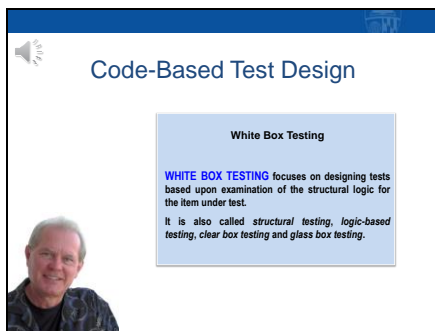


1



In this lecture, I'm going to discuss code-based test design.

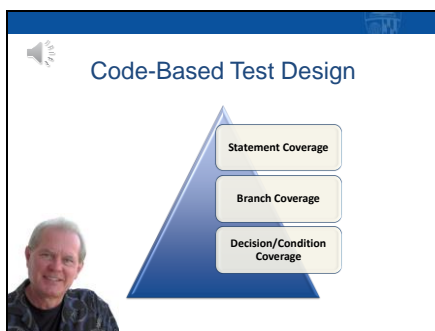
2



Code-based testing, also called white box testing, involves designing tests by using the code itself as one of the inputs to the process. In practice, the requirements for a component, either formal or informal, should also be used so that the actual test results can be compared to the expected test results.

Recall from our testing levels and responsibilities discussion that code-based testing should be done by the software developers...and not by an independent test team.

3

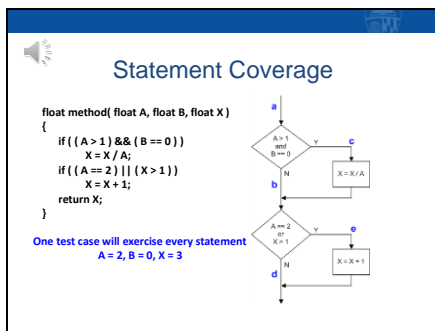


When we deal with code-based testing, one of the things we want to include as part of our testing goals is something called a coverage criteria. Including a coverage criteria ensures that the code under test is adequately exercised.

If the code under test is not adequately exercised it increases the risk that any errors in the code will not be uncovered. For example, suppose we ran ten tests on a code component and they all ran successfully. And...suppose those ten tests exercised only 50 percent of the code. That leaves half the code unexercised. There could be errors that will manifest themselves when that code is ultimately exercised...perhaps in production...that might have been caught if our tests exercised the code more thoroughly.

We'll be discussing three different levels of coverage criteria: statement coverage, branch coverage, and decision/condition coverage. Each of these is at least as thorough as the others...with statement coverage being the least thorough and decision/condition coverage being the most thorough.

4



Statement coverage involves writing enough tests so that each programming statement in the component under test is exercised at least once.

Let's look at this example...a simple C++ function that gets passed three floating point variables when it is called. We'll assume that this is the component under test. Now, I just made up the code for this function, and it's not important that it really do anything meaningful for purposes of this discussion...because I just want to focus on coverage.

To achieve statement coverage we need to write enough tests so that each statement is executed at least once...and this can be done with a single test case.

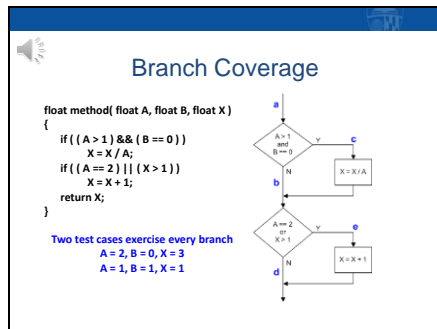
Now, let's think about the adequacy of statement coverage. Here's a flow chart for the function . Note that I didn't bother to include an element for the return statement, since it will always be executed.

The one test case that exercises all statements would exercise the execution path a-c-e. But, there is a second execution path a-b-d, and this would not be covered by our test. And...there are two complex conditional tests here. Suppose the comparison in the second test was supposed to be X equals 1 and not X greater than one. That error would not be detected.

So...as a coverage criteria in practice...even though it is

commonly used...statement coverage is not adequate since the risk of not detecting errors can be significant.

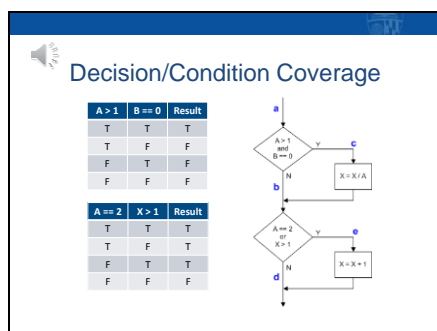
5



The next higher coverage criteria is branch or decision coverage. For branch coverage we need to write enough tests so that each true/false branch is exercised at least once. In practice, branch coverage is a very common coverage criteria and is at least as thorough as statement coverage.

In this example, two test cases would do the job...and would also exercise the two control flow paths...so that's certainly better coverage than statement coverage. But...what about those compound conditionals. They could be evaluated as true or false in quite a few different ways...so some errors could slip through undetected.

6



The next type of coverage criteria is decision/condition coverage. This requires that we write enough tests so that each true/false decision branch is executed at least once and all combinations that can cause a decision to be true and false be tested.

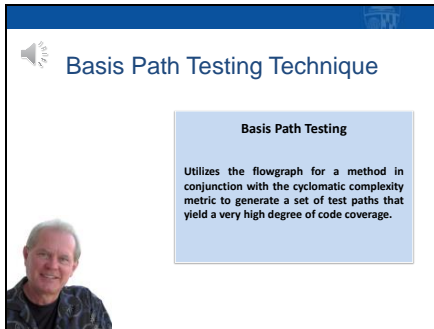
For our sample function, there are four combinations that would need to be tested for the first decision and four for the second decision, as illustrated here. As you can see, that's a lot of work...but it's a lot more thorough than statement or branch coverage.

The bottom line here is that an organization needs to

set coverage criteria that makes sense and mitigates risk to an acceptable level. As I mentioned in an earlier lecture...testing is an exercise in risk management.

And...as we saw in these examples...some combination of statements, branches, conditions, and execution paths are all involved in ensuring reasonable test coverage...so it can be quite challenging to design an adequate set of test cases.

7



Basis Path Testing Technique

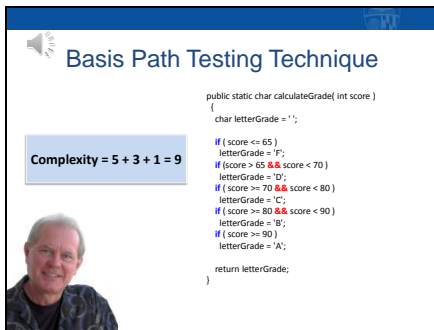
Basis Path Testing

Utilizes the flowgraph for a method in conjunction with the cyclomatic complexity metric to generate a set of test paths that yield a very high degree of code coverage.

It's one thing to specify coverage criteria and another thing for engineers to be able to design tests that satisfy that criteria other than by hit or miss. Fortunately...there are techniques that can help us do this pretty easily. One technique, called basis path testing, is pretty straightforward to apply and yields a very high degree of statement, branch, condition, and path coverage.

The technique utilizes the cyclomatic complexity that you learned about in the module on software quality metrics. Let's see how it works.

8



Basis Path Testing Technique

Complexity = 5 + 3 + 1 = 9

```
public static char calculateGrade( int score )
{
    char letterGrade = '';

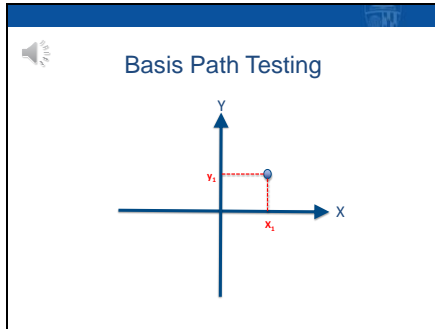
    if ( score <= 65 )
        letterGrade = 'F';
    if ( score > 65 && score < 70 )
        letterGrade = 'D';
    if ( score >= 70 && score < 80 )
        letterGrade = 'C';
    if ( score >= 80 && score < 90 )
        letterGrade = 'B';
    if ( score >= 90 )
        letterGrade = 'A';

    return letterGrade;
}
```

Recall that the cyclomatic complexity metric was introduced as a measure of the complexity of a software component's control flow logic.

There are a number of formulas that can be used to calculate cyclomatic complexity. One way is to count the number of decision keywords, the number of ands and ors, and then add one. In this example of a Java method, there are five decision keywords, highlighted in blue, and three "and" keywords, highlighted in red...so the cyclomatic complexity of the method is 9.

So...how does this relate to testing? Let's find out.



Cyclomatic complexity and basis path testing have their roots in mathematics. While it's not necessary to know or understand the mathematics, I think they're quite interesting and are worth a short discussion.

If you've ever taken physics or linear algebra you've probably been exposed to the concept of a vector space. Here's a simple example in case you haven't. Suppose I wanted to make a two-dimensional plot...maybe plot some points as illustrated here. Now the X and Y axes form a basis for a two-dimensional vector space. There are a countably infinite number of points that can be drawn in two-dimensional space, but only two pieces of information are required to describe any of those points...a distance along the x-axis and a distance along the y-axis. Note that the x and y axes are perpendicular, or orthogonal, to each other. In mathematical terms we would say that they are independent axes...we can't express x in terms of y and vice versa...hence, they form the basis of a two-dimensional vector space.

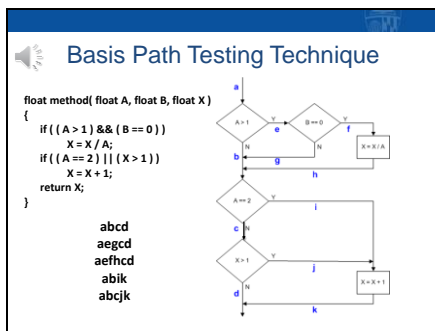
Now...a little bit of history behind cyclomatic complexity and how it relates to basis path testing. This metric comes from graph theory. The flowchart for a software component can be considered to be a form of a mathematical graph. And, there's a theorem in math that says that all the paths through a graph are members of a vector space...and that the dimension of that vector space equals the cyclomatic number associated with the graph. For software, the paths correspond to execution paths and the cyclomatic number corresponds to the cyclomatic complexity.

The theorem further states that there exist at least one set of paths, called a basis set, that have the following properties: first, they are independent from one another...kind of like our x and y axes but at a more complex level...and second...the remaining paths in their vector space can be generated with a simple linear combinations of the basis set paths...kind of like every point in two-dimensional space can be constructed from an x value and a y value.

From a testing standpoint, suppose we have a software component that has a cyclomatic complexity of five. And, suppose it has a thousand possible execution paths. The cyclomatic theorem tells us that there exists at least one set of 5 paths that are independent from one another, and from which the remaining 995 could be constructed. So, in some respects, if we found those 5 basis paths and tested them, then all the remaining execution paths could be formed from some combination of those five...so some would say that testing more than the basis paths is, in some respects, redundant. So...how can we use this in practice?

In practice, if we find a set of basis paths for a software component and test those paths, guess what? Every statement, branch, and condition will also be exercised. So...we can use basis path testing as a quick way of coming up with an effective covering set of test cases.

10



Let's re-visit our earlier example. Here's the code for our very simple method and an associated flowchart. I wrote the flowchart a little differently than before...by breaking down the compound decision constructs into primitive level decisions.

Take a few seconds to calculate the cyclomatic complexity of this method. It should be five. There are two "if" keywords, an "and" keyword represented by the double ampersand, an "or" represented by the double vertical bar. Adding one yields a cyclomatic complexity of five.

And... here's a set of basis paths. If we develop a set of test cases that exercise the basis paths then every statement, branch, and condition will have been tested.

