**Lecture 1**

**Agile**: Agile is an iterative development methodology, where both development and testing activities are concurrent. Testing is Testing is not a separate phase; Coding and Testing are done interactively and incrementally, resulting in quality end product, which the meets customer requirements. Further, continuous integration results in early defect removal and hence time, effort and cost savings.

**Agile Manifesto:**

1. Individuals and interactions over processes and tools.
2. Working software over comprehensive documentation.
3. Customer collaboration over contract negotiation.
4. Responding to change over following a plan.

**XP:**

• Most prominent Agile Software development method

• Prescribes a set of daily stakeholder practices

• “Extreme” levels of practicing leads to more responsive software.

• Changes are more realistic, natural, inescapable.

Practices

➢If testing is good, let everybody test all the time

➢If code reviews are good, review all the time

➢If design is good, refactor all the time

➢If integration testing is good, integrate all the time

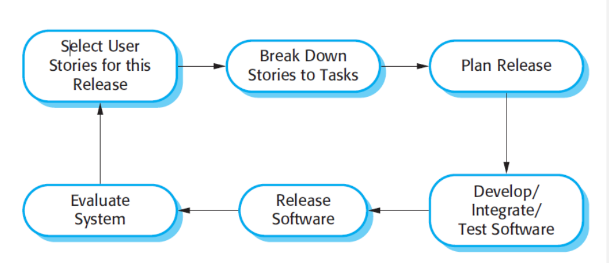
➢If simplicity is good, do the simplest thing that could possibly work

➢If short iterations are good, make them really, really short

XP Practices:

Incremental planning, small releases, Simple design, Test-first dev, Refactoring.

Pair programming, Collective ownership, Continuous integration, Sustainable pace, On-site customer.

**XP release cycle:**

**Scrum:** Basic Parts:

Role: 1.Scrum Master 2.Product Owner 3.Team

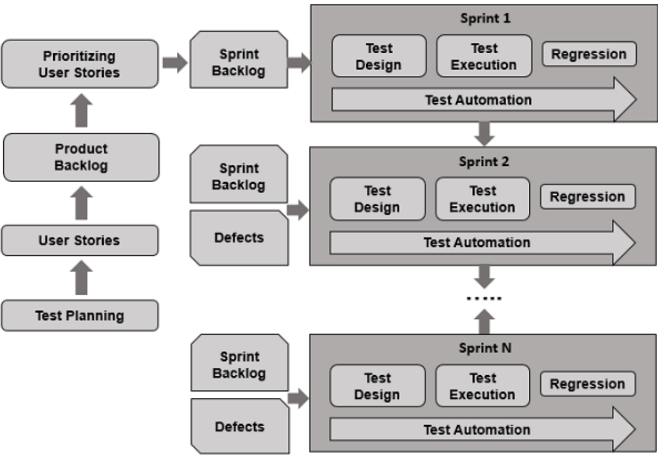
Artifacts: 1.Product backlog 2.Sprint backlog 3.Burn-down charts

Meetings: 1.Sprint planning 2. Sprint review 3.Sprint retrospectives 4.Daily scrum

**Agile Testing:** Agile Testing involves all members of the project team, with special expertise contributed by testers. Testing is not a separate phase and is interwoven with all the development phases such as requirements, design and coding and test case generation. Testing takes place simultaneously through the Development Life Cycle. It covers all the levels of testing and all types of testing.

Activities:

|  |  |
| --- | --- |
| The Agile Testing Activities at Project Level are −  • Release Planning (Test Plan)  • For every Iteration,  • Agile Testing Activities during an Iteration  • Regression Testing  • Release Activities (Test Related) | The Agile Testing Activities during an iteration include −  • Participating in iteration planning  • Estimating tasks from the view of testing  • Writing test cases using the feature descriptions  • Unit Testing  • Integration Testing  • Feature Testing  • Defect Fixing  • Integration Testing  • Acceptance Testing  • Status Reporting on Progress of Testing  • Defect Tracking |

**Sprint lifecycle:**

**Test case example:**

Input:

1. A number in mg representing a single dose of the drug.

2. A number representing the number of single doses per day.

Tests:

1. Test for inputs where the single dose is correct but the frequency is too high.

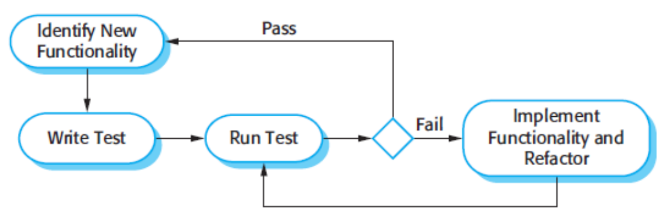
2. Test for inputs where the single dose is too high and too low.

3. Test for inputs where the single dose \* frequency is too high and too low.

4. Test for inputs where single dose \* frequency is in the permitted range.

Output: OK or error message indicating that the dose is outside the safe range.

**Test-driven development (TDD):**

Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development. Writing tests before code clarifies the requirements to be implemented. Tests are written before code and ‘passing’ the tests is the critical driver of development. You develop code incrementally, along with a test for that increment. You don’t move on to the next increment until the code that you have developed passes its test.

**TDD Activities:**

Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.

Write a test for this functionality and implement this as an automated test.

Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.

Implement the functionality and re-run the test.

Once all tests run successfully, you move on to implementing the next chunk of functionality.

**Benefits of TDD:**

Code coverage, Regression testing, Simplified debugging, System documentation.

**Sprint ZERO:**

Sprint Zero involves preparation activities before the first sprint. A tester needs to collaborate with the team on the following activities −

• Establishing a business case for the project

• Establish the boundary conditions and the project scope

• Outline the key requirements and use cases that will drive the design trade-offs

• Outline one or more candidate architectures

• Identifying the risk

• Cost estimation and prepare a preliminary project

**Agile Testing Practices:**

An Agile tester needs to adapt Agile practices for testing in an agile project.

• Pairing • Incremental Test • Mind Mapping.

**Lecture 2**

**V&V:**

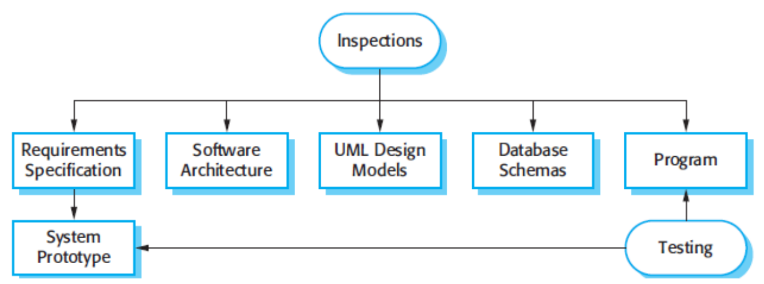
Verification refers to the set of tasks that ensure that software correctly implements a specific function.

Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

• Verification: "Are we building the product right?" • Validation: "Are we building the right product?"

**Who tests the software:**

|  |  |
| --- | --- |
| Developer: Understands the system. But, will test gently and is driven by delivery. | Independent tester: Must learn about the system, but, will attempt to break it and, is driven by quality |

**Inspections and testing**

• Software inspections: Concerned with analysis of the static system representation to discover problems. (static verification).

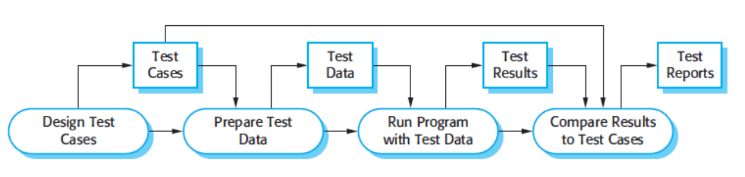
• Software testing Concerned with exercising and observing product behavior (dynamic verification).

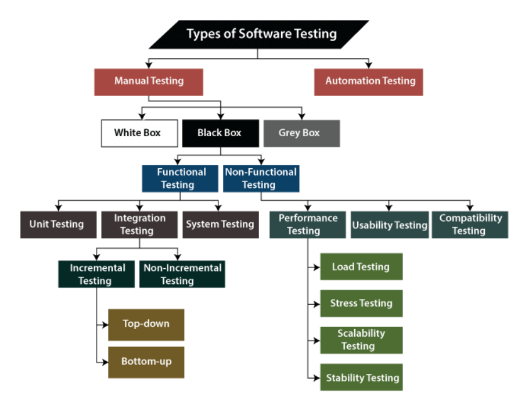
**Advantages of inspections**

• During testing, errors can mask (hide) other errors. Because inspection is a static process, you don’t have to be concerned with interactions between errors.

• Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.

• As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

**Software testing process model:**

****

**Types of Testing:**

**Stages of Testing**

L1: Unit testing: Test the individual component.

L2: Integration testing: Test integrated component.

L3: System testing: Test the entire system.

L4: Test the final system.

**Unit testing:**

Unit testing is the process of testing individual components in isolation.

• It is a defect testing process.

Units may be:

• Individual functions or methods within an object

• Object classes with several attributes and methods

• Composite components with defined interfaces used to access their functionality.

**Integration Testing**

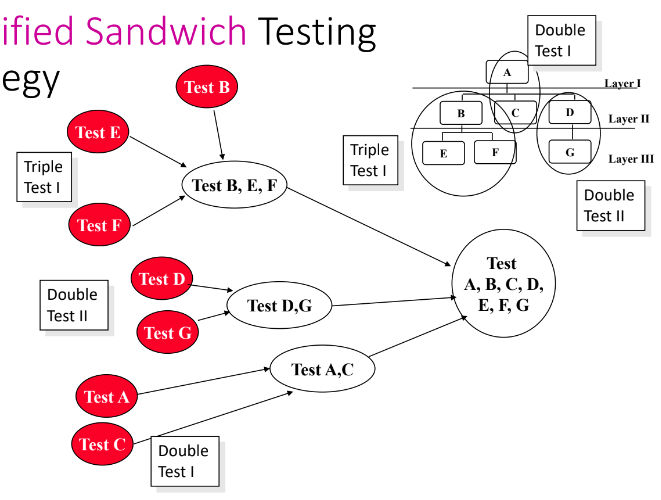
Once we are successfully implementing the unit testing, we will go integration testing. It is the second level of functional testing, where we test the data flow between dependent modules or interface between two features is called integration testing.

• The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design.

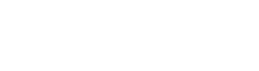
• The order in which the subsystems are selected for testing and integration determines the testing strategy

Types of Integration Testing

|  |  |
| --- | --- |
| • Incremental Testing  • Bottom up integration  • Top down integration | • Non-Incremental Testing  • Big bang integration |

**Modified Sandwich Testing Strategy**

Test in parallel:



• Middle layer with drivers and stubs

• Top layer with stubs

• Bottom layer with drivers

Test in parallel:

• Top layer accessing middle layer (top layer replaces drivers)

• Bottom accessed by middle layer (bottom layer replaces stubs)

**Driver and Stub:**

Stubs: Stubs are basically known as a “called programs” and are used in the Top-down integration testing. Stubs are used when lower-level of modules are missing or in a partially developed phase, and we want to test the main module.

Driver: D rivers are the “calling program” and are used in bottom-up integration testing. Drivers are used when higher-level of modules are missing or in a partially developed phase, and we want to test the lower(sub)- module.

**Regression Testing**

• Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

• Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.

• Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

• Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

**Smoke Testing**

• A common approach for creating “daily builds” for product software.

• Smoke testing is the initial testing process exercised to check whether the software under test is ready/stable for further testing.

• The term ‘Smoke Testing‘ comes from the hardware testing, in the hardware testing initial pass is done to check if it did not catch the fire or smoke in the initial switch on.

**Non-Functional Testing**

|  |  |
| --- | --- |
| • Stress Testing: Stress limits of system (maximum # of users,  peak demands, extended operation)  • Volume testing: Test what happens if large amounts of data  are handled  • Configuration testing: Test the various software and hardware configurations  • Compatibility test: Test backward compatibility with existing systems  • Security testing: Try to violate security requirements | • Timing testing: Evaluate response times and time to perform  a function.  • Environmental test: Test tolerances for heat, humidity, motion, portability  • Quality testing: Test reliability, maintain- ability & availability of the system  • Recovery testing: Tests system’s response to presence of errors or loss of data.  • Human factors testing: Tests user interface with user |

**User testing**

User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.

Types of user testing

• Alpha testing: Users of the software work with the development team to test the software at the developer’s site.

• Beta testing: A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.

• Acceptance testing: Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

**Lecture 3:**

**Software Requirements**

A requirement is a feature that the system must have or a constraint that it must satisfy to be accepted by the client.

Types of requirement

• User requirements: Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.

• System requirements: A structured document setting out detailed descriptions of the system’s functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

**Sample of a Test Case**

Title: Login Page – Authenticate Successfully on gmail.com

Description: A registered user should be able to successfully login at gmail.com.

• Precondition: the user must already be registered with an email address and password.

• Assumption: a supported browser is being used.

Test Steps:

• Navigate to gmail.com

• In the ’email’ field, enter the email of the registered user.

• Click the ‘Next’ button.

• Enter the password of the registered user

• Click ‘Sign In’

Expected Result: A page displaying the gmail user’s inbox should load, showing any new message at the top of the page.

**Equivalence Partitioning**

Equivalence partitioning is a method of deriving test cases when there is a large number of input data ranges.

Equivalence partitioning helps to cut down exponentially on the number of cases required to test system. It is an attempt to get good test coverage, to find more errors with a smaller number of test cases.

In this method, classes of input conditions called equivalence classes are identified, with same kind of processing. Therefore, it leads to the generation of the same output.

Example

• Requirement: Employee ID should be 6 digits.

• System inputs have to be partitioned into ‘equivalence class.’ Input is a 6-digit integer, which can be from 100,000 to 999,999.

• Valid and Invalid equivalence partitions:

• Valid: 100000 to 999999

• Invalid: < 100000

• Invalid: > 999999

**Boundary Value Analysis**

• Boundary Value Analysis or BVA is a black-box test design technique in which test cases are designed based on testing the boundaries between partitions for both valid and invalid boundaries.

• Boundary value is an input or output value, which is on the edge of an equivalence partition or at the smallest incremental distance on either side of an edge.

Example

• A program accepts a number in the range, -100 to +100. Here, the three sets of valid equivalent partitions are:

• First class: Negative range: -100 to -1

• For the lower boundary, the values that need to be checked include -101, -100, -99,-2, and -1.

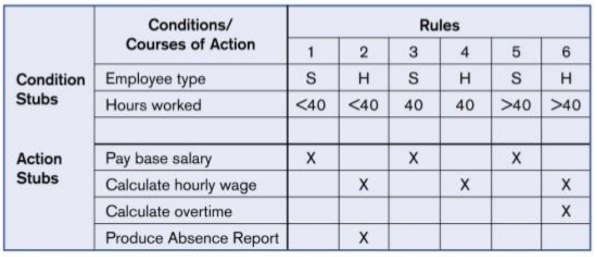
• For the upper boundary, the values that need to be checked include -1, 0, and +1.

• Second class: Zero

• Third class: Positive range: Between 1 and 100

• For the positive range, the lower boundary is 1, and the upper boundary is 100. Therefore, the values that need to be checked include 0, 1, 2, 99, 100, and 101.

• In this example, note that there is some duplication in values for the test data. When the duplicate inputs are removed, then, the final input conditions are: -101, -100, -99, -1, 0, 1, 99, 100, and 101.

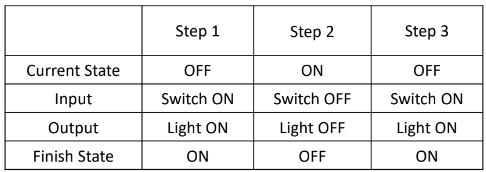
**Decision Table**

• A decision table displays a combination of inputs and/or stimuli, which is termed as ‘causes,’ with their associated outputs and/or actions, which are termed as ‘effects.’

• Permutations and combinations of these inputs are used to design test cases and are called decision table. This technique is also referred to as cause-effect' table as there is an associated logical diagramming technique called cause-effect graphing,' which is sometimes used to help derive the decision table.

**State Transition Testing**

State transition testing is a black box test design technique in which test cases are designed to execute valid and invalid state transitions.

Consider the switch is in ‘off’ state. An event occurs when the switch is on, and the action of this event is ‘light is in on state.’ Moving the switch from off state to on state is a transition.

**Steps:**

Consider the current state and provide input to see the output and finish state for 3 different steps.

• Step 1: In step 1, the current state is OFF, and input is Switch ON. In this case, the output is Light ON and finish state is ON.

• Step 2: In step 2, the current state is ON, and input is Switch OFF. In this case, the output is Light OFF

and finish state is OFF.

• Step 3: In step 3, the current state is OFF, and input is Switch ON. In this case, the output is Light ON

and finish state is ON.

**Use Case Testing**

• A use case is a description of a particular use of the system by a user. Each use case describes the user interactions with the system, to achieve a specific task.

• Use case testing is a technique that helps to identify test cases that exercise the system on a transaction by transaction basis from start to finish. It helps in designing acceptance testing and identifying integration defects and defects from common real-life scenarios.

Example

An online bill payment that has three main use cases. They are:

• Login

• Online Payment

• Logout

These use cases define all the conditions regarding the requirement for the logical part of the system or application. These use cases are inter-related. Before making an online payment or logging out of the application, the user needs to log into the application.

Test cases are derived from use cases to represent end-to-end transactions in the system.

**Coverage of Structure-Based Testing Techniques**

A Coverage is defined as the number of items covered in testing divided by the total number of items. Coverage, therefore, defines the extent of code that has been tested out of the total code in the system.

Coverage Types

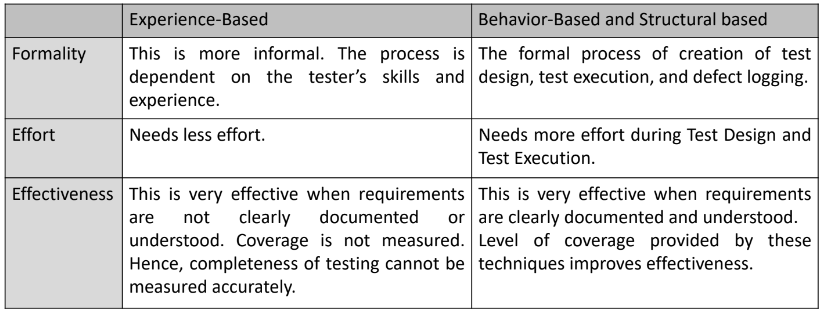
|  |  |
| --- | --- |
| • Statement Coverage  • Decision Coverage | • Condition Coverage  • Multiple Condition Coverage |

**Experience-Based Testing Techniques – Types**

• Exploratory Testing

• Fault Attack/ Error Guessing

• Random Testing

**Experience-Based Techniques vs. Other techniques**

**Lecture 4**

**Defects**

When expected result of test mismatched with actual result of the application, this mismatch is called defect.

Following are the common types of defects that occur during development:

|  |  |  |
| --- | --- | --- |
| • Arithmetic Defects  • Logical Defects | • Syntax Defects  • Multithreading Defects | • Interface Defects  • Performance Defects |

Defect Classification

Defects are classified in two perspectives:

• From the QA team perspective as Priority

• From the development perspective as Severity (complexity of code to fix it).

**Priority**

Priority is defined as the order in which the defects should be resolved.

The Priority status is set based on the requirements of the end users.

A Priority can be categorized in the following ways −

• Low − This defect can be fixed after the critical ones are fixed.

• Medium − The defect should be resolved in the subsequent builds.

• High − The defect must be resolved immediately because the defect affects the application to a considerable extent and the relevant modules cannot be used until it is fixed.

• Urgent − The defect must be resolved immediately because the defect affects the application or the product severely and the product cannot be used until it has been fixed.

**Severity**

Severity is defined as the impishness of defect on the application and complexity of code to fix it from development perspective. It is related to the development aspect of the product.

Severity can be decided based on how bad/crucial is the defect for the system.

Severity can be categorized in the following ways −

• Critical /Severity 1 − Defect impacts most crucial functionality of Application and the QA team cannot continue with the validation of application under test without fixing it. For example, App/Product crash frequently.

• Major / Severity 2 − Defect impacts a functional module; the QA team cannot test that particular module but continue with the validation of other modules. For example, flight reservation is not working.

• Medium / Severity 3 − Defect has issue with single screen or related to a single function, but the system is still functioning. The defect here does not block any functionality. For example, Ticket# is a representation which does not follow proper alpha numeric characters like the first five characters and the last five as numeric.

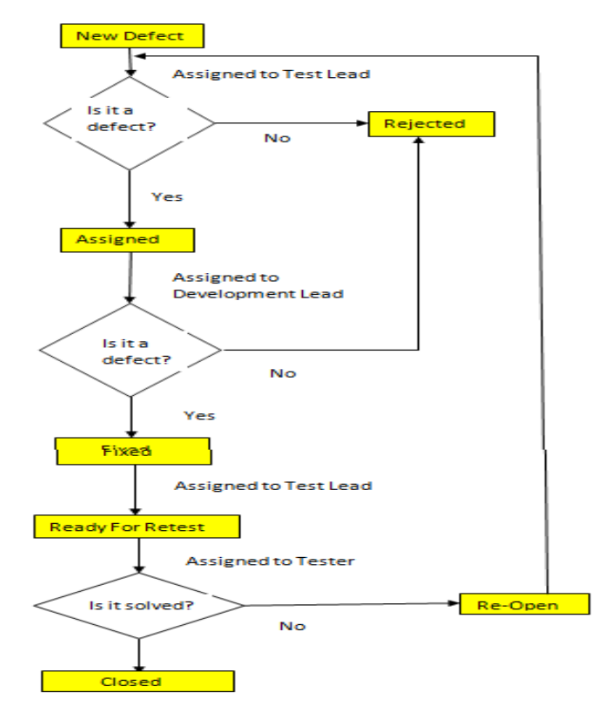
• Low / Severity 4 − It does not impact the functionality. It may be a cosmetic defect, UI inconsistency for a field or a suggestion to improve the end user experience from the UI side. For example, the background color of the Submit button does not match with that of the Save button.

**Defect tracking**

Defect tracking is an important process in software engineering as complex and business critical systems have hundreds of defects. One of the challenging factors is Managing, evaluating and prioritizing these defects. The number of defects gets multiplied over a period of time and to effectively manage them, defect tracking system is used to make the job easier.

Defects are tracked based on various parameters such as:

|  |  |  |
| --- | --- | --- |
| • Defect Id  • Priority  • Severity | • Created by  • Created Date  • Assigned to | • Resolved Date  • Resolved By  • Status |

**Defect Life Cycle**

Defect life cycle, also known as Bug Life cycle is the journey of a defect cycle, which a defect goes through during its lifetime.

It varies from organization to organization and also from project to project as it is governed by the software testing process and also depends upon the tools used.

**Defect Life Cycle States:**

• New - Potential defect that is raised and yet to be validated.

• Assigned - Assigned against a development team to address it but not yet resolved.

• Active - The Defect is being addressed by the developer and investigation is under progress. At this stage there are two possible outcomes; viz - Deferred or Rejected.

• Test - The Defect is fixed and ready for testing.

• Verified - The Defect that is retested and the test has been verified by QA.

• Closed - The final state of the defect that can be closed after the QA retesting or can be closed if the defect is duplicate or considered as NOT a defect.

• Reopened - When the defect is NOT fixed, QA reopens/reactivates the defect.

• Deferred - When a defect cannot be addressed in that particular cycle it is deferred to future release.

• Rejected - A defect can be rejected for any of the 3 reasons; viz - duplicate defect, NOT a Defect, Non-Reproducible.

**Lecture 5**

**Quality plans**

Quality plan structure

|  |  |
| --- | --- |
| ▪ Product introduction;  ▪ Product plans;  ▪ Process descriptions | ▪ Quality goals;  ▪ Risks and risk management. |

Quality plans should be short, succinct documents, If they are too long, no-one will read them.

**Quality conflicts**

It is not possible for any system to be optimized for all of these attributes – for example, improving robustness may lead to loss of performance.

The quality plan should therefore define the most important quality attributes for the software that is being developed.

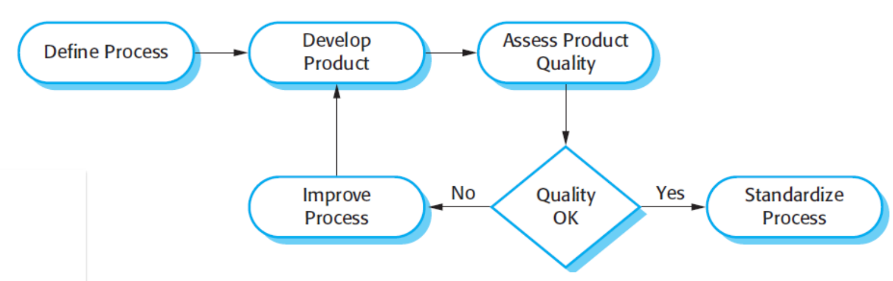
The plan should also include a definition of the quality assessment process, an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

**Process and product quality**

The quality of a developed product is influenced by the quality of the production process.

This is important in software development as some product quality attributes are hard to assess.

However, there is a very complex and poorly understood relationship between software processes and product quality.

Process-based quality:

**Software standards**

Standards define the required attributes of a product or process. They play an important role in quality management.

Standards may be international, national, organizational or project standards.

Product standards define characteristics that all software components should exhibit e.g. a common programming style.

Process standards define how the software process should be enacted.

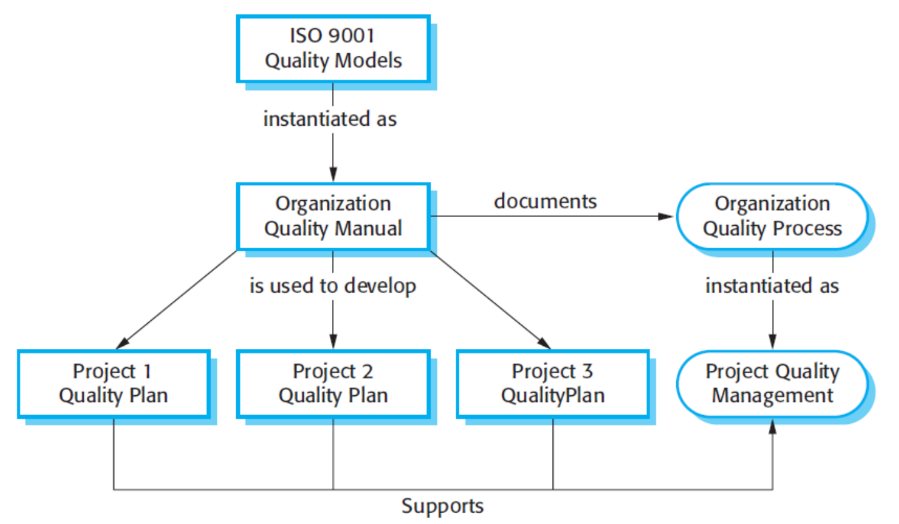
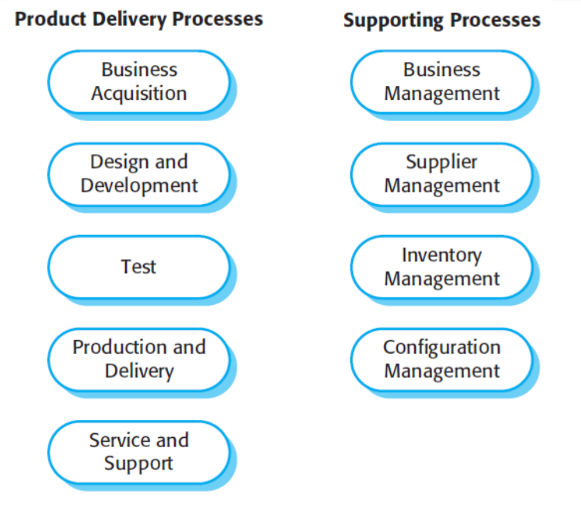
**ISO 9001 standards framework**

An international set of standards that can be used as a basis for developing quality management systems.

ISO 9001, the most general of these standards, applies to organizations that design, develop and maintain products, including software.

The ISO 9001 standard is a framework for developing software standards.

|  |  |
| --- | --- |
| **ISO 9001 core processes** | **ISO 9001 and quality management** |



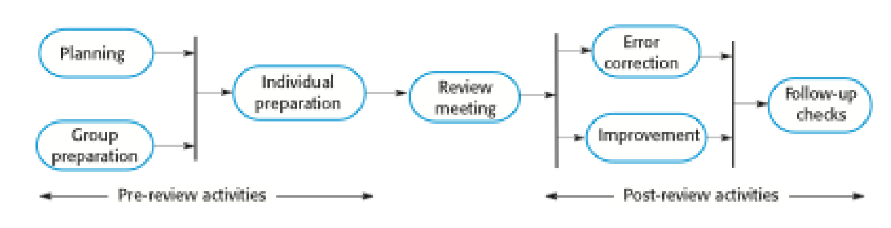
**ISO 9001 certification**

Quality standards and procedures should be documented in an organizational quality manual.

An external body may certify that an organization’s quality manual conforms to ISO 9000 standards.

Some customers require suppliers to be ISO 9000 certified although the need for flexibility here is increasingly recognized.

**The software review process**



**Program inspections**

These are peer reviews where engineers examine the source of a system with the aim of discovering anomalies and defects.

Inspections do not require execution of a system so may be used before implementation.

They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).

They have been shown to be an effective technique for discovering program errors.

**Inspection checklists**

Checklist of common errors should be used to drive the inspection.

Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.

In general, the 'weaker' the type checking, the larger the checklist.

**Inspection checklist – Example**

Write down faults with example

|  |  |
| --- | --- |
| Fault class | Inspection check |
| Data faults | • Are all program variables initialized before their values are used?  • Have all constants been named?  • Should the upper bound of arrays be equal to the size of the array or Size -1?  • If character strings are used, is a delimiter explicitly assigned?  • Is there any possibility of buffer overflow? |
| Control faults | • For each conditional statement, is the condition correct?  • Is each loop certain to terminate?  • Are compound statements correctly bracketed?  • In case statements, are all possible cases accounted for?  • If a break is required after each case in case statements, has it been included? |
| Input/output faults | • Are all input variables used?  • Are all output variables assigning a value before they are output?  • Can unexpectedly inputs cause corruption? |
| Interface faults | • Do all function and method calls have the correct number of parameters?  • Do formal and actual parameter types match?  • Are the parameters in the right order?  • If components access shared memory, do they have the same model of the shared memory structure? |
| Storage management  faults | If a linked structure is modified, have all links been correctly reassigned?  • If dynamic storage is used, has space been allocated correctly?  • Is space explicitly deallocated after it is no longer required? |
| Exception management  faults | • Have all possible error conditions been taken into account? |

**Agile methods and inspections**

Agile processes rarely use formal inspection or peer review processes.

Rather, they rely on team members cooperating to check each other’s code, and informal guidelines, such as ‘check before check-in’, which suggest that programmers should check their own code.

Extreme programming practitioners argue that pair programming is an effective substitute for inspection as this is, in effect, a continual inspection process.

Two people look at every line of code and check it before it is accepted.

**Lecture 7**

**Reliability**

A simple measure of reliability is mean-time-between-failure (MTBF), where

MTBF = MTTF + MTTR

Where MTTF= mean-time-to-failure and MTTR= mean-time-to-repair

Availability = [MTTF/(MTTF + MTTR)] x 100%

**PUM**

PUM = Total Problems that customers reported (true defect and non-defect-oriented problems) for a time period + Total

number of license months of the software during the period

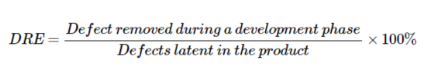
Where, Number of license-month of the software = Number of install license of the software × Number of months in the

calculation period

**Defect Density** DD = Total defects/Size (FP)

**BMI** BMI = (No. of problems closed during the month / No. of problems arrived during the month) × 100%

**Percent Delinquent Fixes:** Percent delinquent fixes = (No. fixes that exceeded the response time criteria of severity level / No. of fixes delivered in a specified time) × 100%

**Defect removal effectiveness**