Unit-III

LEVELS OF TESTING

The need for Levels of Testing – Unit Test – Unit Test Planning – Designing the Unit Tests – The Test Harness – Running the Unit tests and Recording results – Integration tests – Designing Integration Tests – Integration Test Planning – Scenario testing – Defect bash elimination System Testing – Acceptance testing – Performance testing – Regression Testing – Internationalization testing – Ad-hoc testing – Alpha, Beta Tests – Testing OO systems – Usability and Accessibility testing – Configuration testing – Compatibility testing – Testing the documentation – Website testing.

The need for Levels of Testing

- there will be 3-4 levels: unit test, integration test, system test, and some type of acceptance test
- At the system level - performance, usability, reliability, and other quality-related requirements.

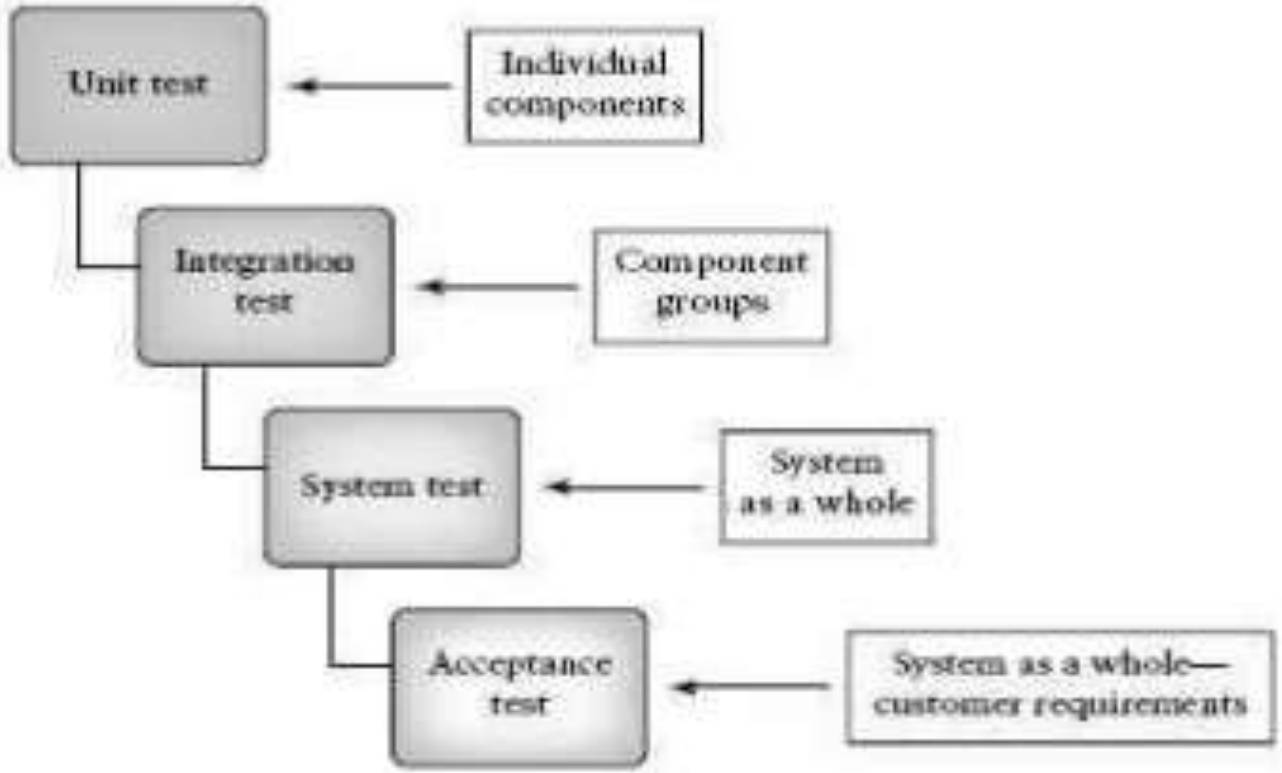


FIG. 6.1
Levels of testing.

- The approach used to design and develop impact on testers plan and design suitable tests.
- There are two major approaches to system development—bottom-up, and top-down

A unit is the smallest possible testable software component. It can be characterized in several ways.

For example, a unit in a typical procedure-oriented software system:

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

- A unit is traditionally viewed as a function or procedure implemented in a procedural (imperative) programming language.
- A unit may also be a small-sized COTS(Commercial off-the-shelf) component purchased from an outside vendor.

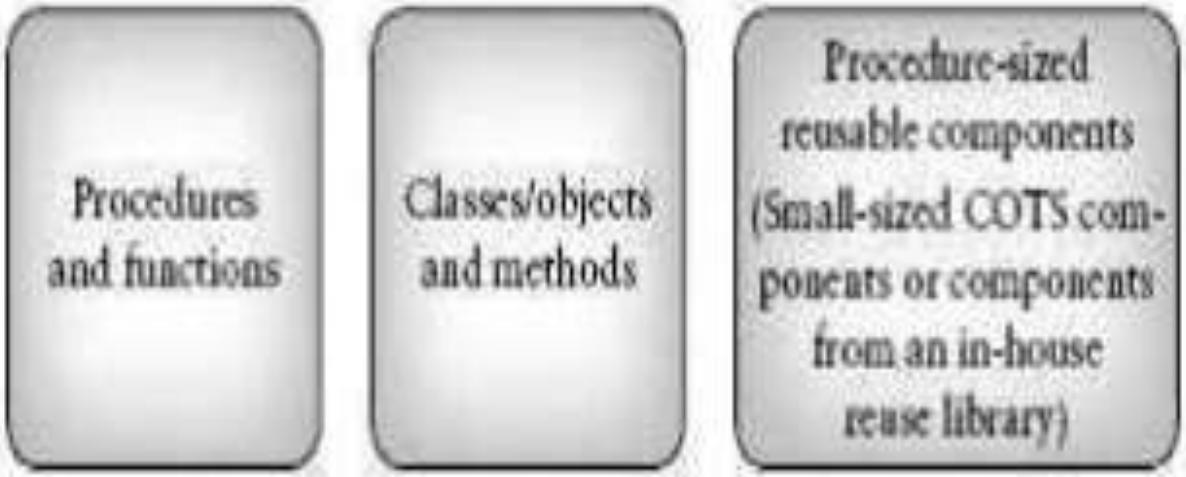


Fig. 6.2

Some components suitable for unit test.

Unit test planning

- A general unit test plan should be prepared & component of the master test plan.
- Documents that provide inputs for the unit test plan are the project plan, as well the requirements, specification, and design documents that describe the target units.
- Components of a unit test plan are described in detail the *IEEE Standard for Software Unit Testing* .

- Excellent guide for the test planner.
- In each phase, a set of activities based on IEEE unit test standard .

Phase 1: Describe Unit Test Approach and Risks

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

(i) identifies test risks;

(ii) describes techniques for designing the test cases for the units;

(iii)describes techniques for data validation and recording of test results;

(iv) describes the requirements for test harnesses and other software that interfaces with the units to be tested, for example, any special objects needed for testing object- oriented units.

- During this phase the planner also identifies completeness requirement what will be covered by the unit test and to what degree (states, functionality, control, and data flow patterns).
- The planner also identifies termination conditions for the unit tests.
- This includes coverage requirements, and special cases. Special cases may result in abnormal termination of unit test (e.g., a major design flaw).

- Strategies for handling these special cases need to be documented.
- Finally, the planner estimates resources needed for unit test, such as hardware, software, and staff, and develops a tentative schedule under the constraints identified at that time.

Phase 2: Identify Unit Features to be Tested

- This phase requires information from the unit specification and detailed design description.
- The planner determines which features of each unit will be tested, for example: functions, performance requirements, states, and state transitions, control structures, messages, and data flow patterns.
- If some features will not be covered by the tests, they should be mentioned and the risks of not testing them be assessed.
- Input/output characteristics associated with each unit should also be identified, such as variables with an allowed ranges of values and performance at a certain level.

- **Phase 3: Add Levels of Detail to the Plan**
- In this phase the planner refines the plan as produced in the previous two phases.
- The planner adds new details to the approach, resource, and scheduling portions of the unit test plan. As an example, existing test cases that can be reused for this project can be identified in this phase.
- Unit availability and integration scheduling information should be included in the revised version of the test plan.

- The planner must be sure to include a description of how test results will be recorded.
- Test-related documents that will be required for this task, for example, test logs, and test incident reports, should be described, and references to standards for these documents provided.
- Any special tools required for the tests are also described.
- The next steps in unit testing consist of designing the set of test cases, developing the auxiliary code needed for testing, executing the tests, and recording and analyzing the results.

Designing the unit tests

Part of the preparation work for unit test involves unit test design. It is important to specify

(i) the test cases (including input data, and expected outputs for each test case), and,

(ii) the test procedures (steps required run the tests).

- Berard has described a test case specification notation.
- He arranges the components of a test case into a semantic network with parts, Object_ID, Test_Case_ID, Purpose, and List_of_Test_Case_Steps.
- Each of these items has component parts. In the test design specification Berard also includes lists of relevant states, messages (calls to methods), exceptions, and interrupts.

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

- Test case design at the unit level - based on the black and white box test design strategies .
- Considering the relatively small size of a unit, it makes sense to focus on white box test design for procedures/functions and the methods in a class.
- This approach gives the tester the opportunity to exercise logic structures and/or data flow sequences, or to use mutation analysis

- Some black box-based testing is also done at unit level; however, the bulk of black box testing is usually done at the integration and system levels and beyond.

The test harness

A **test harness** or automated **test** framework is a collection of software and **test** data configured to **test** a program unit, by running it under varying conditions and monitoring its behavior and outputs.

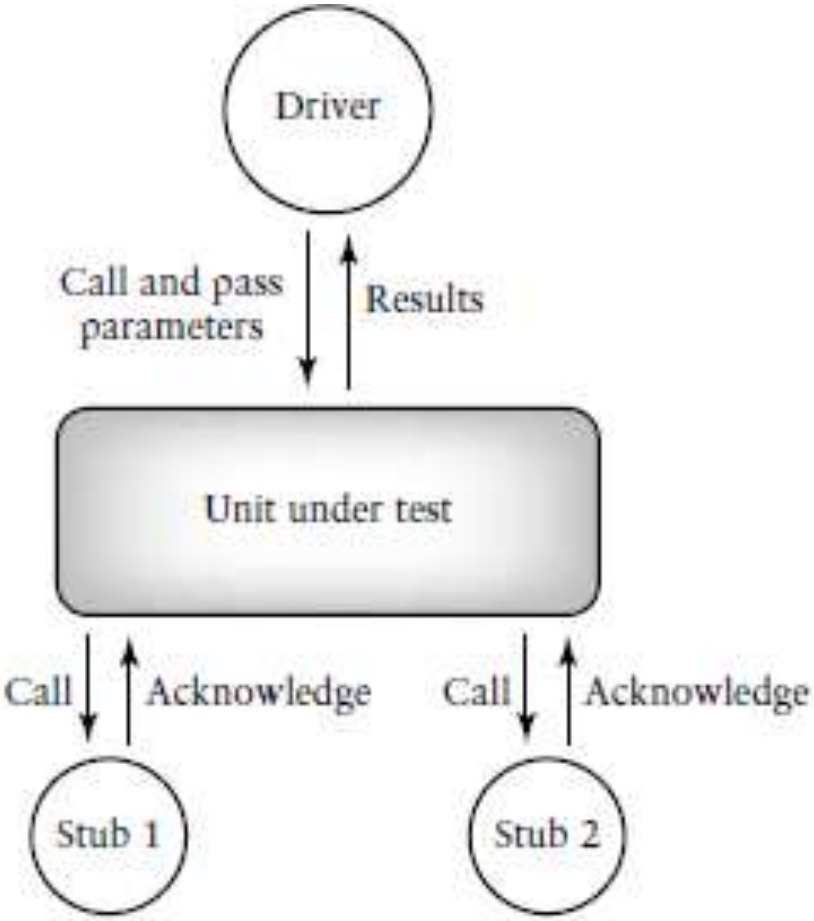
In addition to developing the test cases, supporting code must be developed to exercise each unit and to connect it to the outside world.

Since the tester is considering a stand-alone function/procedure/class, rather than a complete system, code will be needed to call the target unit, and also to represent modules that are called by the target unit.

This code called the test harness, is developed especially for test and is in addition to the code that composes the system under development.

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

- The drivers and stubs must be tested themselves to insure they are working properly and that they are reusable for subsequent releases of the software.


Driver
Call and pass parameters
Results
Unit under test
Call
Acknowledge
Call
Acknowledge
Stub 1
Stub 2

- For example, a driver could have the following options and combinations of options:

(i) call the target unit;

(ii) do 1, and pass inputs parameters from a table;

(iii)do 1, 2, and display parameters;

(iv)do 1, 2 and display results (output parameters).

- The test planner must realize that, the higher the degree of functionally for the harness, the more resources it will require to design, implement, and test.
- Developers/testers will have to decide

  depending on the nature of the code under test, just how complex the test harness needs to be.

Running the unit tests and recording results

- Unit tests can begin when

(i) the units becomes available from the developers (an estimation of availability is part of the test plan),

(ii) the test cases have been designed and reviewed, and

(iii) the test harness, and any other supplemental supporting tools, are available.

- The status of the test efforts for a unit, and a summary of the test results, could be recorded in a simple format
- These forms can be included in the test summary report, and are of value at the weekly status meetings that are often used to monitor test progress.
- It is very important for the tester at any level of testing to carefully record, review, and check test results. The tester must determine from the results whether the unit has passed or failed the test.

- If the test is failed, the nature of the problem should be recorded in what is sometimes called a test incident report
- Differences from expected behavior should be described in detail.
- This gives clues to the developers to help them locate any faults. During testing the tester may determine that additional tests are required.

- When a unit fails a test there may be several reasons for the failure. The most likely reason for the failure is a **fault in the unit implementation (the code).** Other likely causes :

- a fault in the test case specification (the input or the output was not specified correctly);

- a fault in test procedure execution (the test should be rerun);

- a fault in the test environment (perhaps a database was not set up properly);

- a fault in the unit design (the code correctly adheres to the design specification, but the latter is incorrect).

Summary sheet

Unit Test Worksheet

Unit Name: _____

Unit Identifier: _____

Tester: _____

Date: _____

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

- The causes of the failure should be recorded in a test summary report.
- when a unit has been completely tested and finally passes all of the required tests it is ready for integration.
- Under some circumstances unit may be given a conditional acceptance for integration test.

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

Integration tests

Integration test for procedural code has two major goals:

(i) to detect defects that occur on the interfaces of units;

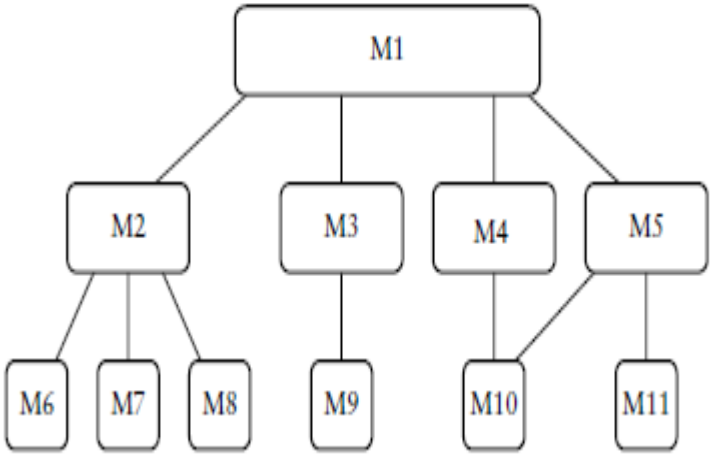
(ii) to assemble the individual units into working subsystems and finally a complete system that is ready for system test.

- Integration test should only be performed on units that have been reviewed and have successfully passed unit testing.
- A tester might believe erroneously that since a unit has already been tested during a unit test with a driver and stubs, it does not need to be retested in combination with other units during integration test.
- However, a unit tested in isolation may not have been tested adequately for the situation where it is combined with other modules.

- One unit at a time is integrated into a set of previously integrated modules
- The interfaces and functionality of the new unit in combination with the previously integrated units is tested.
- When a subsystem is built from units integrated in this stepwise manner, then performance, security, and stress tests can be performed on this subsystem.

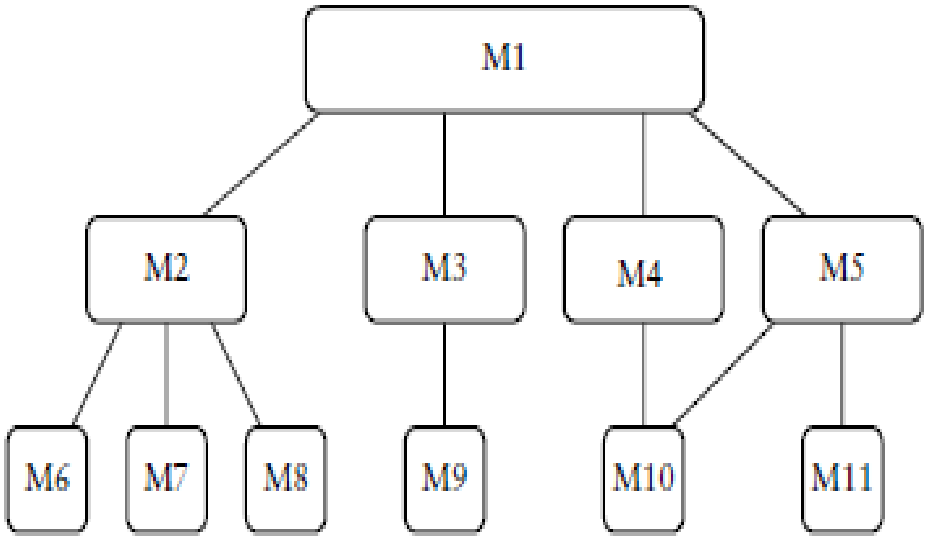
Integration Strategies for Procedures and Functions

- top-down and bottom-up

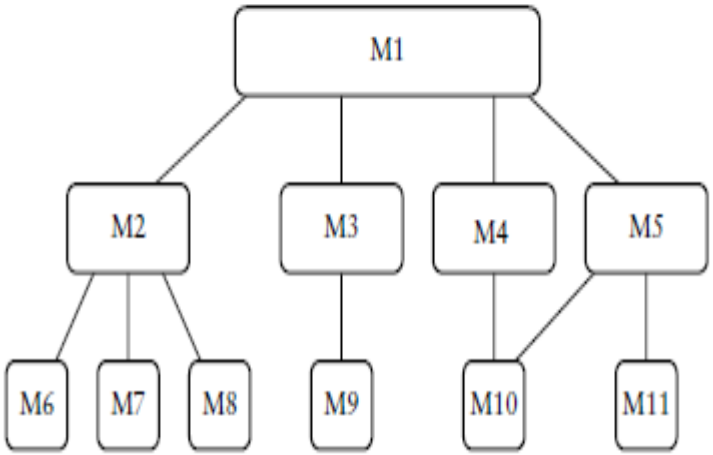


- These charts show hierarchical calling relationships between modules.
- Each node, or rectangle in a structure chart, represents a module or unit
- The edges or lines between them represent calls between the units.

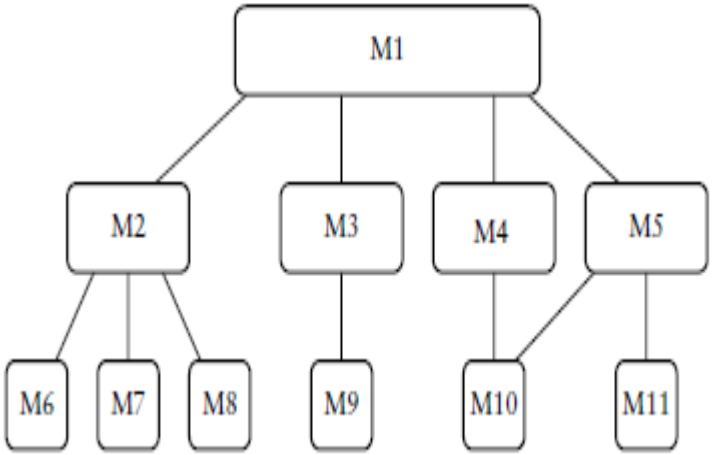
- The upper level module calls the lower module.
- Bottom-up integration of the modules begins with testing the lowest level modules M6, M7, M8, M9, M10, M11.
- These are modules that do not call other modules.



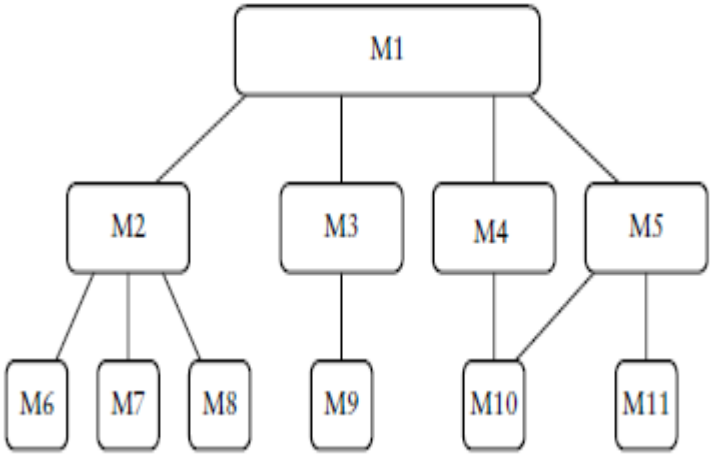
- Drivers are needed to test these modules.
- The next step is to integrate modules on the next upper level of the structure chart whose subordinate modules have already been tested.
- For example, if we have tested M6, M7, and M8, then we can select M2 and integrate it with M6, M7, and M8.



- Accordingly we can integrate M9 with M3 when M9 is tested, and M4 with M10 when M10 is tested, and finally M5 with M11 and M10 when they are both tested.
- The M4 and M5 subsystems have overlapping dependencies on M10.



- To complete the subsystem represented by M5, both M10 and M11 will have to be tested and integrated.
- M4 is only dependent on M10.
- In that way the system is finally integrated as a whole.

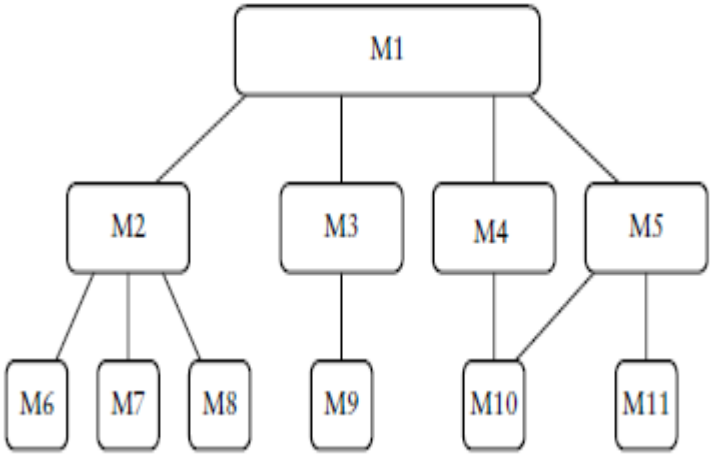


- A rule of thumb for bottom-up integration says that all of a module's subordinate modules must have been tested previously.

Top-down integration

- The rule of thumb for selecting candidates for the integration sequence says that when choosing a candidate module to be integrated next, at least one of the module's superordinate (calling) modules must have been previously tested.

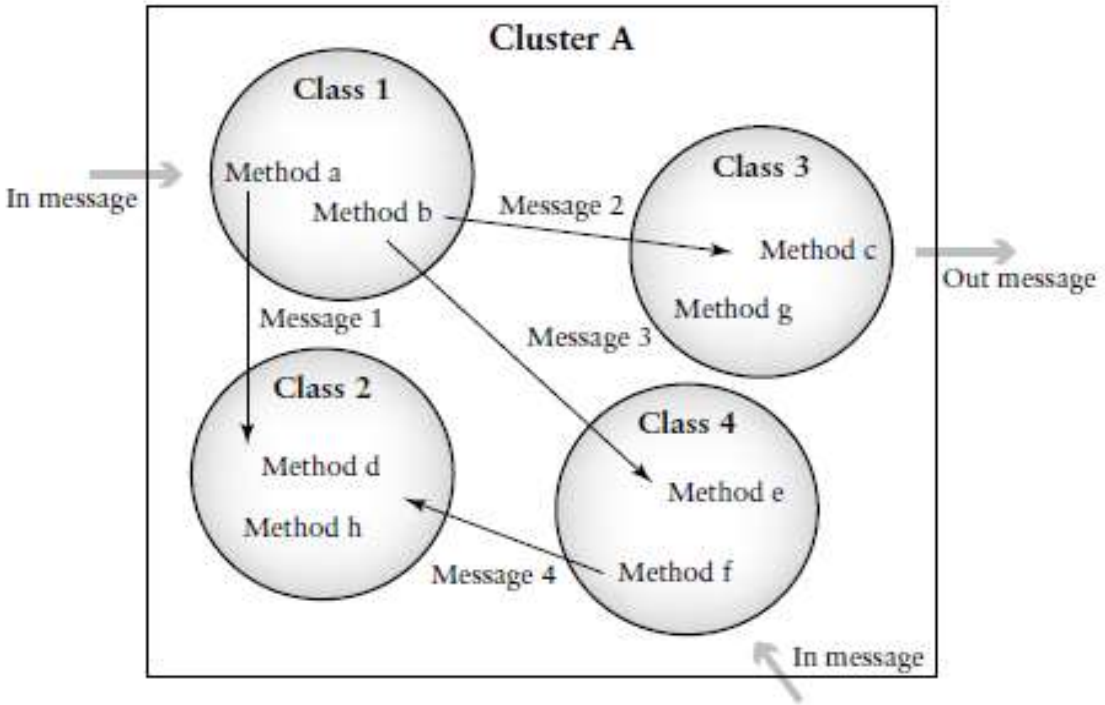
- we begin top-down integration with module M1. We create four stubs to represent M2, M3, M4, and M5.
- M2-M1,M2-M6,M7,M8
- Same as M3-M9,M4-M10,M5-M11



Integration Strategies for Classes

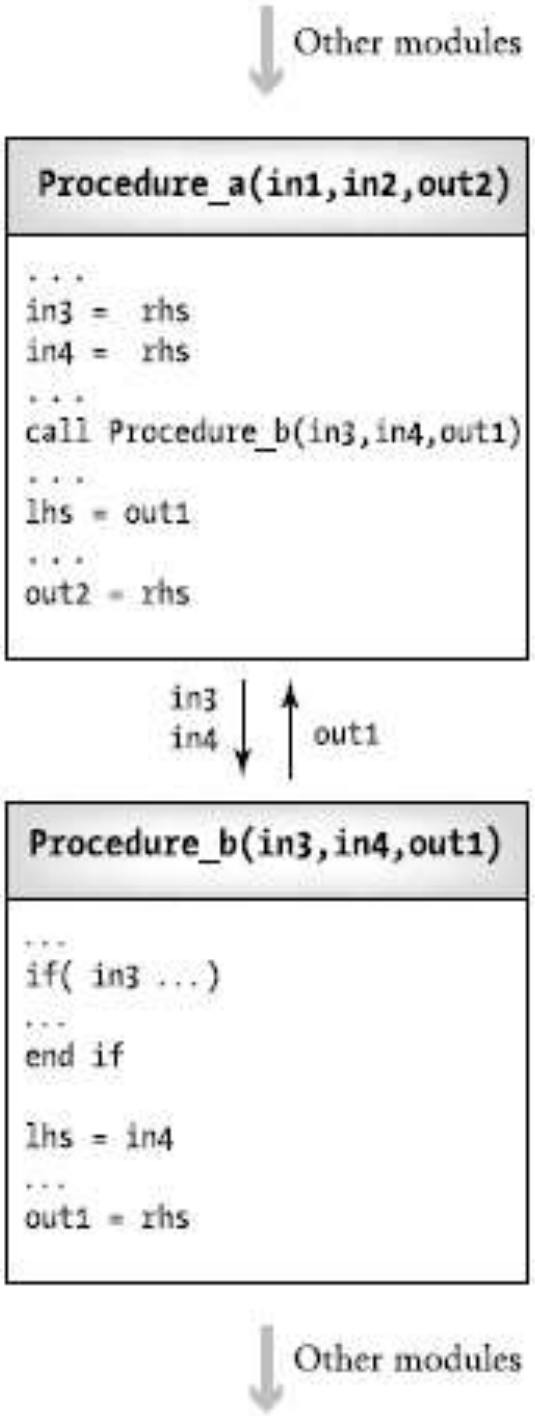
- A good approach to integration of an object-oriented system is to make use of the concept of object clusters.
- **A cluster consists of classes that are related, for example, they may work together (cooperate) to support a required functionality for the complete system.**

- To integrate an object-oriented system using the cluster approach a tester could select clusters of classes that work together to support simple functions as the first to be integrated.



Designing integration tests

- Integration tests for procedural software can be designed using a black or white box approach. Both are recommended.
- Since many errors occur at module interfaces, test designers need to focus on exercising all input/output parameter pairs, and all calling relationships.
- The tester must also insure that once the parameters are passed to a routine they are used correctly.



- Procedure_b is being integrated with Procedure_a.
- Procedure_a calls Procedure_b with two input parameters in3, in4. Procedure_b uses those parameters and then returns a value for the output parameter out1.
- Terms such as *lhs* and *rhs* could be any variable or expression.

- The actual usage of the parameters must be checked at integration time.
- Data flow–based (def-use paths) and control flow (branch coverage) test data generation methods are useful here to insure that the input parameters, in3, in4, are used properly in Procedure_b.
- Data flow methods (def-use pairs) could also be used to check that the proper sequence of data flow operations is being carried out to generate the correct value for out1 that flows back to Procedure_a.

- Black box tests are useful in this example for checking the behavior of procedures.
- For this example test input values for the input parameters in1 and in2 should be provided, and the outcome in out2 should be examined for correctness.

Integration Test Planning

- Integration test must be planned.
- Planning can begin when high-level design is complete so that the system architecture is defined.
- Other documents relevant to integration test planning are the requirements document, the user manual, and usage scenarios.

- These documents contain structure charts, state charts, data dictionaries, cross-reference tables, module interface descriptions, data flow descriptions, messages and event descriptions, all necessary to plan integration tests.
- The testing objectives are to assemble components into subsystems and to demonstrate that the subsystem functions properly

- Testing resources and schedules for integration should be included in the test plan
- The plan includes the following items:

(i) clusters this cluster is dependent on;

(ii) a natural language description of the functionality of the cluster to be tested;

(iii) list of classes in the cluster;

(iv) a set of cluster test cases.

Scenario testing

- **Scenario testing** is a software testing activity that uses scenarios: hypothetical stories to help the tester work through a complex problem or test system.
- The ideal scenario test is a credible, complex, compelling or motivating story ,the outcome of which is easy to evaluate.
- These tests are usually different from test cases in that test cases are single steps whereas scenarios cover a number of steps

- **Scenario testing is done to make sure** that the end to end functioning of software is working fine, or all the business process flows of the software are working fine.
- Elements:

✓ Setting
✓ Agents or actors
✓ Goals or objectives
✓ Plot
✓ Actions and events

Characteristics

- The test is **based on a coherent story** about how the program is used, including goals and emotions of people.
- The **story is credible**. Stakeholders will believe that something like it probably will happen.
- Failure of the test would motivate a stakeholder with influence to **argue it should be fixed**
- The story **involves complexity**: a complex use of the program or a complex environment or a complex set of data.
- Test results are **easy to evaluate**. This is important for scenarios because they are complex.

Benefits

- Learn the product
- Connect testing to documented requirements
- Expose failures to deliver desired results
- Explore export use of the product

16 lines of inquiry for suites of scenarios

1. List possible users. Analyze their interests and objectives.
2. Work alongside users (or interview them) to see how they work and what they do.
3. Interview users about famous challenges and failures of the old system.
4. Look at the specific transactions that people try to complete, such as opening a bank account or sending a message. What are all the steps, data items, outputs, displays, etc.?
5. Look for sequences: People (or the system) typically do task X in an order. What are the most common orders (sequences) of subtasks in achieving X?
6. Consider disfavored users. Analyze their interests, objectives, capabilities, and potential opportunities.

7. What forms do the users work with? Work with them (read, write, modify, etc.)
8. Write life histories for objects in the system. How was the object created, what happens to it, how is it used or modified, what does it interact with, when is it destroyed or discarded?
9. List system events. How does the system handle them?
10. List special events. What accommodations does the system make for these?

11. List benefits and create end-to-end tasks to check them.
12. Read about what systems like this are supposed to do. Play with competing systems.
13. Study complaints about the predecessor to this system or its competitors.
14. Create a mock business. Treat it as real and process its data.
15. Try converting real-life data from a competing or predecessor application.
16. Look at the output that competing applications can create. How would you create these reports / objects / whatever in your application?

Methods

- **System scenarios**
- In this method only those sets of realistic, user activities that cover several components in the system are used as scenario tests. Development of system scenario can be done using:
- Story lines
- State transitions
- Business verticals
- Implementation story from customers

Use-case and role-based scenarios

In this method the focus is on how a user uses the system with different roles and environment.

- Test script
- Test suite
- Session-based testing(time-**based** rather than context-**based**.)

- A **test script** in software testing is a set of instructions that will be performed on the system under test to test that the system functions as expected.
- There are various means for executing test scripts.
- Manual testing.
- Automated testing

- A **test suite**, less commonly known as a 'validation suite', is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviours.

Defect bash elimination

- Defect bash is a 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 bash is not based on written test cases.

- What is to be tested is left to an individual's decision and creativity.
- They can also try some operations which are beyond the product specifications.
- Defect bash brings together plenty of good practices that are popular in testing industry.

- They are as follows.

  .Enabling people “Cross boundaries and test beyond assigned areas”

  .Bringing different people performing different roles together in the organization for testing—“Testing isn’t for testers alone”

  .Letting everyone in the organization use the product before delivery—“Eat your own food”

  .Bringing fresh pairs of eyes to uncover new defects—“Fresh eyes have less bias”

- Bringing in people who have different levels of product understanding to test the product together randomly—“Users of software are not same”

  Let testing doesn’t wait for lack of/time taken for documentation—“Does testing wait till all documentation is done?”

System testing

- It conducts testing of whole software product to identify the errors in software.
- Types
  - -Functional
  - -Performance
  - -Stress
  - -Configuration
  - -Security
  - -Recovery
  - -Reliability
  - -Usability

- What is Acceptance Testing?
- Acceptance testing, a testing technique performed to determine whether or not the software system has met the requirement specifications.
- There are various forms of acceptance testing:

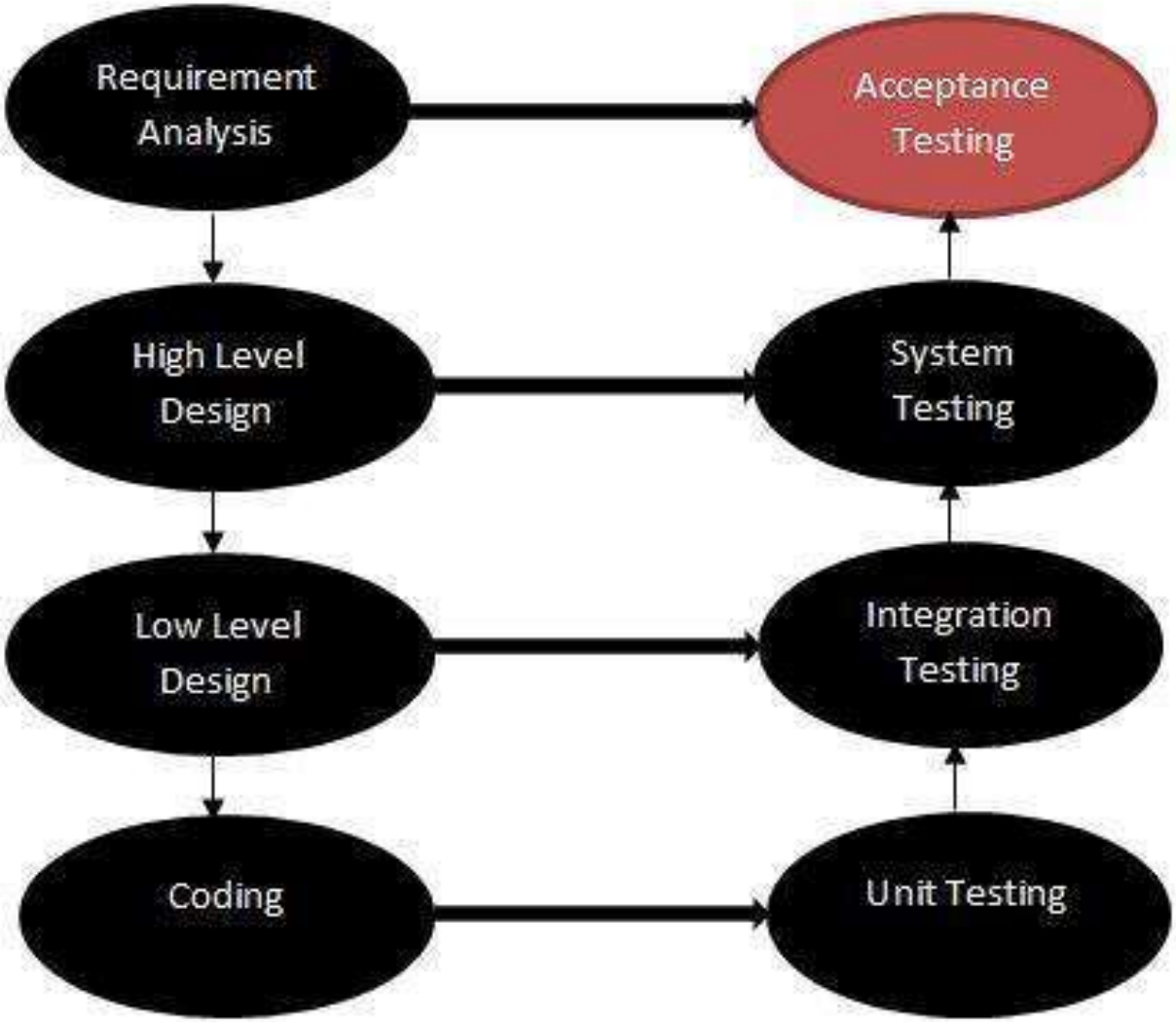
✓ User acceptance Testing

✓ Business acceptance Testing

✓ Alpha Testing

✓ Beta Testing

Acceptance Testing - In SDLC



- Acceptance Test Plan - Attributes
- The acceptance test activities are carried out in phases. Firstly, the basic tests are executed, and if the test results are satisfactory then the execution of more complex scenarios are carried out.

The Acceptance test plan has the following attributes:

- Introduction
- Acceptance Test Category
- operation Environment
- Test case ID
- Test Title
- Test Objective
- Test Procedure
- Test Schedule
- Resources

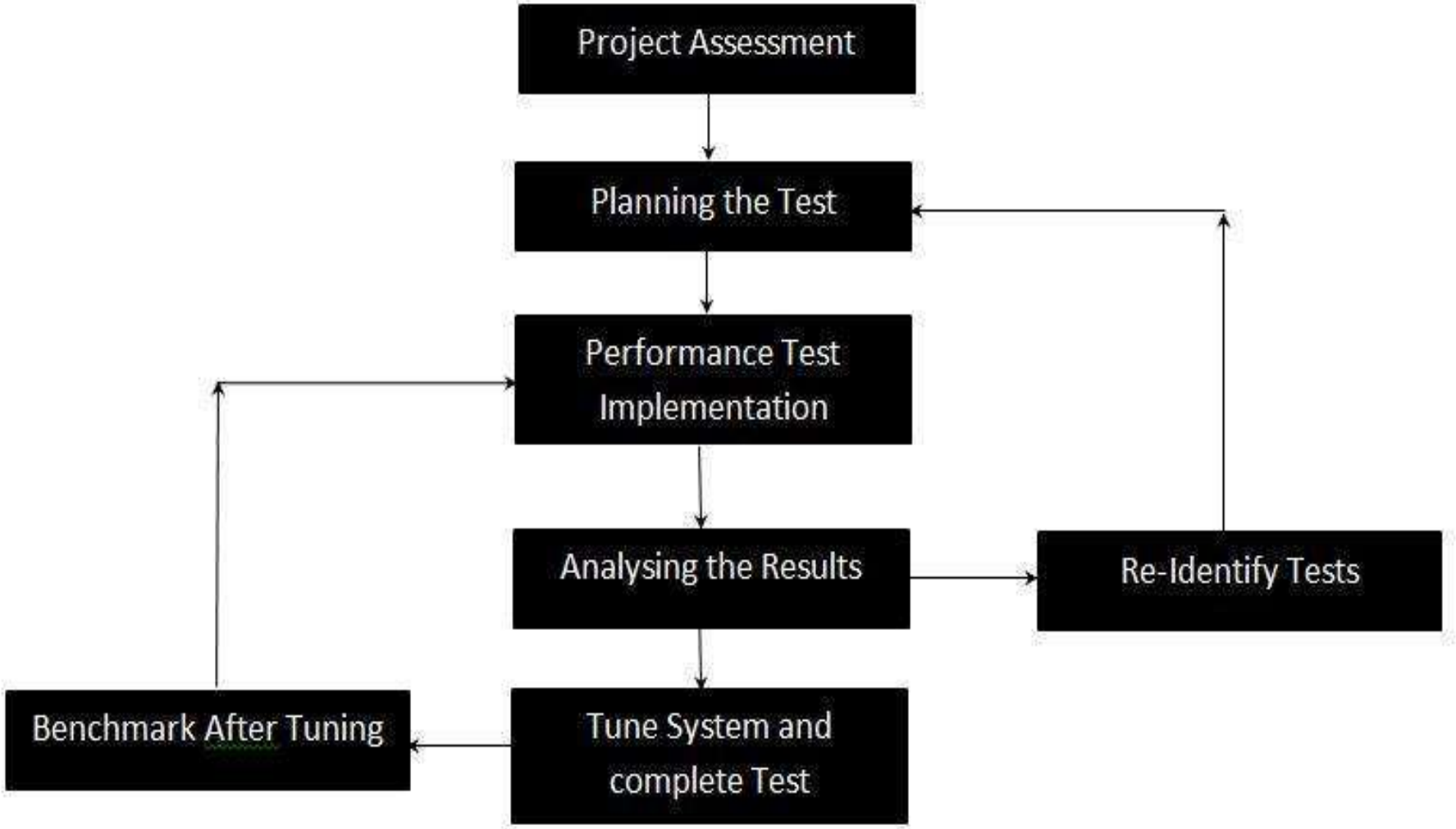
Types

- User Acceptance Testing (UAT) -**User acceptance** is a type of **testing** performed by the Client to certify the system with respect to the requirements that was agreed upon.
- Business Acceptance Testing (BAT)-BAT is conducted to verify whether the system can support day to day business and user scenarios to data correctness, various workflows, validate rules and overall fit for use and ensure whether the system fits business requirements.

Performance Testing

- Performance testing, a non-functional testing technique performed to determine the system parameters in terms of responsiveness and stability under various workload. Performance testing measures the quality attributes of the system, such as scalability, reliability and resource usage.

Performance testing process



What is Regression Testing?

- Regression testing a black box testing technique that consistsre-executing those tests that are impacted by the code changes. of

Selecting Regression Tests:

- Requires knowledge about the system and how it affects by the existing functionalities.
- Tests are selected based on the area of frequent defects.
- Tests are selected to include the area, which has undergone code changes many a times.
- Tests are selected based on the criticality of the features.

Regression Testing Steps:

- Select the Tests for Regression.
- Choose the apt tool and automate the Regression Tests
- Verify applications with Checkpoints
- Manage Regression Tests/update when required
- Schedule the tests
- Integrate with the builds
- Analyze the results