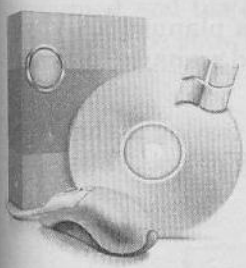

CHAPTER 5

INTEGRATION TESTING

5.1. INTEGRATION TESTING

Integration testing involves the 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. The primary objective of integration testing is to test the module interface, *i.e.,* there are no errors in the parameter passing, when one module involves another module. In the module interfaces 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-testing. If it needs stubs and drivers, it is tested by developers.

Integration testing also tests the functionalty 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".

The integration testing phase also involves developing and executing test cases that cover multiple components and functionality. When the functionality of different components are combined and tested together for a sequence

Chapter Outline

- Integration Testing
- Top-Down Integration Testing
- Bottom-Up Integration Testing
- Bi-Directional Integration Testing
- System Integration Testing
- Scenario Integration Testing
- Defect Bash
- Big-Bang Testing

of related operations, they are called "scenarios". Scenario testing is a planned activity to explore different usage patterns and combine them into test cases called scenario test cases. We will see scenario testing in more detail in the earlier sections.

The integration testing view can be shown in Fig. 5.1. Integration testing is the second level of testing. Its goal is to see if the modules are integrated properly or not. This testing activity can be considered testing the design.

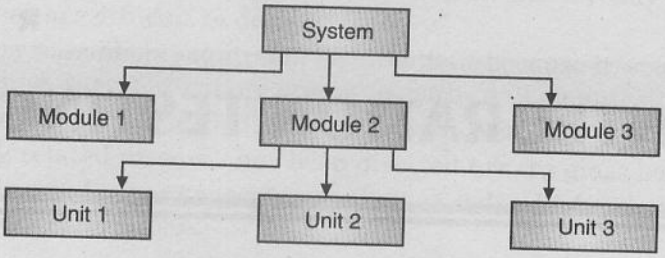

Fig. 5.1. Integration Testing View.

During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realize the full system. After each integration step, the partially integrated system is tested. An important factor that guides the integration plan is the module dependency graph. The structure chart (or module dependency graph) denotes the order in which different modules call each other. By examining the structure chart the integration plan can be developed. There are four types of integration testing approaches depending upon how the system is integrated. Any one (or a mixture) of the following approaches can be used to develop the integration test plan. Different approaches have different benefits and limitations. These approaches are:

- Bottom-up integration testing.
- Top-down integration testing.
- Mixed integration testing.
- Big-bang integrating testing.

These all are discussed in next sections.

5.2. TOP-DOWN INTEGRATION TESTING

(GBTU Exam., 2010-11)

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. Also referred as incremental integration testing technique which begins by testing the top-level module and progressively adds in lower level module one-by-one. Lower level modules are normally simulated by stubs which mimic functionality of lower level modules. As you add lower level code, stubs will replace with actual components. Top-down approach needs design and implementation of stubs so 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. Approaches like prototyping, formal proof of concept and test driver development uses this approach for testing. Top-down integration can be

performed and tested in breadth first or depth first manner. A top-down integration and testing can be shown in Fig. 5.2.

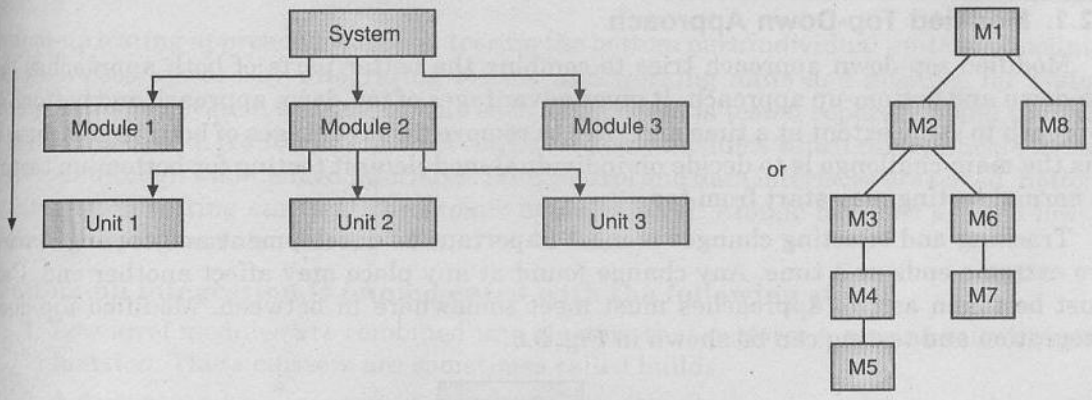

Fig. 5.2. Top-down Integration Approach.

In depth first all modules on a construct path are integrated first and in breadth first all modules directly subordinate at each level are integrated together.

Advantages of Top-Down Integration Testing

- Feasibility of an entire program can be determined easily at a very early stage as the top most 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 user. Prototyping is used extensively in agile application development where user requirement can be clarified by preparing a model. If software development is considered as an activity associated with user learning, then prototyping given 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.
- It provides early working module of program and so design defects can be found and corrected early.

Disadvantage 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 can't be found in such integration.
- In the absence of lower-level routines, many times it may become difficult to exercise the top level routines in the desired manner since the lower-level routines perform several low-level functions such as Input/Output functions.
- 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 if for usage.
- Stubs are to be written and tested before they can be used in integrated 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.

- It is difficult to have other people or third parties to perform this testing, mostly developers will have to spend time on this.

5.2.1. Modified Top-Down Approach

Modified top-down approach tries to combine the better parts of both approaches *viz.* top-down and bottom-up approach. It gives advantages of top-down approach and bottom-up approach to same extent at a time and tries to remove disadvantages of both approaches. In this 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 any place may affect another end. Care must be taken as two approaches must meet somewhere in between. Modified top-down integration and testing can be shown in Fig. 5.3.

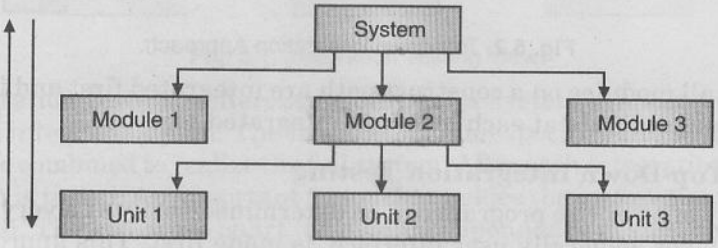

Fig. 5.3. Modified Top-Down Integration Approach

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 is used in general.
- It also saves times 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. Stubs are required for all parts as integration happens from top to bottom.
- Definition of critical units is very important. Critical unit must be tested individually before any integration is done.
- This approach actually means testing the system twice or at least 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.

5.3. BOTTOM-UP INTEGRATION TESTING (GBTU Exam., 2010-11)

Bottom-up testing approach focuses on testing the bottom part/individual units and modules and then goes upward by integrating testing and working units and modules for system testing and inter-system testing. In this each sub-system is tested separately and then the full system is tested. A sub-system must consist of many modules which communicate among each other through well-defined interfaces. Both control and data interfaces are tested. Bottom-up integration testing starts at the atomic modules level. Atomic modules are the lowest levels in the program structure.

A bottom-up integration is implemented with the following steps:

1. Low level modules are combined into clusters that perform a specific software sub-function. These clusters are sometimes called builds.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The build is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

Implementation is shown in Fig. 5.4.

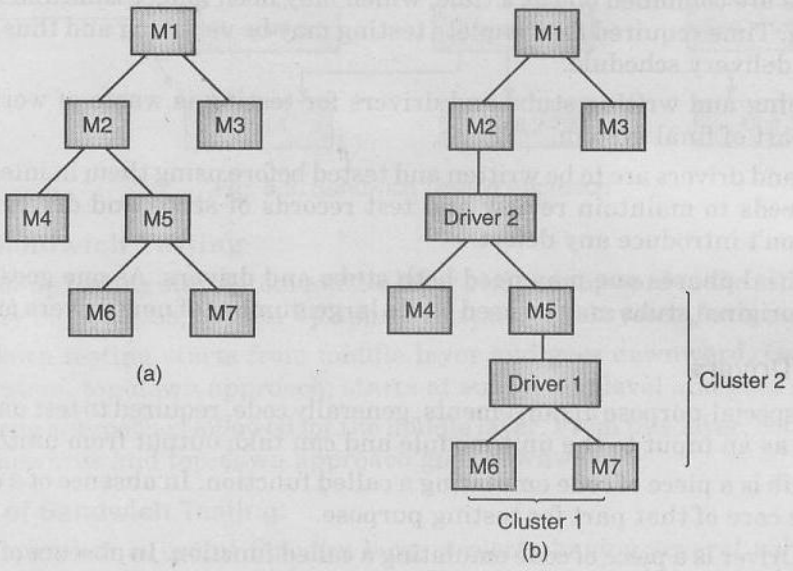

Fig. 5.4. (a) Program Module, (b) Bottom-up Integration applied to (a).

Figure 5.4 shows how the bottom-up integration is done. Whenever a new module is added to as a part of integration testing, the program structure changes. There may be new data flow paths, some new Input/Output or some new control logic. These changes may cause problems with functions in the tested modules, which were working fine previously. To detect these errors regression testing is done.

Bottom-up approach is suitable for the following

- 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 for them bottom-up testing is the recommended approach. This approach is used extensively in defining libraries.

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 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 don't form part of final system.
- Stubs and drivers are to be written and tested before using them in integration testing. One needs to maintain review and test records of stubs and driver to ensure that they don't 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.

5.3.1. Stubs/Drivers

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

Stubs: 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.

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

Difference Between Stubs and Drivers

- Stubs are mainly created for integration testing like top-down approach. Drivers are mainly created for integration testing like bottom-up approach.
- Both must be very simple to develop and use. They must not introduce any defect in the applicable. Most of times, they are software programs, but it is not a rule.

  Both taken away before delivering code to customer/user.
- Reusability of both can improve productivity, while designing and coding multi-purpose stub/driver can be a big challenge.

Library
United International University

5.4. BI-DIRECTIONAL INTEGRATION

Bi-directional integration also referred as sandwich testing or mixed Integration Testing. It is a kind of integration testing process that combine top-down and bottom-up integration testings. 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.

In top-down approach testing can start only after the top-level modules have been coded and unit tested. Similarly,, bottom up testing can start only after the bottom level modules are ready. Sandwich approach overcomes this shortcoming of top-down and bottom up approaches. In sandwich approach, testing can start as and when modules become available. Therefore, this is one of the most commonly used integration testing approach. Sandwich testing is also a vertical incremental testing strategy that tests the bottom layers and top layers and tests the integrated system in the computer software development process.

Using stubs, it test the user interface in isolation as well as tests the lower level functions using drivers. Fig. 5.5. shows a sandwich testing approach.

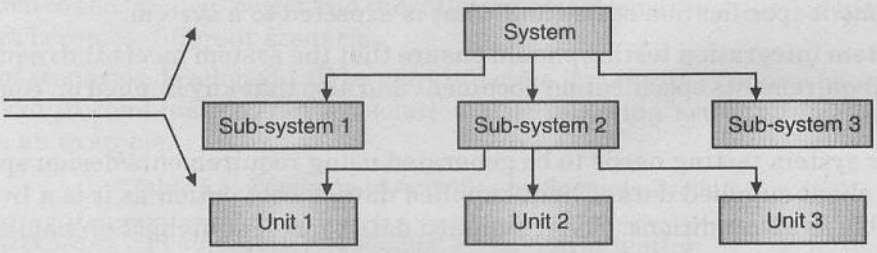

Fig. 5.5. Sandwich Testing Approach.

Process of Sandwich Testing

1. Bottom-up testing starts from middle layer and goes upward to the top layer. Generally for very big systems, bottom-up approach starts a subsystem level and goes upwards.
2. Top-down testing starts from middle layer and goes downward. Generally for very big system, top-down approach, starts at subsystem level and goes downwards.
3. Big-bang approach is followed for the middle layer. From this layer, bottom-up approach goes upwards and top-down approach goes downwards.

Advantages of Sandwich Testing:

1. This approach is useful for very large projects having several sub-projects. When development follows a spiral model and the module itself is as large as a system.
2. 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.
3. It needs more resources and big teams for performing both methods of testing at a time or one after the other.

Disadvantage of Sandwich Testing:

1. It represents very high cost of testing as lot of testing is done.
2. It cannot be used for smaller systems with huge interdependence between different modules. It makes sense when the individual sub-system is as good as complete system.

3. 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.

5.5. SYSTEM INTEGRATION (GBTU Exam., 2010-11)

System integration is the bringing together of the component sub-systems into one system and ensuring that the sub-systems function together as a system. In information technology, system integration is the process of linking together different computing systems and software applications physically and functionally. System integration also involves integrating existing sub-systems. The sub-systems will have interface. System integration is also about adding value to the system, capabilities that are possible because of interactions between sub-systems.

The testing should be driven by the requirement specification and the system integration engineer's knowledge of which the system should do.

No requirement specification can fully define a system, their will always be gap in what the requirement specification states and what is expected to a system.

The system integration testing should ensure that the system meets the requirements of the formal requirements specification document and also that any implied or 'common sense' requirements are met.

Data for system testing needs to be generated using requirements/design specifications or one may client supplied data. Client supplied data is best option as it is a live data may not be feasible in all conditions. Client supplied data represent highest probability data and must be taken for testing irrespective of any techniques applied.

During the system testing phase, non-functional testing also comes into picture and performance, load, stress, scalability all these types of testing are performed in this phase.

System testing is conducted on the complete integrated system and on a replicated production environment.

Given below are various system tests for computer based systems:

1. **Recovery Testing:** Many computer based systems need to recover from faults and resume processing within a particular time. In certain cases, a system needs to fault-tolerant. Recovery testing is a system test that faces the software to fail in various ways and verifies the recovery is performed correctly.
2. **Security Testing:** Security testing tries to verify that protection approaches built into a system will protect it from improper penetration.
3. **Stress Testing:** Stress testing executes a system in the demands resources in abnormal quantity, frequently or volume. A variation of stress testing is an approach called sensitivity testing.

5.6. SCENARIO TESTING

Scenario testing is defined as a "set of realistic user activities that are used for evaluating the product". It is also defined as the testing involving customer scenarios.

There are two methods to evolve scenarios:

1. System Scenarios
2. Use case scenarios/role based scenarios.

5.6.1. System Scenarios

System scenario is a method whereby the set of activities used for scenario testing covers several components in the system.

The following approaches can be used for develop system scenarios:

1. **Story Line:** Develop a story line that combines various activities of the product that may be executed by an end user. A use enters in his or her office, logs into the system, checks mail, responds to some mails performs unit testing and so on. All these typical activities carried out in the course of normal work when coined together became a scenario.
2. **Lifecycle/State Transition:** Consider an object, derive the different transitions that happen to the object and derive, scenarios to cover them. For example, in savings bank account, start with opening an account with a certain amount of money, make a deposit, perform a withdrawal, calculate interest and so on. All these activities are applied to the "money" object and the different transformations applied to the "money" object becomes different scenarios.

The set of scenarios developed will be more effective if majority of approaches mentioned above are used in combination, not in isolation. The following activity table explains the concept with an example.

Table 5.1: Coverage of Activities by Scenario Testing

End User Activity	Frequency	Priority	Application Environments	No. of Times Covered
1. Login to application	High	High	W 2000, W 2003, XP	10
2. Create on object	High	Medium	W 2000, XP	7
3. Modify Parameters	Medium	Medium	W 2000, XP,	5
4. List Object Parameters	Low	Medium	W 2000, Xp, W 2003	3
5. Compose E-Mail	Medium	Medium	W 2000, XP	6
6. Attach Files	Low	Low	W 2000, XP	2
7. Send Compose Mail	High	High	W 2000, Xp	10

From the table it is clear that important activities have been very well covered by set of scenarios in the system by set of scenarios in the system scenario test. This kind of table also help us to ensure that all activities are covered according to their frequency of usage in customer place and according to the relative priority assigned based on customer usage.

5.6.2. Use Case Scenarios

Use Case Scenarios is a stepwise procedure on how a user intends to use a system, with different user roles and associated parameters. A use case scenario can include stories, pictures and deployment details. Use cases are useful for explaining customer problems and how the software can solve those problems without any ambiguity.

A use case can involve several roles or class of users who typically perform different activities that are very specific and can be performed only by the use belonging to a particular role and there are some activities that are common across roles. Use case scenarios term the users with different roles as actors. What the product should do for a particular activity is termed as system behaviour. Users with a specific role to interact between the actors and system are called agents.

To explain the concept of use case scenarios, let us take an, *e.g.,* of withdrawing cash from a bank. A customer fills up a check and gives it to an official in bank. The official verified the balance in account from the computer and gives the required cash to the customer. The customer in this example is actor, the clerk the agent and the response given by the computer, which gives the balance in the account, is called the system response. Fig. 5.6. gives it in detail.

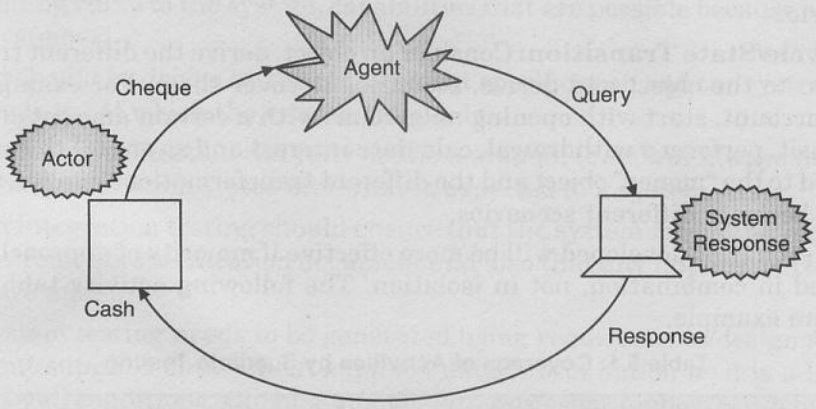

Fig. 5.6. Example of a use case scenario in a bank.

This way of describing different roles in test cases helps in testing the product without getting into the details of the product.

In above, *e.g.,* the actor (the customer) need not know what the official is doing and what command is using to interact with the computer. Actor is only concerned about getting the cash. The agent (who is official) is not concerned about the logic of how the computer works. He or she is only interested in knowing from the computer whether he or she can give cash or not. However, the system behaviour (computer logic) needs to be tested before applying the sequence of agent and actor activities.

Actor and agent are roles that represent different types (classes) of users. Simulating different types of users again needs a clear understanding of business and system response for each of the user needs a clear understanding of how the product is implemented.

The agent part of the use cases are not needed in all cases. In a completely automated system involving the customer and the system, use cases can be written without considering the agent portion.

Let us extend the earlier example of cash withdrawal using an ATM. The given table illustrate how the actor and system response can be described in the use case.

Table 5.2: Actor and System Response in use Case for ATM Cash Withdrawal

Actor	System Response
User likes to with draw cash and inserts the card in ATM machine.	Request for password or PIN.
User fills in the password or PIN.	Validate the password or PIN. Give a list containing types of accounts.
User selects an account type.	Ask the user for amount to withdraw.
User fills in the amount of cash required.	Check availability of funds Update account balance Prepare receipt Dispense cash.
Retrieve cash from ATM.	Print Receipt.

This way of documenting a scenario testing makes of simple and also makes it realistic for customer usage. Use cases are not used only for testing. In some product implementations, use cases are prepared prior to the design and coding phases and they are used as a set of requirement for design and coding phases. All development activities are performed based on use case documentation. In extreme programming models these are termed as user stories and form the basis for architecture/design and coding phases.

Use cases are useful in combining the business perspective and implementation detail and testing them together.

5.6.3. Characteristics of Scenario Testing

An Ideal Scenario test has several characteristics:

1. The test is based on a story about how the program is used, including information about the motivations of the people involved.
2. The story is motivating. A stakeholder with influence would push to fix a program that failed this test.
3. The story is credible. It not only could happen in real world; stakeholders would believe that something like of probably will happen.
4. The story involves a complex use of the program or a complex environment or a complex set of data.
5. The test results are easy to evaluate. This is valuable for all tests, but is especially important for scenarios because they are complex:
6. **Deployment/implementation stories from customer:** Develop a scenario from a known customer deployment/implementation details and create a set of activities by various users in that implementation.
7. **Business Verticals:** Visualize how a product/software will be applied to different verticals and create a set of activities as scenario to address specific vertical business. For example, take the purchasing function. It may be done differently in different verticals like pharmaceuticals, software houses and government organizations. Visualizing these different types of tests make the product "multi-purpose".

8. **Battle Gound:** Create some scenarios to justify that "the product works" and same scenarios to "try to break the system" to justify "the product doesn't work". This adds flavour to the scenarios mentioned above.

5.6.4. Twelve Ways to Create Good Scenarios

1. Write life histories for objects in the system.
2. List possible users, a analyze their interests and objectives.
3. Consider disfavoured users, how do they want to abuse your system?
4. List "system events". How does the system handle them?
5. List "special events". What accommodations does the system make for these?
6. List benefits and create end-to-end tasks to check them.
7. Interview users about famous challenges and failures of the old system.
8. Work alongside users to see how they work and what they do.
9. Read about what systems like this are supposed to do.
10. Study complaints about the predecessor to this system or its competitors.
11. Create a mock business. Treat it as real and process its data.
12. Try converting real-life data from a competing or predecessor application.

5.6.5. Risks of Scenario Testing

There are three serious problems with scenario tests:

1. Other approaches are better for testing early, unstable code. The scenario test is complex, involving many features. If the first feature is broken, the rest of the test can't be run. Once that feature is fixed, the next broken feature blocks the test. In some companies, complex test fail and fail all through the project, exposing one or two new bugs at a time. Discovery of some bugs has been delayed a long time until scenario-blocking bugs were cleared out of way.
2. Scenario tests are not designed for coverage of program. It takes exceptional care to cover all the features or requirements in a set of scenario tests. Covering all the program's statements simply isn't achieved this way.
3. Scenario tests are often heavily documented and used time and again. This seems efficient, given all the work it can take to create a good scenario. But scenario tests often expose design errors rather than coding errors. Scenarios are interesting tests for coding errors because they combine so many features and so much data.

5.7. DEFECT BASH

Also referred as Bug-Bash. Defect bash is an adhoc testing where people performing different roles in an organization test the product together at the same time. This is very popular among application development companies, where the product can be used by people who perform different roles. The testing by all the participants during defect bashing is not based on written test cases. What is to be tested is left to an individual's decision and creativity. A usual defect bash lasts half a day and is usually done when the software is close to being ready to release.

There are two types of defects that will emerge during a defect bash.

1. **Functional Defects:** Defects that are in the product, as reported by the users.
2. **Non-Functional Defects:** Defects that are unearthed while monitoring system resources, such as memory leak, long turn around time, high impact and utilization of system resources and so on.

Defect bash is a unique testing method which can bring out both functional and non-functional defects.

Defects bash brings together plenty of good practices that are popular in testing industry. They are as follows:

1. Enabling people "cross boundaries and test beyond assigned areas."
2. Bringing different people performing different roles together in the organization for testing – "Testing isn't for testers alone".
3. Letting everyone in the organisation use the product before delivery – "Eat your own dog food".
4. Bringing fresh pairs of eyes to uncover new defects – "Fresh eyes have less bias".
5. Bringing in people who have different levels of product understanding to test the product together randomly–"Users of software are not same".
6. Let testing doesn't wait for lack of / time taken for documentation–"Does testing wait till all documentation is done?".
7. Enabling people to say "system works" as well as enabling them to "break the system"– "Testing isn't to conclude the system works or doesn't work".

All the activities in the defect bash are planned activities, except for what to be tested. It involves several steps.

Step 1: Choosing the frequency and duration of defect bash.

Step 2: Selecting the right product build.

Step 3: Communicating the objective of each defect bash to everyone.

Step 4 : Setting up and monitoring the lab for defect bash.

Step 5: Taking actions and fixing issues.

Step 6: Optimizing the effort involved in defect bash.

A defect bash in an adhoc testing done by people performing different roles in the same time duration during the integration testing phase, to bring out all types of defects that may have been left out by planned testing.

5.8. 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

1. 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.

2. No. Stub/driver is required to be designed and coded in this approach. The cost involved is very less as it does not involved much creation of test artifacts. Sometimes test plan is also not created.
3. 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

1. Problems found in this approach are hard to debug. Many times defects found in random testing cannot be reproduced as one may not remember steps followed in testing.
2. It is difficult to say that system interfaces are working correctly and will work in all cases.
3. Location of defects may not be found easily. In this event if we can reproduce the defects, it can be very difficult to locate the problematic areas for correcting them.
4. Interface faults may not be distinguishable from other defects.
5. Testers conduct testing based on few test cases by heuristic approach and certify whether the system works/does not work.

Summary

- Integration testing involves the integration units to make a module to make a system with environmental variables if required to create a real life application.
- Approaches of Integration Testing are :

  Bottom-up Integration Testing

  Top-down Integration Testing

  Mixed or Sandwich Integration Testing

  Big-bang Integration Testing
- In top-down testing approach, the top level of application is tested first and then it goes downward till it reaches the final component of the system.
- In Bottom-up testing approach it focuses on testing the bottom part and modules and then goes upward by integrating testing and working units and modules for system testing and inter-system testing.
- Stubs is a piece of code emulating a called function.
- Drivers is also a piece of code emulating a called function.
- Bi-directional integration also referred as sandwidth or mixed integration testing is a kind of process that combine top-down and bottom-up integration testings.
- System integration is the process of linking together different computing systems and software application physically and functionally.
- Scenario testing is defined as a set of realistic user activities that are used for evaluating the product.

- There are two methods to evolve scenarios
  ⇒ System scenarios
  ⇒ Use Case/role based scenarios
- Defect bash also referred as Bug Bash in an adhoc testing where people performing different roles in an organization test the product together at the same time.
- Big Bang testing approach is used where the system is tested completely after development is over.

Review Questions

1. What is Integration Testing? Explain different types of approaches used in Integration Testing.
2. Define Stubs and drivers.
3. Write twelve ways to create Good scenarios.
4. Define defect bash and write its advantages and disadvantages?
5. Consider a typical university academic information system. Identify the typical agents, actors and the expected system behaviours for the various types of use cases in the system.
6. Give examples to show cases where integration testing can be taken as white box approach (*i.e.*, with access to code).

❐❐❐