CHAPTER 7

System Integration Testing

Pruebas unitarias → buscamos defectos DENTRO de una unidad

> I criticize by creation, not by finding fault.
>
> — *Marcus Tullius Cicero*

Pruebas de integración → buscamos defectos en las INTERFACES de las unidades

7.1 CONCEPT OF INTEGRATION TESTING

A software module, or component, is a self-contained element of a system. Modules have well-defined interfaces with other modules. A module can be a subroutine, function, procedure, class, or collection of those basic elements put together to deliver a higher level service. A system is a collection of modules interconnected in a certain way to accomplish a tangible objective. A subsystem is an interim system that is not fully integrated with all the modules. It is also known as a subassembly.

In moderate to large projects, from tens to hundreds of programmers implement their share of the code in the form of modules. Modules are individually tested, which is commonly known as *unit testing*, by their respective programmers using white-box testing techniques. At the unit testing level, the system exists in pieces under the control of the programmers. The next major task is to put the modules, that is, pieces, together to construct the complete system. Constructing a working system from the pieces is not a straightforward task, because of numerous interface errors. Even constructing a reasonably stable system from the components involves much testing. The path from tested components to constructing a deliverable system contains two major testing phases, namely, integration testing and system testing. The primary objective of integration testing is to assemble a reasonably stable system in a laboratory environment such that the integrated system can withstand the rigor of a full-blown system testing in the actual environment of the system. The importance of integration testing stems from three reasons as outlined below.

- Different modules are generally created by groups of different developers. The developers may be working at different sites. In spite of our best effort in system design and documentation, misinterpretation, mistakes, and

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

oversights do occur in reality. Interface errors between modules created by different programmers and even by the same programmers are rampant. We will discuss the sources of interface errors in Section 7.2.

- Unit testing of individual modules is carried out in a controlled environment by using test drivers and stubs. Stubs are dummy modules which merely return predefined values. If a module under unit test invokes several other modules, the effectiveness of unit testing is constrained by the programmer's ability to effectively test all the paths. Therefore, with the inherent limitations of unit testing, it is difficult to predict the behavior of a module in its actual environment after the unit testing is performed.
- Some modules are more error prone than other modules, because of their inherent complexity. It is essential to identify the ones causing most failures.

The objective of system integration is to build a "working" version of the system by (i) putting the modules together in an incremental manner and (ii) ensuring that the additional modules work as expected without disturbing the functionalities of the modules already put together. In other words, system integration testing is a systematic technique for assembling a software system while conducting tests to uncover errors associated with interfacing. We ensure that unit-tested modules operate correctly when they are combined together as dictated by the design. Integration testing usually proceeds from small subassemblies containing a few modules to larger ones containing more and more modules. Large, complex software products can go through several iterations of build-and-test cycles before they are fully integrated.

Integration testing is said to be complete when the system is fully integrated together, all the test cases have been executed, all the severe and moderate defects found have been fixed, and the system is retested.

Retesting: repetimos las pruebas si hemos encontrado
defectos y los hemos depurado
Regression testing: repetimos las pruebas anteriores
hayamos depurado defectos o no

Retesting es diferente
de Regression Testing!!

7.2 DIFFERENT TYPES OF INTERFACES AND INTERFACE ERRORS

Modularization is an important principle in software design, and modules are interfaced with other modules to realize the system's functional requirements. An interface between two modules allows one module to access the service provided by the other. It implements a mechanism for passing control and data between modules. Three common paradigms for interfacing modules are as follows:

- **Procedure Call Interface:** A procedure in one module calls a procedure in another module. The caller passes on control to the called module. The caller can pass data to the called procedure, and the called procedure can pass data to the caller while returning control back to the caller.
- **Shared Memory Interface:** A block of memory is shared between two modules. The memory block may be allocated by one of the two modules

or a third module. Data are written into the memory block by one module and are read from the block by the other.

- **Message Passing Interface:** One module prepares a message by initializing the fields of a data structure and sending the message to another module. This form of module interaction is common in client–server-based systems and web-based systems.

Programmers test modules to their satisfaction. The question is: If all the unit-tested modules work individually, why can these modules not work when put together? The problem arises when we "put them together" because of rampant interface errors. Interface errors are those that are associated with structures existing outside the local environment of a module but which the module uses [1]. Perry and Evangelist [2] reported in 1987 that interface errors accounted for up to a quarter of all errors in the systems they examined. They found that of all errors that required a fix within one module, more than half were caused by interface errors. Perry and Evangelist have categorized interface errors as follows:

Cuando hacemos "asunciones" que no nos han especificado, cuando nos "inventamos" resultados esperados ... las pruebas unitarias funcionarán, PERO, cuando hagamos las pruebas de integración, todos esos "añadidos" por nuestra cuenta crearán errores adicionales que tendremos que depurar!!!

1. *Construction*: Some programming languages, such as C, generally separate the interface specification from the implementation code. In a C program, programmers can write a statement #include header.h, where header.h contains an interface specification. Since the interface specification lies somewhere away from the actual code, programmers overlook the interface specification while writing code. Therefore, inappropriate use of #include statements cause construction errors.
2. *Inadequate Functionality*: These are errors caused by implicit assumptions in one part of a system that another part of the system would perform a function. However, in reality, the "other part" does not provide the expected functionality—intentionally or unintentionally by the programmer who coded the other part.
3. *Location of Functionality*: Disagreement on or misunderstanding about the location of a functional capability within the software leads to this sort of error. The problem arises due to the design methodology, since these disputes should not occur at the code level. It is also possible that inexperienced personnel contribute to the problem.
4. *Changes in Functionality*: Changing one module without correctly adjusting for that change in other related modules affects the functionality of the program.
5. *Added Functionality*: A completely new functional module, or capability, was added as a system modification. Any added functionality after the module is checked in to the version control system without a CR is considered to be an error.
6. *Misuse of Interface*: One module makes an error in using the interface of a called module. This is likely to occur in a procedure–call interface. Interface misuse can take the form of wrong parameter type, wrong parameter order, or wrong number of parameters passed.

7. *Misunderstanding of Interface*: A calling module may misunderstand the interface specification of a called module. The called module may assume that some parameters passed to it satisfy a certain condition, whereas the caller does not ensure that the condition holds. For example, assume that a called module is expected to return the index of an element in an array of integers. The called module may choose to implement binary search with an assumption that the calling module gives it a sorted array. If the caller fails to sort the array before invoking the second module, we will have an instance of interface misunderstanding.

8. *Data Structure Alteration*: These are similar in nature to the functionality problems discussed above, but they are likely to occur at the detailed design level. The problem arises when the size of a data structure is inadequate or it fails to contain a sufficient number of information fields. The problem has its genesis in the failure of the high-level design to fully specify the capability requirements of the data structure. Let us consider an example in which a module reads the data and keeps it in a record structure. Each record holds the person name followed by their employee number and salary. Now, if the data structure is defined for 1000 records, then as the number of record grows beyond 1000, the program is bound to fail. In addition, if management decides to award bonuses to a few outstanding employees, there may not be any storage space allocated for additional information.

9. *Inadequate Error Processing*: A called module may return an error code to the calling module. However, the calling module may fail to handle the error properly.

10. *Additions to Error Processing*: These errors are caused by changes to other modules which dictated changes in a module error handling. In this case either necessary functionality is missing from the current error processing that would help trace errors or current techniques of error processing require modification.

11. *Inadequate Postprocessing*: These errors are caused by a general failure to release resources no longer required, for example, failure to deallocate memory.

12. *Inadequate Interface Support*: The actual functionality supplied was inadequate to support the specified capabilities of the interface. For example, a module passes a temperature value in Celsius to a module which interprets the value in Fahrenheit.

13. *Initialization/Value Errors*: A failure to initialize, or assign, the appropriate value to a variable data structure leads to this kind of error. Problems of this kind are usually caused by simple oversight. For example, the value of a pointer can change; it might point to the first character in a string, then to the second character, after that to the third character, and so on. If the programmer forgets to reinitialize the pointer before using that function once again, the pointer may eventually point to code.

14. *Violation of Data Constraints*: A specified relationship among data items was not supported by the implementation. This can happen due to incomplete detailed design specifications.
15. *Timing/Performance Problems*: These errors were caused by inadequate synchronization among communicating processes. A race condition is an example of these kinds of error. In the classical race, there are two possible events event a and event b happening in communicating processes process A and process B, respectively. There is logical ground for expecting event a to precede event b. However, under an abnormal condition event b may occur before event a. The program will fail if the software developer did not anticipate the possibility of event b preceding event a and did not write any code to deal with the situation.
16. *Coordination of Changes*: These errors are caused by a failure to communicate changes to one software module to those responsible for other interrelated modules.
17. *Hardware/Software Interfaces*: These errors arise from inadequate software handling of hardware devices. For example, a program can send data at a high rate until the input buffer of the connected device is full. Then the program has to pause until the device frees up its input buffer. The program may not recognize the signal from the device that it is no longer ready to receive more data. Loss of data will occur due to a lack of synchronization between the program and the device.

Interface errors cannot be detected by performing unit testing on modules since unit testing causes computation to happen within a module, whereas interactions are required to happen between modules for interface errors to be detected. It is difficult to observe interface errors by performing system-level testing, because these errors tend to be buried in system internals. The major advantages of conducting system integration testing are as follows:

- Defects are detected early.
- It is easier to fix defects detected earlier.
- We get earlier feedback on the health and acceptability of the individual modules and on the overall system.
- Scheduling of defect fixes is flexible, and it can overlap with development.

System integration testing is performed by the system integration group, also known as a build engineering group. The integration test engineers need to know the details of the software modules. This means that the team of engineers who built the modules needs to be involved in system integration. The integration testers should be familiar with the interface mechanisms. The system architects should be involved in the integration testing of complex software systems because of the fact that they have the bigger picture of the system.

7.3 GRANULARITY OF SYSTEM INTEGRATION TESTING

NO

System integration testing is performed at different levels of granularity. Integration testing includes both white- and black-box testing approaches. *Black-box* testing ignores the internal mechanisms of a system and focuses solely on the outputs generated in response to selected inputs and execution conditions. The code is considered to be a big black box by the tester who cannot examine the internal details of the system. The tester knows the input to the black box and observes the expected outcome of the execution. *White-box* testing uses information about the structure of the system to test its correctness. It takes into account the internal mechanisms of the system and the modules. In the following, we explain the ideas of *intrasystem* testing, *intersystem* testing, and *pairwise* testing.

aquí no estamos hablando de DISEÑO!

1. *Intrasystem Testing:* This form of testing constitutes low-level integration testing with the objective of combining the modules together to build a cohesive system. The process of combining modules can progress in an incremental manner akin to constructing and testing successive builds, explained in Section 7.4.1. For example, in a client–server-based system both the client and the server are distinct entities running at different locations. Before the interactions of clients with a server are tested, it is essential to individually construct the client and the server systems from their respective sets of modules in an incremental fashion. The low-level design document, which details the specification of the modules within the architecture, is the source of test cases.

2. *Intersystem Testing:* Intersystem testing is a high-level testing phase which requires interfacing independently tested systems. In this phase, all the systems are connected together, and testing is conducted from end to end. The term *end to end* is used in communication protocol systems, and end-to-end testing means initiating a test between two access terminals interconnected by a network. The purpose in this case is to ensure that the interaction between the systems work together, but not to conduct a comprehensive test. Only one feature is tested at a time and on a limited basis. Later, at the time of system testing, a comprehensive test is conducted based on the requirements, and this includes functional, interoperability, stress, performance, and so on. Integrating a client–server system, after integrating the client module and the server module separately, is an example of intersystem testing. Integrating a call control system and a billing system in a telephone network is another example of intersystem testing. The test cases are derived from the high-level design document, which details the overall system architecture.

3. *Pairwise Testing:* There can be many intermediate levels of system integration testing between the above two extreme levels, namely intrasystem testing and intersystem testing. Pairwise testing is a kind of intermediate level of integration testing. In pairwise integration, only two interconnected systems in an overall system are tested at a time. The purpose of pairwise testing is to ensure that two systems under consideration can function together, assuming that the other systems

within the overall environment behave as expected. The whole network infrastructure needs to be in place to support the test of interactions of the two systems, but the rest of the systems are not subject to tests. The network test infrastructure must be simple and stable during pairwise testing. While pairwise testing may sound simple, several issues can complicate the testing process. The biggest issue is unintended side effects. For example, in testing communication between a network element (radio node) and the element management systems, if another device (radio node controller) within the 1xEV-DO wireless data network, discussed in Chapter 8, fails during the test, it may trigger a high volume of traps to the element management systems. Untangling this high volume of traps may be difficult.

7.4 SYSTEM INTEGRATION TECHNIQUES

One of the objectives of integration testing is to combine the software modules into a working system so that system-level tests can be performed on the complete system. Integration testing need not wait until all the modules of a system are coded and unit tested. Instead, it can begin as soon as the relevant modules are available. A module is said to be available for combining with other modules when the module's *check-in request form*, to be discussed in this section, is ready. Some common approaches to performing system integration are as follows:

- Incremental
- Top down
- Bottom up
- Sandwich
- Big bang

In the remainder of this section, we explain the above approaches.

7.4.1 Incremental

In this approach, integration testing is conducted in an incremental manner as a series of test cycles as suggested by Deutsch [3]. In each test cycle, a few more modules are integrated with an existing and tested build to generate a larger build. The idea is to complete one cycle of testing, let the developers fix all the errors found, and continue the next cycle of testing. The complete system is built incrementally, cycle by cycle, until the whole system is operational and ready for system-level testing.

The system is built as a succession of layers, beginning with some core modules. In each cycle, a new layer is added to the core and tested to form a new core. The new core is intended to be self-contained and stable. Here, "self-contained" means containing all the necessary code to support a set of functions, and "stable" means that the subsystem (i.e., the new, partial system) can stay up for 24 hours without any anomalies. The number of system integration test cycles and the total integration time are determined by the following parameters:

- Number of modules in the system
- Relative complexity of the modules (cyclomatic complexity)
- Relative complexity of the interfaces between the modules
- Number of modules needed to be clustered together in each test cycle
- Whether the modules to be integrated have been adequately tested before
- Turnaround time for each test–debug–fix cycle

Constructing a build is a process by which individual modules are integrated to form an interim software image. A *software image* is a compiled software binary. A *build* is an interim software image for internal testing within the organization. Eventually, the final build will be a candidate for system testing, and such a tested system is released to the customers. Constructing a software image involves the following activities:

- Gathering the latest unit tested, authorized versions of modules
- Compiling the source code of those modules
- Checking in the compiled code to the repository
- Linking the compiled modules into subassemblies
- Verifying that the subassemblies are correct
- Exercising version control

A simple build involves only a small number of modules being integrated with a previously tested build on a reliable and well-understood platform. No special tool or procedure needs to be developed and documented for a simple build. On the other hand, organized, well-documented procedures are applied for complex builds. A build process becomes complicated if a large number of modules are integrated together, and a significant number of those modules are new with complex interfaces. These interfaces can be between software modules and hardware devices, across platforms, and across networks. For complex builds, a version control tool is highly recommended for automating the build process and for fast turnaround of a test–debug–fix cycle.

Creating a daily build [4] is very popular in many organizations because it facilitates to a faster delivery of the system. It puts emphasis on small incremental testing, steadily increasing the number of test cases, and regression testing from build to build. The integrated system is tested using automated, reusable test cases. An effort is made to fix the defects that were found during the testing cycle. A new version of the system is constructed from the existing, revised, and newly developed modules and is made available for retesting. Prior versions of the build are retained for reference and rollback. If a defect is not found in a module of a build in which the module was introduced, the module will be carried forward from build to build until one is found. Having access to the version where the defective module was originally introduced is useful in debugging and fixing, limiting the side effects of the fixes, and performing a root cause analysis. During system development, integration, and testing, a typical practice is to retain the past 7–10 builds.

The software developer fills out a check-in request form before a new software module or a module with an error fix is integrated into a build. The form is reviewed by the build engineering group for giving approval. Once it is approved, the module can be considered for integration. The main portions of a check-in form are given in Table 7.1. The idea behind having a check-in request mechanism is fourfold:

1. All the files requiring an update must be identified and known to other team members.
2. The new code must have been reviewed prior to its integration.
3. The new code must have been unit tested.
4. The scope of the check-in is identified.

A release note containing the following information accompanies a build:

- What has changed since the last build?
- What outstanding defects have been fixed?
- What are the outstanding defects in the build?
- What new modules or features have been added?

TABLE 7.1 Check-in Request Form

Author	Name of the person requesting this check-in
Today's date	month, day, year
Check-in request date	month, day, year
Category (identify all that apply)	New Feature: (Y, N)
	Enhancement: (Y, N)
	Defect: (Y, N); if yes: defect numbers:
	Are any of these major defects: (Y, N)
	Are any of these moderate defects: (Y, N)
Short description of check-in	Describe in a short paragraph the feature, the enhancement, or the defect fixes to be checked in.
Number of files to be checked in	Give the number of files to be checked in. Include the file names, if possible.
Code reviewer names	Provide the names of the code reviewers.
Command line interface changes made	(Y, N); if yes, were they:
	Documented? (Y, N)
	Reviewed? (Y, N, pending)
Does this check-in involve changes to global header?	(Y, N); if yes, include the header file names.
Does this check-in involve changes in output logging?	(Y, N); if yes, were they documented? (Y, N)
Unit test description	Description of the unit tests conducted
Comments	Any other comments and issues

- What existing modules or features have been enhanced, modified, or deleted?
- Are there any areas where unknown changes may have occurred?

A test strategy is created for each new build based on the above information. The following issues are addressed while planning a test strategy:

- What test cases need to be selected from the system integration test plan, as discussed in Section 7.6, in order to test the changes? Will these test cases give feature coverage of the new and modified features? If necessary, add new test cases to the system integration test plan.
- What existing test cases can be reused without modification in order to test the modified system? What previously failed test cases should now be reexecuted in order to test the fixes in the new build?
- How should the scope of a partial regression test be determined? A full regression test may not be run on each build because of frequent turnaround of builds. At the least, any earlier test cases which pertain to areas that have been modified must be reexecuted.
- What are the estimated time, resource demand, and cost to test this build? Some builds may be skipped based on this estimate and the current activities, because the integration test engineers may choose to wait for a later build.

7.4.2 Top Down

Systems with hierarchical structures easily lend themselves to top-down and bottom-up approaches to integration. In a hierarchical system, there is a first, top-level module which is decomposed into a few second-level modules. Some of the second-level modules may be further decomposed into third-level modules, and so on. Some or all the modules at any level may be terminal modules, where a terminal module is one that is no more decomposed. An internal module, also known as a nonterminal module, performs some computations, invokes its subordinate modules, and returns control and results to its caller. In top-down and bottom-up approaches, a design document giving the module hierarchy is used as a reference for integrating modules. An example of a module hierarchy is shown in Figure 7.1, where module A is the topmost module; module A has been decomposed into modules B, C, and D. Modules B, D, E, F, and G are terminal modules, as these have not been further decomposed. The top-down approach is explained in the following:

Step 1: Let IM represent the set of modules that have already been integrated and the required stubs. Initially, IM contains the top-level module and stubs corresponding to all the subordinate modules of the top-level

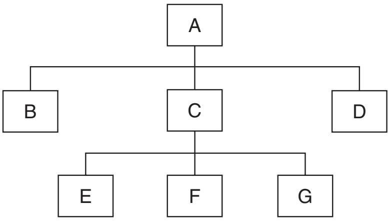

Figure 7.1 Module hierarchy with three levels and seven modules.

module. It is assumed that the top-level module has passed its entry criteria.

Step 2: Choose a stub member M′ in set IM. Let M be the actual module corresponding to stub M′. We obtain a new set CM from IM by replacing stub M′ with M and including in CM all stubs corresponding to the subordinate modules of M. We consider CM to be a union of four sets: {M}, CMs, CMi, CMr, where CMs is the set of stubs, CMi is the set of modules having direct interfaces with M, and CMr is the rest of the modules in CM.

Step 3: Now, test the combined behavior of CM. Testing CM means applying input to the top-level module of the system. It may be noted that though the integration team has access to the top module of the system, all kinds of tests cannot be performed. This is apparent from the fact that CM does not represent the full system. In this step, the integration team tests a subset of the system functions implemented by the actual modules in CM. The integration team performs two kinds of tests:

1. Run test cases to discover any interface defects between M and members of CMi.
2. Perform regression tests to ensure that integration of the modules in the two sets CMi and CMr is satisfactory in the presence of module M. One may note that in previous iterations the interfaces between modules in CMi and CMr were tested and the defects fixed. However, the said tests were executed with M′—a stub of M—and not M. The presence of M in the integrated system up to this moment allows us to test the interfaces between the modules in the combined set of CMi and CMr, because of the possibility of the system supporting more functionalities with M.

   The above two kinds of tests are continued until the integration team is satisfied that there is no known interface error. In case an interface error is discovered, the error must be fixed before moving on to the next step.

Step 4: If the set CMs is empty, then stop; otherwise, set IM = CM and go to step 2.

Now, let us consider an example of top-down integration using Figure 7.1. The integration of modules A and B by using stubs C′ and D′ (represented by grey boxes) is shown in Figure 7.2. Interactions between modules A and B is severely constrained by the dummy nature of C′ and D′. The interactions between A and B are concrete, and, as a consequence, more tests are performed after additional modules are integrated. Next, as shown in Figure 7.3, stub D′ has been replaced with its actual instance D. We perform two kinds of tests: first, test the interface between A and D; second, perform regression tests to look for interface defects between A and B in the presence of module D. Stub C′ has been replaced with the actual module C, and new stubs E′, F′, and G′ have been added to the integrated system (Figure 7.4). We perform tests as follows: First, test the interface between A and C; second, test the combined modules A, B, and D in the presence of C (Figure 7.4). The rest of the integration process is depicted in Figures 7.5 and 7.6 to obtain the final system of Figure 7.7.

The advantages of the top-down approach are as follows:

- System integration test (SIT) engineers continually observe system-level functions as the integration process continues. How soon such functions are observed depends upon their choice of the order in which modules are integrated. Early observation of system functions is important because it gives them better confidence.
- Isolation of interface errors becomes easier because of the incremental nature of top-down integration. However, it cannot be concluded that an interface error is due to a newly integrated module M. The interface error

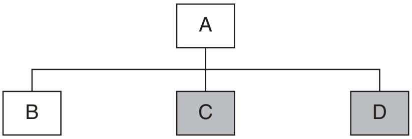

Figure 7.2 Top-down integration of modules A and B.

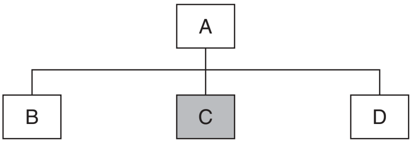

Figure 7.3 Top-down integration of modules A, B, and D.

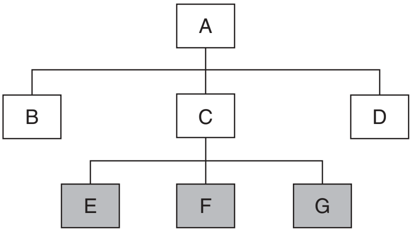

Figure 7.4 Top-down integration of modules A, B, D, and C.

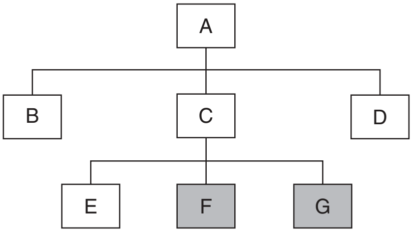

Figure 7.5 Top-down integration of modules A, B, C, D, and E.

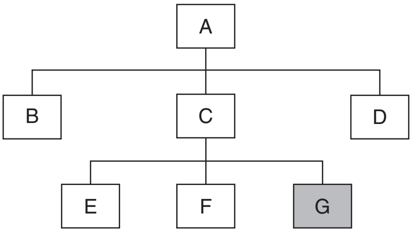

Figure 7.6 Top-down integration of modules A, B, C, D, E, and F.

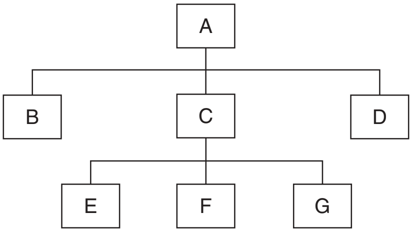

Figure 7.7 Top-down integration of modules A, B, C, D, E, F and G.

may be due to faulty implementation of a module that was already integrated much earlier. This is possible because earlier tests were conducted with or without a stub for M, and the full capability of M simply allowed the test engineers to conduct more tests that were possible due to M.

- Test cases designed to test the integration of a module M are reused during the regression tests performed after integrating other modules.
- Since test inputs are applied to the top-level module, it is natural that those test cases correspond to system functions, and it is easier to design those test cases than test cases designed to check internal system functions. Those test cases can be reused while performing the more rigorous, system-level tests.

The limitations of the top-down approach are as follows:

- Until a certain set of modules has been integrated, it may not be possible to observe meaningful system functions because of an absence of lower level modules and the presence of stubs. Careful analysis is required to identify an ordering of modules for integration so that system functions are observed as early as possible.

- Test case selection and stub design become increasingly difficult when stubs lie far away from the top-level module. This is because stubs support limited behavior, and any test run at the top level must be constrained to exercise the limited behavior of lower level stubs.

7.4.3 Bottom Up

In the bottom-up approach, system integration begins with the integration of lowest level modules. A module is said to be at the lowest level if it does not invoke another module. It is assumed that all the modules have been individually tested before. To integrate a set of lower level modules in this approach, we need to construct a test driver module that invokes the modules to be integrated. Once the integration of a desired group of lower level modules is found to be satisfactory, the driver is replaced with the actual module and one more test driver is used to integrate more modules with the set of modules already integrated. The process of bottom-up integration continues until all the modules have been integrated.

Now we give an example of bottom-up integration for the module hierarchy of Figure 7.1. The lowest level modules are E, F, and G. We design a test driver to integrate these three modules, as shown in Figure 7.8. It may be noted that modules E, F, and G have no direct interfaces among them. However, return values generated by one module is likely to be used in another module, thus having an indirect interface. The test driver in Figure 7.8 invokes modules E, F, and G in a way similar to their invocations by module C. The test driver mimics module C to integrate E, F, and G in a limited way, because it is much simpler in capability than module C. The test driver is replaced with the actual module—in this case C—and a new test driver is used after the testers are satisfied with the combined behavior of E, F, and G (Figure 7.9). At this moment, more modules, such as B and D, are integrated with the so-far integrated system. The test driver mimics the behavior of module A. We need to include modules B and D because those are invoked by A and the test driver mimics A (Figure 7.9). The test driver is replaced with module A (Figure 7.10), and further tests are performed after the testers are satisfied with the integrated system shown in Figure 7.9.

The advantages of the bottom-up approach are as follows. If the low-level modules and their combined functions are often invoked by other modules, then it is more useful to test them first so that meaningful effective integration of other modules can be done. In the absence of such a strategy, the testers write stubs to emulate the commonly invoked low-level modules, which will provide only a limited test capability of the interfaces.

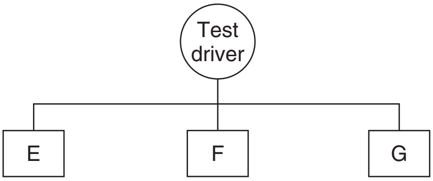

Figure 7.8 Bottom-up integration of modules E, F, and G.

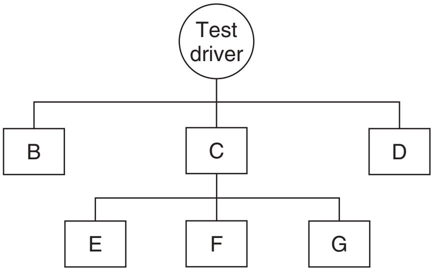

Figure 7.9 Bottom-up integration of modules B, C, and D with E, F, and G.

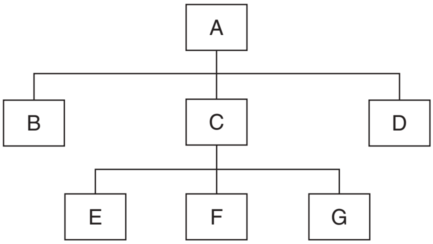

Figure 7.10 Bottom-up integration of module A with all others.

The disadvantages of the bottom-up approach are as follows:

- Test engineers cannot observe system-level functions from a partly integrated system. In fact, they cannot observe system-level functions until the top-level test driver is in place.
- Generally, major design decisions are embodied in top-level modules, whereas most of the low-level modules largely perform commonly known input–output functions. Discovery of major flaws in system design may not be possible until the top-level modules have been integrated.

Now we compare the top-down and bottom-up approaches in the following:

- **Validation of Major Design Decisions:** The top-level modules contain major design decisions. Faults in design decisions are detected early if integration is done in a top-down manner. In the bottom-up approach, those faults are detected toward the end of the integration process.
- **Observation of System-Level Functions:** One applies test inputs to the top-level module, which is akin to performing system-level tests in a very limited way in the top-down approach. This gives an opportunity to the SIT personnel and the development team to observe system-level functions early in the integration process. However, similar observations can be done in the bottom-up approach only at the end of system integration.
- **Difficulty in Designing Test Cases:** In the top-down approach, as more and more modules are integrated and stubs lie farther away from the top-level module, it becomes increasingly difficult to design stub behavior and test input. This is because stubs return predetermined values, and a

test engineer must compute those values for a given test input at the top level. However, in the bottom-up approach, one designs the behavior of a test driver by simplifying the behavior of the actual module.

- **Reusability of Test Cases:** In the top-down approach, test cases designed to test the interface of a newly integrated module is reused in performing regression tests in the following iteration. Those test cases are reused as system-level test cases. However, in the bottom-up approach, all the test cases incorporated into test drivers, except for the top-level test driver, cannot be reused. The top-down approach saves resources in the form of time and money.

7.4.4 Sandwich and Big Bang

In the sandwich approach, a system is integrated by using a mix of the top-down and bottom-up approaches. A hierarchical system is viewed as consisting of three layers. The bottom layer contains all the modules that are often invoked. The bottom-up approach is applied to integrate the modules in the bottom layer. The top layer contains modules implementing major design decisions. These modules are integrated by using the top-down approach. The rest of the modules are put in the middle layer. We have the advantages of the top-down approach where writing stubs for the low-level module is not required. As a special case, the middle layer may not exist, in which case a module falls either in the top layer or in the bottom layer. On the other hand, if the middle layer exists, then this layer can be integrated by using the big-bang approach after the top and the bottom layers have been integrated.

In the big-bang approach, first all the modules are individually tested. Next, all those modules are put together to construct the entire system which is tested as a whole. Sometimes developers use the big-bang approach to integrate small systems. However, for large systems, this approach is not recommended for the following reasons:

- In a system with a large number of modules, there may be many interface defects. It is difficult to determine whether or not the cause of a failure is due to interface errors in a large and complex system.
- In large systems, the presence of a large number of interface errors is not an unlikely scenario in software development. Thus, it is not cost effective to be optimistic by putting the modules together and hoping it will work.

Solheim and Rowland [5] measured the relative efficacy of top-down, bottom-up, sandwich, and big-bang integration strategies for software systems. The empirical study indicated that top-down integration strategies are most effective in terms of defect correction. Top-down and big-bang strategies produced the most reliable systems. Bottom-up strategies are generally least effective at correcting defects and produce the least reliable systems. Systems integrated by the sandwich strategy are moderately reliable in comparison.

→ NO lo vamos a ver

7.5 SOFTWARE AND HARDWARE INTEGRATION

A component is a fundamental part of a system, and it is largely independent of other components. Many products require development of both hardware and software components. These two kinds of components are integrated to form the complete product. In addition, a third kind of component, a product documentation, is developed in parallel with the first two components. A product documentation is an integration of different kinds of individual documentations. The overall goal is to reduce the time to market of the product by removing the sequential nature of product development processes.

On the hardware side, the individual hardware modules, or components, are diverse in nature, such as a chassis, a printed circuit board, a power supply, a fan tray for cooling, and a cabinet to hold the product. On the documentation side, the modules that are integrated together include an installation manual, a troubleshooting guide, and a user's manual in more than one natural language.

It is essential to test both the software and the hardware components individually as much as possible before integrating them. In many products, neither component can be completely tested without the other. Usually, the entry criteria for both the hardware and software components are establishedand satisfied before beginning to integrate those components. If the target hardware is not available at the time of system integration, then a hardware emulator is developed. The emulator replaces the hardware platform on which the software is tested until the real hardware is available. However, there is no guarantee that the software will work on the real hardware even if it worked on the emulator.

Integration of hardware and software components is often done in an iterative manner. A software image with a minimal number of core software modules is loaded on the prototype hardware. In each step, a small number of tests are performed to ensure that all the desired software modules are present in the build. Next, additional tests are run to verify the essential functionalities. The process of assembling the build, loading on the target hardware, and testing the build continues until the entire product has been integrated. If a problem is discovered early in the hardware/software integration and the problem can be resolved easily, then the problem is fixed without any delay. Otherwise, integration of software and hardware components may continue in a limited way until the root cause of the problem is found and analyzed. The integration is delayed until the fixes, based on the outcome of the root cause analysis, are applied.

7.5.1 Hardware Design Verification Tests NO

A hardware engineering process is viewed as consisting of four phases: (i) planning and specification, (ii) design, prototype implementation, and testing, (iii) integration with the software system, and (iv) manufacturing, distribution, and field service. Testing of a hardware module in the second phase of hardware development without software can be conducted to a limited degree. A hardware design verification test (DVT) plan is prepared and executed by the hardware group before integration with the software system. The main hardware tests are discussed below.

Diagnostic Test Diagnostic tests are the most fundamental hardware tests. Such tests are often imbedded in the basic input–output system (BIOS) component and are executed automatically whenever the system powers up. The BIOS component generally resides on the system's read-only memory (ROM). This test is performed as a kind of sanity test of the hardware module. A good diagnostic test covers all the modules in the system. A diagnostic test is the first test performed to isolate a faulty hardware module.

Electrostatic Discharge Test The concept of electrostatic discharge (ESD) testing is very old and it ensures that the system operation is not susceptible to ESD after having taken commonly accepted precautions. There are three common industry standards on ESD testing based on three different models: the human body model (HBM), the machine model (MM), and the charged device model (CDM). The HBM is the oldest one, and it is the most widely recognized ESD model. It was developed at a time when most ESD damages occurred as people touched hardware components without proper grounding. The capacitance and impedance of the human body vary widely, so the component values in the model were arbitrarily set to facilitate comparative testing. Devices damaged by the HBM generally have thermally damaged junctions, melted metal lines, or other types of damages caused by a high peak current and a high charge dissipated over several hundred nanoseconds. This model still applies whenever people handle devices so one should perform HBM testing on all new devices.

The MM is used primarily in Japan and Europe. This model was developed originally as a "worst-case" HBM to duplicate the type of failures caused by automated pick-and-place machines used to assemble printed circuit boards (PCBs). The model simulates a low-impedance machine that discharges a moderately high capacitance (e.g., 200 pF) through a device. A discharge produced using the MM can cause damage at relatively low voltages. Finally, the CDM reproduces realistic ESD damages that occur in small, plastic-packaged devices. As a packaged device slides on a surface, it accumulates charge due to triboelectric (friction) action between the plastic body and the surface. Thus, the device picks up a charge that produces a potential. In the HBM and the MM, something external to the device accumulates the charge. For small devices, the potential can be surprisingly high. Potentials of at least 650 V are needed to duplicate the observed damage.

Electromagnetic Emission Test Tests are conducted to ensure that the system does not emit excessive radiation to impact operation of adjacent equipments. Similarly, tests are conducted to ensure that the system does not receive excessive radiation to impact its own operation. The emissions of concern are as follows:

- Electric field radiated emissions
- Magnetic field radiated emissions
- Alternating-current (AC) power lead conducted emission (voltage)
- AC and direct-current (DC) power and signal lead conducted emission (current)
- Analog voice band lead conducted emission

Electrical Test A variety of electrical tests are performed on products with a hardware component. One such test is called "signal quality" testing in which different parts of the system are checked for any inappropriate voltages or potential current flows at the externally accessible ports and peripherals. Another type of electrical test is observing how the system behaves in response to various types of power conditions such as AC, DC, and batteries. In addition, tests are conducted to check the safety limits of the equipment when it is exposed to abnormal conditions. An abnormal condition can result from lightning surges or AC power faults.

Thermal Test Thermal tests are conducted to observe whether or not the system can stand the temperature and humidity conditions it will experience in both operating and nonoperating modes. The system is placed in a thermal chamber and run through a series of temperature and humidity cycles. The heat producing components, such as CPU and Ethernet cards, are instrumented with thermal sensors to verify whether the components exceed their maximum operating temperatures. A special kind of thermal test is thermal shock, where the temperature changes rapidly.

Environmental Test Environmental tests are designed to verify the response of the system to various types of strenuous conditions encountered in the real world. One such test involves shock and vibration from adjacent constructions and highways. Nearby heavy machinery, heavy construction, heavy industrial equipment, truck/train traffic, or standby generators can result in low-frequency vibration which can induce intermittent problems. It is important to know if such low-frequency vibration will cause long-term problems such as connectors becoming loose or short-term problems such as a disk drive crashing. Environmental tests also include other surprises the system is likely to encounter. For example, in a battlefield environment, the computers and the base transceiver stations are often subjected to smoke, sand, bullets, fire, and other extreme conditions.

Equipment Handling and Packaging Test These tests are intended to determine the robustness of the system to normal shipping and installation activities. Good pack and packaging design ensures that the shipping container will provide damage-free shipment of the system. Early involvement of these design skills will provide input on the positioning of handles and other system protrusions that can be the sources of failure. Selection of reasonable metal thickness and fasteners will provide adequate performance of systems to be installed into their final location.

Acoustic Test Acoustic noise limits are specified to ensure that personnel can work near the system without exceeding the safety limits prescribed by the local Occupational Safety and Health Administration (OSHA) agency or other negotiated levels. For example, noise levels of spinning hard disks, floppies, and other drives must be tested against their limits.

Safety Test Safety tests are conducted to ensure that people using or working on or near the system are not exposed to hazards. For example, many portable computers contain rechargeable batteries which frequently include dangerous toxic substances such as cadmium and nickel. Adequate care must be taken to ensure that these devices do not leak the dangerous chemicals under any circumstances.

Reliability Test Hardware modules tends to fail over time. It is assumed that (i) modules have constant failure rates during their useful operating life periods and (ii) module failure rates follow an exponential law of distribution. Failure rate is often measured in terms of the mean time between failures (MTBF), and it is expressed as MTBF = total time/number of failures. The probability that a module will work for some time T without failure is given by $R(T) = \exp(-T/\text{MTBF})$. The MTBF metric is a reliability measurement metric for hardware modules. It is usually given in units of hours, days, or months.

The MTBF for a module or a system is derived from various sources: laboratory tests, actual field failure data, and prediction models. Another way of calculating the reliability and lifetime of a system is to conduct highly accelerated life tests (HALTs). The HALTs rely on the principle of *logarithmic time compression* to simulate a system's entire life in just a few days. This is done by applying a much higher level of stress than what exists in actual system use. A high level of stress forces failures to occur in significantly less time than under normal operating conditions. The HALTs generally include rapid temperature cycling, vibrating on all axes, operating voltage variations, and changing clock frequency until the system fails. The HALTs require only a few units of the product and a short testing period to identify the fundamental limits of the technologies in use. Generally, every weak point must be identified and fixed (i.e., redesigned) if it does not meet the system's specified limits. Understanding the concept of product reliability is important for any organization if the organization intends to offer warranty on the system for an extended period of time. One can predict with high accuracy the exact cost associated with the returns over a limited and an extended period of warranty. For example, for a system with an MTBF of 250,000 hours and an operating time of interest of five years (438,000 hours), we have $R(438{,}000) = \exp(-43{,}800/250{,}000) = 0.8392$, which says that there is a probability of 0.8932 that the product will operate for five years without a failure. Another interpretation of the quantity is that 83.9% of the units in the field will still be working at the end of five years. In other words, 16.1% of the units need to be replaced within the first five years.

7.5.2 Hardware and Software Compatibility Matrix ND

The hardware and software compatibility information is maintained in the form of a compatibility matrix. Such a matrix documents the compatibility between different revisions of the hardware and different versions of the software and is used for official release of the product. An engineering change order (ECO) is a formal document that describes a change to the hardware or software. An ECO document includes the hardware/software compatibility matrix and is distributed to

TABLE 7.2 Example Software/Hardware Compatibility Matrix

Software Release	RNC Hardware Version	RN Hardware Version	EMS Hardware Version	PDSN Hardware Version	Tested by SIT	Tested by ST
2.0	hv1.0	hv2.0	hv3.2	hv2.0.3	Yes	Yes
2.5	hv2.0 and hv1.0	hv2.0	hv4.0 and hv3.2	hv3.0 and hv2.0.3	Yes	Yes
3.0	hv3.0	hv3.0 and hv2.0	hv5.0	hv4.0 and hv3.0	Yes	Not yet
3.0	hv4.0 and hv3.0	hv3.0	hv5.0	hv4.5	Not yet	Not yet
Not yet decided	hv4.0 and hv3.0	hv3.0	hv6.0	hv5.0	Not yet	Not yet

the operation, customer support, and sales teams of the organization. An example compatibility matrix for a 1xEV-DO wireless data network, discussed in Chapter 8, is given in Table 7.2.

In the following, we provide the hardware and software ECO approval process in an organization. The first scenario describes the ECO process to incorporate a new hardware in the product. The second scenario describes the ECO process for a software revision.

Scenario 1 A hardware ECO process is shown in Figure 7.11. Assume that the hardware group needs to release a new revision of a hardware or has to recommend a new revision of an original equipment manufacturer (OEM) hardware to the operation/manufacturing group. The steps of the ECO process to incorporate the new hardware in the product are as follows:

1. The hardware group issues a design change notification for the new hardware revision. This notification includes identification of specific hardware changes likely to impact the software and incompatibilities between the revised hardware and other hardware. The software group reviews the notification with the hardware group to assess the impact of the hardware changes and to identify software changes that affect the hardware structure.
2. The hardware group creates an ECO and reviews it with the change control board (CCB) to ensure that all impacts of the ECO are understood and agreed upon and that the version numbering rules for software components are followed. The CCB constitutes a group of individuals from multiple departments responsible for reviewing and approving each ECO.
3. The ECO is released, and the hardware group updates the hardware/software compatibility matrix based on the information received from the review process.

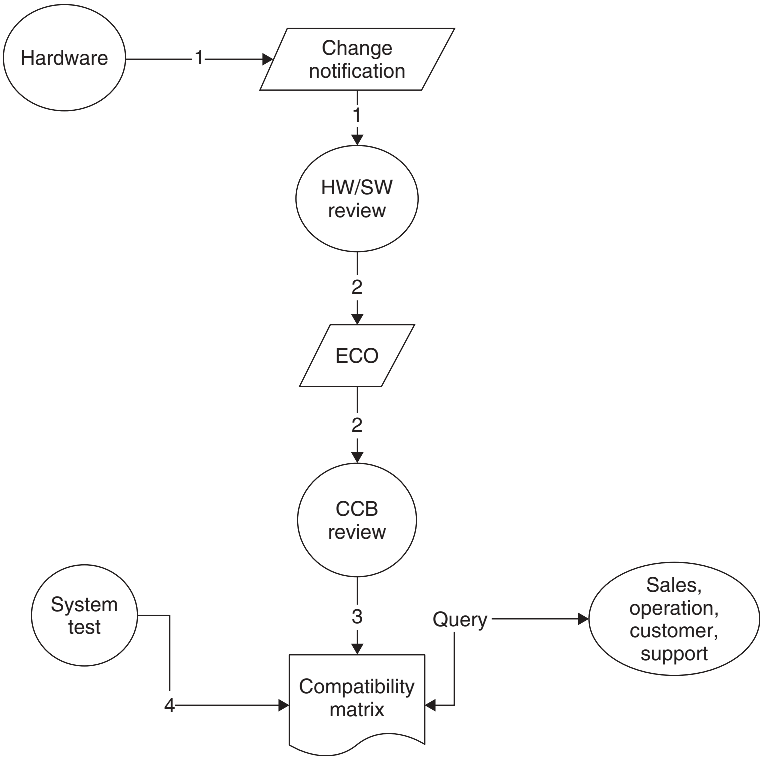

Figure 7.11 Hardware ECO process.

4. The system testing group updates the compatibility matrix after it has tested a given combination of released hardware and software versions.

Scenario 2 A software release ECO process is shown in Figure 7.12. Assume that the group needs to release a new version of software to the operation/manufacturing group. The steps of the ECO process to incorporate the new version of the software product are as follows:

1. The system integration group releases a build with a release note identifying the hardware compatibility information to the system test group.
2. The system test group tests the build and notifies the software group and other relevant groups in the organization of the results of the tests.
3. The system test group deems a particular build to be viable for customer release. The system test group calls a cross-functional readiness review meeting to ensure, by verifying the test results, that all divisions of the organization are prepared for an official release.
4. The software group writes an ECO to officially release the software build and reviews it with the CCB after the readiness review is completed.

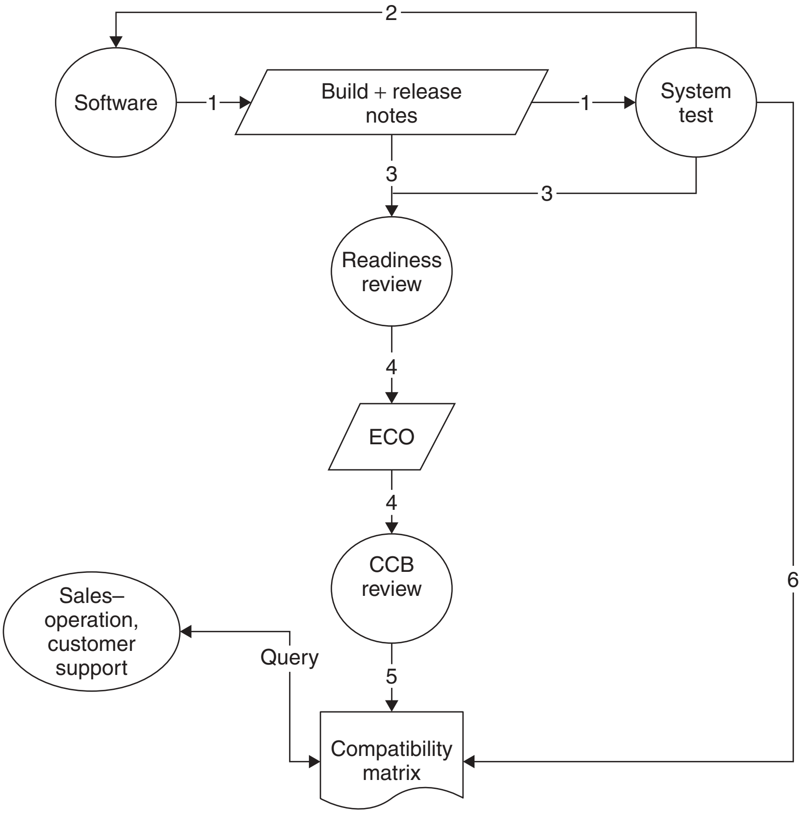

Figure 7.12 Software ECO process.

The build is considered to be released after the ECO is approved and documented.

5. The software group updates the hardware/software compatibility matrix with information about the new release.
6. The system test group updates the compatibility matrix after it has tested a given combination of released hardware and software versions.

7.6 TEST PLAN FOR SYSTEM INTEGRATION NO ahora!

System integration requires a controlled execution environment, much communication between the developers and the test engineers, judicious decision making along the way, and much time, on the order of months, in addition to the fundamental tasks of test design and test execution. Integrating a large system is a challenging task, which is handled with much planning in the form of developing a SIT plan. A useful framework for preparing an SIT plan is outlined in Table 7.3.

TABLE 7.3 Framework for SIT Plan

1. Scope of testing
2. Structure of integration levels
   a. Integration test phases
   b. Modules or subsystems to be integrated in each phase
   c. Building process and schedule in each phase
   d. Environment to be set up and resources required in each phase
3. Criteria for each integration test phase *n*
   a. Entry criteria
   b. Exit criteria
   c. Integration Techniques to be used
   d. Test configuration set-up
4. Test specification for each integration test phase
   a. Test case ID number
   b. Input data
   c. Initial condition
   d. Expected results
   e. Test procedure
      How to execute this test?
      How to capture and interpret the results?
5. Actual test results for each integration test phase
6. References
7. Appendix

In the scope of testing section, one summarizes the system architecture. Specifically, the focus is on the functional, internal, and performance characteristics to be tested. System integration methods and assumptions are included in this section.

The next section, structure of integration levels, contains four subsections. The first subsection explains the division of integration testing into different phases, such as functional, end-to-end, and endurance phases. The second subsection describes the modules to be integrated in each of the integration phases. The third subsection describes the build process to be followed: daily build, weekly build, biweekly build, or a combination thereof. A schedule for system integration is given in the third subsection. Specifically, one identifies the start and end dates for each phase of testing. Moreover, the availability windows for unit-tested modules are defined. In the fourth subsection, the test environment and the resources required are described for each integration phase. The hardware configuration, emulators, software simulators, special test tools, debuggers, overhead software (i.e., stubs and drivers), and testing techniques are discussed in the fourth subsection.

An important decision to be made for integration testing is establishing the start and stop dates of each phase of integration testing. The start date and stop date for a phase are specified in terms of *entry criteria* and *exit criteria*, respectively. These criteria are described in the third section of the plan. A framework for defining entry criteria to start system integration is given in Table 7.4. Similarly, the exit criteria for system integration are given in Table 7.5. Test configuration and integration techniques (e.g., top down or bottom up) to be used in each of these phases are described in this section.

The test specification section describes the test procedure to be followed in each integration phase. The detailed test cases, including the input and expected outcome for each case, are documented in the test specification section. The history of actual test results, problems, or peculiarities is recorded in the fifth section of the SIT plan. Finally, references and an appendix, if any, are included in the test plan.

System integration testing is performed in phases of increasing complexity for better efficiency and effectiveness. In the first phase interface integrity and functional validity within a system are tested. In the second phase end-to-end and pairwise tests are conducted. Finally, in the third phase, stress and endurance tests are performed. Each of the system integration phases identified in the SIT plan delineates a broad functionality category within the software structure, and it can be related to a specific domain of the system software structure. The categories

TABLE 7.4 Framework for Entry Criteria to Start System Integration

Softwave functional and design specifications must be writen, reviewed, and approved.
Code is reviewed and approved.
Unit test plan for each module is written, reviewed, and executed.
All of the unit tests passed.
The entire check-in request form must be completed, submitted, and approved.
Hardware design specification is written, reviewed, and approved.
Hardware design verification test is written, reviewed, and executed.
All of the design verification tests passed.
Hardware/software integration test plan is written, reviewed, and executed.
All of the hardware/software integrated tests passed.

TABLE 7.5 Framework for System Integration Exit Criteria

All code is completed and frozen and no more modules are to be integrated.
All of the system integration tests passed.
No major defect is outstanding.
All the moderate defects found in the SIT phase are fixed and retested.
Not more than 25 minor defects are outstanding.
Two weeks system uptime in system integration test environment without any anomalies, i.e., crashes.
System integration tests results are documented.

of system integration tests and the corresponding test cases discussed below are applicable for the different test phases.

Interface Integrity Internal and external interfaces are tested as each module is integrated into the structure. When two modules are integrated, the communication between them is called internal, whereas when they communicate with the outside environment, it is called external. Tests that are required to be designed to reveal interface errors are discussed in Section 7.2. An important part of interface testing is to map an incoming message format to the expected message format of the receiving module and ensure that the two match. Tests are designed for each message passing through the interface between two modules. Essentially, tests must ensure that:

- The number of parameters sent in a message agree with the number of parameters expected to be received.
- The parameter order in the messages match the order expected.
- The field sizes and the data types match.
- The boundaries of each data field in a message match the expected boundaries.
- When a message is generated from stored data prior to being sent, the message truly reflects the stored data.
- When a received message is stored, data copying is consistent with the received message.

Functional Validity Tests are designed to uncover functional errors in each module after it is integrated with the system. Errors associated with local or global data structures are uncovered with such tests. Selected unit tests that were designed for each module are reexecuted during system integration by replacing the stubs with their actual modules.

End-to-End Validity Tests are performed to ensure that a completely integrated system works together from end to end. Interim checkpoints on an end-to-end flow provide a better understanding of internal mechanisms. This helps in locating the sources of failures.

Pairwise Validity Tests are performed to ensure that any two systems work properly when connected together by a network. The difference between pairwise tests and end-to-end tests lies in the emphasis and type of test cases. For example, a toll-free call to an 800 number is an end-to-end test of a telephone system, whereas a connectivity test between a handset and local private branch exchange (PBX) is a pairwise test.

Interface Stress Stress is applied at the module level during the integration of the modules to ensure that the interfaces can sustain the load. On the other hand, full-scale system stress testing is performed at the time of system-level testing. The following areas are of special interest during interface stress testing:

- **Error Handling:** Trigger errors that should be handled by the modules.
- **Event Handling:** Trigger events (e.g., messages, timeouts, callbacks) that should be handled by the modules.
- **Configuration:** Repeatedly add, modify, and delete managed objects.
- **Logging:** Turn on the logging mechanism during stress tests to ensure proper operation for boundary conditions.
- **Module Interactions:** Run tests repeatedly that stress the interactions of a module with other modules.
- **CPU Cycles:** Induce high CPU utilization by using a CPU overload tool; pay attention to any resulting queue overflows and other producer/consumer overrun conditions.
- **Memory/Disk Usage:** Artificially reduce the levels of heaps and/or memory buffers and disk space;
- **Starvation:** Ensure that the processes or tasks are not starved; otherwise eventually the input queues overflow for the starved processes.
- **Resource Sharing:** Ensure that resources, such as heap and CPU, are shared among processes without any contention and bottlenecks.
- **Congestion:** Run tests with a mechanism that randomly discards packets; test modules with congested links.
- **Capacity:** Run tests to ensure that modules can handle the maximum numbers of supporting requirements such as connections and routes.

System Endurance A completely integrated system is expected to stay up continuously for weeks without any crashes. In the case of communication protocol systems, formal rules govern that two systems communicate with each other via an interface, that is, a communication channel. The idea here is to verify that the format and the message communication across the interface of the modules works for an extended period.

7.7 OFF-THE-SHELF COMPONENT INTEGRATION NO

Instead of developing a software component from scratch, organizations occasionally purchase off-the-shelf (OTS) components form third-party vendors and integrate them with their own components [6]. In this process, organizations create less expensive software systems. A major issue that can arise while integrating different components is mismatches among code pieces developed by different parties usually unaware of each other [7, 8].

Vigder and Dean [9] have presented elements of an architecture for integration and have defined rules that facilitate integration of components. They have identified a useful set of supporting components for integrating the actual, serving components. The supporting components are *wrappers*, *glue*, and *tailoring*.

A wrapper is a piece of code that one builds to isolate the underlying components from other components of the system. Here *isolate* means putting restrictions around the underlying component to constrain its capabilities. A glue component provides the functionality to combine different components. Component tailoring refers to the ability to enhance the functionality of a component. Tailoring is done by adding some elements to a component to enrich it with a functionality not provided by the vendor. Tailoring does not involve modifying the source code of the component. An example of tailoring is “scripting,” where an application can be enhanced by executing a script upon the occurrence of some event. Rine et al. [10] have proposed the concept of adapters to integrate components. An adapter is associated with each component; the adapter runs an interaction protocol to manage communications among the components. Components request services from others through their associated adapters. An adapter is responsible for resolving any syntactic interface mismatch.

7.7.1 Off-the-Shelf Component Testing

NO

Buyer organizations perform two types of testing on an OTS component before purchasing: (i) acceptance testing of the OTS component based on the criteria discussed in Chapter 14 and (ii) integration of the component with other components developed in-house or purchased from a third party. The most common cause of problems in the integration phase is inadequate acceptance testing of the OTS component. A lack of clear documentation of the system interface and less cooperation from the vendor may create an ordeal in integration testing, debugging, and fixing defects. Acceptance testing of an OTS component requires the development and execution of an acceptance test plan based on the acceptance criteria for the candidate component. All the issues are resolved before the system integration process begins. During integration testing, additional software components, such as a glue or a wrapper, can be developed to bind an OTS component with other components for proper functioning. These new software components are also tested during the integration phase. Integration of OTS components is a challenging task because of the following characteristics identified by Basili and Boehm [11]:

> The buyer has no access to the source code.
>
> The vendor controls its development.
>
> The vendor has nontrivial installed base.

Voas [12] proposed three types of testing techniques to determine the suitability of an OTS component:

- **Black-Box Component Testing:** This is used to determine the quality of the component.
- **System-Level Fault Injection Testing:** This is used to determine how well a system will tolerate a failing component. System-level fault injection does not demonstrate the reliability of the system; instead, it can predict the behavior of the system if the OTS component fails.

- **Operational System Testing:** This kind of test is used to determine the tolerance of a software system when the OTS component is functioning correctly. Operational system testing is conducted to ensure that an OTS component is a good match for the system.

The OTS components produced by the vendor organizations are known as commercial off-the-shelf (COTS) components. A COTS component is defined by Szyperski et al. [13] (p. 34) as “a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” Interfaces are the access points of COTS components through which a client component can request a service declared in an interface of the service providing component. Weyuker [14] recommends that vendors building COTS components must try to envision many possible uses of the component and develop a comprehensive range of test scenarios. A component should be defect free because prudent buyers will perform significant amount of testing while integrating the component with their system. If potential buyers encounter a large number of defects in a COTS component, they may not accept the component. Therefore, it is in the best interest of the component builder to ensure that components are thoroughly tested. Several related artefacts should be archived and/or modified for each COTS component, because a potential buyer may demand these artifacts to be of high quality. The artefacts to be archived include the following:

- Individual requirements, including the pointers between the software functional specification and the corresponding implementations: This information makes it easy to track when either the code or the specification is modified, which implies that the specification remains up to date.
- The test suite, including the requirement traceability matrix: This will show which part of the functionality is tested inadequately or not at all. In addition, it identifies the test cases that (i) must be executed as regression tests or (ii) need to be updated when the component undergoes a change.
- The individual pass–fail result of each test case in the test suite: This indicates the quality of the COTS component.
- The details of individual test cases, including input and expected output: This facilitates regression testing of changes to a component.
- The system test report corresponding to the final release of the COTS component, which includes performance characteristics, scalability limitation, stability observations, and interoperability of the COTS component: This document is a useful resource for potential buyers before they begin acceptance testing of the COTS components.

7.7.2 Built-in Testing NO

A component reused in a new application environment requires real-time detection, diagnosis, and handling of software faults. Built-in test (BIT) methods for producing self-testable software components hold potential for detecting faults during run time. A software component can contain test cases or can possess facilities that are capable of generating test cases which can be accessed by a component user on demand [15]. The corresponding capabilities allowing this are called built-in testing capabilities of software components. In the BIT methodology, testability is incorporated into software components, so that testing and maintenance can be self-contained.

Wang et al. [16] have proposed a BIT model that can operate in two modes, namely, *normal* mode and *maintenance* mode. In the normal mode, the BIT capabilities are transparent to the component user, and the component does not differ from other non-BIT-enabled components. In the maintenance mode, however, the component user can test the component with the help of its BIT features. The component user can invoke the respective methods of the component, which execute the test, evaluate autonomously its results, and output the test summary. The authors describe a generic technical framework for enhancing BIT. One of their assumptions is that the component is implemented as a class. A benefit of such an implementation is that the methods for BIT can be passed to a subclass by inheritance.

Hörnstein and Edler [17] have proposed a *component + BIT* architecture comprising three types of components, namely, BIT, testers, and handlers. The BIT components are the BIT-enabled components. These components implement certain mandatory interfaces. Testers are components which access the BIT capabilities of BIT components through the corresponding interfaces and contain the test cases in a certain form. Finally, handlers are components that do not directly contribute to testing but provide recovery mechanisms in case of failures.

7.8 SUMMARY

This chapter began with the objective and a description of system integration testing. One creates a "working version of the system" by putting the modules together in an incremental manner while performing tests to uncover different types of errors associated with interfacing. Next, we explored various levels of granularities in system integration testing: intrasystem testing, intersystem testing, and pairwise testing.

We then examined five types of commonly used system integration techniques: topdown, bottomup, sandwich, bigbang, and incremental. We compared those techniques in detail. The incremental technique is widely used in the industry.

We described the integration of hardware and software components to form a complete product. This led to the discussion of the hardware engineering process and, specifically, of different types of hardware design verification tests: diagnostic, electrostatic discharge, electromagnetic emission, electrical, thermal, environmental, packaging and handling, acoustic, safety, and reliability. Finally, we described two scenarios of an engineering change order process. The two scenarios are used to keep track of the hardware/software compatibility matrix of a released product.

We provided a framework of an integration test plan. The following categories of tests, which are included in an integration test plan, were discussed in detail: interface integrity tests, functional validity tests, end-to-end validity tests, pairwise validity tests, interface stress tests, and endurance tests.

Finally, we described the integration of OTS components with other components. An organization, instead of developing a software component from scratch, may decide to purchase a COTS software from a third-party source and integrate it with its own software system. A COTS component seller must provide BITs along with the components, whereas a buyer organization must perform three types of testing to assess the COTS software: (i) acceptance testing of the OTS software component to determine the quality of the component, (ii) system-level fault injection testing, which is used to determine the tolerance of the software system to a failing OTS component, and (iii) operational system testing, which is used to determine the tolerance of the software system to a properly functioning OTS component.

LITERATURE REVIEW

For those actively involved in software testing or interested in knowing more about common software errors, appendix A of the book by C. Kaner, J. Falk, and H. Q. Nguyen (*Testing Computer Software*, Wiley, New York, 1999) is an excellent repository of real-life software errors. The appendix contains 12 categories of approximately 400 different types of errors with illustrations.

A good discussion of hardware test engineering, such as mechanical, electronics, and accelerated tests, can be found in Patrick O'Connor's book (*Test Engineering: A Concise Guide to Cost-effective Design, Development and Manufacture*, Wiley, New York, 2001). The author (i) describes a broad spectrum of modern methods and technologies in hardware test engineering, (ii) offers principles of cost-effective design, development, and manufacture of products and systems, and (iii) gives a breakdown of why product and systems fail and which methods would best prevent these failures.

Researchers are continuously working on topics related to certification of COTS components. The interested reader is recommended to read the following articles. Each of these articles helps in understanding the issues of software certification and why it is important:

J. Voas, "Certification Reducing the Hidden Costs of Poor Quality," *IEEE Software*, Vol. 16, No. 4, July/August 1999, pp. 22–25.

J. Voas, "Certifying Software for High-Assurance Environments," *IEEE Software*, Vol. 16, No. 4, July/August 1999, pp. 48–54.

J. Voas, "Developing Usage-Based Software Certification Process," *IEEE Computer*, Vol. 16, No. 8, August 2000, pp. 32–37.

S. Wakid, D. Kuhn, and D. Wallace, "Toward Credible IT Testing and Certification," *IEEE Software*, Vol. 16, No. 4, July/August 1999, pp. 39–47.

Readers actively involved in COTS component testing or interested in a more sophisticated treatment of the topic are recommended to read the book edited by S. Beydeda and V. Gruhn (*Testing Commercial-off-the-Shelf Components and Systems*, Springer, Bonn, 2005). The book contains 15 articles that discuss in great detail: (i) testing components context independently, (ii) testing components in the context of a system, and (iii) testing component–based systems. The book lists several excellent references on the subject in a bibliography.

REFERENCES

1. V. R. Basili and B. T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, January 1984, pp. 42–52.
2. D. E. Perry and W. M. Evangelist. An Empirical Study of Software Interface Faults—An Update. In *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, Hawaii Vol. II, IEEE Computer Society Press, Piscataway, NJ, January 1987, pp. 113–126.
3. M. S. Deutsch. *Software Verification and Validation: Realistic Project Approaches*. Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 95–101.
4. M. Cusumano and R. W. Selby. How Microsoft Builds Software. *Communications of the ACM*, June 1997, pp. 53–61.
5. J. A. Solheim and J. H. Rowland. An Empirical Study of Testing and Integration Strategies Using Artificial Software Systems. *IEEE Transactions on Software Engineering*, October 1993, pp. 941–949.
6. S. Mahmood, R. Lai, and Y. S. Kim. Survey of Component-Based Software Development. *IET Software*, April 2007, pp. 57–66.
7. G. T. Heineman and W. T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Reading, MA, 2001.
8. A. Cechich, M. Piattini, and A. Vallecillo. Assessing Component Based Systems. Component Based Software Quality. *Lecture Notes in Computer Science*, Vol. 2693, 2003, pp. 1–20.
9. M. R. Vigder and J. Dean. An Architectural Approach to Building Systems from COTS Software Components. In *Proceedings of the 22nd Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, MD, December 1997, pp. 99–113, NRC41610.
10. D. Rine, N. Nada, and K. Jaber. Using Adapters to Reduce Interaction Complexity in Reusable Component-Based Software Development. In *Proceedings of the 1999 Symposium on Software Reusability*, Los Angeles, CA, ACM Press, New York, May 1999, pp. 37–43.
11. V. R. Basili and B. Boehm. COTS-Based Systems Top 10 List. *IEEE Computer*, May 2001, pp. 91–93.
12. J. Voas. Certifying Off-the-Shelf Software Component. *IEEE Computer*, June 1998, pp. 53–59.
13. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley, Reading, MA, 2002.
14. E. J. Weyuker. Testing Component-Based Software: A Cautionary Tale. *IEEE Software*, September/October 1998, pp. 54–59.
15. S. Beydeda and V. Gruhn. Merging Components and Testing Tools: The Self-Testing COTS Components (STECC) Strategy. In *Proceedings of the 29th EUROMICRO Conference (EUROMICRO'03)*, Belek-Antalya, Turkey, IEEE Computer Society Press, Piscataway, September 2003, pp. 107–114.
16. Y. Wang, G. King, and H. Wickburg. A Method for Built-in Tests in Component-Based Software Maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance and Reengineering (CSMR-99)*, University of Amsterdam, The Netherland, IEEE Computer Society Press, Piscataway, March 1999, pp. 186–189.

17. J. Höornstein and H. Edler. Test Reuse in CBSE Using Built-in Tests. In *Proceedings of the 9th IEEE Conference and Workshops on Engineering of Computer Based Systems*, Workshop on Component-Based Software Engineering, Lund University, Lund, Sweden, IEEE Computer Society Press, Piscataway, 2002.

Exercises

1. Describe the difference between black-box and white-box testing techniques?
2. If a program passes all the black-box tests, it means that the program should work properly. Then, in addition to black-box testing, why do you need to perform white-box testing?
3. Describe the difference between unit testing and integration testing?
4. Why should integration testing be performed? What types of errors can this phase of testing reveal?
5. Discuss the advantages and disadvantages of top-down and bottom-up approaches to integration testing.
6. Does automation of integration tests help the verification of the daily build process? Justify your answer.
7. Using the module hierarchy given in Figure 7.13, show the orders of module integration for the top-down and bottom-up integration approaches. Estimate the number of stubs and drivers needed for each approach. Specify the integration testing activities that can be done in parallel, assuming you have three SIT engineers. Based on the resource needs and the ability to carry out concurrent SIT activities, which approach would you select for this system and why?
8. Suppose that you plan to purchase COTS components and integrate them with your communication software project. What kind of acceptance criteria will you develop to conduct acceptance testing of the COTS components?
9. During integration testing of COTS components with a software system, it may be required to develop a wrapper software around the OTS component to limit what it can do. Discuss the general characteristics that a wrapping software

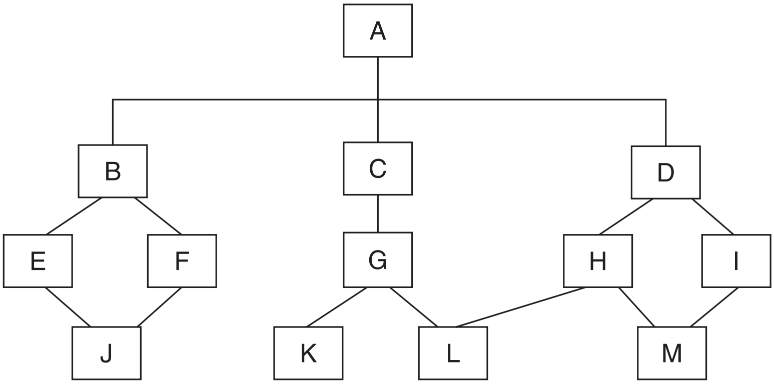

Figure 7.13 Module hierarchy of software system.

should have in order to be able to integrate COTS with the software system without any problem.

10. Describe the circumstances under which you would apply white-box testing, back-box testing, or both techniques to evaluate a COTS component.
11. For your current test project, develop an integration test plan.
12. Complete Section 5 (i.e., actual test results for each integration test phase) of the integration test plan after executing the integration test cases you developed in exercise 11.