Education and Research

We enable you to leverage knowledge anytime, anywhere!

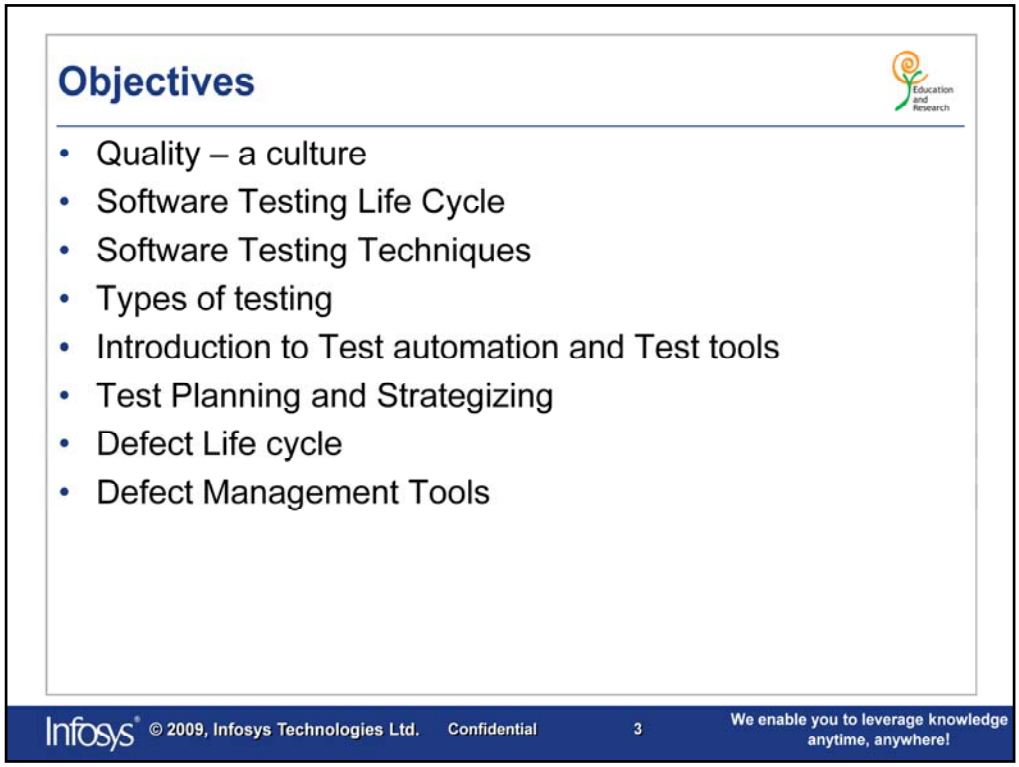
Software Testing

Education & Research

"© 2009 Infosys Technologies Ltd. This document contains valuable confidential and proprietary information of Infosys. Such confidential and proprietary information includes, amongst others, proprietary intellectual property which can be legally protected and commercialized. Such information is furnished herein for training purposes only. Except with the express prior written permission of Infosys, this document and the information contained herein may not be published, disclosed, or used for any other purpose."

Confidential Information

- *This Document is confidential to Infosys Technologies Limited. This document contains information and data that Infosys considers confidential and proprietary ("Confidential Information").*
- *Confidential Information includes, but is not limited to, the following:*
  - *Corporate and Infrastructure information about Infosys;*
  - *Infosys' project management and quality processes;*
  - *Project experiences provided included as illustrative case studies.*
- *Any disclosure of Confidential Information to, or use of it by a third party, will be damaging to Infosys.*
- *Ownership of all Infosys Confidential Information, no matter in what media it resides, remains with Infosys.*
- *Confidential information in this document shall not be disclosed, duplicated or used – in whole or in part – for any purpose without specific written permission of an authorized representative of Infosys.*



Testing is a stage where the quality of the software is measured. To set the stage, first we need to know what is 'quality' so as to imbibe it as a culture during software development.

We will discuss about 'software testing life cycle', a subset of 'software development life cycle' (SDLC).

Main objective of software testing is uncover as many defects as possible in the software. We will see various testing techniques which will help in detecting maximum defects.

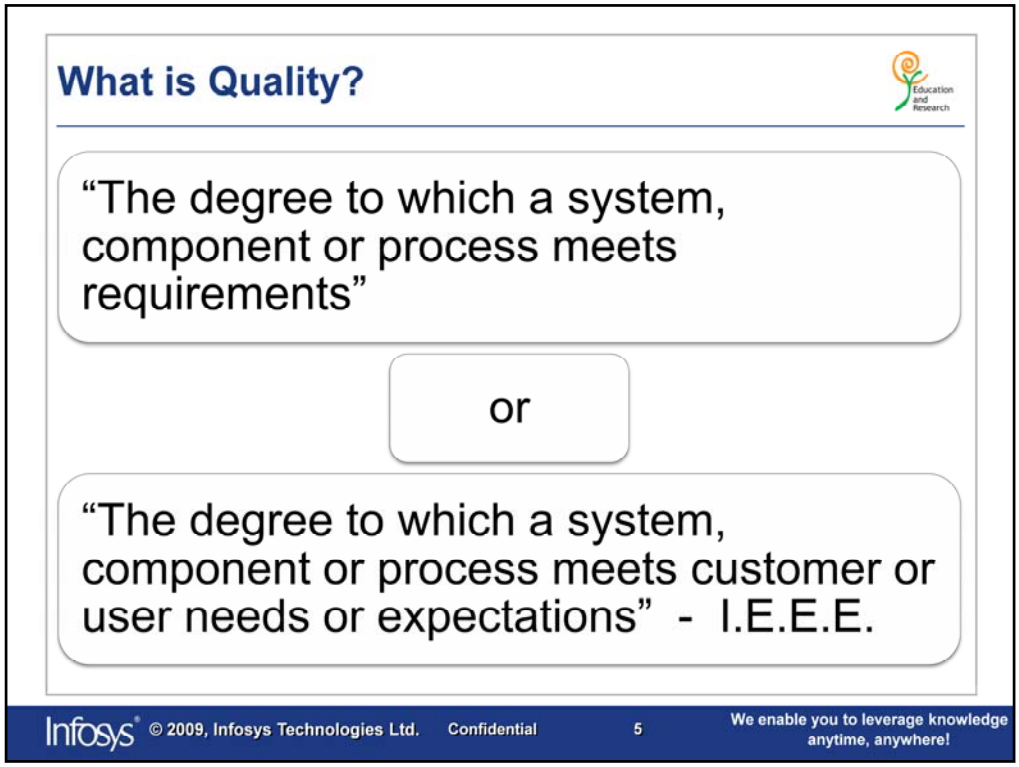
There are different types of testing; unit testing, integration testing, system testing – to name a few. We will discuss in depth about various types of testing.

At times, testing may become a tedious process if same set of activities are involved for different test cases. We will see how to automate testing using various tools in such cases

As every form or type of testing is a planned activity, we will see how do we plan for various testing activities and use different strategies to determine what type or form of testing suits what kind of scenarios.

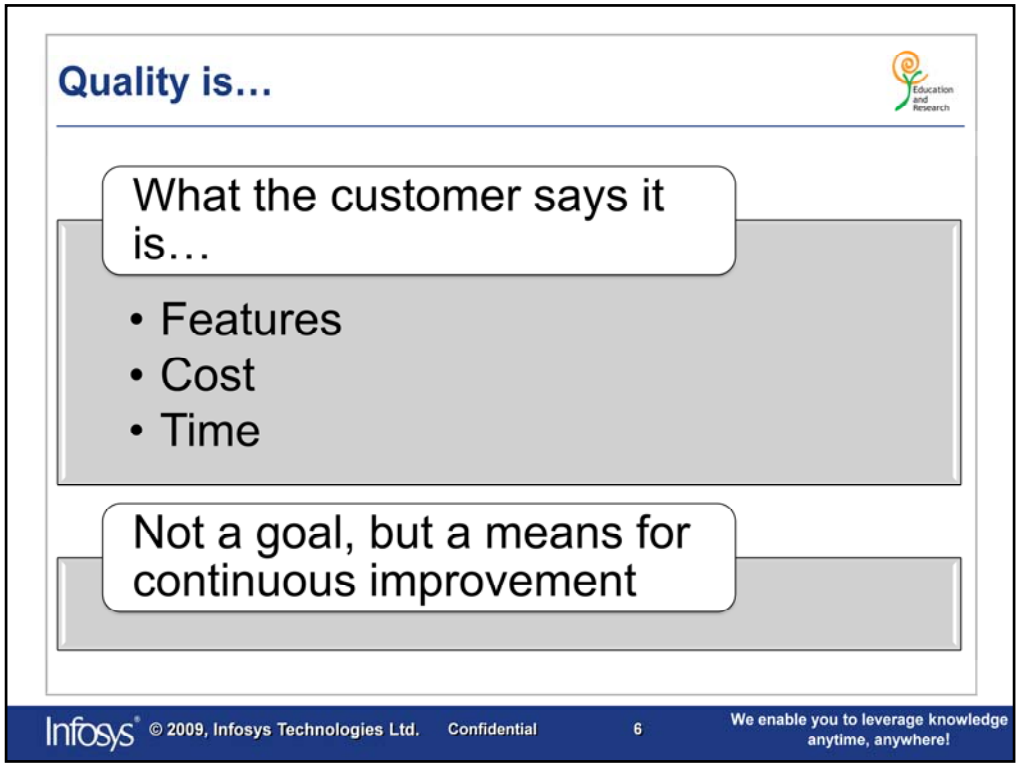
References

- Boris Beizer, *Software Testing Techniques*, International Thomson Computer Press, 1990
- Glenford J. Myers, *The art of software testing,* Wiley-Interscience, 1995
- Edward Kit, *Software Testing In The Real World*, Addison-Wesley
- Bill Hetzel, *The Complete Guide to Software Testing*, Wiley –QED
- Brian Marick, *The Craft of Software Testing*, Prentice Hall
- Mark Fewster & Dorothy Graham, *Software Test Automation,* Addison –Wesley
- *Pride*, http://172.25.103.130/Pride/Default.aspx

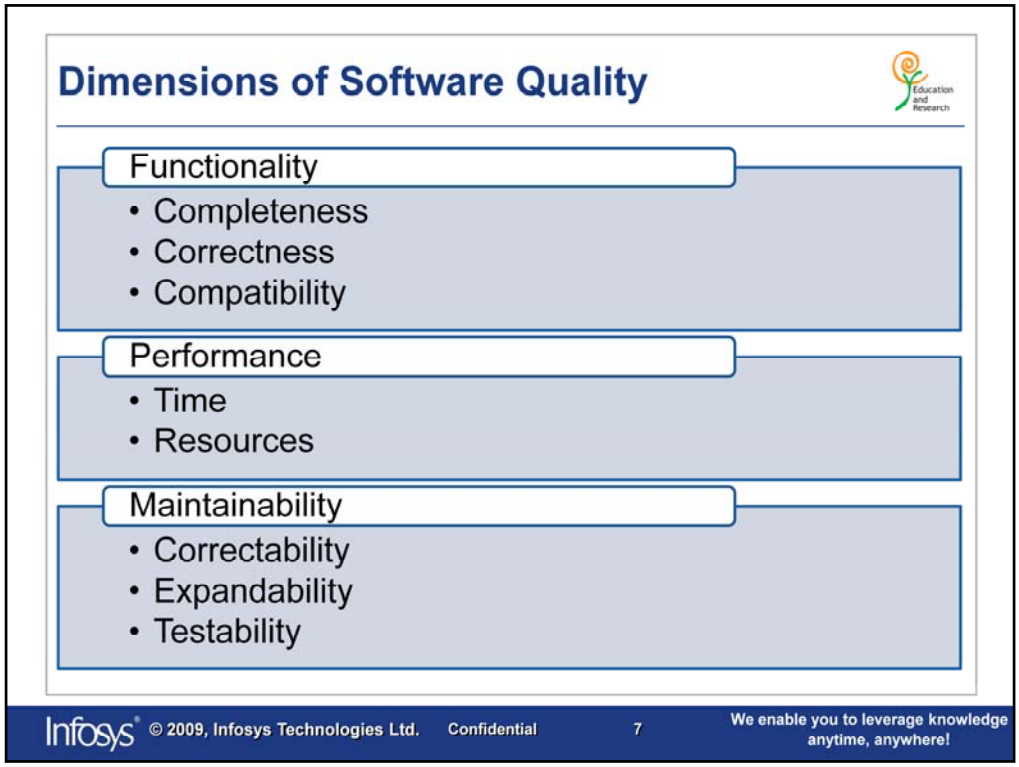


There are enough definitions for the term 'quality'. As per ISO 9000 standard "quality" is 'the totality of features or characteristic of a product or service that bear on its ability to satisfy either stated or implied needs'.

From the perspective of testing, quality is the one which separates the black from the white. Identify those 'features' (call it 'defects'!) in, the software which were not asked for by its stakeholders.



Ultimately for the customer lack of quality in a software is lack of required or specified features. If the customer doesn't get what s/he wanted, it is needless to say that the effort and cost invested in the whole exercise is futile.



If the quality is expressed more in a subjective way (as it may seem more common), it cannot be measured in any quantitative terms.

Quality has many dimensions. These are mentioned above from the software engineering view point. Here are their details:

Functionality – the core dimension which focuses on the required functionality of software

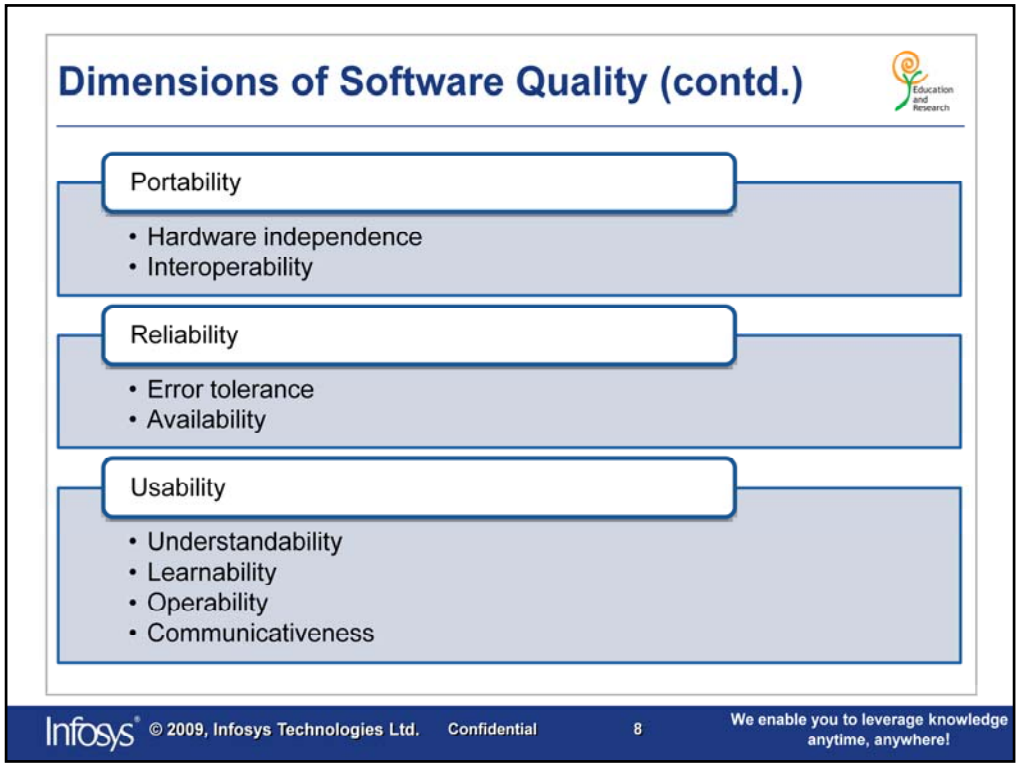
- Completeness: All requirements that are specified are implemented in the software
- Correctness: All features implemented are as per the specified requirements and are correct
- Compatibility: The software is compatible with other OS, browsers, databases, clients, hardware, system, etc. as specified

Performance – Dimension focusing on the overall performance of the system from response time, resource consumption and other perspectives

- Time: Response time for a given transaction. The speed in which the system responds to user's input, usually expressed in terms of millisec.
- Resources: Measure of utilization of resources like CPU, Memory and Disk. Which are critical resources and their parameters

Maintainability - the ease with which a software system or its component/s can be modified to correct defects (if any), improve the performance, or other attributes or adapt to a new or changed environment or add new features

- Correctability: The measure of effort required to correct the software defects and to cope with its user complaints
- Expandability: The measure of effort required to modify or improve the efficiency of a system or its component. Note that it may be very difficult to expand a software which is not modular by design.
- Testability: The measure of effort required to test the software efficiently. If the requirements are not very clear or ambiguous, it becomes very difficult to test the software and so is the case with haphazardly designed software.



Portability – measure of (effort) how easily a software can be moved from one environment to another. The effort may also involve data migration, documentation and other similar activities

- Hardware independence: The ability of a system or software to output the same set of results independent of the hardware that it is installed on.

- Interoperability: It is the ability of two or more systems or subsystems or components to exchange information and to use the information further that has been exchanged

Reliability - According to ANSI, Software Reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment. However it is not a direct function of time. It is one of the very important attributes of quality and will become harder to achieve as the complexity of the software increases.

- Error Tolerance: It is the ability of a system or component to continue normal operation despite the presence of erroneous inputs either from the user or from a different interface system

- Availability: It is the proportion of time that the system is up and running i.e. available for its intended users. It can be measure as: Uptime / (Uptime + Downtime)

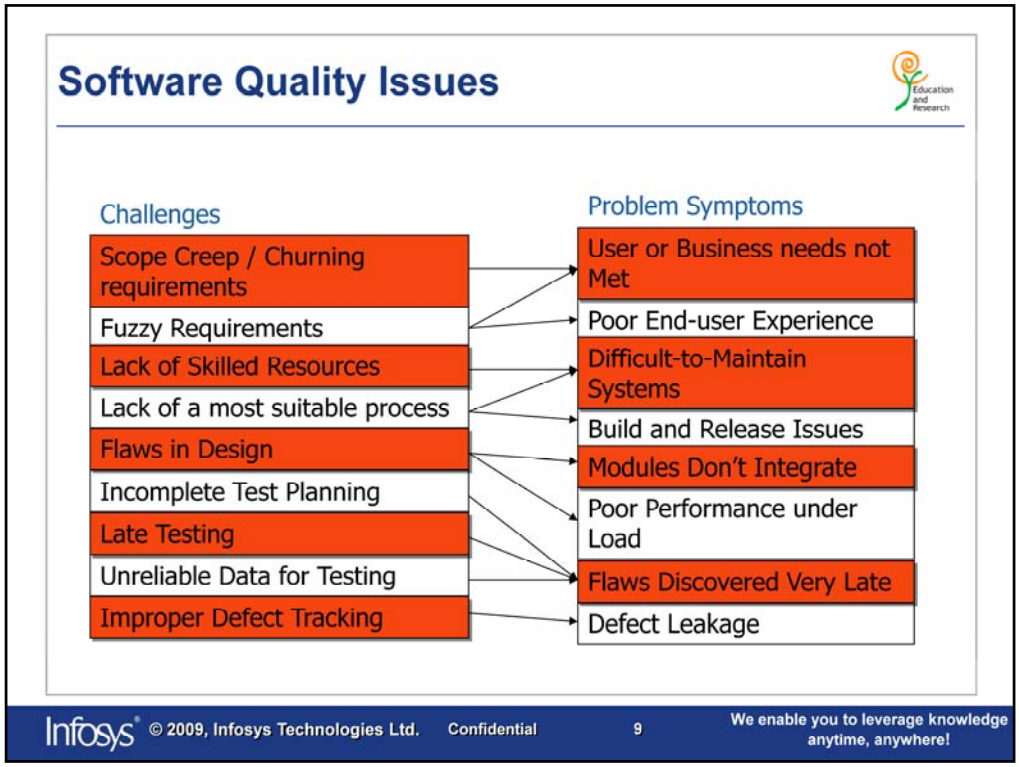
Usability – Measure of how easy or difficult it is to use a software. It refers to the efficiency, comfort, safety and satisfaction with which intended users of a software and under a variety of conditions can perform their tasks

- Understandability: The measure of effort required understand the overall functionality of a system

- Learnability: How easy is it for users to accomplish basic tasks the first time they encounter the system features

- Operability: It is the ability to keep a system in a functioning and operating condition.

- Communicativeness: The ability to express or communicate the messages or status of the system to the external world in a clear and unambiguous way



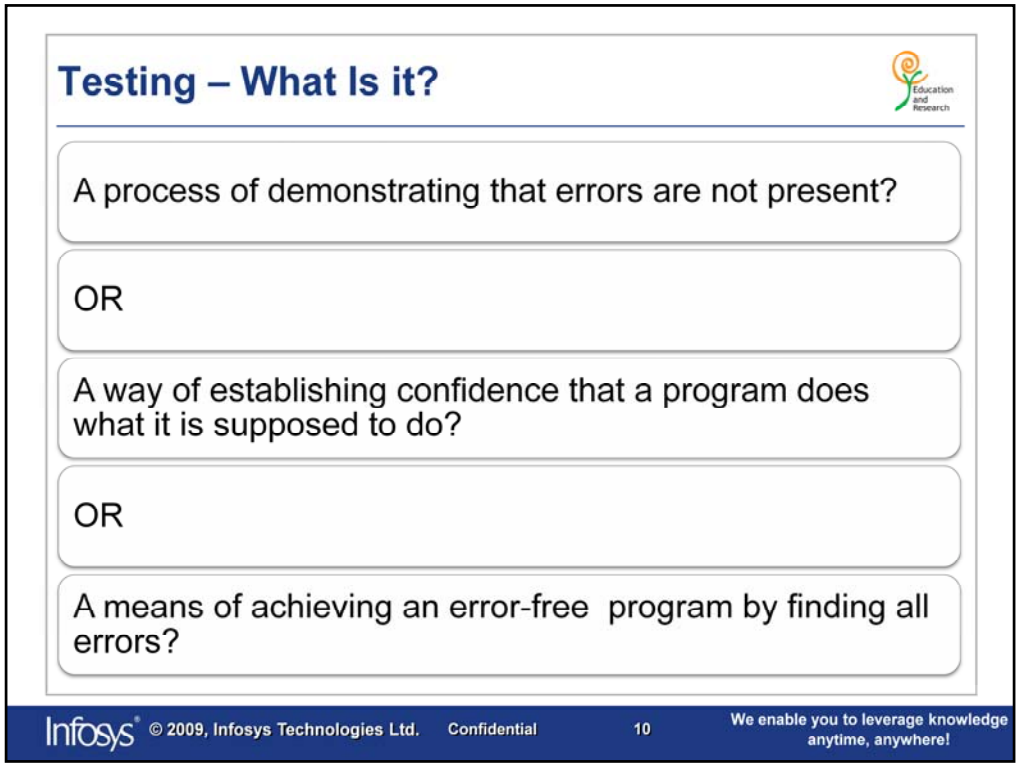
There are many challenges the project team has to face during a typical software development process. If these challenges are not addressed appropriately at the right time, it may lead to many problems down the line as can be seen in this slide above.

Slow increase in scope of the project or scope creep (after the requirements are frozen), is one of the common challenges most project teams face. Fuzzy or unclear requirements too is another challenge the team has to face where the stakeholders of the project may not be able define the requirements clearly at the beginning, especially if the project scope is very large where nobody would have a whole picture of the final system. If the system is implemented without properly taking care of these issues, ultimately it is the end users who are going to suffer having a poor experience with the system.

From the testing perspective, if the plans for testing is incomplete or if it is started late in the development cycle or even if the data used for testing is not realistic (for e.g. using values like 454, 6756.8, etc. for "person's age" field or 0.001, 435435 for "percentage" field), lead to ineffective testing and the defects getting detected very late in the cycle. Note that once the defects are detected very late, the cost of fixing them is very high.

You may recollect fact that if it takes $1 to fix a defect during requirements analysis stage, the same defect may take $100-$200 to fix it once the system is in production depending upon the complexity of the system. So it is very important to fix the defects at the early stages of project life cycle.

In addition, if the defects are not tracked correctly, there is a possibility of defects getting leaked to the next stage leading to higher cost of repair.



So, what is testing?

Is it a process of demonstrating that errors are not present? (demonstration oriented) OR

Is it a process of demonstrating that errors are present in the software (destruction oriented) OR

A way of establishing confidence that a program does what it is supposed to do? (evaluation oriented) OR

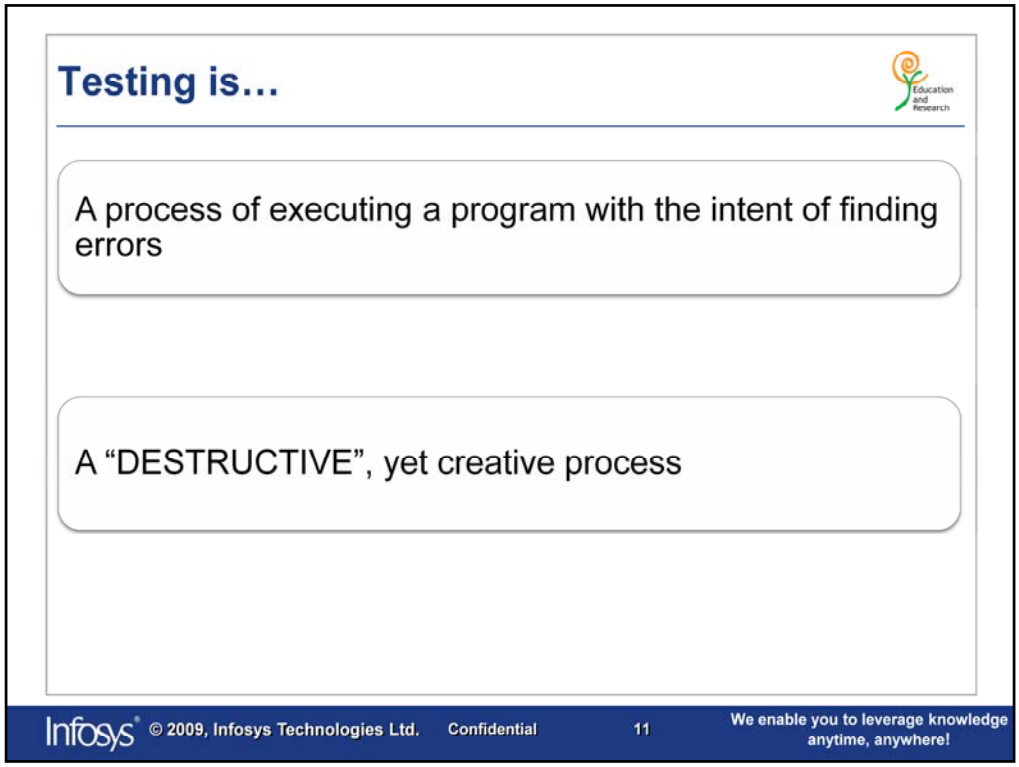
A means of achieving an error-free program by finding all errors? (prevention oriented)

Well, it is all of these. At the very beginning stage, a dignified developer would like to show that there are no errors present in the module s/he developed. While proving this, s/he would try to fix all the defects that s/he encountered trying to make the module bug-free.

Once the module is released for testing, the tester (developer's friend!?), would try to show that there are still some defects are present in the module. This would help reduce many defects towards an error-free software, in a way establishing confidence in the program that is doing what it is supposed to do.

There is a kind of attitudinal progression in the different phases above. Whatever may be the case, it is important to note that testing shows the 'presence' of bugs and not their 'absence'.

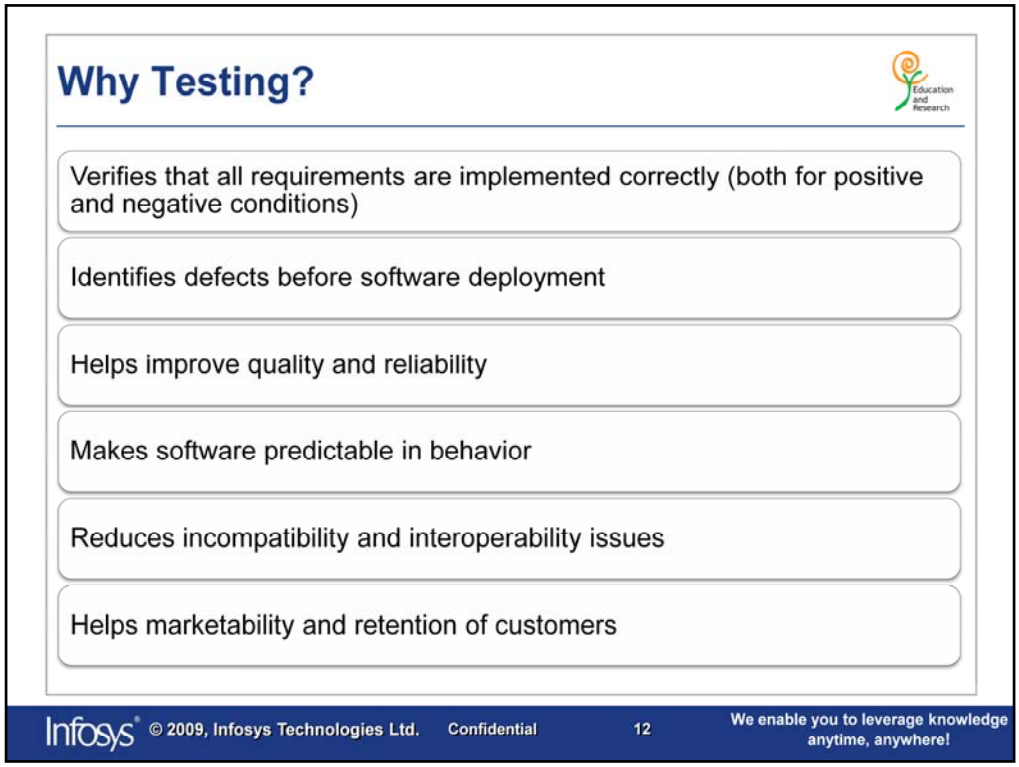
IEEE definition of testing: "Testing is the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements, or to identify differences between expected and actual results."



So, testing is a process of executing a program with an intent of finding errors. A 'destructive' (with a good intention though), yet a creative process. There are many ways that one can show his/her creativity in testing to improve its effectiveness – there are various techniques and methods which will be discussed in the subsequent slides.

It is a critical element of software quality assurance. Post-release removal of defects is very expensive. Significant portion of life cycle effort is spent on testing.

20-40% of project effort spent on testing (much more in the case of "human-rated" software). At Microsoft, Tester-to-Developer ratio is 1:1. At NASA shuttle development centre (SEI Level 5), the ratio is 7:1.



Listed above are some of the advantages of testing:

Implementing what the customer wants in a software is very important. By testing the software against the specified requirements both for proper and improper values, ensures this condition. Detecting the defects at early stages of the software life cycle helps reduce unpleasant surprises to the end users.

It's pretty evident that a good testing done on the software helps improve its quality and reliability. This makes software predictable in behaviour (Software released with 'known bugs'). An effective integration testing phase would help resolve issues related to software compatibility and operability with other systems.

The following is a list of major computer system failures caused by software bugs. The following examples highlight the kind of catastrophic consequences that software bugs have on business, on life and property.

•In April of 1999 a software bug caused the failure of a $1.2 billion military satellite launch, the costliest unmanned accident in the history of Cape Canaveral launches. The failure was the latest in a string of launch failures, triggering a complete military and industry review of U.S. space launch programs, including software integration and testing processes. Congressional oversight hearings were requested.

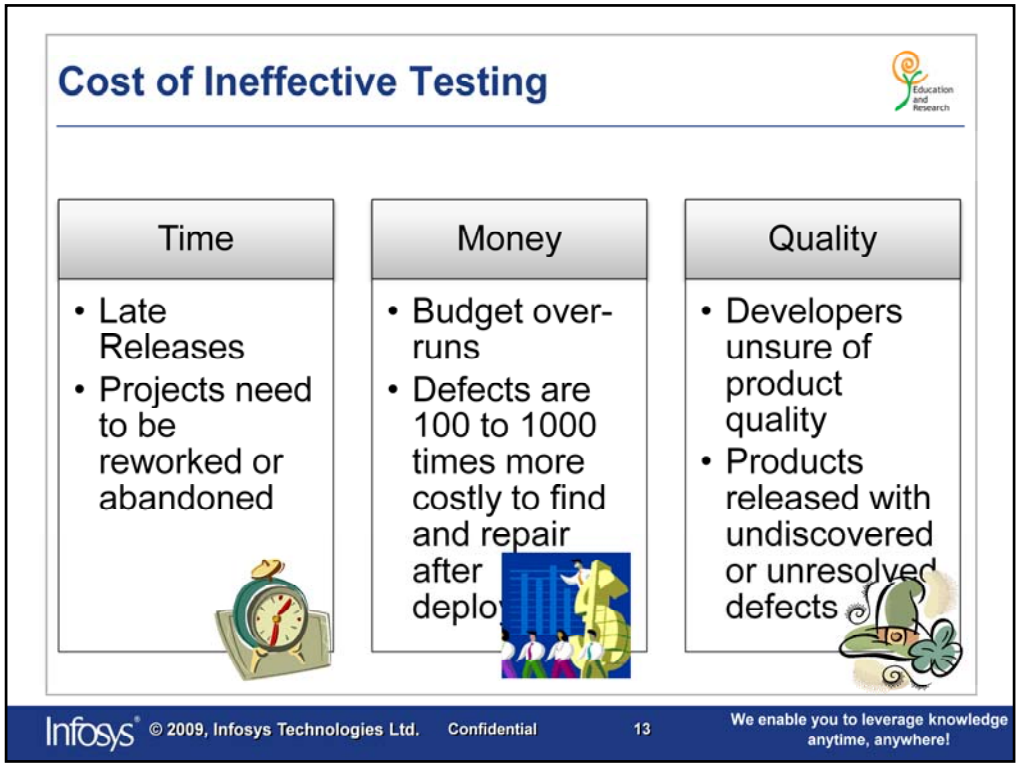
•On June 4, 1996 the first flight of the European Space Agency's new Ariane 5 rocket failed shortly after launch, resulting in an estimated uninsured loss of half a billion dollars. It was reportedly due to the lack of exception handling of a floating-point error in a conversion from a 64-bit integer to a 16-bit signed integer.

•The computer system of a major online U.S. stock trading service failed during trading hours several times over a period of days in February of 1999 according to nationwide news reports. The problem was reportedly due to bugs in a software upgrade intended to speed online trade confirmations.

•In November of 1997 the stock of a major health industry company dropped 60% due to reports of failures in computer billing systems, problems with a large database conversion, and inadequate software testing. It was reported that more than $100,000,000 in receivables had to be written off and that multi-million dollar fines were levied on the company by government agencies.

•Software bugs caused the bank accounts of 823 customers of a major U.S. bank to be credited with $924,844,208.32 each in May of 1996, according to newspaper reports. The American Bankers Association claimed it was the largest such error in banking history. A bank spokesman said the programming errors were corrected and all funds were recovered.

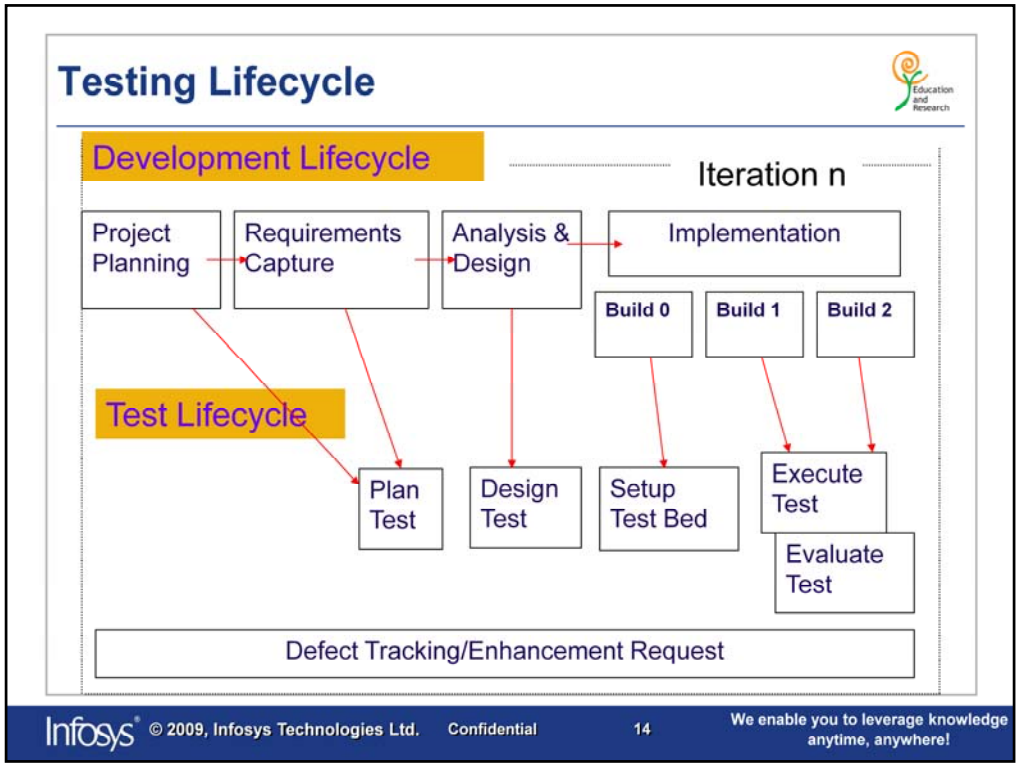
All the above incidents only reiterate the significance of thorough testing of software applications and products before they are put on production. It clearly demonstrates that cost of rectifying defect during development is much less than rectifying a defect in production.



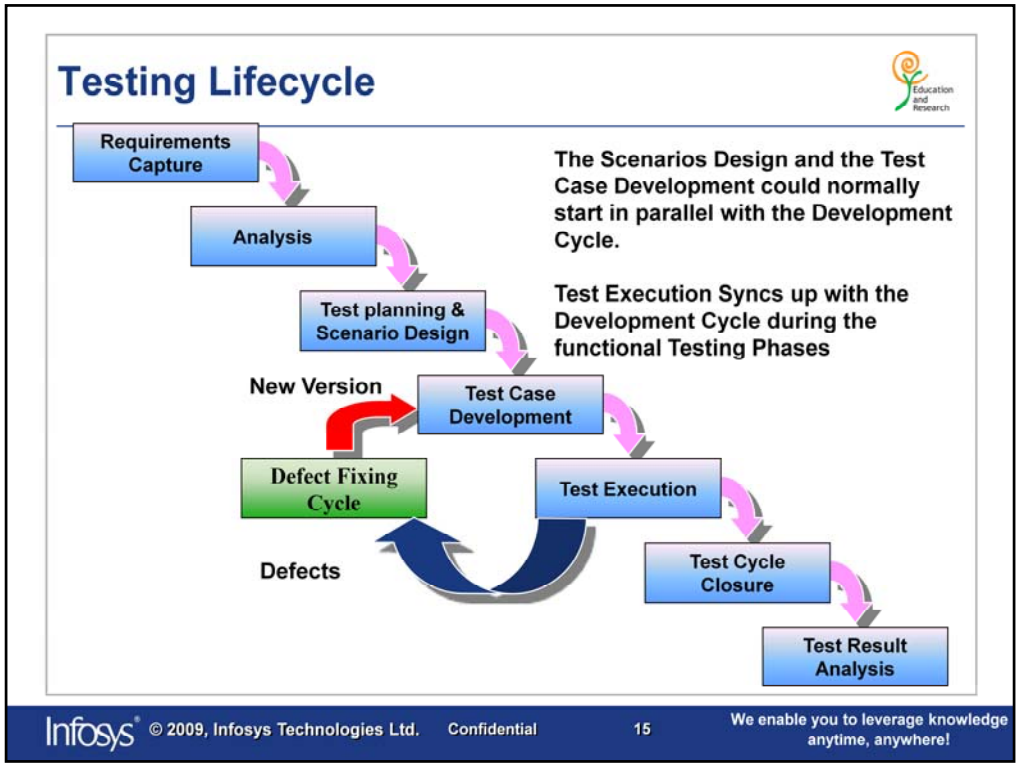
Every project can predict a no. of defects in it based on the 'defect injection rate'. (Obtained from Process Capability Baseline – PCB in Pride: http://192.168.15.77/quality/docs/pcb.htm). If the no. of defects found during the testing stages is no where near this no., one can suspect the overall effectiveness of testing (barring those cases where the software has been built to almost perfection!). The defects detected at the later stages lead to late release of the project or if the defects go undetected to the final stage, the project may have to be reworked completely or abandoned to prevent further losses.

Needless to say, above situation results in budget overruns.

Project team is not confident about the quality and so is its predictability if the testing is not proper. Only those end users would know their heart-burns using the defective software impacting their every day business severely.

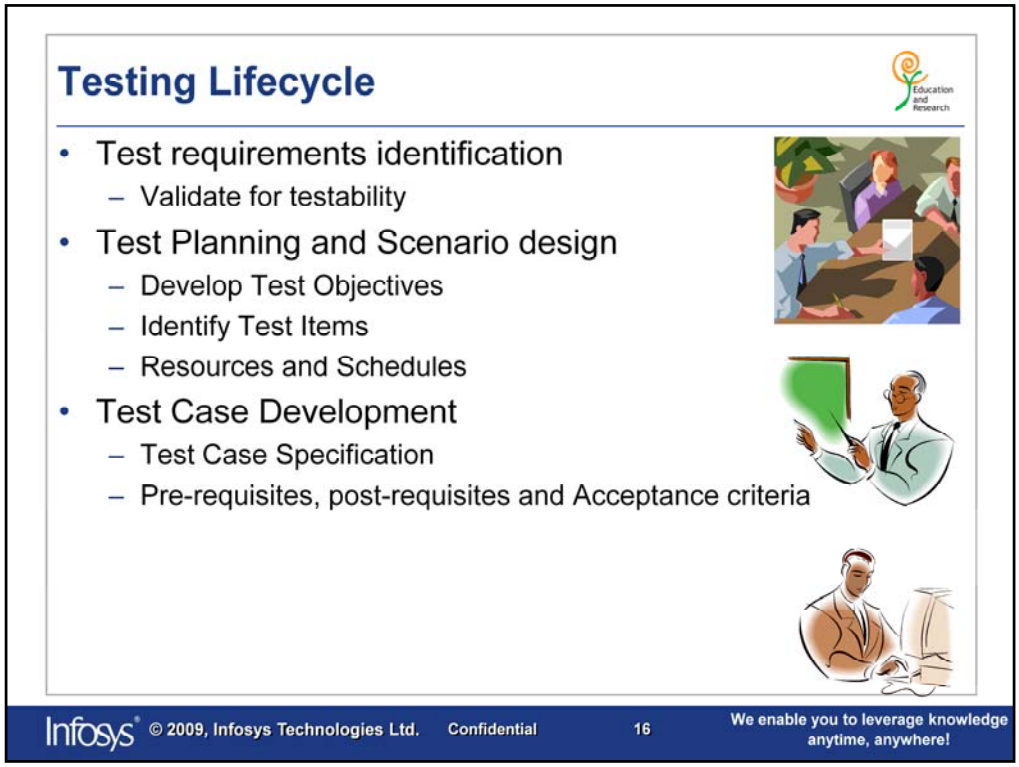


Requirements analysis, Design, Build, Testing and Implementation form a typical Software Development Life Cycle (SDLC). Testing Life Cycle (TLC), a subset of SDLC involves planning for testing, designing proper testing strategy, setting up the test bed, executing the test cases and evaluating the test results. Let's look at each of them in detail in subsequent slides.



As can be seen the diagram above, inputs for testing activities flow in right from the requirements stage. Test cases have to designed based on each use-case or business scenario. It should test for both positive and negative scenarios. More on scenario design is discussed further.

Every valid defect reported by the tester will have to be fixed suitably at the right time. This fixed defect will have to go thro' another round of testing – defect fixing cycle. So, the testing process is an iterative process. Thus the testing process is an unending process. So, when do we stop testing? It's discussed in one of the subsequent slides



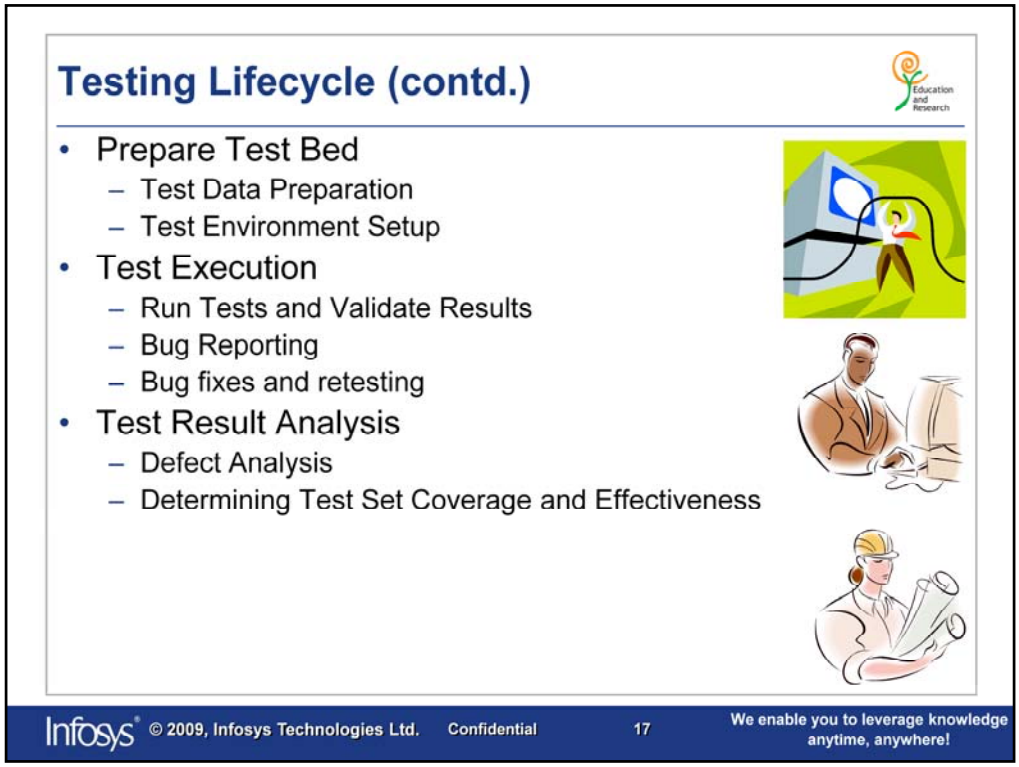
Plan Test:

This phase involves defining test scope, test environment setup, network connectivity, deciding different test phases, approaches & methodologies, planning for manual vs. automated testing, effort & person hrs estimation, defect tracking mechanism, configuration management & risk management (both from testing perspective) and evaluation & identification of testing & defect tracking tools. Note that all these cannot be done on day one of the project itself. Strategic decisions involving investments like setting up test environments which may involve setting up separate test beds / servers, hardware & software, say purchase of desktops, testing tools, etc. will have to be decided in the early stages.

Requirements for testing are identified based on the software requirements specification, functional spec, design spec and use case documents as appropriate.

Design Test:

Designing for test involves activities like scoping out test coverage (one may use a traceability matrix for this), identifying different testing scenarios & appropriate test cases, preparation of test scripts and test data and base-lining these as per configuration management requirements. Test scripts will be required in case if any testing tools are used. In most cases we may have luxury of copying existing production data to the test bed for testing. But there are cases where production data may not be available due to security and other reasons, where project team has to prepare the test data. If this test data to be generated is huge, decision has to be taken whether to purchase a tool for it or develop it in-house.

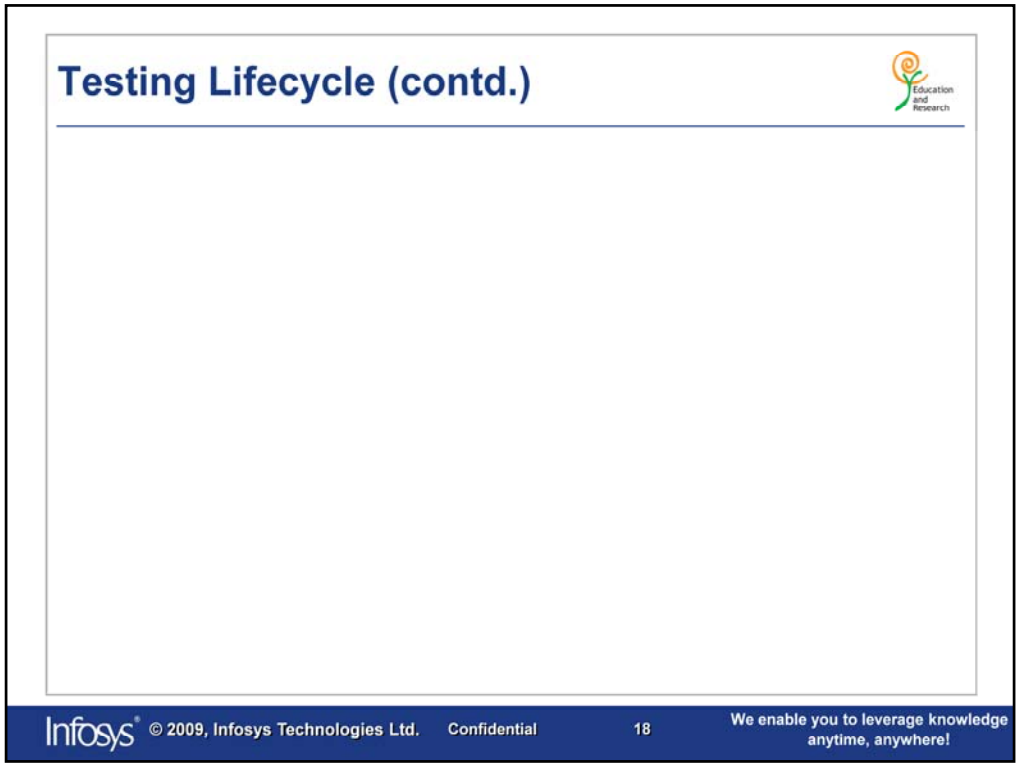


Setup / Prepare Test Bed:

Test bed installation and configuration is one of the key activities in it. A place to copy the code, executables, other installables, etc. need to be decided at this stage. Setting up a database (Oracle, DB2, SQL Server, etc.), creating tables, inserting data and defining access controls are other major activities in this phase. Configuring the network connectivity too will take a considerable effort if testing involves multiple locations and teams.

Execute Test:

The stage where actual testing is done by the testing team by executing the test cases and/or test scripts either manually or using the testing tools as guided by the test plans. Testing team tests the code / software as per these test plans and logs the defect in a defect tracking system (RADAR in Infy – refer to KShop: http://kshop/kshop/default.asp for more information & guidelines on RADAR usage).



Evaluate Test:

The defects reported by the testing team need to be reviewed and analyzed for their validity. Not all of them may be defects. Attributes of some of the defects like severity, defect type, root cause, injected stage, etc. might have to be changed. Both testing team and project team need to discuss and resolve if there are any conflicts or mismatches.

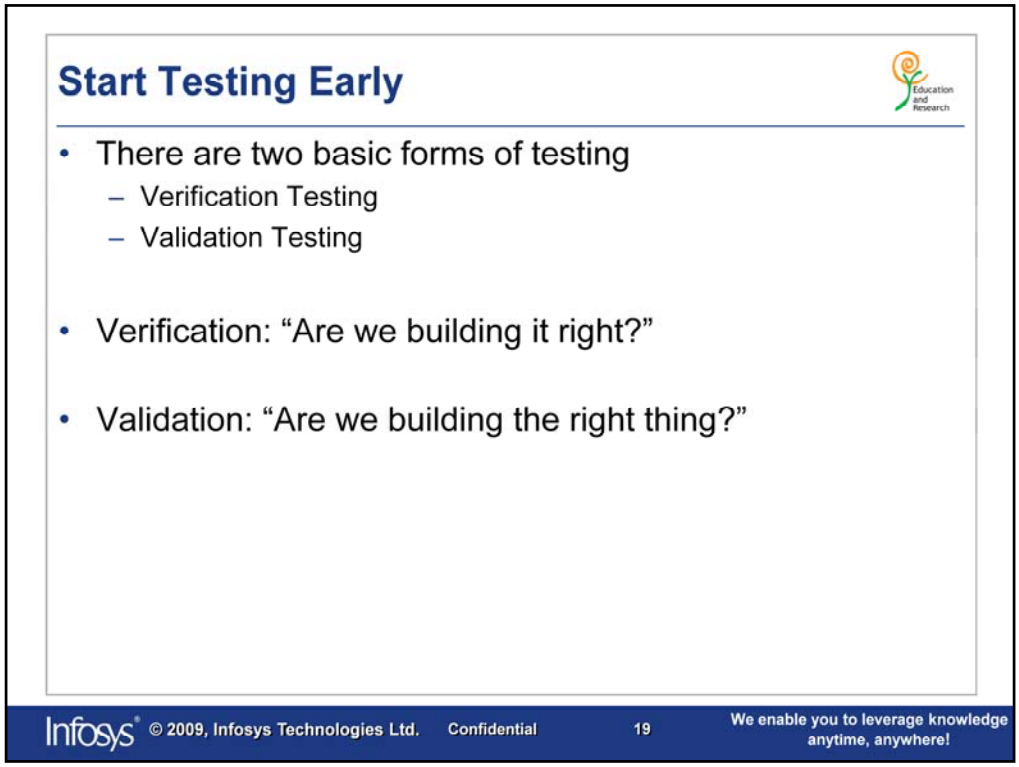
Once all these issues are resolved, the development team has to fix it as per priority, severity and other deciding attributes. Note that this process may involve redoing the whole activity from requirements to high-level design to low-level design based on the severity of the defects. Another round of testing will have to follow after this fixing exercise. So, the testing process is an iterative process. Once the defects are fixed, the testing has to be done repeatedly. Thus the testing process is an unending process. So, when do we stop testing? It's discussed in one of the subsequent slides.

A similar exercise will have to be done during a bug-fix or enhancement request in a typical maintenance project.

Once all the defects are fixed and verified as per the plan, the testing team makes necessary updates in the defect tracking system. All necessary final test summary reports to be submitted to higher-ups can be submitted at this time. Collection of test metrics for further process improvements can also be done at this time.

Test Result Analysis

An important activity at the end of testing cycle is 'test result analysis' or what we call as 'DP (Defect Prevention) exercise' – where all the defects logged are analyzed (root cause analysis) based on their types, injected stage, test effectiveness, defect injection rate and so on and so forth based on project's requirements. Findings of this analysis will be used for further defect prevention in future projects. This exercise also tells about effectiveness of testing and its coverage.

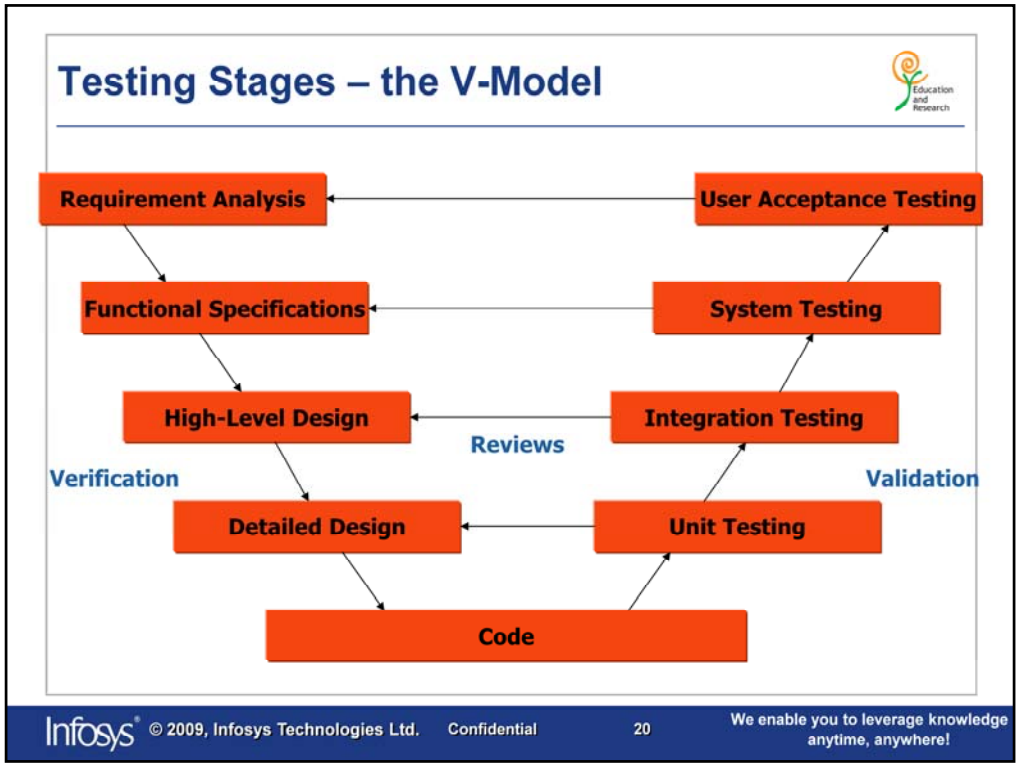


Coming to the forms of testing, there two basic forms: Verification and Validation.

Verification – ensures that the final product satisfies or matches the original intention — i.e., you built the product right. This is done through static testing or review. (Human-based). Verification typically involves reviews and meetings to evaluate documents, plans, code, requirements, and specifications. This can be done with checklists, issues lists, walkthroughs, and inspection meetings.

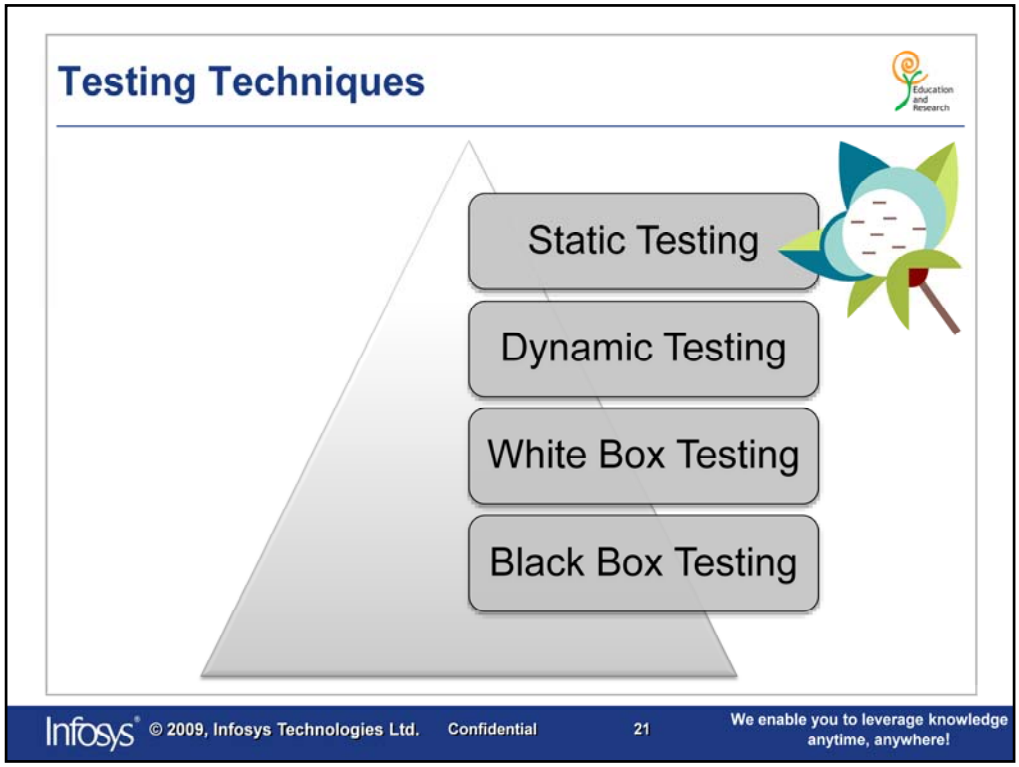
Validation - checks that the software satisfies or fits to the original requirements — i.e., you built the right product. This is done through dynamic testing and other forms of review. (Computer-based)

Software development has historically been concerned more with code than with requirements or design, hence the emphasis on validation in testing.



The verification and validation forms are further emphasized in the diagram above. This is the famous V-mode of software testing. It can be construed as extension of Waterfall model wherein, the steps after 'coding' are shown upwards against the corresponding 'verification phase' against which they are 'validated'. It demonstrates the relationships between each phase of the development life cycle and its associated phase of testing.

So, on the left hand side you have 'verification' phases and on the right side, you have 'validation' phases clearly showing the relationships among them.



Did you know that 'testing' does not always involve 'execution of code / programs'? Testing can also be done without executing even a single line of code! That is by means of 'reviews' – requirements spec review, design doc review, functional spec review and so on. This also is a form of testing, which is called 'static testing'. The one which involves execution of code is typically done in 'dynamic testing'.

White box testing is done based on the structure of the code – the conditions, loops, branches, etc. in it.

Black box testing focuses on the business functionality of the module – expect the output based on a set of inputs; not bothered about the code structure.

There are various testing techniques one needs to use to uncover various kinds of defects as there is no 'one-size-fits-all' kind of a testing technique which will detect all defects. This situation can be compared to a farmer using different types of pesticides or insecticides to kill the bugs (pests or insects) in his farm, which is explained in more aptly called 'Pesticide Paradox':

Farmers use various pesticides or insecticides of different compounds to kill different kind of bugs. For e.g. if a farmer uses DDT this year to kill a particular type of pests, not all of them may get killed this year. Say, if 98% of them get killed and 2% remain and become immune to DDT. Next year, the farmer has to use a different pesticide to kill those remaining 2% (which would have grown by no. now) of bugs which are immune to DDT along with DDT. Again, it may not kill all of those bugs leading the farmer to use a one more different pesticide and the cycle continues. This is called Pesticide Paradox**

Similarly, every testing technique that you use to prevent or detect the defects in a software leaves a residue of subtler defects against which those techniques are not effective. To top it, software becomes more complex by addition of new features and interfaces release after release making it more difficult to identify the defects.

** The Pesticide Paradox seemed to have been resolved (especially for cotton crop) by surrounding the cotton field with a sacrificial crop that a particular type of pests prefer to cotton. Unfortunately, we cannot create a 'sacrificial module' in a software that would attract all the bugs from the main ones!

Testing Techniques (contd.)

- Static Testing/Reviews
  - Scope
    - Requirements Feasibility
    - Technical Architecture
    - Design
    - Program specifications
    - Test Plans
    - Test Cases and Data
    - User Documents

Infosys © 2009, Infosys Technologies Ltd. Confidential We enable you to leverage knowledge anytime, anywhere!

In Static Testing, The Design, code or any document may be inspected and reviewed against a stated requirement and for some standards against some guideline/checklist. Static Testing includes:

•Walk-through - code reading and inspections with a team

•Reviews - Formal techniques for specification and design, code reading and inspections.

Many studies show that the single most cost-effective defect reduction process is the classic structural test - the code inspection or walk-through. Code inspection is like proof-reading - it can find the mistakes the author missed - the "typo's" and logic errors that even the best programmers can produce

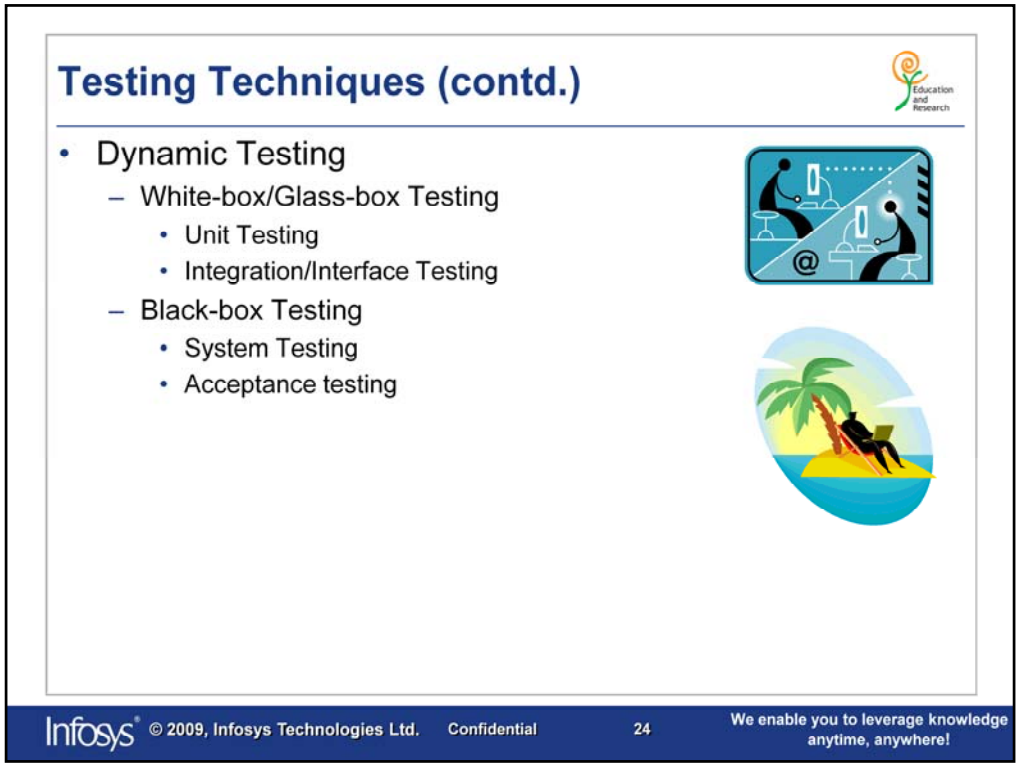
Testing Techniques (contd.)

- Static Testing
  - Advantages
    - Capture defects early, so saves rework cost
    - Checklist-based approach
    - Focuses on coverage
    - Group perspective
    - Highest probability of finding defects
  - Disadvantages
    - Time-consuming
    - Cannot test data dependencies
    - High skill levels required

Infosys © 2009, Infosys Technologies Ltd. Confidential We enable you to leverage knowledge anytime, anywhere!

Advantages and disadvantages of static testing are listed above. The effectiveness of static testing if done thoroughly and meticulously can be much more than that of dynamic testing. The reason being, it doesn't need many test cases to execute different paths or conditions as in dynamic testing. A scrupulous expert review would be sufficient enough to uncover many defects. Moreover, if it's done at the early stages, many defects can be avoided leaking to next stages can be avoided.

Disadvantages: If a thorough inspection of every item in scope it may be time-consuming at times. Also, if there is data dependency (A related B which is related to C & D), this method's effectiveness may go down. Moreover, its effectiveness is dependent upon the experience level of the reviewer.



Dynamic testing involves execution of the program or code. Primarily, a program or a set of programs (say module) can be tested in two ways – white box or glass box way of testing and the other, black-box testing.

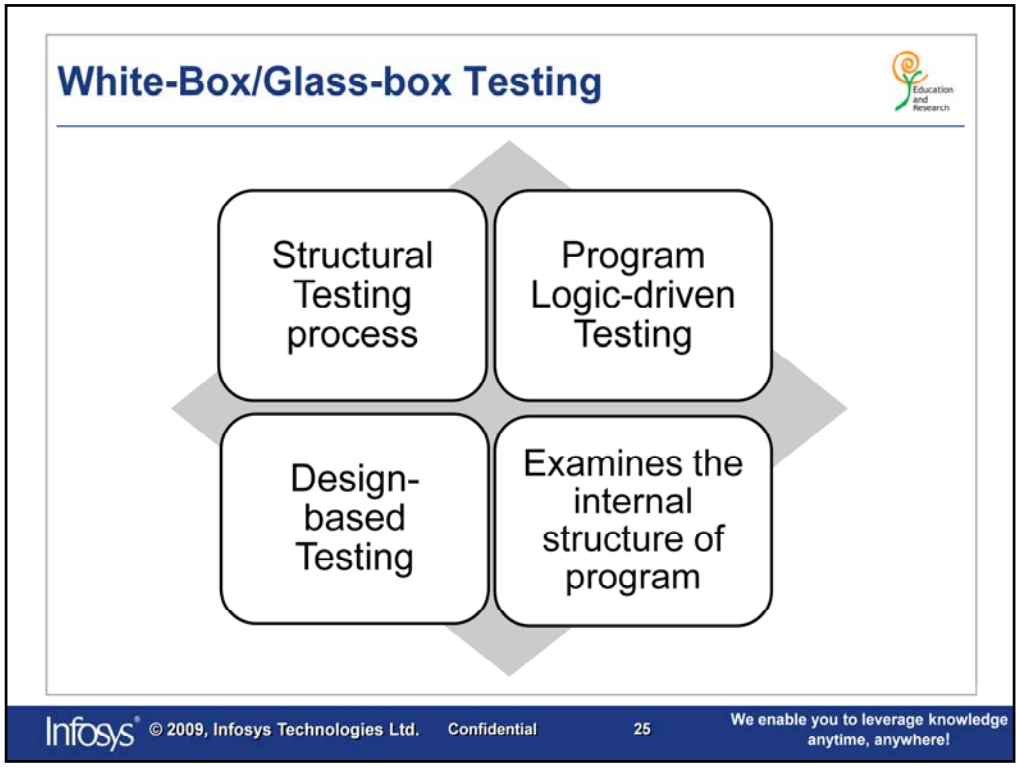
White or Glass box testing as the name suggests, is the testing with an intention of keeping the program structure in view. Primarily testing the program for its various conditions, loops, paths, etc.). Or if the two programs are getting integrated (one routine calling other, one system interfacing with other), testing the integration point with all parameters in view.

Typically, initial phases of testing - unit testing and integration testing (for smaller modules / interfaces) are planned to be executed this way.

Black-box testing on the other hand, doesn't bother about the implementation details of the program. It treats the program or software as a black-box, focusing on the 'functionality' of the system. For a given set of inputs what's the output expected in conformance with the inputs. This information can be obtained from the requirements or functional specs document.

Typically, higher levels of testing like system and acceptance testing are done in this way. Reason being, going for while-box testing at this stage i.e. designing test cases with the structure / flow of the code in mind, which will have several programs, will be very tedious and highly time-consuming.

Planning for both white-box as well as black-box is very important while strategizing for testing techniques.



White-box testing or behavioral testing or structural testing looks at implementation details of the program. Things as programming style or logic, control method, language used, database design and coding details are the key factors in structural testing. Program logic are decided during the low-level design phase. So, the test cases in white-box testing are based on the design of the programs – more of a designer's or programmer's perspective.

Let's see an example where using a white-box testing method would tend to identify the defect more easily than a black-box testing method:

```
function FACTORIAL (X)
if  X=1 then
                    FACTORIAL : = 1
else
                    FACTORIAL : = X *  FACTORIAL (X-1)
return
```

Suppose a programmer stores Factorials of all these no.s in an array and returns the Factorial of an input no. using this array – the program would work fine for these no.s. Whereas it would fail for the no.s which are not in this range.

```
function FACTORIAL (X)
data                FACTORIAL _ TABLE
                    /  1,  2,  6,  24,  120, ...   /
FACTORIAL : =  FACTORIAL _ TABLE (X)
return
```

In such cases, a white-box testing method would easily find out a potential bug in it whereas there is a possibility of missing it in a black-box testing method.

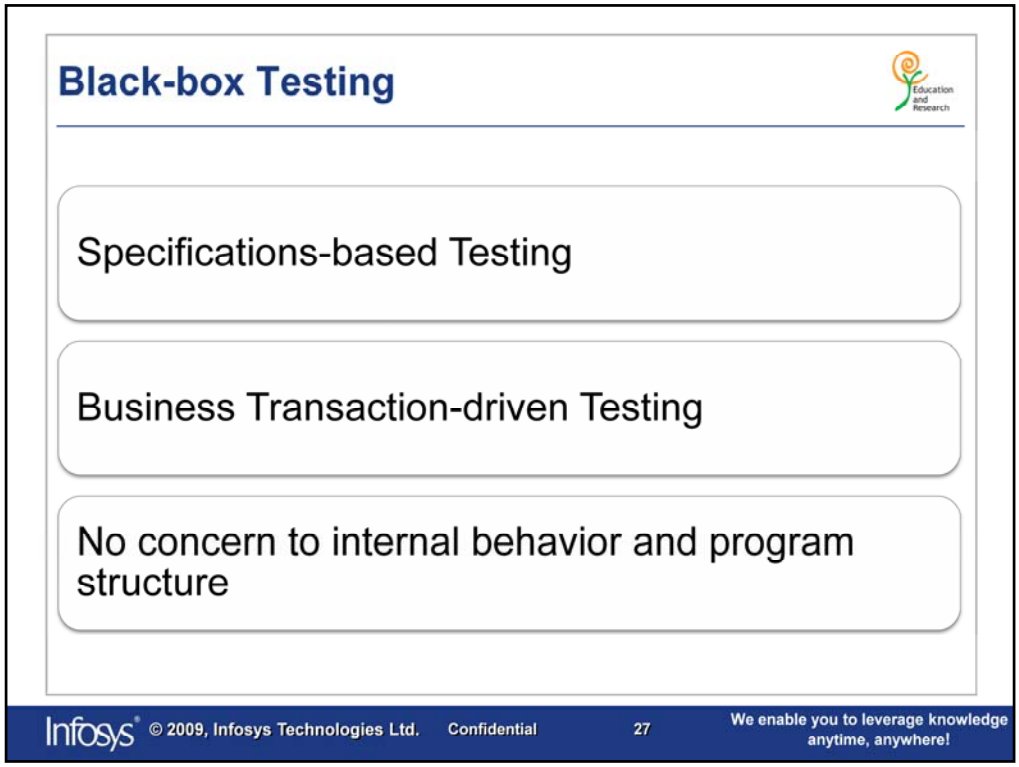
White-Box Testing – Advantages

- High Code coverage – Exhaustive (thorough) path testing
- Program logic is tested
- Internal boundaries are tested
- Testing is algorithm specific

Infosys © 2009, Infosys Technologies Ltd. Confidential 26 We enable you to leverage knowledge anytime, anywhere!

Since the white-box testing looks at the code structure while designing the test cases, code coverage it is very high. Note that main objective of white-box testing is to attain highest level of code coverage. Needless to say all the program logic and its internal program boundaries are tested.

Traditionally, this kind of testing has been done using interactive debuggers, or by actually changing the source code. Now, there are a number of testing tools that let you perform white-box testing on executables, without modifying the source and without incurring the high overhead of an interactive debugger. These tools speed up testing and debugging because there is no need to wait for test support code to be inserted into the program. Unit testing and some (initial) part of integration testing come under white box testing category.



Black box testing is derived from functional design specifications, without regard to the internal program structure. This test is done without any internal knowledge of system / product being tested. Functional tests examine the observable behavior of software as evidenced by its outputs, without any reference to internal functions. This is the essence of 'black box' testing.

•Black box tests better attack the quality target. It is an advantage to create the quality criteria from this point of view from the beginning.

•In black box testing, software is exercised over a full range of inputs and the outputs are observed for correctness. How those outputs are achieved, or what is inside the box are immaterial.

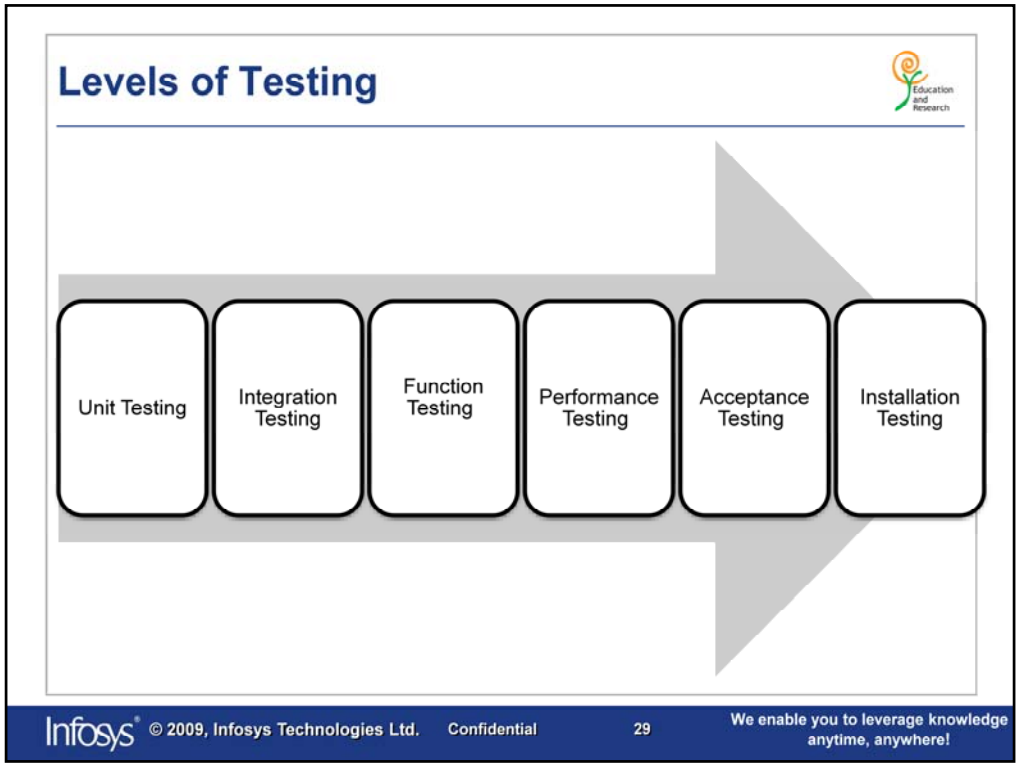
•Black Box testing technique can be applied once unit and integration testing is completed i.e. each line code has been covered through white-box testing.

Black-box Testing – Advantages

- User's perspective
- Focus on features, not implementation
- Big-picture approach

Infosys © 2009, Infosys Technologies Ltd. Confidential 28 We enable you to leverage knowledge anytime, anywhere!

Advantages: More of end-user's perspective so, not concerned about internal structure of programs. That is, focus is on the features than the implementation. It's usually done at the later stages of testing (after the initial set of testing, typically using white-box methods). So, it takes a big picture approach.



Coming to various levels of testing, we have:

Unit Testing – carried out to test the programs / small modules

Integration Testing – executed to test the integration points (between two modules, sub-systems or larger systems)

Function Testing – Focusing on the functionality of the overall system

Performance Testing – To test the performance of the system keeping in mind various performance parameters

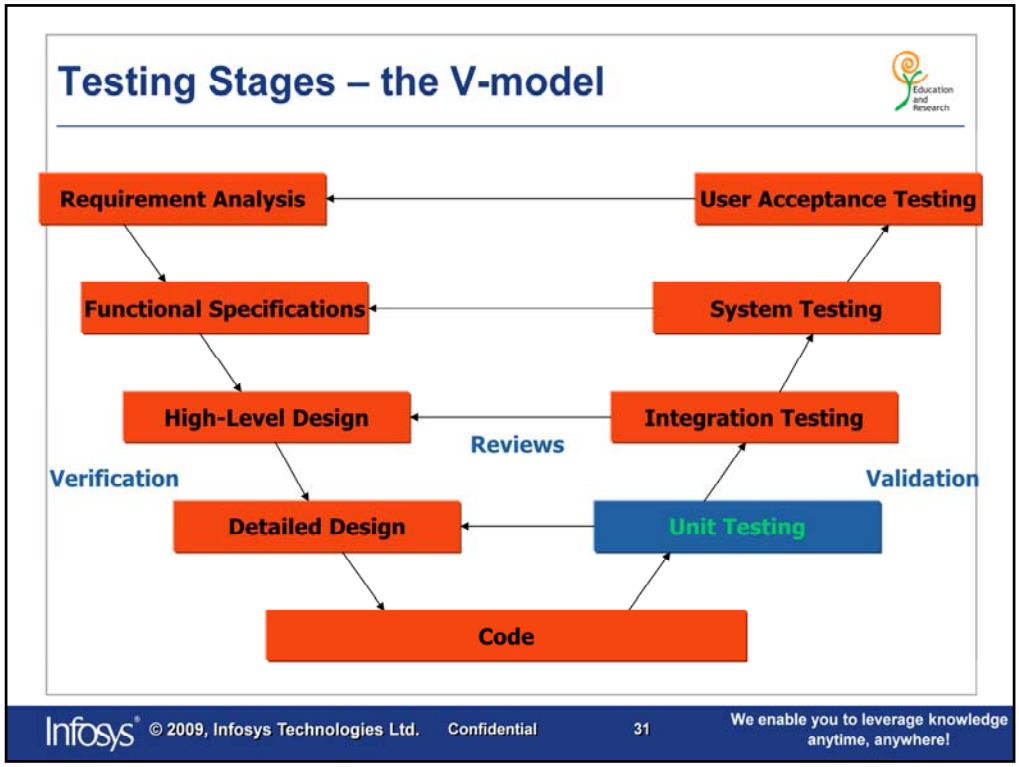
Acceptance Testing – Done by the end user / customer before accepting the software

Installation Testing – to check if the software installs successfully at the client site

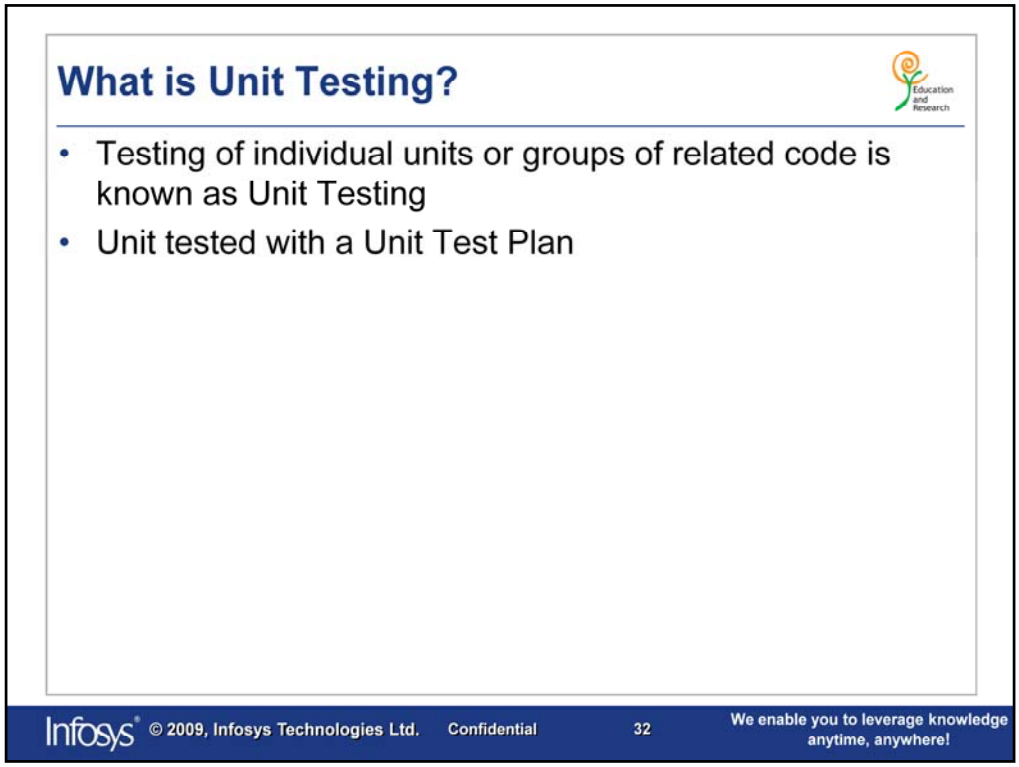
Note that last 4 levels of testing are nothing but different forms of 'system testing'.

Let's look at each one of them in detail in the subsequent slides …

Let's discuss about 'unit testing' and various techniques used to improve its effectiveness in detail.



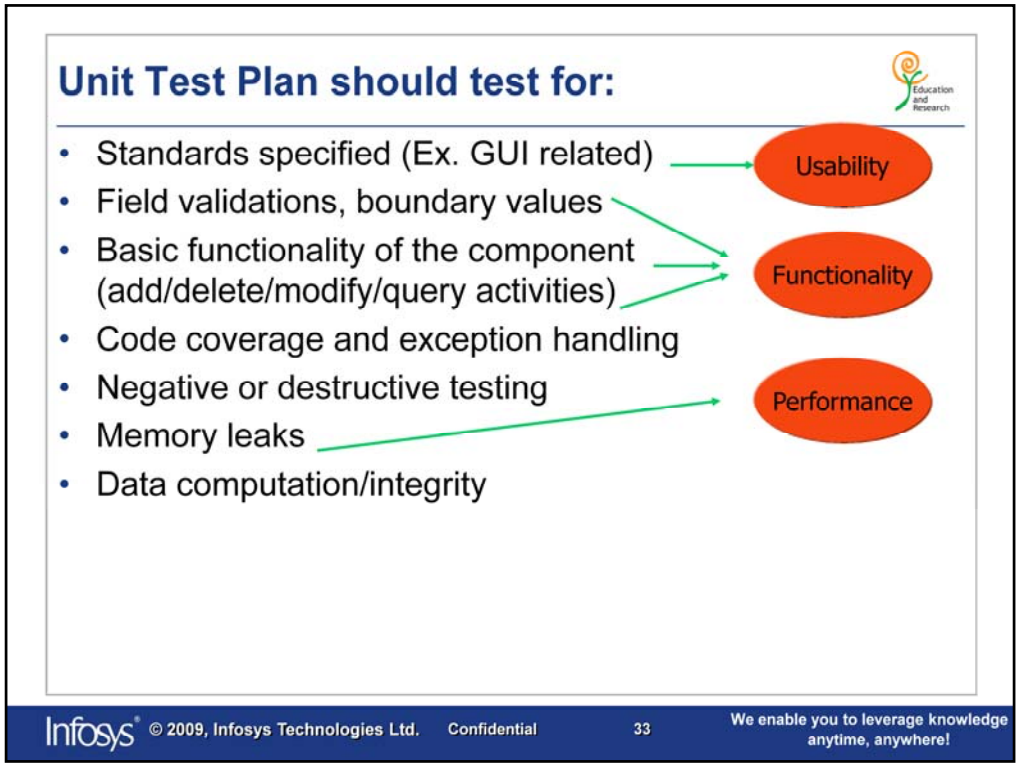
As can be seen, unit testing is based on or 'validated' against the detailed design in the famous V-model of testing. It is the first formal testing stage after coding. Though the developer self-tests his/her own code while developing it, it's mostly considered as part of coding only.



Unit testing is testing of smallest testable part of the software. It can be an individual program or a group of programs (forming a unit). The definition of 'unit' can be left each project team. In some cases one single program can be considered as a unit or in some other cases, a set of related programs can be considered as a unit. This discretion can be left to individual project teams. In some cases it is also referred as component testing.

It is very important that unit testing needs to be carried out with a help of a unit test plan. Note that this unit test plan should have been prepared during the detail design stage and not after coding! Why?**

** This is because, if you prepare a unit test plan based on your code, any defect in the code will creep into the unit test plan too defeating the very purpose of unit testing.



Unit testing is at the lowest level of testing and is done at the initial stages of testing. It should test for:

Standards specified – GUI standards for screen elements, messages, reports, etc.

Field validations, boundary values – date, numeric fields and test for their extreme values

Basic functionality of the component (add/delete/modify/query activities) - typical activities to ensure data is getting saved properly

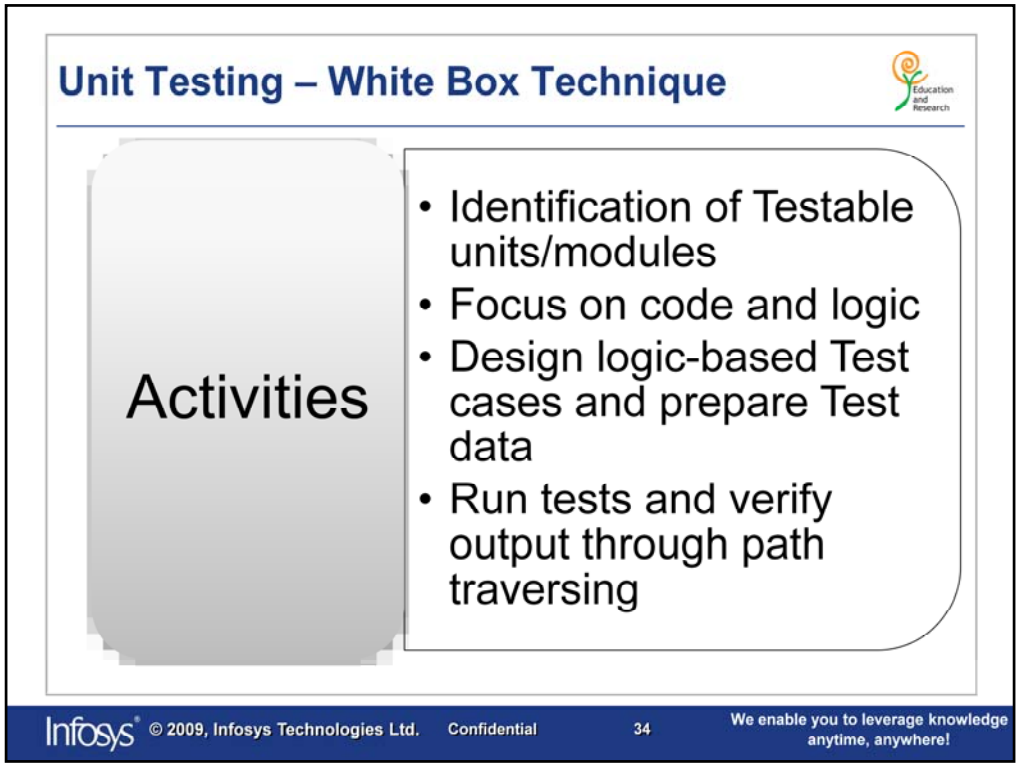
Code coverage and exception handling – various loops, conditions, branches and paths of code. Test to ensure exceptions are handled correctly

Negative or destructive testing – check for negative values, invalid values

Memory leaks – check if there is any memory leaks when the program is running (may have to use some tools)

Data computation/integrity – check for formula, computations and integrity of data

As you can note in the slide above, we are testing for various quality dimensions by testing these various items.



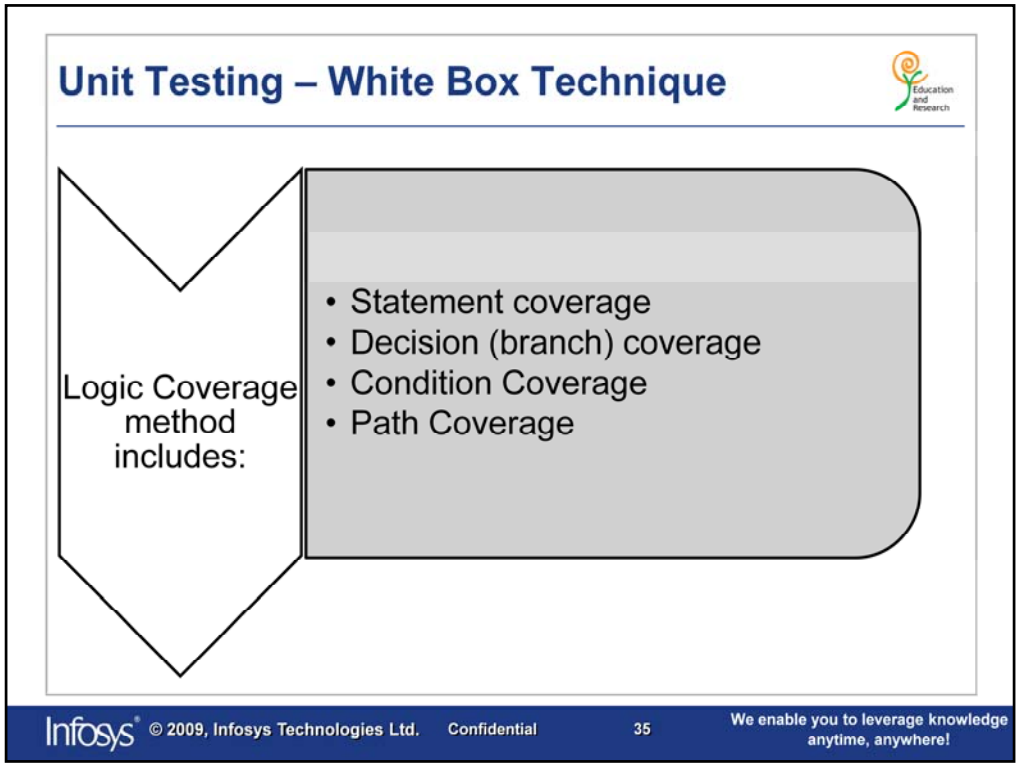
Activities involved in unit testing:

-Identification of testable units/modules: A program or a set of programs which are intended to do a unit of work or a transaction are the likely candidate to become the unit. It makes the programmer to check whether this piece/s of code works properly.

-Focus on code and logic: Being at a lowest level of testing, unit testing should focus on the code and its logic. More on this is dealt in code coverage in subsequent slides

-Design logic-based test cases and prepare test data: Designing test cases based on logic is an important activity. For example, for a simple 'if A > B' condition, minimum 2 test cases have to be created (one with A > B and the other one with A < B) to check the functioning of this 'if' construct. To add to it, boundary value and equivalence partitioning methods will call for more test cases.

-Run tests and verify output through path traversing: The program flow or path have to verified as per the test cases designed based on various conditions, branches, loops, etc.



A very important part of unit testing – logic coverage. As it's the lowest level of testing, in no other level of testing you'd have chance to test the program logic. There are various techniques to maximize the code coverage.

- Statement coverage: Test to ensure 'has each line of code been executed?'. Lines which are not tested may have potential bugs in them or there may be unreachable code (say, inside a condition which is always false like "if a=0" in C language)

- Decision (branch) coverage: Test to ensure 'has each branch of the code as a result of true or false condition been tested?'. For e.g. if ((A > B) or (B > C)) – checking for true or false output of this condition as a whole

- Condition Coverage: Test to ensure 'has each evaluation point which results in either true or false condition been tested?'. For e.g. if ((A > B) or (B > C)) – checking for true or false output for each condition inside the parentheses.

- Path Coverage: Test to ensure 'has every possible route in the given program been tested?'. If the program has multiple conditions and branches, test to ensure each path or route across all branches are tested

Unit Testing – White Box Technique (contd ...)

- Logic Coverage method includes:
  - See Notes Page Below
- Identify how many test cases are required to test following code using different logic coverage methods:

```
IF CODE IS BLANK OR NOT IN DATABASE
  DISABLE "ACCOUNT HISTORY"
ELSE
  IF NO CREDIT AND AMOUNT<100
   DISABLE "ACCOUNT HISTORY"
  ELSE
   ENABLE "ACCOUNT HISTORY"
  ENDIF
ENDIF
DISABLE "SPECIAL REGION"
IF REGION = "NORTH EAST"
   ENABLE "SPECIAL REGION"
ENDIF
```

Infosys © 2009, Infosys Technologies Ltd. Confidential 36 We enable you to leverage knowledge anytime, anywhere!

Consider following small C routine to calculate Factorial of a no. between 0 and 100 (line no.s are not part of program syntax):

```
long int MyFact (int MyNum)
{
1:          long int liFact = 1;
2:          int i = 1;
            if (MyNum = 0)      // bug: '=' used instead of '==' operator
3:                              return 1;
            if (MyNum < 0 || MyNum > 100)
            {
4:                              printif ("Pls enter no. between 0 and 100\n");
5:                              return -1;
            }
            while (i <= MyNum)
6:                              liFact = liFact * i++;
7:          return liFact;
}
```

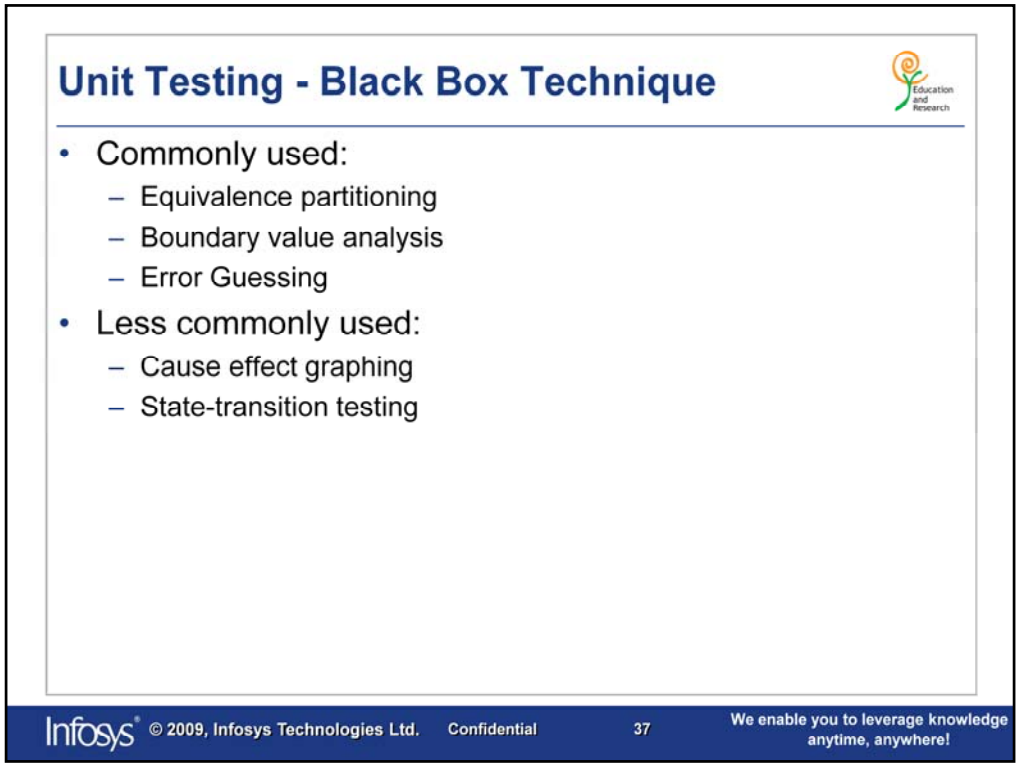
Statement coverage: Need to come up with a test case to cover all 7 executable statements. Min. 3 test cases would be required to cover them. But pls note that statement no. 3 would never be reached since if condition: if (MyNum = 0) always resolves to 'false' as 'assignment' operator is used instead of 'comparison' operator.

Decision or Branch coverage: This also would require 3 test cases to cover all decision statements

Condition coverage: It would require MyNum to have values <0, 0, 0< MyNum < 100, 100, > 100 cases to test all conditions

Path coverage: There are 3 paths in this routine. So it would at least require 3 test cases.

Optimum no. of test cases need to be created to cover all these scenarios. Test cases that cover most of the code need to be selected.



An ideal testing procedure should test the outputs for all values of all inputs. Like for example, a variable of type (signed) 'int' in C, can have values from -32768 to +32767. A string variable of size 10 bytes can have values up to 2^80 combinations. Testing for all these combinations would take billions of years even with the help of tools. Nonetheless, one wouldn't require this kind of an exhaustive testing. Such integer variables tested for 100 should work fine for value 200 too. Therefore, testing is done by dividing the input values into different partitions – valid values and invalid values.

Another technique which has higher possibility of finding defects is 'boundary value analysis'. (compare cases like A > B and A >= B, for i=0, i<=100, i++ - note that this 'for' loop runs for 101 times instead of intended 100 times). This testing is done using boundary values.

There are less commonly used techniques like Cause-effect graphing and State-transition testing.

Cause-effect graph provides a representational technique for capturing the relationship between software's inputs (causes) and its outputs (effects). It is a good way to uncover ambiguities and incompleteness in specifications.

For e.g. If a person is earning >= Rs. X total gross income per year, age >= 25 yrs, sex = male: are the inputs or 'cause'

Std. deduction = Rs. Y and the tax should be Y% on the total gross income – std. deduction: are the output or 'effect'

'State transition testing' technique which is based on finite state automation method, is a model of behaviour of the system or software composed of finite no. of states, transition between those states and action. Take for example, an elevator control system. Following can be its possible states, transition condition and transition actions for an elevator:

States: Idle / Waiting, Door open, Door closed, Faulty

Transition conditions: Button Press, Switch on, Switch off

Transition actions: Moving up, Moving Down, Open door, Close door

Based on the various features of an elevator, these parameters may vary. These parameters when put in the form of a table can be used for coming up with various test case scenarios.

Equivalence Partitioning

- Systematic process of identifying a set of input conditions to be tested
- The two distinct steps involved:
  - Identify equivalence classes
  - Identify test cases

Infosys © 2009, Infosys Technologies Ltd. Confidential 38 We enable you to leverage knowledge anytime, anywhere!

Since exhaustive testing is almost an impossible task, Equivalence Partitioning technique is used to reduce the no. of test cases to an optimum no. and to select the right kind of test cases to encompass the maximum coverage. Though in very rare cases this technique is also applied to outputs of a component, typically it is applied to the inputs of a tested component. For e.g. here are the equivalence partitions for an input of type month:

. . . -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, *13, 14, 15, . . .*

- -

Invalid partition 1 | Valid Partition (1 to 12) | *Invalid partition 2 (13 onwards)*

Similar partitions are possible for every input of the system. Identifying the equivalence classes (as above) and identifying the test cases based on these classes (both from valid and invalid) are the major steps involved in this technique. Note that, it's advisable to take values from the middle of the partitions leaving the values near the edge to 'boundary value analysis' technique.

Come up with 'equivalence partitioning' cases for following conditions:

Inputs: Age: 18 – 150, Gender = M,F, Married = 0,1

if age <25 premium+=100

if age >45 and <65 premium - =100

if female or married premium - =200

Equivalence Partitioning

Equivalence Classes – Valid Data

- User supplied command
- Responses to system prompts
- File names
- Computational data
- Physical parameters, bounding values, initialization values
- Output data formatting commands
- Responses to error messages
- Graphical data (e.g. mouse picks)

Infosys © 2009, Infosys Technologies Ltd. Confidential We enable you to leverage knowledge anytime, anywhere!

Some of the examples from various functions for equivalence partitioning – valid data is listed here.

Equivalence Partitioning

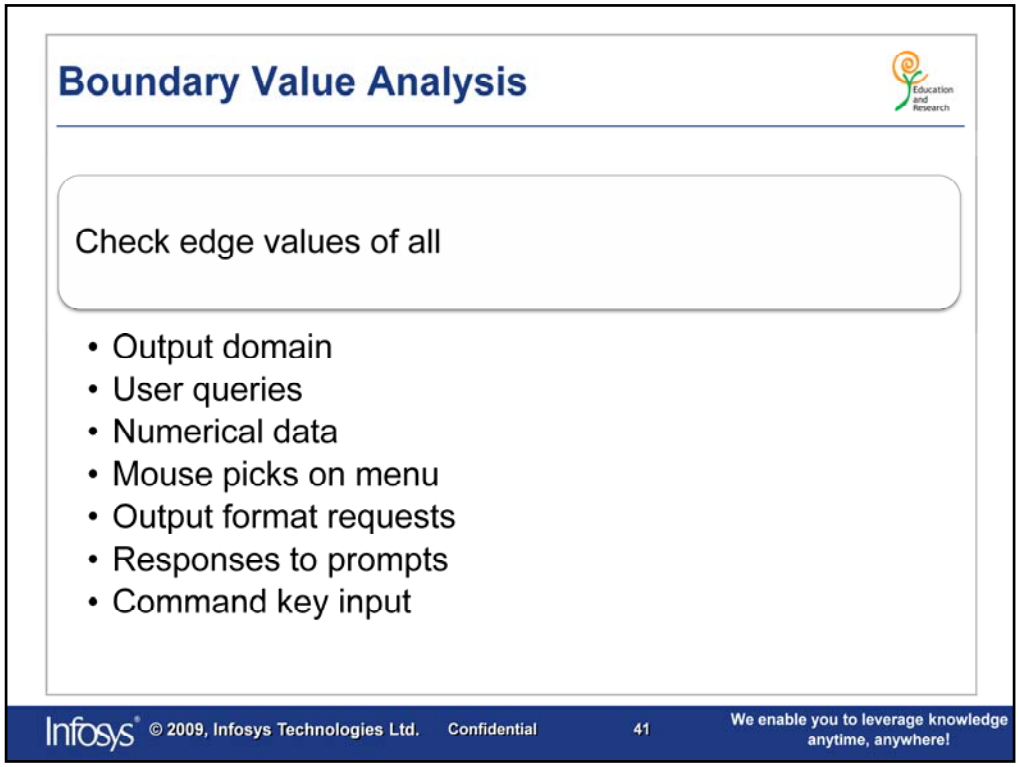
Equivalence Classes – invalid Data

- Data outside bounds of the program
- Physically impossible data
- Proper value supplied in the wrong place

Infosys © 2009, Infosys Technologies Ltd. Confidential 40 We enable you to leverage knowledge anytime, anywhere!

Some of the examples from various functions for equivalence partitioning – invalid data is listed here.

If the user-interface for the input doesn't allow the user to select invalid partitions – like a list box, radio button, push button, menu items or other similar interfaces, only valid values can be considered. It's a kind of defect prevention.



Boundary Value Analysis (BVA) testing technique for coming up with test cases that concentrate on the boundary or edges of input values; essentially considered as a subset of equivalence partitioning (EP). BVA has a higher probability of finding defects than EP. BVA scenario can be considered similar to as 'fence-post error': If you build a fence 100 feet long with posts 10 feet apart, how many posts do you need? Though the most probable answer for this is 10, which is wrong!

Taking the previous example of month input, possible boundary values for it would be: -1, 0, 1, 11, 12 & 13 (edge values of lower and upper ranges). If a programmer has coded to check the validity of a month in this way (common mistake):

if (month >= 0 && month < 13)

Instead of: if (month > 0 && month < 13)

the defect in the former case might get detected using BVA technique for test-cases. Examples of various scenarios where BVA technique can be used are listed above.

The guidelines for boundary-value analysis are:

•If an input specifies a range of valid values, write test cases for the end of the range and invalid-input test cases for the conditions just beyond the ends.

•Example: If the input requires a real number in the range 0.0 to 90.0 degrees, then write test cases for 0.0, 90.0, -0.001 and 90.001

•If an input specifies a number of valid values, write test cases for the minimum and maximum number of values and one beneath and beyond these values.

•Example: If the input requires the titles of at least 3, but no more than 8 books, then write test cases for 2, 3, 8 and 9 books

•Use the above guidelines for each output condition.

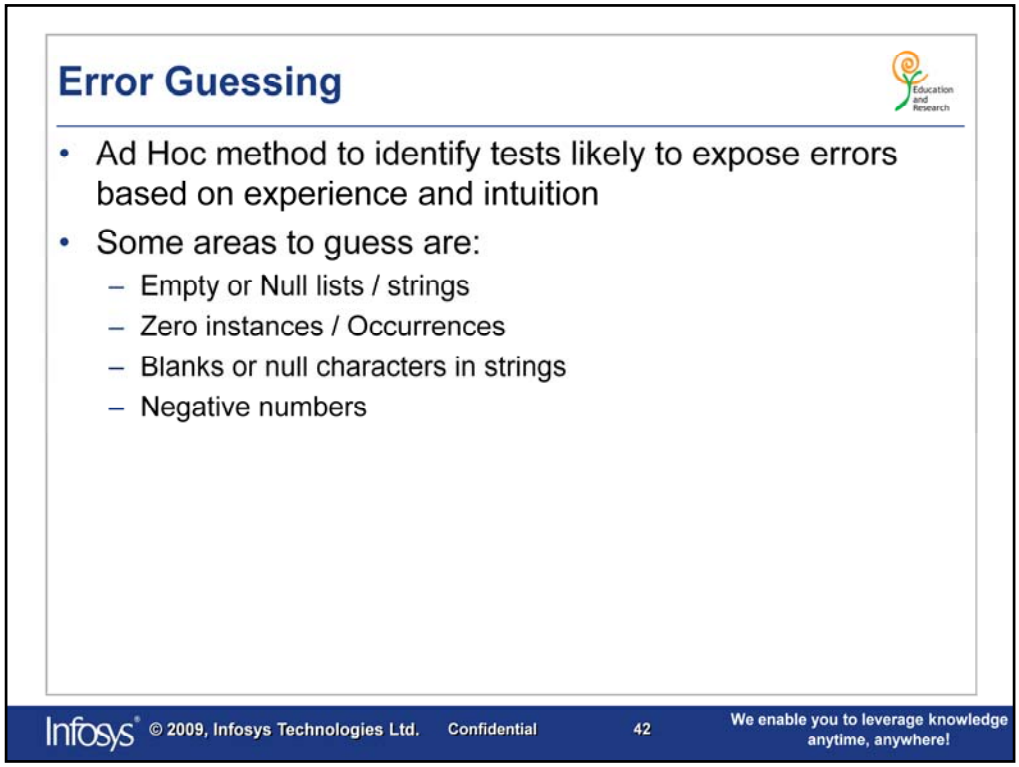
Come up with 'boundary value analysis' cases for following conditions:

Inputs: Age: 18 – 150, Gender = M,F, Married = 0,1

if age <25 premium+=100

if age >45 and <65 premium - =100

if female or married premium - =200



Error Guessing is a test case design technique where the tester has to guess what defects might be present in the program and to design the tests to represent them. It is advisable that an experienced tester uses this technique to come up with the test cases as there is formal hard-and-fast rule to guess the errors.

An experienced tester can guess the probable areas of defects (especially if it's the novice programmer who did the coding) in the program. Un-terminated strings, blanks or null characters in strings and similar cases as listed above.

Advantage/Disadvantage

- Equivalence Partitioning
  - Significantly reduces number of input conditions to be tested
  - Does not test combinations of input conditions
- Boundary Value Analysis
  - Boundary conditions sometimes subtle and difficult to identify
  - Does not test combinations of input conditions

Infosys © 2009, Infosys Technologies Ltd. Confidential We enable you to leverage knowledge anytime, anywhere!

As discussed earlier, equivalence partitioning significantly reduces the no. of input conditions to be tested for given set of inputs from the infinitesimal combinations. However, it may fail to come up with test cases conditions for a combination of input conditions (one input depending upon other).

BVA technique has higher probability of finding the defects. The BVA conditions are at times subtle and difficult to identify. It too fails to test conditions for combinations of inputs conditions.

Recap

- What is quality
- Dimensions of Quality
- Testing Techniques
- Unit Test

Infosys © 2009, Infosys Technologies Ltd. Confidential

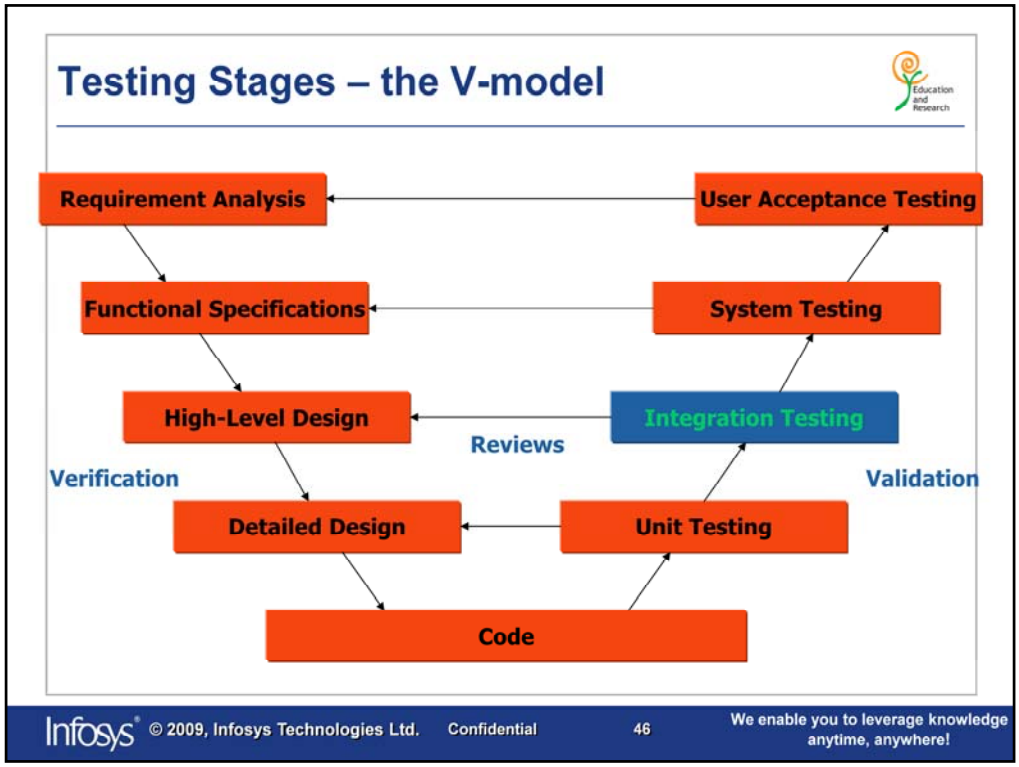
We enable you to leverage knowledge anytime, anywhere!

Education and Research

We enable you to leverage knowledge anytime, anywhere!

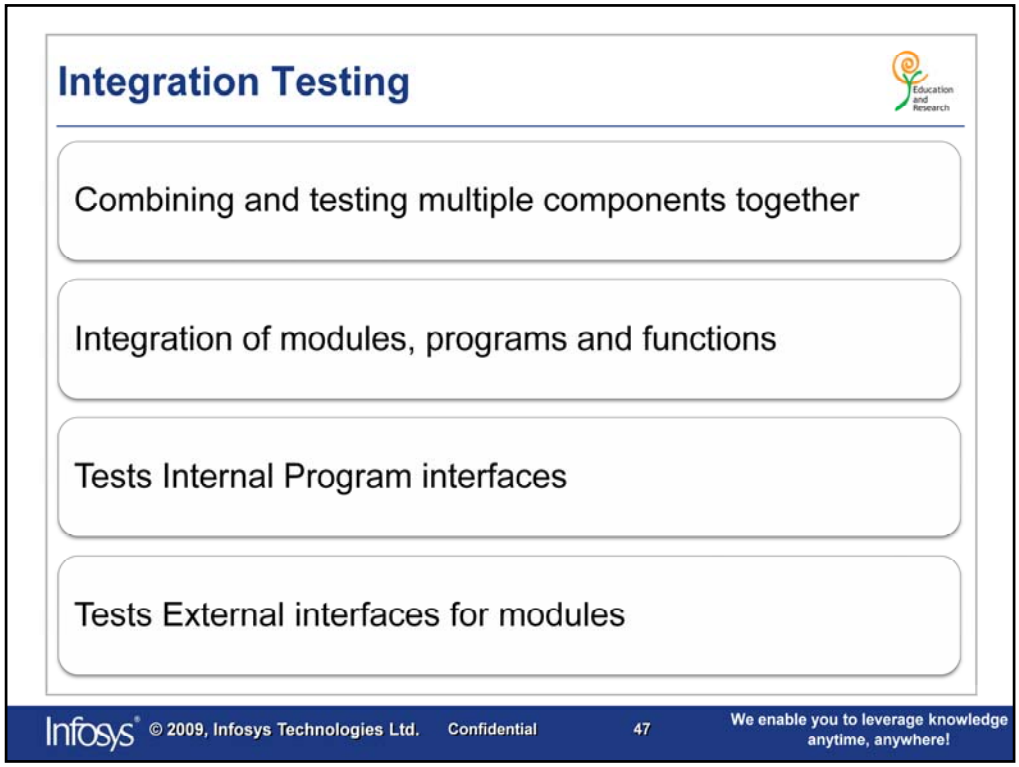
Integration Testing

© 2009, Infosys Technologies Ltd.



Next stage of unit testing is integration testing** which is based on or 'validated' against the high-level design in the famous V-model of testing. It is done to test if the modules are integrated (fitted) together properly.

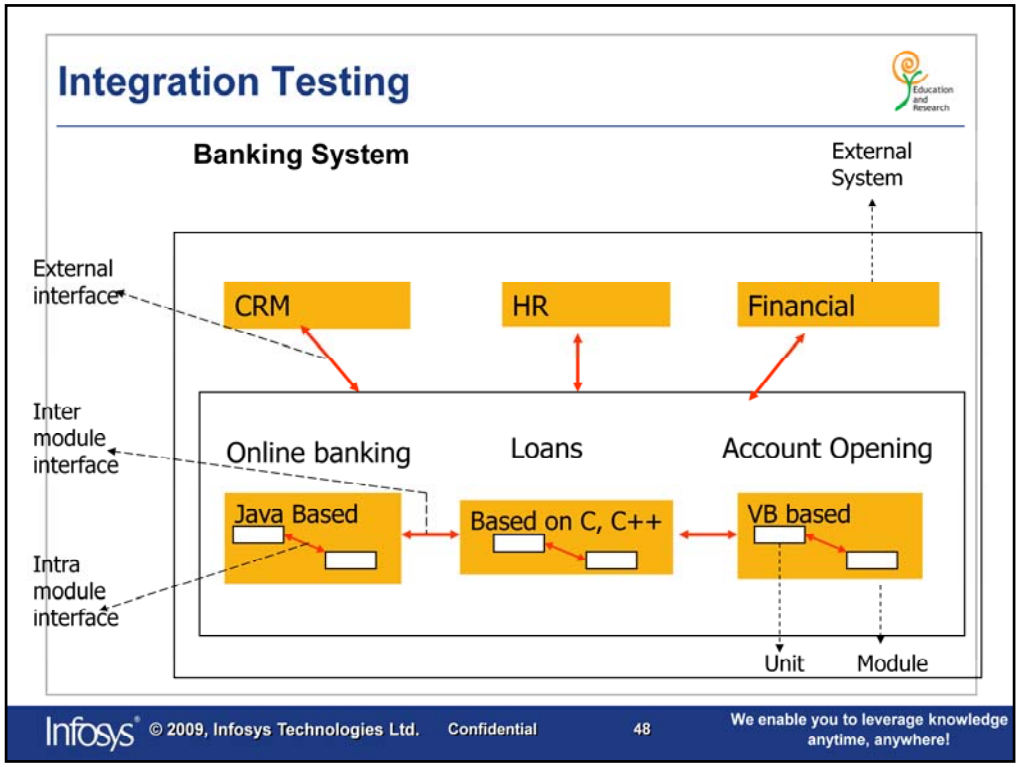
** In some projects in Infosys terms 'Integration Testing' and 'System Testing' are interchanged as per client standards.



Once the Unit Testing is complete, the modules are integrated together and testing done to check if the integration is seamless is called Integration Testing. This may again be done by the developers themselves or independently amongst them. Integration between modules / systems can happen in different ways:

1. Program-to-program integration (Intra-module interface)
2. Module (set of programs)-to-module integration (Inter-module interface)
3. System-to-system integration (External interface)

Integration Testing needs to focus on all these integration touch points. A pre-requisite to integration testing is that all the components that are being integrated should have been unit tested.



Consider an example in a typical Banking scenario. There are many modules / components to take care of various business functions: Account Opening, Online Banking, Loans Processing and others. These core banking modules may interact with external functions like HR, CRM and other Financial systems. Integration Testing should focus on various integration points in intra-module, inter-module and external integrations as shown in the diagram above.

Integration test plan should take care that:

1. Multiple components being integrated provide necessary communications, links and data sharing.
2. Integrated components work cohesively to perform a specified task.

Larger systems (like the scenario above) often require several integration steps.

Approach for testing interfaces

	Intra-Module	Inter-Module	External
Test Techniques	White Box	Black Box	Black Box
Test Environment Setup	Development environment	Isolated development environment	Simulate live environment
Test Approach	Check for parameter / procedure called. Database updation	Identify pre-process and approach to Interface. Identify the post-process to validate the interface	Identify external / internal interface and validate against external system

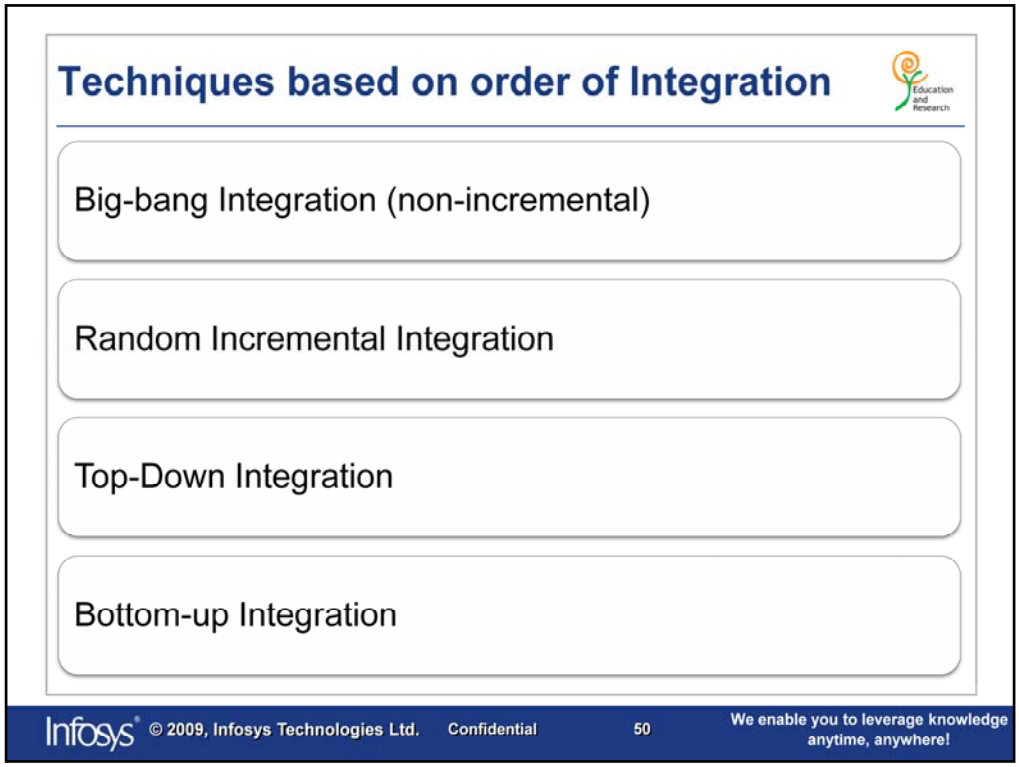
Infosys © 2009, Infosys Technologies Ltd. Confidential 49 We enable you to leverage knowledge anytime, anywhere!

Referring to the Banking System diagram in the previous slide, there are various integration points:

1. A function, say ProcessLoan() developed by a Hari, calling another function CalculateEMI(), created by Lisa – when Hari unit tested ProcessLoan() function, the sub-routine CalculateEMI() might not have been ready. In such case, Hari would create a 'stub' (a dummy sub-routine which would simply return a required EMI value as if CalculateEMI() routine returned it) to test the functionality of ProcessLoan(). Similarly, Lisa would have created a 'driver' program (a dummy program to just call the routine with required parameters) to test her function. When both functions are ready for integration testing, the focus should be on ProcessLoan() function calling the actual CalculateEMI() routine with various combination of values for input parameters of CalculateEMI() routine as well as on different 'return' conditions of that routine. Since it focuses on the internal structure of functions, white-box method of testing would need to be applied in such 'inter-module' integration cases. It is typically done in development environment.

2. Online banking module has an interface to the user to access his/her loan balances. For it to show this status, this module would need to interact with the 'loans' module. It can either be through the database, remote procedure call or other similar mechanisms – an inter-module integration. In this case, it's more of a black-box testing that is used where for a set of inputs, a specified set of outputs are expected. White-box method would prove tedious and time consuming. Typically it is (recommended to be) done in an isolated testing environment.

3. Banking system has an interface with other external systems like CRM, HR, Financial, etc. The interface would be through some middle-layers. Account Opening module may need to interact with CRM system to store certain data about new customers. Needless to say, testing done to verify this integration point should use black-box method and it should be done in a separate simulated live environment for it to be effective.



There are two basic integration test methods:

•Big-bang (All-at-once): In non-incremental 'big-bang' integration, all components (programs) are combined at once to form the program. The integrated result is then tested. While often used in practice, it is one of the least effective approaches. Debugging is difficult since an error can be associated with any component.

•Incremental (Random): Here we unit test the next program component after combining it with a set of previously tested components.

Advantages of incremental integration:

•It requires less work in the sense of fewer drivers and stub modules.

•Errors involving mismatched component interfaces will be detected earlier.

•Debugging is easier because errors found are associated with the most recently added component.

•More thorough testing may result, because the testing of each new component can provide further function and logic coverage of the previously integrated components

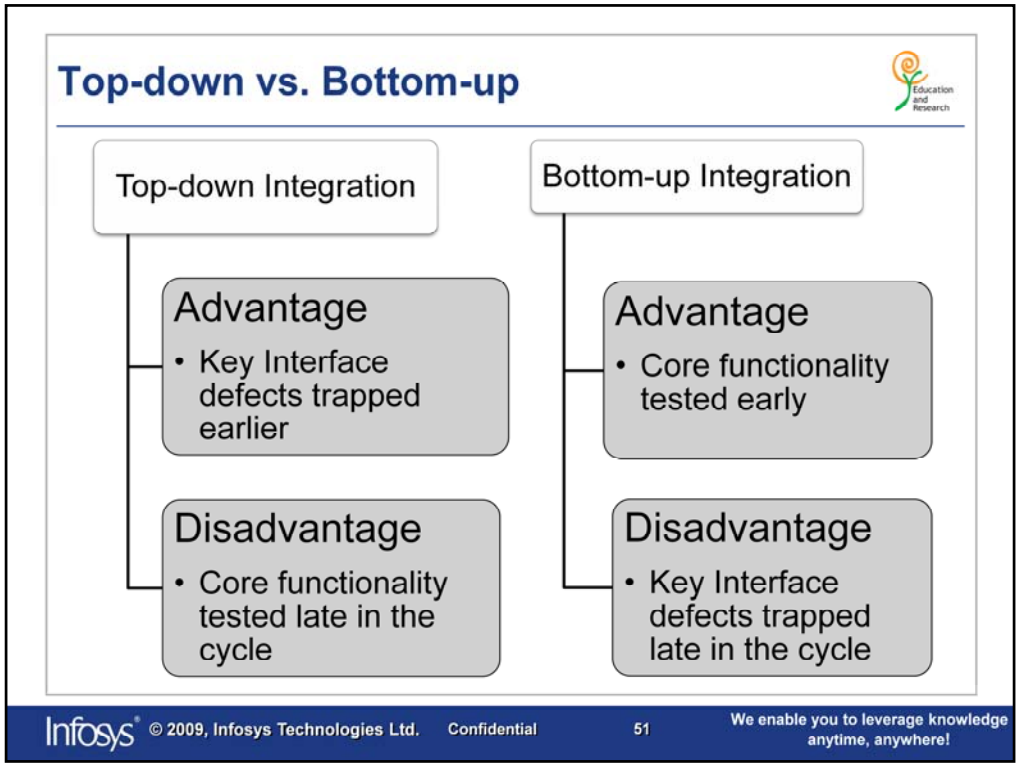
There are two approaches to incremental integration: bottom-up and top-down.

•Bottom-up: Steps in bottom-up integration are:

- •Begin with the terminal modules (those that do not call other modules) of the hierarchy
- •A driver module is produced for every module.
- •The next module to be tested is any module whose subordinate modules (the modules it calls) have been tested.
- •After a module has been tested, its driver is replaced by an actual module (the next one to be tested) and its driver.

•Top-down: Steps in top-down integration are:

- •Begin with top module in the execution hierarchy.
- •Stub modules are developed, and some may require multiple versions
- •The next module to be tested is any module with at least one previously tested super ordinate (calling) module.
- •After a module has been tested, one of its stubs is replaced by an actual module (the next one to be tested) and its required stubs



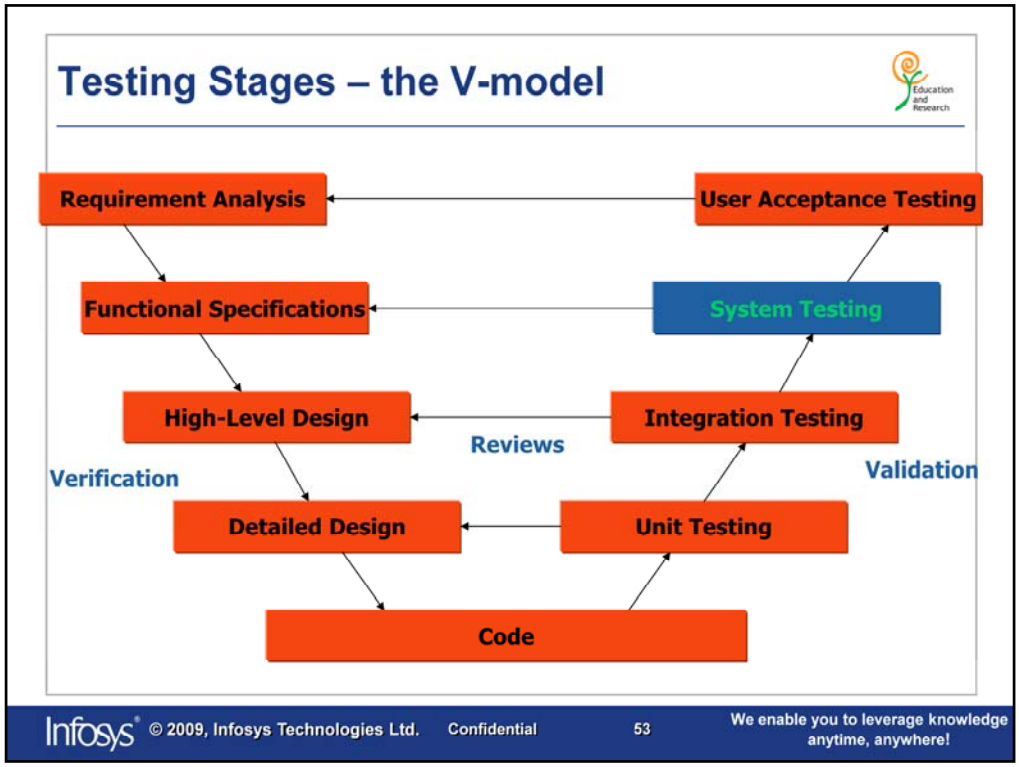
Advantages and disadvantages of top-down and bottom-up approaches are listed here.

Education and Research

We enable you to leverage knowledge anytime, anywhere!

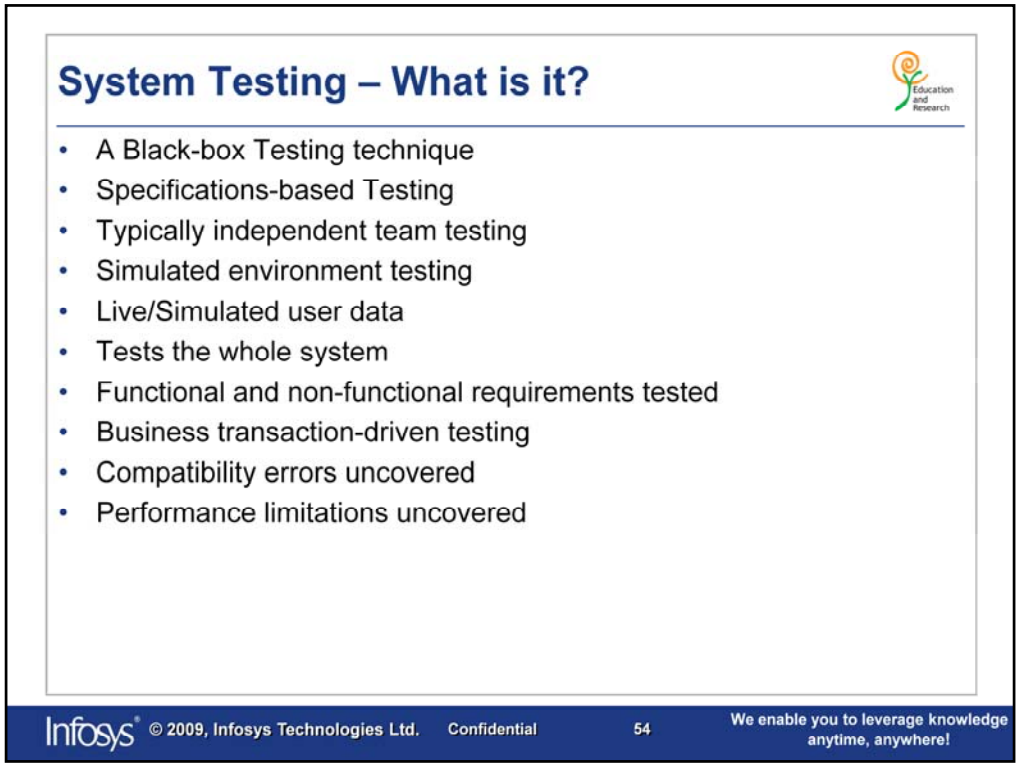
System Testing

© 2009, Infosys Technologies Ltd.



System Testing is the next stage of integration testing** which is based on or 'validated' against the functional specifications in the famous V-model of testing. It is done to test if the complete system as a whole is functioning as expected. During System testing one focuses solely on the outputs generated in response to inputs provided and execution conditions, rather than verifying the internal structure of the program/system. i.e. the system is validated against its functional requirements. System testing is carried out when Unit and Integration testing have been completed successfully.

** In some projects in Infosys terms 'Integration Testing' and 'System Testing' are interchanged as per client standards.



System testing is a black-box testing technique which is done based on the functional specifications of the system. It is typically done by an independent testing team in a separate simulated live environment using the live or simulated data. It treats the whole system as a black-box, for a given set of inputs (for various business scenarios) verify if the specified outputs are obtained.

It also focuses on non-functional aspect like performance, compatibility, usability, etc. of the system.

A system tester should have a good understanding of architecture and application domain of the system. The System test plan needs take care of the following details:

1. Compliance of a system or component with its specified functional requirements.
2. Validation of business rules
3. Validation of system resource (H/W) and the application response when multiple components are executed simultaneously
4. Destructive tests to determine if the software does things it shouldn't do.
5. Inter and intra components dependencies.
6. Controlled environment under which the testing will be carried out
7. Data that will go as inputs to the various tests and the corresponding expected output
8. Coverage of all critical decision making components

System Testing – Different Types

- Functional testing (Sanity / Regression)
- Performance and Scalability testing
- Load/Stress testing
- Volume Testing
- Usability testing
- Configuration testing
- Installability testing
- Disaster and Recovery testing
- Reliability / Availability testing
- Statistical Testing based on Usage Models

Infosys © 2009, Infosys Technologies Ltd. Confidential 55 We enable you to leverage knowledge anytime, anywhere!

•Sanity testing involves a minimal set of test cases to ensure that basic functionality (High probability of acceptable operation and performance) of software is intact. The selection of test case could be based on several factors viz. key customer functions, likelihood of functions breaking from code changes, historical breakage frequency, etc.

•Regression testing involves re-testing of the system or component to verify that modifications / enhancements have not introduced unintended defects on some existing features / modules. It would also involve re-testing prior bug fixes to ensure they work with the current fix / enhancement. The emphasis of regression testing is to ensure that the new modifications have not adversely affected existing functionality.

•Load testing is a kind of boundary testing condition. Run the test on the software that should be able to pass (with say, max. no. of users / terminals) and another test so that it fails (too many users / terminals).

•Stress testing is a form of testing that is used to determine the stability of a given software or a system. It involves testing beyond normal operational capacity, often to a breaking point, in order to observe the results.

•Volume testing refers to testing a software with certain (usually large) volume of data. The volume can be the database size or could be size of a file that the program reads. Main objective is to check how the system handles such large volume.

•Usability testing is a measure of how well the end user's can use different components of a system like a web-page, a report, a document, an interface, a device, etc.

•Configuration testing involves testing the system for different variants of its configuration (such as memory, CPU, clock speed, hard disk) as specified in the requirements specs

•Every software will have certain installation requirements. Testing done to verify if the software gets installed as per the specified installation procedure is installability testing

•Every company would have certain disaster recovery plans which will contain the actions to be taken when a disaster strikes. Testing done to check if DR plans are good enough to handle the disaster cases is called disaster recovery testing. Done in a simulated environment.

•Testing done to verify the reliability of the software (a measure of how much time is it continuously available for use) is Reliability testing. The concept of reliability here is the probability of execution without failure for some specified interval (available time).

•A software usage model is the (estimated) distribution of the possible uses of a given software in its intended environment. This usage model can serve as input to statistical testing based on usage model.

Functional Testing – Sanity Test

- Very basic minimal number of tests to verify the product for the feature/ protocol compliance
- Could be a subset of regression test used to validate only high level functionality
- Typically an initial testing effort to determine if a new software version is performing well enough to accept it for a major testing effort
- Sanity testing will be performed in scenarios when cursory testing is sufficient to prove that the application is functioning properly

Infosys © 2009, Infosys Technologies Ltd. Confidential We enable you to leverage knowledge anytime, anywhere!

More of a subjective testing done by experienced testers. May not always be considered as a formal, complete testing.

Why Sanity Testing?

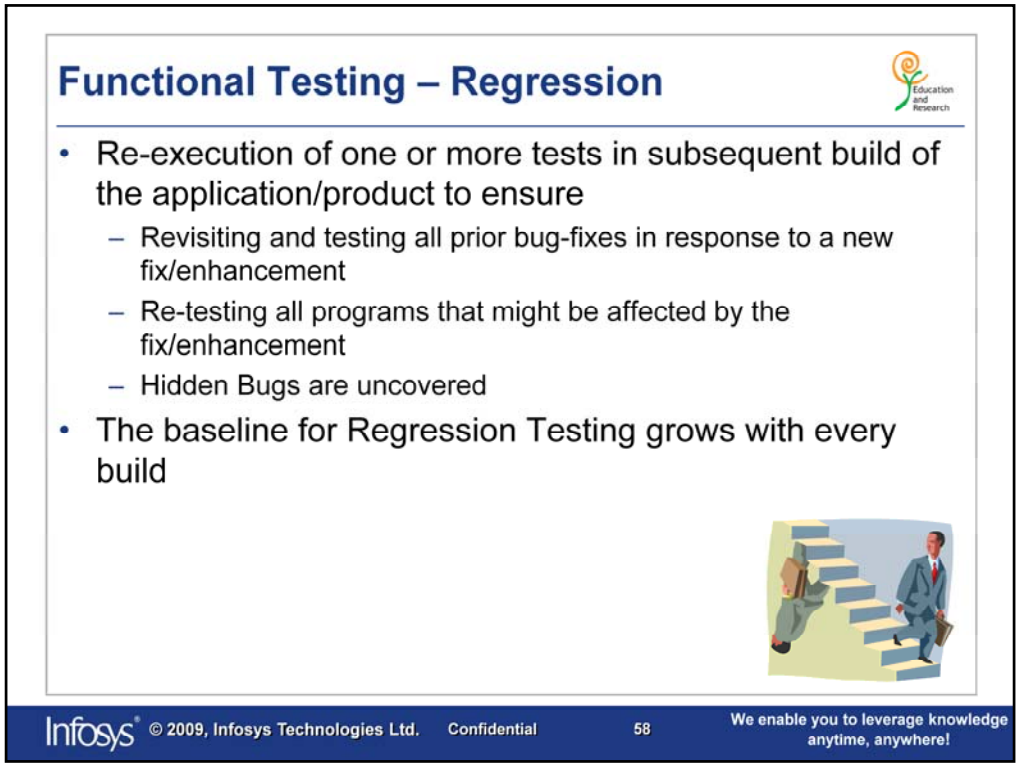
Regression testing is more time taking and more effort is required whereas Sanity testing is quicker focusing on important functionality of software

Saves time as only selected high level functionality areas are only tested. Bugs found this way would take almost triple the time when found through regression testing

Efficient in scenarios wherein releases comes in pretty often with very few changes

Infosys © 2009, Infosys Technologies Ltd. Confidential 57 We enable you to leverage knowledge anytime, anywhere!

Sanity testing, if done properly by an experienced tester, can save lot of time in system testing. It can also detect a major bug (which otherwise would have been detected during a formal, time-consuming testing process) early in the testing cycle.

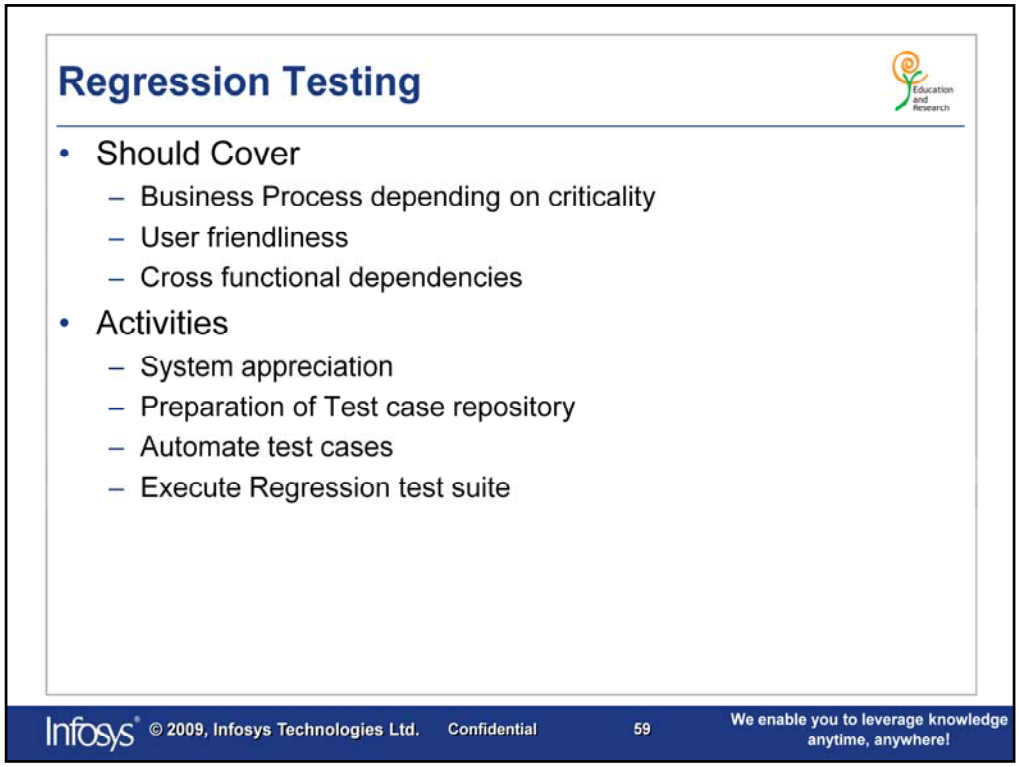


Consider a sample scenario: When you make a small change to a sentence in one of the paragraphs of a document, you re-read the whole paragraph again (at times, previous or subsequent paragraphs too) just to ensure that the change made does not impact the overall flow of the document. This is a kind of regression testing.

A change made to the software (however small it is) should not impact or change the existing feature of it unintentionally / unknowingly. Regression testing is helpful in identifying such hidden bugs.

Regression testing is applicable from the 2nd version / release onwards of any system or software. A test-case base-lined in the earlier version of the software is used for regression testing in the current release. This test-case results are matched with the results of earlier version (stored earlier). If the results differ and the difference is intentional (i.e. implemented as part of current release), the change passes the regression testing; otherwise, it may be due to a hidden bug that is introduced in the current release which should be fixed before going ahead with the release.

Once all the bugs are fixed, new test-case/s (passing thro' new features implemented in the current release) need to be created and base-lined for regression testing of future releases. So, baseline for Regression Testing grows with every build.



Regression test cases should be based on the critical business functions that are implemented in each release of the software. As they get base-lined in every release, it is very important that these test-cases should focus on cross-functional dependencies. Any inadvertent bug that is introduced in any release may prove very costly to the system in the future.

Here is how a typical regression testing is done:

•Once the system appreciation is (a one-time activity during the initial release of software) complete, the testing team has to create a set of test-cases which pass thro' critical business functions of the system and base-line them along with their results (in the database or can be in the form of separate artifacts like screens, reports or similar outputs)

•During the subsequent release of the software, once all other testing stages are complete (unit, integration & system), choose the base-lined test-cases for regression testing which would pass thro' those business functions which would have probable impact in the current release due to the new enhancements or bug-fixes

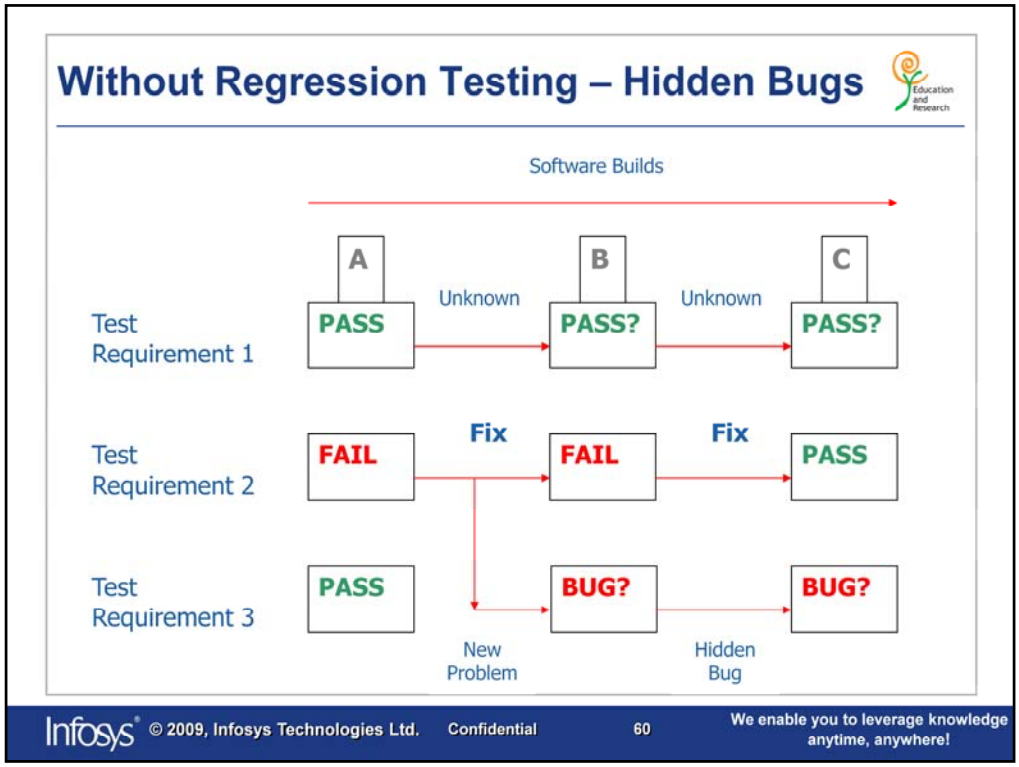
•Execute these test-cases on the new release of the software, collate the results (database table rows, screens, reports, etc.) and compare them with the corresponding results of these test-cases which were stored in the previous release

•If there is any difference, check if it is due the new enhancement or bug-fix that is carried out during the current release. If yes, this difference is expected and apparent.

•If not, it could be due to a bug that would have got introduced in the current release. Identify the root cause of it, fix it and conduct the regression testing again (after all other testing) and make sure that this hidden bug is fixed

•At the end of regression testing, create few more test-cases which pass through critical business functions that are introduced in the current release, baseline them along with their results for future use.

Regression testing scenarios more conducive for automating the test-cases; it should be explored wherever possible.



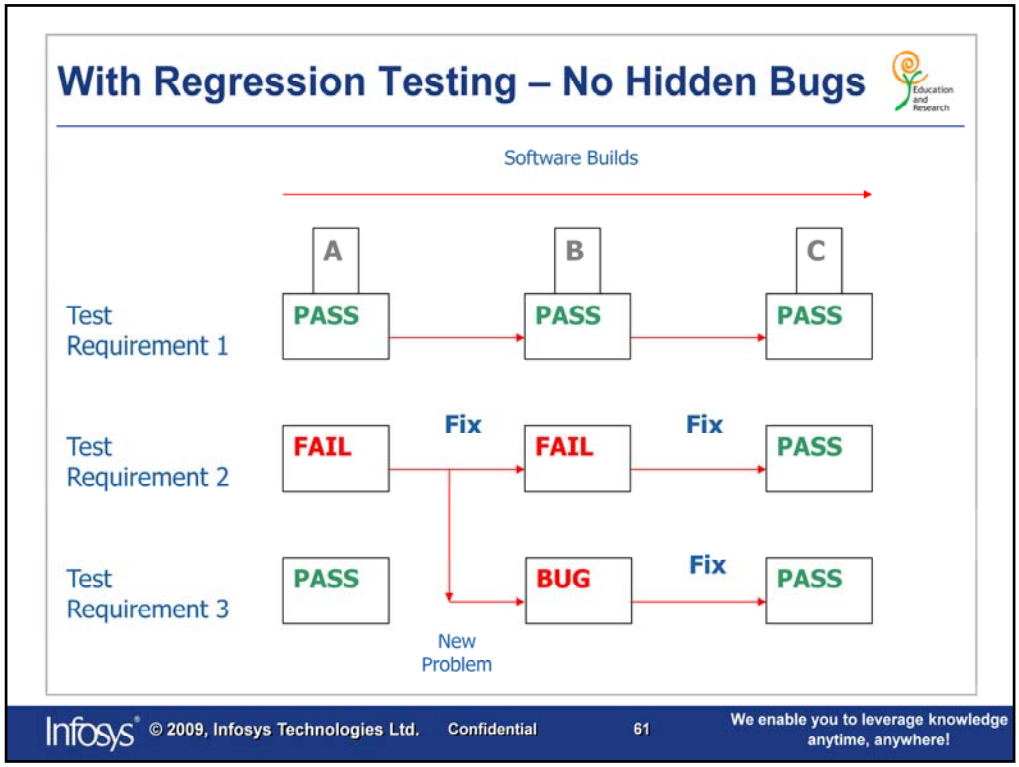
Consider these scenarios of a typical software release process:

3 requirements are being tested in 3 different stages of build “WITHOUT explicit regression testing” stage:

Test requirement 1: No defects in the 1st pass. Subsequent tests report that there are no bugs in it (without actually regression testing it)

Test requirement 2: Has defects in 1st build phase. Developer tried to fix it, but fails again in the 2nd phase. Ultimately gets fixed it in the 3rd phase

Test requirement 3: No defects in the 1st phase. But the developer while fixing the defect of TR-2, inadvertently introduced a defect in TR-3 which might go undetected till the release since no regression testing is done during these releases.

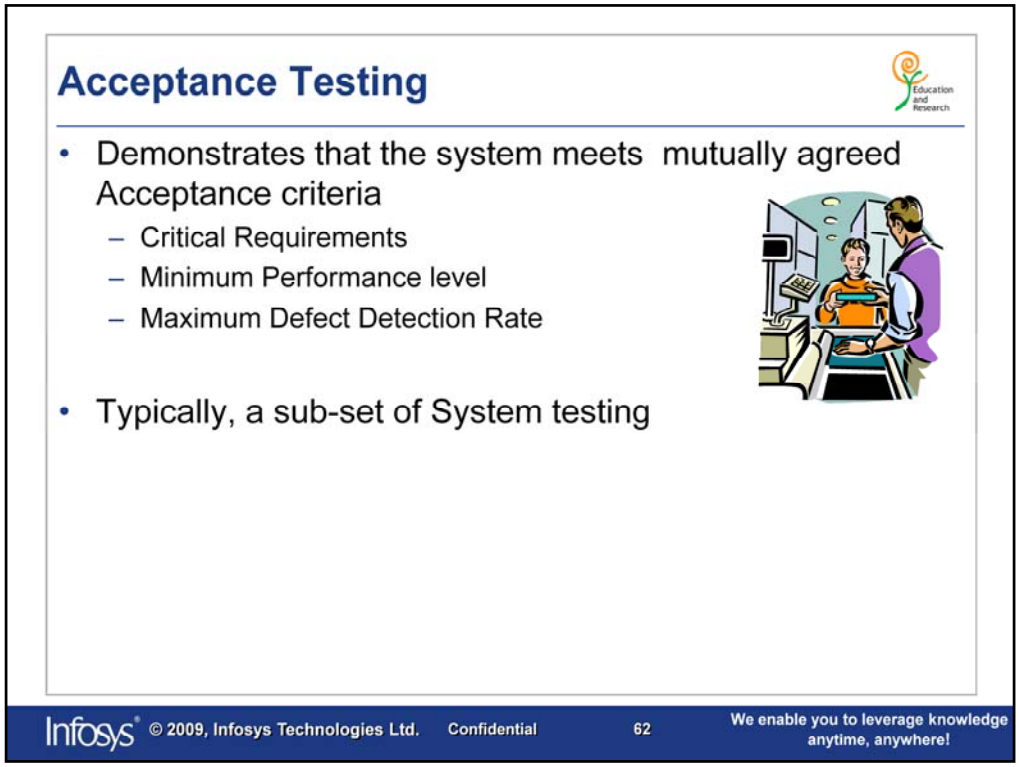


Requirements are being tested in 3 different stages of build "WITH explicit regression testing" stage:

TR-1: No defects. Passes thro' other couple of tests without any issues since regression testing confirms that there are no hidden bugs

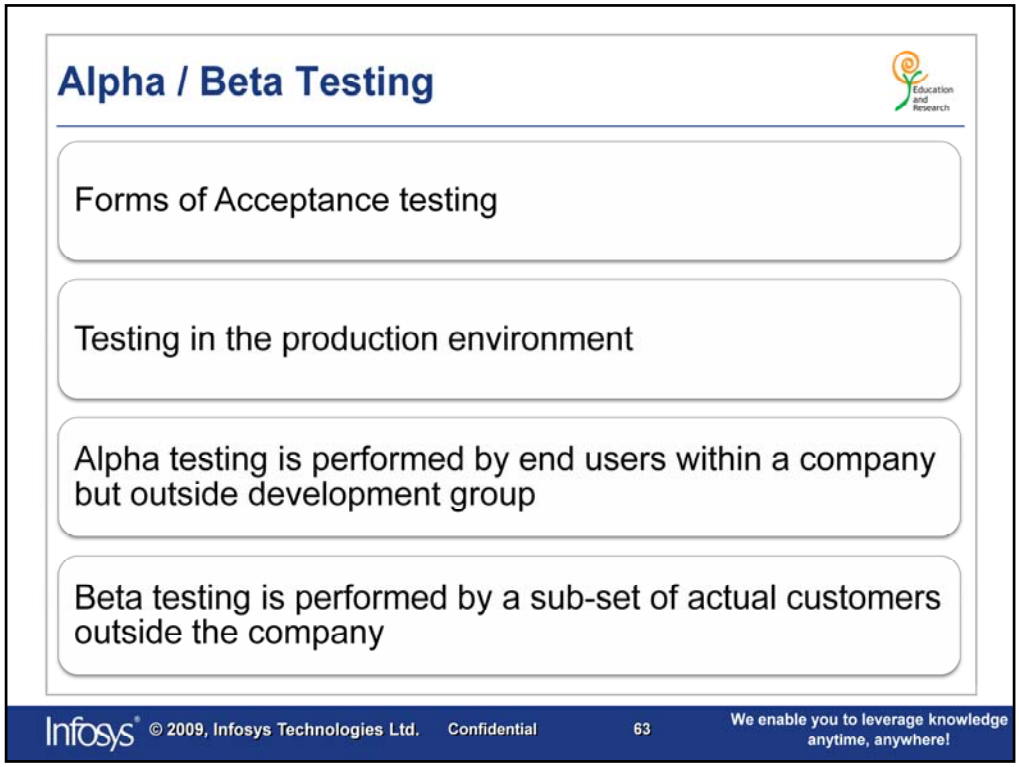
TR-2: Defective in 1st build. Developer tried to fix it, but fails again in the 2nd phase. Meanwhile, introduced a bug (unknowingly) during this fix stage. Ultimately gets fixed it in the 3rd phase.

TR-2: No defects in 1st phase. Regression testing detects the hidden bug in it. Programmer fixes it during the 2nd phase. Passes thro' all tests during the next build phase.



Acceptance testing is a black-box testing performed on a system just before its delivery (implementing it in production area), typically done by the end-user or customer. It generally involves running a set of test-cases on the complete system focusing on the critical business functions. Recall that the acceptance criteria should have been mentioned at the end of requirements cycle. Various business scenarios and the minimum performance levels in which the system should work (screen should come up in *m* sec, report should be generated in *n* sec, batch program shouldn't take more than *p* hrs, etc.), would have been mentioned in those criteria.

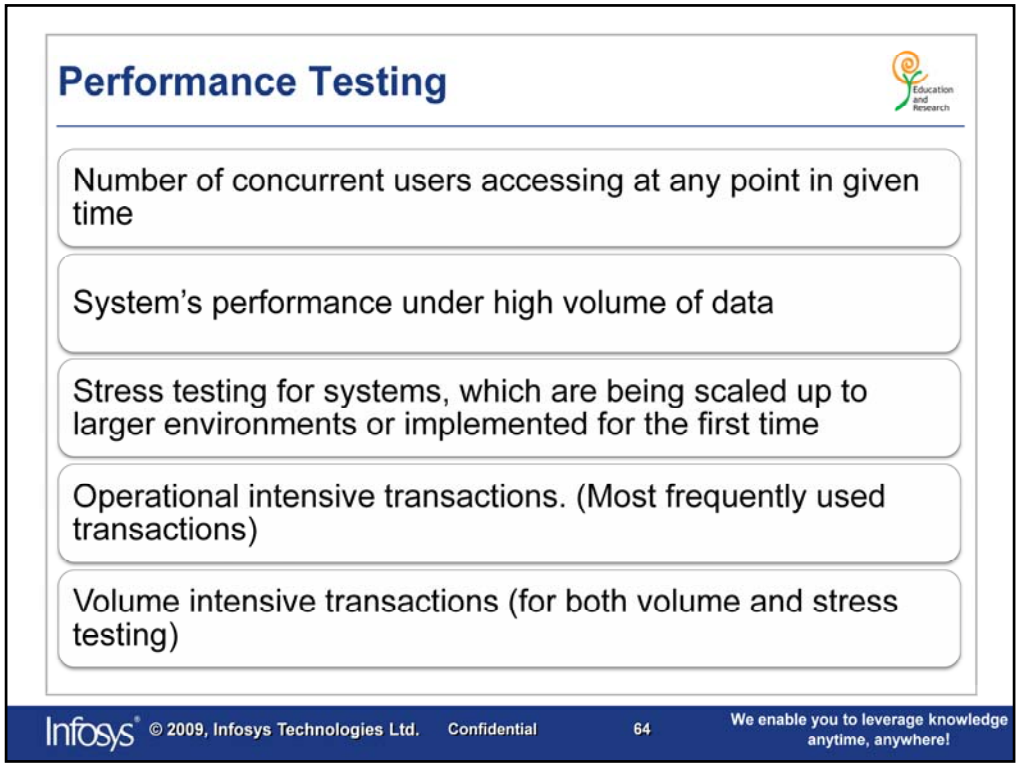
Acceptance testing, carried out after all other testing stages are complete (however, it is a sub-set of system testing), where the system is complete in all respects, usually has a maximum defect detection rate (applying Murphy's law, if things can go wrong, it will!).



Alpha testing is simulated or actual operational testing by potential end-users/customers or an independent test team at the developers' site. Alpha testing is often employed for software products as a form of internal acceptance testing, before the software goes to beta testing.

Beta testing is done after alpha testing. Releases of the software, known as beta releases or versions, are released to a limited set of people outside of the organization. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

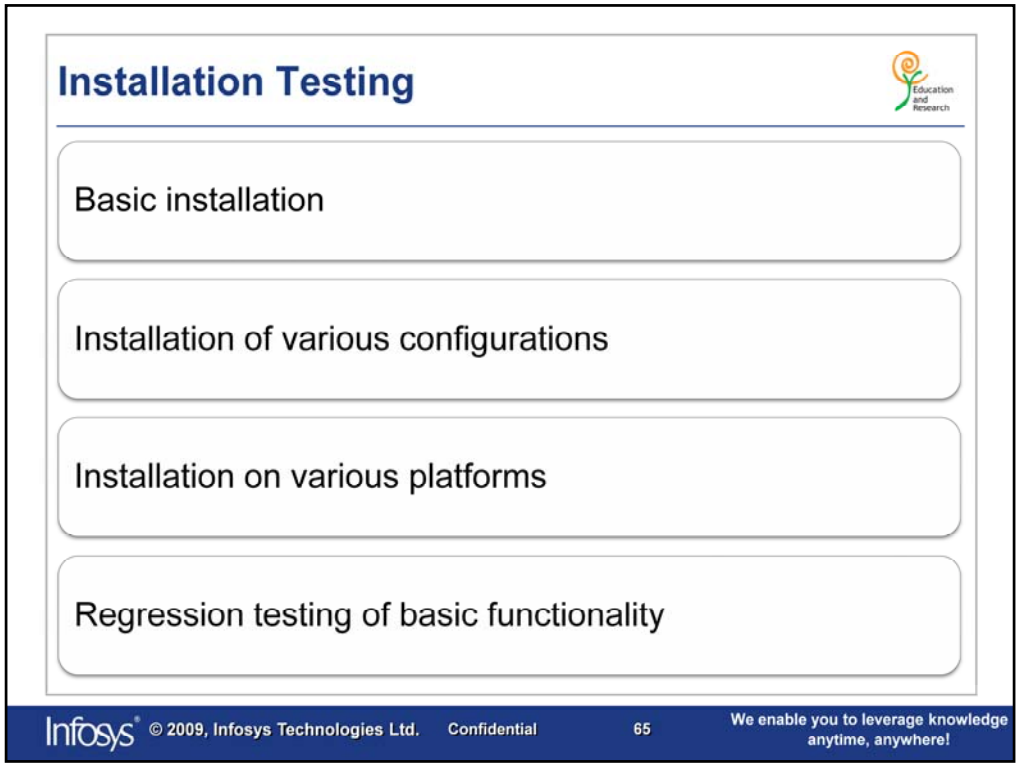
For e.g. New release of Windows would be available for beta testing in the market before actual shipping of its final version.



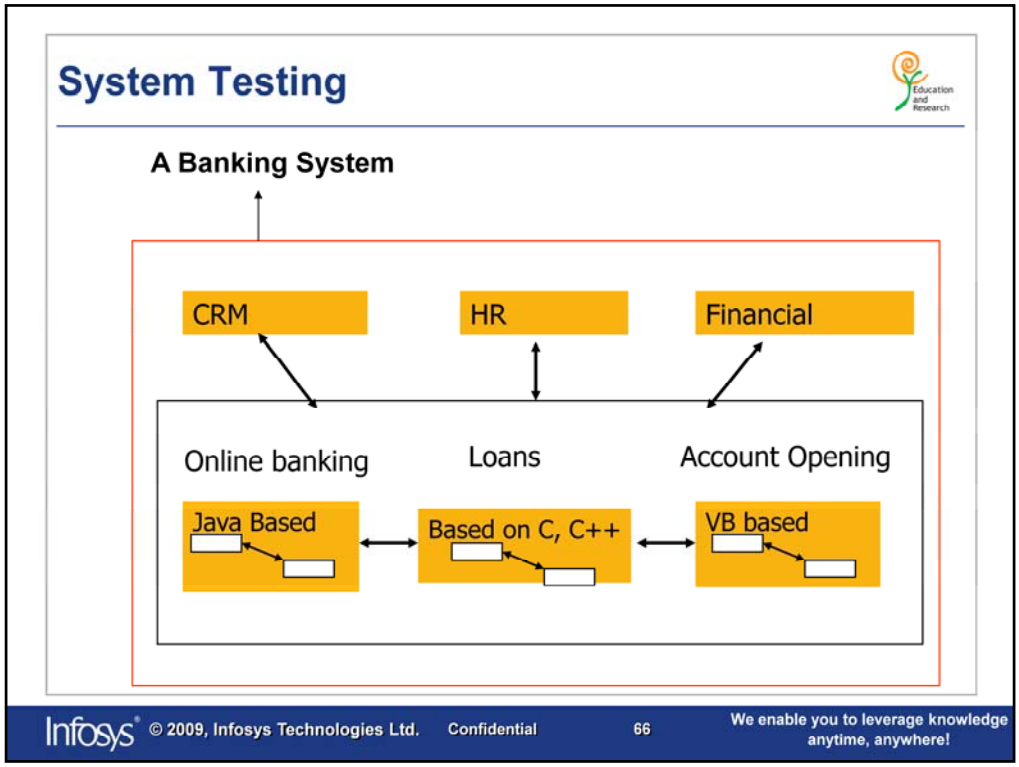
Performance test verifies that the suite of systems meets expectations for performance, and that it does not unexpectedly degrade performance of the business as a whole. Performance test conditions should be derived from pre-determined service level agreements (SLAs) to ensure that the proper focus is given to risk areas.

Performance testing requires special skills. Many Performance testing tools are available, and it is always preferred that tools be used for performance testing. These tools simulate load, in terms of the number of concurrent users accessing the system. Performance tests are referred to as non-functional tests. Performance test plan should take care of:

1. Number of concurrent users accessing at any point in given time
2. System's performance under high volume of data
3. Stress testing for systems, which are being scaled up to larger environments or implemented for the first time
4. Operational intensive transactions. (Most frequently used transactions)
5. Volume intensive transactions (for both volume and stress testing)



Testing done to ensure that the system gets installed properly without any hassles in the specified destination (platform / environment). Different modules of the system may have different configuration and platform requirements (hardware, software, network, OS, App server, Web server, etc.). Installation testing should test for all these conditions to ensure successful installation of the system. It also involves the regression testing of basic functionality of the system.



That is about the complete System Testing and its different forms – considering the whole system as a black-box and testing based on various specifications. In the example shown above, whole Banking System including all its interfacing components is considered for System Testing.

Typical scenarios would be:

•A new customer of the bank opening an SB a/c and viewing her a/c details thro' "Online Banking" and her details being made available to CRM & Financial module

•Same customer going for a loan in the bank, linking her loan a/c to SB a/c (for say, loan EMI payment) and the bank furnishing this details to 'financial' module for its financial statements

When do you stop testing?

- When time runs out
- Specific test case design methods
- Target test coverage attained
- Certain number of errors found*
- Error detection rate drops below a threshold*
- Predicted density of errors left drops below a threshold

Sequence test cases in order of priority

* Based on Infosys Process Capability Baseline (PCB)

Infosys © 2009, Infosys Technologies Ltd. Confidential 67 We enable you to leverage knowledge anytime, anywhere!

It is very well perceived that testing like any other activity, takes time, effort and cost. Testing should be stopped when any of the above parameter exceeds budget. It is always advised to discuss budget required for testing with the customer. Ensure optimization in terms of functionality, reliability, effort, time and risks associated with each of the above parameters, is covered during discussion. Document the same as part of the Test strategy and get the same signed off by the customer. During Test execution these parameters provide guidelines for further action.

The following points describe situations how and when testing can be stopped:

•The test stop criteria can depend on the methodology adopted for software development. E.g. If the project works on incremental model the testing can be restricted by the functionality developed during that cycle taking into account Time to Market.

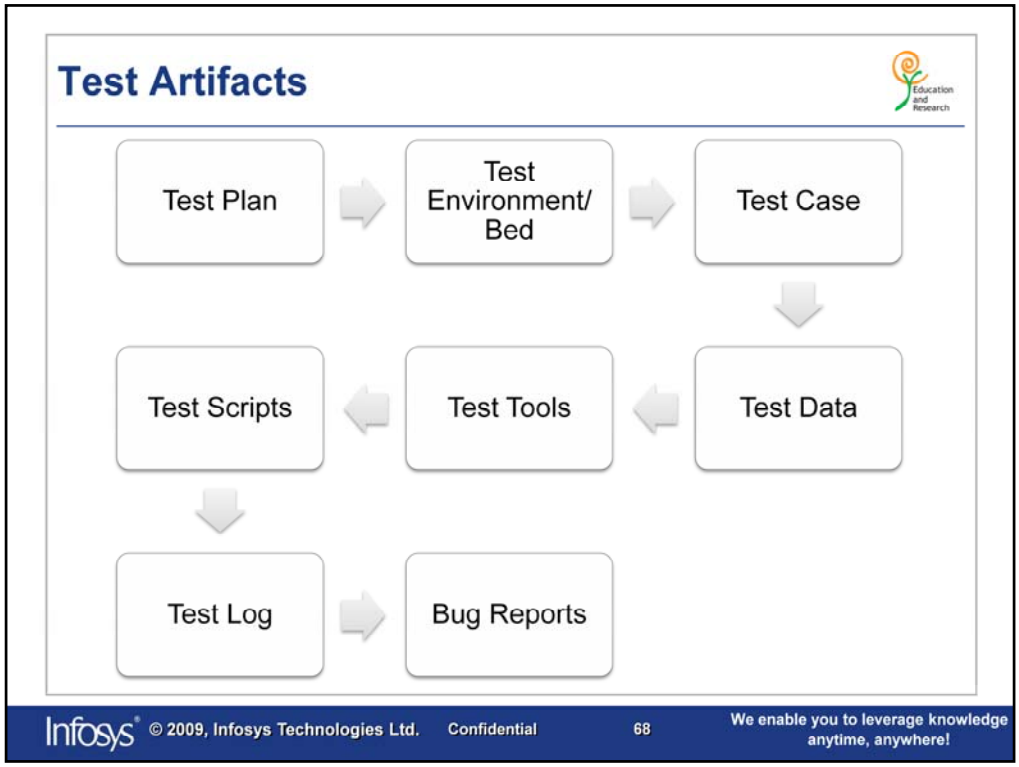
•If changes are frequent, and changes are implemented in parallel with development, then testing of both will not help Time to Market. Under these circumstances it is advised, that you list critical transactions and prioritize and test only those transactions.

•Timelines of the project may be very narrow and you may not be able to test 100% functionality. Under these circumstances arrive at the percentage of functional testing that can be completed within the specified timeline so that quality is not compromised. Once again prioritize critical transactions.

•In the test strategy, severity of defects needs to be defined where you specify the nature of fatal, major, minor and cosmetic error. Project teams can specify in the test strategy, that software will be released with minor/cosmetic bugs.

•Predicted no. of defects based on the defect-injection-rate (DIR) and the error detection rate (review efficiency) can also be the parameters that could be considered to stop testing

•There are times when testing cannot be continued simply because the defect found is a showstopper and there is no round about way of continuing testing for the remaining part of the software. In such a situation, the tester will not have any choice but to fix and close the defect quickly and then proceed.



Test Artifacts:

•Test Plan: One of the most important artifacts of testing. It is a plan document that covers the overall plan for entire Testing activities for the project. This includes the scope and objective of the plan, references to other documents, the test items and their prioritization, tests to be conducted, tools to be used, environment setup, resources and infrastructure, test entry and exit criteria, test measurements

•Test Environment/Bed: A combination of hardware and software platforms together with other third-party products (if any), application configuration and data on which a Testing activity is performed

•Test Case: A series of steps for executing a program or a set of programs, along with a set of data that should be provided as inputs. In addition, a test case also specifies the expected output for each step to be executed

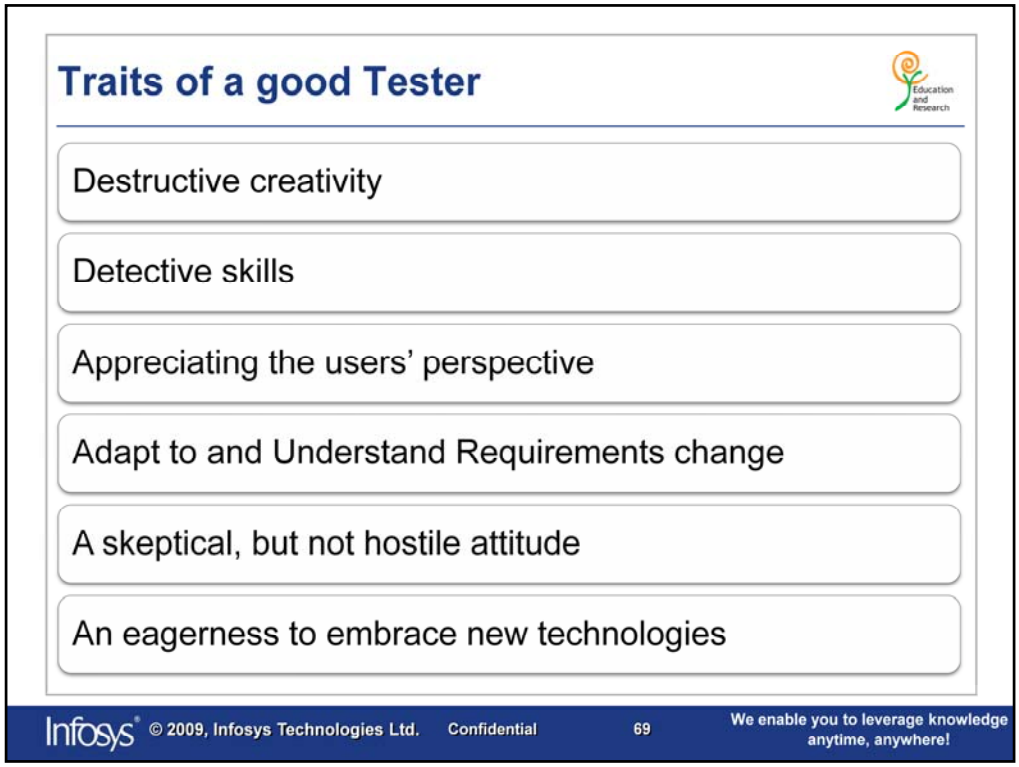
•Test Data: A set of values that should be provided as input while executing a test, along with the expected output associated with that step.

•Test Tools: These are third-party products or in-house tools that aid in automating the Testing activity. These tools typically work on a record-and-playback mechanism and aid in simulation of production-like loads for simulated Load and Performance Testing. Most of these tools also provide output/error logging and reporting features for a consolidated view of results and to help compare/contrast them.

•Test Scripts: A set of scripts, which when executed in a particular sequence, runs the test and logs/displays the output as desired. Test scripts are typically used while regression testing where the test sequence is recorded and played back later, as desired through the scripts. The scripting language varies with the tools being used.

•Test Log: It is a file that logs the output /errors during manual /automated testing activities. These files provide a consolidated view of results and help compare/contrast them. They provide data on test coverage and the proportion of tests that passed/failed. They also log the errors reported against each test executed along with the input data.

•Bug Reports: A consolidated account of all the bugs reported during various stages and cycles of testing, along with their severity and priority for fixing, and their current status of fix. Bug reports also typically include a short note on the fix provided and its associated impact on other modules of the system. RADAR (Review and Defect Analyzer) is used in Infosys for tracking the bugs.



Software test engineers require technical skills, but they also require other characteristics that help them appreciate the customer or user's perspective while testing it. The following are some traits that can help a tester test a system/application thoroughly:

•Destructive creativity: A software test engineer cannot be afraid of causing a product to crash and burn. In software testing, boundaries are meant to be crossed, not obeyed.

•Detective skills: In an ideal world, the software to be tested should be well documented and updated regularly. Due to time pressures, it may not be possible to update requirements document, or requirements may just not be available. Under such situations, it is difficult to track changes and prepare a test plan.

Because we do not create software in an ideal world, the test engineer must be able to figure out how things work on her own. Typically, some design or functional specifications will be available, so the test engineer can use these documents to begin her research. Skilled testers should find ways and means of getting right information required for testing.

•Formal System

- •Requirements Documents
- •Functional Specifications
- •Design Specifications
- •User Documents

•Informal System

- •Communicating with developers
- •Communicating with software support personnel
- •Documents from related products
- •Defects from related projects
- •"Local knowledge" from working on related, or earlier versions of the products

Traits of a good Tester (Contd..)

Infosys © 2009, Infosys Technologies Ltd. Confidential 70 We enable you to leverage knowledge anytime, anywhere!

•Understanding the product as the sum of its parts: The developer will concentrate on his piece of code rather than the entire system; this induces bugs when components are assembled and tested, whereas a tester cannot afford to have the same mentality. Her strength lies in understanding individual components of a system, and testing the same from a system perspective.

•Appreciating the users' perspective: The test engineer must also act as an advocate for the customer. The program under test may perform reliably and may meet all requirements, but it may not be operable under production. So it is essential for tester to simulate production like environment.

•Requirements change: Testers should have the patience to test and retest the same system or component repeatedly. It is the nature of software that it evolves every day. A customer with ever changing market needs is prone to request for frequent changes in software. The objective of a tester is to identify a defect early in software life cycle, thus minimizing defects that get released along with software.

•A skeptical, but not hostile attitude: The test engineer must not accept things at face value, but must be tenacious in questioning everything until it's all proven. The engineer must, however, balance this skepticism and stubbornness with a spirit of cooperation with the rest of the project team. The test engineer must remember to assault the integrity of the programs, not the programmers.

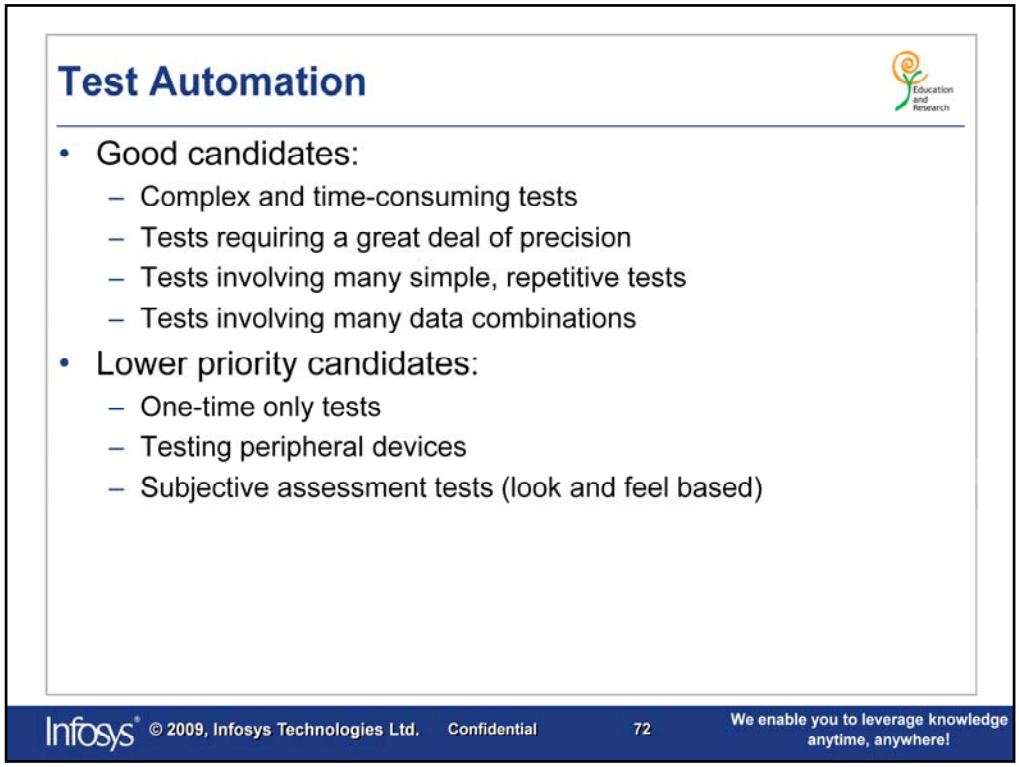
•An eagerness to embrace new technologies: The Internet is increasing the rate at which technology is changing. A testing team needs to be as much equipped technologically, as the development team itself. Knowing the technology herself helps her appreciate what is achievable within the given architectural framework and what is not. This helps build a stronger rapport with the development team. The development team and testing team need to coexist to achieve a common quality goal.

Test Engineers

- Role - Execute tests, record test scripts, maintain statistics and metrics, check test data setup, test environment setup, execute re-tests
- Knowledge - Understanding of the importance of testing, awareness of tools, how to progress against a plan
- Skills - Observation, accuracy, methodical, co-ordination, problem solver
- Experience - Following instructions, problem reporting and solving, and relevant testing tools

Infosys © 2009, Infosys Technologies Ltd. Confidential We enable you to leverage knowledge anytime, anywhere!

Responsibilities of a test engineer, the knowledge, experience and different skills that s/he should possess are listed here.



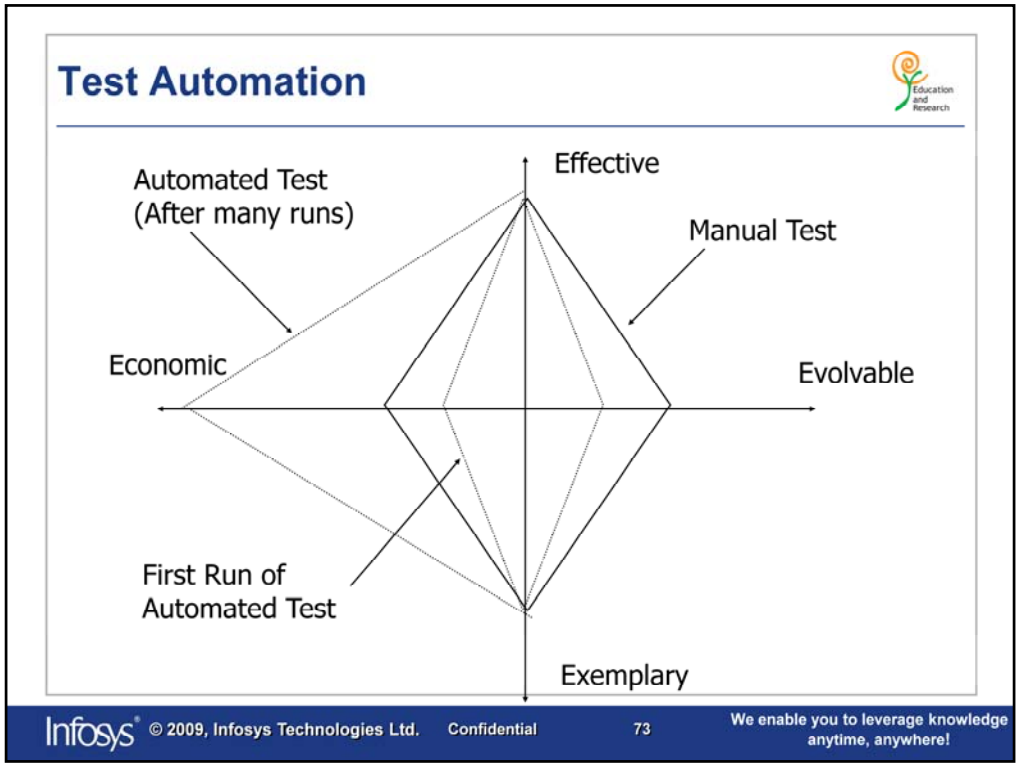
Automating software testing can significantly reduce the effort required for adequate testing. Mature test automation environment will allow testing at the touch of a button with tests run overnight when the devices would have been otherwise idle. Another advantage of automating testing is that these tests are repeatable with exactly the same inputs and environment conditions, which is difficult in manual testing. This allows smallest of maintenance changes to be tested with minimal effort. Thirdly it eliminates the need of many menial chores.

One of the drawbacks of automation is that it is expensive when compared to the manual testing. Hence in order to gain benefits from test automation we should select those tests, which are more repeatable in nature. So that once automation is in place the cost of running the tests becomes lesser. Maintenance of automated scripts could be very expensive if automation is not done skillfully at the beginning.

Basic benefits of Automation are:

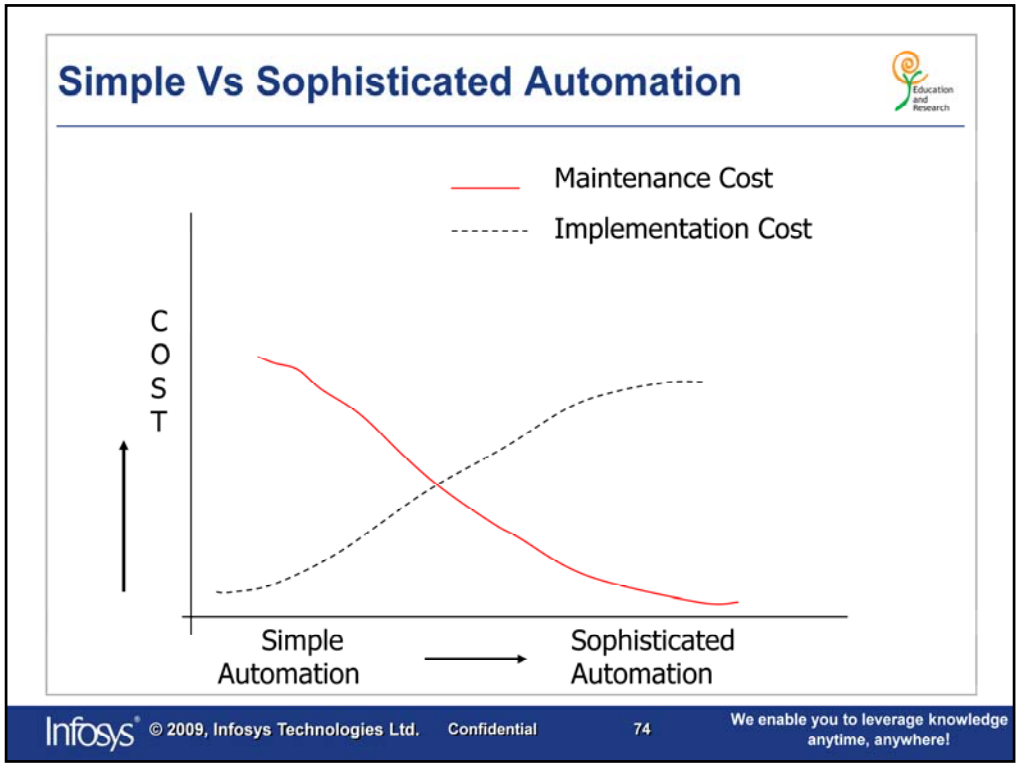
1. Run existing tests (regression) on new version of the software.
2. Run more tests more often
3. Perform certain tests, which would be difficult to do manually.
4. Better use of resources
5. Consistency and repeatability of tests.
6. Reuse of tests
7. Earlier time to market
8. Increased confidence

Hence through judicious use of automation efficient and repeatable testing is achieved with less effort giving better quality and productivity



Looking at the quantitative advantages of test automation from different parameters perspective: as you can see, automated test is highly economic after many runs (using it for repetitive testing). However, it's less economic than manual testing for the first run since it involves additional setup related effort.

Also note that manual test is more evolvable than automated tests since it is easier to come up with more techniques and methods in the manual mode of testing than the automated ones as the automated methods are more confined to the constraints of the tools.



Simple Vs Sophisticated Automation – As can be seen in the graph above, higher the magnitude of sophistication in automation, lesser will be the maintenance cost; however, the maintenance cost will be high if we go for simple automation methods. On the contrary, a simple automation technique (may be an in-house tool), will involve a lesser implementation cost initially but will need a higher cost as you go for higher sophistication.

Docs on automation in KShop: Search for them using keywords “Automation Testing”

Education and Research

We enable you to leverage knowledge anytime, anywhere!

Testing Tools

© 2009, Infosys Technologies Ltd.

Types of Testing Tools

- The software testing tools broadly of the following type:
  - Functional / regression
  - Load /Stress: Performance testing under high load / stress
  - Web Testing: testing the Web Sites
  - Data Generation: Generation of test data

Infosys © 2009, Infosys Technologies Ltd. Confidential 76 We enable you to leverage knowledge anytime, anywhere!

Tool support is available in testing in every stage of software development life cycle. There are various tools that can be used in various life cycle stages.

1. Test Design Tools: These tools help to derive test inputs and test data. Also referred to as test generator tools.

2. Static Analysis tools: These tools analyze code without actually executing the code.

3. Management tools: These include tools used to assist in planning, tracking and traceability

4. Test Execution & comparison tools: These tools enable tests to be run automatically and results to be compared to some baseline values. These are applicable to tests at all levels.

5. Dynamic Analysis and debugging tools: These tools asses the system while software is running and are typically used to detect memory leaks, CPU usage etc. Debugging tools are essentially not testing tools but are used as debugging that forms a part of testing.

6. Performance simulator tools: These tools measure the performance statistics of the software like load testing / stress testing, etc.

There are a large number of companies, which are in the business of Testing Tools (about 100). Most of the companies seem to be focused on a niche area with only a few tools. Some companies in the consulting business focus on providing expertise on Software Testing tools. These companies help other software developers and users to use the wide range of testing tools available in the market. There are however a few companies that have a wide spectrum of products that cater to a wide range of software testing needs.

Benefits

Reduces manpower and time

More coverage within the same testing time.

Testing products which are very difficult to test manually. Example, Load Tests

Can look inside the software for memory leaks, redundant code etc.

Infosys © 2009, Infosys Technologies Ltd. Confidential We enable you to leverage knowledge anytime, anywhere!

Test Tools (Client Server)

Category	Tool	Vendor
Source Code Testing	BoundsChecker	Numega
	Pure Coverage	Rational
	Purify	Rational
	Jprobe	Sitraka Software
	ATTOLCoverage	ATTOL Software
Functional Testing	QA Run	Compuware
	Win Runner	Mercury
	Team Test	Rational
	QA Center	Compuware
Performance Testing	OptimizeIT	VM Gear
	Load Runner	Mercury
	QA Load	Compuware

Infosys © 2009, Infosys Technologies Ltd. Confidential 78 We enable you to leverage knowledge anytime, anywhere!

A cursory glance of various discussion forums and other resources on the web clearly brings out the fact the industry leaders in this segment are Mercury Interactive, Rational Software, Compuware & Segue Software. These vendors provide separate products for each type of testing.

Apart from these vendors, firms such as IBM, Sun & Microsoft also have Testing Tools, which are for specific purposes. IBM has recently launched a product Fault Analyzer for OS/390. Compuware's QABatch enables MVS batch testing. Sun's range of products is primarily for testing Java applications. Another popular tool for testing java applications is Jsoft from Parasoft. Microsoft product WAST - Web Application Stress Testing – is a tool to test web applications.

Test Tools (Web Applications)

Category	Tool	Vendor
Functional Testing	Rational Robot	Rational
	Silk Test	Segue Software
Performance Testing	Silk Performer	Segue Software
	Web Load	Radview
Link & HTML Testing	NetMechanic	Monte Sano
	Linkbot	Tetranet
	Site Inspector	
	Doctor HTML	Imageware

Infosys © 2009, Infosys Technologies Ltd. Confidential

We enable you to leverage knowledge anytime, anywhere!

Test Tools (Network & Security)

Category	Tool	Vendor
Security & Communication	SNMP Test Suite	Interworking Labs Inc.
	Sniffer Pro	Network Associates Inc.
	Lophtcrack	
	Saint, Satan, Sara	SwTech
	TCP Wrappers	
	TCP Dump	

Infosys © 2009, Infosys Technologies Ltd. Confidential

We enable you to leverage knowledge anytime, anywhere!

Test Planning/Strategizing

- Includes –
  - Testing Objectives and Goals
  - Test Strategy/Approach based on customer priorities
  - Test Environment (Hardware, Software, Network, Communication etc.)
  - Features to test with priority/criticality
  - Test Deliverables
  - Test Procedure – Activities and tools
  - Test Entry and Exit criteria
  - Test Organization and Scheduling
  - Testing Resources and Infrastructure
  - Test Measurements/Metrics

Infosys © 2009, Infosys Technologies Ltd. Confidential 81 We enable you to leverage knowledge anytime, anywhere!

Test Planning / Strategy involves the following activities:

•First and foremost activity while strategizing for testing is to identify its objectives and goals. What are we going to test, why, how are we going to test them, when and by whom – these are basic questions that need to be addressed. What are goals – identify probable no. of defects that could be detected in each stage, review effectiveness, test effectiveness, etc.

•What are different test scenarios that need to be tested (based on customer's priorities); how many test-cases to be created for various stages. What are the different testing techniques that need to be adopted based on various scenarios / paths. What are the main features that need to be tested based on their priority and/or criticality.

•During the initial phase, need to decide on any investment on test environment – hardware (servers, desktops or other kind of hardware), software (testing tools, 3rd part software, etc.), network infrastructure, etc.

•Identify different deliverables (bug reports, test logs, test-cases, status reports, etc.) and detailed test procedures that need to be done during testing (apart from whatever is mentioned in test-cases) and tools related activities

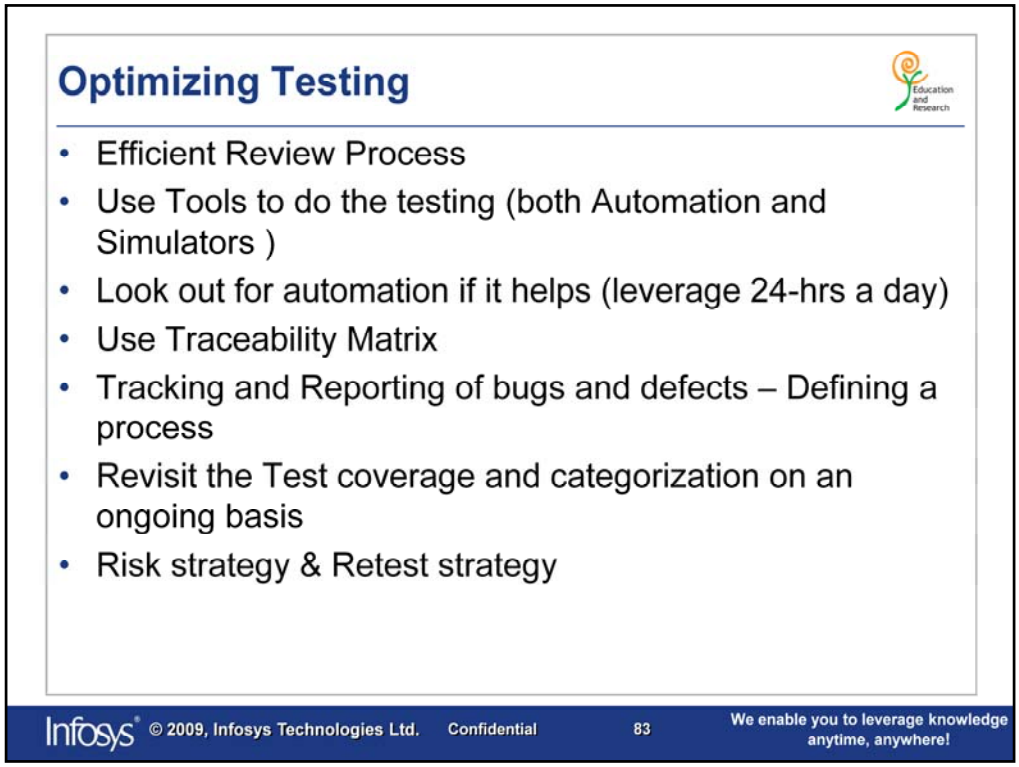
•Each test will have an entry and exit criteria – need to identify these clearly

•Other test management related activities like organizing the testing, scheduling, resources, etc.

•At the end of testing, planning for collecting different test metrics and planning for a DP (Defect Prevention) analysis

Test Planning / Strategizing

- Benefits
  - Sets clear and common objectives
  - Helps prioritize tests
  - Facilitates Technical tasks
  - Helps improve coverage
  - Provides structure to activities
  - Improves communication
  - Streamlines tasks, roles and responsibilities
  - Improves test efficiency
  - Improves test measurability



Here are a few tips to optimize the overall testing process:

•Efficient Review Process: Have a robust review process in place. Can think of novel ideas to reviewing like group review with brainstorming ideas, usage of rigorous checklists, etc.

•Use Tools to do the testing (both Automation and Simulators): Appropriate usage of automation tools help reduce the testing effort. Note that tools need not always be bought from a 3rd party; it can also be developed in-house

•Look out for automation if it helps (leverage 24-hrs a day)

•Use Traceability Matrix: A properly maintained traceability matrix is one very good artifact to ensure that no requirement or design element has been left out for testing

•Tracking and Reporting of bugs and defects – Defining a process: A rigorous process to ensure proper tracking and closure of defects so that defects get fixed on-time.

•Revisit the Test coverage and categorization on an ongoing basis

•Risk strategy & Retest strategy

For more tips on testing, refer to

•KShop with keywords “Testing Tips”. It fetches many useful documents

•http://www.sqatester.com/testingtips/index.htm

Defect Life Cycle

© 2009, Infosys Technologies Ltd.

What is Defect?

–**Defect (or Fault)** is a result of an entry of erroneous information into software.

–A software defect can be regarded as any failure to address end-user requirements.

Common defects include missed or misunderstood requirements and errors in design, functional logic, data relationships, process timing, validity checking, coding, etc.

Defects are commonly categorized by severity, and the numbers in each category are used for planning.

Infosys © 2009, Infosys Technologies Ltd. Confidential We enable you to leverage knowledge anytime, anywhere!

The defect management approach is based on counting and managing defects.

More mature software development organizations use tools such as defect leakage metrics (for counting the numbers of defects that pass through development phases prior to detection)

Defect Life Cycle

- Defect life cycle starts when a tester finds a discrepancy in the application under test.
- The Defect life cycle flow is as follows:

Properties of a defect

A defect is raised, it should have the following information associated with it.

- **Defect_ID**
- **Test Description**
- **Severity**
- **Priority**
- **Status**
- **Date Raised**
- **Date Fixed**
- **Owner**

Infosys © 2009, Infosys Technologies Ltd. Confidential 87 We enable you to leverage knowledge anytime, anywhere!

Defect_ID – Unique identification for the defect which will help in easily tracking the defect

Test Description – Detailed description of the defect

Severity – Severity of the defect according to the tester. The values it can hold are Critical, Cosmetic, Major, Minor

Priority – Urgency at which the defect needs a fix. It can hold the values Medium, Low, High

Status – Status of the defect (Can be Open, Closed, Assigned etc)

Date Raised – Date when the defect is raised

Date Fixed – Date

Owner – Owner of **this** state of the defect.

Cost of Quality

- Cost of quality comprises of three components
  - Prevention Cost
  - Appraisal Cost
  - Failure Cost
- Total cost of quality = Prevention cost + appraisal cost + failure cost
- Prevention cost - Cost incurred in preventing an error from occurring.
- Appraisal cost – Cost incurred for testing and review.
- Failure cost – Cost incurred in removing all the defects in the software.

Infosys © 2009, Infosys Technologies Ltd. Confidential We enable you to leverage knowledge anytime, anywhere!

Prevention Cost:

E.g. Carrying out a thorough requirements analysis, design, process definition for maintaining quality. This cost increases with the increased degree of Quality.

Appraisal Cost:

As this cost increases, the degree of quality increases only up to a limit. After certain limit there will not be considerable increase in the degree of quality with respect to the Appraisal cost

Failure Cost:

This is the costliest of all the costs and as the Failure cost increases, the increase in the degree of quality reduces.

Defect Management Tools

Features of a defect management tool

- User friendly
- Email notification
- File attachment
- Audit Trail
- Configuration Management
- Customizable Fields
- Metric Reports
- Remote Administering
- Report Cross-Referencing
- Security Implementation
- Web-Based Client

Defect Management Tools

- Popular tools available in the market are
  - Bugzilla - Open Source Bug Tracking System
  - HP Quality Center - Test management tool by HP, formerly Test Director by Mercury Interactive
  - ClearQuest - Defect tracking tool by IBM
  - Trac - Open source enhanced wiki and issue tracking system by Edgewall Software
  - JIRA - Defect tracking tool by Atlassian
  - Scarab - Open source issue tracking tool.

Infosys © 2009, Infosys Technologies Ltd. Confidential We enable you to leverage knowledge anytime, anywhere!

Different stakeholders get different set of advantages provided by the tool.

Customer - Easy tracking of the defects.
Project Manager - For getting quick access to the project statistics
Tester - Efficient reporting and tracking of defects
Developer - Defect fixes can be recorded accurately

Defect Management Tools – Mozilla 's Bugzilla

- A bug-tracking product from Mozilla.org. It is an enterprise-class piece of software that tracks millions of bugs and issues for hundreds of organizations around the world.
- An account is created via "Open a new Bugzilla account" link. This is essential in order to use Bugzilla.
- Defects can be accessed through the 'Bug' screen.
- Features offered for complete defect management include
  - Adding defects
  - Managing defects (allocation, change status etc.)
  - Analysis of defect data

Infosys © 2009, Infosys Technologies Ltd. Confidential We enable you to leverage knowledge anytime, anywhere!

Defects-Add, Modify, Delete defect etc. can be managed.

Searching for Bugs-Functionality to search for any information related to the defects logged within the system

Bug Lists- For listing similar kind of bugs.

Report- Is a view of the current state of the bug database.

Charts- Is a view of the state of the bug database over time.

Recap

- Integration Testing
- System Testing
- Test Automation
- Testing Tools
- Defect Life Cycle
- Defect Management Tools

Infosys © 2009, Infosys Technologies Ltd. Confidential

We enable you to leverage knowledge anytime, anywhere!

Education and Research

We enable you to leverage knowledge anytime, anywhere!

Thank You

© 2009, Infosys Technologies Ltd.