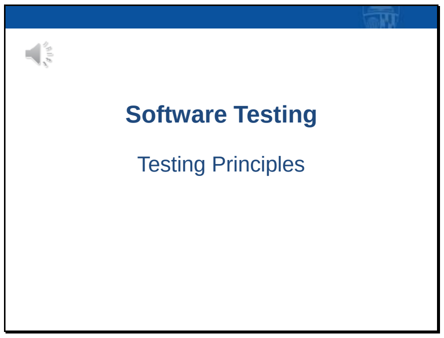
1

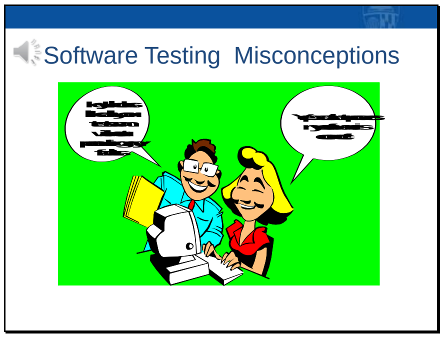


Software Testing

Testing Principles

In this lecture we'll discuss basic software testing principles

2



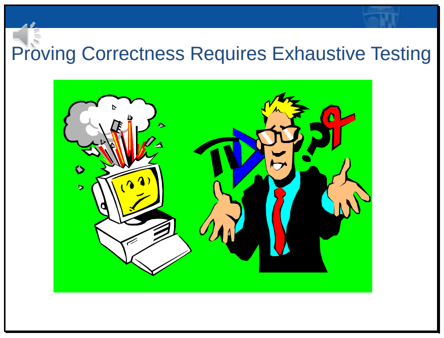
Software Testing Misconceptions

There are a number of misconceptions about software testing in practice. One major misconception is that software testing can prove that a software product is free from defects. Testing is a defect detection activity...and just because a suite of tests runs successfully doesn't prove that a product is defect-free. Depending upon how the test suite is designed, such a result might give us a reasonable degree of confidence that we think the product is defect-free...but it does not suffice as a proof.

When I talk to some practitioners about this, particularly managers, it often shatters some long-held beliefs.

In practice, dynamic testing can only show the presence of defects...not their absence.

3



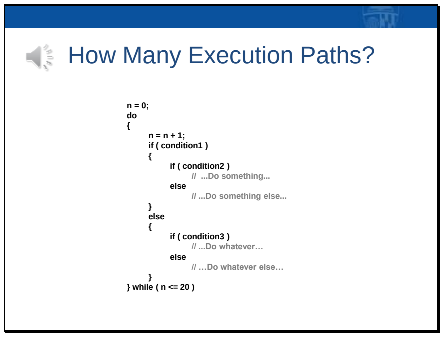
Proving Correctness Requires Exhaustive Testing

In order for dynamic testing to prove the correctness of a software product we would have to do exhaustive testing...basically, exposing the product to every sequence and combinations of input conditions that are possible.

For example, in the triangle exercise we did earlier, to prove through dynamic testing that the product always evaluates an equilateral triangle correctly, we would have to execute it using every valid triplet of equal numbers representing the values of the sides. If the product uses integers internally to represent the values of sides, and the product was coded using the Java language, then there would be more than two billion tests required...since the range of valid positive values for an integer variable in Java is 1 to 2,147,483,647.

If it took one second to run and evaluate each of those tests, the total time required would be more than 596,523 hours…the equivalent of about 68 years…just to prove that one function. Hence, my use of the term "exhaustive" testing.

4



How Many Execution Paths?

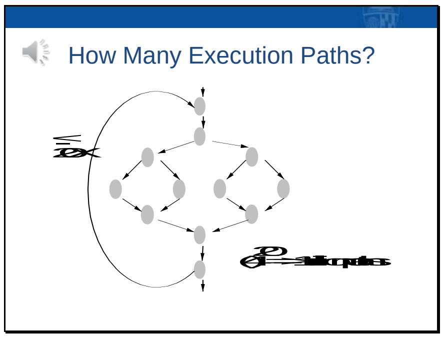
```
n = 0;
do
{
    n = n + 1;
    if ( condition1 )
    {
        if ( condition2 )
            //  ...Do something...
        else
            // ...Do something else...
    }
    else
    {
        if ( condition3 )
            // ...Do whatever…
        else
            // …Do whatever else…
    }
} while ( n <= 20 )
```

Rather than test every possible combination of program inputs, which is clearly infeasible, some practitioners feel that it is sufficient to run enough tests to ensure that every possible program execution path is exercise at least once under test...since different input data values can generally cause some of the same paths to be exercised. This also becomes exhaustive as well.

As an example...here's some pseudo-code for a verysimple function. For our purposes, it's not important to care about what the function does...it's just important to understand its structure.

The code consists of one iteration construct that repeats 20 times...and has 3 if-else constructs. Now...how many possible execution paths are there in this code? It's kind of hard to figure out by looking at the code. So, let's look a a flowchart instead.

5



How Many Execution Paths?

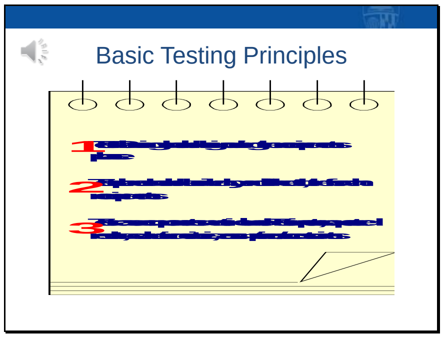
Here's a flowchart version of the code. It's a lot easier to calculate the number of execution paths from it. For any iteration of the looping construct there are 4 possible paths that could be executed...and there are 20 iterations...so there are 4 to the 20th power possible execution paths. That's equivalent to just over 10 to the 12th power...or just over a trillion paths.

So...whether we look at a products input domain or execution paths...exhaustive testing is not feasible in practice.

What does that tell us? It tells us that there is risk associated with testing...in particular, dynamic testing.

I'm going to introduce some basic testing principles in this lecture, that will be elaborated on as we progress through this course module, and which can help to mitigate some of the risks associated with relying on testing to ensure product correctness.

6

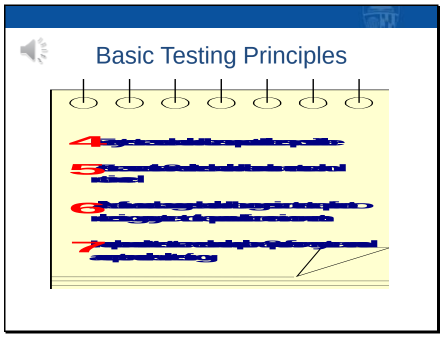


I‘m going to pretty much just mention some basic testing principles here...and I‘ll elaborate on them in subsequent lectures.

The first two principles are that testing activities should begin as early as the requirements phase, and that test plans should be driven by, and linked to, the product requirements.

The third principle states the components of a test case. This is very important, and I often see it violated in practice. Recall that triangle exercise that you did earlier. When I give that exercise in my in-person seminars, I often see people writing down test case data that corresponds to the input values for the sides of the triangle. But they usually fail to write down the expected result for each set of inputs...and that‘s not good. Expected results for each test case, and how the expected and actual results will be compared...which is what I mean by evaluation criteria...need to be mandatory components of a test case.

7



Principle number four states that a test case should be as repeatable as possible. This is important for tests that have to be re-run and for regression testing.

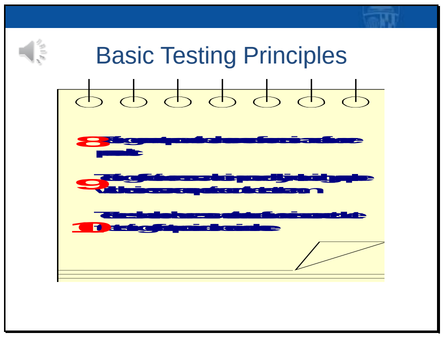
Now, in practice, this may not always be able to be done, but it should be incorporated as much as possible. An example of kinds of tests that may not be able to be completely repeatable would be those that are run off of a database...may be a database of transactions that are being used to test a number of different functions. If transactions are added or deleted, the state of the database can change. This can be dealt with in some situations by either keeping a baseline repository or by having tests that undo what was done by other tests...but this may be difficult to do when the same database is being shared among several testers.

Principle five is pretty obvious...keep a test repository.

Principle six involves regression testing. Now, in practice, it usually isn‘t necessary to completely re-run all tests whenever a change is made, particularly when testing at the system level. And...using a test traceability table, which you‘ll see an example of a bit later, and really help in determining what tests may need to be re-run.

Principle seven is an important one. In practice, more defects are generally found during testing in projects where an independent test team participates. For smaller projects, and in some organizations where testing is more informal and the testing is done by developers, independent test teams may be difficult to apply. In those situations, it is extremely important that developers be trained in effective testing techniques.

8



Basic Testing Principles

[illegible]

I already discussed principle eight, so I won‘t repeat myself here.

Principle nine is also a very important one, particularly in environments where there have been historical problems with the quality of requirements. In that earlier triangle exercise, we saw that because the requirements were incomplete...missing an important business rule...it was possible that testing would not uncover the problem in which the sum of any two sides must be strictly greater than the third side rule was not built into the product. Since, in that example, the root cause of the problem was in the requirements, it would have been quicker and cheaper and best to have caught the problem in the requirements phase...but, if the problem wasn‘t caught there, at least having the right business expertise as part of the test effort would increase the likelihood of catching it during testing.

The final principle is also very important...and is one of the reasons that independent testing adds value. When developers test their own software, there‘s often a tendency to focus on conformance-directed tests instead of fault-directed tests...and there‘s an underlying assumption that the product is correct.