The Contents of the syllabus for Unit II of Advanced Software Engineering (MCA-303-CR) has been in toto adopted from Chapter 6 of the book entitled "***Practical Software Testing – A Process Oriented Approach***" by Ilene Burnstien published by Springer. All the pages in this pdf have been extracted from the same book without any modification. The pdf should as such be used only for educational purposes and any credit/reference for the scholarly work should be given to author of the book.

Burnstein, I. (2006). *Practical software testing: a process-oriented approach*. Springer Science & Business Media.

In case of any problem / clarification please mail me at:

suf[dot]cs[at]uok.[dot]edu[dot]in

LEVELS OF TESTING

6.0 The Need for Levels of Testing

Execution-based software testing, especially for large systems, is usually carried out at different levels. In most cases there will be 3–4 levels, or major phases of testing: unit test, integration test, system test, and some type of acceptance test as shown in Figure 6.1. Each of these may consist of one or more sublevels or phases. At each level there are specific testing goals. For example, at unit test a single component is tested. A principal goal is to detect functional and structural defects in the unit. At the integration level several components are tested as a group, and the tester investigates component interactions. At the system level the system as a whole is tested and a principle goal is to evaluate attributes such as usability, reliability, and performance.

An orderly progression of testing levels is described in this chapter for both object-oriented and procedural-based software systems. The major testing levels for both types of system are similar. However, the nature of the code that results from each developmental approach demands dif-

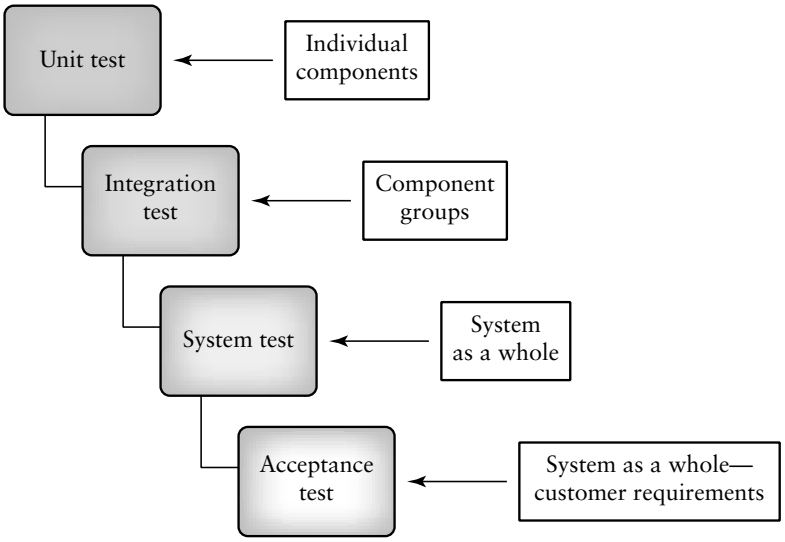


FIG. 6.1
Levels of testing.

ferent testing strategies, for example, to identify individual components, and to assemble them into subsystems. The issues involved are described in Sections 6.0.1, 6.1, and 6.2.3 of this chapter. For both types of systems the testing process begins with the smallest units or components to identify functional and structural defects. Both white and black box test strategies as discussed in Chapters 4 and 5 can be used for test case design at this level. After the individual components have been tested, and any necessary repairs made, they are integrated to build subsystems and clusters. Testers check for defects and adherence to specifications. Proper interaction at the component interfaces is of special interest at the integration level. In most cases black box design predominates, but often white box tests are used to reveal defects in control and data flow between the integrated modules.

System test begins when all of the components have been integrated successfully. It usually requires the bulk of testing resources. Laboratory equipment, special software, or special hardware may be necessary, especially for real-time, embedded, or distributed systems. At the system level the tester looks for defects, but the focus is on evaluating performance, usability, reliability, and other quality-related requirements.

If the system is being custom made for an individual client then the next step following system test is acceptance test. This is a very important testing stage for the developers. During acceptance test the development organization must show that the software meets all of the client's requirements. Very often final payments for system development depend on the quality of the software as observed during the acceptance test. A successful acceptance test provides a good opportunity for developers to request recommendation letters from the client. Software developed for the mass market (i.e., shrink-wrapped software) often goes through a series of tests called alpha and beta tests. Alpha tests bring potential users to the developer's site to use the software. Developers note any problems. Beta tests send the software out to potential users who use it under real-world conditions and report defects to the developing organization.

Implementing all of these levels of testing require a large investment in time and organizational resources. Organizations with poor testing processes tend to skimp on resources, ignore test planning until code is close to completion, and omit one or more testing phases. This seldom works to the advantage of the organization or its customers. The software released under these circumstances is often of poor quality, and the additional costs of repairing defects in the field, and of customer dissatisfaction are usually under estimated.

6.0.1 Levels of Testing and Software Development Paradigms

The approach used to design and develop a software system has an impact on how testers plan and design suitable tests. There are two major approaches to system development—bottom-up, and top-down. These approaches are supported by two major types of programming languages—procedure-oriented and object-oriented. This chapter considers testing at different levels for systems developed with both approaches using either traditional procedural programming languages or object-oriented programming languages. The different nature of the code produced requires testers to use different strategies to identify and test components and component groups. Systems developed with procedural languages are generally viewed as being composed of passive data and active procedures. When test cases are developed the focus is on generating input data to pass to the procedures (or functions) in order to reveal defects. Object-

oriented systems are viewed as being composed of active data along with allowed operations on that data, all encapsulated within a unit similar to an abstract data type. The operations on the data may not be called upon in any specific order. Testing this type of software means designing an order of calls to the operations using various parameter values in order to reveal defects. Issues related to inheritance of operations also impact on testing.

Levels of abstraction for the two types of systems are also somewhat different. In traditional procedural systems, the lowest level of abstraction is described as a function or a procedure that performs some simple task. The next higher level of abstraction is a group of procedures (or functions) that call one another and implement a major system requirement. These are called subsystems. Combining subsystems finally produces the system as a whole, which is the highest level of abstraction. In object-oriented systems the lowest level is viewed by some researchers as the method or member function [1–3]. The next highest level is viewed as the class that encapsulates data and methods that operate on the data [4]. To move up one more level in an object-oriented system some researchers use the concept of the cluster, which is a group of cooperating or related classes [3,5]. Finally, there is the system level, which is a combination of all the clusters and any auxiliary code needed to run the system [3]. Not all researchers in object-oriented development have the same view of the abstraction levels, for example, Jorgensen describes the thread as a highest level of abstraction [1]. Differences of opinion will be described in other sections of this chapter.

While approaches for testing and assembling traditional procedural type systems are well established, those for object-oriented systems are still the subject of ongoing research efforts. There are different views on how unit, integration, and system tests are best accomplished in object-oriented systems. When object-oriented development was introduced key beneficial features were encapsulation, inheritance, and polymorphism. It was said that these features would simplify design and development and encourage reuse. However, testing of object-oriented systems is not straightforward due to these same features. For example, encapsulation can hide details from testers, and that can lead to uncovered code. Inheritance also presents many testing challenges, among those the retesting of inherited methods when they are used by a subclass in a different context.

It is also difficult to define a unit for object-oriented code. Some researchers argue for the method (member function) as the unit since it is procedurelike. However, some methods are very small in size, and developing test harnesses to test each individually requires a large overhead. Should a single class be a unit? If so, then the tester need to consider the complexity of the test harness needed to test the unit since in many cases, a particular class depends on other classes for its operation. Also, object-oriented code is characterized by use of messages, dynamic binding, state changes, and nonhierarchical calling relationships. This also makes testing more complex. The reader should understand that many of these issues are yet to be resolved. Subsequent sections in this chapter will discuss several of these issues using appropriate examples. References 1–9 represent some of the current research views in this area.

6.1 Unit Test: Functions, Procedures, Classes, and Methods as Units

A workable definition for a software unit is as follows:

A unit is the smallest possible testable software component.

It can be characterized in several ways. For example, a unit in a typical procedure-oriented software system:

- performs a single cohesive function;
- can be compiled separately;
- is a task in a work breakdown structure (from the manager's point of view);
- contains code that can fit on a single page or screen.

A unit is traditionally viewed as a function or procedure implemented in a procedural (imperative) programming language. In object-oriented systems both the method and the class/object have been suggested by researchers as the choice for a unit [1–5]. The relative merits of each of these as the selected component for unit test are described in sections that follow. A unit may also be a small-sized COTS component purchased from an outside vendor that is undergoing evaluation by the purchaser,

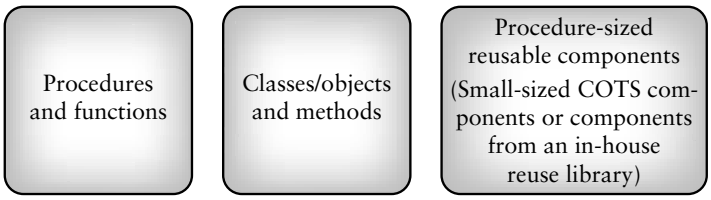


Fig. 6.2
Some components suitable for unit test.

or a simple module retrieved from an in-house reuse library. These unit types are shown in Figure 6.2.

No matter which type of component is selected as the smallest testable component, unit test is a vital testing level. Since the software component being tested is relatively small in size and simple in function, it is easier to design, execute, record, and analyze test results. If a defect is revealed by the tests it is easier to locate and repair since only the one unit is under consideration.

6.2 Unit Test: The Need for Preparation

The principal goal for unit testing is insure that each individual software unit is functioning according to its specification. Good testing practice calls for unit tests that are planned and public. Planning includes designing tests to reveal defects such as functional description defects, algorithmic defects, data defects, and control logic and sequence defects. Resources should be allocated, and test cases should be developed, using both white and black box test design strategies. The unit should be tested by an independent tester (someone other than the developer) and the test results and defects found should be recorded as a part of the unit history (made public). Each unit should also be reviewed by a team of reviewers, preferably before the unit test.

Unfortunately, unit test in many cases is performed informally by the unit developer soon after the module is completed, and it compiles cleanly. Some developers also perform an informal review of the unit. Under these circumstances the review and testing approach may be ad

hoc. Defects found are often not recorded by the developer; they are private (not public), and do not become a part of the history of the unit. This is poor practice, especially if the unit performs mission or safely critical tasks, or is intended for reuse.

To implement best practices it is important to plan for, and allocate resources to test each unit. If defects escape detection in unit test because of poor unit testing practices, they are likely to show up during integration, system, or acceptance test where they are much more costly to locate and repair. In the worst-case scenario they will cause failures during operation requiring the development organization to repair the software at the clients' site. This can be very costly.

To prepare for unit test the developer/tester must perform several tasks. These are:

(**i**) plan the general approach to unit testing;
(**ii**) design the test cases, and test procedures (these will be attached to the test plan);
(**iii**) define relationships between the tests;
(**iv**) prepare the auxiliary code necessary for unit test.

The text sections that follow describe these tasks in detail.

6.3 Unit Test Planning

A general unit test plan should be prepared. It may be prepared as a component of the master test plan or as a stand-alone plan. It should be developed in conjunction with the master test plan and the project plan for each project. Documents that provide inputs for the unit test plan are the project plan, as well the requirements, specification, and design documents that describe the target units. Components of a unit test plan are described in detail the *IEEE Standard for Software Unit Testing* [10]. This standard is rich in information and is an excellent guide for the test planner. A brief description of a set of development phases for unit test planning is found below. In each phase a set of activities is assigned based on those found in the IEEE unit test standard [10]. The phases allow a steady evolution of the unit test plan as more information becomes avail-

able. The reader will note that the unit test plan contains many of the same components as the master test plan that will be described in Chapter 7. Also note that a unit test plan is developed to cover all the units within a software project; however, each unit will have its own associated set of tests.

Phase 1: Describe Unit Test Approach and Risks

In this phase of unit testing planning the general approach to unit testing is outlined. The test planner:

(i) identifies test risks;
(ii) describes techniques to be used for designing the test cases for the units;
(iii) describes techniques to be used for data validation and recording of test results;
(iv) describes the requirements for test harnesses and other software that interfaces with the units to be tested, for example, any special objects needed for testing object-oriented units.

During this phase the planner also identifies completeness requirements—what will be covered by the unit test and to what degree (states, functionality, control, and data flow patterns). The planner also identifies termination conditions for the unit tests. This includes coverage requirements, and special cases. Special cases may result in abnormal termination of unit test (e.g., a major design flaw). Strategies for handling these special cases need to be documented. Finally, the planner estimates resources needed for unit test, such as hardware, software, and staff, and develops a tentative schedule under the constraints identified at that time.

Phase 2: Identify Unit Features to be Tested

This phase requires information from the unit specification and detailed design description. The planner determines which features of each unit will be tested, for example: functions, performance requirements, states, and state transitions, control structures, messages, and data flow patterns. If some features will not be covered by the tests, they should be mentioned

and the risks of not testing them be assessed. Input/output characteristics associated with each unit should also be identified, such as variables with an allowed ranges of values and performance at a certain level.

Phase 3: Add Levels of Detail to the Plan

In this phase the planner refines the plan as produced in the previous two phases. The planner adds new details to the approach, resource, and scheduling portions of the unit test plan. As an example, existing test cases that can be reused for this project can be identified in this phase. Unit availability and integration scheduling information should be included in the revised version of the test plan. The planner must be sure to include a description of how test results will be recorded. Test-related documents that will be required for this task, for example, test logs, and test incident reports, should be described, and references to standards for these documents provided. Any special tools required for the tests are also described.

The next steps in unit testing consist of designing the set of test cases, developing the auxiliary code needed for testing, executing the tests, and recording and analyzing the results. These topics will be discussed in Sections 6.4–6.6.

6.4 Designing the Unit Tests

Part of the preparation work for unit test involves unit test design. It is important to specify (i) the test cases (including input data, and expected outputs for each test case), and, (ii) the test procedures (steps required run the tests). These items are described in more detail in Chapter 7. Test case data should be tabularized for ease of use, and reuse. Suitable tabular formats for test cases are found in Chapters 4 and 5. To specifically support object-oriented test design and the organization of test data, Berard has described a test case specification notation [8]. He arranges the components of a test case into a semantic network with parts, Object_ID, Test_Case_ID, Purpose, and List_of_Test_Case_Steps. Each of these items has component parts. In the test design specification Berard also

includes lists of relevant states, messages (calls to methods), exceptions, and interrupts.

As part of the unit test design process, developers/testers should also describe the relationships between the tests. Test suites can be defined that bind related tests together as a group. All of this test design information is attached to the unit test plan. Test cases, test procedures, and test suites may be reused from past projects if the organization has been careful to store them so that they are easily retrievable and reusable.

Test case design at the unit level can be based on use of the black and white box test design strategies described in Chapters 4 and 5. Both of these approaches are useful for designing test cases for functions and procedures. They are also useful for designing tests for the individual methods (member functions) contained in a class. Considering the relatively small size of a unit, it makes sense to focus on white box test design for procedures/functions and the methods in a class. This approach gives the tester the opportunity to exercise logic structures and/or data flow sequences, or to use mutation analysis, all with the goal of evaluating the structural integrity of the unit. Some black box–based testing is also done at unit level; however, the bulk of black box testing is usually done at the integration and system levels and beyond. In the case of a smaller-sized COTS component selected for unit testing, a black box test design approach may be the only option. It should be mentioned that for units that perform mission/safely/business critical functions, it is often useful and prudent to design stress, security, and performance tests at the unit level if possible. (These types of tests are discussed in latter sections of this chapter.) This approach may prevent larger scale failures at higher levels of test.

6.5 The Class as a Testable Unit: Special Considerations

If an organization is using the object-oriented paradigm to develop software systems it will need to select the component to be considered for unit test. As described in Section 6.1, the choices consist of either the individual method as a unit or the class as a whole. Each of these choices requires special consideration on the part of the testers when designing and running the unit tests, and when retesting needs to be done. For example, in the case of the method as the selected unit to test, it may call

other methods within its own class to support its functionality. Additional code, in the form of a test harness, must be built to represent the called methods within the class. Building such a test harness for each individual method often requires developing code equivalent to that already existing in the class itself (all of its other methods). This is costly; however, the tester needs to consider that testing each individual method in this way helps to ensure that all statements/branches have been executed at least once, and that the basic functionality of the method is correct. This is especially important for mission or safety critical methods.

In spite of the potential advantages of testing each method individually, many developers/testers consider the class to be the component of choice for unit testing. The process of testing classes as units is sometimes called component test [11]. A class encapsulates multiple interacting methods operating on common data, so what we are testing is the intra-class interaction of the methods. When testing on the class level we are able detect not only traditional types of defects, for example, those due to control or data flow errors, but also defects due to the nature of object-oriented systems, for example, defects due to encapsulation, inheritance, and polymorphism errors. We begin to also look for what Chen calls object management faults, for example, those associated with the instantiation, storage, and retrieval of objects [12].

This brief discussion points out some of the basic trade-offs in selecting the component to be considered for a unit test in object-oriented systems. If the class is the selected component, testers may need to address special issues related to the testing and retesting of these components. Some of these issues are raised in the paragraphs that follow.

Issue 1: Adequately Testing Classes

The potentially high costs for testing each individual method in a class have been described. These high costs will be particularly apparent when there are many methods in a class; the numbers can reach as high as 20 to 30. If the class is selected as the unit to test, it is possible to reduce these costs since in many cases the methods in a single class serve as drivers and stubs for one another. This has the effect of lowering the complexity of the test harness that needs to be developed. However, in some cases driver classes that represent outside classes using the methods of the class under test will have to be developed.

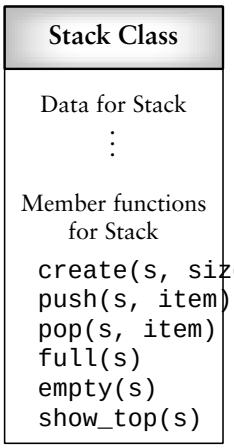


Fig. 6.3
Sample stack class with multiple methods.

In addition, if it is decided that the class is the smallest component to test, testers must decide if they are able to adequately cover all necessary features of each method in class testing. Some researchers believe that coverage objectives and test data need to be developed for each of the methods, for example, the *create, pop, push, empty, full,* and *show_top* methods associated with the stack class shown in Figure 6.3. Other researchers believe that a class can be adequately tested as a whole by observation of method interactions using a sequence of calls to the member functions with appropriate parameters.

Again, referring to the stack class shown in Figure 6.3, the methods *push, pop, full, empty,* and *show_top* will either read or modify the state of the stack. When testers unit (or component) test this class what they will need to focus on is the operation of each of the methods in the class and the interactions between them. Testers will want to determine, for example, if *push* places an item in the correct position at the top of the stack. However, a call to the method *full* may have to be made first to determine if the stack is already full. Testers will also want to determine if *push* and *pop* work together properly so that the stack pointer is in the correct position after a sequence of calls to these methods. To properly test this class, a sequence of calls to the methods needs to be specified as

part of component test design. For example, a test sequence for a stack that can hold three items might be:

create(s,3), empty(s), push(s,item-1), push(s,item-2), push(s,item-3), full(s), show_top(s), pop(s,item), pop(s,item), pop(s,item), empty(s), . . .

The reader will note that many different sequences and combination of calls are possible even for this simple class. Exhaustively testing every possible sequence is usually not practical. The tester must select those sequences she believes will reveal the most defects in the class. Finally, a tester might use a combination of approaches, testing some of the critical methods on an individual basis as units, and then testing the class as a whole.

Issue 2: Observation of Object States and State Changes

Methods may not return a specific value to a caller. They may instead change the state of an object. The state of an object is represented by a specific set of values for its attributes or state variables. State-based testing as described in Chapter 4 is very useful for testing objects. Methods will often modify the state of an object, and the tester must ensure that each state transition is proper. The test designer can prepare a state table (using state diagrams developed for the requirements specification) that specifies states the object can assume, and then in the table indicate sequence of messages and parameters that will cause the object to enter each state. When the tests are run the tester can enter results in this same type of table. For example, the first call to the method *push* in the stack class of Figure 6.3, changes the state of the stack so that *empty* is no longer true. It also changes the value of the stack pointer variable, *top*. To determine if the method *push* is working properly the value of the variable *top* must be visible both before and after the invocation of this method. In this case the method *show_top* within the class may be called to perform this task. The methods *full* and *empty* also probe the state of the stack. A sample augmented sequence of calls to check the value of *top* and the *full/empty* state of the three-item stack is:

empty(s), push(s,item-1), show_top(s), push(s,item-2), show_top(s), push(s,item-3), full(s), show_top(s), pop(s,item), show_top(s), pop(s,item), show_top(s), empty(s), . . .

Test sequences also need to be designed to try to push an item on a full stack and pop an item from an empty stack. This could be done by adding first an extra *push* to the sequence of pushes, and in a separate test adding an extra *pop* to the sequence of pops.

In the case of the stack class, the class itself contains methods that can provide information about state changes. If this is not the case then additional classes/methods may have to be created to show changes of state. These would be part of the test harness. Another option is to include in each class methods that allows state changes to be observable. Testers should have the option of going back to the designers and requesting changes that make a class more testable. In any case, test planners should insure that code is available to display state variables, Test plans should provide resources for developing this type of code.

Issue 3: The Retesting of Classes—I

One of the most beneficial features of object-oriented development is encapsulation. This is a technique that can be used to hide information. A program unit, in this case a class, can be built with a well-defined public interface that proclaims its services (available methods) to client classes. The implementation of the services is private. Clients who use the services are unaware of implementation details. As long as the interface is unchanged, making changes to the implementation should not affect the client classes. A tester of object-oriented code would therefore conclude that only the class with implementation changes to its methods needs to be retested. Client classes using unchanged interfaces need not be retested. This is not necessarily correct, as Perry and Kaiser explain in their paper on adequate testing for object-oriented systems [13]. In an object-oriented system, if a developer changes a class implementation that class needs to be retested as well as all the classes that depend on it. If a superclass, for example, is changed, then it is necessary to retest all of its subclasses. In addition, when a new subclass is added (or modified), we must also retest the methods inherited from each of its ancestor superclasses. The new (or changed) subclass introduces an unexpected form of dependency because there now exists a new context for the inherited components. This is a consequence of the antidecomposition testing axiom as described in Chapter 5 [13].

Issue 4: The Retesting of Classes—II

Classes are usually a part of a class hierarchy where there are existing inheritance relationships. Subclasses inherit methods from their superclasses. Very often a tester may assume that once a method in a superclass has been tested, it does not need retested in a subclass that inherits it. However, in some cases the method is used in a different context by the subclass and will need to be retested.

In addition, there may be an overriding of methods where a subclass may replace an inherited method with a locally defined method. Not only will the new locally defined method have to be retested, but designing a new set of test cases may be necessary. This is because the two methods (inherited and new) may be structurally different. The antiextentionality axiom as discussed in Chapter 5 expresses this need [13].

The following is an example of such as case using the shape class in Figure 6.4. Suppose the shape superclass has a subclass, triangle, and triangle has a subclass, equilateral triangle. Also suppose that the method *display* in shape needs to call the method *color* for its operation. Equilateral triangle could have a local definition for the method *display*. That method could in turn use a local definition for *color* which has been defined in triangle. This local definition of the *color* method in triangle has been tested to work with the inherited *display* method in shape, but not with the locally defined *display* in equilateral triangle. This is a new context that must be retested. A set of new test cases should be developed. The tester must carefully examine all the relationships between members of a class to detect such occurrences.

Many authors have written about class testing and object-oriented testing in general. Some have already been referenced in this chapter. Others include Desouza, Perry, and Rangaraajan who discuss the issue of when the retesting of methods within classes and subclasses is necessary [2,13,14]. Smith, Wilde, Doong, McGregor, and Tsai describe frameworks and tools to assist with class test design [3,6,7,11,15]. Harrold and co-authors have written several papers that describe the application of data flow testing to object-oriented systems [16,17]. The authors use data flow techniques to test individual member functions and also to test interactions among member functions. Outside of class interactions are also covered by their approach. Finally, Kung has edited a book that contains

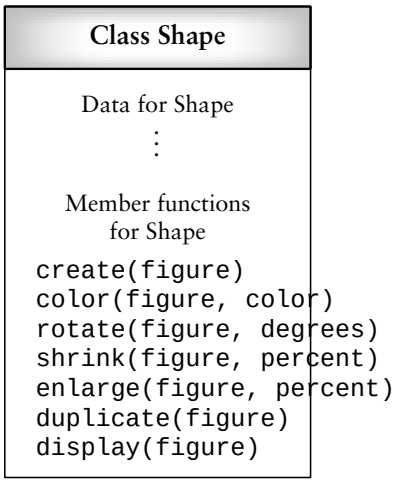


Fig. 6.4
Sample shape class.

key papers devoted to object-oriented testing. The papers discuss many of the above issues in detail [18].

6.6 The Test Harness

In addition to developing the test cases, supporting code must be developed to exercise each unit and to connect it to the outside world. Since the tester is considering a stand-alone function/procedure/class, rather than a complete system, code will be needed to call the target unit, and also to represent modules that are called by the target unit. This code called the test harness, is developed especially for test and is in addition to the code that composes the system under development. The role is of the test harness is shown in Figure 6.5 and it is defined as follows:

> **The auxiliary code developed to support testing of units and components is called a test harness. The harness consists of drivers that call the target code and stubs that represent modules it calls.**

The development of drivers and stubs requires testing resources. The drivers and stubs must be tested themselves to insure they are working prop-

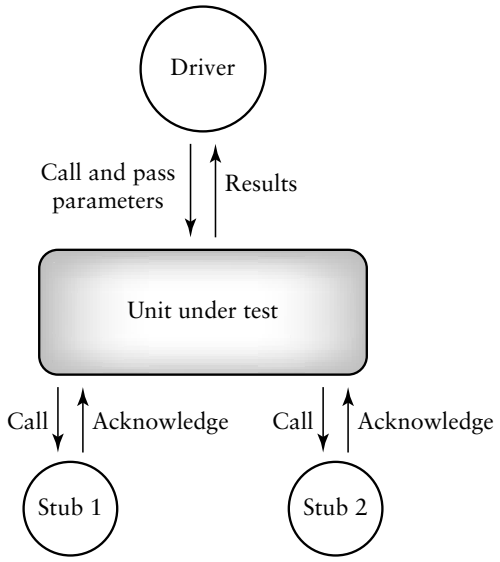


Fig. 6.5
The test harness.

erly and that they are reusable for subsequent releases of the software. Drivers and stubs can be developed at several levels of functionality. For example, a driver could have the following options and combinations of options:

(**i**) call the target unit;
(**ii**) do 1, and pass inputs parameters from a table;
(**iii**) do 1, 2, and display parameters;
(**iv**) do 1, 2, 3 and display results (output parameters).

The stubs could also exhibit different levels of functionality. For example a stub could:

(**i**) display a message that it has been called by the target unit;
(**ii**) do 1, and display any input parameters passed from the target unit;
(**iii**) do 1, 2, and pass back a result from a table;
(**iv**) do 1, 2, 3, and display result from table.

Drivers and stubs as shown in Figure 6.5 are developed as procedures and functions for traditional imperative-language based systems. For object-oriented systems, developing drivers and stubs often means the design and implementation of special classes to perform the required testing tasks. The test harness itself may be a hierarchy of classes. For example, in Figure 6.5 the driver for a procedural system may be designed as a single procedure or main module to call the unit under test; however, in an object-oriented system it may consist of several test classes to emulate all the classes that call for services in the class under test. Researchers such as Rangaraajan and Chen have developed tools that generate test cases using several different approaches, and classes of test harness objects to test object-oriented code [12,14].

The test planner must realize that, the higher the degree of functionally for the harness, the more resources it will require to design, implement, and test. Developers/testers will have to decide depending on the nature of the code under test, just how complex the test harness needs to be. Test harnesses for individual classes tend to be more complex than those needed for individual procedures and functions since the items being tested are more complex and there are more interactions to consider.

6.7 Running the Unit Tests and Recording Results

Unit tests can begin when (i) the units becomes available from the developers (an estimation of availability is part of the test plan), (ii) the test cases have been designed and reviewed, and (iii) the test harness, and any other supplemental supporting tools, are available. The testers then proceed to run the tests and record results. Chapter 7 will describe documents called test logs that can be used to record the results of specific tests. The status of the test efforts for a unit, and a summary of the test results, could be recorded in a simple format such as shown in Table 6.1. These forms can be included in the test summary report, and are of value at the weekly status meetings that are often used to monitor test progress.

It is very important for the tester at any level of testing to carefully record, review, and check test results. The tester must determine from the results whether the unit has passed or failed the test. If the test is failed, the nature of the problem should be recorded in what is sometimes called

Unit Test Worksheet

Unit Name: _____

Unit Identifier: _____

Tester: _____

Date: _____

Test case ID	Status (run/not run)	Summary of results	Pass/fail

TABLE 6.1

Summary work sheet for unit test results.

a test incident report (see Chapter 7). Differences from expected behavior should be described in detail. This gives clues to the developers to help them locate any faults. During testing the tester may determine that additional tests are required. For example, a tester may observe that a particular coverage goal has not been achieved. The test set will have to be augmented and the test plan documents should reflect these changes.

When a unit fails a test there may be several reasons for the failure. The most likely reason for the failure is a fault in the unit implementation (the code). Other likely causes that need to be carefully investigated by the tester are the following:

- a fault in the test case specification (the input or the output was not specified correctly);
- a fault in test procedure execution (the test should be rerun);
- a fault in the test environment (perhaps a database was not set up properly);
- a fault in the unit design (the code correctly adheres to the design specification, but the latter is incorrect).

The causes of the failure should be recorded in a test summary report, which is a summary of testing activities for all the units covered by the unit test plan.

Ideally, when a unit has been completely tested and finally passes all of the required tests it is ready for integration. Under some circumstances a unit may be given a conditional acceptance for integration test. This may occur when the unit fails some tests, but the impact of the failure is not significant with respect to its ability to function in a subsystem, and the availability of a unit is critical for integration test to proceed on schedule. This a risky procedure and testers should evaluate the risks involved. Units with a conditional pass must eventually be repaired.

When testing of the units is complete, a test summary report should be prepared. This is a valuable document for the groups responsible for integration and system tests. It is also a valuable component of the project history. Its value lies in the useful data it provides for test process improvement and defect prevention. Finally, the tester should insure that the test cases, test procedures, and test harnesses are preserved for future reuse.

6.8 Integration Test: Goals

Integration test for procedural code has two major goals:

(i) to detect defects that occur on the interfaces of units;
(ii) to assemble the individual units into working subsystems and finally a complete system that is ready for system test.

In unit test the testers attempt to detect defects that are related to the functionality and structure of the unit. There is some simple testing of unit interfaces when the units interact with drivers and stubs. However, the interfaces are more adequately tested during integration test when each unit is finally connected to a full and working implementation of those units it calls, and those that call it. As a consequence of this assembly or integration process, software subsystems and finally a completed system is put together during the integration test. The completed system is then ready for system testing.

With a few minor exceptions, integration test should only be performed on units that have been reviewed and have successfully passed unit testing. A tester might believe erroneously that since a unit has al-

ready been tested during a unit test with a driver and stubs, it does not need to be retested in combination with other units during integration test. However, a unit tested in isolation may not have been tested adequately for the situation where it is combined with other modules. This is also a consequences of one of the testing axioms found in Chapter 4 called anticomposition [13].

Integration testing works best as an iterative process in procedural-oriented system. One unit at a time is integrated into a set of previously integrated modules which have passed a set of integration tests. The interfaces and functionally of the new unit in combination with the previously integrated units is tested. When a subsystem is built from units integrated in this stepwise manner, then performance, security, and stress tests can be performed on this subsystem.

Integrating one unit at a time helps the testers in several ways. It keeps the number of new interfaces to be examined small, so tests can focus on these interfaces only. Experienced testers know that many defects occur at module interfaces. Another advantage is that the massive failures that often occur when multiple units are integrated at once is avoided. This approach also helps the developers; it allows defect search and repair to be confined to a small known number of components and interfaces. Independent subsystems can be integrated in parallel as long as the required units are available.

The integration process in object-oriented systems is driven by assembly of the classes into cooperating groups. The cooperating groups of classes are tested as a whole and then combined into higher-level groups. Usually the simpler groups are tested first, and then combined to form higher-level groups until the system is assembled. This process will be described in the next sections of this chapter.

6.9 Integration Strategies for Procedures and Functions

For conventional procedural/functional-oriented systems there are two major integration strategies—top-down and bottom-up. In both of these strategies only one module at a time is added to the growing subsystem. To plan the order of integration of the modules in such system a structure chart such as shown in Figure 6.6 is used.

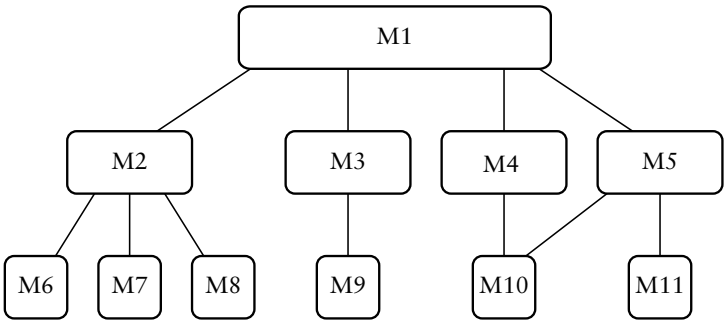


Fig. 6.6
Simple structure chart for integration test examples.

Structure charts, or call graphs as they are otherwise known, are used to guide integration. These charts show hierarchical calling relationships between modules. Each node, or rectangle in a structure chart, represents a module or unit, and the edges or lines between them represent calls between the units. In the simple chart in Figure 6.6 the rectangles M1–M11 represent all the system modules. Again, a line or edge from an upper-level module to one below it indicates that the upper level module calls the lower module. Some annotated versions of structure charts show the parameters passed between the caller and called modules. Conditional calls and iterative calls may also be represented.

Bottom-up integration of the modules begins with testing the lowest-level modules, those at the bottom of the structure chart. These are modules that do not call other modules. In the structure chart example these are modules M6, M7, M8, M9, M10, M11. Drivers are needed to test these modules. The next step is to integrate modules on the next upper level of the structure chart whose subordinate modules have already been tested. For example, if we have tested M6, M7, and M8, then we can select M2 and integrate it with M6, M7, and M8. The actual M2 replaces the drivers for these modules.

In the process for bottom-up integration after a module has been tested, its driver can be replaced by an actual module (the next one to be integrated). This next module to be integrated may also need a driver, and this will be the case until we reach the highest level of the structure chart. Accordingly we can integrate M9 with M3 when M9 is tested, and

M4 with M10 when M10 is tested, and finally M5 with M11 and M10 when they are both tested. Integration of the M2 and M3 subsystems can be done in parallel by two testers. The M4 and M5 subsystems have overlapping dependencies on M10. To complete the subsystem represented by M5, both M10 and M11 will have to be tested and integrated. M4 is only dependent on M10. A third tester could work on the M4 and M5 subsystems.

After this level of integration is completed, we can then move up a level and integrate the subsystem M2, M6, M7, and M8 with M1 when M2 has been completed tested with its subordinates, and driver. The same conditions hold for integrating the subsystems represented by M3, M9, M4, M10, and M5, M10, M11 with M1. In that way the system is finally integrated as a whole. In this example a particular sequence of integration has been selected. There are no firm rules for selecting which module to integrate next. However, a rule of thumb for bottom-up integration says that to be eligible for selection as the next candidate for integration, all of a module's subordinate modules (modules it calls) must have been tested previously. Issues such as the complexity, mission, or safety criticalness of a module also impact on the choices for the integration sequence.

Bottom-up integration has the advantage that the lower-level modules are usually well tested early in the integration process. This is important if these modules are candidates for reuse. However, the upper-level modules are tested later in the integration process and consequently may not be as well tested. If they are critical decision-makers or handle critical functions, this could be risky. In addition, with bottom-up integration the system as a whole does not exist until the last module, in our example, M1, is integrated. It is possible to assemble small subsystems, and when they are shown to work properly the development team often experiences a boost in morale and a sense of achievement.

Top-down integration starts at the top of the module hierarchy. The rule of thumb for selecting candidates for the integration sequence says that when choosing a candidate module to be integrated next, at least one of the module's superordinate (calling) modules must have been previously tested. In our case, M1 is the highest-level module and we start the sequence by developing stubs to test it. In order to get a good upward flow of data into the system, the stubs may have to be fairly complex (see

Section 6.2.3). The next modules to be integrated are those for whom their superordinate modules has been tested. The way to proceed is to replace one-by-one each of the stubs of the superordinate module with a subordinate module. For our example in Figure 6.6, we begin top-down integration with module M1. We create four stubs to represent M2, M3, M4, and M5. When the tests are passed, then we replace the four stubs by the actual modules one at a time. The actual modules M2–M5 when they are integrated will have stubs of their own. Figure 6.7 shows the set up for the integration of M1 with M2.

When we have integrated the modules M2–M5, then we can integrate the lowest-level modules. For example, when, M2 has been integrated with M1 we can replace its stubs for M6, M7, and M8 with the actual modules, one at a time, and so on for M3, M4, and M5. One can traverse the structure chart and integrate the modules in a depth or breadth-first manner. For example, the order of integration for a depth-first approach would be M1, M2, M6, M7, M8, M3, M9, M4, M10, M5, M11. Breadth-first would proceed as M1, M2, M3, M4, M5, M6, M7, M8, M9, M10, M11. Note that using the depth-first approach gradually forms subsystems as the integration progresses. In many cases these subsystems can be assembled and tested in parallel. For example, when the testing of M1 is completed, there could be parallel integration testing of subsystems M2 and M3. A third tester could work in parallel with these testers on the subsystems M4 and M5. The test planner should look for these opportunities when scheduling testing tasks.

Top-down integration ensures that the upper-level modules are tested early in integration. If they are complex and need to be redesigned there will be more time to do so. This is not the case with bottom-up integration. Top-down integration requires the development of complex stubs to drive significant data upward, but bottom-up integration requires drivers so there is not a clear-cut advantage with respect to developing test harnesses. In many cases a combination of approaches will be used. One approach is known as sandwich, where the higher-level modules are integrated top-down and the lower-level modules integrated bottom-up.

No matter which integration strategy is selected, testers should consider applying relevant coverage criteria to the integration process. Linnenkugel and Mullerburg have suggested several interprocedural control and data flow–based criteria [19]. Example control flow criteria include:

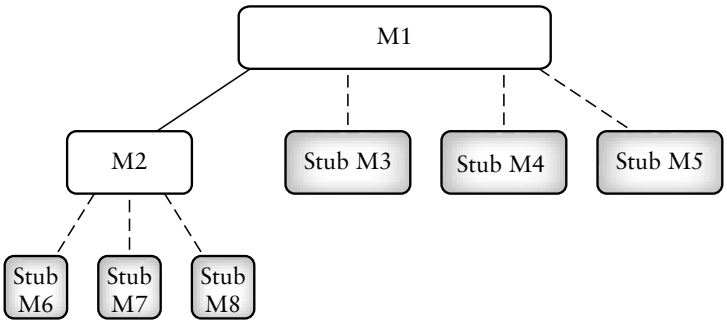


Fig. 6.7
Top-down integration of modules M1 and M2.

all modules in the graph or chart should be executed at least once (all nodes covered), all calls should be executed at least once (all edges covered), and all descending sequences of calls should be executed at least once (all paths covered).

The smart test planner takes into account risk factors associated with each of the modules, and plans the order of integration accordingly. Some modules and subsystems may be handling mission/safety/business critical functions; some might be complex. The test planner will want to be sure that these are assembled and tested early in the integration process to insure they are tested adequately. For example, in the sample structure chart shown in Figure 6.6, if modules M6 and M10 are complex and/or safety critical modules, a tester would consider bottom-up integration as a good choice since these modules would be integrated and tested early in the integration process.

Another area of concern for the planner is the availability of the modules. The test planner should consult the project plan to determine availability dates. Availability may also be affected during the testing process depending on the number and nature of the defects found in each module. A subsystem may be assembled earlier/later then planned depending in the amount of time it takes to test/repair its component modules. For example, we may be planning to integrate branch M2 before branch M3; however, M2 and its components may contain more defects then M3 and its components, so there will be a delay for repairs to be made.

6.10 Integration Strategies for Classes

For object-oriented systems the concept of a structure chart and hierarchical calling relationships are not applicable. Therefore, integration needs to proceed in a manner different from described previously. A good approach to integration of an object-oriented system is to make use of the concept of object clusters. Clusters are somewhat analogous to small subsystems in procedural-oriented systems.

A cluster consists of classes that are related, for example, they may work together (cooperate) to support a required functionality for the complete system.

Figure 6.8 shows a generic cluster that consists of four classes/objects interacting with one another, calling each others methods. For purposes of illustration we assume that they cooperate to perform functions whose result (Out message) is exported to the outside world. As another illustration of the cluster concept we can use the notion of an object-oriented system that manages a state vehicle licensing bureau. A high-level cluster of objects may be concerned with functions related to vehicle owners, while another cluster is concerned with functions relating to the vehicles themselves. Coad and Yourdon in their text on object-oriented analysis give examples of partitioning the objects in a system into what they call subject layers that are similar in concept to clusters. The partitioning is based on using problem area subdomains [20]. Subject layers can be identified during analysis and design and help to formulate plans for integration of the component classes.

To integrate an object-oriented system using the cluster approach a tester could select clusters of classes that work together to support simple functions as the first to be integrated. Then these are combined to form higher-level, or more complex, clusters that perform multiple related functions, until the system as a whole is assembled.

An alternative integration approach for object-oriented systems consists of first selecting classes for integration that send very few messages and/or request few, or no, services from other classes. After these lower-level classes are integrated, then a layer of classes that use them can be selected for integration, and so on until the successive selection of layers of classes leads to complete integration.

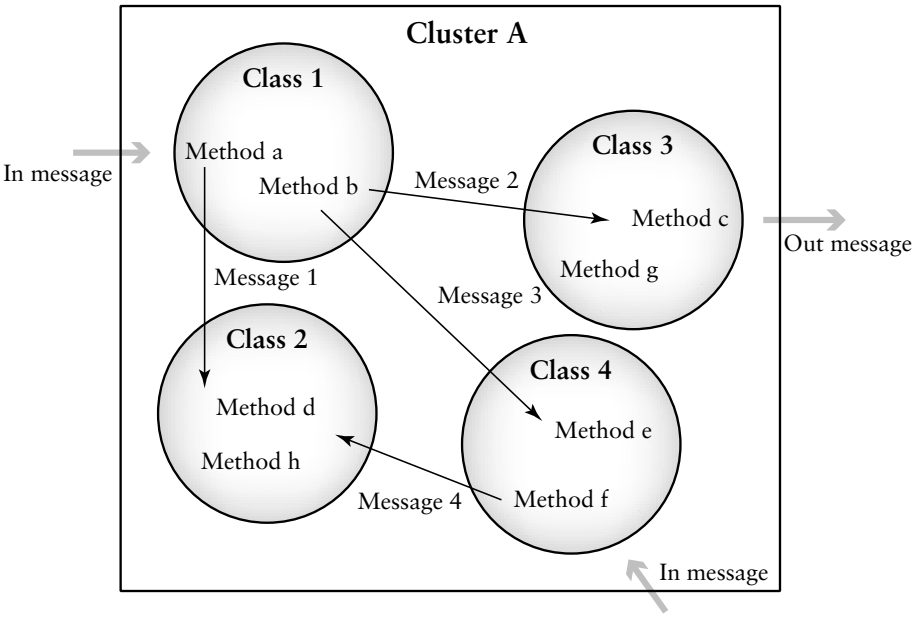


Fig. 6.8
An generic class cluster.

6.11 Designing Integration Tests

Integration tests for procedural software can be designed using a black or white box approach. Both are recommended. Some unit tests can be reused. Since many errors occur at module interfaces, test designers need to focus on exercising all input/output parameter pairs, and all calling relationships. The tester needs to insure the parameters are of the correct type and in the correct order. The author has had the personal experience of spending many hours trying to locate a fault that was due to an incorrect ordering of parameters in the calling routine. The tester must also insure that once the parameters are passed to a routine they are used correctly. For example, in Figure 6.9, Procedure_b is being integrated with Procedure_a. Procedure_a calls Procedure_b with two input parameters in3, in4. Procedure_b uses those parameters and then returns a value for the output parameter out1. Terms such as *lhs* and *rhs* could be any variable or expression. The reader should interpret the use of the variables in the broadest sense. The parameters could be involved in a number of *def* and/or *use* data flow patterns. The actual usage patterns of the parameters

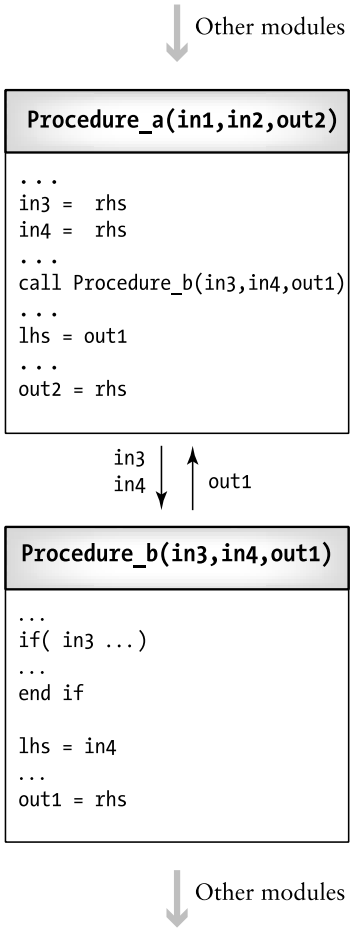


Fig. 6.9
Example integration of two procedures.

must be checked at integration time. Data flow–based (def-use paths) and control flow (branch coverage) test data generation methods are useful here to insure that the input parameters, in3, in4, are used properly in Procedure_b. Again data flow methods (def-use pairs) could also be used to check that the proper sequence of data flow operations is being carried out to generate the correct value for out1 that flows back to Procedure_a. Black box tests are useful in this example for checking the behavior of the pair of procedures. For this example test input values for the input parameters in1 and in2 should be provided, and the outcome in out2 should be examined for correctness.

For conventional systems, input/output parameters and calling relationships will appear in a structure chart built during detailed design.

Testers must insure that test cases are designed so that all modules in the structure chart are called at least once, and all called modules are called by every caller. The reader can visualize these as coverage criteria for integration test. Coverage requirements for the internal logic of each of the integrated units should be achieved during unit tests.

Some black box tests used for module integration may be reusable from unit testing. However, when units are integrated and subsystems are to be tested as a whole, new tests will have to be designed to cover their functionality and adherence to performance and other requirements (see example above). Sources for development of black box or functional tests at the integration level are the requirements documents and the user manual. Testers need to work with requirements analysts to insure that the requirements are testable, accurate, and complete. Black box tests should be developed to insure proper functionally and ability to handle subsystem stress. For example, in a transaction-based subsystem the testers want to determine the limits in number of transactions that can be handled. The tester also wants to observe subsystem actions when excessive amounts of transactions are generated. Performance issues such as the time requirements for a transaction should also be subjected to test. These will be repeated when the software is assembled as a whole and is undergoing system test.

Integration testing of clusters of classes also involves building test harnesses which in this case are special classes of objects built especially for testing. Whereas in class testing we evaluated intraclass method interactions, at the cluster level we test interclass method interaction as well. We want to insure that messages are being passed properly to interfacing objects, object state transitions are correct when specific events occur, and that the clusters are performing their required functions. Unlike procedural-oriented systems, integration for object-oriented systems usually does not occur one unit at a time. A group of cooperating classes is selected for test as a cluster. If developers have used the Coad and Yourdon's approach, then a subject layer could be used to represent a cluster. Jorgenson et al. have reported on a notation for a cluster that helps to formalize object-oriented integration [1]. In their object-oriented testing framework the method is the entity selected for unit test. The methods and the classes they belong to are connected into clusters of classes that are represented by a directed graph that has two special types of entities. These are method-message paths, and atomic systems functions that

represent input port events. A method-message path is described as a sequence of method executions linked by messages. An atomic system function is an input port event (start event) followed by a set of method-messages paths and terminated by an output port event (system response). Murphy et al. define clusters as classes that are closely coupled and work together to provide a unified behavior [5]. Some examples of clusters are groups of classes that produce a report, or monitor and control a device. Scenarios of operation from the design document associated with a cluster are used to develop test cases. Murphy and his co-authors have developed a tool that can be used for class and cluster testing.

6.12 Integration Test Planing

Integration test must be planned. Planning can begin when high-level design is complete so that the system architecture is defined. Other documents relevant to integration test planning are the requirements document, the user manual, and usage scenarios. These documents contain structure charts, state charts, data dictionaries, cross-reference tables, module interface descriptions, data flow descriptions, messages and event descriptions, all necessary to plan integration tests. The strategy for integration should be defined. For procedural-oriented system the order of integration of the units of the units should be defined. This depends on the strategy selected. Consider the fact that the testing objectives are to assemble components into subsystems and to demonstrate that the subsystem functions properly with the integration test cases. For object-oriented systems a working definition of a cluster or similar construct must be described, and relevant test cases must be specified. In addition, testing resources and schedules for integration should be included in the test plan.

For readers integrating object-oriented systems Murphy et al. has a detailed description of a Cluster Test Plan [5]. The plan includes the following items:

(**i**) clusters this cluster is dependent on;
(**ii**) a natural language description of the functionality of the cluster to be tested;
(**iii**) list of classes in the cluster;
(**iv**) a set of cluster test cases.

As stated earlier in this section, one of the goals of integration test is to build working subsystems, and then combine these into the system as a whole. When planning for integration test the planner selects subsystems to build based upon the requirements and user needs. Very often subsystems selected for integration are prioritized. Those that represent key features, critical features, and/or user-oriented functions may be given the highest priority. Developers may want to show clients that certain key subsystems have been assembled and are minimally functional. Hetzel has an outline for integration test planning that takes these requirements into consideration [21].

6.13 System Test: The Different Types

When integration tests are completed, a software system has been assembled and its major subsystems have been tested. At this point the developers/testers begin to test it as a whole. System test planning should begin at the requirements phase with the development of a master test plan and requirements-based (black box) tests. System test planning is a complicated task. There are many components of the plan that need to be prepared such as test approaches, costs, schedules, test cases, and test procedures. All of these are examined and discussed in Chapter 7.

System testing itself requires a large amount of resources. The goal is to ensure that the system performs according to its requirements. System test evaluates both functional behavior and quality requirements such as reliability, usability, performance and security. This phase of testing is especially useful for detecting external hardware and software interface defects, for example, those causing race conditions, deadlocks, problems with interrupts and exception handling, and ineffective memory usage. After system test the software will be turned over to users for evaluation during acceptance test or alpha/beta test. The organization will want to be sure that the quality of the software has been measured and evaluated before users/clients are invited to use the system. In fact system test serves as a good rehearsal scenario for acceptance test.

Because system test often requires many resources, special laboratory equipment, and long test times, it is usually performed by a team of testers. The best scenario is for the team to be part of an independent testing group. The team must do their best to find any weak areas in the software; therefore, it is best that no developers are directly involved.

There are several types of system tests as shown on Figure 6.10. The types are as follows:

- Functional testing
- Performance testing
- Stress testing
- Configuration testing
- Security testing
- Recovery testing

Two other types of system testing called reliability and usability testing will be discussed in Chapter 12. The TMM recommends that these be formally integrated into the testing process by organizations at higher levels of testing maturity since at that time they have the needed expertise and infrastructure to properly conduct the tests and analyze the results.

Not all software systems need to undergo all the types of system testing. Test planners need to decide on the type of tests applicable to a particular software system. Decisions depend on the characteristics of the system and the available test resources. For example, if multiple device configurations are not a requirement for your system, then the need for configuration test is not significant. Test resources can be used for other types of system tests. Figure 6.10 also shows some of the documents useful for system test design, such as the requirements document, usage profile, and user manuals. For both procedural- and object-oriented systems, use cases, if available, are also helpful for system test design.

As the system has been assembled from its component parts, many of these types of tests have been implemented on the component parts and subsystems. However, during system test the testers can repeat these tests and design additional tests for the system as a whole. The repeated tests can in some cases be considered regression tests since there most probably have been changes made to the requirements and to the system itself since the initiation of the project. A conscientious effort at system test is essential for high software quality. Properly planned and executed system tests are excellent preparation for acceptance test. The following

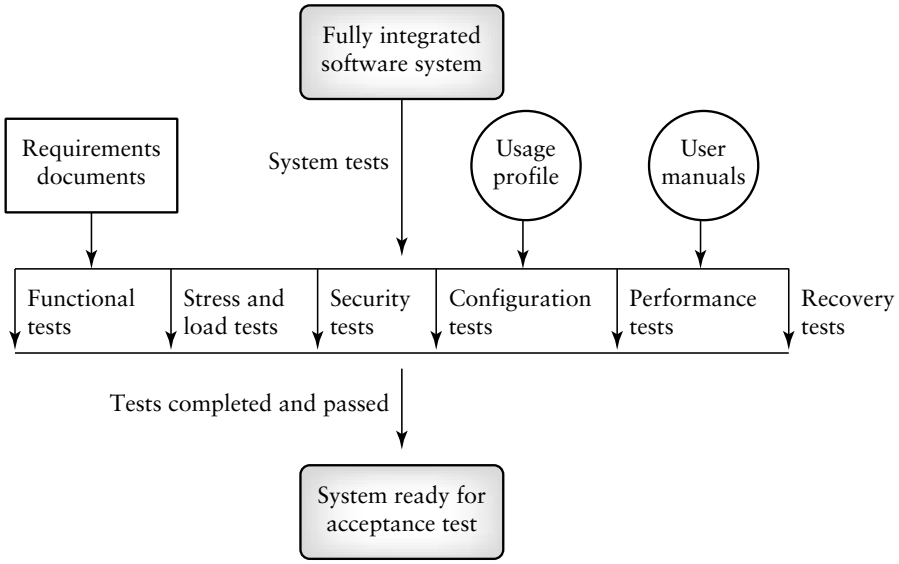


Fig. 6.10
Types of system tests.

sections will describe the types of system test. Beizer provides additional material on the different types of system tests [22].

Paper and on-line forms are helpful for system test. Some are used to insure coverage of all the requirements, for example, the Requirements Traceability Matrix, which is discussed in Chapter 7. Others, like test logs, also discussed in Chapter 7, support record keeping for test results. These forms should be fully described in the organization's standards documents.

An important tool for implementing system tests is a load generator. A load generator is essential for testing quality requirements such as performance and stress.

A load is a series of inputs that simulates a group of transactions.

A transaction is a unit of work seen from the system user's view [19]. A transaction consists of a set of operations that may be performed by a person, software system, or a device that is outside the system. A use case can be used to describe a transaction. If you were system testing a telecommunication system you would need a load that simulated a series of

phone calls (transactions) of particular types and lengths arriving from different locations. A load can be a real load, that is, you could put the system under test to real usage by having actual telephone users connected to it. Loads can also be produced by tools called load generators. They will generate test input data for system test. Load generators can be simple tools that output a fixed set of predetermined transactions. They can be complex tools that use statistical patterns to generate input data or simulate complex environments. Users of the load generators can usually set various parameters. For example, in our telecommunication system load generator users can set parameters for the mean arrival rate of the calls, the call durations, the number of wrong numbers and misdials, and the call destinations. Usage profiles, and sets of use cases can be used to set up loads for use in performance, stress, security and other types of system test.

6.13.1 Functional Testing

System functional tests have a great deal of overlap with acceptance tests. Very often the same test sets can apply to both. Both are demonstrations of the system's functionality. Functional tests at the system level are used to ensure that the behavior of the system adheres to the requirements specification. All functional requirements for the system must be achievable by the system. For example, if a personal finance system is required to allow users to set up accounts, add, modify, and delete entries in the accounts, and print reports, the function-based system and acceptance tests must ensure that the system can perform these tasks. Clients and users will expect this at acceptance test time.

Functional tests are black box in nature. The focus is on the inputs and proper outputs for each function. Improper and illegal inputs must also be handled by the system. System behavior under the latter circumstances tests must be observed. All functions must be tested.

Many of the system-level tests including functional tests should be designed at requirements time, and be included in the master and system test plans (see Chapter 7). However, there will be some requirements changes, and the tests and the test plan need to reflect those changes. Since functional tests are black box in nature, equivalence class partitioning and boundary-value analysis as described in Chapter 4 are useful

methods that can be used to generate test cases. State-based tests are also valuable. In fact, the tests should focus on the following goals.

- All types or classes of legal inputs must be accepted by the software.
- All classes of illegal inputs must be rejected (however, the system should remain available).
- All possible classes of system output must exercised and examined.
- All effective system states and state transitions must be exercised and examined.
- All functions must be exercised.

As mentioned previously, a defined and documented form should be used for recording test results from functional and all other system tests. If a failure is observed, a formal test incident report should be completed and returned with the test log to the developer for code repair. Managers keep track of these forms and reports for quality assurance purposes, and to track the progress of the testing process. Readers will learn more about these documents and their importance in Chapter 7.

6.13.2 Performance Testing

An examination of a requirements document shows that there are two major types of requirements:

1. *Functional requirements*. Users describe what functions the software should perform. We test for compliance of these requirements at the system level with the functional-based system tests.
2. *Quality requirements*. There are nonfunctional in nature but describe quality levels expected for the software. One example of a quality requirement is performance level. The users may have objectives for the software system in terms of memory use, response time, throughput, and delays.

The goal of system performance tests is to see if the software meets the performance requirements. Testers also learn from performance test

whether there are any hardware or software factors that impact on the system's performance. Performance testing allows testers to tune the system; that is, to optimize the allocation of system resources. For example, testers may find that they need to reallocate memory pools, or to modify the priority level of certain system operations. Testers may also be able to project the system's future performance levels. This is useful for planning subsequent releases.

Performance objectives must be articulated clearly by the users/clients in the requirements documents, and be stated clearly in the system test plan. The objectives must be quantified. For example, a requirement that the system return a response to a query in "a reasonable amount of time" is not an acceptable requirement; the time requirement must be specified in quantitative way. Results of performance tests are quantifiable. At the end of the tests the tester will know, for example, the number of CPU cycles used, the actual response time in seconds (minutes, etc.), the actual number of transactions processed per time period. These can be evaluated with respect to requirements objectives.

Resources for performance testing must be allocated in the system test plan. Examples of such resources are shown in Figure 6.11. Among the resources are:

- A source of transactions to drive the experiments. For example if you were performance testing an operating system you need a stream of data that represents typical user interactions. Typically the source of transaction for many systems is a load generator (as described in the previous section).
- An experimental testbed that includes hardware and software the system-under-test interacts with. The testbed requirements sometimes include special laboratory equipment and space that must be reserved for the tests.
- Instrumentation or probes that help to collect the performance data. Probes may be hardware or software in nature. Some probe tasks are event counting and event duration measurement. For example, if you are investigating memory requirements for your software you could use a hardware probe that collected information on memory usage (blocks allocated, blocks deallocated for different types of memory per unit time) as the system executes. The tester must keep

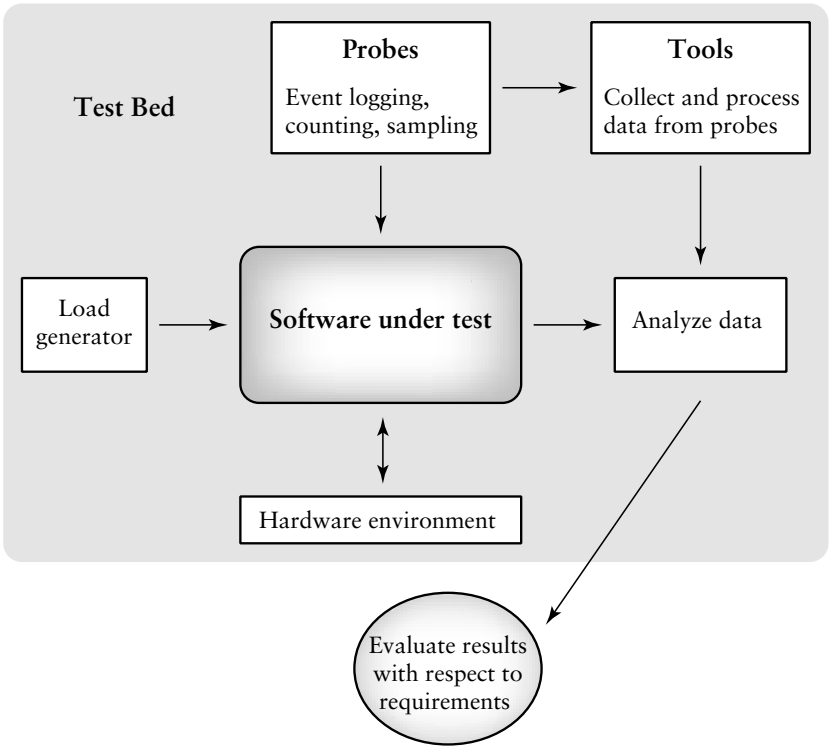


Fig. 6.11
Examples of special resources needed for a performance test.

in mind that the probes themselves may have an impact on system performance.

- A set of tools to collect, store, process, and interpret the data. Very often, large volumes of data are collected, and without tools the testers may have difficulty in processing and analyzing the data in order to evaluate true performance levels.

Test managers should ascertain the availability of these resources, and allocate the necessary time for training in the test plan. Usage requirements for these resources need to be described as part of the test plan.

6.13.3 Stress Testing

When a system is tested with a load that causes it to allocate its resources in maximum amounts, this is called stress testing. For example, if an

operating system is required to handle 10 interrupts/second and the load causes 20 interrupts/second, the system is being stressed. The goal of stress test is to try to break the system; find the circumstances under which it will crash. This is sometimes called "breaking the system." An everyday analogy can be found in the case where a suitcase being tested for strength and endurance is stomped on by a multiton elephant!

Stress testing is important because it can reveal defects in real-time and other types of systems, as well as weak areas where poor design could cause unavailability of service. For example, system prioritization orders may not be correct, transaction processing may be poorly designed and waste memory space, and timing sequences may not be appropriate for the required tasks. This is particularly important for real-time systems where unpredictable events may occur resulting in input loads that exceed those described in the requirements documents. Stress testing often uncovers race conditions, deadlocks, depletion of resources in unusual or unplanned patterns, and upsets in normal operation of the software system. System limits and threshold values are exercised. Hardware and software interactions are stretched to the limit. All of these conditions are likely to reveal defects and design flaws which may not be revealed under normal testing conditions.

Stress testing is supported by many of the resources used for performance test as shown in Figure 6.11. This includes the load generator. The testers set the load generator parameters so that load levels cause stress to the system. For example, in our example of a telecommunication system, the arrival rate of calls, the length of the calls, the number of misdials, as well as other system parameters should all be at stress levels. As in the case of performance test, special equipment and laboratory space may be needed for the stress tests. Examples are hardware or software probes and event loggers. The tests may need to run for several days. Planners must insure resources are available for the long time periods required. The reader should note that stress tests should also be conducted at the integration, and if applicable at the unit level, to detect stress-related defects as early as possible in the testing process. This is especially critical in cases where redesign is needed.

Stress testing is important from the user/client point of view. When system operate correctly under conditions of stress then clients have con-

fidence that the software can perform as required. Beizer suggests that devices used for monitoring stress situations provide users/clients with visible and tangible evidence that the system is being stressed [22].

6.13.4 Configuration Testing

Typical software systems interact with hardware devices such as disc drives, tape drives, and printers. Many software systems also interact with multiple CPUs, some of which are redundant. Software that controls real-time processes, or embedded software also interfaces with devices, but these are very specialized hardware items such as missile launchers, and nuclear power device sensors. In many cases, users require that devices be interchangeable, removable, or reconfigurable. For example, a printer of type X should be substitutable for a printer of type Y, CPU A should be removable from a system composed of several other CPUs, sensor A should be replaceable with sensor B. Very often the software will have a set of commands, or menus, that allows users to make these configuration changes. Configuration testing allows developers/testers to evaluate system performance and availability when hardware exchanges and reconfigurations occur. Configuration testing also requires many resources including the multiple hardware devices used for the tests. If a system does not have specific requirements for device configuration changes then large-scale configuration testing is not essential.

According to Beizer configuration testing has the following objectives [22]:

- Show that all the configuration changing commands and menus work properly.
- Show that all interchangeable devices are really interchangeable, and that they each enter the proper states for the specified conditions.
- Show that the systems' performance level is maintained when devices are interchanged, or when they fail.

Several types of operations should be performed during configuration test. Some sample operations for testers are [22]:

(i) rotate and permutate the positions of devices to ensure physical/logical device permutations work for each device (e.g., if there are two printers A and B, exchange their positions);
(ii) induce malfunctions in each device, to see if the system properly handles the malfunction;
(iii) induce multiple device malfunctions to see how the system reacts.

These operations will help to reveal problems (defects) relating to hardware/software interactions when hardware exchanges, and reconfigurations occur. Testers observe the consequences of these operations and determine whether the system can recover gracefully particularly in the case of a malfunction.

6.13.5 Security Testing

Designing and testing software systems to insure that they are safe and secure is a big issue facing software developers and test specialists. Recently, safety and security issues have taken on additional importance due to the proliferation of commercial applications for use on the Internet. If Internet users believe that their personal information is not secure and is available to those with intent to do harm, the future of e-commerce is in peril! Security testing evaluates system characteristics that relate to the availability, integrity, and confidentially of system data and services. Users/clients should be encouraged to make sure their security needs are clearly known at requirements time, so that security issues can be addressed by designers and testers.

Computer software and data can be compromised by:

(i) criminals intent on doing damage, stealing data and information, causing denial of service, invading privacy;
(ii) errors on the part of honest developers/maintainers who modify, destroy, or compromise data because of misinformation, misunderstandings, and/or lack of knowledge.

Both criminal behavior and errors that do damage can be perpetuated by those inside and outside of an organization. Attacks can be random or systematic. Damage can be done through various means such as:

(i) viruses;
(ii) trojan horses;
(iii) trap doors;
(iv) illicit channels.

The effects of security breaches could be extensive and can cause:

(i) loss of information;
(ii) corruption of information;
(iii) misinformation;
(iv) privacy violations;
(v) denial of service.

Physical, psychological, and economic harm to persons or property can result from security breaches. Developers try to ensure the security of their systems through use of protection mechanisms such as passwords, encryption, virus checkers, and the detection and elimination of trap doors. Developers should realize that protection from unwanted entry and other security-oriented matters must be addressed at design time. A simple case in point relates to the characteristics of a password. Designers need answers to the following: What is the minimum and maximum allowed length for the password? Can it be pure alphabetical or must it be a mixture of alphabetical and other characters? Can it be a dictionary word? Is the password permanent, or does it expire periodically? Users can specify their needs in this area in the requirements document. A password checker can enforce any rules the designers deem necessary to meet security requirements.

Password checking and examples of other areas to focus on during security testing are described below.

Password Checking–Test the password checker to insure that users will select a password that meets the conditions described in the password checker specification. Equivalence class partitioning and boundary value analysis based on the rules and conditions that specify a valid password can be used to design the tests.

Legal and Illegal Entry with Passwords–Test for legal and illegal system/data access via legal and illegal passwords.

Password Expiration–If it is decided that passwords will expire after a certain time period, tests should be designed to insure the expiration period is properly supported and that users can enter a new and appropriate password.

Encryption–Design test cases to evaluate the correctness of both encryption and decryption algorithms for systems where data/messages are encoded.

Browsing–Evaluate browsing privileges to insure that unauthorized browsing does not occur. Testers should attempt to browse illegally and observe system responses. They should determine what types of private information can be inferred by both legal and illegal browsing.

Trap Doors–Identify any unprotected entries into the system that may allow access through unexpected channels (trap doors). Design tests that attempt to gain illegal entry and observe results. Testers will need the support of designers and developers for this task. In many cases an external "tiger team" as described below is hired to attempt such a break into the system.

Viruses–Design tests to insure that system virus checkers prevent or curtail entry of viruses into the system. Testers may attempt to infect the system with various viruses and observe the system response. If a virus does penetrate the system, testers will want to determine what has been damaged and to what extent.

Even with the backing of the best intents of the designers, developers/testers can never be sure that a software system is totally secure even after extensive security testing. If security is an especially important issue, as in the case of network software, then the best approach if resources permit, is to hire a so-called "tiger team" which is an outside group of penetration experts who attempt to breach the system security. Although a testing group in the organization can be involved in testing for security breaches, the tiger team can attack the problem from a different point of view. Before the tiger team starts its work the system should be thoroughly tested at all levels. The testing team should also try to identify any trap

doors and other vulnerable points. Even with the use of a tiger team there is never any guarantee that the software is totally secure.

6.13.6 Recovery Testing

Recovery testing subjects a system to losses of resources in order to determine if it can recover properly from these losses. This type of testing is especially important for transaction systems, for example, on-line banking software. A test scenario might be to emulate loss of a device during a transaction. Tests would determine if the system could return to a well-known state, and that no transactions have been compromised. Systems with automated recovery are designed for this purpose. They usually have multiple CPUs and/or multiple instances of devices, and mechanisms to detect the failure of a device. They also have a so-called "checkpoint" system that meticulously records transactions and system states periodically so that these are preserved in case of failure. This information allows the system to return to a known state after the failure. The recovery testers must ensure that the device monitoring system and the checkpoint software are working properly.

Beizer advises that testers focus on the following areas during recovery testing [22]:

1. *Restart*. The current system state and transaction states are discarded. The most recent checkpoint record is retrieved and the system initialized to the states in the checkpoint record. Testers must insure that all transactions have been reconstructed correctly and that all devices are in the proper state. The system should then be able to begin to process new transactions.
2. *Switchover*. The ability of the system to switch to a new processor must be tested. Switchover is the result of a command or a detection of a faulty processor by a monitor.

In each of these testing situations all transactions and processes must be carefully examined to detect:

(i) loss of transactions;
(ii) merging of transactions;

(iii) incorrect transactions;
(iv) an unnecessary duplication of a transaction.

A good way to expose such problems is to perform recovery testing under a stressful load. Transaction inaccuracies and system crashes are likely to occur with the result that defects and design flaws will be revealed.

6.14 Regression Testing

Regression testing is not a level of testing, but it is the *retesting* of software that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes. Regression testing can occur at any level of test, for example, when unit tests are run the unit may pass a number of these tests until one of the tests does reveal a defect. The unit is repaired and then retested with all the old test cases to ensure that the changes have not affected its functionality. Regression tests are especially important when multiple software releases are developed. Users want new capabilities in the latest releases, but still expect the older capabilities to remain in place. This is where regression testing plays a role. Test cases, test procedures, and other test-related items from previous releases should be available so that these tests can be run with the new versions of the software. Automated testing tools support testers with this very time-consuming task. Later chapters will describe the role of these testing tools.

6.15 Alpha, Beta, and Acceptance Tests

In the various testing activities that have been described so far, users have played a supporting role for the most part. They have been involved in requirements analysis and reviews, and have played a role in test planning. This is especially true for acceptance test planning if the software is being

custom made for an organization. The clients along with test planners design the actual test cases that will be run during acceptance test.

Users/clients may also have participated in prototype evaluation, usage profile development, and in the various stages of usability testing (see Chapter 12). After the software has passed all the system tests and defect repairs have been made, the users take a more active role in the testing process. Developers/testers must keep in mind that the software is being developed to satisfy the users requirements, and no matter how elegant its design it will not be accepted by the users unless it helps them to achieve their goals as specified in the requirements. Alpha, beta, and acceptance tests allow users to evaluate the software in terms of their expectations and goals.

When software is being developed for a specific client, acceptance tests are carried out after system testing. The acceptance tests must be planned carefully with input from the client/users. Acceptance test cases are based on requirements. The user manual is an additional source for test cases. System test cases may be reused. The software must run under real-world conditions on operational hardware and software. The software-under-test should be stressed. For continuous systems the software should be run at least through a 25-hour test cycle. Conditions should be typical for a working day. Typical inputs and illegal inputs should be used and all major functions should be exercised. If the entire suite of tests cannot be run for any reason, then the full set of tests needs to be rerun from the start.

Acceptance tests are a very important milestone for the developers. At this time the clients will determine if the software meets their requirements. Contractual obligations can be satisfied if the client is satisfied with the software. Development organizations will often receive their final payment when acceptance tests have been passed.

Acceptance tests must be rehearsed by the developers/testers. There should be no signs of unprofessional behavior or lack of preparation. Clients do not appreciate surprises. Clients should be received in the development organization as respected guests. They should be provided with documents and other material to help them participate in the acceptance testing process, and to evaluate the results. After acceptance testing the client will point out to the developers which requirement have/have

not been satisfied. Some requirements may be deleted, modified, or added due to changing needs. If the client has been involved in prototype evaluations then the changes may be less extensive.

If the client is satisfied that the software is usable and reliable, and they give their approval, then the next step is to install the system at the client's site. If the client's site conditions are different from that of the developers, the developers must set up the system so that it can interface with client software and hardware. Retesting may have to be done to insure that the software works as required in the client's environment. This is called installation test.

If the software has been developed for the mass market (shrink-wrapped software), then testing it for individual clients/users is not practical or even possible in most cases. Very often this type of software undergoes two stages of acceptance test. The first is called alpha test. This test takes place at the developer's site. A cross-section of potential users and members of the developer's organization are invited to use the software. Developers observe the users and note problems. Beta test sends the software to a cross-section of users who install it and use it under real-world working conditions. The users send records of problems with the software to the development organization where the defects are repaired sometimes in time for the current release. In many cases the repairs are delayed until the next release.

6.16 Summary Statement on Testing Levels

In this chapter we have studied the testing of software at different levels of abstraction as summarized in Figure 6.1. The reader should note that each testing level:

- focuses on a specific level of abstraction of the software;
- has a set of specific goals;
- is useful for revealing different types of defects (problems);
- uses a specific set of documents as guides for designing tests;

- is useful for evaluating certain functional and quality attributes of the software;
- is associated with a level-oriented test plan (described in Chapter 7).

The study of the material in this chapter gives the reader an appreciation of the size and complexity of the entire testing effort. To achieve testing goals and to perform testing in an effective manner, testers must be motivated, have good technical and communication skills, and be good planners and managers. It is the goal of the testing group working along with developers and other software professionals to release a software system to the customer that meets all requirements.

6.17 The Special Role of Use Cases

The importance of software models as aids to the tester has been described throughout this book. For example, the role of state models, data flow, and control flow models in designing black and white box test cases is described in Chapters 4 and 5. In this chapter, another important model is introduced called the "use case." A description of a use case is as follows.

A use case is a pattern, scenario, or exemplar of usage. It describes a typical interaction between the software system under development and a user.

A use case scenario begins with some user of the system (human, hardware device, an interfacing software system) initiating a transaction or a sequence of events. The interaction is often depicted as a diagram or graphical drawing showing the entities involved. In addition, a textual description of the interaction often accompanies the graphic representation. The text describes the sequence of events that occurs when such a transaction is initiated. All the events that occur and the system's responses to the events are part of the textural description (scenario script). The design of use cases typically begins in the requirements phase. User interactions with respect to primary system functions are collected and analyzed. Each of the scenarios/interactions in the collection are modeled in the form of a use case. As development continues the use cases can be

refined to include, for example, exception conditions. Scenarios for interactions involving secondary system functions can be added. The entire collection of use cases gives a complete description of system use by users. The use cases are usually reviewed with testers as participants in the review process. Customers also review the use cases.

The development of use cases is often associated with object-oriented development. They are used frequently to describe object responsibilities and/or to model object interactions. Current uses include application to both object-oriented and procedure-oriented systems to model user–system interactions. Use cases are especially of value to testers, and are useful for designing tests at the integration or system level for both types of systems. For example, a given use case may model a thread or transaction implemented by a subsystem or a class cluster. A set of use cases can also serve as a model for transactions involving the software system as whole. Use cases are also very useful for designing acceptance tests with customer participation. A reasonable testing goal would be to test uses of the system through coverage of all of the use cases. Each thread of functionality should be covered by a test.

Jacobson et al. briefly describe how use cases support integration and system test in object-oriented systems [23]. They suggest tests that cause the software to follow the expected flow of events, as well as tests that trigger odd or unexpected cases of the use case—a flow of events different than expected. An example use case associated with an automated payment system is shown in Figure 6.12. The customer initiates the transaction by inserting a card and a PIN. The steps following describe the interaction between the customer and the system in detail for a particular type of transaction—the automated payment. For our use case example we could develop a set of test inputs during integration or system test so that a typical interaction for the automated payment system would follow the flow of steps. The set of inputs for the test includes:

Valid user PIN: IB-1234

Selection option: Automated payment

Valid account number of the recipient: GWB-6789

Selection payment schedule: Weekly

Selection of day of the month: 15

Automated Payment Sequence
ATM Machine Software System

1. Customer inserts card in machine slot and enters PIN.
2. The card and PIN are verified and the main menu is shown.
3. Customer selects the "transaction services" menu, the menu is then displayed by the ATM.
4. Customer selects the "automated payment" service from the menu.
5. Customer is prompted for the account number of the recipient of the payment.
6. Customer enters the recipient's account number.
7. Recipient's account number is verified, and "payment schedule" menu is displayed.
8. Customer selects monthly payment schedule from menu. Choices include: weekly, monthly, yearly, etc. Secondary menu displays choices to refine payment schedule.
9. Customer inputs day-of-month for periodic payments.
10. Secondary menu displays options to set fixed and maximum amount.
11. Customer selects the maximum amount option (e.g., a $60 value). A menu for start date is displayed. On this date the payment is due.
12. Customer selects today's date as the start date.
13. The transaction is verified, and the main menu is displayed.

Fig. 6.12
Example text-based use case.

Maximum payment option value: $50

Start date: 3-15-01

In our example, use of an invalid recipient account number as input would alter the flow of events described and could constitute one odd use of the case. Other exceptions could include a payment entered that is larger than the customer's current account holding.

6.18 Levels of Testing and the TMM

The material covered in this chapter is associated with the TMM level 2 maturity goal, "institutionalize basic testing techniques and methods,"

which addresses important technical issues related to execution-based testing. The focus is on the different levels of testing that must occur when a complex software system is being developed. An organization at TMM level 2 must work hard to plan, support, and implement these levels of testing so that it can instill confidence in the quality of its software and the proficiency of its testing process. This is why the maturity goal to "institutionalize basic testing techniques and methods," which encompasses multilevel testing, appears at lower levels of the maturity goal hierarchy (TMM level 2). In Chapter 7 you will learn how these testing levels are supported by test planning. Later chapters will describe the tools to support these levels of testing.

Readers can see that testing at different levels requires many resources. Very often when budgets and schedules are tight, these levels are sacrificed and the organization often reverts to previous immature practices such as "big bang" integration and multiple test, debug, patch, and repair cycles. This will occur unless there are concerted efforts by the developers, quality personnel, and especially management to put these levels of testing into place as part of a concerted TMM-based test process improvement effort. Otherwise the organization's reputation for consistently releasing quality products will not be maintainable.

The three maturity goals at TMM level 2 are interrelated and support areas from the three critical groups overlap. In Chapters 4 and 5 you learned how the members of the three critical groups support the adaptation and application of white and black box testing methods. Chapter 7 will describe how critical group members support test planning and policy making. Below is a brief description of how critical group members support multilevel testing. Again, you will notice some overlap with the group responsibilities described in Chapters 4, 5, and 7.

Managers can support multilevel testing by:

- ensuring that the testing policy requires multilevel testing;
- ensuring that test plans are prepared for the multiple levels;
- providing the resources needed for multilevel testing;
- adjusting project schedules so that multilevel testing can be adequately performed;

- supporting the education and training of staff in testing methods and techniques needed to implement multilevel testing.

Developers/testers give their support by:

- attending classes and training sessions to master the knowledge needed to plan and implement multilevel testing;
- supporting management to ensure multilevel testing is a part of organizational policy, is incorporated into test plans, and is applied throughout the organization;
- working with project managers (and test managers at TMM level 3 and higher) to ensure there is time and resources to test at all levels;
- mentoring colleagues who wish to acquire the necessary background and experience to perform multilevel testing;
- work with users/clients to develop the use cases, usage profiles, and acceptance criteria necessary for the multilevel tests;
- implement the tests at all levels which involves:
  - —planning the tests
  - —designing the test cases
  - —gathering the necessary resources
  - —executing unit, integration, system, and acceptance tests
  - —collecting test results
  - —collecting test-related metrics
  - —analyzing test results
  - —interacting with developers, SQA staff, and user/clients to resolve problems

The *user/clients* play an essential role in the implementation of multilevel testing. They give support by:

- providing liaison staff to interact with the development organization testing staff;
- working with analysts so that system requirements are complete and clear and testable;

- providing input for system and acceptance test;
- providing input for the development of use cases and/or a usage profile to support system and acceptance testing;
- participating in acceptance and/or alpha and beta tests;
- providing feedback and problem reports promptly so that problems can be addressed after the system is in operation.

LIST OF KEY TERMS

Cluster
Load
Test harness
Unit
Use case

EXERCISES

1. How would you define a software unit? In terms of your definition, what constitutes a unit for procedural code; for object-oriented code?

2. Summarize the issues that arise in class testing.

3. The text gives example sequences of inputs and calls to test the stack class as shown in Figure 6.3. Develop sequences for testing the stack that try to push an item on a full stack, and to pop an item from an empty stack.

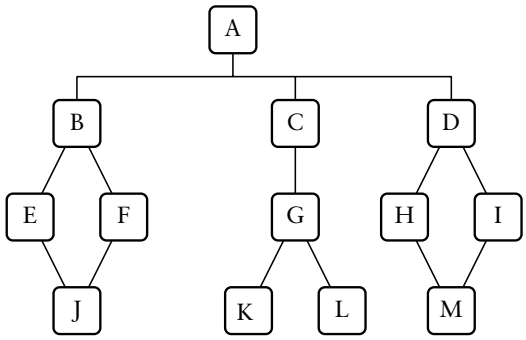
4. Why is it so important to design a test harness for reusability?

5. Suppose you were developing a stub that emulates a module that passes back a hash value when passed a name. What are the levels of functionality you could implement for this stub? What factors could influence your choice of levels?

6. What are the key differences in integrating procedural-oriented systems as compared to object-oriented systems?

7. From your knowledge of defect types in Chapter 3 of this text, what defect types are most likely to be detected during integration test of a software system? Describe your choices in terms of both the nature of integration test and the nature of the defect types you select.

8. Using the structure chart shown below, show the order of module integration for the top-down (depth and breadth first), and bottom-up integration approaches. Estimate the number of drivers and stubs needed for each approach. Specify integration testing activities that can be done in parallel, assuming you have a maximum of three testers. Based on resource needs and the ability to carry out parallel testing activities, which approach would you select for this system and why?



9. This chapter describe several types of system tests. Select from these types those you would perform for the software described below. For each category you choose (i) specify the test objectives, and (ii) give a general description of the tests you would develop and tools you would need. You may make any assumptions related to system characteristics that are needed to support your answers.

An on-line fast food restaurant system. The system reads customer orders, relays orders to the kitchen, calculates the customer's bill, and gives change. It also maintains inventory information. Each wait-person has a terminal. Only authorized wait-persons and a system administrator can access the system.

10. An air-traffic control system can have one or many users. It interfaces with many hardware devices such as displays, radar detectors, and communications devices. This system can occur in a variety of configurations. Describe how you would carry out configuration tests on this system.

11. As in Problem 9, describe the types of system tests you would select for the following software. The project is a real-time control system for a new type of laser that will be used for cancer therapy. Some of the code will be used to control hardware devices. Only qualified technicians can access the system.

12. Discuss the importance of regression testing when developing a new software release. What items from previous release would be useful to the regression tester?

13. From your experience with online and/or catalog shopping, develop a use case to describe a user purchase of a television set with a credit card from a online vendor using web-based software. With the aid of your use case, design a set of tests you could use during system test to evaluate the software.

14. Describe the Activities/Tasks and Responsibilities for developer/testers in support of multilevel testing.

REFERENCES

[1] P. Jorgensen, C. Erikson, "Object-oriented integration Test," *CACM,* Vol. 37, No. 9, 1994, pp. 30–38.

[2] R. D'Souza, R. LeBlanc, "Class testing by examining pointers," *J. Object Oriented Programming,* July–August, 1994, pp. 33–39.

[3] M. Smith, D. Robson, "A framework for testing object-oriented programs," *J. Object Oriented Programming,* June 1992, pp. 45–53.

[4] S. Fiedler, "Object-oriented unit testing," *Hewlett-Packard Journal,* April, 1989, pp. 69–74.

[5] G. Murphy, P. Townsend, P. Wong, "Experiences with cluster and class testing," *CACM,* Vol. 37, No. 9, 1994, pp. 39–47.

[6] N. Wilde, "Testing your objects," *The C Users Journal,* May 1993, pp. 25–32.

[7] R. Doong, P. Frankl, "The ASTOOT approach to testing object-oriented programs," *ACM Transactions of Software Engineering and Methodology,* Vol. 3, No., 1994, pp 101–130.

[8] E. Berard, *Essays on Object-Oriented Software Engineering,* Volume 1, Prentice Hall, Englewood Cliffs, NJ, 1993.

[9] B. Marick, *The Craft of Software Testing,* Prentice Hall, Englewood Cliffs, NJ, 1995.

[10] IEEE/ANSI Std 1008-1987 (Reaff 1993), Standard for Software Unit Testing, Institute of Electrical and Electronics Engineers, Inc., 1987.

[11] J. McGregor, A. Kare, "Parallel architecture for component testing of object-oriented software," *Proc. Ninth International Quality Week Conf.,* May 1996.

[12] M. Chen, M. Kao, "Investigating test effectiveness on object-oriented software: a case study," *Proc. Twelfth International Quality Week Conf.,* May 1999.

[13] D. Perry. G. Kaiser, "Adequate testing and object-oriented programming," *J. Object Oriented Programming,* Vol. 2, No. 5, 1990, pp. 13–19.

[14] K Rangaraajan, P. Eswar, T. Ashok, "Retesting C++ classes," *Proc. Ninth International Quality Week Conf.,* May 1996.

[15] B. Tsai, S. Stobart, N. Parrington, I. Mitchell, "A state-based testing approach providing data flow coverage in object-oriented class testing," *Proc. Twelfth International Quality Week Conf.,* May 1999.

[16] M Harrold, J. McGregor, K. Fitzpatrick, "Incremental testing of object-oriented class structures," *Proc. 14th International Conf. on Software Engineering,* May 1992, pp. 68–80.

[17] M. Harrold, G. Rothermel, "Performing data flow testing on classes," *Proc. Second ACM SIGSOFT Symposium on Foundations of Software Engineering,* Dec. 1994, pp. 154–163.

[18] D. Kung, P. Hsia, J. Gao, *Testing Object-Oriented Software,* IEEE Computer Society Press, Los Alamitos, CA, 1998.

[19] U. Linnenkugel, M. Mullerburg, "Test data selection criteria for (software) integration testing," *Proc. First International Conf. Systems Integration,* April 1990, pp. 709–717.

[20] P. Coad, E. Yourdon, *Object-Oriented Analysis,* second edition, Yourdon Press, Englewood Cliffs, NJ, 1991.

[21] B. Hetzel *The Complete Guide to Software Testing,* second edition, QED Information Sciences, Inc., Wellesley, MA. 1988.

[22] B. Beizer, *Software Testing and Quality Assurance,* Von Nostrand Reinhold, New York, 1984.

[23] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach,* Addison-Wesley, Reading, MA, 1992.