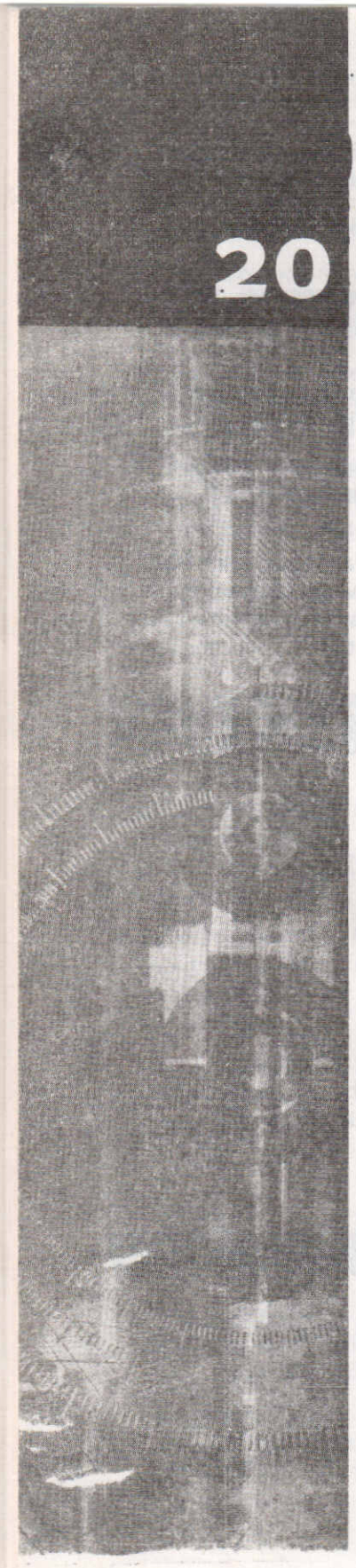


20 Software testing

Objectives

The objective of this chapter is to introduce techniques that may be used to test programs to discover program faults. When you have read the chapter, you will:

- understand a number of testing techniques that are used to discover program faults;
- have been introduced to guidelines that support the testing of component interfaces;
- understand specific approaches to component testing and integration testing for object-oriented systems;
- understand the principles of operation of CASE tool support for testing

Contents

In Chapter 3, I discussed a general testing process that started with the testing of individual program units such as functions or objects. These were then integrated into sub-systems and systems and the interactions of these units were tested. Finally, after completion of the system, the customer may carry out a series of acceptance tests to check that the system performs as specified.

A more abstract view of this testing process is shown in Figure 20.1. The component testing stage is concerned with testing the functioning of clearly identifiable components. These may be functions or groups of methods collected together into a module or objects. During integration testing, these components are integrated to form sub-systems or the complete system. At this stage, testing should focus on interactions between the components and on the functionality and performance of the system as a whole. Inevitably, however, defects in components that have been missed during earlier testing are discovered during integration testing.

As part of the V & V planning process, managers have to make decisions on who should be responsible for the different stages of testing. For most systems, programmers take responsibility for testing their own code (modules or objects). Once this is completed, the work is handed over to an integration team who integrate the modules from different developers, build the software and test the system as a whole. However, for critical systems, a more formal process may be used where independent testers are responsible for all stages of the testing process. Tests are developed separately and detailed records are maintained.

When testing critical systems, a detailed specification of each software component is used by the independent team to derive the tests for the system. However, in most other cases, testing is a more intuitive process as there is not time to write detailed specifications of every part of a software system. Rather, the interfaces of the major system components are specified and individual programmers and programming teams take the responsibility for the design, development and testing of these components. Testing is usually based on an intuitive understanding of how these components should operate.

Integration testing, however, must be based on a written system specification. This can be a detailed system requirements specification as discussed in Chapter 5 or it can be a user-oriented specification of the features that should be implemented in the system. A separate team is always responsible for integration testing. As discussed in Chapter 3, they use the user and system requirements documents to develop detailed integration testing plans (see Figure 3.12).

Books on testing, such as those by Beizer (1990), Kit (1995) and Perry (1995), are based on practical experience where systems are developed using a functional model. They do not specifically address the testing of object-oriented systems. From

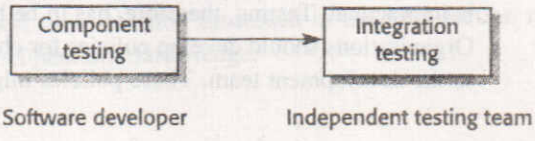


Figure 20.1 Testing phases

a testing perspective, object-oriented systems differ from function-oriented systems in two fundamental ways:

1. In function-oriented systems, there is a fairly clear distinction between basic program units (functions) and collections of these program units (modules). In object-oriented systems, there is no such distinction. Objects may be simple entities such as a list or complex entities such as the weather station object described in Chapter 12 which includes a number of other objects.
2. There is often no clear hierarchy of objects as is commonly found in function-oriented systems. The integration strategies such as top-down or bottom-up integration that are discussed in section 20.2 are often inappropriate.

These differences mean that the boundary between component testing and integration testing is blurred for object-oriented systems. It is a continuation of the seamless object-oriented development process where objects are the basic structure used at all stages of the software process. While many testing techniques are independent of the system design, some techniques have been developed that are specifically geared to testing object-oriented systems. I discuss approaches to object-oriented testing in section 20.3.

20.1 Defect testing

The goal of defect testing is to expose latent defects in a software system before the system is delivered. This contrasts with validation testing which is intended to demonstrate that a system meets its specification. Validation testing requires the system to perform correctly using given acceptance test cases. A successful defect test is a test which causes the system to perform *incorrectly* and hence exposes a defect. This emphasises an important fact about testing. It demonstrates *the presence*, not the absence, of program faults.

A general model of the defect testing process is shown in Figure 20.2. Test cases are specifications of the inputs to the test and the expected output from the system plus a statement of what is being tested. Test data are the inputs which have been devised to test the system. Test data can sometimes be generated automatically. Automatic test case generation is impossible because the test output cannot be predicted.

Exhaustive testing, where every possible program execution sequence is tested, is impractical. Testing, therefore, has to be based on a subset of possible test cases. Organisations should develop policies for choosing this subset rather than leave this to the development team. These policies might be based on general testing policies

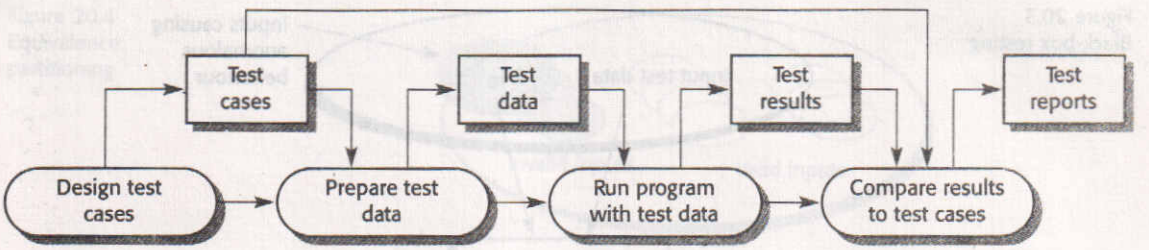


Figure 20.2 The defect testing process

such as a policy that all program statements should be executed at least once. Alternatively, the testing policies may be based on experience of system usage and may focus on testing the features of the operational system. For example:

1. All system functions that are accessed through menus should be tested.
2. Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
3. Where user input is provided, all functions must be tested with both correct and incorrect input.

It is clear from experience with major software products such as word processors or spreadsheets that comparable guidelines have been used during product testing. Unusual combinations of functionality can result in errors but the most commonly used functions usually work correctly.

20.1.1 Black-box testing

Functional or black-box testing is an approach to testing where the tests are derived from the program or component specification. The system is a 'black box' whose behaviour can only be determined by studying its inputs and the related outputs. Another name for this is *functional testing* because the tester is only concerned with the functionality and not the implementation of the software.

Figure 20.3 illustrates the model of a system which is assumed in black-box testing. This approach is equally applicable to systems that are organised as functions or as objects. The tester presents inputs to the component or the system and examines the corresponding outputs. If the outputs are not those predicted then the test has *successfully* detected a problem with the software.

The key problem for the defect tester is to select inputs that have a high probability of being members of the set I_e. In many cases, the selection of these test cases is based on the previous experience of test engineers. They use domain knowledge to identify test cases which are likely to reveal defects. However, the systematic approach to test data selection discussed in the next section may also be used to supplement this heuristic knowledge.

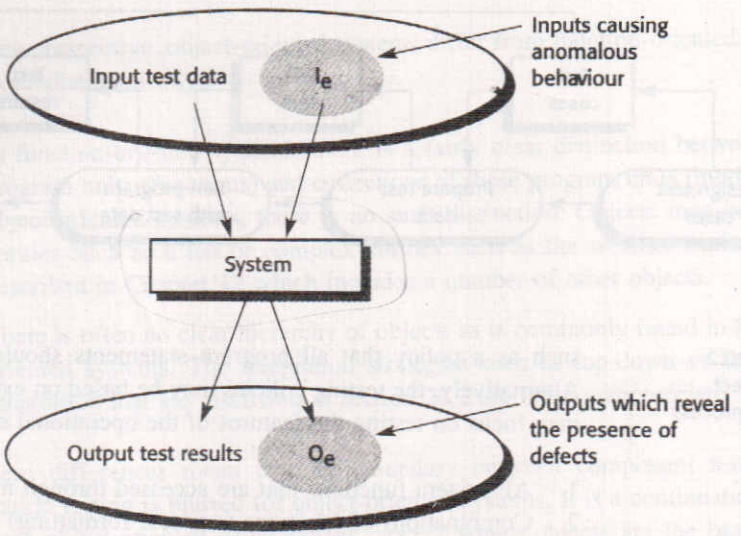


Figure 20.3
Black-box testing

20.1.2 Equivalence partitioning

The input data to a program usually fall into a number of different classes. These have common characteristics, e.g. positive numbers, negative numbers, strings without blanks, etc. Programs normally behave in a comparable way for all members of a class. Because of this equivalent behaviour, these classes are sometimes called equivalence partitions or domains (Beizer, 1990). One systematic approach to defect testing is based on identifying all equivalence partitions which must be handled by a program. Test cases are designed so that the inputs or outputs lie within these partitions.

In Figure 20.4, each equivalence partition is shown as an ellipse. Input equivalence partitions are sets of data where all of the set members should be processed in an equivalent way. Output equivalence partitions are program outputs that have common characteristics so can be considered as a distinct class. You also identify partitions where the inputs are outside the other partitions that you have chosen. These test if the program handles invalid input correctly. Valid and invalid inputs also form equivalence partitions.

Once you have identified a set of partitions, you then choose test cases from each of these partitions. A good guideline to follow for test case selection is to choose test cases on the boundaries of the partitions plus cases close to the mid-point of the partition. The rationale for this guideline is that designers and programmers tend to consider typical values of inputs when developing a system. These are tested by choosing the mid-point of the partition. Boundary values are often atypical (e.g. zero may behave differently from other non-negative numbers) so are overlooked by developers. Program errors often occur when processing these atypical values.

The equivalence partitions may be identified by using the program specification or user documentation and by the tester using experience to predict which classes

Figure 20.4 Equivalence partitioning

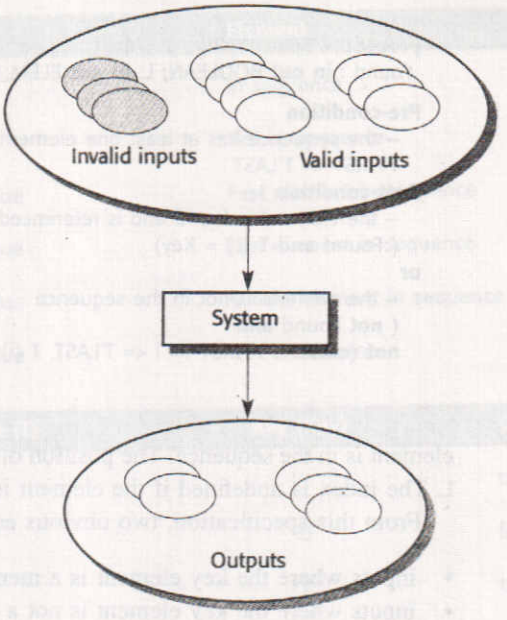
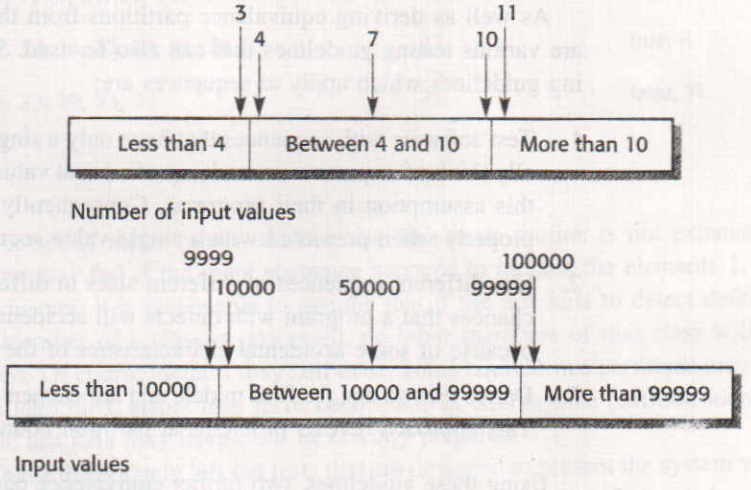


Figure 20.5 Equivalence partitions



of input value are likely to detect errors. For example, say a program specification states that the program accepts 4 to 10 inputs which are five-digit integers greater than 10,000. Figure 20.5 shows the identified equivalence partitions for this situation and possible test input data values.

To illustrate the derivation of test cases, I use the specification of a simplified example (a search routine) that searches a sequence of elements for a given element (the key). It returns the position of that element in the sequence. The specification of this routine, using pre- and post-conditions, is shown in Figure 20.6. The pre-condition states that the search routine has not been designed to work with empty sequences. The post-condition states that the variable Found is set if the key

Figure 20.6 The specification of a search routine

```
procedure Search (Key : ELEM ; T: SEQ of ELEM ;
 Found : in out BOOLEAN; L: in out ELEM_INDEX) ;
Pre-condition
  -- the sequence has at least one element
  T'FIRST <= T'LAST
Post-condition
  -- the element is found and is referenced by L
  ( Found and T (L) = Key)
or
  -- the element is not in the sequence
  ( not Found and
  not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key ))
```

element is in the sequence. The position of the key element is indicated by the index L. The index is undefined if the element is not in the sequence.

From this specification, two obvious equivalence partitions can be identified:

- inputs where the key element is a member of the sequence (Found = true);
- inputs where the key element is not a sequence member (Found = false).

As well as deriving equivalence partitions from the system specification, there are various testing guidelines that can also be used. Some examples of these testing guidelines which apply to sequences are:

1. Test software with sequences that have only a single value. Programmers naturally think of sequences as made up of several values and sometimes they embed this assumption in their programs. Consequently, the program may not work properly when presented with a single-value sequence.
2. Use different sequences of different sizes in different tests. This decreases the chances that a program with defects will accidentally produce a correct output because of some accidental characteristics of the input.
3. Derive tests so that the first, middle and last elements of the sequence are accessed. This approach reveals problems at partition boundaries.

Using these guidelines, two further equivalence partitions of the input array can be identified:

- The input sequence has a single value.
- The number of elements in the input sequence is greater than 1.

These partitions must be combined with the previously identified equivalence partitions, giving the equivalence partitions summarised in Figure 20.7.

A set of possible test cases based on these partitions is also shown in Figure 20.7. If the key element is not in the sequence, the value of L is undefined ('??'). The guideline that different sequences of different sizes should be used has been applied in these test cases.

Figure 20.7 Equivalence partitions for search routine

Array	Element
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

The set of input values used to test the search routine is not exhaustive. The routine may fail if the input sequence happens to include the elements 1, 2, 3 and 4. However, it is reasonable to assume that if the test fails to detect defects when one member of a class is processed, no other members of that class will identify defects. Of course, defects may still exist. Some equivalence partitions may not have been identified, errors may have been made in equivalence partition identification or the test data may have been incorrectly prepared.

I have deliberately left out tests that are designed to present the system with parameters in the wrong order, of the wrong type, etc. This type of error is best detected using program inspection or automated static analysis. Similarly, the tests do not check for unexpected corruption of data outside the component. It does not make sense for black-box tests to check such corruption. Code inspection, as discussed in Chapter 19, can reveal whether this kind of problem is likely to arise.

20.1.3 Structural testing

Structural testing (Figure 20.8) is an approach to testing where the tests are derived from knowledge of the software's structure and implementation. This approach is

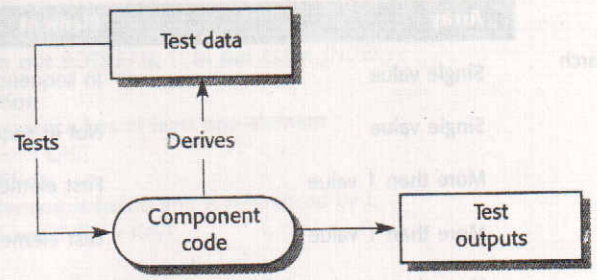


Figure 20.8
Structural testing

Figure 20.9 Java implementation of a binary search routine

```
class BinSearch {
// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes, namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types by
// reference to a function and so return two values
// The key is -1 if the element is not found
    public static void search ( int key, int [] elemArray, Result r )
    {
        int bottom = 0 ;
        int top = elemArray.length - 1 ;
        int mid ;
        r.found = false ; r.index = -1 ;
        while ( bottom <= top )
        {
            mid = (top + bottom) / 2 ;
            if (elemArray [mid] == key)
            {
                r.index = mid ;
                r.found = true ;
                return ;
            } // if part
            else
            {
                if (elemArray [mid] < key)
                    bottom = mid + 1 ;
                else
                    top = mid - 1 ;
            }
        } //while loop
    } // search
} //BinSearch
```

sometimes called 'white-box' testing , 'glass-box' testing or 'clear-box' testing to distinguish it from black-box testing.

Structural testing is usually applied to relatively small program units such as subroutines or the operations associated with an object. As the name implies, the tester

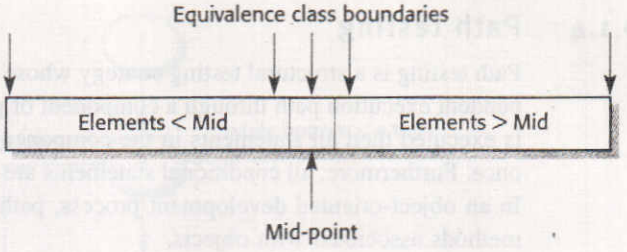


Figure 20.10 Binary search equivalence classes

can analyse the code and use knowledge about the structure of a component to derive test data. The analysis of the code can be used to find how many test cases are needed to guarantee that all of the statements in the program or component are executed at least once during the testing process.

Knowledge of the algorithm used to implement some function can be used to identify further equivalence partitions. To illustrate this, I have instantiated the search routine specification (Figure 20.6) as a binary search routine (Figure 20.9). Of course, this has stricter pre-conditions. The sequence is implemented as an array, that array must be ordered and the value of the lower bound of the array must be less than the value of the upper bound.

By examining the code of the search routine, we can see that binary searching involves splitting the search space into three parts. Each of these parts makes up an equivalence partition (Figure 20.10). Test cases where the key lies at the boundaries of each of these partitions should be chosen to exercise the code.

The test cases shown in Figure 20.7 must therefore be modified so that the input array is arranged in ascending order. Further cases based on knowledge of the algorithm used should be added to the test set. These are elements which are adjacent to the mid-point of the array. Figure 20.11 shows a set of test cases for the binary search routine.

Figure 20.11 Test cases for search routine

Input array (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

20.1.4 Path testing

Path testing is a structural testing strategy whose objective is to exercise every independent execution path through a component or program. If every independent path is executed then all statements in the component must have been executed at least once. Furthermore, all conditional statements are tested for both true and false cases. In an object-oriented development process, path testing may be used when testing methods associated with objects.

The number of paths through a program is usually proportional to its size. As modules are integrated into systems, it becomes unfeasible to use structural testing techniques. Path testing techniques are therefore mostly used at the unit testing and module testing stages of the testing process.

Path testing does not test all possible combinations of all paths through the program. For any components apart from very trivial ones without loops, this is an impossible objective. There are an infinite number of possible path combinations in programs with loops. Defects which manifest themselves when particular path combinations arise may still be present even although all program statements have been executed at least once.

The starting point for path testing is a program flow graph. This is a skeletal model of all paths through the program. A flow graph consists of nodes representing decisions and edges showing flow of control. The flow graph is constructed by replacing program control statements by equivalent diagrams. If there are no goto statements in a program, it is a simple process to derive its flow graph. Sequential statements (assignments, procedure calls and I/O statements) can be ignored in the flow graph construction. Each branch in a conditional statement (if-then-else or case) is shown as a separate path and loops are indicated by an arrow looping back to the loop condition node. Loops and conditional branches are illustrated in the flow graph for the binary search routine (Figure 20.12).

The objective of structural testing is to ensure that each independent program path is executed at least once. An independent program path is one which traverses at least one new edge in the flow graph. In program terms, this means exercising one or more new conditions. Both the true and false branches of all conditions must be executed.

The flow graph for the binary search procedure is shown in Figure 20.12. By tracing the flow, therefore, we see that the independent paths through the binary search flow graph are:

1, 2, 3, 8, 9
1, 2, 3, 4, 6, 7, 2
1, 2, 3, 4, 5, 7, 2
1, 2, 3, 4, 6, 7, 2, 8, 9

If all of these paths are executed we can be sure that:

1. every statement in the method has been executed at least once, and
2. every branch has been exercised for true and false conditions.

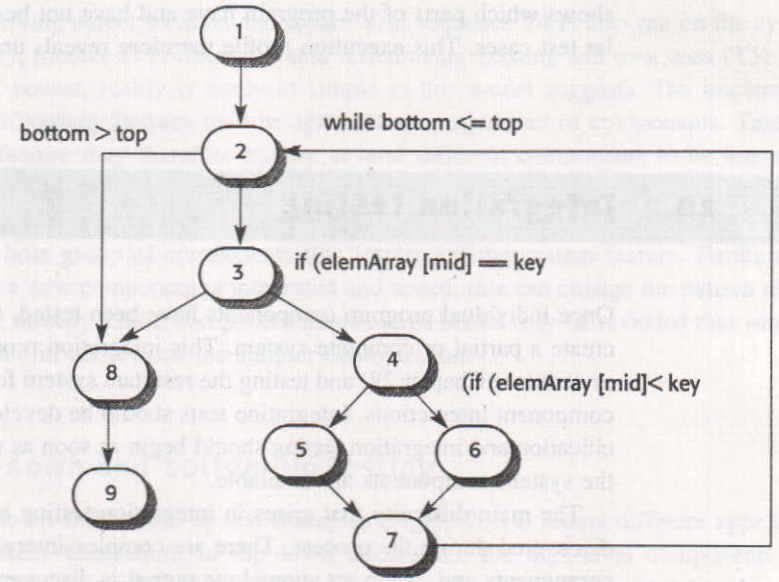


Figure 20.12 Flow graph for a binary search routine

The number of independent paths in a program can be discovered by computing the cyclomatic complexity (McCabe, 1976) of the program flow graph. The cyclomatic complexity, CC, of any connected graph G may be computed according to the following formula:

$$CC(G) = \text{Number (edges)} - \text{Number (nodes)} + 2$$

For programs without goto statements, the value of the cyclomatic complexity is one more than the number of conditions in the program. In a compound condition with more than one test you should count each test. Thus, if there are six if-statements and a while loop, with all conditional expressions simple, the cyclomatic complexity is 8. If a conditional expression is a compound expression with two logical operators ('and' or 'or') the cyclomatic complexity is 10. The cyclomatic complexity of the binary search routine (Figure 20.10) is 4.

After discovering the number of independent paths through the code by computing the cyclomatic complexity, the next step is to design test cases to execute each of these paths. The minimum number of test cases required to test all program paths is equal to the cyclomatic complexity.

Test case design is straightforward in the case of the binary search routine. However, when programs have a complex branching structure, it may be difficult to predict how any particular test case will be processed. In these cases, a dynamic program analyser can be used to discover the program's execution profile.

Dynamic program analysers are testing tools that work in conjunction with compilers. During compilation, additional object code instructions are added to the generated code. These count the number of times each program statement has been executed. After the program has been run, an execution profile can be printed which

shows which parts of the program have and have not been executed using particular test cases. This execution profile therefore reveals untested program sections.

20.2 Integration testing

Once individual program components have been tested, they must be integrated to create a partial or complete system. This integration process involves building the system (see Chapter 29) and testing the resultant system for problems that arise from component interactions. Integration tests should be developed from the system specification and integration testing should begin as soon as usable versions of some of the system components are available.

The main difficulty that arises in integration testing is localising errors that are discovered during the process. There are complex interactions between the system components and, when an anomalous output is discovered, it may be hard to find the source of the error. To make it easier to locate errors, you should always use an incremental approach to system integration and testing. Initially, you should integrate a minimal system configuration and test this system. You then add components to this minimal configuration and test after each added increment.

In the example shown in Figure 20.13, test sequences T1, T2 and T3 are first run on a system composed of module A and module B (the minimal system). If these reveal defects, these are corrected. Module C is integrated and tests T1, T2 and T3 are repeated to ensure that there have not been unexpected interactions with A and B. If problems arise in these tests, this probably means that they are due to interactions with the new module. The source of the problem is localised, thus

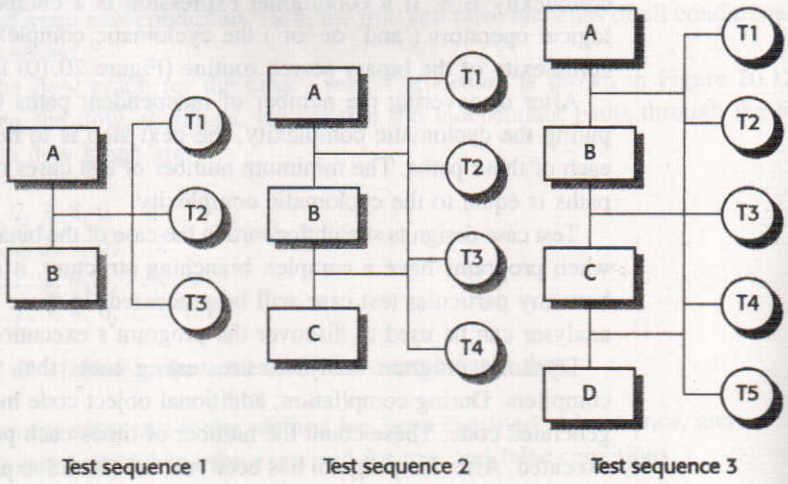


Figure 20.13 Incremental integration testing

simplifying defect location and repair. Test sequence T4 is also run on the system. Finally, module D is integrated and tested using existing and new tests (T5).

Of course, reality is rarely as simple as this model suggests. The implementation of system features may be spread across a number of components. Testing a new feature may therefore require several different components to be integrated. The testing may reveal errors in the interactions between these individual components and other parts of the system. Repairing errors may be difficult because it affects the whole group of components that implement the system feature. Furthermore, when a new component is integrated and tested, this can change the pattern of previous, already tested, component interactions. Errors may be revealed that were not exposed in the tests of the simpler configuration.

20.2.1 Top-down and bottom-up testing

Top-down and bottom-up test strategies (Figure 20.14) reflect different approaches to system integration. In top-down integration, the high-level components of a system are integrated and tested before their design and implementation has been

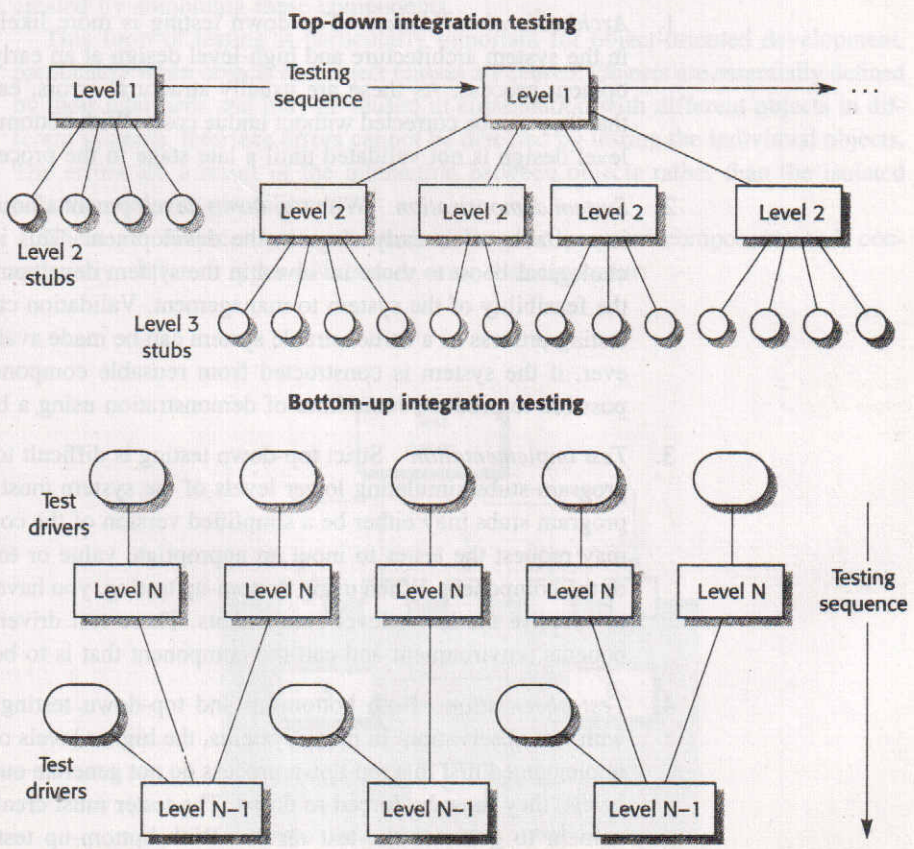


Figure 20.14 Top-down and bottom-up integration testing

completed. In bottom-up integration, low-level components are integrated and tested before the higher-level components have been developed.

Top-down testing is an integral part of a top-down development process where the development process starts with high-level components and works down the component hierarchy. The program is represented as a single abstract component with sub-components represented by stubs. Stubs have the same interface as the component but very limited functionality. After the top-level component has been programmed and tested, its sub-components are implemented and tested in the same way. This process continues until the bottom-level components are implemented. The whole system has then been completely tested.

By contrast, bottom-up testing involves integrating and testing the modules at the lower levels in the hierarchy, and then working up the hierarchy of modules until the final module is tested. This approach does not require the architectural design of the system to be complete so it can start at an early stage in the development process. It may be used where the system reuses and modifies components from other systems.

Top-down and bottom-up integration testing can be compared under four headings:

1. *Architectural validation* Top-down testing is more likely to discover errors in the system architecture and high-level design at an early stage in the development process. As these are usually structural errors, early detection means that they can be corrected without undue costs. With bottom-up testing, the high-level design is not validated until a late stage in the process.

2. *System demonstration* With top-down development a limited, working system is available at an early stage in the development. This is an important psychological boost to those involved in the system development. It demonstrates the feasibility of the system to management. Validation can begin early in the testing process as a demonstrable system can be made available to users. However, if the system is constructed from reusable components, it may also be possible to produce some kind of demonstration using a bottom-up approach.

3. *Test implementation* Strict top-down testing is difficult to implement because program stubs simulating lower levels of the system must be produced. These program stubs may either be a simplified version of the component required or may request the tester to input an appropriate value or to simulate the action of the component. When using bottom-up testing, you have to write test drivers to exercise the lower-level components. These test drivers simulate the components' environment and call the component that is to be tested.

4. *Test observation* Both bottom-up and top-down testing can have problems with test observation. In many systems, the higher levels of the system that are implemented first in a top-down process do not generate output but, to test these levels, they must be forced to do so. The tester must create an artificial environment to generate the test results. With bottom-up testing, it may also be

necessary to create an artificial environment (the test drivers) so that the execution of the lower-level components can be observed.

In reality, systems are usually developed and tested using a mixture of top-down and bottom-up approaches. Different development schedules for different parts of the system mean that the integration and testing team must work with whatever components are available. Therefore, a mixture of stubs and test drivers must inevitably be developed during the integration testing process.

20.2.2 Interface testing

Interface testing takes place when modules or sub-systems are integrated to create larger systems. Each module or sub-system has a defined interface which is called by other program components. The objective of interface testing is to detect faults which may have been introduced into the system because of interface errors or invalid assumptions about interfaces.

Figure 20.15 illustrates interface testing. The arrows to the box boundary mean that the test cases are not applied to the individual components but to the sub-system created by combining these components.

This form of testing is particularly important for object-oriented development, particularly when objects and object classes are reused. Objects are essentially defined by their interfaces and may be reused in combination with different objects in different systems. Interface errors cannot be detected by testing the individual objects. The errors are a result of the interaction between objects rather than the isolated behaviour of a single object.

There are different types of interface between program components and, consequently, different types of interface error that can occur:

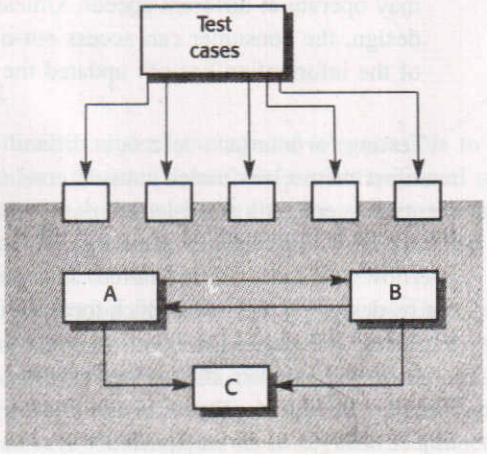


Figure 20.15 Interface testing

1. *Parameter interfaces* These are interfaces where data or sometimes function references are passed from one component to another.
2. *Shared memory interfaces* These are interfaces where a block of memory is shared between sub-systems. Data is placed in the memory by one sub-system and retrieved from there by other sub-systems.
3. *Procedural interfaces* These are interfaces where one sub-system encapsulates a set of procedures which can be called by other sub-systems. Objects and abstract data types have this form of interface.
4. *Message passing interfaces* These are interfaces where one sub-system requests a service from another sub-system by passing a message to it. A return message includes the results of executing the service. Some object-oriented systems have this form of interface, as do client–server systems.

Interface errors are one of the most common forms of error in complex systems (Lutz, 1993). These errors fall into three classes:

- *Interface misuse* A calling component calls some other component and makes an error in the use of its interface. This type of error is particularly common with parameter interfaces where parameters may be of the wrong type, may be passed in the wrong order or the wrong number of parameters may be passed.
- *Interface misunderstanding* A calling component misunderstands the specification of the interface of the called component and makes assumptions about the behaviour of the called component. The called component does not behave as expected and this causes unexpected behaviour in the calling component. For example, a binary search routine may be called with an unordered array to be searched. The search would then fail.
- *Timing errors* These occur in real-time systems which use a shared memory or a message passing interface. The producer of data and the consumer of data may operate at different speeds. Unless particular care is taken in the interface design, the consumer can access out-of-date information because the producer of the information has not updated the shared interface information.

Testing for interface defects is difficult because some interface faults may only manifest themselves under unusual conditions. For example, say an object implements a queue as a fixed-length data structure. A calling object may assume that the queue is implemented as an infinite data structure and may not check for queue overflow when an item is entered. This condition can only be detected during testing by designing test cases which force the queue to overflow and cause that overflow to corrupt the object behaviour in some detectable way.

A further problem may arise because of interactions between faults in different modules or objects. Faults in one object may only be detected when some other object behaves in an unexpected way. For example, an object may call some other

object to receive some service and may assume that the response is correct. If there has been a misunderstanding about the value computed, the returned value may be valid but incorrect. This will only manifest itself when some later computation goes wrong.

Some general guidelines for interface testing are:

1. Examine the code to be tested and explicitly list each call to an external component. Design a set of tests where the values of the parameters to the external components are at the extreme ends of their ranges. These extreme values are most likely to reveal interface inconsistencies.
2. Where pointers are passed across an interface, always test the interface with null pointer parameters.
3. Where a component is called through a procedural interface, design tests which should cause the component to fail. Differing failure assumptions are one of the most common specification misunderstandings.
4. Use stress testing, as discussed in the following section, in message passing systems. Design tests which generate many more messages than are likely to occur in practice. Timing problems may be revealed in this way.
5. Where several components interact through shared memory, design tests which vary the order in which these components are activated. These tests may reveal implicit assumptions made by the programmer about the order in which the shared data is produced and consumed.

Static techniques are often more cost-effective than testing for discovering interface errors. A strongly typed language such as Java allows many interface errors to be trapped by the compiler. Where a weaker language, such as C, is used, a static analyser such as LINT (see Chapter 19) can detect interface errors. Program inspections can concentrate on component interfaces and questions about the assumed interface behaviour can be asked during the inspection process.

20.2.3 Stress testing

Once a system has been completely integrated it is possible to test the system for emergent properties (see Chapter 2) such as performance and reliability. Performance tests have to be designed to ensure that the system can process its intended load. This usually involves planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.

Some classes of system are designed to handle a specified load. For example, a transaction processing system may be designed to process up to 100 transactions per second; an operating system may be designed to handle up to 200 separate terminals. Stress testing continues these tests beyond the maximum design load of the system until the system fails. This type of testing has two functions:

1. It tests the failure behaviour of the system. Circumstances may arise through an unexpected combination of events where the load placed on the system exceeds the maximum anticipated load. In these circumstances, it is important that system failure should not cause data corruption or unexpected loss of user services. Stress testing checks that overloading the system causes it to 'fail-soft' rather than collapse under its load.
2. It stresses the system and may cause defects to come to light which would not normally manifest themselves. Although it can be argued that these defects are unlikely to cause system failures in normal usage, there may be unusual combinations of normal circumstances which the stress testing replicates.

Stress testing is particularly relevant to distributed systems based on a network of processors. These systems often exhibit severe degradation when they are heavily loaded. The network becomes swamped with coordination data which the different processes must exchange so the processes become slower and slower as they wait for the required data from other processes.

20.3 Object-oriented testing

In Figure 20.1, I suggested that there were two fundamental activities in the testing process. These are component testing where system components are tested individually and integration testing where collections of components are integrated into sub-systems and the final system for testing. These activities are equally applicable to object-oriented systems. However, there are important differences between object-oriented systems and systems developed using a functional model:

1. Objects as individual components are often larger than single functions.
2. Objects that are integrated into sub-systems are usually loosely coupled and there is no obvious 'top' to the system.
3. If objects are reused, the testers may have no access to the source code of the component for analysis.

These differences mean that white-box testing approaches based on code analysis have to be extended to cover larger-grain objects and that alternative approaches to integration testing may have to be adopted. However, once a system has been integrated, the fact that it has been developed as an object-oriented system should not be apparent to system users.

In an object-oriented system, four levels of testing can be identified:

1. *Testing the individual operations associated with objects* These are functions or procedures and the black-box and white-box approaches discussed above may be used.
2. *Testing individual object classes* The principle of black-box testing is unchanged but the notion of an equivalence class must be extended to cover related operation sequences. Similarly, structural testing requires a different type of analysis. This is discussed in section 20.3.1.
3. *Testing clusters of objects* Strict top-down or bottom-up integration is inappropriate to create groups of related objects. Other approaches such as scenario-based testing should be used. These are discussed in section 20.3.2.
4. *Testing the object-oriented system* Verification and validation against the system requirements specification is carried out in exactly the same way as for any other type of system.

Testing techniques for object-oriented systems have matured rapidly and there is now a good deal of information available on object-oriented testing techniques (Binder, 1999). The following sections give an overview of a basic approach to object-oriented system testing.

20.3.1 Object class testing

The approach to test coverage discussed in section 20.1.3 was concerned with ensuring that all statements in a program were executed at least once and that all paths through the program were executed. When testing objects, complete test coverage should include:

1. the testing in isolation of all operations associated with the object;
2. the setting and interrogation of all attributes associated with the object;
3. the exercise of the object in all possible states. This means that all events that cause a state change in the object should be simulated.

Consider, for example, the weather station from Chapter 12 whose interface is shown in Figure 20.16. It has only a single attribute, which is its identifier. This is a constant that is set when the weather station is installed. You therefore only need a test that checks if it has been set up. You need to define test cases for **reportWeather**, **calibrate**, **test**, **startup** and **shutdown**. Ideally, you should test methods in isolation but, in some cases, some test sequences are necessary. For example, to test **shutdown** you need to have executed the **startup** method.

To test the states of the weather station, you use a state model as shown in Figure 12.14. Using this model, you can identify sequences of state transitions that have to be tested and define event sequences to force these transitions. In principle,

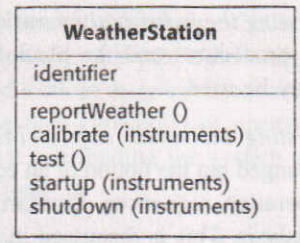


Figure 20.16
The weather station object interface

you should test every possible state transition sequence although in practice this may be too expensive. Examples of state sequences that should be tested in the weather station include:

Shutdown → Waiting → Shutdown

Waiting → Calibrating → Testing → Transmitting → Waiting

Waiting → Collecting → Waiting → Summarising → Transmitting → Waiting

If inheritance is used, this makes it more difficult to design object class tests. Where a superclass provides operations that are inherited by a number of subclasses, all of these subclasses should be tested with all inherited operations. The reason for this is that the inherited operation may make assumptions about other operations and attributes and these may have been changed when inherited. Equally, when a superclass operation is overridden, the overwriting operation must be tested.

The notion of equivalence classes may also be applied to object classes. Tests that fall into the same equivalence class might be those that use the same attributes of the objects. Therefore, equivalence classes should be identified which initialise, access and update all object class attributes.

20.3.2 Object integration

When object-oriented systems are developed, the levels of integration are less distinct. Clearly, operations and data are integrated to form objects and object classes. Testing these object classes corresponds to unit testing. There is no direct equivalent to module testing in object-oriented systems. However, Murphy *et al.* (1994) suggest that groups of classes which act in combination to provide a set of services should be tested together. They call this *cluster testing*.

Neither top-down nor bottom-up integration is really appropriate for object-oriented systems. In these systems, there is no obvious 'top' that provides a goal for the integration nor is there a clear hierarchy of objects that can be created. Clusters therefore have to be created using knowledge of their operation and the features of the system that are implemented by these clusters. There are three possible approaches to integration testing that may be used:

1. *Use-case or scenario-based testing* Use-cases or scenarios (see Chapter 6) describe one mode of use of the system. Testing can be based on these scenario descriptions and object clusters created to support the use-cases that relate to that mode of use.

2. *Thread testing* Thread testing is based on testing the system's response to a particular input or set of input events. Object-oriented systems are often event-driven so this is a particularly appropriate form of testing to use. To use this approach, you have to identify how the processing of events threads its way through the system.

3. *Object interaction testing* A related approach to testing groups of interacting objects is proposed by Jorgensen and Erickson (1994). They suggest that an intermediate level of integration testing can be based on identifying 'method-message' paths. These are traces through a sequence of object interactions which stop when an object operation does not call on the services of any other object. They also identify a related construct which they call an 'Atomic System Function' (ASF). An ASF consists of some input event followed by a sequence of MM-paths which is terminated by an output event. This is similar to the notion of a thread in a real-time system.

Scenario-based testing is often the most effective of these approaches as it can be organised so that the most likely scenarios are tested first with unusual or exceptional scenarios considered later in the testing process. This satisfies a fundamental principle of testing that most testing effort should be devoted to those parts of the system that receive the most use.

To illustrate scenario-based testing, I again use an example from the weather station system. Scenarios can be identified from the use-cases that you may have developed but these may not include enough information for testing purposes. To supplement these use-cases when planning testing, you can use interaction diagrams that show in more detail how a scenario is implemented by the objects in a system.

Consider Figure 20.17 (taken from Chapter 12) which shows the sequence of operations in the weather station when it responds to a request to collect data for the mapping system. You can use this diagram to identify operations that will be tested and to help design the test cases to execute the tests.

Therefore, you can see that issuing a request for a report will result in the following thread of methods to be executed:

CommsController:request → WeatherStation:report → WeatherData:summarise

When choosing scenarios to devise system tests, it is important to ensure that you execute each method in each class at least once. Therefore, you should create a checklist of object classes and methods and, when a scenario is chosen, you should then tick the methods that have been executed. Of course, it isn't possible to execute all combinations of methods but this at least ensures that all individual methods have been tested as part of a sequence.

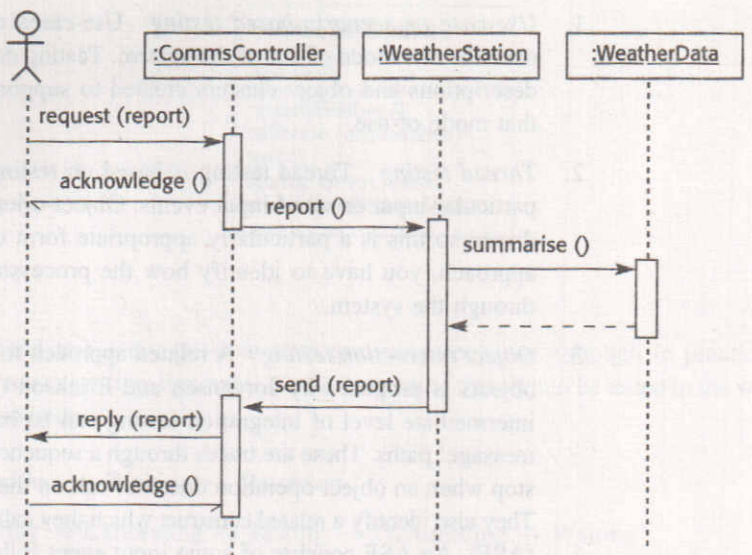


Figure 20.17 Collect weather data sequence chart

The sequence diagram can also be used to identify inputs and outputs that have to be created for the test:

1. An input of a request for a report should have an associated acknowledgement and a report should ultimately be returned from the request. During the testing, you should create summarised data that can be used to check that the report is correctly organised.
2. An input request for a report to WeatherStation results in a summarised report being generated. You can test this in isolation by creating raw data corresponding to the summary that you have prepared for the test of CommsController and checking that the WeatherStation object correctly produces this summary.
3. This raw data is also used to test the WeatherData object.

Of course, I have simplified the sequence chart in Figure 20.17 so that it does not show exceptions. A complete scenario test must also take these into account and ensure that these are correctly handled by the objects.

20.4 Testing workbenches

Testing is an expensive and laborious phase of the software process. As a result, testing tools were among the first software tools to be developed. These tools now

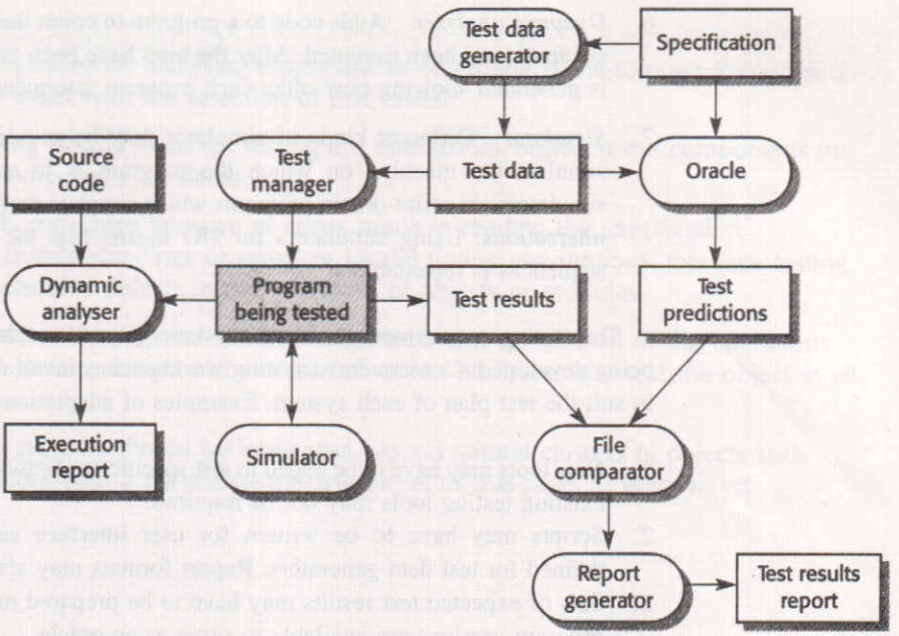


Figure 20.18
A testing workbench

offer a range of facilities and their use significantly reduces the cost of the testing process. Different testing tools may be integrated into testing workbenches.

Figure 20.18 shows some tools which may be included in a testing workbench and the interactions between these tools. Tools included are:

1. *Test manager* Manages the running of program tests. The test manager keeps track of test data, expected results and program facilities tested.
2. *Test data generator* Generates test data for the program to be tested. This may be accomplished by selecting data from a database or by using patterns to generate random data of the correct form.
3. *Oracle* Generates predictions of expected test results. Oracles may be either previous program versions or prototype systems. Back-to-back testing involves running the oracle and the program to be tested in parallel. Differences in their outputs are highlighted.
4. *File comparator* Compares the results of program tests with previous test results and reports differences between them. Comparators are essential in regression testing where the results of executing old and new versions are compared. Differences in these results indicate potential problems with the new version of the system.
5. *Report generator* Provides report definition and generation facilities for test results.

6. *Dynamic analyser* Adds code to a program to count the number of times each statement has been executed. After the tests have been run, an execution profile is generated showing how often each program statement has been executed.
7. *Simulator* Different kinds of simulator may be provided. Target simulators simulate the machine on which the program is to execute. User interface simulators are script-driven programs which simulate multiple simultaneous user interactions. Using simulators for I/O means that the timing of transaction sequences is repeatable.

The testing requirements for large systems depend on the application which is being developed. Consequently, testing workbenches invariably have to be adapted to suit the test plan of each system. Examples of adaptations are:

1. New tools may have to be added to test specific application characteristics. Some existing testing tools may not be required.
2. Scripts may have to be written for user interface simulators and patterns defined for test data generators. Report formats may also have to be defined.
3. Sets of expected test results may have to be prepared manually if no previous program versions are available to serve as an oracle.
4. Special-purpose file comparators may have to be written which include knowledge of the structure of the test results on file.

A significant amount of effort and time is usually needed to create a comprehensive testing workbench. Complete test workbenches as shown in Figure 20.18 are therefore only used when large systems are being developed. For these systems, the overall testing costs may be up to 50 per cent of the total development costs so it is cost-effective to invest in high-quality CASE tool support for testing.

KEY POINTS

- It is more important to test the parts of the system which are commonly used rather than parts which are only rarely exercised.
- Equivalence partitioning is a way of deriving test cases. It depends on finding partitions in the input and output data sets and exercising the program with values from these partitions. Often, the value which is most likely to lead to a successful test is a value at the boundary of a partition.
- Black-box testing does not need access to source code. Test cases are derived from the program specification.

- Structural testing relies on analysing a program to determine paths through it and using this analysis to assist with the selection of test cases.
- Integration testing should focus on testing the interactions between the components in a system and component interfaces.
- Interface defects may arise because of errors made in reading the specification, specification misunderstandings or errors or invalid timing assumptions. Interface testing is intended to discover defects in the interfaces of objects or modules.
- When testing object classes, you should design tests that exercise all of the operations associated with a class, assign and evaluate all object attributes and test the object in all possible states.
- Object-oriented systems should be integrated around natural clusters of objects such as those associated with a particular use-case or set of use-cases or with object threads.

FURTHER READING

Testing Object-oriented Systems: Models, Patterns and Tools. This is an immense book (1000+ pages) that provides very complete coverage of object-oriented testing. Its completeness means that it shouldn't be the first thing that you read on object-oriented testing (most books on object-oriented development have a testing chapter) but it is the object-oriented equivalent of Beizer's book. (R. V. Binder, 1999, Addison Wesley Longman.)

'How to design practical test cases'. A how-to article on test case design by an author from a Japanese company that has a very good reputation for delivering software with very few faults. (T. Yamaura, *IEEE Software*, 15(6), November 1998.)

'Regression testing in an industrial environment'. This paper discusses the importance of regression testing and gives practical advice on the process. (A. K. Onoma, W-T Tsai, M. Poonawala and H. Suganuma, *Comm ACM*, 41(5), May 1998.)

Software Testing Techniques. This is the definitive book on defect testing for programs which are developed using a function-oriented approach. It is incredibly detailed and thorough. Beizer's approach is practical rather than theoretical and he gives detailed advice for system testers. (B. Beizer, 1990, Van Nostrand Rheinhold.)

EXERCISES

20.1 Discuss the differences between black-box and structural testing and suggest how they can be used together in the defect testing process.

20.2 What testing problems might arise in numerical routines designed to handle very large and very small numbers?

20.3 Derive a set of test cases for the following components:

- A sort routine which sorts arrays of integers.
- A routine which takes a line of text as input and counts the number of non-blank characters in that line.
- A routine which examines a line of text and replaces sequences of blank characters with a single blank character.
- An object that implements variable length character strings. Operations should include concatenation, length (to give the length of a string) and substring selection.
- An object representing a keyed table where entries are made and retrieved using some alphabetic key.

20.4 Program the first three of these routines using a language of your choice and, for each routine, derive its cyclomatic complexity.

20.5 Show, using a small example, why it is practically impossible to exhaustively test a program.

20.6 By examining the code of the routines which you have written, derive further test cases in addition to those you have already considered. Has the code analysis revealed omissions in your initial set of test cases?

20.7 Implement (using Java or C++) an object class called SYMBOL_TABLE which could be used as part of a compilation system. You should include operations to add a name and associated type information to the table, to delete a name, to modify the information associated with a name and to search the table. You should maintain information about where the object is declared. Organise a program inspection of this object (see Chapter 19) and keep a careful account of the errors discovered. Test the object using a black-box approach and compare errors which are revealed by testing with those discovered by inspection.

20.8 Explain why interface testing is necessary given that individual units have been extensively validated through unit testing and program inspections.

20.9 Explain why bottom-up and top-down testing may be inappropriate testing strategies for object-oriented systems.

20.10 Derive test cases to test the states of the microwave oven whose state model is defined in Figure 7.5.