# Module 6

## Software Testing

**Objectives:**

1. Understand what is entailed by unit testing, integration testing, and system testing.
2. Understand several strategies for performing unit testing.
3. Know what the essential components of a test case are.
4. Know the difference between validation testing and defect testing.
5. Gain experience performing a cyclomatic complexity analysis on source code.
6. Understand the difference between software verification and software validation (V & V).
7. Become familiar with several techniques used in V & V.

**Reading:**

1. *Sommerville*, chapter 8.

**Assignment:**

Complete the cyclomatic complexity assignment for Module 6.

**Quiz:**

Take the quiz for Module 6.

**Who likes testing?**

Software testing is one of the least coveted tasks associated with building software. Why? Some of the reasons that are usually given include:

1. It's boring. Who wants to sit around all day long running test case after test case after test case...?

2. It's tedious. Thorough testing usually involves thousands of test cases.

3. Testing is a task usually assigned to interns or entry-level programmers, so if you're stuck doing testing you've either gotten yourself demoted or you aren't considered all that valuable of an employee.

4. Testing doesn't allow you to be creative, like design and coding do.

5. You can test like crazy and still miss a bug, and if that bug causes problems after release, you might be the one who gets in trouble for it, not the engineer who actually wrote the code.

6. Did I mention boring and tedious?

7. Other reasons are possible...

The list can change, depending on the project and the organization. Nevertheless, most software engineers would rather spend their time designing or coding.

**OK, you may not like testing, but here's why it's important:**

The catalog of software blunders is regrettably large. Here are only a few examples. By the way, I don't have first-hand accounts of any of these examples, so if anyone has any comments on the accuracy of the information, please share them.

<u>The Ariane-5 Disaster</u>

On its maiden voyage in 1996, the Ariane-5 rocket exploded 40 seconds after liftoff. The disaster cost the European space program 10 years of effort and around $7 billion. The rocket and cargo alone were valued at about $500 million. It was determined that the cause of the crash was software failure. A 64-bit floating point number was erroneously converted to a 16-bit signed integer. Since the number was greater than 32,767, the conversion failed. A detailed report can be found at: http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

<u>The Therac 25</u>

This one may have been mentioned earlier in the course. The Therac 25 was a medical linear accelerator used for radiation therapy treatments of cancer patients. Several patients died as a result of massive accidental overdoses of radiation. A very good resource for this particular example can be found at: http://www.computingcases.org/case_materials/therac/therac_case_intro.html. Here is a quote from that source:

"These massive radiation overdoses were the result of a convergence of many factors including

- simple programming errors
- inadequate safety engineering
- poor human computer interaction design
- a lax culture of safety in the manufacturing organization
- inadequate reporting structure at the company level and as required by the U.S. government

Some of the actual case reports are frightening. One man apparently screamed about his face feeling like it was on fire. This is a good example of a socio-technical system.

Patriot Missile Snafu in the Persian Gulf War

(There is some controversy over this particular case, but I will post what I have gleaned from the consensus I found). In the 1991 Gulf War a Patriot Missile failure allowed a SCUD missile to hit an army barracks, killing 28 U.S. soldiers. According to reports there was a rounding error in a software subroutine that calculated elapsed time since the system had been booted. The magnitude of the error increased the longer the system was maintained in operation without rebooting it. The end result was that the Patriot battery in question ignored a SCUD missile, dismissing it as a valid target. You can find information on this case all over the internet. A very brief accounting can be found at:
http://www.ima.umn.edu/~arnold/455.f96/disasters.html.

Software Hijacks Plane?

In September 2005, s oftware in the Air Data Inertial Reference Unit (ADIRU) aboard a Malaysian Airlines 777 flying from Perth to Kuala Lumpur was apparently responsible for trying to cause the aircraft to stall while in flight. The system also did not want to relinquish control of the plane to the pilot. It'll be interesting to see what the actual underlying cause was. An article on this one can be found at:
http://catless.ncl.ac.uk/Risks/24.05.html#subj1.

Granted, all the examples above involve critical systems. Software failures in non-critical systems are more likely to cause annoyance, or some loss of profit, than death. The unifying theme in most software failures, though, is that the vast majority of them could have been prevented if only the software had been thoroughly tested by engineers who know how to test software.

**Categories of Faults**

Software failures are a direct result of software faults. Software faults, like so many other things in life, can be categorized and grouped according to their characteristics. Some examples of fault categories, or fault classes, include:

1. Algorithmic Faults

   These are logical faults caused by incorrect reasoning. Somewhere in deriving the set of steps for solving some problem, a mistake was made.

2. Syntax Faults

   Syntax faults can be caused by typos or a misunderstanding of how programming language constructs are built. Fortunately, a good compiler can catch most of these, virtually assuring they will never make it into released software.

3. Computation Faults

   Computation faults can be caused by using the wrong formula, neglecting to account for precision (especially when converting from one type to another), or going beyond the range of values the language can handle.

4. Overload Faults

   These faults exceed the system's capacity in one way or another. Two examples of overload faults are buffer overruns and exceeding available RAM.

5. Capacity Faults

   Capacity faults are very similar to overload faults, and sometimes are a result of overload faults. Examples include pushing the CPU to its maximum operating capabilities, and bogging the system down with excessive network traffic.

6. Timing/Coordination Faults

   These faults occur predominantly in real-time systems, where operations must be completed within narrow time windows. If an operation is not completed before its time window expires, a failure occurs.

7. Performance Faults

   Here, performance refers to how the user perceives the system is performing. Whether or not the system fails is often dependent upon the user. For example, one user who types very slowly may not see a problem with having to wait 1 second for a character to appear on the screen when using a word processor. A user who types quickly, however, will see such performance as a failure.

8. Recovery Faults

   Recovery faults only apply to those systems that have built-in failure recovery mechanisms to restore the state of a system following a software failure or a power failure. The faults, in this case, lie in the routines responsible for recovering the system.

## Unit Testing

Unit testing is used to test individual “units” in isolation from other “units”. A unit can be a single method, a single module, or a single class, depending on the situation. Unit testing is usually the first level of testing that is performed. Usually this is automated as much as possible. Automating unit testing involves writing test harness modules, sometimes known as scaffolding code, that can pass inputs to a unit and capture the output. Modules that provide the input are called drivers.

## Test Cases, Test Suites, and Oracles

Most of you are probably already aware of what the essential components are for a single test case, but I’ll repeat them here. You need at least 2 things:

1. test input(s)

2. expected output

That is, you need to have input in order to get any output at all, and you need to know what the correct result should be in order to determine whether the test case passed or failed. I like to add a third item: the rationale for the test case. Every test case should have a good reason for being created and used, and including the rationale can help ensure you are covering all the important fault classes.

A **test suite** is merely the collection of all the test cases you intend to use. An oracle is something that compares the actual output from an executed test case with the expected output and returns a result of either pass or fail. An oracle can be a human that manually compares test results with the expected results, or it can be integrated into an automated testing system. The latter often requires significant effort, but can be enormously beneficial in that it can save you a lot of time. You have to make sure your trust in the automated oracle is warranted, though.

## Black Box vs. White Box Testing

These two categories of testing are distinguished by whether or not you get to see the source code when you design your test cases. In white box testing, sometimes called clear box testing or open box testing, you **do** get to see the source code, and you use the source code as a guide for designing your test cases. In black box testing, also called closed box testing, you **do not** get to see the source code.

Both types of testing should be employed. White box testing allows you to examine the details of the implementation, but does not necessarily test the logic behind the implementation. If the implementation does not match the requirements, white box testing will not detect it. Black box testing may detect it, because without the source code to give you ideas for test cases, you are forced to turn to the requirements to make sure none of the requirements are violated. Black box testing used in this way is also sometimes called **acceptance testing**, since the software is only accepted if it passes its acceptance tests.

## Strategies for White Box Testing

There are a lot of things you can do when you have access to the source code during test case design. Below I list several strategies for testing a particular piece of code.

1. Make sure all classes of inputs are tested. For example:

    a. negative numbers
    b. positive numbers
    c. zero
    d. boundary values (e.g., values at upper or lower limits of a loop, array, etc.)
    e. values of incorrect data type (e.g., character instead of integer, floating point instead of integer)
    f. null values
    g. the empty set

2. Statement testing – make sure every statement is executed at least once.

3. Branch testing - every branch of every decision point is executed at least once. It is possible to have complete statement testing without complete branch testing. For example, look at the code below:

```
void foo(int x)
{
      if (x > 0)
            System.out.println(x);

      System.out.println(x * x);
}
```

    In this code you can achieve complete statement testing with a single test input (e.g., x == 3). This does not give you complete branch testing, however, because it does not test the case where the condition in the "if" statement is false. To get complete branch testing you also need to include a second test input that will cause the condition to evaluate to false, such as x == -4.

4. Path testing – every distinct path through the code tested at least once. This can be tricky to do, depending on the code, but it guarantees you will get both complete branch coverage and complete statement coverage. To do path testing, you first need to calculate the cyclomatic complexity of the code. This is done by creating a flow graph for the code, in which each statement is represented by a node, and edges connect nodes (statements) that are executed in sequence. Once the flow graph is made, you can calculate the cyclomatic complexity, V(G), in one of three ways:

a. *V(G) = # of regions* (a region is an area enclosed by edges; there is always one additional region, the region surrounding the entire flow graph)

b. *V(G) = E – N + 2*, where *E* = # of edges, *N* = # of nodes

c. *V(G) = P + 1*, where *P* = # of predicate nodes (nodes having 2 or more edges going out of them; i.e., it is a node involved in a decision)

The resulting value tells you the minimum number of distinct paths needed to achieve complete statement and branch coverage. The higher the complexity, the more distinct paths there will be. Generally you want to keep this number low, if possible. If the value is high, you can reduce it by breaking down the functionality into several smaller methods or units. The exact boundary between "low" and "high" is arbitrary. Figure 6.1 shows a diagram of the flow graphs for commonly encountered programming constructs.

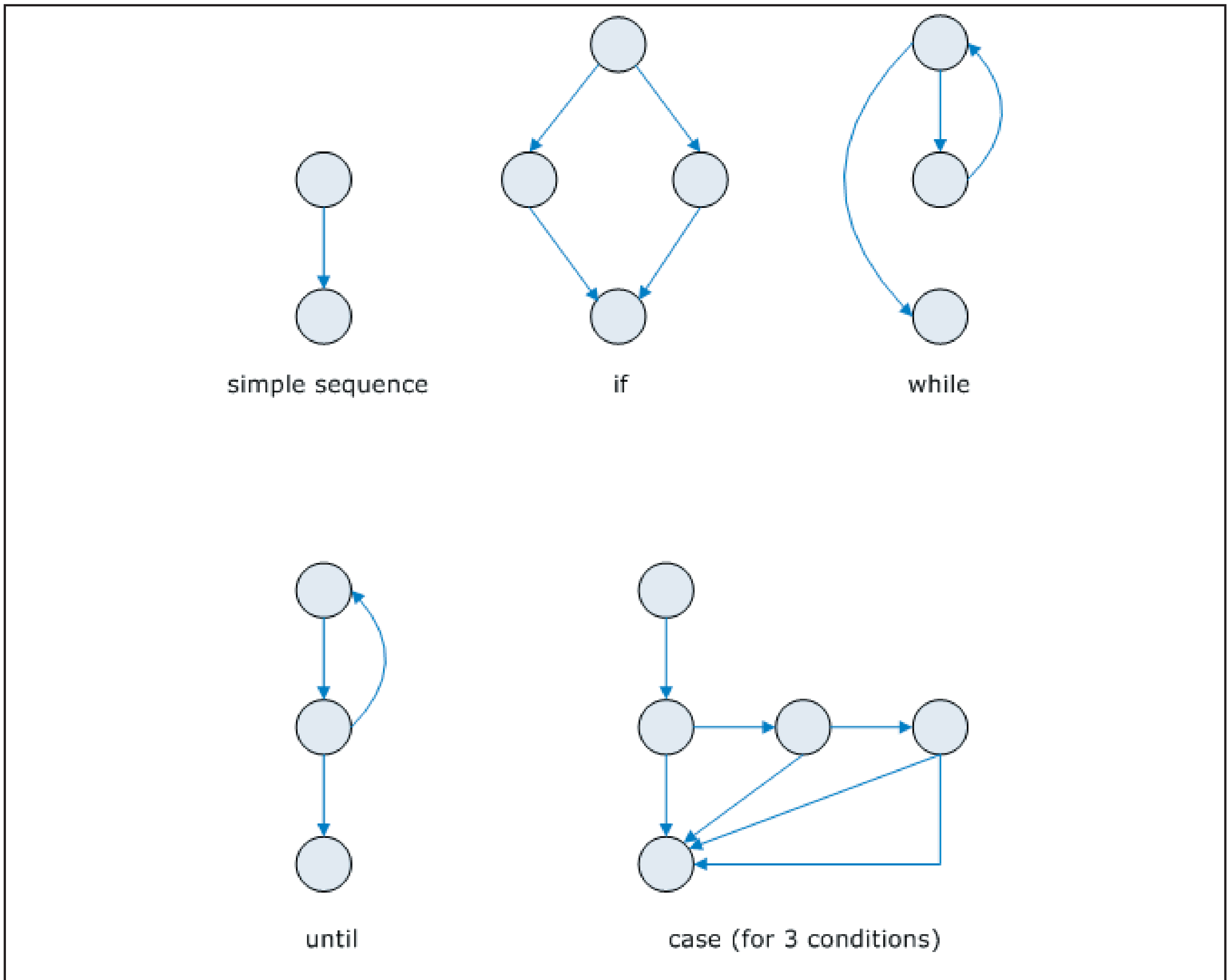


Figure 6.1. Flow graphs for common programming constructs.

## Cyclomatic Complexity Example

To better understand how cyclomatic complexity is used, let's look at an example. In Figure 6.2 I show the code for a LinkedList method, getNode, that returns the node at a specified index in the list. The statements relevant for determining cyclomatic complexity have been numbered to the left in red. Note that each boolean condition counts as a separate statement, so some lines may have more than one number. Each number will represent a node in the flow graph. An extra, "termination" node is included at the very end. All paths must end at that node. The resulting flow graph is shown in Figure 6.3.

```
private Node getNode( int idx )
{
1   Node p;

2,3 if( idx < 0 || idx > size( ) )
4       return null;

5   if( idx < size( ) / 2 )
    {
6       p = beginMarker.next;
7       for( int i = 0; i < idx; i++ )
8           p = p.next;
    }
    else
    {
9       p = endMarker;
10      for( int i = size( ); i > idx; i-- )
11          p = p.prev;
    }

12  return p;
}

13  (end of method)
```

Figure 6.2. Code for an internal method of a LinkedList that retrieves a node at a particular index.

In Figure 6.3, each numbered statement has been converted to a node in the graph. Directed edges indicate the sequence of execution from one node to the next. The cyclomatic complexity has been calculated using all three methods, and the results are shown in the lower left corner of Figure 6.3. This particular piece of code has a cyclomatic complexity of 6, which means complete statement and branch coverage can be achieved using a minimum of 6 test cases. The 6 basis paths are:

1. 1 --> 2 --> 4 --> 13
2. 1 --> 2 --> 3 --> 4 --> 13
3. 1 --> 2 --> 3 --> 5 --> 6 --> 7 --> 12 --> 13
4. 1 --> 2 --> 3 --> 5 --> 6 --> 7 --> [8 --> 7]* --> 12 --> 13
5. 1 --> 2 --> 3 --> 5 --> 9 --> 10 --> 12 --> 13
6. 1 --> 2 --> 3 --> 5 --> 9 --> 10 --> [11 --> 10]* --> 12 --> 13

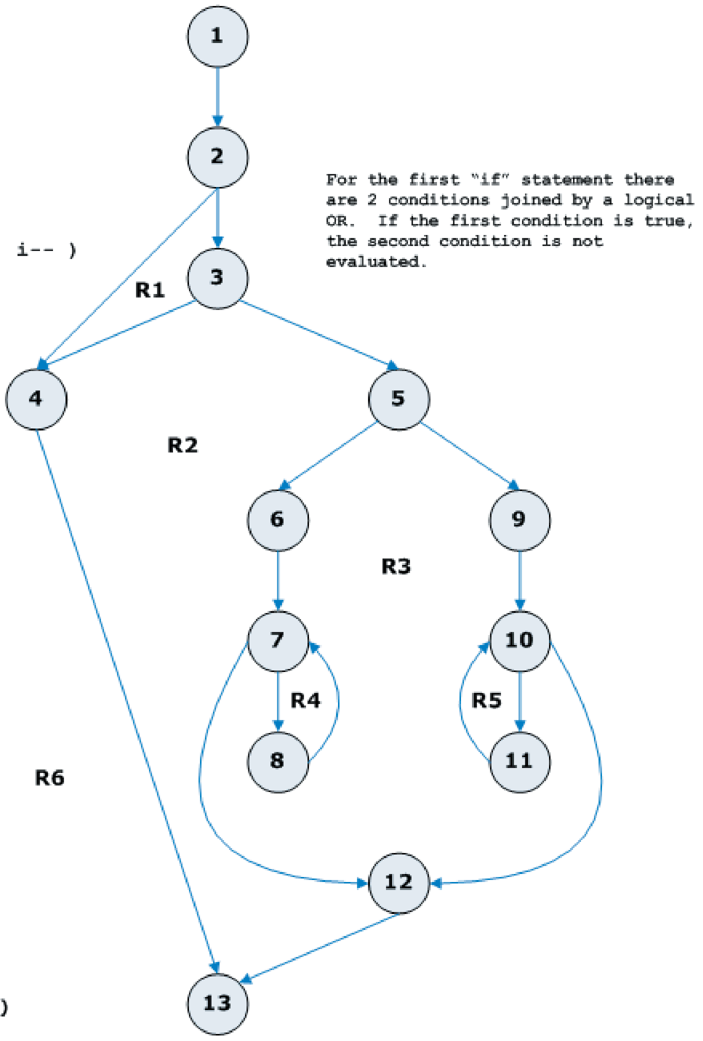
```
    private Node getNode( int idx )
    {
1       Node p;

2,3     if( idx < 0 || idx > size( ) )
4           return null;

5       if( idx < size( ) / 2 )
        {
6           p = beginMarker.next;
7           for( int i = 0; i < idx; i++ )
8               p = p.next;
        }
        else
        {
9           p = endMarker;
10          for( int i = size( ); i > idx; i-- )
11              p = p.prev;
        }

12      return p;
    }

13      (end of method)
```



1
2
3
4
5
6
7
8
9
10
11
12
13
R1
R2
R3
R4
R5
R6
For the first "if" statement there are 2 conditions joined by a logical OR. If the first condition is true, the second condition is not evaluated.

**calculate *V(G)*:**

1. ***V(G) = # of regions = 6***

2. ***V(G) = E – N + 2 = 17 – 13 + 2 = 6***

3. ***V(G) = P + 1 = 5 + 1 = 6***
   **(Nodes 2, 3, 5, 7, and 10 are predicate nodes)**

Figure 6.3. Flow graph for the code shown in Figure 6.2.

Statements enclosed in brackets and followed by a "*" indicate that execution can loop between those statements an arbitrary number of times. Technically only one loop is needed for a basis path.

A word of caution is in order here. Just because path testing gives you both statement and branch coverage does not mean you should restrict yourself to only those test cases indicated by the basis paths. Rather, the basis paths should be considered as a starting point for your tests. For example, where loops are concerned it is usually a good idea to check at least 1, 2, and 3 iterations, in addition to any cases where the loop might be bypassed.

This brings up an interesting question: is it possible to achieve complete path testing? Not always. Some loops can potentially iterate indefinitely. For example, a loop that waits for user input will execute until input is provided. There is no way to completely test every path in such code, since each iteration of the loop essentially constitutes a new path. The same reasoning can be applied to code that is only executed in response to a hardware interrupt. Each increment of additional time waiting for the interrupt is part of a new path through the code.

**Integration Testing**

It is very tempting to think that as long as all components pass unit testing, there should be no problem when they are all assembled. Unfortunately, it doesn't always work out that way. Once the individual components have been integrated, problems may emerge that could not be detected during unit testing. An example of this is the additive effects of loops. Suppose you have a method that uses a single loop that runs in O(N) time. That method calls another method that also runs in O(N) time. The second method calls a third method that runs in O(N). The end result is an actual running time of O(N3). The individual methods may run fine, but when they are integrated, the system comes to a standstill.

Integration testing does not always have to wait for unit testing to be completed. It can be started as soon as there are unit-tested components that can be linked together. There are three basic strategies for approaching integration testing. All three strategies assume the individual units can be related in a hierarchical fashion, with higher-level, controller units at the top and lower-level, menial units at the bottom.

1. <u>Bottom-up</u>

   The bottom-up approach starts with the lowest level units and proceeds upwards towards higher-level units. Test drivers are needed to provide inputs if automated testing is to be performed. Less scaffolding code is required for this approach, but the higher-level units cannot be tested until later. This sometimes poses problems since the higher-level units tend to be more important, and testing them last means they may not receive sufficient attention.

2. <u>Top-down</u>

   Top-down integration testing begins with the higher-level units and proceeds downwards to the low-level units. This allows the high-level units to receive more testing, but at a price. In order to test from the top down, additional test modules known as stubs must be written to mimic the behavior of downstream units.

3. <u>Sandwich</u>

   Both the top-down and the bottom-up approaches suffer from one fundamental problem. They are both one-way processes, meaning the units at the opposite end can't be tested until end of integration testing. The sandwich strategy addresses this problem by doing both top-down and bottom-up integration testing simultaneously.

**System Testing**

System testing, as the name implies, tests the entire system. It is the final level of integration testing. When this is done in lieu of unit and integration testing it is known as "big-bang" testing (probably named because you usually end up with a big system that blows up when you run it!).

System testing is not usually geared toward finding traditional bugs. Instead, it is more concerned with things like:

1. resource consumption (e.g., RAM, virtual memory, external storage, network bandwidth, etc.)

2. overall performance (speed, smoothness, load, operating capacity, etc.)

3. security

4. disaster recovery

5. how well it works with other software

6. detection of instabilities that may arise with increased time in operation

Initial system testing is usually performed in-house, where the software was built. It is also always a good idea to do system testing after the system has been installed in its actual operating environment. In-house system testing cannot always accurately simulate the true environment the software will be dealing with.

## Regression Testing

Suppose, in the midst of testing, one of your test cases fails. You identify a fault in your code and correct it. You rerun that test case, and it passes. You should be free to continue with the next test case, right? Wrong. How do you know that your bug fix didn't inadvertently create a new bug? If it did, how do you know whether this bug will be detected by your remaining test cases, or by test cases you have already run? The only way you can know for sure is to rerun your entire test suite on the code. When you rerun your entire test suite in this way it is known as **regression testing**. Regression testing is absolutely vital whenever you make any changes to your code, either to fix a bug or to deliberately change the functionality.

If you are manually inputting the test values, you may have a very long road ahead of you, which is why automated testing is especially handy. When a change is made to the code, you just rerun the automated test suite. You get the results more quickly, and you don't have to worry about forgetting to include any of the tests, since they are all stored as part of the automated testing process.

## Assessing the Quality of the Testing Process

This lecture is quite long already, so I will conclude with two final topics that are related to each other. Thr first topic is, how do you know for sure how effective your testing process is at finding faults? You may have a test suite with 100,000 test cases, but how do you know whether it's finding all the bugs? In actuality, you can never be 100% sure, but there is something you can do to give you some idea how effective your testing process is: **fault seeding**. Fault seeding involves deliberately planting bugs in the source code. The types and locations of the bugs are known and recorded so they can be distinguished from real faults. You then run your test suite, and determine how many of the seeded faults were detected. The percentage of seeded faults found gives you a rough idea of the effectiveness of your test suite.

For example, suppose your test suite found 40% of the seeded faults. You would assume your test suite probably only found about 40% of the real faults as well, which means that it probably missed about 60% of the real faults. Obviously there is room for improvement here. What if you test suite found 100% of the seeded faults? Can you assume it also found 100% of the real faults? No, because you have no way of proving the correlation between the seeded faults and the real faults is 100% accurate. However, if you do this correctly, you can assume that your test suite is pretty good, and that it probably found most of the real faults.

The second topic is, what happens if your test harness has bugs? You could get false positive test results. That is, some tests may pass, when they should have failed. This means, of course, that you need to test your testing code. It generally does not mean, however, that you then have to test the code you used to test your testing code, etc. That would be too impractical. Fortunately, though, the bulk of automated testing code involves doing the same types of activities, such as reading input data from a test file, passing the input to a unit, and then doing something with the output such as writing it to a separate result file or comparing the results with an oracle. Most bugs can be eliminated by avoiding hardcoded data, and making sure the correct filenames are used.

Incidentally, you also need to make sure your test data are correct, meaning they contain input values that are appropriate for the cases they are testing. Test data files can contain mistakes the same as source code files.

## Verification and Validation

In chapter 8, Sommerville makes it a point to distinguish between **validation** and **verification**. Both terms apply to the testing phase of development. Validation ensures the right software has been built; that is, the software satisfies the expectations set forth by the customer. Verification ensures the software conforms to its specifications; that is, the software is correct. Verification also ensures the proper standards and practices were used in building the software. Validation is less strict than verification in that an incorrect product can be considered valid if the customer is satisfied with it.

Validation and verification (a.k.a. V & V) is a lengthy and costly process. In addition to testing the code in the ways described in this module, there may be periodic code reviews and inspections, where the code is not actually executed, but it is statically examined for faults.

V & V is so important to ensuring the quality of software that most larger organizations have staff dedicated to these tasks. In most cases, though, it is better to have an independent entity perform validation and verification in addition to any V & V performed by the organization itself. The reason is that the organization has a vested interest in getting the software completed and put into operation, and is therefore biased. Defects may be “overlooked” or ignored in order to meet deadlines or rush the software to market. Independent V & V entities have no such vested interest. Their livelihood will not be threatened regardless of whether the software succeeds or fails.