10 Testing and inspecting to ensure high quality

Disciplined attention to requirements analysis, modeling and design helps produce good quality software that solves the customers' problems. However, any human activity, no matter how carefully performed, will involve mistakes. It is therefore necessary to verify meticulously that the system performs as required. In this chapter we discuss two important verification techniques, testing and inspection. We will also put these in the context of quality assurance as a whole.

In this chapter you will learn about the following

- The distinction between such terms as 'failure' and 'defect'.
- Strategies for efficiently and effectively finding defects in software, by testing and inspecting it.
- Particular types of defects and how to detect them.
- How to write test cases.
- Other ways to help ensure quality such as root cause analysis, and implementing a process standard.

10.1 Basic definitions

People often talk rather interchangeably about such things as 'errors', 'mistakes', 'bugs', 'failures' and 'defects'. It is important, however, carefully to define and distinguish among these terms so that the discussion in the remainder of the chapter is clearer.

An extreme and easily understood kind of failure is an outright crash. However, any violation of requirements should be considered a failure,

Definition: a *failure* is an unacceptable behavior exhibited by a system.

including such things as the production of incorrect outputs, the system running too slowly, or the user having trouble finding online help.

The frequency of failures, as encountered by testers and end-users, allows you to measure the *reliability* of a system. An important design objective is to achieve a very low failure rate, and hence high reliability.

Most failures are violations of requirements that are stated *explicitly* in a requirements document. However, it is possible to have a failure resulting from a violation of an *implicit* requirement – something not written in the requirements document, but which the user or tester discovers when running the system. Such a failure indicates that the requirements need improving. Remember that failures may arise from violations of either functional or quality requirements.

Definition: a *defect* is a flaw in any aspect of the system including the requirements, the design and the code, that contributes, or may potentially contribute, to the occurrence of one or more failures. A defect is also known as a *fault*.

The techniques presented in this chapter are intended to enable software engineers to discover and remove defects.

Often it takes several defects, working together, to cause a particular failure. For example, imagine in the SimpleChat program that a user tries to forward messages to himself or herself. It would be a design defect if it were possible to set up such a forwarding situation to start with. Imagine, nevertheless, that this first defect existed. In the worst case, such a defect would result in an infinite loop, and a resulting 'hang' or crash, as the system tries to forward the same message over and over again. However, this worst case could only occur if there were a second defect, where the system does not detect, during the actual sending of a message, that it is forwarding a message around a loop. A tester would only be able to uncover the second defect if the first were present. On the other hand, an inspector, studying the code in detail, might notice both. In this chapter we will study both testing and inspecting.

Definition: an *error* is a slip-up or inappropriate decision by a software developer that leads to the introduction of a defect into the system.

An error can be made at any stage of the software development process, from requirements to implementation and maintenance. Improved education and disciplined approaches to software engineering should lead to fewer errors, and hence fewer defects and fewer failures.

Pejorative words for bad things

There are a wide variety of words for 'bad' things that happen in software engineering, and the words are often used with subtly different meanings.

In colloquial speech, 'failures' and 'defects' are often not distinguished from each other – they are both simply called 'bugs' or 'problems'. However, software developers should make an effort to distinguish consciously between failures and defects. Doing so helps them to think more clearly about the quality, or the lack of quality, in a system. It also allows them to monitor and measure that quality, and thus improve both the product and the development process.

In addition to meaning a 'mistake' made by a person, the word 'error' also has a second meaning: the amount of deviation from the correct value in a numerical calculation, due to rounding off or truncation. We will discuss such errors later in this chapter. When a software engineer makes an error, it is a 'bad thing'; however, when a round-off error occurs, it is only bad (i.e. a defect) if it is not anticipated and correctly handled.

Exercise

E193 Categorize the following according to whether each describes a *failure*, a *defect* or an *error*:

(a) A software engineer, working in a hurry, unintentionally deletes an important line of source code.

(b) On 1 January 2040 the system reports the date as 1 January 1940.

(c) No design documentation or source code comments are provided for a complex algorithm.

(d) A fixed size array of length 10 is used to maintain the list of courses taken by a student during one semester. The requirements are silent about the maximum number of courses a student may take at any one time.

10.2 Effective and efficient testing

Testing is the process of deliberately trying to cause failures in a system in order to detect any defects that might be present. As with all engineering activities, efficiency and effectiveness are both important. To test *effectively*, you must use a strategy that uncovers as many defects as possible. To test *efficiently*, you must find the largest possible number of defects using the fewest possible tests. An effective and efficient testing strategy is often called a *high-yield* strategy.

Black-box and glass-box testing

Most of the time, testers treat a system as a *black box*. This means they provide the system with inputs and observe the outputs, but they cannot see what is

Properly handled exceptions are not failures

Object-oriented programs throw 'exceptions'; these should not be confused with failures. In most cases, exceptions are thrown in situations that are anticipated by programmers, such as reaching the end of a file.

It is a failure, however, when an unexpected exception is thrown. The corresponding defect is the absence of code to handle the exception. Even in these cases, though, it is better that an exception is thrown, and debugging information captured, rather than the program crashing with no clue as to what happened.

It is particularly bad practice to blindly catch all exceptions without justification; defects can then remain hidden. In particular, code like the following should be avoided if possible:

```
try {...} catch(Exception e) {/* do nothing */}
```

going on inside. In particular, they can see neither the source code, the internal data, nor any of the design documentation describing the system's internals.

An alternative approach to testing is to treat the system as a *glass box*. In glass-box testing, the tester can examine the design documents and the code, as well as observe at run time the steps taken by algorithms and their internal data. He or she can therefore design tests that will exercise all aspects of each algorithm and data structure. Glass-box testing is widely referred to as *white-box* testing or structural testing; however, we prefer the term glass-box testing since the notion of looking inside a glass box clearly provides a better metaphor.

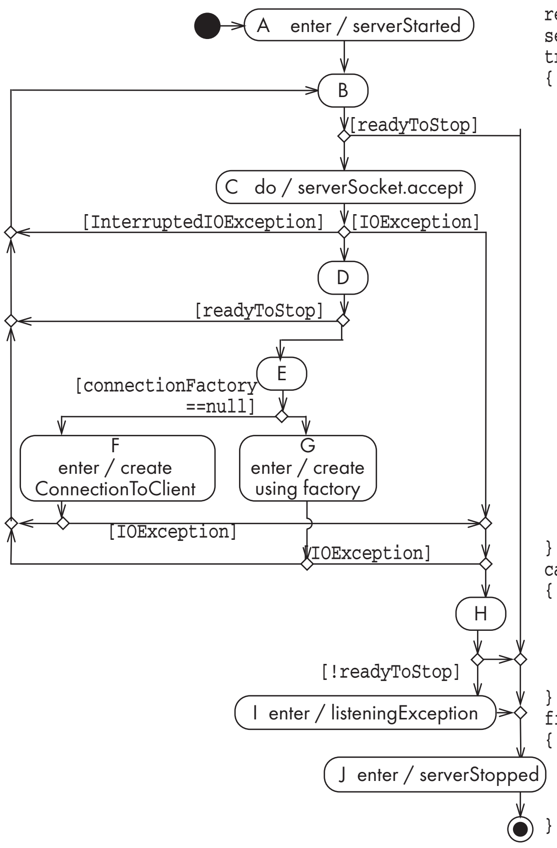
Glass-box testing is more time-consuming than black-box testing, but removes much of the guesswork, and allows the tester to be more thorough. When performing glass-box testing, a tester can analyze the code to ensure that his or her testing strategy has reached a targeted *coverage* of statements and branches: the tester can ensure that, for example, 90 per cent of all statements are executed and 80 per cent of all branches are taken.

Individual programmers often informally employ glass-box testing when they are verifying their own code. It is only used as part of a formal testing process when testing critical or complex components. Therefore, most of the techniques we will discuss in this chapter are oriented towards black-box testing.

Example 10.1 *Discuss how you would employ glass-box testing to test the run method of the OCSF* `AbstractServer` *class found on the book's web site (www.lloseng.com).*

Using glass-box testing means that you will base your testing strategy on the actual implementation, rather than the requirements.

To help do this systematically, you can first draw a flow graph of the code as shown in Figure 10.1. This graph has a node for all the possible places where the code can make a branch (such as `if` statements, `while` statements and statements that can throw an exception), the places where those branches can lead (the bodies of loops, `if` or `else` clauses, and `catch` statements), and the places where two separate paths come together (the statements after a loop or if-then-else body, as well as `finally` clauses or the statements after `catch` clauses).



```
readyToStop= false;
serverStarted();     // A
try
{
  while(!readyToStop)  // B
  {
    try
    {
      Socket clientSocket = serverSocket.accept();  // C
      synchronized(this)
      {
        if (!readyToStop) // D
        {
          if (connectionFactory == null) {  // E
            new ConnectionToClient(         // F
              this.clientThreadGroup, clientSocket, this);
          } else {
            connectionFactory.createConnection(    // G
              this.clientThreadGroup, clientSocket, this);
          }
        }
      }
    }
    catch (InterruptedIOException exception) { }
  }
}
catch (IOException exception)
{
  if (!readyToStop)   // H
  {
    listeningException(exception);  // I
  }
}
finally
{
  readyToStop = true;  // J
  connectionListener = null;
  serverStopped();
}
```

Figure 10.1 **Flow graph of the OCSF AbstractServer run method. The code is extracted from that found on the book's web site (http://www.lloseng.com)**

In Figure 10.1, we have labeled each node with a letter. This letter is also shown as a comment in the code. We show calls to callback methods in appropriate nodes.

Once you have a flow graph you must then choose from the following strategies for glass-box testing:

- **Covering all possible paths**. This is infeasible in graphs with loops since there would be an infinite number of paths (i.e. repeatedly looping).
- **Covering all possible edges**. This is probably the most efficient strategy. You devise a sufficient set of tests to make sure that each of the outgoing edges of all nodes is taken. The following tests would ensure edge coverage of Figure 10.1; a black-box tester would most likely not be able to devise a similar set of tests. The most difficult aspect of executing these tests is to force I/O exceptions to be thrown at the correct time, as in tests 3 through 6. Bold text is used below to show the first time a given edge is covered.

1. **A-B-C-B**-C-…-B-**C-D-E-F-B**-C-…-B-C-**B-J**: run the server, connect a client, and send the stop command.
2. A-B-C-B-C-…-B-C-D-E-**G-B**-C-…-B-C-B-J: repeat the same sequence, but this time use a connection factory.
3. A-B-C-B-C-…-B-**C-H-I-J**: same as 2, but force an I/O exception to occur while accepting the connection.
4. A-B-C-B-C-…-B-C-D-**E-G-H-J**: same as 2, but force an exception to occur during creation of the connection class (e.g. during stream creation); also send the stop command at the same time so that path H-J is taken.
5. A-B-C-B-C-…-B-C-D-E-**F-H**-J: same as 4, but with a regular `ConnectionToClient`.
6. A-B-C-B-C-…-B-C-**D-B**-J: same as 1, except send the stop command just after the connection is accepted.

- **Covering all nodes**. This is a less exhaustive, and therefore less effective strategy. For example, in the above, tests 1, 3 and 4 together cover all nodes, but the remainder are needed to cover all the edges.

Testing is like detective work

The job of the tester has certain similarities with that of the detective:

- A detective must try to understand the criminal mind. Similarly, the tester must try to understand how programmers, designers and users think, so as to better find defects. For example, detectives know that criminals tend to repeat certain patterns of behavior (their *modus operandi*); if detectives know these patterns, they can more easily track down the criminals. Similarly, testers know that software developers implement algorithms in certain ways and tend to have habits that can lead to errors, and hence to defects. Testers can uncover defects by anticipating typical errors made by developers. Testers can also uncover defects by anticipating unusual things that users might try to do.
- Detective work is painstaking. The detective must not leave anything uncovered, and must be suspicious of everything. Similar suspicion and attention to detail are the hallmark of an effective tester. However, like detective work, it does not pay to take an excessive amount of testing time – as we mentioned earlier, both tester and detective have to be *efficient*.

Equivalence classes: a strategy for choosing what to test

Imagine you are asked to test a Java method `validMonth` that takes an `int` argument that is supposed to correspond to a valid month. The method returns `true` if the input is in the range 1 to 12 inclusive, and `false` otherwise. We will look at how you should test such a month-validation method to make sure it behaves correctly.

To confirm correct behavior, it is equally as important to check that the method returns `false` in the case of an invalid month, as it is to check that it returns `true` in the case of a valid month.

One possible testing strategy would be to use *brute force*, i.e. to test the method using *every possible* value of Java's `int` datatype – all integers from -2^{31} to 2^{31}−1. Common sense, however, tells us that such testing of all possibilities would not only take such a huge amount of time as to be impractical, but would also be pointless. It would be pointless because any given defect is likely to cause failures with many different input values. For example, if a failure always occurs with input 243, then it is almost certain that a failure will also occur with input 245. Hence you do not need to test both values. Our knowledge of software design and programming tells us that there is no reason to believe 243 and 245 would be treated any differently by any reasonable algorithm that might be used to validate a month number.

Therefore, in order to test efficiently, you should divide the possible inputs into groups that you believe will be treated similarly by reasonable algorithms. Such groups are called *equivalence classes.* A tester needs only to run one test per equivalence class, using a *representative* member of that equivalence class as input (plus boundary tests that we will discuss later). For the month-validation problem, there are three equivalence classes, as shown in Table 10.1.

Table 10.1 **Equivalence classes for month validation, where the input value is a Java `int`**

Equivalence class	*Range of values*
Invalid – larger	13 to $2^{31}-1$ (can be loosely expressed as 13 to $\propto$)
Valid	1 to 12
Invalid – smaller	-2^{31} to 0 (can be loosely expressed as $-\propto$ to 0)

As mentioned, determining equivalence classes helps the tester to do efficient yet thorough black-box testing. However, determining the equivalence classes is often not easy. The tester has to understand the required input and the domain-specific rules that govern what input is acceptable. Also, the tester has to have knowledge of computer science and software design. For example, he or she would have to know about the ranges of integer data types in order to determine the upper and lower bounds of the two invalid equivalence classes in the month-validation problem.

Exercise

E194 Create a table of equivalence classes for each of the following single-input problems. Some of these might require some careful thought and/or some research. Remember: put an input in a separate equivalence class if there is even a slight possibility that some reasonable algorithm might treat the input in a special way.

(a) A telephone number to be used by an automatic dialer.

(b) A person's name (written in a Latin character set).

(c) A time zone, which can be specified either numerically as a difference from UTC (i.e. GMT), or alphabetically from a set of standard codes (e.g. EST, BST, PDT).

(d) The speed of a vehicle, which is a 3-digit integer. This may be followed by the units 'km/h', 'm/s' or 'mph'. 'km/h' is the default.

(e) A credit card number.

(f) An FM broadcast radio frequency.

(g) A URL.

Combinations of equivalence classes

Imagine you are designing a system that is to contain information about all kinds of land vehicles, including passenger vehicles and racing vehicles. Such a system might require the user to enter specifications of a new type of vehicle.

Your job is to decide how to divide this system into equivalence classes for testing. You are told that the user can enter the following data:

1. Whether the manufacturers give data about the vehicle in metric or traditional (Imperial or US) units. The user selects this using pair of 'radio buttons'. This impacts how the rest of the data will be interpreted (two equivalence classes; there is no possibility of invalid input).

2. Maximum speed, an integer ranging from 1 to 750 km/h or 1 to 500 mph (four equivalence classes: [−∝..0], [1..500], [501..750], [751..∝]).

3. Type of fuel, one of a set of 10 possible strings that the user explicitly types (11 equivalence classes – the eleventh class is any string other than the 10 valid ones).

4. Average fuel efficiency, a fixed-point value with one decimal place, ranging from 1 to 240 L/100 km or 1 to 240 mpg (three equivalence classes).

5. Time to accelerate to 100 km/h or 60 mph. This is a fixed-point value with one decimal place, ranging from 1 to 100s (three equivalence classes).

6. Range, an integer from 1 to 5000 km or 1 to 3000 miles (four equivalence classes: [−∝..0], [1..3000], [3001..5000], [5001..∝]).

The first thing to notice is that each of the six inputs has its own set of equivalence classes. A second thing to notice is that some of the inputs *depend* on the values of others. In particular, the valid values for inputs 2 and 6 depend on whether metric or traditional units has been specified as input 1. This adds an extra equivalence class for inputs 2 and 6, giving them four each instead of three.

A string-valued input, such as input 3, would have just one equivalence class if any arbitrary string were valid and no algorithm would actually try to process the string. However, in the case of input 3, every legal value should be considered a separate equivalence class because the tester must verify that the designer has explicitly written code to recognize every valid type of fuel. Since the user types strings, the program must do string comparisons to determine which fuel type is entered; a common bug in the code would be to misspell one of the fuel types in the code that makes the comparison.

The set of equivalence classes for the system as a whole is the set of all possible *combinations* of inputs. The total number of equivalence classes for the system is therefore the product of the number of classes of the individual inputs; in this case 2×4×11×3×3×4 = 3168. Since, like this system, most systems have many distinct inputs, the total number of system equivalence classes can become very large. This is called a *combinatorial explosion* of the space of required tests.

The combinatorial explosion implies that you cannot realistically test every possible system-wide equivalence class. A reasonable approach to testing is therefore as follows. You should first make sure that at least one test is run with every equivalence class of every individual input. Then you should also, where possible, test all combinations where one input is likely to *affect the interpretation of* another. Therefore, for example, you should use both metric and traditional units (specified by input 1) when running tests for inputs 2 and 6.

Using the above strategy, the number of tests for the vehicles example can be reduced to 11. You run one test for each equivalence class of input 3; while doing this you also vary the eight possible combinations of inputs 1 and 2, the eight combinations of inputs 1 and 6, and the three possible equivalence classes of the other two inputs. You should test a few other random combinations of equivalence classes.

In addition to testing representative values from each equivalence class, you should also test values at the boundaries of equivalence classes. This is discussed in the next subsection.

Exercise

E195 Describe a good set of equivalence class tests for the following situations:

(a) A personal information form that asks for family name, first name, date of birth, street address, city, country, postal code and home telephone number.

(b) The set of commands available in the client side of SimpleChat Phase 2.

(c) The GANA system. Hint: inputs come from other places than just the user interface.

Testing at boundaries of equivalence classes

More errors in software occur at the boundaries of equivalence classes than in the 'middle'. For example, when testing a method that inputs an integer and checks whether a valid day number has been entered for the month of November, the valid equivalence classes are integers less than 1, integers from 1 to 30 and integers greater than 30. However, the most likely error is that the number 31 is accepted as valid (i.e. the system treats it the same way as integers from 1 to 30). Another common error might be the acceptance of zero as a valid day number.

Therefore we should expand the idea of equivalence class testing to specifically test values at the extremes of each equivalence class.

Exercise

E196 Expand each of your answers to include equivalence class boundary values that should be tested.

Detecting specific categories of defects

As mentioned earlier, a tester must act like a detective, trying to uncover any defects that other software engineers might have introduced. This means not only testing at equivalence classes and their boundaries, but also designing tests that explicitly try to catch a range of specific types of defects that commonly occur.

The next few sections give a non-exhaustive list of some of these most common categories of defects. This list will help you to design appropriate test cases. It will also be useful when designing software, since it will help you to create designs that avoid these defects.

10.3 Defects in ordinary algorithms

The following subsections list some of the most common kinds of defects found in all types of algorithms. In some cases, these algorithmic defects can be found in the specification, but they are more often introduced by the designer or the programmer.

Incorrect logical conditions

Defects The logical conditions that govern looping and if-then-else statements are wrongly formulated. Sometimes a condition needs completely restructuring, but often the defect is more subtle, such as nesting parentheses incorrectly, reversing comparison operators (e.g. **>** becomes **<**), or mishandling the equality case (e.g. **>=** becomes **>** or vice versa).

Testing strategy Use equivalence class and boundary testing. To compute the equivalence classes, consider each variable used in the logical condition as an input.

Example Imagine that an aircraft's alarm is supposed to sound if the landing gear is not deployed when the aircraft is close to the ground. The specifications might state this as follows: 'The landing gear must be deployed whenever the plane is within 2 minutes from landing or take-off, or within 2000 feet from the ground. If visibility is less than 1000 feet, then the landing gear must be deployed whenever the plane is within 3 minutes from the landing or lower than 2500 feet.' Note that in this case a false alarm is just as bad as a failure of the alarm to sound when it should.

Java code for the alarm might be written as follows (in deliberately bad style):

```
if(!landingGearDeployed &&
  (min(now-takeoffTime,estLandTime-now))<
    (visibility < 1000 ? 180 :120) ||
  relativeAltitude <
    (visibility < 1000 ? 2500 :2000)
  )
{
  throw new LandingGearException();
}
```

Unfortunately, this type of bad style that gives rise to defects is rather a common practice. There is at least one critical defect in this code – see if you can understand the code and find the defect. It is likely that a programmer might not notice it due to the nested parentheses and the overall complexity of the condition.

As shown in Table 10.2, the inputs are the variables used in the condition. The total number of system equivalence classes in the aircraft example would be 3×3×3×2×2 = 108. Since this number is manageably small, and since this is a safety-critical system, all the classes should be tested.

Figure 10.2 shows how the equivalence classes relate to each other. The gray area indicates the classes where the alarm should sound if the landing gear is not deployed. The most critical classes are those next to the boundary of the gray area, each of which is marked with a letter. In these classes, a change in a single variable can affect the outcome. Table 10.3 describes the equivalence classes we need to test areas *a* and *d*.

We also need to perform boundary tests for each of the equivalence classes. For example, testing what happens if the visibility is zero, very close to 1000 feet, and unlimited. Particularly important boundary cases occur while the plane is on the ground, since it is likely that special processing might take place while there.

Table 10.2 **Equivalence classes of conditions that can lead to a decision about whether the landing-gear alarm should sound**

Variable affecting condition	*Equivalence classes*
Time since take-off	3: Within 2 minutes after take-off, 2–3 minutes after take-off, more than 3 minutes after takeoff
Time to landing	3: Within 2 minutes prior to landing, 2–3 minutes prior to landing, more than 3 minutes prior to landing
Relative altitude	3: < 2000 feet, 2000 feet to 2500 feet, 2500 feet
Visibility	2: < 1000 feet, 1000 feet
Landing gear deployed	2: true, false

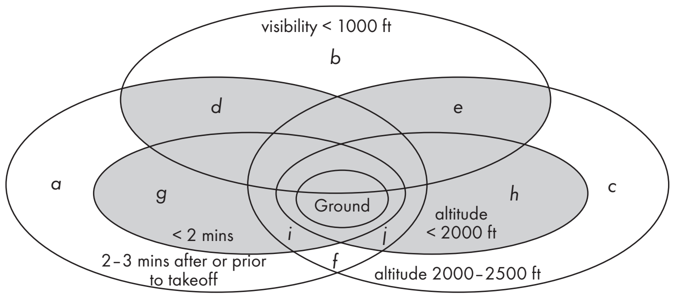


Figure 10.2 **Venn diagram, with shading showing situations where the landing gear must be down, and hence the alarm must sound if it is not down. Conversely, the alarm should never sound in the unshaded areas**

Table 10.3 **The eight system equivalence classes corresponding to the boundary between regions *a* and *d* of Figure 10.2**

Flight stage	*Visibility*	*Landing gear*	*Required result*
2–3 minutes after take-off	≤ 1000 ft	Deployed	No alarm
2–3 minutes prior to landing	≤ 1000 ft	Deployed	No alarm
2–3 minutes after take-off	≤ 1000 ft	Not deployed	No alarm
2–3 minutes prior to landing	≤ 1000 ft	Not deployed	No alarm
2–3 minutes after take-off	< 1000 ft	Deployed	No alarm
2–3 minutes prior to landing	< 1000 ft	Deployed	No alarm
2–3 minutes after take-off	< 1000 ft	Not deployed	Alarm sounds
2–3 minutes prior to landing	< 1000 ft	Not deployed	Alarm sounds

You might ask the question, how can I test a landing-gear system like this since it is to be embedded in a real aircraft? The answer is that the alarm module needs to be thoroughly tested in a *simulated* environment before being tested in a real aircraft.

Exercises

E197 Write Java code for the aircraft landing-gear example and test it according to the equivalence classes and boundary cases suggested. Record and fix any defects. Note that to save time, you should write a program to test it automatically (this is called a *test harness*).

E198 Create tables that will help you test the conditions corresponding to the following requirements.

(a) User C sends a message to A. The message must be forwarded from A to B if the following are all true: 1) A has requested that his or her messages be forwarded to B; 2) B is logged on; and 3) B has neither requested that messages from A be blocked nor that messages from C be blocked. Furthermore, if the message is declared to be an emergency message then no blocking occurs.

(b) Applicants under the age of 18 should use form A, while those 18 or above should use form B. However, disabled people of any age, except those on social assistance, should use form C. People on social assistance should complete form D in addition to either A or B. Senior citizens should complete form E in addition to the other forms, unless they are on social assistance or are living in a subsidized nursing home. Anybody who is not disabled and not a senior citizen, and who earns over £15,000 per year, should complete form F in addition to other forms.

(c) The navigation system must announce to drivers that they have to turn onto a new road two minutes in advance of the required turn. Exceptions arise in the following circumstances. Firstly, if a driver is still completing a previous turn, then the warning of the new turn should be delayed until the first turn is complete – although 15 seconds warning must still be given. Secondly, in a city environment (where a driver driving at the speed limit would encounter at least one possible turn every 30 seconds), the system should warn the driver only 30 seconds before the turn is required.

(d) The burglar alarm system should sound the alarm immediately if a movement of magnitude greater than 4 is detected while the alarm is activated. An alarm should also sound if a movement of magnitude 3 persists for more than 5 seconds, or a movement of magnitude 2 persists for more than 15 seconds. Additionally, an alarm should sound if the total amount of magnitude 2 or 3 movement exceeds 30 seconds in any 2-minute period. If two or more sensors detect movement simultaneously, as might be the case in an earthquake, then the above time-periods are doubled. The alarm is never sounded within 1 minute of activation.

Performing a calculation in the wrong part of a control construct

Defect The program performs an action when it should not, or does not perform an action when it should. These are typically caused by inappropriately excluding the action from, or including the action in, a loop or if-then-else construct.

Testing strategies Design tests that execute each loop *zero* times, exactly *once*, and *more than once*. Also, ensure that anything 'bad' or 'unusual' that could happen while looping is made to occur on the first iteration and the last iteration.

This kind of defect is not always reliably caught using black-box testing; in such cases glass-box testing or inspections may be more effective.

Examples The following Java code illustrates a typical case:

```
while(j<maximum)
{
  k=someOperation(j);
  j++;
}
if(k==-1) signalAnError();
```

In this case, `signalAnError` was supposed to be called if any of the calls to `someOperation` resulted in a value of –1. Unfortunately, here it can only be called following the last call to `someOperation`. The final line should therefore have been placed inside a loop. This kind of defect may be missed if the loop normally executes only once.

Another common type of control-construct defect occurs in the following situation (using deliberately bad style):

```
if(j<maximum)
  doSomething();
if(debug) printDebugMessage();
else doSomethingElse();
```

Here, the lack of curly brackets and proper indenting has obscured the logic. A programmer added the third line while debugging, but that means that while not debugging, `doSomethingElse()` is always called, which was not intended. Diligent testing should be able to detect this.

Not terminating a loop or recursion

Defect A loop or a recursion does not always terminate, that is, it is 'infinite'.

Testing strategies Although the programmer should have analyzed all loops or recursions to ensure they reach a terminating case, a tester should nevertheless assume that the programmer has made an error. The tester should analyze what causes a repetitive action to be stopped, and should run test cases that the tester anticipates might not be handled correctly.

Example Imagine that a program is supposed to count the total number of atoms in a complex organic molecule. It might do this by starting at an arbitrary molecule and traversing it from bond to bond, walking down each branch.

A tester might wonder, however, whether the algorithm would work correctly if there were one or more *circular* structures in the molecule; he or she might be suspicious that a circular structure could cause the algorithm to try to loop forever, counting ever higher as it went round and round the molecule. Of course, any reasonable programmer should also have thought of this situation, but the tester must not trust this.

Not setting up the correct preconditions for an algorithm

Defect When specifying an algorithm, one specifies *preconditions* that state what must be true before the algorithm should be executed. A defect would exist if a program proceeds to do its work, even when the preconditions are not satisfied.

Testing strategy Run test cases in which each precondition is not satisfied. Preferably its input values are just beyond what the algorithm can accept.

Example In the organic chemistry program, a precondition might be that the input is a single molecule. The tester should therefore try to test by giving as input two disjoint molecules.

Not handling null conditions

Defect A null condition is a situation where there normally exists one or more data items to process, but sometimes there are none. It is a defect when a program behaves abnormally when a null condition is encountered. In these situations, the program should 'do nothing, gracefully'.

Testing strategy Determine all possible null conditions and run test cases that would highlight any inappropriate behavior.

Examples Imagine you want to calculate the average sales of members of each division in an organization. But what if some division has no members? Perhaps they all quit, or the division has just been formed and nobody has been hired. A typical defect would be that this relatively unusual situation results in an attempt to divide by zero (zero sales, divided by zero members).

As a related example, imagine you are asked to find the salesperson who has sold the most in the above division. In attempting to perform this calculation, an algorithm might loop zero times and hence never actually set the maximum value, or leave it set to some arbitrary, but incorrect value.

Not handling singleton or non-singleton conditions

Defect A singleton condition is like a null condition. It occurs when there is normally *more than one* of something, but sometimes there is only one. A non-singleton condition is the inverse – there is almost always *one* of something, but occasionally there can be more than one. Defects occur when the unusual case is not properly handled.

Testing strategy Brainstorm to determine unusual conditions and run appropriate tests.

Examples The following are two examples of these conditions:

- Imagine that in a web browser you can set up a series of 'personal profiles'. Each user of the computer has their own personal profile that includes their name, bookmarks list, and 'cookies'. Imagine you created a personal profile under your name, 'John Smith'. Later on you accidentally created another profile, using the same name. Then you decided to get rid of one of the two profiles; you therefore selected a profile and issued the 'delete' command. Unfortunately, the deletion operation might assume that there can be only one profile using a given name, so that it simply traverses the list of profiles, deleting *all* the profiles by that name. The result? Both of the profiles are deleted. Anticipating this kind of defect, a tester can enter several identically-named profiles and make sure that only one is deleted.
- Imagine that a program is designed to randomly assign members of a sports club into pairs who will play against each other. Does the program do something intelligent with the left-over person when there is an odd number of members? And what happens if there is only one member? (Maybe the club should be disbanded, but it would still be best if the program did not crash.)

Off-by-one errors

Defect A program inappropriately adds or subtracts 1, or inappropriately loops one too many times or one too few times. This is a particularly common type of defect.

Testing strategy Develop boundary tests in which you verify that the program computes the correct numerical answer, or performs the correct number of iterations.

Since graphical applications are common places where off-by-one errors are found, study the display to see if objects slightly overlap or have slight gaps.

Examples Assuming 0-based indexing, as is the case in Java, then the following loop would always skip the first element, and loop one too few times.

```
for (i=1; i<arrayname.length; i++) { /* do something */ }
```

If 1-based indexing had been used, as in some other programming languages, then the code would inappropriately skip the *last* element. Note that the use of the `Iterator` class in Java helps prevent such coding defects, but the job of the tester is to assume the worst and guess that designers have made errors like this.

Similar errors can occur when using a prefix operator instead of the postfix operator in certain calculations. In the following example, the variable `val` is probably being incremented too early, so that its initial value is not actually passed to `anOperation`.

```
while (iterator.hasNext())
{
  anOperation(++val);
}
```

Operator precedence errors

Defect An operator precedence error occurs when a programmer omits needed parentheses, or puts parentheses in the wrong place. Operator precedence errors are often extremely obvious, but can occasionally lie hidden until special conditions arise.

Testing strategy In software that computes formulae, run tests that anticipate defects such as those described in the example below. However, code inspections, discussed later, are likely to be better at catching this kind of defect.

Example A program may compute `z+(x*y)`, when it was supposed to compute `(z+x)*y`. In this case, the programmer probably wrote `z+x*y` and forgot that multiplication takes precedence over addition. If `z` is normally zero, or all three variables are normally 1, then this defect could remain hidden. Testing for errors like this therefore means thinking carefully about which values of `x`, `y` and `z` to use.

Use of inappropriate standard algorithms

Defect An inappropriate standard algorithm is one that is unnecessarily inefficient or has some other property that is widely recognized as being bad.

Testing strategies The tester has to know the properties of algorithms and design tests, such as those in the following examples, that will determine whether any undesirable algorithms have been implemented.

Examples There is not enough space in this book to discuss choice of algorithms and algorithm analysis – that is the subject of separate books. Nevertheless, the following are some bad choices of algorithms that testers should try to detect:

- ❑ **An inefficient sort algorithm**. The most classical 'bad' choice of algorithm is sorting using a so-called 'bubble sort' instead of a more efficient approach to sorting. A tester can test for such a defect by increasing the number of items being sorted and observing how execution time is affected. A bad sorting algorithm will increase with the square of the number of items – if you *double* the number of items a bad sorting algorithm will take about *four* times as long. A better algorithm will normally increase much more slowly. Imagine, for example, that you sort 10,000 items and notice that the system takes 5 seconds. Then you double the number of items to 20,000. If a bad algorithm were being used, you would expect the time to increase to about 20 seconds. On the other hand, if a better quality algorithm were being used, you would expect sorting time to be instead about 11 seconds on average, which is slightly more than double the time.

- **An inefficient search algorithm**. You would expect that searching through a very large list of sorted items should be done quite rapidly. You can test for this in a similar manner to testing for an inefficient sort algorithm. Ensure that the search time does not increase unacceptably as the list gets longer. Also check that the position in the list of the item you are looking for does not have a noticeable impact on search time.
- **A non-stable sort**. A non-stable sort will take equal elements and sometimes switch their order after the sorting process. For example, imagine that you start with a list of people ordered alphabetically by their name. If you now sort the people alphabetically by city, you would normally want to see the people in each city still sorted by name. A tester should therefore deliberately run an experiment like this to see if the results are as expected. A non-stable sort would make the names of people in each city appear randomly, or at best imperfectly ordered.
- **A search or sort that is case sensitive when it should not be, or vice versa**. The tester should test algorithms with mixed-case data to see if the algorithm behaves as expected.

Note that the first two types of inappropriate algorithm will only be noticed by users when the amount of data to be handled is very large. In either of these cases, if only a few dozen items are ever likely to be in a list, then the tester need not worry about the algorithm.

Exercise

E199 Java has a built-in sorting capability, found in classes `Array` and `Collection`. Test experimentally whether these classes contain efficient and stable algorithms.

10.4 Defects in numerical algorithms

Numerical computation defects are a special class of algorithmic defect. They can occur in any software that performs mathematical calculations, especially calculations involving floating point values. Whole books are devoted to numerical algorithms; the following are some of the typical classes of defects that testers should try to find.

Not using enough bits or digits to store maximum values

Defect A system does not use variables that are capable of representing the largest possible values that could be stored. When the capacity of a data type is exceeded, an unexpected exception might be thrown, or else the data may be stored incorrectly.

Testing strategies Test using very large numbers to ensure that the system has a wide enough margin of error.

Example Imagine that you were going to be storing the monthly salary of an employee in a short integer (whose value ranges up to 32765). Although this limit would work for most employees, failures would occur in the case of a) highly-paid executives, b) a period of high inflation, or c) a foreign currency that has previously experienced high inflation and now uses very large numbers for ordinary transactions.

Using insufficient places after the decimal point or too few significant figures

Defects This problem occurs with floating point or fixed-point values. A floating-point value might not be 'wide' enough to store enough significant figures. A fixed-point value might not store enough places after the decimal point. These defects force the system to round excessively, which can mean that data is stored inaccurately and can also lead to a build-up of errors as discussed in the next defect category.

Testing strategies Perform calculations that involve many significant figures, and large differences in magnitude. Verify that the calculated results are correct in such cases.

Example Imagine an investment application that tracks a portfolio of shares. Typical shares are quoted using three or four significant digits, hence, prices might be \$135.5 or \$33.16. However, if a share 'crashes' in value for some reason, it might drop to only a few cents. In such a case, your system might have to record its value as \$0.0344. If your system were only able to record values to two decimal places after the point, then it could not correctly manipulate such stocks.

Ordering operations poorly so that errors build up

Defects This defect occurs when you do small operations on large floating-point numbers, and excessive rounding or truncation errors build up. In particular, if you take a very large floating-point number, for example 3.54×10^{28}, and add or subtract a very small number from it, for example 1, then the answer will be exactly equal to the large number. This is because the large floating-point number does not store enough significant figures for the operation to have any effect on it. It is not, per se, a defect that this occurs. However, it is a defect if the programmer intended the large number to be modified. Although the above example was an extreme case, this kind of defect can also occur in less obvious situations, as illustrated in the examples below.

Testing strategies If a numerical application is designed to work with floating-point numbers, then make sure it works with inputs that vary widely in magnitude, including both large positive and large negative exponents. Pay particular attention to the accuracy of the result when a floating-point value is being repeatedly decremented or incremented by small amounts.

Examples Imagine a fictitious processor that stored only four significant figures. Now imagine you stored the value 9876 in a variable and wanted to subtract the following values from it: 0.42, 0.35, 0.27 and 0.47. If you subtracted these values from 9876 one after the other, the result would remain 9876. This is a defect because you know that subtracting any very small value from 9876 will always result in 9876, to four decimal places. On the other hand, if you perform the calculation by first adding the small values to each other (result = 1.51) and then subtracting this from 9876, the result will be 9874, which is correct.

A similar example: imagine you are adding a large number of small credits to an account. If the account's total were in the thousands of dollars, you might always round it to the nearest dollar. However, in this situation, small transactions of a few cents would never affect the account balance.

Assuming a floating-point value will be exactly equal to some other value

Defect If you perform an arithmetic calculation on a floating-point value, then the result will very rarely be computed exactly. It is therefore a defect to test if a floating-point value is exactly equal to some other value. You should instead test if it is within a small range around that value.

Testing strategies Standard boundary testing should detect this type of defect.

Example The following is at risk of resulting in an infinite loop, since d may never equal precisely 10.0, but may instead equal 9.99999999999 after 10 iterations.

```
for (double d = 0.0; d != 10.0; d+=2.0) {...}
```

The correct expression should have been:

```
for (double d = 0.0; d < 10.0; d+=2.0) {...}
```

Exercise

E200 Describe the type of numerical defects present when the following are implemented in a program. Assuming you did not actually know the implementation, describe how you would attempt to detect such defects.

```
(a) double x, y;... if(x/3.0 == y) {...}
```

```
(b) int totalCorporateAssets; // In Euros
```

```
(c) short priceOfOil; // In US 10ths of a cent per gallon
```

(d) You run a web site that implements micro-payments; it charges users a third of a cent for every page click. You accumulate each client's bill in an integer where each unit represents a 10th of a cent. However, you record the money earned by each page in a float value.

10.5 Defects in timing and co-ordination: deadlocks, livelocks and critical races

Timing and co-ordination defects arise in situations involving some form of concurrency. They occur when several threads or processes interact in inappropriate ways. In this section we will use the word 'thread' to mean both 'thread' and 'process'.

The three most important kinds of timing and co-ordination defects are *deadlocks*, *livelocks* and *critical races*. Ways to design software so as to avoid these situations are discussed in books on real-time and concurrent software. Here we present the concepts of these defects and some thoughts about how to test for them.

Deadlock and livelock

Defects A deadlock is a situation where two or more threads or processes are stopped, waiting for each other to do something before either can proceed. Since neither can do anything, they permanently stop each other from proceeding. A classic example of real-life deadlock is the 'gridlock' sometimes encountered in busy cities. This is illustrated in Figure 10.3. Vehicles waiting to move in one direction hold up vehicles waiting to travel in other directions; however, each set of vehicles is ultimately delaying itself. Everybody therefore waits indefinitely.

As a software example of deadlock, imagine that thread A is waiting to access object O. Object O is locked by thread B, perhaps using Java's synchronization mechanism. Thread B, however, might be waiting to access object P, which is in turn locked by thread A. Neither thread can ever continue its work unless some outside thread forces a break in the deadlock. This situation is illustrated in Figure 10.4.

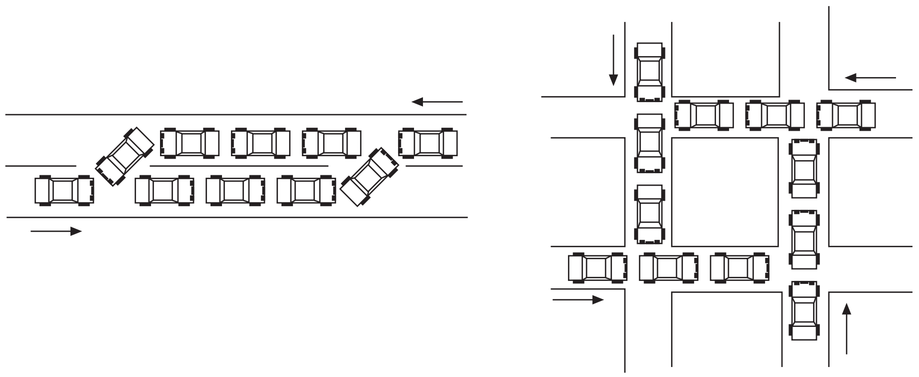


Figure 10.3 Two examples of gridlock: a form of deadlock

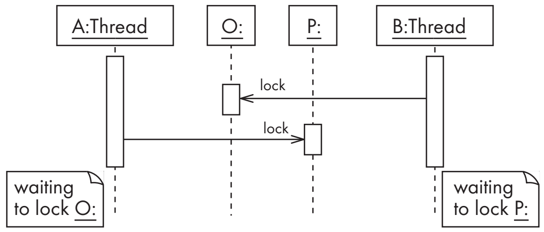


Figure 10.4 **A deadlock situation in software**

Livelock is similar to deadlock, in the sense that the system is stuck in a particular behavior that it cannot get out of. The difference is as follows: whereas in deadlock the system is normally hung, with nothing going on, in livelock, the system can do some computations, but it can never get out of a limited set of states.

The left part of Figure 10.5 shows livelock in traffic. Several cars are trapped in a grid of one-way streets, in which some directions are blocked, perhaps by accidents or road construction. The cars can keep moving round and round in circles, but they cannot go anywhere outside of the loop.

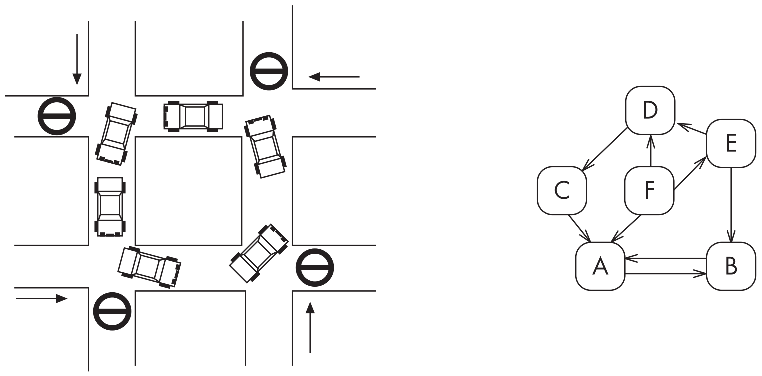


Figure 10.5 **Livelock in traffic (left) and software (right)**

The right part of Figure 10.5 shows a livelock situation in a state diagram. There are several transitions leading into states A and B; but once there, the system can only alternate backwards and forwards between the two states.

Both deadlocks and livelocks can appear as 'hung' systems. The difference is that in livelock the system can continue to consume CPU time. When you encounter a hung system it is certainly a failure, but it is not necessarily caused by a deadlock or livelock: the hang could result from a single thread waiting for an unavailable resource, an infinite loop, or a crash. Also, not all deadlocks and livelocks will completely hang a system, since other threads may still be running.

Testing strategies Deadlocks and livelocks tend to occur as a result of unusual combinations of conditions that are hard to anticipate or reproduce. It is often most effective to use *inspection* to detect such defects, rather than testing alone. There are some tools that can be used to detect deadlock potential, but these are beyond the scope of this book.

Whether testing or inspecting, a person with a background in real-time systems should be employed – such a person can apply his or her knowledge and experience to best anticipate timing and co-ordination defects.

If its cost is justifiable, glass-box testing is one of the best ways to uncover these defects, since you can actually study the progress of the various threads.

If black-box testing is the only possibility, then you can try some of the following tactics:

❑ Vary the time consumption of different threads by giving them differing amounts of input, or running them on hardware that varies in speed.

❑ Run a large number of threads concurrently.

❑ Deliberately deny resources to one or more threads (e.g. temporarily cut a network connection, or make a file unreadable).

Critical races

Defects A critical race is a defect in which one thread or process can sometimes experience a failure because another thread or process interferes with the 'normal' sequence of events. The defect is not that the other thread tries to do something, but that the system allows interference to occur. Critical races are often simply called 'race conditions', although the word 'critical' should be used to distinguish a defect from a race that has no bad consequences.

One type of critical race occurs when two processes or threads normally work together to achieve some outcome; however, if one is sped up or slowed down then the outcome is incorrect. This is illustrated in Figure 10.6, where in the abnormal case the data is read by thread B before it is created by thread A.

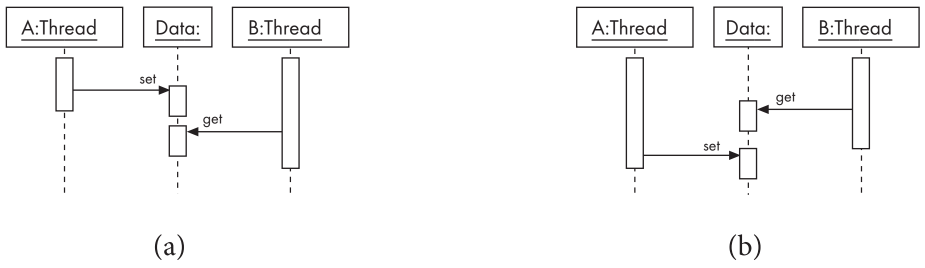


Figure 10.6 **A critical race situation caused by a change in speed of execution**

A second type of critical race occurs when one thread unexpectedly changes data that is being operated on by another thread, resulting in incorrect results. This is illustrated in Figure 10.7.

Designers can prevent critical races by using various mechanisms that allow data items to be locked so that they cannot be accessed by other threads when they are not ready. One widely used locking mechanism is called a *semaphore*.

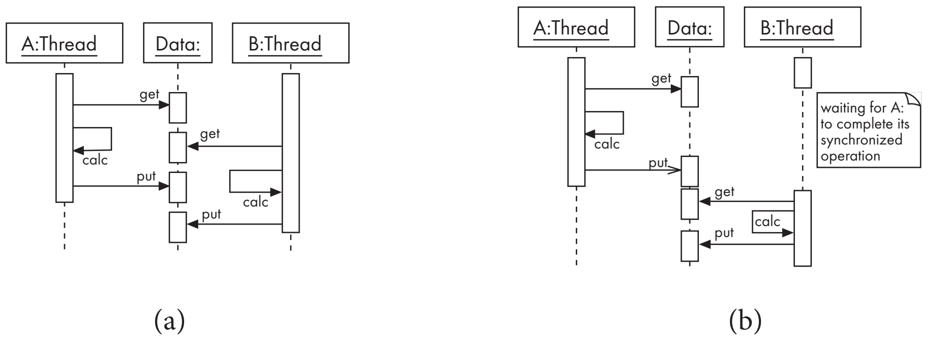


Figure 10.7 **A critical race situation caused by a second thread unexpectedly accessing non-synchronized data**

In Java, the `synchronized` keyword can be used to ensure that no other thread can access an object until the synchronized method terminates.

Unfortunately, overuse of locking can result in deadlocks or livelocks. Locking can also dramatically slow performance since it takes a lot of extra work to manipulate and check the locks.

Testing strategies Testing for critical races is done using the same strategies as testing for deadlocks and livelocks. Once again, inspection is often a better solution.

It is particularly hard to test for critical races using black-box testing alone, since you often do not know the extent of the concurrency going on inside the system, and you cannot always manipulate the various threads to cause race conditions. The timing differences that can give rise to problems can be on the order of milliseconds. Therefore, even if you detect a critical race during testing, you may not be able to reproduce the problem reliably. One possible, although invasive, strategy is to deliberately slow down one of the threads by adding a call to the `sleep` method.

Exercise

E201 Write a short program to generate a critical race situation. Your program will start two threads that sleep for a random amount of time and then do a calculation based on a stored value. They also update the stored value. The result of the calculations should be different depending on which thread sleeps for the longest period of time. This exercise is designed to heighten your awareness of critical races, not to teach you good practice!

10.6 Defects in handling stress and unusual situations

The defects discussed in this section are encountered only when a system is being heavily used, or forced to its limits in some other way. These defects represent a lack of robustness. To test for such defects you must run the system intensively, supply it with a very large amount of input, run many copies of it, run it on a computer or network that is busy running other systems, and run it in atypical environments.

The following subsections give some of the most important categories of stress defects. Many of the kinds of defects discussed here are not explicitly discussed in typical requirements documents. However, they *should* be in the requirements since designers need to arrange explicitly for them to be handled, and testers need to write test cases for them.

Insufficient throughput or response time on minimal configurations

Defect A minimally configured platform is one that barely conforms to the environment specified in the requirements. It has exactly the minimum amount of memory and disk space specified, the slowest CPU, the oldest permitted release of the operating system and other support software, as well as the slowest allowed network connection.

It is a defect if, when the system is tested on a minimal configuration, its throughput or response time fails to meet requirements.

Testing strategy Perform testing using minimally configured platforms. For extra reliability, you could test the system in an environment that has a configuration that is worse than the minimum. In such conditions, it should report that it could not run properly.

Incompatibility with specific configurations of hardware or software

Defect It is extremely common for a system to work properly with certain configurations of hardware, operating systems and external libraries, but fail if it is run using other configurations.

Testing strategy Extensively execute the system with a wide variety of configurations that might be encountered by users. By a 'configuration' we mean a particular combination of hardware, operating system and other software.

Examples A system might fail if a different graphics card is installed, if certain fonts are missing, or a newer or older version of a web browser is installed.

Defects in handling peak loads or missing resources

Defects If a computer becomes short of resources such as memory, disk space or network bandwidth, then programs cannot be expected to continue running normally. However, it is a defect if the system does not gracefully handle such situations.

Other types of shortage of resources include missing files or data, lack of permission to perform certain operations, inability to start new threads, or running out of space in a fixed size data structure. Such problems might be temporary, resulting from a peak in the workload of the software system, of the computer or of the network. On the other hand, the problem might be more permanent, resulting from a failure of some other system.

No matter what the cause, the program being tested should report the problem in such a way that the user will understand. It should then either wait for the problem to be resolved, terminate gracefully or deal with the situation in some other sensible way.

Testing strategies In any of these cases, the tester has to deny the program under test the resources it normally uses, by employing such methods as deleting files, concurrently running other resource-hungry programs, making less memory available, or disconnecting the system from the network. The tester can also run a very large number of copies of the program being tested, all at the same time. To do this effectively, you have to write a special program called a *load generator* that constantly provides input to the system. Load generators are available commercially.

Inappropriate management of resources

Defect A program does not manage resources effectively if it uses certain resources but does not make them available to other programs when it no longer needs them.

Testing strategy Run the program intensively over a long period of time in such a way that it uses many resources, relinquishes them and then uses them again repeatedly.

Examples A program might open many files, but not close them so as to enable other programs to open them. Or a program might abusively consume a very large amount of bandwidth on a network.

A memory leak is a special case of in-appropriate management of resources. Programs written in C, C++ and certain other languages can request memory but not release it when it is no longer needed. If this occurs repeatedly and a program runs for an extended period of time, the computer can eventually begin to perform poorly or even run out of memory. To detect a memory leak, a tester has to run a program for an extended period, and use a utility (see the sidebar) that indicates the amount of memory being used. If the amount of memory steadily increases, this suggests that a memory leak is present.

Although memory leaks are a widespread problem in languages such as C++, you can also get a similar effect in Java if you add objects to an instance of some collection class, and then do not remove them when no longer needed – the garbage collector only removes objects that are not referenced by any other object.

Defects in the process of recovering from a crash

Defects Any system will undergo a sudden failure if its hardware fails, or if its power is turned off. When this occurs, it is a defect if the system is left in an unstable state and hence is unable to recover fully once the power is restored or the hardware

How to determine if a program has a memory leak

The most definitive way to determine whether a program has a memory leak is to run it for a period of time and watch how much memory it uses. If the amount of memory steadily increases, yet the application is not dealing with larger volumes of data, then there is probably a leak.

Most computers allow you to study the memory usage of a computer in detail. On Unix, Linux, and Mac OS X you can type `ps -aux` or `top` in a terminal window to get memory usage data. You can also find out the complete memory map of a process using `pmap` on Solaris systems. Several programs are available to do similar things in the Windows environment.

Besides the above, there are also software libraries that you can link with C or C++ code that replace the normal memory management utilities. These will allow you to obtain a listing of all blocks of allocated memory and make leaks easier to find. An example of such a package is Purify from IBM Rational Software http://www.ibm.com/software/awdtools/purify/.

repaired. It is also a defect if a system does not correctly deal with the crashes of related systems.

Testing strategies An approach for testing for defects in the recovery process is to kill either your program or the programs with which it communicates, at various times during execution. You could also try turning the power off; however, operating systems themselves are often intolerant of doing that.

Examples Unstable states occur when data is half-written, or if a client program does not detect that a server has been restarted.

It is often permissible for a system that recovers from a catastrophic failure to lose the data that was in the process of being entered. However, any earlier transactions should be fully recoverable.

Exercises

E202 Run the SimpleChat server (Phase 3 or higher) and connect a few clients to it. Then make the `passwords.txt` file write-protected so that the server cannot register new clients. Try to connect a new guest client. Does the server react in a sensible way?

E203 If you were designing the following types of software, what stress tests and unusual situations should you subject the system to?

(a) A new web browser.

(b) A flight simulator.

(c) The SimpleChat system.

(d) The GANA navigation system.

10.7 Documentation defects

In the previous sections we have discussed how to detect classes of defects that arise from errors on the part of designers. However, we pointed out at the beginning of the chapter that a failure occurs any time a user has difficulty – and one source of difficulty is using the documentation. You therefore need to 'test' the documentation carefully.

Defect The software has a defect if the user manual, reference manual or online help gives incorrect information or fails to give information relevant to a problem.

Testing strategy Examine all the end-user documentation (in both paper and online forms), making sure it is correct. In particular, make sure it has correct solutions to problems that users might encounter, and correct instructions to help beginners learn how to use the software. Work through the use cases, making sure that each of them is adequately explained to the user. You should use the strategies we discussed in Chapter 7 for evaluating usability: ask users to read the documentation and tell you what they do not understand or do not like.

10.8 Writing formal test cases and test plans

A *test case* is an explicit set of instructions designed to detect a particular class of defect in a software system, by bringing about a failure.

A test case can give rise to many *tests*. Each test is a particular run of the test case on a particular version of the system.

A *test plan* is a document that contains a complete set of test cases for a system, along with other information about the testing process. The test plan is one of the standard types of documentation that should be produced in most software engineering projects. If a project does not have a test plan, then testing will inevitably be done in an ad-hoc manner, leading to poor quality software.

The test plan should be written long before the testing starts. You can start to develop the test plan once you have developed the requirements. A set of use cases can help you design test cases, as discussed in Chapter 4.

Information to include in a formal test case

Each test case should have the following information:

A. **Identification and classification**. Each test case should have a number, and may also be given a descriptive title that indicates its purpose. The system, subsystem or module being tested should also be clearly indicated, with a reference to the related requirements and design documents. Finally, the importance of the test case should be indicated, as discussed in the next subsection.

B. **Instructions**. These tell the tester exactly what to do. The instructions must tell the tester how to put the system into the required initial state and what inputs to provide. The tester should not normally have to refer to the specifications or to any other documentation in order to execute the instructions.

C. **Expected result**. This tells the tester how the system should behave in response to the instructions – i.e. what it should output and what state it should then be in. The tester reports a failure if he or she does not encounter the expected result.

D. **Cleanup (when needed).** This tells the tester how to make the system go 'back to normal' or shut down after the test. For example, if a test case requires that some erroneous data be added to a database, then the cleanup section of the test case would require that the data be deleted (or that the database be reloaded from a backup) so as not to disrupt future tests.

Test cases can be organized into groups or tables, and some of the classification information and instructions can be associated with an entire group, rather than repeated for each test case.

It is becoming increasingly common to completely automate the testing process. Each test case then may become a method that throws an exception if the test fails. The same information described above would still be needed; for example, the test case method would need to report an identification of what failed.

Levels of importance of test cases

It is a good idea to classify test cases according to their importance, or severity level. The most important test cases are executed first, and are designed to detect the most severe classes of defect. A typical scheme for levels of importance is as follows, although each organization may develop its own scheme:

Level 1 First pass critical test cases. These are designed to verify that the system runs and is safe. Any level 1 failure normally means that no further testing is possible.

Level 2 General test cases. These verify that the system performs its day-to-day functions correctly and is therefore a 'success'. A level 2 failure, while important to fix, may still permit testing of other aspects of the system to continue in the meantime.

Level 3 Test cases of lesser importance. For example, these may test 'cosmetic' aspects of the user interface such as the whether a button or menu item is 'grayed out' when it cannot be used. If desired, level 3 test cases can also be used to provide some redundancy – for example, extra testing of additional combinations of input. If there are many failures of level 3 test cases then the system can probably be used, but is lacking in overall quality.

JUnit

JUnit is a Java framework for automated testing. It is becoming widely used among Java developers. See http://junit.org.

Determining test cases by enumerating attributes

It is important that the test cases test every aspect of the requirements. Each detail in the requirements is called an *attribute* – an attribute can be thought of as something that is testable. A good first step when creating a set of test cases is to *enumerate* the attributes.

A simple approach to enumerating attributes is to circle all the important points in the requirements document. However, there are often many attributes that are *implicit*. For example, the requirements may simply state that the look and feel of a program will conform to Microsoft Windows look and feel guidelines. That means that the guidelines contain hundreds of implicit attributes to be tested. The requirements may be silent about all the various versions of software and hardware on which the system should run, and will probably not mention that the program should have no memory leaks. Nevertheless, these should be considered attributes, since a good test plan will include test cases to detect these situations.

Structuring the requirements as a set of use cases (using the format discussed in Chapter 4) can often make the explicit attributes easy to determine. For example, the system's responses to each step performed by the user become attributes, as do the postconditions. Since exceptional situations are listed as separate use cases, these give rise to their own sets of attributes.

Once you have determined the attributes, the next thing to do is to think of all the various tests you need to create to actually test the attributes. You will need to consider equivalence class and boundary tests, as well as tests that try to discover the sets of defects discussed in the last few sections.

Example 10.2 *Create a list of attributes for the following example: a dialog box is to be displayed when the user quits an email program before sending a message he or she is composing. The dialog is to say, 'You have not yet sent your mail, do you want to send it before quitting?' The user has the option of selecting, 'Cancel', 'Send then quit', or 'Quit without sending'. The last is the default.*

Here are some of the testable attributes of this example. Most of these are implicit. These would expand into a much longer list of test cases.

- Are the three buttons lined up at the bottom of the screen?
- Does the dialog box follow the standard appearance of the platform?
- Is the 'Quit without sending' button darkened, indicating that it is the default, and is its action taken when the user hits the return key?
- Can the user (in Windows) press the ALT key plus a letter to invoke any of the options without the mouse?
- Does the dialog box fail to appear when the user quits while not in the middle of composing mail?

- Does the dialog box appear when the user has just started composing mail, but has not yet typed anything? (The requirements need clarifying about this – note that it is not sensible to actually send such a message.)
- Are 'tool tips' displayed when the user passes the mouse over each button?
- When each button is clicked, does it do what it says it will do?
- Does the system behave reasonably when the user selects 'Send then quit', but the system encounters an error when trying to send (the requirements are unclear whether the quitting is abandoned or not)?
- Does the system behave reasonably if the user tries to switch to another program while this dialog is displayed?
- Does the system behave reasonably when the dialog box is brought up and dismissed many times (this helps detect memory leaks)?

Example 10.3 *Discuss the attributes that will need to be tested to test a method with the following specification: method* `mutPrime(x,y)` *takes two* `int` *values and returns* `true` *if* `x` *and* `y` *are mutually prime (have no common divisors except 1) and returns* `false` *otherwise.*

It will be necessary to test situations in which either or both of `x` and `y` are a) equal to one, b) equal to zero, c) negative, d) prime, e) non-prime, but mutually prime with respect to the other, and f) non-mutually-prime.

Exercises

E204 Write four complete test cases for each of the testing situations listed in Exercise E198.

E205 Expand Table 10.3 into a complete test plan.

Test-first development

It has always been recommended to develop test cases well before testing starts. However, there is a trend now towards developing test cases even earlier, and using the process to *drive* design and programming. This is particularly popular when following an agile process.

In *test-first development* you first write an automated set of tests for the next iteration of development. Then you design and write the code such that the tests will pass. The tests are therefore basically a re-stating of the requirements in an automated, testable form. In agile test-first development, the tests may actually be the only way that the requirements are stated in detail.

Testing programs versus proving programs correct

When engineering artifacts are built, engineers try to use analysis techniques to *prove* that they will behave as expected. As engineering has advanced, engineers have learned increasingly sophisticated analysis techniques. In the early days of aviation, for example, it was essential to experiment with and extensively test new aircraft to make sure they flew properly. Today, we understand how to analyze and simulate aerodynamics so well that engineers can be confident that a new aircraft will fly properly when it first takes off. Some testing is still performed in case the analytical and simulation techniques (or control software) have defects, but the time and cost to test a new aircraft is greatly reduced.

Many people yearn for the day when we will be able to use similarly powerful analytic techniques to prove that software adheres to its specifications, and that we will thus be able to reduce the need for software testing. In fact, there are tools that allow you to prove certain assertions about programs, although there are some things that cannot be completely proved, such as whether an arbitrary program will terminate.

However, economics dictate that the extra effort to prove most programs correct is currently not worth the benefits. This is because the analytic techniques for software are labor-intensive, and testing is relatively cheap. This contrasts with testing aircraft, where a failure results in an expensive crash. Formal proof techniques should, however, be used for safety-critical software – such as an aircraft's control software.

A recommended practice in test-first development is to run the test cases prior to developing the code. This helps test the test cases themselves – they should all, of course, fail at this point.

A key advantage of test-first development is that writing the automated tests helps ensure the requirements are properly understood.

10.9 Strategies for testing large systems

In the earlier sections we have discussed the kinds of defects to find, as well as how to write clear and effective test cases. But what strategy should you use to test an entire system that has many subsystems and thousands of test cases? In this section we discuss several approaches to testing such a system.

Integration testing: big bang versus incremental approaches

Testing how the parts of a system or subsystem work together is commonly called *integration testing*. It can be contrasted with *unit testing*, which is testing an individual module or component in isolation.

The simplest approach to integration testing is *big bang* testing. In this approach, you take the entire integrated system and test it all at once. This strategy can be satisfactory when testing a small system; however, when a failure occurs while testing a larger system, it may be hard to tell in which subsystem a defect lies.

A better integration testing strategy in most cases is *incremental testing*. In this approach, you first test each individual subsystem in isolation, and then continue testing as you integrate more and more subsystems.

The big advantage of incremental testing is that when you encounter a failure, you can find the defect more easily. You know it is most likely to be in the subsystem you most recently added.

Incremental testing can be performed *horizontally* or *vertically*, depending on the architecture of the system. Horizontal testing can be used when the system is divided into separate sub-applications, such as 'adding new products' and 'selling products'; you simply test each sub-application in isolation. However, for any sub-application that is complex, it is necessary to divide it up vertically into layers.

There are several strategies for vertical incremental testing: top-down, bottom-up and sandwich. The various strategies are illustrated in Figure 10.8.

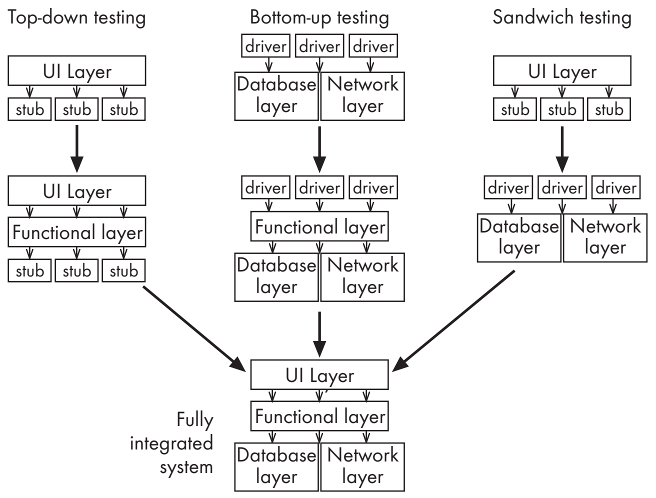


Figure 10.8 **Vertical strategies for incremental integration testing**

Top-down testing

In top-down testing, you start by testing only the user interface, with the underlying functionality simulated by *stubs*. Then you work downwards, integrating lower and lower layers, each time creating stubs for the layers that remain un-integrated. As you integrate each lower layer, you test the system again.

Stubs are pieces of code that have the same interface (i.e. API) as the lower-level layers, but which do not perform any real computations or manipulate any real data. Any call to a stub will typically immediately return with a fixed default value.

If a defect is detected while performing top-down testing, the tester can be reasonably confident that the defect is in the layer that calls the stubs. It could also be in the stubs, but that is less likely since stubs are so simple.

The big drawback to top-down testing is the cost of writing the stubs. However, there are a few automated testing tools that can help generate skeleton code for stubs, relieving the software engineer of some of the tedium of writing them.

Bottom-up testing

To perform bottom-up testing, you start by testing the very lowest levels of the software. This might include a database layer, a network layer, a layer that performs some algorithmic computation, or a set of utilities of some kind.

You need *drivers* to test the lower layers of software. Drivers are simple programs designed specifically for testing; they make calls to the lower layers. Drivers in bottom-up testing have a similar role to stubs in top-down testing, and are time-consuming to write. A driver that fully automates the testing of lower layers is called a *test harness*.

Example 10.4 *Drivers to test OCSF.*

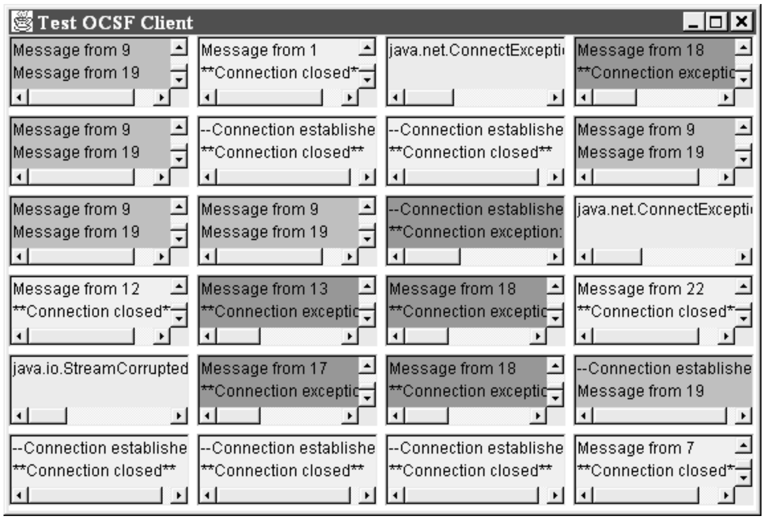
Because of the infinite number of possibilities in the chaining of events, writing a complete set of independent test cases that would exhaustively test the OCSF framework is not possible. The glass-box testing described in Example 10.1 tests several crucial aspects of the framework that might not be thought of in black-box testing. But, as mentioned in that example, it is impossible to cover all paths using glass-box testing. Also, timing and co-ordination failures are more likely to occur under stress situations where several clients are connected; therefore the robustness of the server in these cases has to be assessed.

We have therefore written a test harness consisting of two drivers. This simulates a SimpleChat session where several clients connect to the server and exchange messages. These drivers bypass the user interfaces normally used by end users. They automate the execution of several timing, co-ordination and stress test cases.

On the server side, the `EchoServer` class written using OCSF is put under the control of the first driver. This driver simply calls the service methods, causing the server to transition from one state to another (see Figure 3.3 on page 82). It uses the following sequence of actions in order to take all possible transitions: Listen–Close–Listen–Stop–Close–Listen–Stop. This sequence is infinitely repeated, the server staying in each state for a random period of time. A complete log of all server actions, including any exceptions thrown, is displayed for the tester.

The role of the second driver is to randomly generate and control a set of clients. A `Frame` containing information about each created client is displayed on a grid (Figure 10.9). At random intervals, the client driver takes one of the following actions:

- It creates a new client and connects it to the server. Information about this client is put in a panel of the grid that is not already running a client.



```
Test OCSF Client
Message from 9           Message from 1           java.net.ConnectExcepti  Message from 18
Message from 19          **Connection closed*                              **Connection exceptic

Message from 9           --Connection establishe  --Connection establishe  Message from 9
Message from 19          **Connection closed**    **Connection closed**    Message from 19

Message from 9           Message from 9           --Connection establishe  java.net.ConnectExcepti
Message from 19          Message from 19          **Connection exception:

Message from 12          Message from 13          Message from 18          Message from 22
**Connection closed*     **Connection exceptic    **Connection exceptic    **Connection closed*

java.io.StreamCorrupted  Message from 17          Message from 18          --Connection establishe
                         **Connection exceptic    **Connection exceptic    Message from 19

--Connection establishe  --Connection establishe  --Connection establishe  Message from 7
**Connection closed**    **Connection closed**    **Connection closed**    **Connection closed*
```

Figure 10.9 **Test driver for the OCSF client**

- It randomly selects a client and makes it send a message (the simple string 'Message from X', where X is the grid number).
- It disconnects a random client.

Figure 10.9 is a snapshot of the client driver window. A color convention has been used to help identify each client's state (although the colors are shown here using shades of gray).

At the time the snapshot was taken, six clients were simultaneously connected – these are shown in medium-gray. The panels in dark gray correspond to clients that have been killed by the server. The panels in light gray show other situations such as when the client closed itself, or unsuccessfully attempted to connect. Any exceptions thrown by the OCSF `AbstractClient` class while trying to connect, disconnect or send messages are also displayed in the corresponding panel.

To perform a more severe stress test, the client side of this testing system should be run simultaneously on several machines. The two drivers can be found on the book's web site (www.lloseng.com). You should experiment with these to learn about how to write and use test drivers.

Sandwich testing

Sandwich testing is a hybrid between bottom-up and top-down testing – it is sometimes therefore called *mixed testing*. A typical approach to sandwich testing is to test the user interface in isolation, using stubs, and also to test the very lowest-level functions, using drivers. Then, when the complete system is integrated, only the middle layer remains on which to perform the final set of tests.

For many systems, sandwich testing will be the most effective form of integration testing.

The test–fix–test cycle and regression testing

When a failure occurs during testing or after deployment, most organizations follow a carefully planned process. Each failure report is entered into a failure tracking system. It is then is screened and assigned a priority. It is often too expensive to fix every defect, therefore low-priority failures might be put on a *known bugs list* that is included with the software's *release notes*. Some failure reports might be merged if they appear to result from the same defects.

When a decision is made to resolve a failure, somebody is assigned to investigate it, track down the defect or defects that caused it, and fix those defects. Finally a new version of the system is created, ready to be tested again.

Unfortunately, there is a high probability that efforts to remove defects will actually add new ones – either because the maintainer tries to fix problems without fully understanding the ramifications of the changes, or because he or she makes ordinary human errors. This phenomenon is known as the *ripple effect*, because new defects caused by erroneous fixes of other defects tend to spread through a system like a ripple. The system regresses into a more and more failure-prone state.

You can minimize the ripple effect by applying a very disciplined approach to the process of fixing defects. For example, you can perform *impact analysis* to carefully explore and document all possible effects of a change. You can then have others inspect each change before it is made. However, experience shows that the ripple effect will still persist if you do not have an appropriate testing strategy.

After making any change, you must therefore not only re-run the test case that led to the detection of the defect, but you must also re-test the rest of the system's functionality. This latter process is called *regression testing*. It is normally far too expensive to re-run every single test case whenever a change is made to software, so regression testing involves running a subset of the original test cases. The regression tests are carefully selected to cover as much of the system as possible.

Deciding when to stop testing

You might imagine that you should go on re-testing software until all the test cases have passed. Unfortunately, this is not a practical approach. For the reasons discussed in the last subsection, it is too expensive, and perhaps futile, to try to remove every last bug from most systems. Perfection will normally remain forever elusive.

However, it is also a poor strategy to stop testing merely because you have run out of time or money. This will result in a poor-quality system.

You should, instead, establish a set of criteria like the following to decide when testing should be considered complete:

- All of the level 1 test cases must have been successfully executed.
- Certain predefined percentages of level 2 and level 3 test cases must have been executed successfully. Suitable levels will vary from system to system. In a non-critical system used by a small number of people, test case pass targets of 95% for level 2 and 75% for level 3 might be appropriate. In a more critical system, or one used by a large number of people, then test case pass targets of 99% for level 2 and 90% for level 3 might be set. Ultimately the targets will depend on the cost of failures versus the cost of continued testing and fixing of defects.
- The targets must have been achieved and then maintained for at least two cycles of 'builds'. A *build* involves compiling and integrating all the components of the software, incorporating any changes since the last build. Many organizations do this on a daily or weekly basis; they then perform regression testing on the build. Failure rates can fluctuate from build to build as different sets of regression tests are run or new defects are introduced. It might be by chance that the targets are reached one day but are not met the next day.

The law of conservation of bugs

The 'law of conservation of bugs' states that the number of bugs remaining in a large system is proportional to the number of bugs already fixed. In other words, a defect-ridden system will always tend to remain a defect-ridden system. This is because a poorly designed system will not only have more defects to start with, but will also be harder to fix correctly and hence will suffer more strongly from the ripple effect.

The roles of people involved in testing

Testing is an intellectual exercise that is just as challenging and creative as design. All software engineers should develop their testing skills, although some have a particular talent for testing and may specialize in it.

In a software development organization, the first pass of unit and integration testing is often called *developer testing*. This is preliminary testing performed by the software developers who do the design and programming. Organizations should then employ an *independent testing group* that runs the complete set of test cases, seeking defects left by the developers.

An advantage of having a separate group perform testing is that they do not have a vested interest in seeing as many test cases pass as possible. Although it would be unprofessional for developers to claim that a test case has passed when it has not, they may nevertheless be less hard on themselves than an independent group. Also, an independent group that specializes in testing will develop specific expertise in how to do good testing, and how to use testing tools.

Testing performed by users and clients: alpha, beta and acceptance

So far, our discussion of testing has assumed that it is performed by software engineers in the development organization. However, the testing process also normally involves users. We have already seen in Chapter 7 that users should be heavily involved in testing and evaluating prototypes. In this section we will discuss the involvement of users in testing versions of the system that are almost ready to be put into production. Such testing should occur once the developers believe that the software has reached a sufficient level of quality (it is approaching the quality targets specified in the non-functional requirements).

Alpha testing is testing performed by users and clients, under the supervision of the software development team. The development team normally invites some users to work with the software and to watch for problems that they encounter.

Beta testing is testing performed by the user or client in their normal work environment. It can be initiated either at the request of the software developers, or at the request of users who want to try the system out. Beta testers are selected from the potential user population and given a pre-release version of the software. They know that the software will contain more defects than the final version, but they have the benefit of using the features of the software before others have access to them. Beta testers are responsible for reporting problems when they discover them.

The advantages of alpha and beta testing are the following. You typically are able to have a much larger volume of testing performed, and the users use the software in the same manner that they will use it when it is formally released. Very often, the users do things that the developers of the test plan never anticipated; hence users encounter failures that no test case was written to detect. In the worst case, discovery of these defects during alpha and beta testing may require major changes to the software – but it is better that the defects be found at this stage, rather than following the software's general release.

Some organizations rely very heavily on beta testing, and will release low-quality software to the general population in what is called an *open beta release*. If the software is in heavy demand, this can result in the effective discovery of problems. It may make economic sense to do this from the perspective of the developers, but it is not good engineering practice. It results in wasted time on the part of the users, since many people will discover the same problems. It can also damage the reputation of the developers if many failures occur.

Acceptance testing, like alpha and beta testing, is performed by users and customers. However, the customers do it on their own initiative – to decide whether software is of sufficient quality to purchase. Many large organizations also perform acceptance testing before they will pay a developer they have contracted to develop custom software for them. Other organizations use acceptance testing in order to choose between several competing generic products.

Exercise

E206 Discuss how integration testing could be performed in the SimpleChat system. What stubs or drivers could be written to permit separate testing of individual layers?

Object-oriented testing

All of the testing strategies we have discussed work just as well for object-oriented software as for non-OO software. Nevertheless, there is a special set of techniques called object-oriented testing. These techniques focus on the properties of object-oriented programs. Examples include:

- Writing methods within each class to act as drivers that will test the methods of that class. We mentioned this in the discussion of Design Principle 10 in the last chapter.
- Testing to ensure that the behavior of a subclass is consistent with the behavior of its superclass. This can be done by running, in each subclass, the driver methods of the superclass.

10.10 Inspections

An inspection is an activity in which one or more people systematically examine source code or documentation, looking for defects. Normally, inspection involves a meeting, although participants can also inspect alone at their desks.

Roles on inspection teams

Typically an inspection team will consist of software engineers, who fill the following roles:

- The *author*.
- A *moderator*. This person calls and runs the meeting and makes sure that the general principles of inspection are adhered to. These principles are discussed in the next subsection.
- A *secretary*. He or she is responsible for recording the defects when they are found. This is not an office administrative assistant, but rather a software engineer just like the other inspectors. It takes a thorough knowledge of software engineering to understand and record defects.
- *Paraphrasers*. These people will step through the document explaining it in their own words. We will explain the paraphrasing process in more detail below.

In a small inspection team, an individual can perform more than one of the latter three roles.

Principles of inspecting

Some general principles of inspecting are as follows:

- **Inspect the most important documents of each type**. Inspection should be performed on code, design documents, test plans and requirements. It is not always necessary or economical to inspect every single piece of code or every document, but the most important ones should certainly be inspected. Quality will be higher if more are inspected, but costs will also be higher. When inspecting code, all aspects of code should be considered, including the comments.

- **Choose an effective and efficient inspection team**. Two or more people should participate in an inspection, since more pairs of eyes are better than one. However it is uneconomical to have a very large number of people involved; between two and five people (including the author) is probably a good range; the exact number can depend on the importance of the item being inspected. The inspection team should include experienced software engineers, who are more likely to uncover defects.
- **Require that participants prepare for inspections**. The inspection team should study the code or other documents prior to the meeting and come prepared with a list of defects. If a participant has not prepared, then he or she will spend most of his or her mental energy at the inspection meeting trying to understand the document, rather than helping to find defects.
- **Only inspect documents that are ready**. Inspections should be held once specifications, design documents or code are believed by their authors to be final. However, when preparing for an inspection meeting, the inspectors might immediately realize that the document represents very poor design, that its layout is messy and hard to read, or that it needs some other major change; in these circumstances it is probably best to call off the meeting. The author should be asked to make changes before a new attempt is made at inspection. Attempting to inspect a very poor document will result in defects being missed – and the document will have to be re-inspected anyway.
- **Have 'finding defects' as the only goal**. It is best not to mix inspection meetings with meetings in which other activities are involved. For example, if you are inspecting code, you should not at the same time be teaching newcomers how the system works. Having the secondary goal would detract from the primary goal, which is to find defects. You should also not mix inspections with meetings whose primary role is to make decisions. For example, requirements and design reviews are often arranged so that the teams can debate alternatives – doing this should not be confused with inspecting.
- **Avoid discussing how to fix defects**. Fixing defects is a design issue that can be left to the author. There is no need to consume the time of the entire team during the meeting. If the author needs help with a particular defect, he or she can always ask for it later.
- **Avoid discussing style issues**. It is important that inspections do not become bogged down debating such issues as the naming convention used for variables, or whether the '{' and '}' should always line up with each other vertically or not. Issues such as these may be important, but should be discussed separately, in the context of the project as a whole. A good rule of thumb is the following: an issue should be raised in an inspection meeting only if it may represent a defect. Note, however, that bad style is often a maintainability defect.

- **Do not rush the inspection process**. The schedule should be arranged so that there is time both to complete the necessary inspections and to fix the defects found. A common problem is to be rushing to meet deadlines and hence to skip inspection or to fail to properly fix the defects found. A good speed to inspect is about 200 lines of code per hour (including comments), or ten pages of text per hour.
- **Avoid making participants tired**. Since inspection is very mentally taxing, it is best not to inspect for more than two hours at a time, or for more than four hours a day.
- **Keep and use logs of inspections**. Logs of inspections should be kept. These will list what was inspected and the defects found. Follow-up should be done after every inspection session to ensure that all the defects have, in fact, been resolved. You can also use the logs to track the quality of the design process – ideally as a team gets better at design, the number of defects found per line of code will decrease.
- **Re-inspect when changes are made**. You should re-inspect any document or code that is changed more than 20% for any reason; for example, as a result of adding new features, or fixing problems arising from testing or inspection.

A peer-review process: managers are normally not involved

It is common practice to exclude managers from inspection activities, that is, to have inspections performed only by the peers of the author. The rationale for this is that it may allow the participants to express their criticisms more openly, not fearing repercussions from the manager.

In small organizations, however, there may not be enough people to perform inspections if the manager is not involved. Also, if the team members are professional in their work practices, then management involvement may have no negative consequences. It may also be beneficial for managers to participate since it can help them stay involved in the team's work and hence make better management decisions.

Whether or not a manager participates in inspections, the members of an inspection team should feel they are all working together to create a better document. The author should be made to feel comfortable, and not 'blamed' when defects are found.

Conducting an inspection meeting

The following is a suggested approach to running an inspection meeting:

1. The moderator calls the meeting and distributes the documents to be inspected.
2. As mentioned above, the participants prepare for the meeting in advance.
3. At the start of the meeting, the moderator explains the procedures and verifies that everybody has prepared.

4. At the meeting, paraphrasers take turns explaining the contents of the document or code, without reading it verbatim. Paraphrasing forces everybody to *think* about what they are reading; merely reading verbatim can become a mindless exercise that can be done while almost asleep. Requiring that the paraphraser not be the author ensures that the paraphraser says what he or she *sees*, rather than what the author *intended* to say. Differences between these two views will therefore become clear and thus highlight defects. The author, in particular, may notice the paraphraser say something different from what he or she thought was written.
5. Everybody speaks up when they notice a defect or when the paraphrasing reaches a defect that they had found while preparing. There may be a very brief discussion to ensure that it really is a defect, and then the secretary makes a record of it. After this, paraphrasing continues.

   As an alternative to paraphrasing, the inspectors can simply walk through the lists of defects found while each member was preparing. Some organizations even dispense with meetings entirely, and have the moderator gather lists of defects produced by individual inspectors working alone.

Inspecting compared to testing

Inspection is a quality assurance activity that is complementary to testing:

- Both testing and inspection rely on different aspects of human intelligence. To test effectively, testers have to develop a good set of test cases. They need an aptitude for thinking of what could go wrong without actually studying the software. In an inspection meeting, on the other hand, the participants have to uncover defects by understanding what would happen if the system were run – they have to mentally *execute* the software. Both activities require attention to detail, and people can make mistakes in both activities. However, the chances of mistakes are reduced if both activities are performed, since chances of the same defect slipping by both activities are greatly reduced.
- Testing can find defects whose consequences are obvious but which are buried in complex code, and thus will be hard to detect when inspecting. Inspecting, on the other hand, can more easily find defects whose consequences might be subtle; hence the tester might not have thought to test for them. They become clear when reading the source code or design documents.
- Inspecting can find defects that relate to maintainability or efficiency that are not readily detectable when testing. For example, a tester may observe that the results of a computation take one second. He or she may not be able to tell, without inspecting the code, that an inefficient algorithm is being used, and that the computation time could be dramatically reduced by changing the code. Similarly, very messy code might work adequately when tested; but the maintainability problems would only be noted during an inspection.

Testing or inspecting, which comes first?

It is important to inspect software *before* extensively testing it. This considerably speeds up the overall verification process. The reason for this is that inspecting allows you to get rid of many defects quickly. Not only would testing take a lot longer to find the same number of defects, but then every one of them must be individually investigated and fixed. Also, if you test before inspecting, and the inspectors then recommend that redesign is needed, the testing work has been wasted.

There is a growing consensus that it is most efficient to inspect software before any testing is done – even before developer testing.

Exercise

E207 Working in groups of four or five, conduct inspections of some code and documents you have produced while working on the exercises in this book. Each member of the group should have their turn as author, presenting the group with one item of about 100 lines of code or five pages of text. The members of the group should exchange material several days before the inspection meeting. Group members should prepare for the meeting by making sure they understand what is to be inspected. The meetings themselves, which should last for about half an hour for each author, should be conducted as discussed above. The role of moderator can be rotated among the various group members. The deliverables from this exercise are the inspection logs.

10.11 Quality assurance in general

In this chapter, so far, we have looked at two important and inter-related approaches to ensuring the quality of software: testing and inspecting. Both of these are disciplined processes that software engineers and their organizations should implement.

Quality assurance should, however, be an activity that pervades all aspects of software development. In Chapter 4, for example, we discussed how you can review requirements to ensure that they are valid.

In the following subsections we will look at some other things that can be done to help increase software quality.

Root cause analysis

The objective of root cause analysis is to determine why a software engineer made the error that resulted in a defect occurring. It might have been simple human error – it will never be possible to completely prevent all such errors. However, often there are one or more root causes that you can correct in order to reduce the number of errors made.

The following are examples of root causes.

- Lack of training and experience.
- Schedules that are too tight.
- Building on poor designs or reusable technology.

There are other possible root causes, including unanticipated design complexity, failure to adhere to design or management principles, lack of discipline in the process, failure to properly review requirements and designs, and failure to keep track of the details of the project. These, however, are often not root causes, but rather are due to lack of training and experience. Even too-tight schedules and building on poor designs are signs of project management failure, and hence have lack of training or experience as the ultimate root cause.

Measuring quality as a means to strive for continuous improvement

It is essential to measure quality of both the product and the process. This allows you to plot the quality over a period of time and to determine whether it is improving or not. Measuring quality allows you to set objectives for quality improvement and to motivate yourself and others to meet those objectives.

Two historic software failures that could have been prevented

Better quality assurance could have prevented a number of high-profile disasters, such as the following. To learn more, see the 'For more information' section.

- **Therac-25**. The Therac-25 was a machine designed for radiation therapy in cancer patients. Unfortunately, a software defect resulted in it delivering far more radiation than intended under certain circumstances, with several resulting deaths.
- **Ariane-5**. In 1996 the first flight of the $500 million French Ariane-5 Launcher failed due to a software defect. The failure occurred due to an attempt to fit an integer greater than 2^{15} into a 16-bit integer. That, in turn, was caused by reusing an older piece of code without verifying that the code would work properly in the new release

Entire books have been written about metrics; the following are some of the things you can measure regarding the quality of a software product, and indirectly of the quality of the process.

- The number of failures encountered by users.
- The number of failures found when testing a product.
- The number of defects found when inspecting a product.
- The percentage of code that is reused (more is better, but don't count clones).
- The number of questions posed by users to the help-desk (as a measure of usability and the quality of documentation).

You can compare how these change over time. You can also compare the quality of two systems, after normalizing by dividing by the size of the system.

Post-mortem analysis

Post-mortem analysis involves looking back at a project after it is complete, or after a release. You look at the design and the development process and identify those aspects, which, with benefit of hindsight, you could have done better. You then make plans to do better next time.

Process standards

Organizations can collectively improve software quality by following the guidelines found in several different standards. The following is a list of standardized approaches; the first three of these were developed at the Software Engineering Institute of Carnegie Mellon University.

- **The Personal Software Process** (PSP™). This defines a disciplined approach that an individual software developer can use to improve the quality and efficiency of his or her personal work. Two of the key tenets of the psp are personally inspecting your own work, and measuring the progress you make towards improving the quality of your work.
- **The Team Software Process** (TSP™). This describes how teams of software engineers can work together effectively.
- **The Software Capability Maturity Model** (CMM™). This contains five levels. Organizations start at level 1, and as their processes become better they can move up towards level 5.
- **ISO 9000-3**. This is an international standard that lists a large number of things an organization should do to improve its overall software process.

These standards include testing and inspecting among the things to do; however, they include a great many other best practices as well. In this book we do not have the space to discuss these standards in detail, but references to these and other standards are found in the 'For more information' section at the end of the chapter.

Spectacular software successes

In several places in this book we have pointed to software projects or systems that have failed and resulted in disasters. However, the quality and reliability of many systems is remarkable. The software that runs the space shuttle is a key example: although shuttles have suffered tragic accidents, the software that controls all aspects of space flight has worked with flying colors, despite its incredible complexity. The shuttle software team was one of the first to achieve a CMM level 5 rating. Highly reliable and complex software also controls airplanes, railway systems, telecommunications systems and banking systems. Failures of hardware tend to be at least as common as software failures in these types of systems.

Any practicing software engineer should have a good understanding of these standards and their suitability for his or her organization.

10.12 Test cases for phase 2 of the SimpleChat instant messaging system

On the book's web site (http://www.lloseng.com) you will find a complete set of test cases to test Phases 1 and 2 of SimpleChat. The following is a selection of those test cases.

General setup for test cases in the 2000 series

System: SimpleChat/ocsf Phase: 2
Instructions:

1. Install Java, minimum release 1.2.0, on Windows 95, 98 or ME.
2. Install Java, minimum release 1.2.0, on Windows NT or 2000.
3. Install Java, minimum release 1.2.0, on a Solaris system.
4. Install the SimpleChat – Phase 2 on each of the above platforms.

Test Case 2001
System: SimpleChat Phase: 2
Server startup check with default arguments
Severity: 1
Instructions:

1. At the console, enter: `java EchoServer`.

Expected result:

1. The server reports that it is listening for clients by displaying the following message:
   `Server listening for clients on port 5555`
2. The server console waits for user input.

Cleanup:

1. Hit ctrl+c to kill the server.

Test Case 2002
System: SimpleChat Phase: 2
Client startup check without a login
Severity: 1
Instructions:

1. At the console, enter: `java ClientConsole`.

Expected result:

1. The client reports it cannot connect without a login by displaying:
   `ERROR - No login ID specified. Connection aborted.`
2. The client terminates.

Cleanup: (if client is still active)

1. Hit ctrl+c to kill the client.

Test Case 2003
System: SimpleChat Phase: 2
Client startup check with a login and without a server
Severity: 1
Instructions:

1. At the console, enter: `java ClientConsole <loginID>` where `<loginID>` is the name you wish to be identified by.

Expected result:

1. The client reports it cannot connect to a server by displaying:
   `Cannot open connection. Awaiting command.`
2. The client waits for user input

Cleanup: (if client is still active)

1. Hit ctrl+c to kill the client.

Test Case 2007
System: SimpleChat Phase: 2
Server termination command check
Severity: 2
Instructions:

1. Start a server (Test Case 2001 instruction 1) using default arguments.
2. Type `#quit` into the server's console.

Expected result:

1. The server quits.

Cleanup: (If the server is still active):

1. Hit ctrl+c to kill the server.

Test Case 2013
System: SimpleChat Phase: 2
Client host and port setup commands check

Severity: 2
Instructions:

1. Start a client without a server (Test Case 2003).
2. At the client's console, type `#sethost <newhost>` where `<newhost>` is the name of a computer on the network.
3. At the client's console, type `#setport 1234`.

Expected result:

1. The client displays
   ```
   Host set to: <newhost>
   Port set to: 1234.
   ```

Cleanup:

1. Type `#quit` to kill the client.

Test Case 2016
System: SimpleChat Phase: 2
Multiple remote client disconnections and re-connections
Severity: 2
Instructions:

1. Start a server (Test Case 2001, instruction 1).
2. On different computers, start clients (1 or 2 per computer) and connect them to the server.
3. Exchange data among all the clients and the server.
4. Close the server using the `#close` command.
5. Change the server's listening port by using the `#setport <newport>` command.
6. Restart the server using the `#start` command.
7. Change the ports of the clients, using `#setport`, to correspond to the new port of the server.
8. Reconnect the clients to the server by using the `#login <loginID>` command.

Expected results:

1. The first set of connections occur normally.
2. When the server is closed, all clients are disconnected.
3. The server displays the following message when the `#setport` command is used:
   ```
   port set to: <newport>.
   ```

4. The server restarts and displays:
   `Server listening for connections on port <newport>.`
5. The clients change port as in Test Case 2013.
6. The clients reconnect normally.

Cleanup:

1. Type `#quit` to kill the clients.
2. Type `#quit` to kill the server.

Test Case 2017
System: SimpleChat Phase: 2
Client changing hosts
Severity: 2
Instructions:

1. On two different computers, start servers on the default port.
2. On a third computer, start a client and connect it to one of the two servers.
3. Logoff from that server using the `#logoff` command.
4. Change the host name by using the `#sethost <otherhost>` where `<otherhost>` is the name of the computer running the other server.
5. Log the client on again using the `#login <loginID>` command.

Expected results:

1. The two servers start up normally.
2. The client connects to the first server normally.
3. When the client disconnects it displays
   `Connection closed.`
4. When the client disconnects, the server displays:
   `<loginID> has disconnected.`
5. The client changes host as in Test Case 2013.
6. The client reconnects normally as in Test Case 2016.

Cleanup (Unless proceeding to Test Case 2018):

1. Type `#quit` to kill the servers.
2. Type `#quit` to kill the client.

Test Case 2019
System: SimpleChat Phase: 2
Different platform tests
Severity: 3
Instructions:

1. Repeat test cases 2001 to 2018 on Windows 95, 98, NT or 2000, and Solaris.

Expected results:

1. The same as before.

10.13 Difficulties and risks in quality assurance

- **It is very easy to forget to test some aspects of a software system**. In some projects, the team members feel that 'running the code a few times' is sufficient. Even when a formal test plan is in place, testers often forget to do such things as stress tests and documentation tests. Forgetting certain types of tests diminishes the system's quality.
  Resolution. Use all the testing strategies described in this chapter.

- **There is a conflict between achieving adequate quality levels, and 'getting the product out of the door'**. Although almost everybody recognizes the importance of quality, those activities that can ensure quality are often sacrificed in order to meet deadlines. In particular, companies often judge developers or project managers purely on when they deliver product, not on its quality level. Quality assurance activities are often seen as an overhead expense to be reduced. The result is poor-quality software.
  Resolution. Create a separate department to oversee quality assurance. Publish statistics about quality (within the organization) so that people will be motivated to increase their quality. Build adequate time for all quality assurance activities into the schedule. Consider quality assurance to be an integral and ongoing part of development. Publish cost–benefit analyses to demonstrate to everyone that deadlines should not override a need for quality.

- **People have different abilities and knowledge when it comes to quality**. Some people have a natural tendency to pay attention to detail, whereas other people are better at seeing the 'big picture'. When it comes to testing, the former people might help uncover more problems. Many people are not trained adequately in various aspects of quality, particularly usability and maintainability.
  Resolution. Give people tasks that fit their natural personalities. Train people in testing and inspecting techniques. Give people feedback about their performance in terms of producing quality software so that they have

something measurable to improve. Have developers and maintainers work for several months on a testing team; this will heighten their awareness of quality problems they should avoid when they return to designing software.

10.14 Summary

In this chapter we have given you an overview of several important strategies for efficiently verifying that a system is of sufficient quality, as well as for supporting quality development more generally.

We discussed key terminology, in particular that human errors cause defects, and defects cause failures.

We then looked at strategies for testing. Black-box testing allows you only to control a system's inputs and observe its outputs, whereas glass-box (or white-box) testing allows you to examine the system's internals. Big bang testing involves testing an integrated system all at once, whereas incremental approaches to integration testing involve first testing individual subsystems, and then testing repeatedly as subsystems are put together to create the complete system.

Testing a selected member of each equivalence class, as well as equivalence class boundaries, allows you to detect different defects without exhaustively trying all possible inputs. However, determining suitable equivalence classes is hard. In particular, it is necessary to apply knowledge of the most common types of defects and program design strategies. Typical types of defects to seek include algorithmic defects (such as incorrect logical conditions, or not handling null conditions), defects in which the system cannot handle stress, and defects resulting from failure to adhere to standards.

We discussed how to systematically write both test cases and complete test plans, and to consider test-first development where an automated test plan is used to drive development. We also pointed out that it is important to plan in advance when to stop testing. In addition, we looked at regression testing, which is testing with a subset of tests when the system is changed, alpha testing, which is testing by users under the supervision of developers, and beta testing, which is testing by users in their own environment.

Inspections are another way of verifying software that should be used in conjunction with, and prior to, testing. We suggested a strategy for inspection that involves a team. Team members include the author, a paraphraser and a moderator. The paraphraser proceeds through the code or other documents, explaining the material in his or her own words. The Personal Software Process is a disciplined approach to individual software development that, among other things, emphasizes careful inspection of your own work.

Finally, we looked at quality assurance as a whole, including the notion of continuous improvement.

10.15 For more information

Software quality assurance in general

- Hotlist about software quality assurance and process improvement: http://www.tantara.ab.ca/info.htm
- G. Gordon Schulmeyer (Ed), *Handbook of Software Quality Assurance*, Prentice Hall, 1999

Important software failures

- Peter G. Neumann's Risks Digest. http://catless.ncl.ac.uk/Risks/. Contains numerous reports of software failures. Software engineers should study these failures to ensure they do not re-create them
- N. G. Leveson and C. S. Turner, 'An Investigation of the Therac-25 Accidents', *IEEE Computer*, Vol. 26, No. 7, July 1993, pp. 18–41
- J-M. Jézéquel and B. Meyer, 'Design by Contract: The Lessons of Ariane', *IEEE Computer*, Vol. 30, No. 2, January 1997, pp. 129–130

Software testing

- T. Koomen, M. Pol, H. W. Broeders and H. Voorthuyzen, *Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing*, Addison-Wesley, 1999
- W. E. Lewis, *Software Testing and Continuous Quality Improvement*, CRC Press, 2000
- C. Kaner, H. Q. Nguyen and J. Falk, *Testing Computer Software*, 2nd edition, Wiley, 1999
- Software testing online resources by Roland Untch at Middle Tennessee State University: http://www.mtsu.edu/~storm
- Newsgroup comp.software.testing and its FAQ: http://www.faqs.org/faqs/software-eng/testing-faq
- The DMOZ Open Directory on software testing: http://dmoz.org/Computers/Programming/Software_Testing
- Aptest's software testing links; a particularly strong list of testing tools: http://www.aptest.com/resources.html

Software inspection

- D. A. Wheeler, B. Brykczynski, and R. N. Meeson, Jr. (Eds.), *Software Inspection An Industry Best Practice*, IEEE CS Press, 1996
- T. Gilb, D. Graham and S. Finzi, *Software Inspection*, Addison-Wesley, 1993

Books about process standards

- P. Jalote, *CMM in Practice: Processes for Executing Software Projects at Infosys*, (SEI Series in Software Engineering), Addison-Wesley, 1999
- W. Humphrey, *Introduction to the Personal Software Process*, (SEI Series in Software Engineering), Addison-Wesley, 1996
- W. Humphrey, M. Lovelace and R. Hoppes, *Introduction to the Team Software Process*, (SEI Series in Software Engineering), Addison-Wesley, 1999
- D. Hoyle, *ISO 9000 Quality Systems Handbook*, 4th edition, Butterworth-Heinemann, 2000

Standards

The following are some of the IEEE and British standards covering quality assurance and testing. As mentioned in Chapter 4, access to these requires a subscription. See: http://www.standards.ieee.org/software/index.html for the IEEE standards, and bsonline.techindex.co.uk for the British standards.

- ISO 9126, *Software Product Quality Characteristics*
- IEEE Standard 730, *Software Quality Assurance Plans*
- IEEE Standard 829, *Software Test Documentation*
- IEEE Standard 1012, *Software Verification and Validation*
- IEEE Standard 1028, *Software Reviews*
- British Standard 7925, *Software Testing*

Project exercises

E208 Perform a test case inspection of the test cases for Phase 2 of the SimpleChat system. The purpose of this inspection will be to detect defects in the test cases, not in SimpleChat itself. A legitimate part of an inspection is to uncover important test cases that are missing from the set, therefore make sure the test plan has tests that detect all the types of defects discussed in this chapter.

E209 Create a set of test cases for Phase 3 of the SimpleChat system. This is the version of SimpleChat you created at the end of Chapter 4. Once you have created your test plan, use it to test your work, recording any failures when they occur. (Suggestion: test the work of another group while they test your work.)

E210 Perform a root cause analysis of the failures you encountered in the last exercise.

E211 Stress test the SimpleChat program by having a very large number of people connect clients to the same server and issue many different commands. Try to determine the capacity of the server (or the network) before it slows down to an unacceptable level of performance. Write a summary of your experiences.

E212 Create a complete set of test cases for the Small Hotel Reservation System.