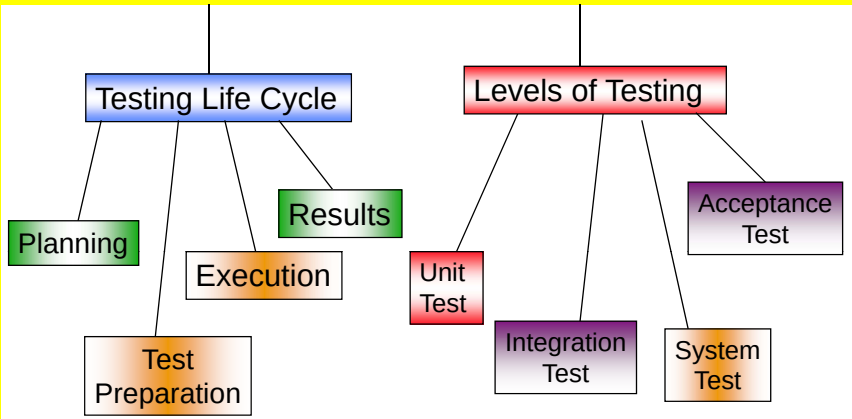
Course Structure

1. Software Quality Assurance
2. Testing Fundamentals – Part II
3. Code-based Techniques
4. Specification-based Techniques
5. Inspection Technique
6. Test Tools
7. Measuring Software Quality
8. TDD

Fundamentals of Testing

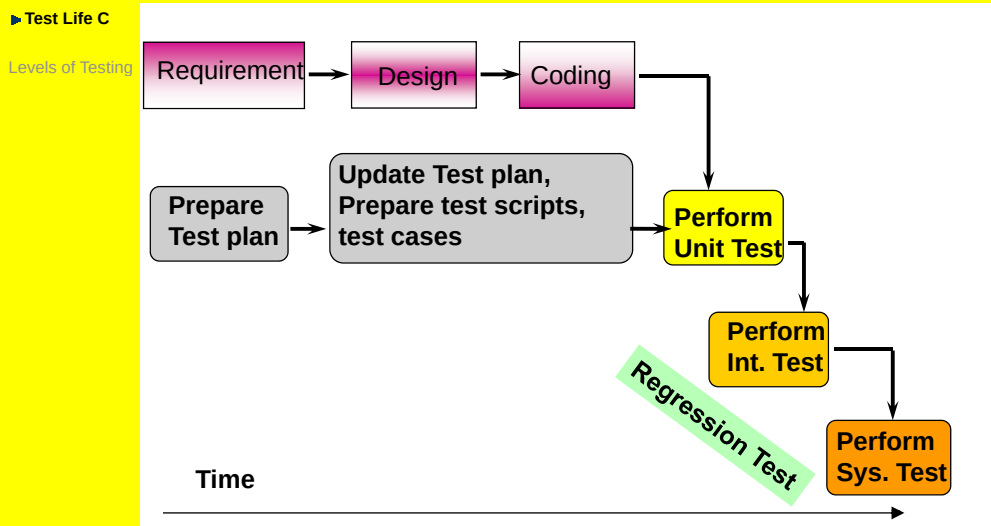


Learning Objectives

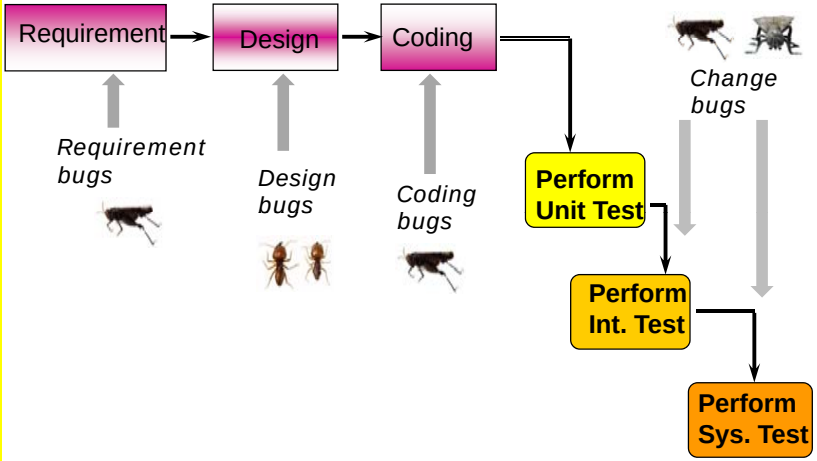
- Understand the software test life cycle
- Develop a test plan
- Understand the different levels of software testing (unit, integration, system, and acceptance testing), and how to effectively perform them
- Describe the common problems encountered in different levels of testing
- Understand 4 common integration testing strategies
- Apply different types of system testing.

"Testing proves the presence, not the absence of bugs." *E.W. Dijkstra*

Testing in the Development Life Cycle



Defects are Everywhere!



A Typical Testing Life Cycle: 4 Phases

Test Planning

Plan the general approach, resources, and schedule
Define testing objectives (determine features to be tested)
Refine the general plan

Test Preparation

Design the test cases
Implement the refined test plan

Test Execution

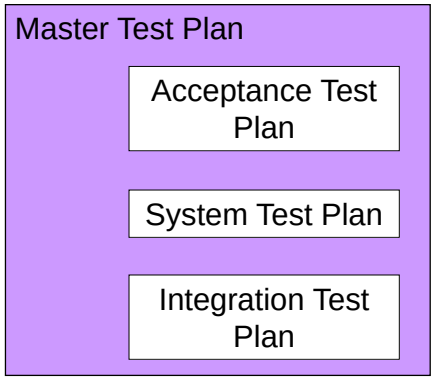
Execute the test procedures

Measure Test

Evaluate software quality
Evaluate the test quality

1. Planning for Testing

- Decide on overall test strategy
  - What type of testing (usability, performance) and their order
  - Whether to automate system tests
  - Whether there is an independent test team
- Decide on coverage strategy
- Identify the test cases and implement them
- Identify regression tests



Master Test Plan includes more detailed test plans.

1. Planning for Testing

- Planning begins at the start of the project

Test plan should include:

Responsibility	***Who will do the testing***
Activities	***What specific activities will be performed***
Deliverables	***What will the output of these activities be***
Standards	***What the policies and guidelines are***
Technique	***How to do the testing***
Metrics	***What metrics to use to measure quality***
Tools	***What will be used/developed to do the testing***
Cost	***How much will it cost***
Risk	***What can go wrong***

Defining Test Objectives (1)

- Specify the type of testing (what is going to be tested) and coverage (how thoroughly it is going to be tested)
- Similar test cases should grouped together, with the objective of each individual case clearly distinguished.

Testing types:

Basic function - tests the basic functions of the software and is useful criteria before going to the next phase of testing

New function - tests new functions

Old function - tests the ability of the software to continue to perform functions that it performed in previous releases (regression testing)

Defining Test Objectives (2)

Coverage Types:

Input domain coverage
Test a sample of the input, e.g., test valid input

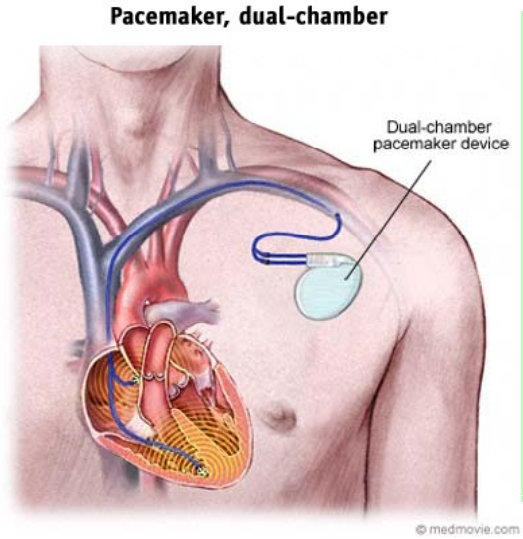
Output range coverage
Test to generate a sample of the output range, e.g., test to generate all reports

Structure coverage
Test a large % of structural elements, e.g., 95% of the branches; 100% statement executed

Defect response
Test all defect-handling codes

Interfaces
Test all interface to other subsystems and systems.

Example: Testing a Pacemaker



If the pacemaker fails, the user will die!

80K-100KLOC

The heart is a natural real-time system!

Common problems of pacemaker:
- Fast heart rate
- Slow heart rate

Example: Testing a Pacemaker

Example test requirement:

Req Tag	Feature	Type	Requirement Text
3426	Pace	Pos/Neg	When an Atrial Pace Pulse is requested the Main CPU software shall turn the Atrial Paced Activity LED ON for 100 ms.

Procedure for testing:

Procedure

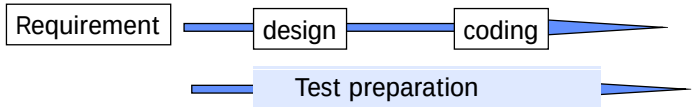
//Atrial only
For 5 cycles
Wait for 1 A
Verify Atrial Paced Activity LED is ON for 100ms
Verify Primary and Secondary Ventricular Paced Activity LEDs are OFF

Example C++ code snippet for test case:

```
459     // Verify the test results
460
461     gLog.WriteMessage("Atrial pacing only");
462     //Verify Atrial LED for 5 cycles
463     for(int i =0; i < 5; i++)
464     {
465         gESS.WaitForEvent("A", 1, 2);
466         gPSA.VerifyLEDStatus(RA_PACE_LED, LED_ON);
467         gPSA.VerifyLEDStatus(RV_PACE_LED, LED_OFF);
468         gPSA.VerifyLEDStatus(LV_PACE_LED, LED_OFF);
469
470         //Verify LED still ON
471         Sleep(LEDWaitTime - PROCESSING_DELAY);
```

2. Preparing for Testing

Test Life C
Planning
▸ Prepare
Execution
Measure
Levels of Testing



What need to be prepared?

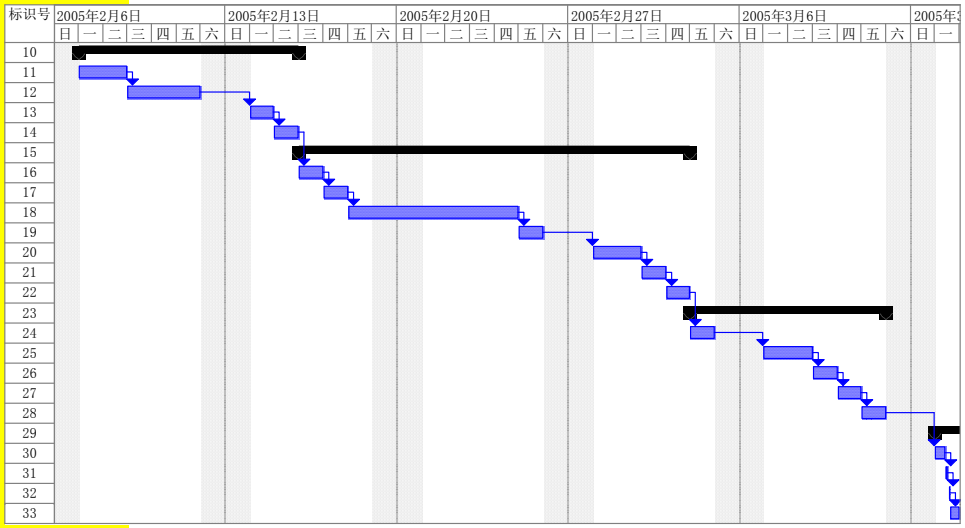
- **Input:** acquire or generate the input needed by the test cases
- **Environment:** develop the environment to be used for testing, including computer hardware and communication lines
- **Tools:** develop or get the necessary test drivers and test tools
- **Scaffolding:** develop or get the **stubs** or **drivers** needed
- **Test cases:** code test case scenarios and run test case generators to develop the test cases
- **Expected Output:** correct output for each test case.

Preparation takes time: One test team with a multinational bank found that they spent 2 days of preparation effort for each 2 hours of test execution.

Example Test Activities

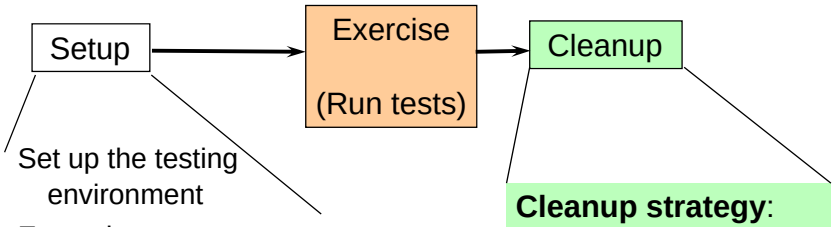
标识号	任务名称	工期	WBS 前置任务
1	Test Planning	7 工作日?	
2	Identify Test Project	1 工作日?	
3	Define Testing Strategy	1 工作日?	1.1
4	Estimate Work	1 工作日?	1.2
5	Identify Rsource	1 工作日?	1.3
6	Schedule Testing activities	1 工作日?	1.4
7	Document test plan	2 工作日?	1.5
8	Specify Test Cases	5 工作日	
9	Determine test cases	5 工作日	1.6
10	Design Test	7 工作日?	
11	Analyze test requirements	2 工作日	2.1
12	Specify test proceudre	3 工作日	3.1
13	Specify test cases	1 工作日?	3.2
14	Review test requirement coverage	1 工作日?	3.3
15	Implement Test	12 工作日?	
16	Establish test implementation environment	1 工作日?	3.4
17	Record and play back prototype scripts	1 工作日?	4.1
18	Develop test procedures	5 工作日	4.2
19	Test and debug test procedures	1 工作日?	4.3
20	Modify test procedures	2 工作日	4.4
21	Establish external data sets	1 工作日?	4.5
22	Re-test and debug test procedures	1 工作日?	4.6
23	Execute System Test	6 工作日?	
24	Set up a test system	1 工作日?	4.7
25	Execute tests	2 工作日	5.1
26	Verify expected results	1 工作日?	5.2
27	Investigate unexpected results	1 工作日?	5.3
28	Log defects	1 工作日?	5.4
29	Evaluate Test	1 工作日	
30	Review test logs	0.25 工作日	5.5
31	Evaluate coverage of test cases	0.25 工作日	6.1
32	Evaluate defects	0.25 工作日	6.2
33	Determine if test completion criteria are met	0.25 工作日	6.3

Example Test Schedule



3. Test Execution (1)

Test Life C
Planning
Prepare
▸ Execution
Measure
Levels of Testing



Set up the testing environment

Example:

➔ Making a phone line busy so that the call forwarding feature on it can be tested.

➔ Changing the status on an accounting record so that an error condition can be tested.

Cleanup strategy:

➔ Reversing any changes made during test execution.

➔ Restore the database by copying it from off-line to on-line storage

3. Test Execution (2)

Group tests together

- A popular method of managing multiple tests is to group them into **test sets** according to the business process, environment or feature.

  Example: all tests designed to verify the login process can be grouped under the 'Login' test set.

Realistic test

- Tests should realistically emulate end-user behavior (realistic usage scenarios).

  Example: if a user logs into the system, enters a new order and then exits the system, it makes sense to arrange the tests in the same sequence: login, insert order, logout.

4. Measure Test

Results should include

- **output** that can be captured for comparison and other analysis
- **reports** that include % of requirements executed, code executed
- **additional test cases** that can be added and executed if reports indicate insufficient coverage
- **discrepancy reports** to analyze test cases that fail

Summary: Testing Steps

1. Before developing a test, we should identify the **objectives** of the test
2. Decide how to carry out a relevant test. We have to decide which test is the most suitable and what sort of test items to use.
3. Develop the test case.
4. Determine the expected results of the test.
5. Execute the test cases.
6. Compare the test results to the expected results.

Responsibilities of Test Team

- ⌘ developing the test plan
- ⌘ defining required resources
- ⌘ designing test cases
- ⌘ constructing test cases
- ⌘ executing test cases according to the test plan
- ⌘ managing test resources
- ⌘ analyzing test results
- ⌘ issuing test reports
- ⌘ recommending application improvements
- ⌘ maintaining test statistics

Many Testing Overheads

Must include these overheads in estimating test effort:

- Unanticipated **time delays** in the availability of test resources, such as testers with specialized skills, test equipment, test db, tools. (5-15%)
- Unanticipated **delays** while waiting for bugs to be fixed. (5-20%)
- Unanticipated delays and **added work** because of enhancements or other changes to system. (5-20%)
- Problems in test environment. (5-15%)
- For multi-site testing projects, the overhead for travel, meetings and coordination. (0-5%)
- Overhead for test-related activities but outside the job for the testers, like version control, providing customer service, and psychological counseling with distraught programmers. (5-15%)
- Overhead for technical writing and user training. (5-30%)
- Turnover & multi-tasking. (5- 50%)
- Over-confidence that newly introduced technologies, tools and methodologies will work, and under-appreciate the debugging and fixing (10- 15%)

These activities can add 50-100% to the budget.

Don't under estimate the effort

In an internal study, IBM found that a loaded hour only includes about 0.65 hour of unloaded work time (time spent on actual work).

i.e., we only get about 5.5 hours of work in a standard 8-hour day. The rest is absorbed by activities like telephone calls, interruptions, and attending meetings not directly associated with the immediate task.

If we take the estimate of the unloaded hours for a project and divide by 8 to calculate the number of days required, instead of dividing by 5.5, we will **underestimate** by 35%.

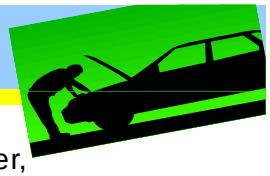
Data from Microsoft

For IE4.0,

- Code development: 6 months
- Stabling (include testing): 8 months
- Testing, stabling and rework can take 80% of the total time.

	Developer: Tester
Microsoft	1:2
China software company	6:1

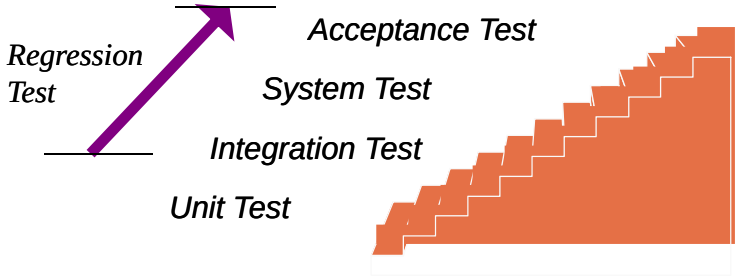
A True Story



A software engineer, a hardware engineer, and a manager were driving on a highway.
Suddenly, their car's brakes failed and they were able to stop in an emergency area.
Manager: "I think we should organize into a committee, divide up the tasks of analyzing the situation and recommend a solution."
Hardware engineer: "No, that will take too long. Look, I have a pocket knife and with a little time I am bound to find what's wrong with the car."

Finally, the **software engineer** said, *"Before we do anything else, I think we should push the car back up the hill and see if it happens again."* **(reproduce the problem)**

Levels of Testing



Each level of testing should define specific entry criteria and pass/fail acceptance or exit criteria.

Entry vs Exit Criteria

Entry criteria are certain conditions that allow us to start something.

Example: to make a phone call we have to have a working phone, a connection, and the phone number of the recipient.

Entry criteria for a phone call includes 3 conditions:

- A working phone is available.
- A connection is available.
- A phone number is known.

Exit criteria are certain conditions that allow us to declare that something is finished.

Example: lunch at the restaurant is finished when the bill is paid. So, Exit criteria for a meal at a restaurant is "The bill is paid."

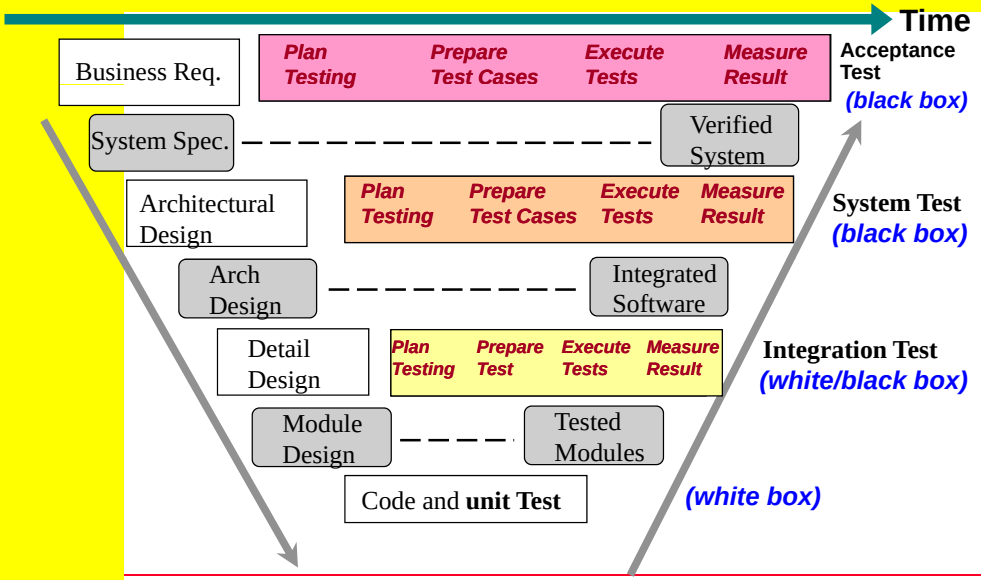
Example entry criteria:

test environment available, data available, code ready to be tested.

Example exit criteria:

No critical defects.

Testing Hierarchy (Levels)



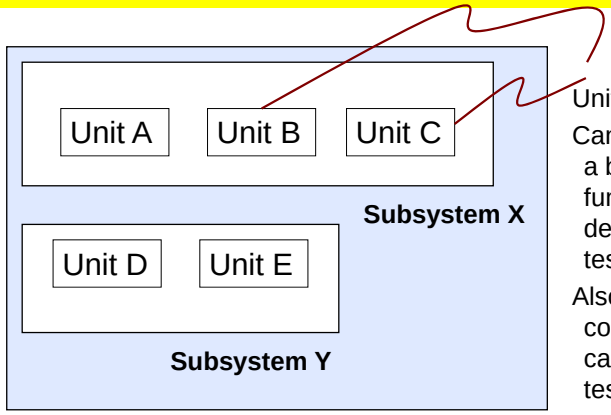
Testing Hierarchy (Levels)

Testing	Definition	Purpose	Traceability
Unit/ Component	Verifies the implementation of low-level design (e.g., function, module).	Ensures program logic is complete and correct. Ensures that component works as designed.	Traces each test to detailed design.
Integration	Test high-level design. Hardware and software elements are combined and tested until the entire system has been integrated.	Ensures that design objectives are satisfied.	Traces each test to a high-level design.
System	Test integration of entire hardware and software.	Ensures that software as a complete entity complies with its operational requirements.	Traces test to system requirements.
Acceptance	Determines if test results satisfy acceptance criteria of project stakeholders.	Ensures that objectives of stakeholders are satisfied	Traces each test to stakeholder requirements.

Unit Testing

Test Life C

▸ **Levels of Testing**



Unit test alone.

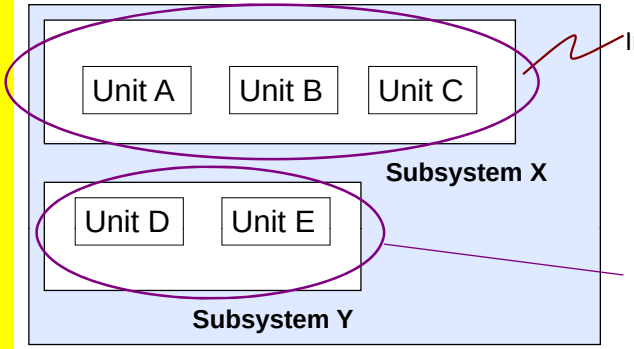
Can treat the unit as a black box. Use its functional description to create test cases.

Also, use the source code to create test cases (white box testing)

A system consists of several subsystems, which contains several units

A **Unit** can be a function, subroutine, module, method within class, package, display or menu

Integration Testing

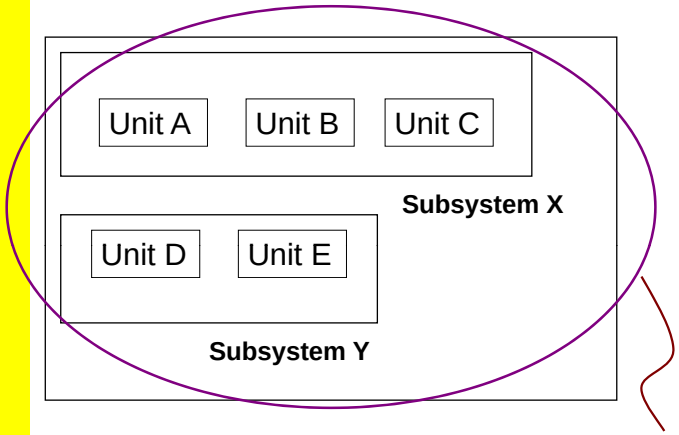


Integration test together.

Integration test together (mainly black box testing, also use white box testing.)

Independently, software might function normally, but when connected to other code, it may act unpredictably.
Hard to predict what will happen when 2 pieces of code come in contact with each other for the first time.

System Testing

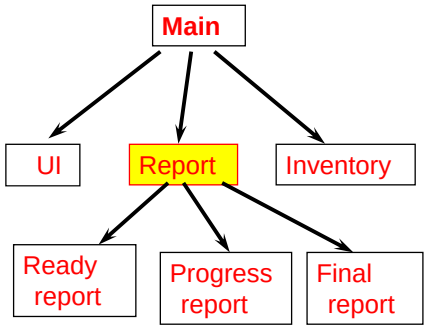


Test everything together

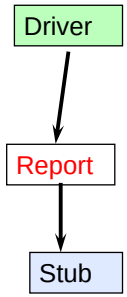
System Testing (black box testing)

Example Unit Testing

Example Call Graph:



Do unit test of **Report**



Report depends on other components that may not have been completed yet.

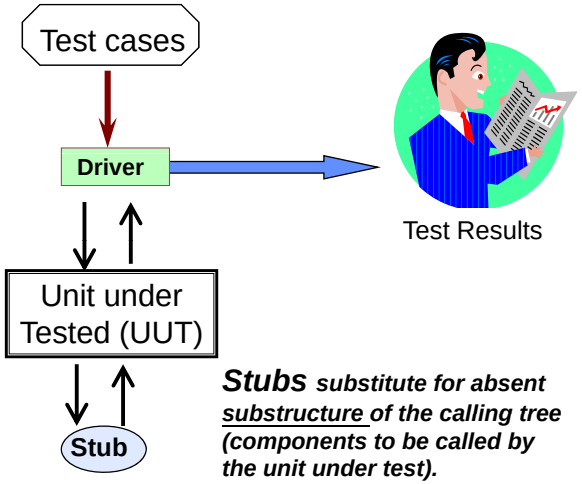
Simulate the function of 'Ready report', 'Progress report' and 'Final report'

What is Unit Testing?

- Test are designed to verify that an individual unit implements all design decisions made in the unit's design specification.
- Lowest level of testing, where individual units are tested in isolation from other parts of the system.

  "Is the foundation working?"
- Apply both code-based (white-box) and specification-based (black-box) testing
- Can be done in parallel for multiple modules
- A thorough unit test should include **positive** testing and **negative** testing.

Unit Testing Environment



***Drivers** substitute for absent superstructure of the calling tree (part of the system calling the unit under test).*

***Stubs** substitute for absent substructure of the calling tree (components to be called by the unit under test). It simulates part of the system that is not under test.*

Drivers

Drivers provide a framework for

- setting input parameters, and the environment
- executing the unit (call the UUT), and
- reading the output parameters (save results and checks they are correct).

Typical Behavior:

- ➔ supply a constant input, or a random input
- ➔ supply a 'canned' input
- ➔ record interim results and traces

Stubs

The response from the stub

- ➔ can be fixed,
- ➔ driven by a script, or
- ➔ derived from the input .

Example use of Stub:

- Simulation of target hardware dependent functions, which cannot be run in test environ.
- Simulation of the other functions
- Arranging hard-to-produce error

Typical behavior:

- ➔ exit immediately
- ➔ return a constant or random output
- ➔ return a value from the user
- ➔ print 'module x entered'
- ➔ return a primitive version of called module

Why do Unit Test?

An analogy:

- Try cleaning a fully assembled food processor
- No matter how much water and detergent is used, little scraps of food will remain stuck in awkward corners.
- But, if it is disassembled, the awkward corners either disappear or become much more accessible, and each part can be cleaned easily.

Unit tests are

- simpler to create,
- easier to maintain and
- more convenient to repeat than integration test.

Relative Cost of Unit Test

	Relative Cost
Unit test	3.3
Integration test	6.2 (2xUT)
System test	11.5 (3xUT)
Field test	11 (3xUT)

Ref: *'Applied software measurement'*, Capers Jones, 1991.

Cost based on time taken to prepare and execute tests, and fix defects (normalized to 1 function point).

Unit test is twice as cost effective as integration test, and more than 3 times as cost effective as system testing.

What to Test in Unit Test?

- interface (information flows into and out)
- local data structure
- boundary conditions (range checking, limit tests)
- independent paths (more on this when we look at white-box testing)
- ***error-handling*** paths (erroneous inputs)

Once a unit is tested successfully, it comes under configuration control.

Example Interface Test Cases

- the number of input parameters and arguments match?
- the parameter and argument attributes match?
- the file attributes correct?
- How are I/O defects handled?

Example Data Structures Test Cases

- improper or inconsistent typing
- erroneous initialization or default values
- incorrect variable numbers
- inconsistent data types
- underflows, overflows, and addressing exceptions

Error Handling

1. **How the system responds to errors made by users**;
   e.g., how the system responds if a Web form is submitted with invalid data in a required field.

2. **How the system reacts to errors that happen when the software is running**.
   Will the system give meaningful error message?

Error Handling paths are not the normal program paths. When the program is running and there is no error, these error-handling paths will not be invoked.

Entry and Exit Criteria of UT

Entry criteria:

- ⌘ Business requirements are at least 80% complete and have been approved to-date
- ⌘ Technical design has been finalized and approved
- ⌘ Development environment has been established and is stable
- ⌘ Code development for the unit is complete

UT

Exit criteria:

- ⌘ Code has version control in place
- ⌘ No known major or critical defects prevents any units from moving to integration testing
- ⌘ A testing transition meeting has been held and the developers signed off
- ⌘ Project manager approval has been received.

Unit Testing A Layered Architecture

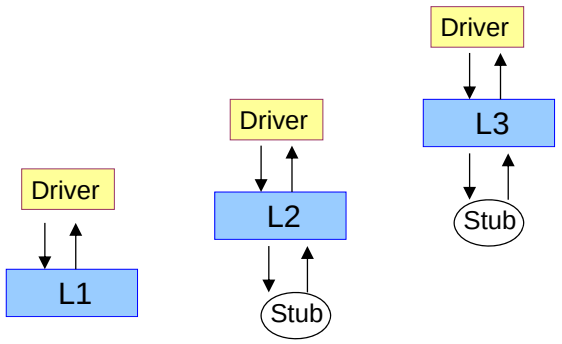
A system with a layered architecture

L3

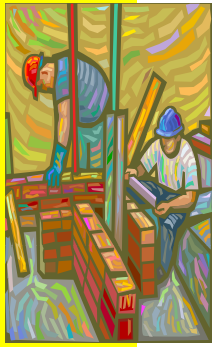
L2

L1

Layers



Testing should be done early



工匠一: 先拉上一根水平线，砌每一块砖时，都与这根水平线进行比较，使得每一块砖都保持水平。

工匠二: 先将一排砖都砌完，然后拉上一根水平线，看看哪些砖有问题，再进行调整。

你会选择哪种工作方法呢？你一定会骂工匠二笨吧！这样多浪费时间呀！

你平时在编写程序的时候又是怎么做的呢？我们就是按工匠二的方法在干活的呀！甚至有时候比工匠二还笨，是整面墙都砌完了，直接进行"集成测试"，经常让整面的墙倒塌。

你还觉得自己的方法高明吗？

Conclusion: Do UT!

Integration Testing

Test Life C
Levels of Testing
Unit Testing
▸Integration Testing
System Testing
Acceptance Testing

How well do the subsystems and components fit together?

What is Integration Testing?

Test Life C
Levels of Testing
Unit Testing
▸Integration Testing
System Testing
Acceptance Testing

- ⌘ Logically related units that were previously tested separately are tested together
  - Verify that many components work correctly together
  - Verify interaction between components
- ⌘ Check components/subsystems **interface** (e.g., interface from one menu to other menus): data exchanged between interfaces meets specification. Interface incompatibilities may be detected.
- ⌘ 50-75% of all errors are interfaces problems.
- ⌘ Misunderstanding/ miscommunication between developer results in mismatches in interface
- ⌘ Usually conducted by an independent test team.

Integration Test Plan

Test Life C
Levels of Testing
Unit Testing
▸Integration Testing
System Testing
Acceptance Testing

Integration Test Plan specifies:

- order of integration
- integration tests at each stage
- test harnesses (or environment)
- test data

End products: integrated system

Basic goal: test how modules/components interact with each other and with data, assuming they have passed unit testing.

- Execute module interactions:
  - ✓ Actual test data and expected results for each potential module interaction
  - ✓ Order of test executions: top-down, bottom-up, etc
  - ✓ Completion criteria: one test per interaction, input/output coverage, data flows coverage

3 Integration Strategies

1. **Incremental**
   Test each model in isolation
   Merge a set of modules (usually just 1 module) at a time to the set of previously tested modules

2. **Nonincremental**
   All modules are grouped together simultaneously and tested. Work well for relatively small projects.

3. **Mixed**
   Use both incremental and non-incremental strategies (Some parts use incremental, other parts use non-incremental)

Incremental Integration: Bottom-up

- The program is merged and tested from the bottom (lowest level) to the top of the call graph.
- All terminal modules are unit tested first in isolation; the next higher level modules are then tested, one at a time, with these tested modules.
- Need to write many drivers.

Advantages:

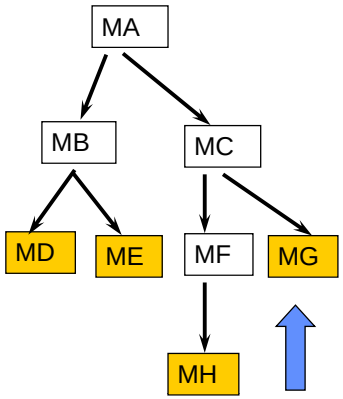
- Test cases may be designed solely from functional design information, requiring no structural design information.
- Particular useful for objects and reuse.

Disadvantages:

- The program as a whole does not exist until the last module is added. Major control and decision problems will be identified **later** in the testing process.
- Requires test drivers (which is easier to develop), not test stubs.
- As testing progresses up the hierarchy, bottom up testing becomes more complicated, and consequently more expensive to develop and maintain; it becomes more difficult to achieve good structural coverage.

Bottom-up Example

A program with the following call graph:

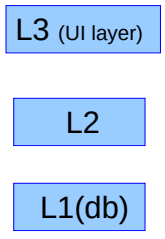


A possible sequence of bottom-up integration is:

- Unit test MD
- Unit test ME
- Unit test MH
- Unit test MG
- Integration test MD, MB
- Integration test MD, MB, ME
- Integration test MD, MB, ME, MA
- Integration test MH, MF
- Integration test MH, MF, MC
- Integration test MH, MF, MC, MG
- Integration test all modules

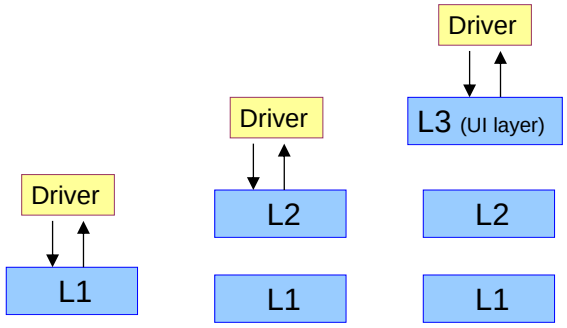
Int. Testing A Layered Architecture

A system with a layered architecture



Layers

Bottom-up strategy



Incremental Integration: Top-down

- Start by testing just the user interface. Starts with the top module, and adds one module at a time to the set of merged modules
- The top module is the only one unit tested in isolation, with all called modules replaced by stubs.
- Testing continues by replacing the stubs with the actual called modules, with lower level modules being stubbed.
- Repeat the process until the lowest level modules have been tested.

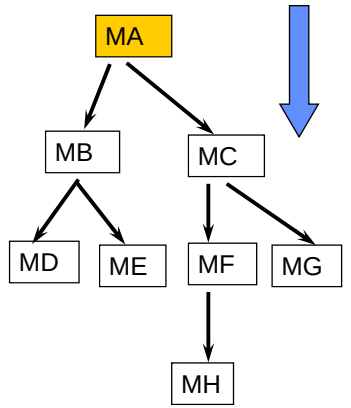
- Top down testing will provide an early integration of 'visible' functionality.
- As testing progresses down the hierarchy, it becomes more difficult to achieve good structural coverage. Testing some low level functionality, especially error handling code, can be impractical.

Advantages:
- A skeletal version of the program can exist early and allows demonstrations.
- Design errors may be found sooner.
- Reduces the need for test drivers.
- It tends to make fault location easier.

Disadvantage: require stubs which could be expensive to build.

Top-down Example

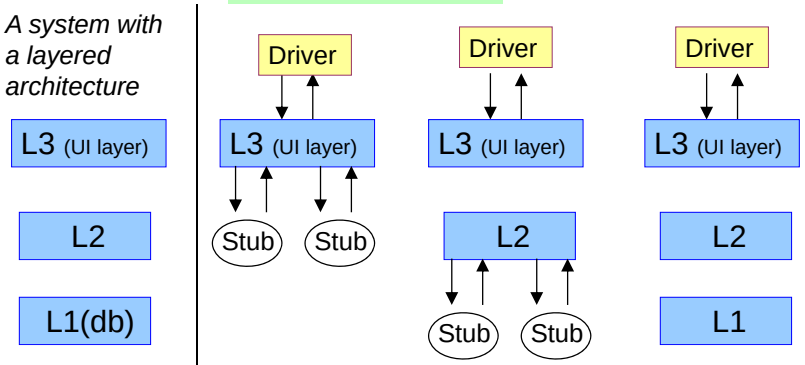
A possible sequence of top-down integration is:
- Unit test MA
- Integration test MA, MB
- Integration test MA, MB, MC
- Integration test MA, MB, MC, MD
- Integration test MA, MB, MC, MD, ME
- Integration test MA, MB, MC, MD, ME, MF
- Integration test MA, MB, MC, MD, ME, MF, MG
- Integration test all modules



Int. Testing a Layered Architecture

Top-down strategy

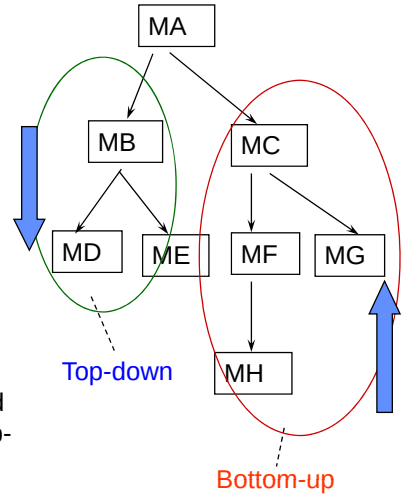
A system with a layered architecture



Incremental Integration: Sandwich

- Combine both the top-down and bottom-up testing
- Top-down and bottom-up testing are applied simultaneously and the program is integrated from both the top and the bottom.
- Test interfaces between portions already tested.

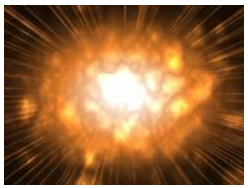
Disadvantages:
- requires careful thought to identify those low-level modules that should be developed early in an overall top-down framework.
- Some stubs & drivers required



Non-Incremental Integration

Big-Bang

- Each module is first tested in isolation
- Then, all the modules are merged together at once and tested; thus the name big-bang.



Advantages:

- *Integration time may be reduced.*
- *Relatively few tests required*

Disadvantages:

☹ *Least effective approaches, but often used in practice.*

☹ *Debugging is difficult since an error can be in any modules; hard to isolate errors*

Which Integration Strategy is better?

Summary of Integration Testing

	Bottom-up	Top-down	Big-bang	Sandwich
Integration	**Early**	**Early**	**Late**	**Early**
Time to working system	**Late**	**Early**	**Late**	**Early**
Drivers needed	**Yes**	**No**	**No**	**Yes**
Stubs needed	**No**	**Yes**	**No**	**Yes**
Ability to Test program paths	**Easy**	**Hard**	**Easy**	**Medium**
Ability to plan and control	**Easy**	**Hard**	**Easy**	**Hard**

Challenges in Integration Testing

- **Require good knowledge of system**: need knowledge of how the various parts should fit into the system. This requires someone who understands the product functionality and the internal units. Not easy to find!!
- **Schedule conflicts:** testing a module would require some other modules in order to proceed. These modules might not be available.

- **Assigning responsibility to problems:** When a problem comes up, it is difficult to pin-point where the problem lies (in which module) and who do the fix.
- **Communication:** ensuring smooth progress and defect resolution require good and well defined communication channels across the various groups.

Successful Integration Testing

Software configuration management. It is important the right version of different components are tested.

Must have a good Software Configuration Management policy.

Automate Build Process where Necessary: Many errors occur because the wrong version of components were sent for the build or missing components.

Document: Document the Integration process/build process to help eliminate the errors of omission or oversight.

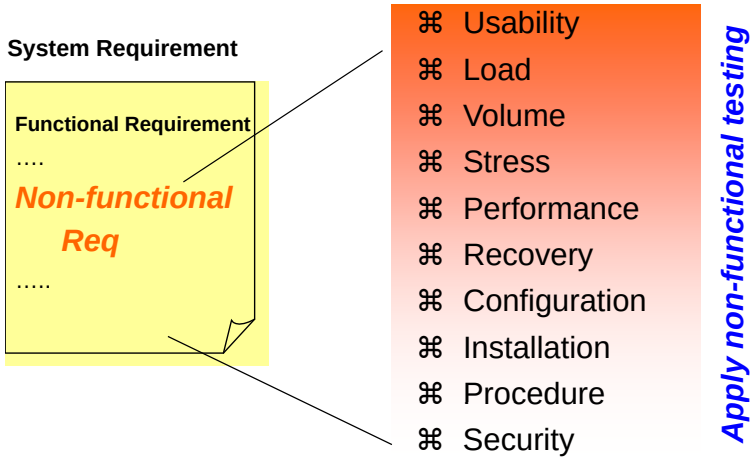
Software build refers to
(1) the *process* of converting source code files into standalone software artifact that can be run on a computer,
(2) the *result* of doing so.
One of the most important steps of a software build is the compilation process where source code files are converted into executable code.

Review Questions

1. Why is it important to thoroughly retest a defect after it has been reported fixed? Give 5 reasons.
2. Compare SQA and V&V. Give 5 differences.

System Testing

Test Life C
Levels of Testing
Unit Testing
Integration Testing
▸ **System Testing**
Acceptance Testing



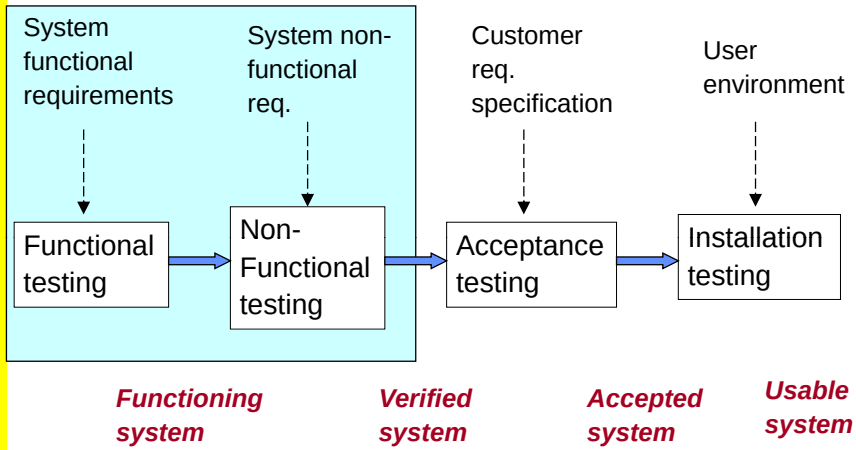
Not all types of system testing are mandatory for every application.

What is System Testing?

Test Life C
Levels of Testing
Unit Testing
Integration Testing
▸ **System Testing**
Acceptance Testing

- ⌘ Test the complete system (fully integrated) in the real environment in which it is to operate
- ⌘ **Ensure system requirements are met "Does the system works as a whole?"**
- ⌘ *Apply **functional testing** + **non-functional testing***
- ⌘ Results are sometimes used for system acceptance
- ⌘ Verify the Software User Manual
- ⌘ Estimate reliability and maintainability

System Testing Steps



Let's look at different types of system testing.

Smoke Test

- It is the First system test
- An informal quick-and-dirty run through of the major functions without bothering with details.
- The term "smoke test" comes from the hardware testing practice of turning on a new piece of equipment for the first time and considering it a success if it doesn't start smoking or burst into flame.

Example smoke test

Turn on ignition and check: engine idles without stalling, can put into forward gear and move 10 feet, then brake to a stop, wheels turn left and right under control.

What's wrong with this UI?

Another Usability Example

If a user forgets to fill in a required field, we might think it is a good idea to present the user with a friendly error message and change the color of the field label to red or some other color.

But, changing the color of the field label would not really help a user who is color-blind.

Enter name

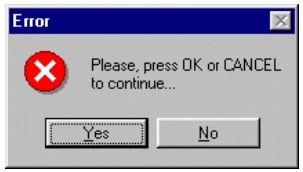
Better to use an additional clue, such as placing an asterisk beside the field in question or additionally making the text bold.

* **Enter name**

Usability

ISO 9241 definition

the *effectiveness, efficiency* and *satisfaction* with which a specified set of users can achieve a specified set of tasks in particular environments (context of use).

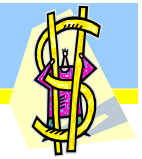


Usable product:

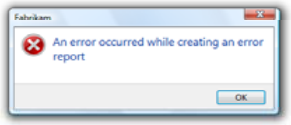
- ⌘ easy to learn
- ⌘ efficient to use
- ⌘ easy to remember
- ⌘ fun to use
- ⌘ visually pleasing
- ⌘ provides quick recovery from errors

Usability applies to the whole product!

Why Test Usability?



- 50% of code is directly related to the interface.
- 30% of software development budgets are devoted to user interface development.
- 80% of software maintenance costs are related to interface redesign.
- Up to 64% of life-cycle costs of software products are directly related to the user interface.



What to Test for Usability?

- ⌘ **Accessibility:** Can users enter, navigate, and exit with ease?
- ⌘ **Responsiveness:** Can users do what they want, when they want, in a way that's clear?
- ⌘ **Efficiency:** Can users do what they want in a minimum number of steps and time?
- ⌘ **Comprehensibility:** Do users understand the product structure, its help system, and the documentation?

Performance Testing

Include:

- **Load testing**
  - ✓ Test normal load
  - ✓ Test max. load
  - ✓ **Volume testing** (test large data)
- **Stress testing** (test low system resource)

Why do load, stress and volume Test?

We want to answer:

- *How many users can use the system at the same time?*
- *Can the system handle a very large file with 1,000,000 records?*
- *Can the system provide acceptable response time when 1000 users using the system simultaneously?*

Definitions

Throughput: the **rate** at which the system processes transactions (e.g., 200 transactions per second)

Processing delay: the time that it takes to process those transactions (e.g., 0.5 second)

Performance refers to the time the application takes to perform a requested task**.**

Load: the rate at which transactions are submitted to a software (measured in arriving transactions per second.) e.g., 100 users using the system at the same time, each invokes 2 transactions/second. Total load: 200 transaction per second

Load Profile: Define the level of system loading expected to occur for a specific business scenarios

Example: Inventory Control System:

Distribution	*Type of user*	*What functions used*
10% users	administrators	function A, B and C
80% users	customers	function K
10% users	managers	function A, and D

Example Response Time Requirements

System Response Time	Performance Standard	Service Level
a) Public search		
i) Before design is identified		
- Exact match	<= 1 sec.	90%
- Wild card	<= 10 sec	90%
ii) After design is identified		
- retrieval of first page of image	<= 10 sec	90%
- retrieval of all subsequent pages	<= 20 sec	90%
b) Record creation	<= 5 sec.	90%
c) Internal record retrieval	<= 5 sec	90%
d) Record updating	<= 5 sec	90%
e) Record printing (per page)	<= 10 sec	90%

90% of the cases should meet the performance standard.

From IPD

Example Volume Requirements

1. Application for registration of designs

Year	2000	2001	2002	2003		
High estimate	5985	6585	7245	-	-	
Best estimate	4665	5130	5640	-	-	
Low estimate	3330	3660	2025	-	-	

2. Renewal of registered designs

Year	2000	2001	2002	2003		
High estimate	-	-	7950	8745		
Best estimate	-	-	5925	7590		
Low estimate	-	-	3900	6435		

3. General enquiries

Year	2000	2001	2002	2003		
High estimate	3750	4125	4538	4992		
Best estimate	2475	2723	3000	3300		
Low estimate	1200	1320	1455	1605		

From IPD

Load Testing

- **Load testing** is a performance test which subjects the system to varying workloads (e.g., 100 users, 500 users, 1000 users) to measure and evaluate its performance behaviors and ability to continue to function properly under these different workloads.
- Load test stimulates the average load of normal business operation (normal usage of the system) and the expected maximum workload.
- Also evaluates **response times, transaction rates**, and other time sensitive issues.
- Use historical data or the system specification to determine the load.

Volume Testing

- **Volume Testing** is a type of load test.
- It subjects the system to large amounts of **data** to determine if limits are reached that cause the system to fail.
- Volume testing also identifies the continuous maximum load or volume the system can handle for a given period.

Example

- If the system is processing a set of database records to generate a report, a Volume Test would use a large test database and check that the software behaved normally and produced the correct report.
- If the system reads text files as inputs, try both an empty text file and a hugh text file.

Stress Testing (1)

- ⌘ Stress testing is a type of performance test to find errors ***due to low resources*** *(e.g. memory, processing power)*.
- ⌘ Check whether the system will handle the volume of activities when the system is at the highest processing demand. e.g., the phone system at mother day.
- ⌘ Try tests requiring
  - ➔ maximum memory, maximum data rate.
  - ➔ extended operation (long time)
  - ➔ peak period cases (e.g., year end)
  - ➔ start-up and shut-down situation
  - ➔ Saturation (overload)

Stress Testing (2)

- ☹ *It takes large amount of time to prepare for the test plus resources consumed during actual execution*
- Low memory or disk space may reveal defects in the system that aren't apparent under normal conditions.
- Other defects might result from competition for shared resource like database locks or network bandwidth.

Most heavy workload occurs at **peak periods**

Example peak periods:

1. 70% of workload on most systems will come from 2 peak periods:
   - ⌘ 10:00 – 12:00 (morning)
   - ⌘ 14:00 – 16:00 (afternoon)
2. Telephone system at Christmas eve.

Performance Testing

Objective of PT: To demonstrate that the system functions to specification with *acceptable response times* while processing the required transaction volumes on a production sized database.

Test cases are designed to:

- Determine the performance of the system
- Verify the optimum use of hardware and software
- Determine response time to user requests
- Determine transaction processing turnaround time

Should simulate the actual working environment.

Guideline: Performance Testing

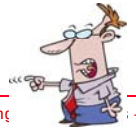
(1) The application must pass its functional tests before testing its performance.

Otherwise, we waste lots of time.

(2) **Performance tuning** consists of finding and eliminating bottlenecks – a condition that occurs when a piece of hardware or software in a server approaches the limits of its capacity.

To correctly tune performance, we must maintain accurate and complete record of each test - exact system configuration, raw data and calculated results from monitoring tools.

More on performance testing when we discuss Test tools



Recovery Testing (1)

- Check ability to recover or restart the system after failures such as
  - loss of input capability;
  - loss of communication;
  - operator error;
  - network malfunction;
  - device I/O failure;
  - invalid database pointer or key;
  - hardware or OS failure.
- Force the software to fail in variety of ways and verify that recovery is properly performed
- Recovery processes are invoked and the system is monitored and / or inspected to verify proper system and data recovery has been achieved.

Recovery Testing (2)

- Check
  - adequate backup data is preserved
  - backup data is stored in a secure location
  - recovery procedures are documented
  - recovery personnel have been assigned and trained
  - recovery tools have been developed and are available.
- The amount of potential loss should determine the amount of resources to be put into disaster planning and recovery testing.
- **HKSAR has a Disaster Recovery Bureau Centre to service other dept in case the primary system fails.**

See sample recovery plan for SQL server:

Configuration Testing

- **Configuration testing** verifies the operation of the system on different software and hardware configurations.
- For production environments, the particular hardware specifications for the client workstations, network connections and database servers vary. Client workstations may have different software loaded (e.g. applications, drivers, etc.) and at any one time many different combinations may be active and using different resources.

Configuration A	Configuration B	Configuration C

Example Configuration

To run our ABC software application:

Configuration

- Required Hardware

  Processor: 2 x Xeon 2.4 Ghz/512kb
  Network controller: 10/100/1000BT network controller
  Hard disk: 500GB Ultra 320 10kpm hot-swap hard disk
  Memory: 50GB DDR ECC RAM
  Floppy Drive: 1

- Required Software

  Small Business Server 2000 English CD/DVD 5 Clt
  MetaFrame XPs starter System with subscription
  Symantec Antivirus Corporate Edition 8.1

Installation Testing

- Problems installing and uninstalling applications is frustrating.
- Installation testing is a common omissions in many test plans.
- Bad installation instructions cause unhappy user.

Purposes

1. The product is packaged correctly.
2. Insure that the product can be installed (using the instructions from the installation documentation) under different conditions, such as a new installation, an upgrade, and a complete or custom installation, and under normal and abnormal conditions (insufficient disk space).
3. Verify that, once installed, the software product operates correctly. Run a few functional tests.

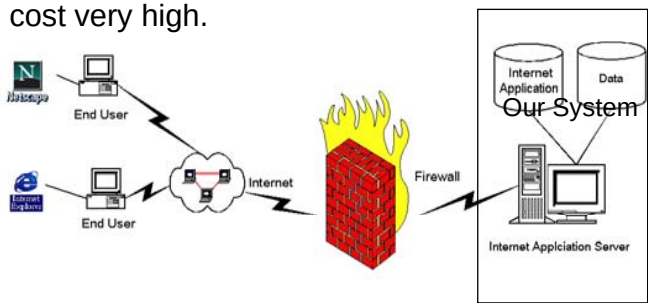
Procedure Testing

- ⌘ Check clarity of documentation on operation and use of system by having users follow the manuals
- ⌘ Ensure that the necessary support mechanisms, such as Job Control Language (JCL), have been prepared and function properly
- ⌘ Evaluate the completeness of operator training
- ⌘ Test to ensure that operators using prepared documentation can operate the system.
- ⌘ Evaluate the process and the execution of the process

Security Testing

Two types: Physical and logical security

- ⌘ Verify that protection mechanisms built into the system can protect it from improper penetration
- ⌘ The tester tries to penetrate the system. e.g., break the password, cause system errors.
- ⌘ Any system can be broken into. Just try to make the penetration cost very high.



Challenges in System Testing

- **Defining realistic scenarios:** what constitutes a good system test for one customer may be inappropriate for another customer.
- **Unique mix of skill sets needed:** system testing requires a big-picture view and detailed knowledge of features of the system. Not many testers have this.
- **Resources needed**: tests like perf. tests require expensive hardware configurations.
- **Reproducibility:** most problems that show up during system testing because of a complex combination of factors from the product, the software environment and the hardware configuration. Hard to reproduce.
- **Problem identification:** problem analysis and responsibility allocation become a serious issue.
  - ✓ Does the problem lie in the product or in the environment?
  - ✓ Which should be corrected?

ST

Entry Criteria

- ⌘ Unit testing for each module has been completed and approved
- ⌘ Each unit is under version control
- ⌘ An incident tracking plan has been approved
- ⌘ Test scripts and schedule are ready
- ⌘ A system testing environment has been established
- ⌘ The system testing schedule is approved

Exit Criteria

- ⌘ Application meets all documented business, functional and non-functional requirements
- ⌘ Performance, stress, and load tests were satisfactorily conducted
- ⌘ Outstanding defects have been identified, documented and presented to the sponsor.
- ⌘ No known critical defects prevent moving to UAT
- ⌘ All appropriate parties have approved the completed tests
- ⌘ A testing transition meeting has been held and everyone has signed off.

User Acceptance Testing

Test Life C
Levels of Testing
Unit Testing
Integration Testing
System Testing
▸ **Acceptance Testing**

Testing moves from the hands of Development team into those of the users.
Test to see if software satisfies the customer's requirements.

Alpha test

Software ← Customer tests

Developer site *Customer site*

Beta test

Developer reviews ← Test results — Software

Developer site *Customer site*

Alpha Testing

- Conducted by users at the **developer's site** (a controlled testing)
- Developers create a testing setting comparable, if not identical, to the target operational setting of the software
- Developers observe and document error and operation problems and correct them before testing terminates

Beta Testing

- Uncontrolled testing
- Conducted at one or more **client sites**, where customers use the software for real-world tasks
- Errors and operations problems must be faithfully documented and communicated to project team
- Mutual trust is essential for successful Beta testing

Example Beta Release Criteria

Test Life C
Levels of Testing
Unit Testing
Integration Testing
System Testing
▸ **Acceptance Testing**

Beta Release Criteria
1. 98% of all the test cases have passed.
2. The system has not crashed in the last one week of all testing.
3. All the *resolved* defects must be in the CLOSED state.
4. A release note with all the defects that are still in OPEN state along with workaround must be available. If the workaround does not exist, then an explanation of the impact on the customer must be incorporated in the release note.
5. No defect with "critical" severity is in the OPEN state.
6. No defect with "high" severity has a probability of hitting at a customer site of more than 0.1.
7. The product does not have more than a certain *number* of defects, with severity "medium." The *number* may be determined by the software project team members.
8. All the test cases for performance testing must have been executed, and the results must be available to the beta test customer.
9. A beta test plan must be available from all the potential beta customers. A beta test plan is nothing but user accepatnce test plan.
10. A draft of the user guide must be available.
11. The training materials on the system are available for field engineers.
12. The beta support team must be available for weekly meeting with each beta customer.
13. All the identified beta blocker defects are in the CLOSED state.

Beta Test at Google

From Google's press center
(http://www.google.com/intl/en/press/descriptions.html)

Google's product development philosophy is centered on rapid and continuous innovation, with frequent releases of new technologies that we seek to improve with every iteration. We often make products available early in their development stages by posting them on Google Labs, at test locations online or directly on Google.com. If our users find a product useful, we promote it to "beta" status for additional testing.
Our beta testing periods often last a year or more.
Once we are satisfied that a product is of high quality and utility, we remove the beta label and make it a core Google product.

User Acceptance Testing

- Select users carefully.
  A mix of business users with varying degrees of experience and subject matter expertise need to participate in a controlled environment.
- The application is tested in real-world situations.
- The tests should cover the full range of business usage.

Common occurrence in UAT:
Once the business users start working with the application they find that it doesn't do exactly what they want it to do or that it does something that, although correct, is not quite optimal.
=> **Many *change requests***

UAT

Entry Criteria:
- ⌘ System testing signoff was obtained
- ⌘ Business requirements have been met or renegotiated with the Business Sponsor
- ⌘ UAT test scripts are ready for execution
- ⌘ The testing environment is established
- ⌘ Security requirements have been documented and necessary user access obtained

Exit criteria:
- ⌘ UAT has been completed and approved by the user community
- ⌘ Change control is managing required modifications and enhancement
- ⌘ Business sponsor agrees that known defects do not impact a production release (no critical or major defects)

How long is this ***Beta*** guy going to keep testing our program?
Real-Life Dilbert Quotes

Summary

- ⌘ Plan testing effort by allowing time for fixing errors.
- ⌘ A good test case is one that has a high probability of detecting defects.
- ⌘ For small-sized projects, the test plan for integration and system testing could be combined.
- ⌘ Test team also finds defects when **preparing** test, not only when running tests!
- ⌘ It costs less to detect and fix defects in the early phases compared to later phases.

- Testing is performed with the primary intent of finding faults in the system.
- 4 levels of testing: unit, integration, system, acceptance.
- Unit test is more effective than integration and system testing.
- Integration testing strategies include top-down, bottom-up, sandwich, and big-bang.
- Integration testing requires test drivers and test stubs.
- System testing involves usability, stress, load, volume, performance, configuration, installation, procedure, and security testing.
- Acceptance testing test the complete system for satisfaction of requirements.
- Don't forget regression testing – rerun test cases every time a change is made.

References

- Usability Inspection Methods, edited by Jakob Nielsen and Robert L. Mack.
- Handbook of Usability Testing, by Jeffrey Rubin.
- Deborah Mayhew, The Usability Engineering Lifecycle. Morgan Kaufmann; 1st edition, April 15, 1999
- http://www.microsoft.com/technet/itsolutions/cits/mo/winsrvmg/wsrvtpo/wsrvtpo2.mspx

Latest News

NEW YORK - People for the Ethical Treatment of Software (PETS) announced today that more software companies have been added to the group's "watch list" of companies that regularly practice software testing.

"There is no need for software to be mistreated in this way so that companies like these can market new products," said Ken Granola, a spokesman for PETS. "Alternative methods of testing these products are available."

According to PETS, these companies force software to undergo lengthy and arduous test - often without rest - for hours or days at a time. Employees are assigned to "break" the software by any means necessary and inside sources report that they often joke about "torturing" the software.

"It's no joke," Granola said. "Innocent programs, from the day they are compiled, are cooped up in tiny rooms and 'crashed' for hours on end. They spend their whole lives on dirty, ill-maintained computers, and they are unceremoniously deleted when they're not needed anymore."

Granola said that the software is kept in unsanitary conditions and is infested with bugs.

"We know alternatives to this horror exist," he said, citing industry giant Microsoft Corp. as a company that has become successful without resorting to software testing.

Supplementary Notes

Top 24 replies by programmers when their programs don't work:

24. "It works fine on MY computer"
23. "Who did you login as ?"
22. "It's a feature"
21. "It's WAD (Working As Designed)"
20. "That's weird..."
19. "It's never done that before."
18. "It worked yesterday."
17. "How is that possible?"
16. "It must be a hardware problem."
15. "What did you type in wrong to get it to crash?"
14. "There is something funky in your data."
13. "I haven't touched that module in weeks!"
12. "You must have the wrong version."
11. "It's just some unlucky coincidence."
10. "I can't test everything!"
9. "THIS can't be the source of THAT."
8. "It works, but it's not been tested."
7. "Somebody must have changed my code."
6. "Did you check for a virus on your system?"
5. "Even though it doesn't work, how does it feel?"
4. "You can't use that version on your system."
3. "Why do you want to do it that way?"
2. "Where were you when the program blew up?"
1. "I thought I fixed that."

taken from xtremetesting

An example: Telematics system

Integrated in-car entertainment and infotainment components: navigation system, radio, audio amplifier, CD changer.

Telematics features are realized by the simultaneous interplay of these components (e.g., receiving a telephone call, the CD player is paused and the audio amplifier allocates audio channels to allow for hands free speech control)

Layout of a telematics module:

1 Main processor
2 GPS
3 Integrated telephone

Connections:

4 GPS antenna
5 Main antenna
6 Emergency antenna
7 Bluetooth antenna

Telematics example

1. Each component is tested alone to verify that the component specification is met and that the basic functionality is available.
2. Integration test verifies applications (e.g., telephony, navigation etc) that are distributed across a number of devices. The communication between the various devices is checked for conformance to the system specification.
3. System test focuses on interoperability of the various features within the telematics system and the *interaction between the telematrics system and its environment*.

Testing at Microsoft

All new hires takes a 2 weeks crash course in testing, to learn the white and black box testing, security testing, testing tools, advanced debugging, etc.

All Microsoft products must get test sign-off on product quality before intermediate and final releases.

Test professionals at Microsoft work closely with software design engineers and program managers to understand product requirements and functionality, design appropriate test plans and test cases to verify features and functionalities, and then identify bugs through systematic testing.

Test professionals also identify business improvement opportunities and potential future projects.

Job Titles in Microsoft

Test Engineer
- Tests and critiques software to assure quality and identify potential improvement opportunities and projects.

Test Lead
- Leads a team of test engineers
- Responsible for the quality of certain product areas/ modules.

Test Manager
- Oversees product testing within a product unit, designs master test plans and schedules, and manages test organization.

Design Engineer in Test
- Tests and critiques software components and interfaces in more technical depth, writes test programs, develops test tools to increase effectiveness.

Test Architect
- Identifies test design and implementation improvements, runs pilot projects to measure their impact, and drives adoption across various Microsoft product development teams.