CHAPTER

9

LEVELS OF TESTING

OBJECTIVES

This chapter details verification and validation activities associated with various stages of software development life cycle starting from proposal. It also covers various ways of integration testing.

9.1 INTRODUCTION

As seen in 'V testing' approach earlier, testing is a life-cycle activity which starts at the time of proposal and ends when acceptance testing is completed and product is finally delivered to customer. Testing begins with a proposal for software/system application development/maintenance, and ends when the system is formally accepted by user/customer. Different agencies/stakeholders are involved in conducting specialised testing required for the specific application. Definition of these stakeholders/agencies depends on the type of testing involved and level of testing to be done in various stages of software development life cycle. Table 9.1 shows in brief the level of testing performed by different agencies. It is an indicative list which may differ from customer to customer, product to product and organisation to organisation.

9.2 PROPOSAL TESTING

A proposal is made to the customer on the basis of Request for Proposal (RFP) or Request for Information (RFI) or Request for Quotation (RFQ) by the customer. Before making any proposal, the supplier must understand the purpose of such request, and devise the proposed solution accordingly. One must understand customer problem and the possible solution. Any proposal prepared in response to such request is reviewed by an organisation before sending it to the customer. It is reviewed by different panels or groups in the organisation such as technical group and commercial group. The intention is to ensure that the organisation must be able to stand by its words, if the proposal gets converted into a contract. The organisation must assess the impact of proposal on the organisation as well as on the prospect/customer.

Technical Review Technical review mainly involves technical feasibility of the kind of system or application proposed, the availability and requirement of skill sets, the hardware/software and other requirements of system. The proposal is also reviewed on the basis of time required for developing such system, and efforts required to make the proposal successful. The technical proposal is a description of overall approach, and not

Table 9.1 Different agencies involved at different levels of testing

Testing	Agencies involved
Proposal review	Customer, Business Analyst, System Analyst, Project Manager
Proposal testing	Customer, Business Analyst, System Analyst, Project Manager
Requirement review	Customer, Business Analyst, System Analyst, Project Manager, Project Leader, Test Leader
Requirements testing	Customer, Business Analyst, System Analyst, Project Manager, Project Leader, Test Leader
Design review	Customer, Business Analyst, System Analyst, Architect, Project Manager, Project Leader, Test Leader, Developer
Design testing	Customer, Business Analyst, System Analyst, Architect, Project Manager, Project Leader, Test Leader, Developer
Code review	Project Leader, Project Team, Customer
Test artifact review	Project Leader, Project Team, Customer, Tester, Test Leader
Unit testing	Project Leader, Project Team, Customer
Module testing	Project Leader, Project Team, Tester, Customer
Integration testing	Project Leader, Project Team, Tester, Customer
System testing	Project Manager, Test Leader, Tester, Customer
Acceptance testing	Project Manager, Tester, User/Customer

the final accepted approach solution/method and may undergo many iterations of changes as per discussions with customer. Requirement and estimations at the stage of proposal are not very specific but rather, they are indicative. It works on rough-cut methods, and estimations are ball-park figures that prospect may expect.

Commercial Review A proposal undergoes financial feasibility and other types of feasibilities involved with respect to the business. Commercial review may stress on the gross margins of the project and fund flow in terms of money going out and coming in the development organisation. Generally, total payment is split into installations depending upon completion of some phases of development activity. As different phases are completed, money is realised at those instances.

Several iterations of proposals and scope changes may happen before all the parties involved in Request for Quotation (RFQ) and proposal agree on some terms and conditions for doing a project.

Validation of Proposal A proposal sometimes involves development of prototypes or proof of concept to explain the proposed approach to the problem of the customer. One must define the approach of handling customer problem in the model or prototype.

In case of product organisation, the product development group along with marketing and sales functions decide about the new release of a product.

9.3 REQUIREMENT TESTING

Requirement creation involves gathering customer requirements and arranging them in a form to verify and validate them. The requirements may be categorised into different types such as technical, economical,

legal, operational and system requirements. Similarly, requirements may be specific, generic, present, future, expressed, implied, etc. The customer may be unaware of many of these requirements. The business analyst and system analyst must study the customer's line of business, problem and solution that the customer is looking for before arriving at these requirements. Assumed, implied or intended requirement is a gray area and may be a reason for many defects in the final product. Requirement testing makes sure that requirements defined in requirement specifications meet the following criteria.

- *Clarity* All requirements must be very clear in their meaning and what is expected from them. Requirements must state very clearly what the expected result of each transaction is, and who are the actors taking part in the transactions. Anything which hampers the clarity of requirements must be removed or clarified.

Illustration 9.1

Many requirement statements have words like 'Application must be user friendly' or 'Application performance must be fair'. Such requirements cannot be implemented as nobody knows the exact requirements.

One may have to raise queries to understand the meaning of user friendlyness or fair performance. If possible such statements must have some numerical figures to illustrate them. Tester will be able to write exact test case on the basis of these numbers.

- *Complete* Requirement statement must be complete and must talk about all business aspects of the application under development. It must consider all permutations and combinations possible when application is used in production. Any assumption must be documented and approved by the customer.

Illustration 9.2

Sometimes requirement statement does not specify the transaction. If requirement statement says 'Enter valid Login and Password', then it is not a complete requirement as (it does not specify how one should enter valid Login and how to go to Password. It could be by 'Tab' or by clicking the mouse or it will automatically go to Password).

If any part of transaction requires an assumption, it indicates incomplete requirements. One has to go back to customer or business analyst/system analyst to get this clarified.

- *Measurable* Requirements must be measurable in numeric terms as far as possible. Such definition helps to understand whether the requirement has been met or not while testing the application. Qualitative terms like 'good, 'sufficient', and 'high' must be avoided as different people may attach different meanings to these words.

Illustration 9.3

Sometimes, requirement statement may states that—enter valid credential and click 'Login' to go to next page. It is not very clear how much time it will take to reach next page.

These can not be verified under usability as performance testing.

- *Testable* The requirement must help in creating use cases/scenario which can be used for testing the application. Any requirement which cannot be tested must be drilled down further, to understand what is to be achieved by implementing these requirements.
- *Not Conflicting* The requirements must not conflict with each other. There is a possibility of trade-off between quality factors which needs to be agreed by the customer. Conflicting requirements indicate possible problem in requirement collection process.
- *Identifiable* The requirements must be distinctively identifiable. They must have numbers or some other way which can help in creating requirement traceability matrix. Indexing requirement is very important.

Validation of Requirements Requirement testing involves writing use cases using requirement statement. No assumption is to be made while writing use cases from requirement statement. If complete use cases can be developed using the requirement statement, requirements can be considered as clear, complete, measurable, testable and not conflicting with each other. Each assumption indicates lacunae in requirement statement. One has to ask the customer whether the assumptions are correct or not. As per feedback given by the customer, requirement statements must be updated.

9.4 DESIGN TESTING

Once the requirement statement satisfies all characteristics defined above, the system architect starts with system high-level architectural designing. The designs made by system architects must be traceable to requirements. The system designer starts building the low-level or detail design with the help of architectural design. Low-level design must be traceable to architectural design which in turn is traceable to requirements. The design must also possess the characteristics given below.

- *Clarity* A design must define all functions, components, tables, stored procedures, and reusable components very clearly. It must define any interdependence between different components and inputs/outputs from each module. It must cover the information to and from the application to other systems.
- *Complete* A design must be complete in all respect. It must define the parameters to be passed/received, formats of data handled, etc. Once the design is finalised, programmers must do their work of implementing designs mechanically and system must be working properly.
- *Traceable* A design must be traceable to requirements. The second column of requirement traceability matrix is a design column. The project manager must check if there is any requirement which does not have corresponding design or vice versa. It indicates a defect if traceability cannot be established completely.
- *Implementable* A design must be made in such a way that it can be implemented easily with selected technology and system. It must guide the developers in coding and compiling the code. The design must include interface requirements where different components are communicating with each other and with other systems.
- *Testable* Testers make structural test cases on the basis of design. Thus, a good design must help testers in creating structural test cases.

Validation of Design Design testing includes creation of data flow diagrams, activity diagrams, information flow diagram, and state transition diagram to show that information can flow through the system completely. Where the flow gets interrupted, it indicates that design is not complete. Flow of data and information in the system must complete the loop. It must consider the data definitions, attributes and input/output formats. Another way of testing design is creating prototypes from design.

9.5 CODE REVIEW

Code reviews include reviewing code files, database schema, classes, object definitions, procedures, and methods. Code review is applied to ensure that the design is implemented correctly by the developers, and guidelines and standards available for the purpose of coding are followed correctly. Code must have following characteristics.

- *Clarity* Code must be written correctly as per coding standards and syntax, requirements for the given platform. It must follow the standards and guidelines defined by the organisation/project/customer, as the case may be. It must declare variables, functions, and loops very clearly with proper comments. Code commenting must include which design part has been implemented, author, date, revision number, etc so that it can be traced to requirements and design.

- *Complete* Code, class, procedure, and method must be complete in all respect. It must suffice the purpose for which it is created. One class doing multiple things, or multiple objects created for same purpose indicates a problem in design. Code must be readable and compilable. Proper indenting, and variable initialisation must be followed as defined by coding standards.

- *Traceable* Code must be traceable with design components. It must declare clearly about the requirements, design, author, date, etc. Code files having no traceability to design can be considered as redundant code which will never get executed even if design is correct.

- *Maintainable* Code must be maintainable. Any developer with basic knowledge and training about coding must be able to handle the code in future while maintaining or bug fixing it.

9.6 UNIT TESTING

Unit is the smallest part of a software system which is testable. It may include code files, classes and methods which can be tested individually for correctness. Unit testing is a validation technique using black box methodology. Black box testing mainly concentrates on requirements of the system. In unit testing, units are tested for the design in addition to requirements as low-level design defines the unit.

- Individual components and units are tested to ensure that they work correctly as an individual as defined in design
- Unit testing requires throwaway drivers and stubs as individual files may not be testable or executable without them
- Unit testing may be performed in debugger mode to find how the variable values are changed during the execution. But, it may not be termed 'black box testing' in such case as code is seen in debugging.
- Gray box testing is also considered as 'unit testing technique' sometimes as it examines the code in detail along with its functioning. Gray box testing may need some tools which can check the code and functionality at the same time.
- Unit test cases must be derived from use cases/design component used at lowest levels of designs.

9.6.1 DIFFERENCE BETWEEN DEBUGGING AND TESTING

Many developers consider unit testing and debugging as the same thing. In reality there is no connection between the two. Difference between them can be illustrated as shown in Table 9.2

Table 9.2 Difference between debugging and unit testing

Debugging	Unit testing
It involves code checking to locate the causes of defect	It checks the defect and not the causes of the defect
Code may be updated during debugging	It does not involve any correction of code
The test cases are not defined for debugging	Test cases are defined based on requirements and design
Generally covers positive cases to see whether unit works correctly or not	Covers positive as well as negative cases

9.7 MODULE TESTING

Many units come together and form a module. The module may work of its own or may need stubs/drivers for its execution, if the module cannot be compiled into an executable. If the module can work independently, it is tested by tester. If it needs stubs and drivers, developers must create the same and perform testing. Module testing mainly concentrates on the structure of the system.

- Module testing is done on related unit-tested components to find whether individually tested units can work together as a module or not.
- Module test cases must be traceable to requirements/design. Generally, module test cases are derived from low-level design.

9.8 INTEGRATION TESTING

Integration testing involves integration of units to make a module/integration of modules to make a system/ integration of system with environmental variables if required to create a real-life application.

Integration testing may start at module level, where different units and components come together to form a module, and go upto system level. If module is self executable, it may be taken for testing by testers. If it needs stubs and drivers, it is tested by developers.

Though integration testing also tests the functionality of software under review, the main stress of integration testing is on the interfaces between different modules/systems. Integration testing mainly focuses on input/ output protocols, and parameters passing between different units, modules and/or system. Focus of integration is mainly on low-level design, architecture, and construction of software. Integration testing is considered as 'structural testing'. Figure 9.1 shows a system schematically

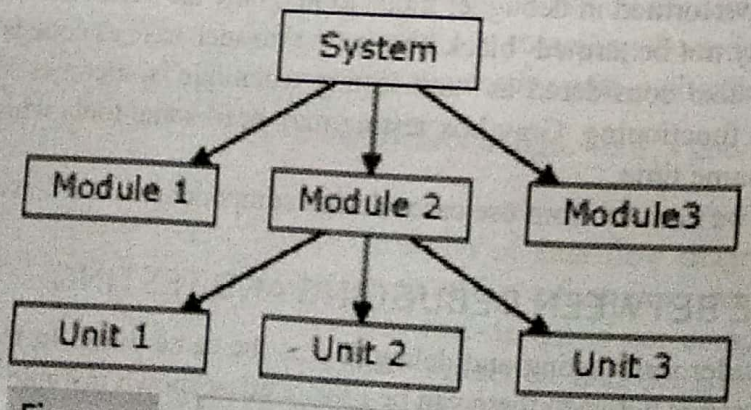

Fig. 9.1 Integration testing view

There are various approaches of integration testing depending upon how the system is integrated. Different approaches have different benefits and limitations.

9.8.1 BOTTOM-UP TESTING

Bottom-up testing approach focuses on testing the bottom part/individual units and modules, and then goes upward by integrating tested and working units and modules for system testing and intersystem testing. The process goes as mentioned below. Figure 9.2 shows a bottom-up integration and testing.

- Units at the lowest level are tested using stubs/drivers. Stubs and drivers are designed for a special purpose. They must be tested before using them for unit testing.
- Once the units are tested and found to be working, they are combined to form modules. Modules may need only drivers, as the low-level units which are already tested and found to be working may act as stubs.
- If required, drivers and stubs are designed for testing as one goes upward. Developers may write the stubs and drivers as the input/output parameters must be known while designing.
- Bottom-up approach is also termed 'classical approach' as it may indicate a normal way of doing things. Theoretically, it is an excellent approach but practically, it is very difficult to implement.
- Each component which is lowest in the hierarchy is tested first and then next level is taken. This gives very robust and defect-free software. But, it is a time consuming approach.

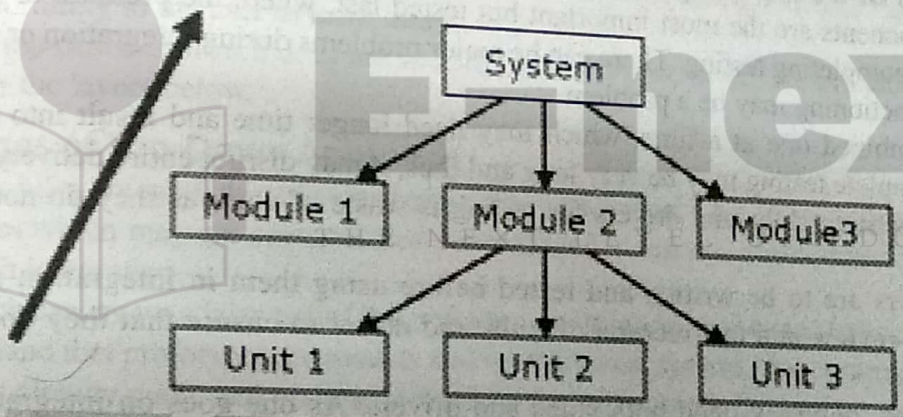

Fig. 9.2 **Bottom-up integration approach**

Bottom-up approach is suitable for the foilowing.

- Object-oriented design where objects are designed and tested individually before using them in a system. When the system calls the same object, we know that they are working correctly (as they are already tested and found to be working in unit testing).
- Low-level components are general-purpose utility routines which are used across the application; bottom-up testing is the recommended approach. This approach is used extensively in defining libraries.

Stubs/Drivers

- Stubs/drivers are special-purpose arrangements, generally code, required to test the units individually which can act as an input to the unit/module and can take output from the unit/module

Stub Stub is a piece of code emulating a called function. In absence of a called function, stub may take care of that part for testing purpose.

Driver Driver is a piece of code emulating a calling function. In absence of actual function calling the piece of code under testing, driver tries to work as a calling function.

- Stubs are mainly created for integration testing like top-down approach. Drivers are mainly created for integration testing like bottom-up approach.
- Stubs/drivers must be very simple to develop and use. They must not introduce any defect in the application. Most of the times, they are software programs, but it is not a rule. Stubs/drivers must be taken away before delivering code to customer/user.
- Reusability of stub/driver can improve productivity, while designing and coding multipurpose stub/driver can be a big challenge.
- Estimation must consider a time required for creating stubs/drivers.

Advantages of Bottom-Up Approach

- Each component and unit is tested first for its correctness. If it found to be working correctly, then only it goes for further integration.
- It makes a system more robust since individual units are tested and confirmed as working
- Incremental integration testing is useful where individual components can be tested in integration

Disadvantages of Bottom-Up Approach

- Top-level components are the most important but tested last, where the pressure of delivery may cause problem of not completing testing. There can be major problems during integration or interface testing, or system-level functioning may be a problem.
- Objects are combined one at a time, which may need longer time and result into slow testing. Time required for complete testing may be very long and thus, it may disrupt entire delivery schedule.
- Designing and writing stubs and drivers for testing is waste of work as they do not form part of final system.
- Stubs and drivers are to be written and tested before using them in integration testing. One needs to maintain the review and test records of stubs and driver to ensure that they do not introduce any defect.
- For initial phases, one may need both stubs and drivers. As one goes on integrating units, original stubs may be used while large number of new drivers may be required (which are to be thrown at the end).

9.8.2 TOP-DOWN TESTING

In top-down testing approach, the top level of the application is tested first and then it goes downward till it reaches the final component of the system. All top-level components called by tested components are combined one by one and tested in the process. Here, integration goes downward. Top-down approach needs design and implementation of stubs. Drivers may not be required as we go downward as earlier phase will act as driver for latter phase while one may have to design stubs to take care of lower-level components which are not available at that time.

Top-level components are the user interfaces which are created first to elicit user requirements or creation of prototype. Agile approaches like prototyping, formal proof of concept, and test-driven development use this approach for testing. Figure 9.3 shows a top down integration and testing.

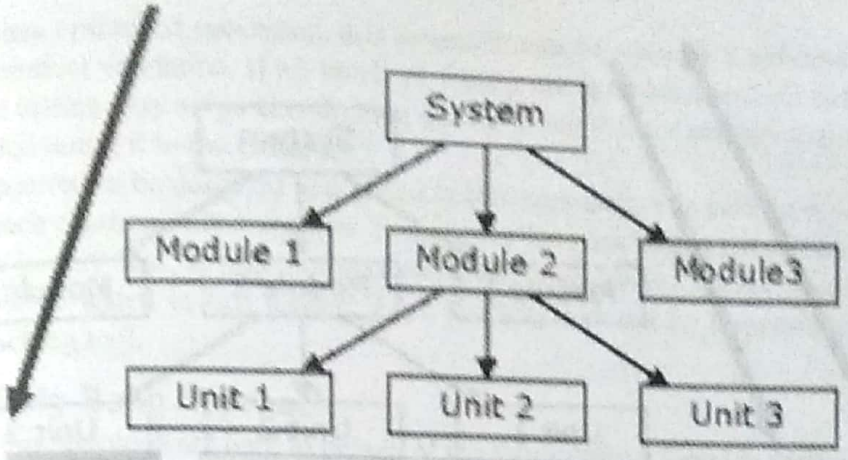

Fig. 9.3 Top-down integration approach

Advantages of Top-Down Approach

- Feasibility of an entire program can be determined easily at a very early stage as the topmost layer, generally user interface, is made first. This approach is good if the application has user interface as a major part.
- Top-down approach can detect major flaws in system designing by taking inputs from the user. Prototyping is used extensively in agile application development where user requirements can be clarified by preparing a model. If software development is considered as an activity associated with user learning, then prototyping gives an opportunity to the user to learn things.
- Many times top-down approach does not need drivers as the top layers are available first which can work as drivers for the layers below.

Disadvantages of Top-Down Approach

- Units and modules are rarely tested alone before their integration. There may be few problems in individual units/modules which may get compensated/camouflaged in testing. The compensating defects cannot be found in such integration.
- This approach can create a false belief that software can be coded and tested before design is finished. One must understand that prototypes are models and not the actual system. The customer may get a feeling that the system is already ready and no time is required for delivering it for usage.
- Stubs are to be written and tested before they can be used in integration testing. Stubs must be as simple as possible, and must not introduce any defect in the software. Large number of stubs may be required which are to be thrown at the end.

9.8.3 MODIFIED TOP-DOWN APPROACH

Modified top-down approach tries to combine the better parts of both approaches, viz. top-down approach and bottom-up approach. It gives advantages of top-down approach and bottom-up approach to some extent at a time, and tries to remove disadvantages of both approaches. Development must follow a similar approach to incorporate modified top–down approach. The main challenge is to decide on individual module/unit testing for bottom-up testing as normal testing may start from top.

Tracking and effecting changes is most important as development and testing start at two extreme ends at a time. Any change found at one place may affect another end. Care must be taken that the two approaches must meet somewhere in between. Configuration management is very important for such an approach. Figure 9.4 shows a modified top-down integration and testing.

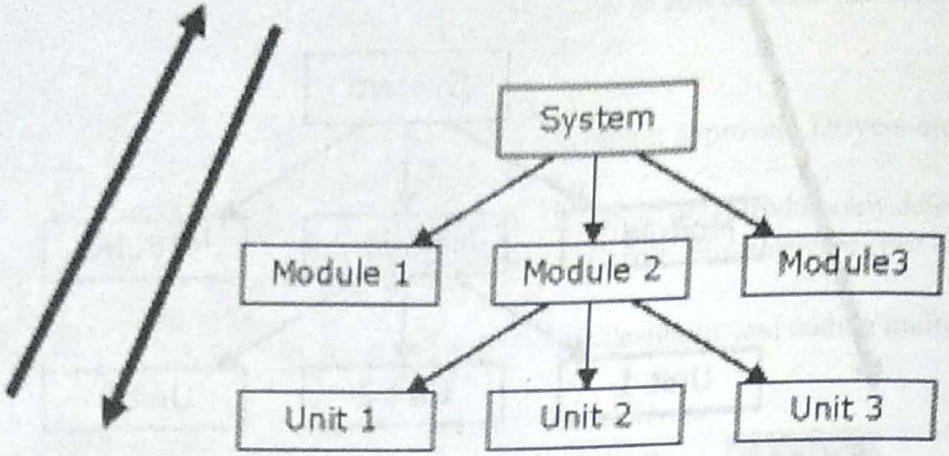

Fig. 9.4 Modified Top-down integration approach

Advantages of Modified Top-Down Approach

- Important units are tested individually, then combined to form the modules, and finally, the modules are tested before system is made. This is done during unit testing followed by integration testing.
- The systems tested by modified approach are better in terms of residual defects as bottom-up approach is used for critical components, and also better for customer-requirement elicitation as top-down approach vis used in general.
- It also saves time as all components are not tested individually. It is expected that all components are not critical for a system.

Disadvantages of Modified Top-Down Approach

- Stubs and drivers are required for testing individual units before they are integrated (atleast for critical units). Stubs are required for all parts as integration happens from top to bottom.
- Definition of critical units is very important. Criticality of the unit must be defined in design. Critical unit must be tested individually before any integration is done.

This approach actually means testing the system twice or atleast more than once. The first part of testing starts from top to bottom as we are integrating units downward, and the second part from bottom to top for selected components which are declared as 'critical units'. This may need more resources in terms of people, and more time for test cycles than top-down approach but less time for test cycles than bottom-up approach. This approach is more practical from usage point of view as all components may not be equally important.

9.9 BIG-BANG TESTING

Big-bang approach is the most commonly seen approach at many places, where the system is tested completely after development is over. There is no testing of individual units/modules and integration sequence. System testing tries to compensate for any other kind of testing, reviews, etc. Sometimes, it includes huge amount of random testing which may not be repeatable.

Advantages of Big-Bang Approach

- It gives a feeling that cost can be saved by limiting testing to last phase of development. Testing is done as a last-phase of the development lifecycle in form of system testing. Time for writing test cases and defining test data at unit level, integration level, etc may be saved.

- If an organisation has optimised processes, this approach can be used as a validation of development process and not a product validation. If all levels of testing are completed and all defects are found and closed, then system testing may act as certification testing to see if there are any major issues still left in the system before delivering it to the customer.
- No stub/driver is required to be designed and coded in this approach. The cost involved is very less as it does not involve much creation of test artifacts. Sometimes test plan is also not created.
- Big-bang approach is very fast. It may not give adequate confidence to users as all permutations and combinations cannot be tested in this testing. Even test log may not be maintained. Only defects are logged in defect-tracking tool.

Disadvantages of Big-Bang Approach

- Problems found in this approach are hard to debug. Many times defects found in random testing cannot be reproduced as one may not remember the steps followed in testing at that particular instance.
- It is difficult to say that system interfaces are working correctly and will work in all cases. Big-bang approach executes the test cases without any understanding of how the system is build.
- Location of defects may not be found easily. In big bang testing, even if we can reproduce the defects, it can be very difficult to locate the problematic areas for correcting them.
- Interface faults may not be distinguishable from other defects.
- Testers conduct testing based on few test cases by heuristic approach and certify whether the system works/does not work.

9.10 SANDWICH TESTING

Sandwich testing defines testing into two parts, and follows both parts starting from both ends i.e., top-down approach and bottom-up approach either simultaneously or one after another. It combines the advantages of bottom-up testing and top-down testing at a time. Figure 9.5 shows a sandwich testing approach.

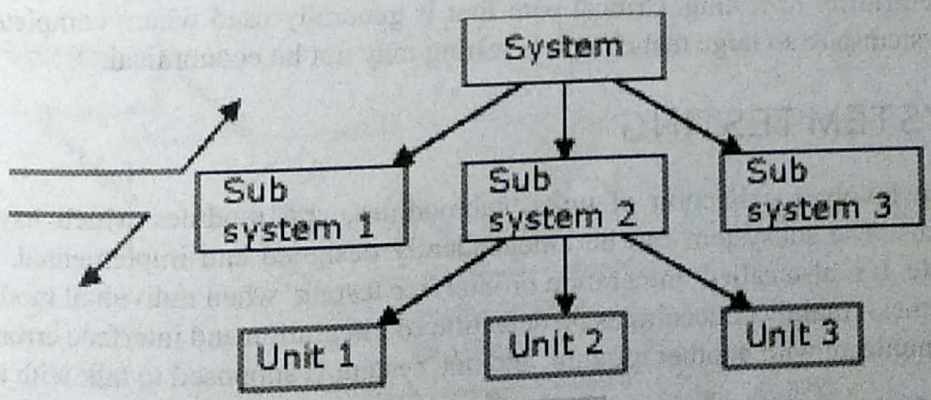

Fig. 9.5 Sandwich testing approach

9.10.1 PROCESS OF SANDWICH TESTING

- Bottom-up testing starts from middle layer and goes upward to the top layer. Generally for very big systems, bottom-up approach starts at subsystem level and goes upwards.
- Top-down testing starts from middle layer and goes downward. Generally for very big systems, top-down approach starts at subsystem level and goes downwards.

- Big-bang approach is followed for the middle layer. From this layer, bottom-up approach goes upwards and top-down approach goes downwards.

9.10.2 ADVANTAGES OF SANDWICH TESTING

- Sandwich approach is useful for very large projects having several subprojects. When development follows a spiral model and the module itself is as large as a system, then one can use sandwich testing.
- Both top-down and bottom-up approaches start at a time as per development schedule. Units are tested and brought together to make a system. Integration is done downwards.
- It needs more resources and big teams for performing both methods of testing at a time or one after the other.

9.10.3 DISADVANTAGES OF SANDWICH TESTING

- It represents very high cost of testing as lot of testing is done. One part has top-down approach while another part has bottom-up approach.
- It cannot be used for smaller systems with huge interdependence between different modules. It makes sense when the individual subsystem is as good as complete system.
- Different skill sets are required for testers at different levels as modules are separate systems handling separate domains like ERP products with modules representing different functional areas.

9.11 CRITICAL PATH FIRST

In critical path first, one must define a critical path which represents the main function of a system, generally represented by P1 requirements or P1 functionalities of an application. Testing defines the critical part of the system which must be covered first. Development team concentrates on design, implementation and testing of critical path of a system first. This is also termed 'skeleton development and testing'. It is applied where critical path of a system is important for system performance and for business from customer's perspective. One must understand the criticality of system functions from user's perspective or from business perspective, and decide on the priority of testing. Critical path first is generally used where complete system testing is impossible and systems are so large that complete testing may not be economical.

9.12 SUBSYSTEM TESTING

Subsystem testing involves collection of units, submodules, and modules which have been integrated to form subsystems. The subsystem can be independently designed and implemented, and also can be a separate executable. It is also called 'integration or interface testing' when individual modules are as good as independent systems. It mainly concentrates on detection of integration and interface errors where one unit is supposed to communicate with another module, and one system is supposed to talk with another system.

9.13 SYSTEM TESTING

System testing represents the final testing done on a system before it is delivered to the customer. It is done on integrated subsystems that make up the entire system, or the final system getting delivered to the customer. System testing validates that the entire system meets its functional/nonfunctional requirements as defined by the customer in software requirement specification. The criteria for system testing may involve an entire domain or selected parts depending upon the scope of testing. Generally system testing goes through the following stages.

- *Functional Testing* Functional testing intends to find whether all the functions as per requirement definition are working or not. Generally, any software is intended for doing some functions which must be

defined in requirement statement. Once functions are tested, all defects related to functions must be fixed to make the system correct.

- *User Interface Testing* Once the functionalities are set correct, the next step is to set the user interface correct (if the system has a user interface). User interface testing may involve colors, navigations, spellings, and fonts. Sometimes, there can be a thin line between functional testing and user interface testing, and one may take a decision depending upon the situation. There are few systems where there is no or very minimal user interface. In such cases, user interface testing may not be applicable.

Once the system is tested for functionalities and user interface, it is ready to go for other types of testing such as security, load, and compatibility. It depends upon the scope of testing, type of system under testing and which types of testing are involved as a part of system testing. If the scope does not include functionality testing, it is not required and one may directly go to user interface testing.

9.14 TESTING STAGES

Generally an organisation performs unit testing as first part of testing. The inputs from unit testing are used to identify and eliminate defects at unit level so that they will not go further down. Once unit testing is over and units are ready for integration, then integration or module testing is done. Module testing may be done by development/test teams depending upon the requirements of stubs and drivers. After module testing, the system goes for subsystem testing, which is also considered as 'integration testing'. Once the system completes all levels of integration, either it goes for system testing, or interface testing and then system testing as the case may be. Interface testing addresses the issues associated with communication of a system with another system that already exists or is expected to exist in future.

Once the system is through with system testing, it is taken for acceptance testing to check whether it fulfils the acceptance criteria or not. The stages of testing are graphically shown in Fig. 9.6.

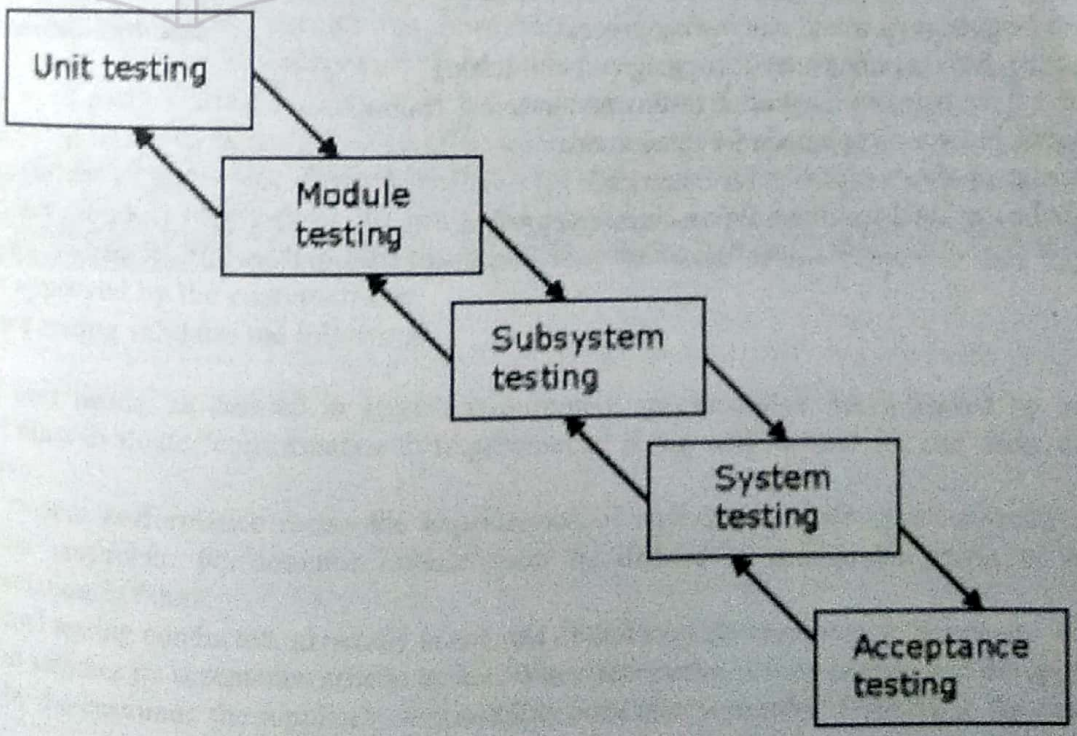

Fig. 9.6 Software testing stages

Tips for life-cycle testing

- Identify various stages of software development life cycle, and verification and validation activities related to these stages. The ultimate aim of all activities is to reduce the defects going to the customer.
- Identify the integration approach designed for the application under testing.

Summary

This chapter details verification and validation activities related to various stages of software development phases. It gives advantages and disadvantages of the various approaches of integration testing.

1) Describe proposal review process.
2) Describe requirement verification and validation process.
3) Describe design verification and validation process.
4) Describe code review and unit testing process.
5) Describe difference between debugging and unit testing.
6) Differentiate between integration testing and interface testing.
7) Describe bottom-up approach for integration.
8) Describe top-down approach for integration.
9) Describe modified top-down approach for integration.