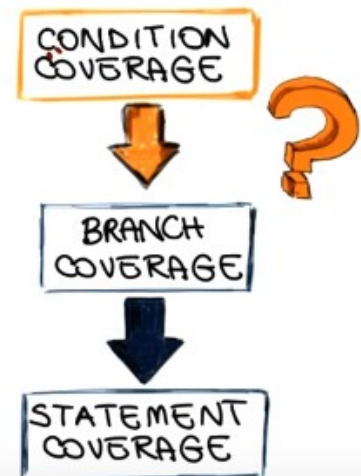


## SUBSUMPTION

Does condition coverage  
imply branch coverage?

☐ Yes

☒ No



15. In this case, the answer is no, and I'm going to show you an example of this in a minute.

16. Let's go back to our test criteria subsumption representation, where we already had branch coverage and statement coverage. In this case, if we want to add condition coverage to this page, we have to put



it here on this side, with no relationship of subsumption with either branch coverage or statement coverage, which means that the criteria are not comparable. So now let's consider again our last example, and let's use a different test with this time. We still have two tests, but the first one has  $x$  is equal to 0 and  $y$  is equal to minus 5. And the second one,  $x$  is equal to 5 and  $y$  is equal to 5 as well. Let's see what happens in terms of condition coverage, when we run these tests. If we look at the first condition,  $x$  is equal to 0. It is both true, for the first test case, and false for the second one. As for the second condition,  $y$  greater than 0, it is false for the first test case and true for the second one. Therefore we've achieved a 100% condition coverage with these two tests. But what about branch coverage? If we consider the whole predicate, instead of just the individual conditions, let's see what happens for the two test cases. If we look at the first one, because  $x$  is equal to 0, the overall predicate is true. As for the second one, because the second condition is true, the overall predicate is true. In other words, despite the fact that we're exercising all the possible values for the two conditions, the overall predicate is always true. And therefore, we're covering only one of the two branches. So our coverage will be 50%. This is the reason why normally the two criteria that we just saw, decision coverage, and condition coverage are considered together. And the resulting criterion is called branch and condition coverage, or also decision and condition coverage. At this point the test requirements and the coverage measure should be pretty straight forward, because they're just considering the two criteria together. As far as the requirements are concerned, they include all the branches and individual conditions in the program. Where as the coverage measure is computed considering both coverage measures, both branch coverage and condition coverage.

# TEST CRITERIA SUBSUMPTION

BRANCH  
COVERAGE

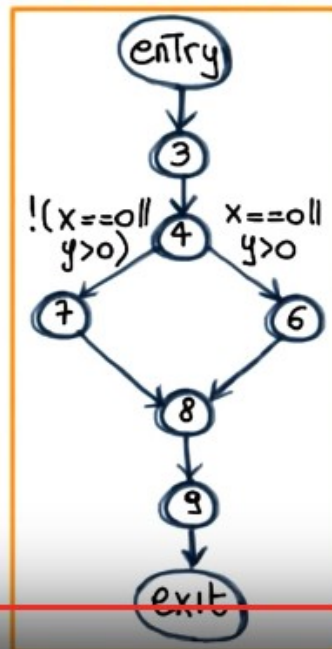


STATEMENT  
COVERAGE

CONDITION  
COVERAGE

## LET'S CONSIDER OUR LAST EXAMPLE

```
1. void main(){
2.   float x,y;
3.   read(x);
4.   read(y);
5.   if ((x==0) || (y>0))
6.     y = y/x;
7.   else x = y+2;
8.   write(x);
9.   write(y);
10. }
```



Tests:  $(x=0, y=-5)$   
 $(x=5, y=5)$

Condition coverage: 100%

What about branch  
coverage? 50%

# BRANCH AND CONDITION COVERAGE (DECISION)

Test  
requirements

branches and individual conditions  
in the program

Coverage  
measure

computed considering both coverage  
measures

17. Let's have another quick quiz about Subsumption. If we consider Branch and Condition Coverage, does it subsume Branch Coverage, and therefore Statement Coverage? Or in other words, does branch and condition coverage imply branch coverage? Answer yes, or no.



# SUBSUMPTION

Does branch and condition coverage imply branch coverage?

☒ Yes  
☐ No

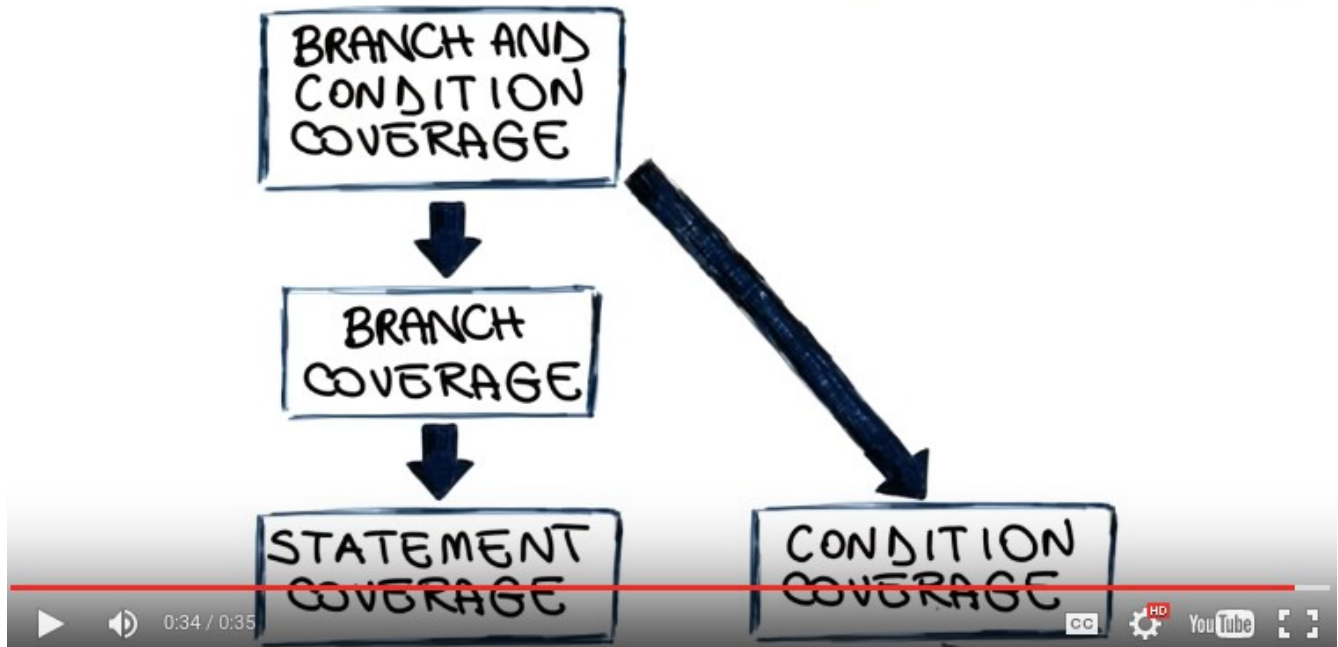


18. In this case it should be clear that [the answer is yes, because branch and condition coverage actually includes branch coverage](#). Therefore, by definition, any test rate that satisfies branch and condition coverage will necessarily satisfy branch coverage.

19. So we can now update our test criteria subsumption. We start from this situation in which there are no relationship between condition coverage, branch coverage and statement coverage. And when we add branch and condition coverage, we can mark the fact that branch and condition coverage subsumes branch coverage and also subsumes condition coverage, for the same reason. Therefore, this is a stronger criterion than both branch coverage and condition coverage, and of course indirectly also soft statement coverage. So once more, to make sure we are all on the same page, if I develop a test rate that satisfies branch and condition coverage, the same test will satisfy also branch coverage, statement coverage and condition coverage, necessarily.



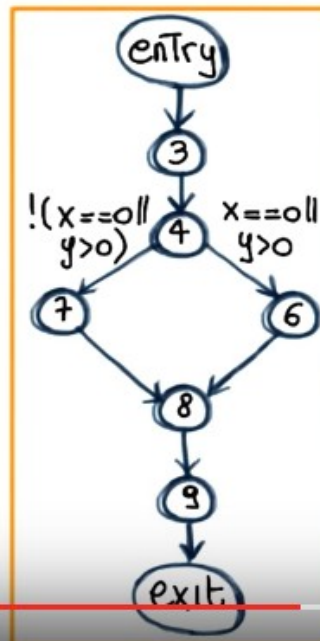
# TEST CRITERIA SUBSUMPTION



20. So let's have another quiz using the example that we just used. This one is about achieving 100% branch and condition coverage. So let me bring back the two test cases that we just saw, and as we saw a few minutes ago, these test cases do not achieve 100% branch and condition coverage despite the fact that they achieve 100% condition coverage. So both conditions are both true and false, for these test cases. So what I want you to do is to [add](#) a test case, to achieve 100% branch and condition coverages. So specify the test case by writing here the value for x, and the value for y.

## quiz ACHIEVING 100% B&C COVERAGE

```
1. void main(){
2.   float x,y;
3.   read x;
4.   read y;
5.   if ((x==0) || (y>0))
6.     y = y/x;
7.   else x = y+2;
8.   write(x);
9.   write(y);
10. }
```



Tests  
(x =  $\emptyset$ , y = -5)  
(x = 5, y = 5)  
Add a Test Case  
To achieve 100%  
B&C Coverage  
(x = 3, y = -2)

21. Obviously there are many possible tests that we can use to reach 100% branch and condition coverage. So I'm just going to show a possible one, which is x equal to 3 and y is equal to negative 2. If we specify this as case, you can see that the overall condition is false, because neither x is equal to 0 nor y is greater than 0. Therefore we will follow the false, false branch, and achieve 100% branch and condition coverage for this code. And we might require to be even more thorough, that all the combinations of our conditions inside each decision, inside each predicate, are tested. Which is what is called, multiple condition coverage. But because of the way this criterion is defined, it is combinatorial, because you have to consider all the possible combinations of conditions. And therefore it's extremely expensive, to the point of being impractical. So instead of defining that criterion, [we're going to find another one which finds a good trade off between thoroughness of the tests and their cost.](#)

22. [And this criterion is called Modified Condition/Decision Coverage, also called MC/DC. This criterion is very important because it is often required for safety critical applications. For example, the FAA, the Federal Aviation Administration, requires for all the software that runs on commercial airplanes to be tested according to the Modified Condition/Decision Coverage.](#) So what is the key idea behind the MC/DC criterion? It is to test only the important combinations of conditions instead of all of them, and limit the testing cost by excluding the other combinations. And the way in which it works is by extending branch and decision coverage with the requirement that each condition should affect the decision outcome independently. So let's see what this means with an example that will also show you how you can reduce the number of combinations in this way. I am going to show you an example of how MC/DC works using this predicate which consists of three conditions. a, b, and c, which are all in and, so the overall predicate is a and b and c. The first thing I'm going to do is to show you how many test cases we will need to satisfy the multiple condition coverage for this simple predicate. Which means, how many test cases we will need to test all the possible combinations of true and false values for these conditions. So [I'm going to populate this table. And as you can see, at the end we have eight test cases.](#) Each test case tests a different combination of values for a, b, and c. I'm also showing, for each test case, the outcome of the overall predicate. So, for example, if we look at the first one, the first test case, will be such that a is true, b is true, and c is true. And therefore, the overall outcome of the predicate is true. Now let's consider the first condition, a. As I said a minute ago, what we want to test are the important combination. Which are the combinations in which a single condition independently affects the outcome of the overall predicate. So if we consider a and we look at this possible set of these cases. [Let's try to find two test cases such that the only difference between the two test cases is the value of a, and the overall outcome of the predicate is different. If we look at the table, we can see that this is true for test cases one and five.](#) If we look at these two cases, we can see that the overall of the predicate in the two cases is true and false, and that the only difference between the value of the conditions is the value of a. So these test cases satisfy exactly what we wanted. There are two test cases in which the value of a independently decides the overall value of the predicate. What we do, therefore, is to add these first two test cases to our set of tests down here. [Now let's focus on b and let's try to find two test cases such that the value of b is the only value that changes between the two test cases, but the overall value of the predicate is different, the same thing we did for a. And in this case, we can see that if we select test case number one, and test case number three, we have exactly that situation.](#) b is true in the first case, false in the second one, a and c don't change, but the overall value of the predicate changes. And now you can notice something else. That even though we selected two test cases, tested two values, one we already had. So, we only need three test cases overall to test a and b according to MC/DC. [Now, let's look at our last condition, c. At this point, we know the game, so we just have to look for two test cases that satisfy our requirements. And in this case, one and two are suitable candidates.](#) And once more, because we already have one, we just have to add two to our list. So as you can see from this example, [we went from having eight test cases needed to cover all the possible](#)

combinations of conditions to only four test cases to satisfy the MC/DC criteria. So let's see where MC/DC stands in our substantiation hierarchy. This is what we had so far in the hierarchy and this is where the MC/DC criterion will stand. MC/DC criterion is stronger than branch and condition coverage. Why? Because it requires every single condition to be true and false. And therefore, this section was a condition coverage criteria. And it also requires every predicate to be true and false and therefore, this section is branch coverage. And in addition, it's got the additional requirements that the true and false values, all the conditions have to also decide the overall value of the predicate. So it's stronger. Which is more thorough than branch and condition coverage and, as usual, also stronger than branch coverage, statement coverage, and condition coverage.

## MODIFIED CONDITION/DECISION COVERAGE (MC/DC)

Key idea: test important combinations  
of conditions and limited testing costs

⇒ extend branch and decision coverage with the  
requirement that each condition should affect  
the decision outcome independently

## MC/DC : EXAMPLE

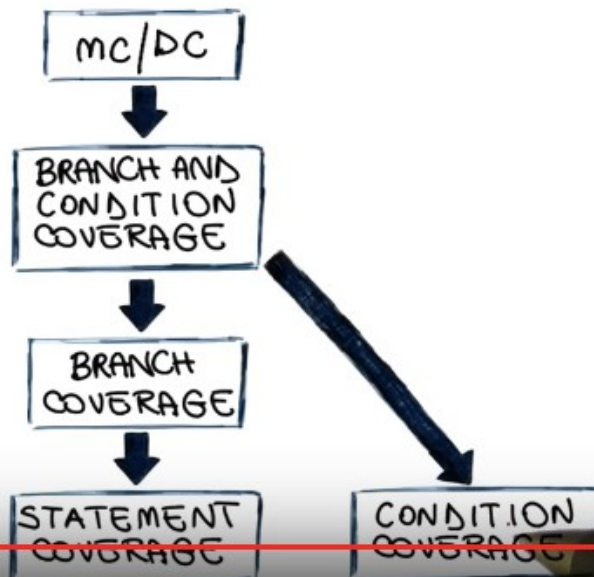
a & b & c

Test case	a	b	c	outcome
1	True	True	True	True
2	True	True	False	False
3	True	False	True	False
4	True	False	False	False
5	False	True	True	False
6	False	True	False	False
7	False	False	True	False
8	False	False	False	False
1	True	True	True	True
5	False	True	True	False
3	True	False	True	False
2	True	True	False	False

8

4

## TEST CRITERIA SUBSUMPTION



23. As I mentioned at the beginning of the class, there are many, many, many, white box criteria. And we're not going to have time to cover them all. So what I like to do now is just to give you the flavor, of

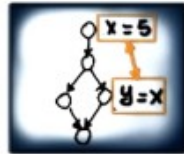


some other criteria by just mentioning them, and saying a couple of things, about the way they work. And the first one I want to mention is path coverage. And in path coverage, the test requirements are all the paths in the program. So what that means is that to satisfy this criteria, we need to generate enough test cases such that all the paths in my program are covered. As you can imagine this is incredibly expensive because any nontrivial program has got a virtual infinite number of paths and therefore we will need a virtual infinite number of test cases to satisfy the path coverage criteria. Another family of criteria I want to mention are data-flow coverage criteria and in data-flow coverage criteria, the focus shifts from the coverage of individual elements in the code to the coverage of pairs of elements. And in particular the coverage of Statements, in which the content of some memory locations modified, and statements in which the content of the same memory location is used. So in this way, our test will exercise the assignments of values to memory, and the usage of those assignments,. Finally, I want to mention mutation coverage. And this is a fairly different and new idea, so the key concept in mutation coverage is that we want to evaluate the goodness of our test by modifying the code. For example here in this small I'm, I'm showing that I might change it. An if statement from K greater than 9, to K greater than or equal to 9. And the reason why I want to do that, is that if I generate enough mutants, enough variation of my program, then I can use them to assess how good are my tests at distinguishing the original program and the mutants. And because I'm changing the code based on the way I expect to introduce errors in the code, the more my test can identify mutants, the better they are at identifying real faults. This is a very high level [UNKNOWN], but just to give you the intuition and the idea behind these criteria, and I'm going to provide as usual, additional pointers to go in more depth on these topics in the notes of the class. So now, let's go back to our test criteria subsumption hierarchy, and see how all of these criteria fit together in this context. Let me start with multiple condition coverage. As we saw, MC/DC is sort of as [UNKNOWN] way of doing multiple condition coverage in the sense that it doesn't consider all the combinations, but only the ones that are more likely to matter. And as such, MC/DC exercises a subset of the elements of the multiple condition coverage exercises. And therefore, multiple condition coverage is more thorough than MC/DC and subsumption, and it's also as we saw incredibly more expensive. Path coverage subsumes branch coverage, because if I cover all of the paths in the code, I necessarily cover all the branches. However, it doesn't subsume multiple condition coverage, MC/CD, or branch and condition coverage. Because this criteria, have additional requirements involving the conditions of the predicate that path coverage does not have. As for data-flow coverage criteria and mutation coverage criteria, there really no relation with the other criteria, because they look at different aspects, of the code. So they're not really comparable, and therefore we're just going to put them on the side, without any relationship with the other ones. The reason why I want to represent them all anyways is because I want to make an important distinction between these different criteria. And that's the distinction between practical criteria, which are criteria that are actually not too expensive to be used in real scenarios. And theoretical criteria, which are criteria that are useful in theory from the conceptual standpoint, but they're not really applicable in practice because they're too expensive, because they require basically too many test cases to be satisfied.

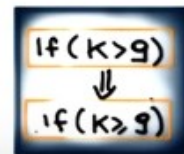
# OTHER CRITERIA



Path coverage



Data-flow coverage



Mutation coverage

# TEST CRITERIA SUBSUMPTION

Theoretical  
Criteria

PATH COVERAGE

MULTIPLE  
CONDITION  
COVERAGE

MUTATION  
COVERAGE

Practical  
Criteria

mc/dc

DATA-FLOW  
COVERAGE

BRANCH AND  
CONDITION  
COVERAGE

BRANCH  
COVERAGE

STATEMENT  
COVERAGE

CONDITION  
COVERAGE

24. White box testing, in general, and coverage criteria in particular, involve some subtle concepts, so before I conclude this lesson, I want to have a few more quizzes to make sure that we all understand these concepts. The first one involves a very simple piece of code, a straight line of code, three statements, in which we simply read an integer and then prints 10 divided by the value of that integer minus 3. Now, let's imagine that we have a test where there consists of three test cases for this code, and what I'm showing in the test cases is the input and the expected output. So for the first one, the input is 1, and I expect the output to be minus 5. For the second one, the input is minus 1, I'm expecting to have 2.5. And for the third one, the input is 0, and I'm expecting to have minus 3.3333 as

the result. Now the first question I want to ask, is if we considered this test suite, and we run it on the code, does it achieve path coverage? And remember that path coverage is one of the strongest coverage criteria that we saw.



```
1. int i;  
2. read(i);  
3. print(10/(i-3));
```

Test suite :  $(1, -5), (-1, 2.5), (0, -3.\bar{3})$

Does it achieve path coverage?

☒ Yes

☐ No

25. And the answer in this case is clearly yes. In fact, any input will really achieve path coverage for this code because there is only one path in the code.

26. But now want to ask a different question which is, does this test reveal the fault at line three? Clearly, here there is a possible problem, because if we pass the right input, we can have a division by zero. Is that revealed if we run this test within the code? Yes or no?



```
1. int i;  
2. read(i);  
3. print(10/(i-3));
```

ONLY EXHAUSTIVE  
TESTING IS ...  
... EXHAUSTIVE!

Test suite : (1, -5), (-1, 2.5), (0, -3.3)

Does it achieve path  
coverage?

☒ Yes  
☐ No

Does it reveal the  
fault at line 3?

☐ Yes  
☒ No

27. And the answer is no. So even path coverage's stronger criteria that we saw is not enough to reveal this problem. And why is that? So let me go back to a concept that I mentioned in a previous lesson, the cost of exhaustive testing. Exhaustive testing is really the only way in which we can exercise all of the possible behaviors of a program. So we can say even though it might sound like topology, that only exhaustive testing is exhausted. All the coverage criteria that we saw are just process and just approximation of a complete test. So test is a best effort kind of activity. Coverage criteria help you assess how well you tested but once more, test can only reveal issues, can only reveal problems. You can never show the absence of problems. And that's something is important to remember when we do testing.

28. Now let's do another quiz considering a slightly different piece of code. In this case we have five layers of code. We have a variable i, a variable j. Variable j is read from the input. And then based on this predicate, we either print the value of i or simply exit. And what I want to know from you is whether you can create a test suite to achieve statement coverage for this simple piece of code. Yes or no?



```
1. int i=0;  
2. int j;  
3. read(j);  
4. if ((j>5) && (i>0))  
5.   print(i); DEAD
```

EVERY PROGRAM  
CONTAINS DEAD  
OR UNREACHABLE  
CODE!

Can you create a test suite to achieve  
statement coverage?

☐ Yes  
☒ No

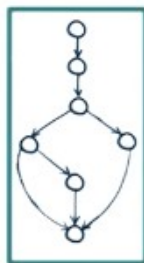
29. And the answer in this case is no. And the reason for this is because this code is unreachable. This is dead code because no matter the value of *j*, this condition will always be false because *i* will always be 0. And notice that this is a small example, but another important truth is that any non-trivial program contains dead or unreachable code, code that no matter how well we test our system, we will never be able to exercise. Why is that? Various reasons. For example, defensive programming or, for example, developing for future releases. So there might be pieces of code that are added, but they're still not activated. They're still not invoked by any other part of the code because of modifications to the code. There are some parts that were executed before and in the new version of the program, they are no longer exercised. They are no longer executable. But they still remain in the code base. And this is a very, very normal situation for this reason and many more. And this is an important concept because this affects the very meaning of coverage measures. If there is some unreachable code, we will never be able to reach 100% code coverage. And in fact, if you remember, at the beginning of the lesson, we discussed this concept and asked you why you think that, in the industry, the target for coverage is not 100% but less than that. And that's the answer, to account for the fact that there are infeasibility problems that I have to take into account when I test the code, infeasible paths, unexecutable statements, conditions that can never be true, and so on. So most criteria, if not all, are affected, and we need to take that into account when we try to achieve a given coverage target.

30. Now let me conclude the lesson by summarizing a few important aspects of white-box testing. The first important aspect is that white-box testing works on a formal model. The code itself are models derived from the code. So when we do white-box testing, we don't need to make subjective decision, for example, on the level of obstruction of our models. Normally, we simply represent what's there. And so what we will obtain are objective, results and objective measures. As I also said at the beginning, coverage criteria allows us to compare different test suites, different sets of tests, because I can measure the coverage achieved by one test suite and by the other, and then decide which one to use based on this measure. And again, remember, these measures aren't perfect, but they at least give you an objective number, an objective measure of the likely effectiveness of your tests. So even though achieving 100% coverage does not mean that you identify all the problems in the code for sure. If your level of coverage is 10%, for example, for statement coverage. That means that you haven't exercised

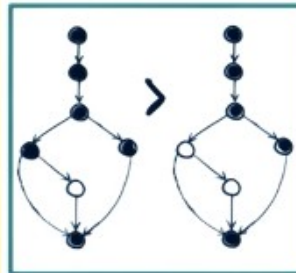


90% of your code, and therefore the trouble is in a piece of software that is inadequately tested and likely to be of inadequate quality. We also saw that there are two broad classes of coverage criteria, practical criteria that we can actually use, and theoretical criteria that are interesting from a conceptual standpoint, but that they are totally impractical. They are too expensive to be used on real world software. And finally, as we also said at the beginning, one of the great things about white box testing and coverage criteria, is that they are fully automatable. There are tools that can take your code, instrument it automatically. And when you run your test cases, they will tell you, what is the level of coverage that you achieve with your test at no cost for you. So there's really no reason not to measure coverage of your code.

## WHITE-BOX TESTING SUMMARY



works on a  
formal model



Comparable



Two broad  
classes:  
practical  
Theoretical



Fully  
automatable