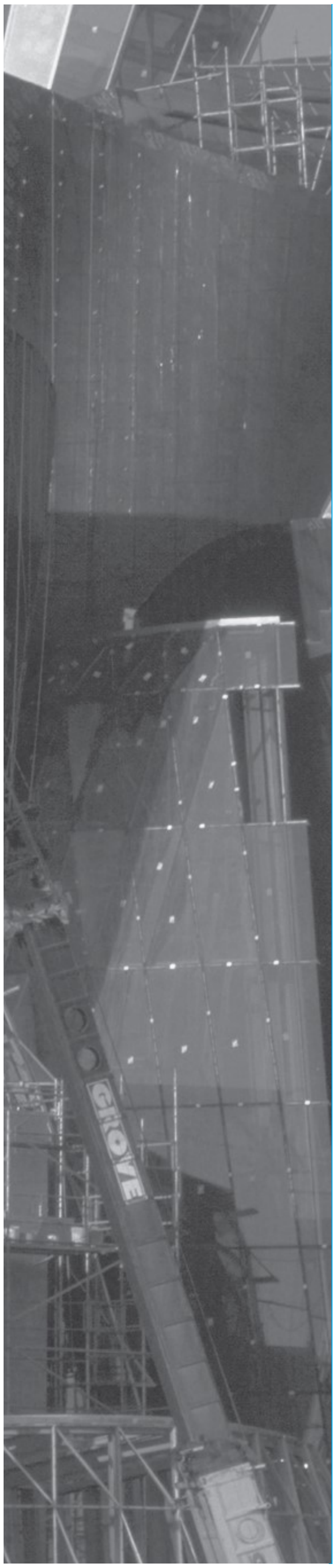

8

Software testing

Objectives

The objective of this chapter is to introduce software testing and software testing processes. When you have read the chapter, you will:

- understand the stages of testing from testing, during development to acceptance testing by system customers;
- have been introduced to techniques that help you choose test cases that are geared to discovering program defects;
- understand test-first development, where you design tests before writing code and run these tests automatically;
- know the important differences between component, system, and release testing and be aware of user testing processes and techniques.

Contents

Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use. When you test software, you execute a program using artificial data. You check the results of the test run for errors, anomalies, or information about the program's non-functional attributes.

The testing process has two distinct goals:

1. To demonstrate to the developer and the customer that the software meets its requirements. For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
2. To discover situations in which the behavior of the software is incorrect, undesirable, or does not conform to its specification. These are a consequence of software defects. Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations, and data corruption.

The first goal leads to validation testing, where you expect the system to perform correctly using a given set of test cases that reflect the system's expected use. The second goal leads to defect testing, where the test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used. Of course, there is no definite boundary between these two approaches to testing. During validation testing, you will find defects in the system; during defect testing, some of the tests will show that the program meets its requirements.

The diagram shown in Figure 8.1 may help to explain the differences between validation testing and defect testing. Think of the system being tested as a black box. The system accepts inputs from some input set I and generates outputs in an output set O. Some of the outputs will be erroneous. These are the outputs in set O_e that are generated by the system in response to inputs in the set I_e. The priority in defect testing is to find those inputs in the set I_e because these reveal problems with the system. Validation testing involves testing with correct inputs that are outside I_e. These stimulate the system to generate the expected correct outputs.

Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstance. It is always possible that a test that you have overlooked could discover further problems with the system. As Edsger Dijkstra, an early contributor to the development of software engineering, eloquently stated (Dijkstra et al., 1972):

Testing can only show the presence of errors, not their absence

Testing is part of a broader process of software verification and validation (V & V). Verification and validation are not the same thing, although they are often confused.

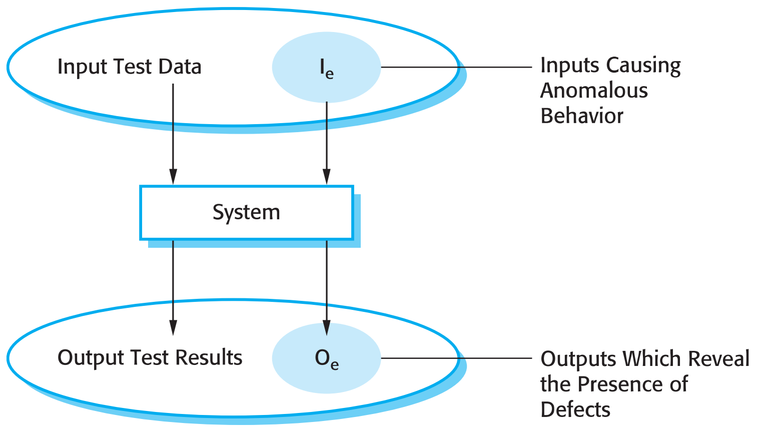
Input Test Data
I_e
Inputs Causing Anomalous Behavior
System
Output Test Results
O_e
Outputs Which Reveal the Presence of Defects

Figure 8.1 An input-output model of program testing

Barry Boehm, a pioneer of software engineering, succinctly expressed the difference between them (Boehm, 1979):

- 'Validation: Are we building the right product?'
- 'Verification: Are we building the product right?'

Verification and validation processes are concerned with checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software. These checking processes start as soon as requirements become available and continue through all stages of the development process.

The aim of verification is to check that the software meets its stated functional and non-functional requirements. Validation, however, is a more general process. The aim of validation is to ensure that the software meets the customer's expectations. It goes beyond simply checking conformance with the specification to demonstrating that the software does what the customer expects it to do. Validation is essential because, as I discussed in Chapter 4, requirements specifications do not always reflect the real wishes or needs of system customers and users.

The ultimate goal of verification and validation processes is to establish confidence that the software system is 'fit for purpose'. This means that the system must be good enough for its intended use. The level of required confidence depends on the system's purpose, the expectations of the system users, and the current marketing environment for the system:

1. *Software purpose* The more critical the software, the more important that it is reliable. For example, the level of confidence required for software used to control a safety-critical system is much higher than that required for a prototype that has been developed to demonstrate new product ideas.

2. *User expectations* Because of their experiences with buggy, unreliable software, many users have low expectations of software quality. They are not surprised when their software fails. When a new system is installed, users may tolerate

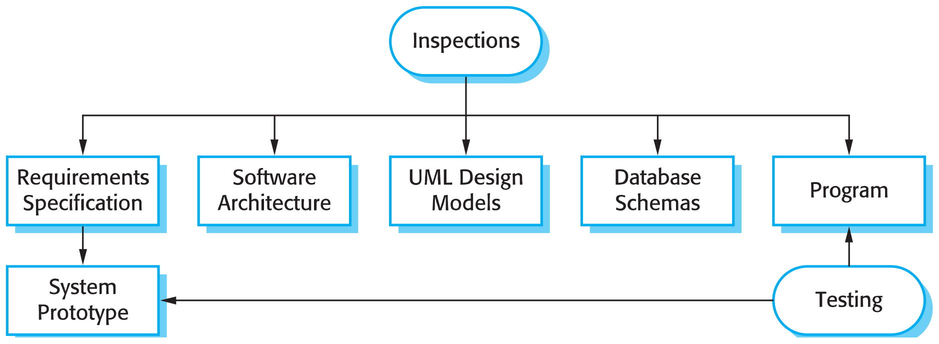

Figure 8.2 Inspections and testing

failures because the benefits of use outweigh the costs of failure recovery. In these situations, you may not need to devote as much time to testing the software. However, as software matures, users expect it to become more reliable so more thorough testing of later versions may be required.

3. *Marketing environment* When a system is marketed, the sellers of the system must take into account competing products, the price that customers are willing to pay for a system, and the required schedule for delivering that system. In a competitive environment, a software company may decide to release a program before it has been fully tested and debugged because they want to be the first into the market. If a software product is very cheap, users may be willing to tolerate a lower level of reliability.

As well as software testing, the verification and validation process may involve software inspections and reviews. Inspections and reviews analyze and check the system requirements, design models, the program source code, and even proposed system tests. These are so-called 'static' V & V techniques in which you don't need to execute the software to verify it. Figure 8.2 shows that software inspections and testing support V & V at different stages in the software process. The arrows indicate the stages in the process where the techniques may be used.

Inspections mostly focus on the source code of a system but any readable representation of the software, such as its requirements or a design model, can be inspected. When you inspect a system, you use knowledge of the system, its application domain, and the programming or modeling language to discover errors.

There are three advantages of software inspection over testing:

1. During testing, errors can mask (hide) other errors. When an error leads to unexpected outputs, you can never be sure if later output anomalies are due to a new error or are side effects of the original error. Because inspection is a static process, you don't have to be concerned with interactions between errors. Consequently, a single inspection session can discover many errors in a system.

Test planning

Test planning is concerned with scheduling and resourcing all of the activities in the testing process. It involves defining the testing process, taking into account the people and the time available. Usually, a test plan will be created, which defines what is to be tested, the predicted testing schedule, and how tests will be recorded. For critical systems, the test plan may also include details of the tests to be run on the software.

http://www.SoftwareEngineering-9.com/Web/Testing/Planning.html

2. Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available. This obviously adds to the system development costs.

3. As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability, and maintainability. You can look for inefficiencies, inappropriate algorithms, and poor programming style that could make the system difficult to maintain and update.

Program inspections are an old idea and there have been several studies and experiments that have demonstrated that inspections are more effective for defect discovery than program testing. Fagan (1986) reported that more than 60% of the errors in a program can be detected using informal program inspections. In the Cleanroom process (Prowell et al., 1999), it is claimed that more than 90% of defects can be discovered in program inspections.

However, inspections cannot replace software testing. Inspections are not good for discovering defects that arise because of unexpected interactions between different parts of a program, timing problems, or problems with system performance. Furthermore, especially in small companies or development groups, it can be difficult and expensive to put together a separate inspection team as all potential members of the team may also be software developers. I discuss reviews and inspections in more detail in Chapter 24 (Quality Management). Automated static analysis, where the source text of a program is automatically analyzed to discover anomalies, is explained in Chapter 15. In this chapter, I focus on testing and testing processes.

Figure 8.3 is an abstract model of the 'traditional' testing process, as used in plan-driven development. Test cases are specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested. Test data are the inputs that have been devised to test a system. Test data can sometimes be generated automatically, but automatic test case generation is impossible, as people who understand what the system is supposed to do must be involved to specify the expected test results. However, test execution can be automated. The expected results are automatically compared with the predicted results so there is no need for a person to look for errors and anomalies in the test run.

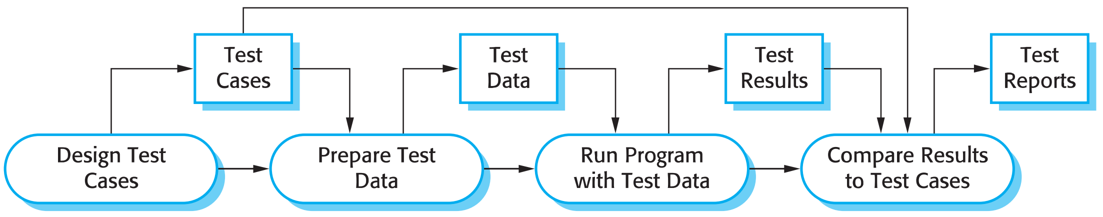

Figure 8.3 A model of the software testing process

Typically, a commercial software system has to go through three stages of testing:

1. Development testing, where the system is tested during development to discover bugs and defects. System designers and programmers are likely to be involved in the testing process.

2. Release testing, where a separate testing team tests a complete version of the system before it is released to users. The aim of release testing is to check that the system meets the requirements of system stakeholders.

3. User testing, where users or potential users of a system test the system in their own environment. For software products, the 'user' may be an internal marketing group who decide if the software can be marketed, released, and sold. Acceptance testing is one type of user testing where the customer formally tests a system to decide if it should be accepted from the system supplier or if further development is required.

In practice, the testing process usually involves a mixture of manual and automated testing. In manual testing, a tester runs the program with some test data and compares the results to their expectations. They note and report discrepancies to the program developers. In automated testing, the tests are encoded in a program that is run each time the system under development is to be tested. This is usually faster than manual testing, especially when it involves regression testing—re-running previous tests to check that changes to the program have not introduced new bugs.

The use of automated testing has increased considerably over the past few years. However, testing can never be completely automated as automated tests can only check that a program does what it is supposed to do. It is practically impossible to use automated testing to test systems that depend on how things look (e.g., a graphical user interface), or to test that a program does not have unwanted side effects.

8.1 Development testing

Development testing includes all testing activities that are carried out by the team developing the system. The tester of the software is usually the programmer who developed that software, although this is not always the case. Some development processes use programmer/tester pairs (Cusamano and Selby, 1998) where each

Debugging

Debugging is the process of fixing errors and problems that have been discovered by testing. Using information from the program tests, debuggers use their knowledge of the programming language and the intended outcome of the test to locate and repair the program error. This process is often supported by interactive debugging tools that provide extra information about program execution.

http://www.SoftwareEngineering-9.com/Web/Testing/Debugging.html

programmer has an associated tester who develops tests and assists with the testing process. For critical systems, a more formal process may be used, with a separate testing group within the development team. They are responsible for developing tests and maintaining detailed records of test results.

During development, testing may be carried out at three levels of granularity:

1. Unit testing, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
2. Component testing, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
3. System testing, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

Development testing is primarily a defect testing process, where the aim of testing is to discover bugs in the software. It is therefore usually interleaved with debugging—the process of locating problems with the code and changing the program to fix these problems.

8.1.1 Unit testing

Unit testing is the process of testing program components, such as methods or object classes. Individual functions or methods are the simplest type of component. Your tests should be calls to these routines with different input parameters. You can use the approaches to test case design discussed in Section 8.1.2, to design the function or method tests.

When you are testing object classes, you should design your tests to provide coverage of all of the features of the object. This means that you should:

- test all operations associated with the object;
- set and check the value of all attributes associated with the object;
- put the object into all possible states. This means that you should simulate all events that cause a state change.

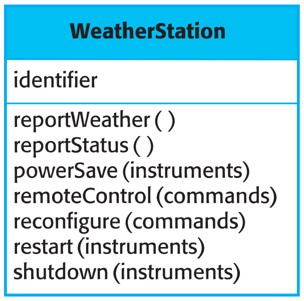

Figure 8.4 The weather station object interface

Consider, for example, the weather station object from the example that I discussed in Chapter 7. The interface of this object is shown in Figure 8.4. It has 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 properly set up. You need to define test cases for all of the methods associated with the object such as reportWeather, reportStatus, etc. Ideally, you should test methods in isolation but, in some cases, some test sequences are necessary. For example, to test the method that shuts down the weather station instruments (shutdown), you need to have executed the restart method.

Generalization or inheritance makes object class testing more complicated. You can't simply test an operation in the class where it is defined and assume that it will work as expected in the subclasses that inherit the operation. The operation that is inherited may make assumptions about other operations and attributes. These may not be valid in some subclasses that inherit the operation. You therefore have to test the inherited operation in all of the contexts where it is used.

To test the states of the weather station, you use a state model, such as the one shown in Figure 7.8 in the previous chapter. 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, 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 → Running → Shutdown
Configuring → Running → Testing → Transmitting → Running
Running → Collecting → Running → Summarizing → Transmitting → Running

Whenever possible, you should automate unit testing. In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests. Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success or failure of the tests. An entire test suite can often be run in a few seconds so it is possible to execute all the tests every time you make a change to the program.

An automated test has three parts:

1. A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.

2. A call part, where you call the object or method to be tested.
3. An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful; if false, then it has failed.

Sometimes the object that you are testing has dependencies on other objects that may not have been written or which slow down the testing process if they are used. For example, if your object calls a database, this may involve a slow setup process before it can be used. In these cases, you may decide to use mock objects. Mock objects are objects with the same interface as the external objects being used that simulate its functionality. Therefore, a mock object simulating a database may have only a few data items that are organized in an array. They can therefore be accessed quickly, without the overheads of calling a database and accessing disks. Similarly, mock objects can be used to simulate abnormal operation or rare events. For example, if your system is intended to take action at certain times of day, your mock object can simply return those times, irrespective of the actual clock time.

8.1.2 Choosing unit test cases

Testing is expensive and time consuming, so it is important that you choose effective unit test cases. Effectiveness, in this case, means two things:

1. The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
2. If there are defects in the component, these should be revealed by test cases.

You should therefore write two kinds of test case. The first of these should reflect normal operation of a program and should show that the component works. For example, if you are testing a component that creates and initializes a new patient record, then your test case should show that the record exists in a database and that its fields have been set as specified. The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

I discuss two possible strategies here that can be effective in helping you choose test cases. These are:

1. Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way. You should choose tests from within each of these groups.
2. Guideline-based testing, where you use testing guidelines to choose test cases. These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

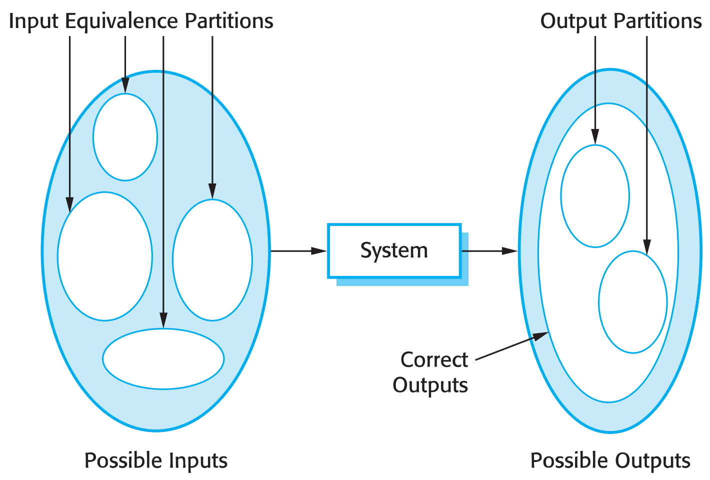

Figure 8.5 Equivalence partitioning

The input data and output results of a program often fall into a number of different classes with common characteristics. Examples of these classes are positive numbers, negative numbers, and menu selections. Programs normally behave in a comparable way for all members of a class. That is, if you test a program that does a computation and requires two positive numbers, then you would expect the program to behave in the same way for all positive numbers.

Because of this equivalent behavior, these classes are sometimes called equivalence partitions or domains (Bezier, 1990). One systematic approach to test case design is based on identifying all input and output partitions for a system or component. Test cases are designed so that the inputs or outputs lie within these partitions. Partition testing can be used to design test cases for both systems and components.

In Figure 8.5, the large shaded ellipse on the left represents the set of all possible inputs to the program that is being tested. The smaller unshaded ellipses represent equivalence partitions. A program being tested should process all of the members of an input equivalence partitions in the same way. Output equivalence partitions are partitions within which all of the outputs have something in common. Sometimes there is a 1:1 mapping between input and output equivalence partitions. However, this is not always the case; you may need to define a separate input equivalence partition, where the only common characteristic of the inputs is that they generate outputs within the same output partition. The shaded area in the left ellipse represents inputs that are invalid. The shaded area in the right ellipse represents exceptions that may occur (i.e., responses to invalid inputs).

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

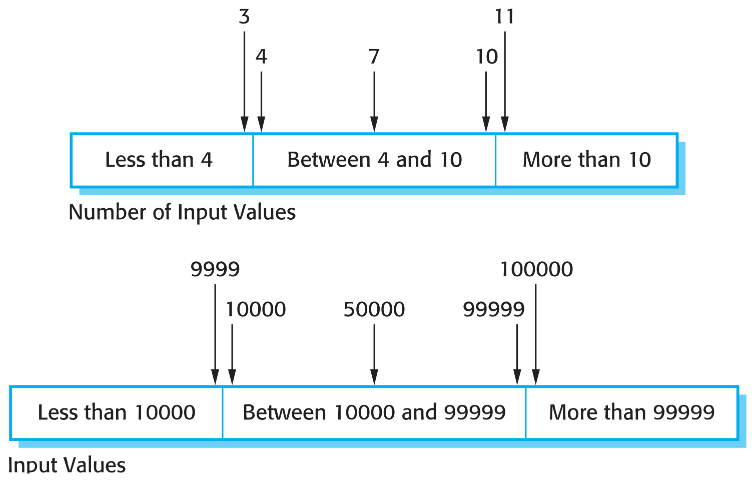

Figure 8.6 Equivalence partitions

You identify partitions by using the program specification or user documentation and from experience where you predict the classes of input value that are likely to detect errors. For example, say a program specification states that the program accepts 4 to 8 inputs which are five-digit integers greater than 10,000. You use this information to identify the input partitions and possible test input values. These are shown in Figure 8.6.

When you use the specification of a system to identify equivalence partitions, this is called 'black-box testing'. Here, you don't need any knowledge of how the system works. However, it may be helpful to supplement the black-box tests with 'white-box testing', where you look at the code of the program to find other possible tests. For example, your code may include exceptions to handle incorrect inputs. You can use this knowledge to identify 'exception partitions'—different ranges where the same exception handling should be applied.

Equivalence partitioning is an effective approach to testing because it helps account for errors that programmers often make when processing inputs at the edges of partitions. You can also use testing guidelines to help choose test cases. Guidelines encapsulate knowledge of what kinds of test cases are effective for discovering errors. For example, when you are testing programs with sequences, arrays, or lists, guidelines that could help reveal defects include:

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, if presented with a single-value sequence, a program may not work properly.

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 is reveals problems at partition boundaries.

Path testing

Path testing is a testing strategy that aims 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. All conditional statements are tested for both true and false cases. In an object-oriented development process, path testing may be used when testing the methods associated with objects.

http://www.SoftwareEngineering-9.com/Web/Testing/PathTest.html

Whittaker's book (2002) includes many examples of guidelines that can be used in test case design. Some of the most general guidelines that he suggests are:

- Choose inputs that force the system to generate all error messages;
- Design inputs that cause input buffers to overflow;
- Repeat the same input or series of inputs numerous times;
- Force invalid outputs to be generated;
- Force computation results to be too large or too small.

As you gain experience with testing, you can develop your own guidelines about how to choose effective test cases. I give more examples of testing guidelines in the next section of this chapter.

8.1.3 Component testing

Software components are often composite components that are made up of several interacting objects. For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration. You access the functionality of these objects through the defined component interface. Testing composite components should therefore focus on showing that the component interface behaves according to its specification. You can assume that unit tests on the individual objects within the component have been completed.

Figure 8.7 illustrates the idea of component interface testing. Assume that components A, B, and C have been integrated to create a larger component or subsystem. The test cases are not applied to the individual components but rather to the interface of the composite component created by combining these components. Interface errors in the composite component may not be detectable by testing the individual objects because these errors result from interactions between the objects in the component.

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

1. *Parameter interfaces* These are interfaces in which data or sometimes function references are passed from one component to another. Methods in an object have a parameter interface.