**LACTURE NOTES**

DEPARTMENT OF COMPUTER SCIENCE

aMERICAN INTERNATIONAL UNIVERSITY – bANGLADESH (aIUB)

**CSC4133 Software quality and testing**

# **COURSE OUTLINE**

**I - Course Code and Title:** CSC 4133 (Software Quality and Testing) **II - Credit:** 3 credit hours (3 hours of theory per week)   
**III- Nature:** Major Course for B.Sc. in CSSE and SE, Elective course for CS and CSE **IV - Prerequisite:** CSC3114 (Software Engineering)

**V - Course Description:**

This course provides a comprehensive study of software quality assurance and testing. Topics include levels and techniques of testing, verification and validation, quality assurance processes and techniques, ISO 9126 and CMMI models. The course focuses on real-life software quality assurance and testing activities as well. The course covers both manual and automated testing techniques with an introduction to functional and regression testing tools like Selenium.

**VI – Objectives:**

**At the completion of the course, the students will be able to understand-**

* Various approaches, techniques, technologies, and methodologies used in software testing and quality assurance.
* The quality assurance process and its role in software development.
* A variety of testing techniques, methods, and tools used in real life.
* The impact of ISO 9126 and the capability maturity model Integration (CMMI) on software quality assurance
* How to write Test Plan
* How to write Test Cases and execute them, and preparing bug-report
* About the different types of software testing tools used in the IT firms
* How to use Selenium to develop test cases and test suite, how to execute test cases and identify bugs

**Table of Contents**

[**COURSE OUTLINE** 1](#_Toc38653894)

[**Chapter 1: Basics of Software Quality Assurance** 5](#_Toc38653895)

[**1.1** **What is Software** 5](#_Toc38653896)

[**1.2** **What is Software Quality** 6](#_Toc38653897)

[**1.3** **Challenges in Software Project** 6](#_Toc38653898)

[**1.4** **Example of Software Defects** 6](#_Toc38653899)

[**1.5** **Software Testing in Project Management** 7](#_Toc38653900)

[**1.6** **Role of Testing in Project Management** 8](#_Toc38653901)

[**1.7** **Software Quality Assurance (SQA)** 9](#_Toc38653902)

[**1.8** **Software Quality Control (SQC)** 9](#_Toc38653903)

[**1.9** **Validation and Verification** 9](#_Toc38653904)

[**1.10** **Difficulties on Quality Expectations** 10](#_Toc38653905)

[**1.11** **Basics of Software Quality Engineering (SQE) and SQE Activities** 10](#_Toc38653906)

[**1.12** **Error, Fault, Failure, and Defect** 10](#_Toc38653907)

[**1.13** **Testing Activities and Complete Testing** 11](#_Toc38653908)

[**Chapter 2: Software Quality** 13](#_Toc38653909)

[**2.1** **Quality Perspective** 13](#_Toc38653910)

[**2.2** **Views in Software Quality** 14](#_Toc38653911)

[**2.3** **Why Measure Software Quality?** 15](#_Toc38653912)

[**2.4** **Software Quality Factor** 15](#_Toc38653913)

[**2.5** **Software Quality Criteria** 16](#_Toc38653914)

[**2.6** **ISO-9126 Quality Framework** 17](#_Toc38653915)

[**Chapter 3: Maturity Models** 17](#_Toc38653916)

[**3.1 Software Process** 17](#_Toc38653917)

[**3.2 Capability Maturity Model (CMM)** 19](#_Toc38653918)

[**3.3 Test Process Improvement (TPI)** 21](#_Toc38653919)

[**3.4 Test Maturity Model (TMM)** 23](#_Toc38653920)

[**Chapter 4: Quality Assurance** 25](#_Toc38653921)

[**4.1 Defect Prevention** 26](#_Toc38653922)

[**4.2 Defect Reduction** 26](#_Toc38653923)

[**4.3 Defect Containment** 27](#_Toc38653924)

[**4.4 QA Activities in Software Development Process** 28](#_Toc38653925)

[**4.4 V&V in QA Software Process** 30](#_Toc38653926)

[**Chapter 5: Software Quality Engineering** 30](#_Toc38653927)

[**5.1 Quality Engineering: Activity and Process** 30](#_Toc38653928)

[**5.2 Pre-QA Activities** 31](#_Toc38653929)

[**5.3 In-QA Activities** 32](#_Toc38653930)

[**5.4 Post-QA Activities** 32](#_Toc38653931)

[**5.5** **Test Case, Test Plan and Test Suite** 32](#_Toc38653932)

[**Chapter 6: Software Testing** 33](#_Toc38653933)

[**6.1 Software Testing** 33](#_Toc38653934)

[**6.2 Principle of Software Testing** 33](#_Toc38653935)

[**6.3 Testing Levels** 34](#_Toc38653936)

[**6.4 White-Box and Black-Box Testing** 35](#_Toc38653937)

[**6.5 Adequacy of Testing** 36](#_Toc38653938)

[**Chapter 7: Testing Activities, Management, and Automation** 37](#_Toc38653939)

[**7.1 Testing Planning and Preparation** 37](#_Toc38653940)

[**7.2 Testing Suite Preparation and Management** 39](#_Toc38653941)

[**7.3 Testing teams: Organization and management** 40](#_Toc38653942)

[**7.4 Test Execution, Result Checking, and Measurement** 41](#_Toc38653943)

[**7.5 Test Automation** 41](#_Toc38653944)

[**Chapter 8: Unit Testing** 43](#_Toc38653945)

[**8.1 Concepts of Unit Testing** 43](#_Toc38653946)

[**8.2 Static Unit Testing** 45](#_Toc38653947)

[**8.3 Code Review Metrics** 49](#_Toc38653948)

[**8.4 Defect Prevention** 50](#_Toc38653949)

[**8.5 Dynamic Unit Testing** 50](#_Toc38653950)

[**8.6 Mutation Testing** 52](#_Toc38653951)

[**8.7 Debugging Techniques** 53](#_Toc38653952)

[**8.8 Unit Testing in Extreme Programming** 54](#_Toc38653953)

[**8.9 JUNIT: Framework for Unit Testing** 54](#_Toc38653954)

[**8.10 Tools for Unit Testing** 54](#_Toc38653955)

[**Chapter 9: Control Flow Testing** 55](#_Toc38653956)

[**9.1 Control Flow Testing Concepts** 56](#_Toc38653957)

[**9.2 Outline of Control Flow Testing** 56](#_Toc38653958)

[**9.3 Control Flow Graph** 57](#_Toc38653959)

[**9.4 Control-Flow Path Selection Criteria** 57](#_Toc38653960)

[**9.5 Containing Infeasible Path** 59](#_Toc38653961)

[**Chapter 10: Data Flow Testing** 59](#_Toc38653962)

[**10.1 Data Flow Testing Concepts** 59](#_Toc38653963)

[**10.2 Data Flow Anomaly** 60](#_Toc38653964)

[**10.3 Data Flow Graph** 61](#_Toc38653965)

[**10.4** **Comparison of Testing Techniques** 62](#_Toc38653966)

[**Chapter 11: Domain Testing** 62](#_Toc38653967)

[**11.1 Domain Testing Concepts** 62](#_Toc38653968)

[**11.2 Testing Domain Errors** 63](#_Toc38653969)

[**11.3 Source of Domain** 63](#_Toc38653970)

[**11.4 Type of Domain** 64](#_Toc38653971)

[**11.5 ON OFF Points** 65](#_Toc38653972)

[**Chapter 12: System Integration Testing** 66](#_Toc38653973)

[**12.1 Integration Testing Concepts** 66](#_Toc38653974)

[**12.2 Different Types of Interface** 67](#_Toc38653975)

[**12.3 Granularity of System Integration Testing** 70](#_Toc38653976)

[**12.4 System Integration Technique** 71](#_Toc38653977)

[**12.5 Test Plan for System Integration** 72](#_Toc38653978)

[**12.6 OFF-THE-SHELF Component Integration** 73](#_Toc38653979)

[**Chapter 13: System Testing** 73](#_Toc38653980)

[**13.1 System Testing Concepts** 73](#_Toc38653981)

[**13.2 Modelling Defects** 74](#_Toc38653982)

[**13.3 Metrics for Tracking System Test** 75](#_Toc38653983)

[**13.4 Metrics for Monitoring Test Execution** 75](#_Toc38653984)

[**13.5 Orthogonal Defect Classification** 76](#_Toc38653985)

[**13.6 Defect Casual Analysis (DCA)** 76](#_Toc38653986)

[**13.7 Beta Testing** 77](#_Toc38653987)

[**13.8 First Customer Shipment** 77](#_Toc38653988)

[**13.8 Product Sustaining** 78](#_Toc38653989)

[**13.9 Measuring Test Effectiveness** 79](#_Toc38653990)

[**13.10 Basic System Test** 79](#_Toc38653991)

# **LECTURE 1: Basics of Software Quality Assurance**

## **What is Software and its types**

We have been witnessing tremendous growth in the software industry over the past 25 years. Software applications have proliferated from the original data processing and scientific computing domains into our daily lives in such a way that we do not realize that some kind of software executes when we do even something ordinary, such as making a phone call, starting a car, turning on a microwave oven, and making a debit card payment.

Software is (1) instructions (computer programs) that when executed provide desired function and performance, (2) data structures that enable the programs to adequately manipulate information, and (3) documents that describe the operation and use of the programs. There are no question that other, more complete definitions could be offered. But we need more than a formal definition. Therefore, we can say that the Software is more than just a computer program. It composed of computer programs, procedures, and possibly associated documentation and data related to the operation of a computer system. Two major types of Software:

* Generic (Buy) – Stand alone, Sold on open market (OTH)
* Customized (Build) – For specific customer or business

BUY & BUILD – Most software that you buy for personal use will likely be ‘commercial off-the-shelf’ (COTS), means the same application will be run by thousands of other users. It may not work exactly how you’d like it to or do everything you’d want it to whereas a customized (bespoke) software solution is built exactly to the business’ specific requirements.

## **Software Development Lifecycel (SDLC)**

The basic idea of testing involves the execution of software and the observation of its behavior or outcome. If a failure is observed, the execution record is analyzed to locate and fix the fault(s) that caused the failure. Otherwise, we gain some confidence that the software under testing is more likely to fulfill its designated functions.

Similar to the situation for many physical systems and products, the purpose of software testing is to ensure that the software systems would work as expected when they are used by their target customers and users. The most natural way to show this fulfillment of expectations is to demonstrate their operation through some “dry-runs” or controlled experimentation in laboratory settings before the products are released or delivered. In the case of software products, such controlled experimentation through program execution is generally called testing.

Because of the relatively defect-free manufacturing process for software as compared to the development process, we focus on testing in the development process. We run or execute the implemented software systems or components to demonstrate that they work as expected. Therefore, “demonstration of proper behavior” is a primary purpose of testing, which can also be interpreted as providing evidence of quality in the context of software QA, or as meeting certain quality goals.

However, because of the ultimate flexibility of software, where problems can be corrected and fixed much more easily than traditional manufacturing of physical products and systems, we can benefit much more from testing by fixing the observed problems within the development process and deliver software products that are as defect-free as our budget or environment allows. As a result, testing has become a primary means to detect and fix software defects under most development environments, to the degree that “detecting and fixing defects” has eclipsed quality demonstration as the primary purpose of testing for many people and organizations.

To summarize, software Testing is the process of executing a system or component under specified conditions with the intent of finding defects/bugs and to verify that it satisfies specified requirements. The goals of Software Testing

* Main goal is to detect issues
* Requirement conformance
* Errors in system operation
* System performance (TCAS in Autopilot system)

Have different levels of testing operation

* Unit testing in developing each module
* Integration testing to combine different module and check incompatibility
* System testing is done at system combination of software and hardware

**Role of Testing in Project Management**

Testing plays an important role in achieving and assessing the quality of a software product. On the one hand, we improve the quality of the products as we repeat a *test–find defects–fix* cycle during development. On the other hand, we assess how good our system is when we perform system-level tests before releasing a product. Thus, the research has succinctly described, software testing is a verification process for software quality assessment and improvement. Generally speaking, the activities for software quality assessment can be divided into two broad categories, namely, *static analysis* and *dynamic analysis*.

**Static Analysis:** As the term “static” suggests, it is based on the examination of a number of documents, namely requirements documents, software models, design documents, and source code. Traditional static analysis includes code review, inspection, walk-through, algorithm analysis, and proof of correctness. It does not involve actual execution of the code under development. Instead, it examines code and reasons over all possible behaviors that might arise during run time. Compiler optimizations are standard static analysis.

**Dynamic Analysis:** Dynamic analysis of a software system involves actual program execution in order to expose possible program failures. The behavioral and performance properties of the program are also observed. Programs are executed with both typical and carefully chosen input values. Often, the input set of a program can be impractically large. However, for practical considerations, a finite subset of the input set can be selected. Therefore, in testing, we observe some representative program behaviors and reach a conclusion about the quality of the system. Careful selection of a finite test set is crucial to reaching a reliable conclusion.

By performing static and dynamic analyses, practitioners want to identify as many faults as possible so that those faults are fixed at an early stage of the software development. Static analysis and dynamic analysis are complementary in nature, and for better effectiveness, both must be performed repeatedly and alternated. Practitioners and researchers need to remove the boundaries between static and dynamic analysis and create a hybrid analysis that combines the strengths of both approaches

## **What is Software Quality**

The question “What is software quality?” evokes many different answers. Quality is a complex concept—it means different things to different people, and it is highly context dependent. The research has analyzed how software quality is perceived in different ways in different domains, such as philosophy, economics, marketing, and management.

Software Quality (as per ISO/ IEC 9126): The totality of functionality and features of a software product that contributes to its ability to satisfy stated or implied needs. It can be customized for organizations.

Software Quality (as IEEE Std 610): The degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations.

## **Challenges in Software Project**

The processes for producing software must meet two broad challenges. First, the processes must produce low-cost software in a short time so that corporations can stay competitive. Second, the processes must produce usable, dependable, and safe software; these attributes are commonly known as quality attributes. Software quality impacts a number of important factors in our daily lives, such as economy, personal and national security, health, and safety.

The primary goal of project management activity is to deliver the product on time, on budget and expected quality. What’s the main challenges of software development now-a-days?

* Time: late, difficult to deliver on time
* Cost: high cost, over budget
* Scope: low quality with faults

## **Example of Software Defects**

Flight Ariane 5 Most Expensive Computer Bug in History on June 4, 1996, the rocket Ariane 5 tore itself apart 37 seconds after launch because of a malfunction in the control software making the fault most expensive computer bug in history.

Lethal X-Rays :Therac-25 system Therac-25 was a radiation therapy machine produced by Atomic Energy of Canada Ltd (AECL) in 1986. But initially lot of people died because of massive overdose of radiation. And this is happened because of a software bug.

## **Software Quality Assurance (SQA)**

Software Quality Assurance develops an effective plan for software development. It represents the function of software quality that assures that the standards, processes and procedures are appropriate for the project and are correctly implemented. It is also defined as a planned and systematic approach to the evaluation of the quality of and adherence to software product standards, processes and procedures. In SQA, there is a set of activities designed to ensure that the development and/or maintenance process is adequate to ensure a system will meet its objectives. It is an umbrella activity that is applied throughout the software process which consists of a means of monitoring the software engineering processes and methods used to ensure quality. An effective approach to produce high quality software.

## **Software Quality Control (SQC)**

Software Quality Control (SQC) involves in executing the software development plan effectively. It is the function of software quality that checks that the project follows its standards processes, and procedures, and that the project produces the required internal and external (deliverable) products. In SQC a set of activities designed to evaluate a developed work product. It includes the following activities: Requirement Review, Design Review, Code Review, Deployment Plan Review, Test Plan Review, and Test Cases Review.

# **LECTURE 2: Basics of Software Quality Assurance**

## **Validation and Verification**

Two similar concepts related to software testing frequently used by practitioners are *verification* and *validation*. Both concepts are abstract in nature, and each can be realized by a set of concrete, executable activities. The two concepts are explained as follows:

**Verification:** This kind of activity helps us in evaluating a software system by determining whether the product of a given development phase satisfies the requirements established before the start of that phase. One may note that a product can be an intermediate product, such as requirement specification, design specification, code, user manual, or even the final product. Activities that check the correctness of a development phase are called *verification activities*.

**Validation:** Activities of this kind help us in confirming that a product meets its intended *use*. Validation activities aim at confirming that a product meets its customer’s expectations. In other words, validation activities focus on the final product, which is extensively tested from the customer point of view. Validation establishes whether the product meets overall expectations of the users.

## **Difficulties on Quality Expectations**

There are various difficulties in achieving good quality. For example, the Size of the software functionalities (LOC in the function code) where manual testing would be difficult to find logical errors in bulky source code. There might be some invariant Complexity in the Product and Project Complexity in S/W is more because it deals with logic where any error in logic may cause significant impact such as autopilot software (e.g. set landing mode). The Environmental stress or constraints might be a difficulty while test the system with it maximum load (e.g. result publish website). The flexibility or adaptability might be difficulty to meet in a software development while frequent changes are requested assuming that software development is flexible, and each time regression testing must be performed after each change. And finally, the cost of testing and market conditions may arise difficulty which involves testing cost with automated tools and license for the tool.

## **Basics of Software Quality Engineering (SQE) and SQE Activities**

Different customers and users have different quality expectations under different market environments. Therefore, we need to move beyond just performing QA activities toward quality engineering by managing these quality expectations as an engineering problem: Our goal is to meet or exceed these quality expectations through the selection and execution of appropriate QA activities while minimizing the cost and other project risks under the project constraints.

In order to ensure that these quality goals are met through the selected QA activities, various measurements need to be taken parallel to the QA activities themselves. Postmortem data often need to be collected as well. Both in-process and post-mortem data need to be analyzed using various models to provide an objective quality assessment. Such quality assessments not only help us determine if the preset quality goals have been achieved, but also provide us with information to improve the overall product quality.

The purpose of SQE is to remove defect and ensure quality. For example, the SQE activities are Defect Prevention which involves a plan for avoid any defect. Formal Verification is the process of finding defects through Inspection, Review, and Walkthrough. Fault Tolerance is the method to enhance the robustness of the system for example through data backup, and autosaving document, etc.

## **Error, Fault, Failure, and Defect**

In the literature on software testing, one can find references to the terms *failure*, *error*, *fault*, and *defect*. Although their meanings are related, there are important distinctions between these four concepts. In the following, we present first three terms as they are understood in the fault-tolerant computing community:

**Failure:** A failure is said to occur whenever the external behavior of a system does not conform to that prescribed in the system specification.

**Error:** An error is a *state* of the system. In the absence of any corrective action by the system, an error state could lead to a failure which would not be attributed to any event subsequent to the error.

**Fault:** A fault is the adjudged cause of an error.

A fault may remain undetected for a long time, until some event activates it. When an event activates a fault, it first brings the program into an intermediate error state. If computation is allowed to proceed from an error state without any corrective action, the program eventually causes a failure. As an aside, in fault-tolerant computing, corrective actions can be taken to take a program out of an error state into a desirable state such that subsequent computation does not eventually lead to a failure. The process of failure manifestation can therefore be succinctly represented as a behavior chain as follows: fault → error → failure. The behavior chain can iterate for a while, that is, failure of one component can lead to a failure of another interacting component.

## **Testing Activities and Complete Testing**

It is not unusual to find people making claims such as “I have exhaustively tested the program.” Complete, or exhaustive, testing means *there are no undiscovered faults at the end of the test phase*. All problems must be known at the end of complete testing. For most of the systems, complete testing is near impossible because of the following reasons:

* The domain of possible inputs of a program is too large to be completely used in testing a system. There are both valid inputs and invalid inputs. The program may have a large number of states. There may be timing constraints on the inputs, that is, an input may be valid at a certain time and invalid at other times. An input value which is valid but is not properly timed is called an *inopportune* input. The input domain of a system can be very large to be completely used in testing a program.
* The design issues may be too complex to completely test. The design may have included implicit design decisions and assumptions. For example, a programmer may use a global variable or a *static* variable to control program execution.
* It may not be possible to create all possible execution environments of the system. This becomes more significant when the behavior of the software system depends on the real, outside world, such as weather, temperature, altitude, pressure, and so on.

We must realize that though the outcome of complete testing, that is, discovering all faults, is highly desirable, it is a near-impossible task, and it may not be attempted. The next best thing is to select a subset of the input domain to test a program.

In order to test a program, a test engineer must perform a sequence of testing activities. Most of these activities are explained in the following. These explanations focus on a single test case.

**Identify an objective to be tested:** The first activity is to identify an *objective* to be tested. The objective defines the intention, or *purpose*, of designing one or more test cases to ensure that the program supports the objective. A clear purpose must be associated with every test case.

**Select inputs:** The second activity is to select test inputs. Selection of test inputs can be based on the requirements specification, the source code, or our expectations. Test inputs are selected by keeping the test objective in mind.

**Compute the expected outcome:** The third activity is to compute the expected outcome of the program with the selected inputs. In most cases, this can be done from an overall, high-level understanding of the test objective and the specification of the program under test.

**Set up the execution environment of the program:** The fourth step is to prepare the right execution environment of the program. In this step all the assumptions external to the program must be satisfied. A few examples of assumptions external to a program are as follows: Initialize the local system, external to the program. This may include making a network connection available, making the right database system available, and so on. Initialize any remote, external system (e.g., remote partner process in a distributed application). For example, to test the client code, we may need to start the server at a remote site.

**Execute the program:** In the fifth step, the test engineer executes the program with the selected inputs and observes the actual outcome of the program. To execute a test case, inputs may be provided to the program at different physical locations at different times. The concept of *test coordination* is used in synchronizing different components of a test case.

**Analyze the test result:** The final test activity is to analyze the result of test execution. Here, the main task is to compare the actual outcome of program execution with the expected outcome. The complexity of comparison depends on the complexity of the data to be observed. The observed data type can be as simple as an integer or a string of characters or as complex as an image, a video, or an audio clip. At the end of the analysis step, a test verdict is assigned to the program. There are three major kinds of test verdicts, namely, *pass*, *fail*, and *inconclusive*, as explained below.

If the program produces the expected outcome and the purpose of the test case is satisfied, then a pass verdict is assigned. If the program does not produce the expected outcome, then a fail verdict is assigned.

However, in some cases it may not be possible to assign a clear pass or fail verdict. For example, if a timeout occurs while executing a test case on a distributed application, we may not be in a position to assign a clear pass or fail verdict. In those cases, an inconclusive test verdict is assigned. An inconclusive test verdict means that further tests are needed to be done to refine the inconclusive verdict into a clear pass or fail verdict.

A *test report* must be written after analyzing the test result. The motivation for writing a test report is to get the fault fixed if the test revealed a fault. A test report contains the following items to be informative: Explain how to reproduce the failure. Analyze the failure to be able to describe it. A pointer to the actual outcome and the test case, complete with the input, the expected outcome, and the execution environment.

## **2.6 Testing Levels**

Testing is performed at different levels involving the complete system or parts of it throughout the life cycle of a software product. A software system goes through four stages of testing before it is actually deployed. These four stages are known as *unit*, *integration*, *system*, and *acceptance* level testing. The first three levels of testing are performed by a number of different stakeholders in the development organization, whereas acceptance testing is performed by the customers.

In unit testing, programmers test individual program units, such as a procedures, functions, methods, or classes, in isolation. After ensuring that individual units work to a satisfactory extent, modules are assembled to construct larger subsystems by following integration testing techniques. Integration testing is jointly performed by software developers and integration test engineers. The objective of integration testing is to construct a reasonably stable system that can withstand the rigor of system-level testing. System-level testing includes a wide spectrum of testing, such as functionality testing, security testing, robustness testing, load testing, stability testing, stress testing, performance testing, and reliability testing. System testing is a critical phase in a software development process because of the need to meet a tight schedule close to delivery date, to discover most of the faults, and to verify that fixes are working and have not resulted in new faults. System testing comprises a number of distinct activities: creating a test plan, designing a test suite, preparing test environments, executing the tests by following a clear strategy, and monitoring the process of test execution.

*Regression testing* is another level of testing that is performed throughout the life cycle of a system. Regression testing is performed whenever a component of the system is modified. The key idea in regression testing is to ascertain that the modification has not introduced any new faults in the portion that was not subject to modification. To be precise, regression testing is not a distinct level of testing. Rather, it is considered as a subphase of unit, integration, and system-level testing

In regression testing, new tests are not designed. Instead, tests are selected, prioritized, and executed from the existing pool of test cases to ensure that nothing is broken in the new version of the software. Regression testing is an expensive process and accounts for a predominant portion of testing effort in the industry. It is desirable to select a subset of the test cases from the existing pool to reduce the cost. A key question is how many and which test cases should be selected so that the selected test cases are more likely to uncover new faults.

After the completion of system-level testing, the product is delivered to the customer. The customer performs their own series of tests, commonly known as *acceptance testing*. The objective of acceptance testing is to measure the quality of the product, rather than searching for the defects, which is objective of system testing. A key notion in acceptance testing is the customer’s *expectations* from the system. By the time of acceptance testing, the customer should have developed their acceptance criteria based on their own expectations from the system.

## **2.7 White-Box and Black-Box Testing**

Test cases need to be designed by considering information from several sources, such as the specification, source code, and special properties of the program’s input and output domains. This is because all those sources provide complementary information to test designers. Two broad concepts in testing, based on the sources of information for test design, are *white-box* and *black-box* testing. White-box testing techniques are also called *structural testing t*echniques, whereas black-box testing techniques are called *functional testing* techniques.

In structural testing, one primarily examines *source code* with a focus on control flow and data flow. Control flow refers to flow of control from one instruction to another. Control passes from one instruction to another instruction in a number of ways, such as one instruction appearing after another, function call, message passing, and interrupts. Conditional statements alter the normal, sequential flow of control in a program. Data flow refers to the propagation of values from one variable or constant to another variable. Definitions and uses of variables determine the data flow aspect in a program.

In functional testing, one does not have access to the internal details of a program and the program is treated as a black box. A test engineer is concerned only with the part that is accessible outside the program, that is, just the input and the externally visible outcome. A test engineer applies input to a program, observes the externally visible outcome of the program, and determines whether or not the program outcome is the expected outcome. Inputs are selected from the program’s requirements specification and properties of the program’s input and output domains. A test engineer is concerned only with the functionality and the features found in the program’s specification.

At this point it is useful to identify a distinction between the scopes of structural testing and functional testing. One applies structural testing techniques to individual units of a program, whereas functional testing techniques can be applied to both an entire system and the individual program units. Since individual programmers know the details of the source code they write, they themselves perform structural testing on the individual program units they write. On the other hand, functional testing is performed at the external interface level of a system, and it is conducted by a separate software quality assurance group.

Let us consider a program unit *U* which is a part of a larger program *P*. A program unit is just a piece of source code with a well-defined objective and well-defined input and output domains. Now, if a programmer derives test cases for testing *U* from a knowledge of the internal details of *U*, then the programmer is said to be performing structural testing. On the other hand, if the programmer designs test cases from the stated objective of the unit *U* and from his or her knowledge of the special properties of the input and output domains of *U*, then he or she is said to be performing functional testing on the same unit *U*.

The ideas of structural testing and functional testing do not give programmers and test engineers a choice of whether to design test cases from the source code or from the requirements specification of a program. However, these strategies are used by different groups of people at different times during a software’s life cycle. For example, individual programmers use both the structural and functional testing techniques to test their own code, whereas quality assurance engineers apply the idea of functional testing.

Neither structural testing nor functional testing is by itself good enough to detect most of the faults. Even if one selects all possible inputs, a structural testing technique cannot detect all faults if there are *missing paths* in a program. Intuitively, a path is said to be missing if there is no code to handle a possible condition. Similarly, without knowledge of the structural details of a program, many faults will go undetected. Therefore, a combination of both structural and functional testing techniques must be used in program testing.

# **LECTURE 3: Software Quality**

## **3.1** **Quality Perspective**

When software quality is concerned, different people would have different views and expectations based on their roles and responsibilities. With the quality assurance (QA) and quality engineering focus of this book, we can divide the people into two broad groups:

***Consumers*** of software products or services, including customers and users, either internally or externally. Sometimes we also make the distinction between the *customers,* who are responsible for the acquisition of software products or services, and the *users,* who use the software products or services for various purposes, although the dual roles of customers and users are quite common. We can also extend the concept of users to include such non-human or “invisible” users as other software, embedded hardware, and the overall operational environment that the software operates under and interacts with.

***Producers***of software products, or anyone involved with the development, management, maintenance, marketing, and service of software products. We adopt a broad definition of producers, which also include third-party participants who may be involved in add-on products and services, software packaging, software certification, fulfilling independent verification and validation (IV&V) responsibilities, and so on.

## **Views in Software Quality**

We next examine the different views of quality in a systematic manner, based on the different roles, responsibilities, and quality expectations of different people, and zoom in on a small set of views and related properties to be consistently followed throughout this manual. Five major views according to (Kitchenham and Pfleeger, 1996; Pfleeger et al., 2002) are: transcendental, user, manufacturing, product, and value-based views, as outlined below:

***Mystical or Transcendental View***: It envisages quality as something that can be recognized but is difficult to define. The transcendental view is not specific to software quality alone but has been applied in other complex areas of everyday life. For example, In 1964, Justice Potter Stewart of the U.S. Supreme Court, while ruling on the case *Jacobellis v. Ohio*, 378 U.S. 184 (1964), which involved the state of Ohio banning the French film *Les Amants* (“The Lovers”) on the ground of pornography, wrote “I shall not today attempt further to define the kinds of material I understand to be embraced within that shorthand description; and perhaps I could never succeed in intelligibly doing so. But *I know it when I see it* , and the motion picture involved in this case is not that” (emphasis added).

***User View*:** It perceives quality as fitness for purpose. According to this view, while evaluating the quality of a product, one must ask the key question: “Does the product satisfy user needs and expectations?”. This view represents the quality concerns the extent to which a product meets user needs and expectations. A product is of good quality if it satisfies a large number of users. It is useful to identify the product attributes which the users consider to be important. This view may encompass many subject elements, e.g. usability, reliability, efficiency, etc.

***Manufacturing View*:** Here quality is understood as conformance to the specification. The quality level of a product is determined by the extent to which the product meets its specifications. It represents the conformance to process standards/requirements where Quality is seen as conforming to requirements leads to consistency in products. Any deviation from the requirements is seen as reducing the quality of the product and the Products are manufactured “right the first time” so that the cost can be reduced (very low changing possibility and without re-work). The Product quality can be incrementally improved by improving the process. The CMM and ISO 9001 models are based on the manufacturing view.

***Product View:*** In this case, quality is viewed as tied to the inherent characteristics of the product. A product’s inherent characteristics, that is, internal qualities, determine its external qualities. In other words, if a product is manufactured with good internal properties, then it will have good external properties

***Value-Based View*:** Quality, in this perspective, depends on the amount a customer is willing to pay for it. The Value-based view represents the merger of two concepts: excellence and worth. Quality is a measure of excellence, and value is a measure of worth. Value-based view makes a trade-off between cost and quality

## **3.3** **Why Measure Software Quality?**

It is useful to measure quality of software for three reasons. First, measurement allows us to develop baselines for quality. Second, since quality improvement has an associated cost, it is important to know how much quality improvement is achieved for a certain cost. Finally, it is useful to know the present level of quality so that further improvement can be planned.

## **Software Quality Factor**

Software quality factors are the behavioral characteristic of a system. Some examples of high-level quality factors are correctness, reliability, efficiency, performance, etc.

**Correctness.** A program must operate correctly, or it provides little value to its users. Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements. When considering the overall quality of a software product, defects are those problems reported by a user of the program after the program has been released for general use. For quality assessment purposes, defects are counted over a standard period of time, typically one year.

***Reliability***The amount of time that the software is available for use as indicated by the following sub attributes: maturity, fault tolerance, recoverability. It is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.

***Performance*** requirements define how well or how rapidly the system must perform specific functions. It also defines speed (e.g. database response times), throughput (transactions per second), capacity (concurrent usage loads) and , timing (hard real-time demands). Performance requirements should also address how the system's performance will degrade in an overloaded situation, such as when a 911 emergency telephone system is flooded with calls. Some performance quality are as follows

*PE-1. Every Web page shall download in 15 seconds or less over a 50 KBps modem connection.*

*PE-2. Authorization of an ATM withdrawal request shall not take more than 10 seconds.*

***Efficiency*** is a measure of how well the system utilizes processor capacity, disk space, memory, or communication bandwidth. Efficiency is related to performance, (response time) another class of nonfunctional requirement. If a system consumes too much of the available resources, users will encounter degraded performance, a visible indication of inefficiency. Poor performance is an irritant to the user who is waiting for a database query to display results. But performance problems can also represent serious risks to safety, such as when a real-time process control system is overloaded. It Considers **minimum hardware configurations** when defining efficiency, capacity, and performance goals.

*Example: EF-1. At least 25 percent of the processor capacity and RAM available to the application shall be unused at the planned peak load conditions.*

## **Software Quality Criteria**

Software quality criteria are the attribute of a quality factor that is related to software development. For example, Modularity, Testability, Maintainability, Reusability, etc.

***Modularity*** is an attribute of the architecture of a software system. A highly modular software allows designers to put cohesive components in one module, thereby increasing the maintainability of the system.

***Maintainability*** indicates how easy it is to correct a defect or modify the software. Maintainability depends on how easily the software can be understood, changed, and tested. It is closely related to flexibility and testability. High maintainability is critical for products that will undergo frequent revision and for products that are being built quickly. Maintainability can be measured in terms of the average time required to fix a problem and the percentage of fixes that are made correctly.

*Example: MA-1. A maintenance programmer shall be able to modify existing reports to conform to revised chemical-reporting regulations from the federal government with 20 labor hours or less of development effort.*

***Testability*** refers to the ease with which software components or the integrated product can be tested to look for defects. It is also known as *verifiability.* Designing for testability is critical if the product has complex algorithms and logic, or if it contains indirect (ambiguous) functionality interrelationships. Testability is also important if the product will be modified often because it will undergo frequent regression testing to determine whether the changes damaged any existing functionality.

Example: *TE-1. The maximum cyclomatic complexity\* of a module shall not exceed 20.*

*\*Cyclomatic complexity* is a measure of the number of logic branches in a source code module

## **ISO-9126 Quality Framework**

ISO-9126 stands for International Organization for Standardization. It is the mostly influential one in the software engineering community today. It provides a hierarchical framework for quality definition, organized into quality characteristics and sub-characteristics. There are Six top-level quality characteristics, each associated with its own exclusive (non-overlapping) sub-characteristics. Functionality: what is needed. Reliability, Usability, Efficiency, Maintainability and, Portability.

# **LECTURE 4: Maturity Models**

## **4.1 Basic Idea in Software Process**

The concept of a *process* plays an important role in today’s software development. Without a repeatable process, the only repeatable results you are likely to produce are errors. In the software engineering context, a process comprises a set of activities that are executed to develop software products. The activities find expressions in the form of methods, techniques, strategies, procedures, and practices. Those activities heavily rely on information repositories such as documents, standards, and policies. Different processes are driven by different goals and availability of resources. The concept of a process is applied not just to develop source code, but to other software-related products, such as a project plan, requirements document, design document, and user manual, as well. It is effective to follow a defined process because of the following benefits:

* The process can be repeated in subsequent projects.
* The process can be evaluated by using a variety of metrics, such as cost, quality, and time to deliver a product.
* Actions can be taken to improve the process to achieve better results.

A process for software development consists of several different processes for its different component tasks. For example, gathering the requirements of a system, constructing a functional specification of a system, designing a system, testing a system, and maintaining a system are different tasks to be performed in the life cycle of a software system. Those different tasks, which are quite different from each other, need different processes to be followed. A requirement gathering process is fundamentally different from a testing process. Similarly, a design process is very different from a testing process. However, some similarities between a maintenance process and a testing process cannot be ruled out because a maintenance task may involve significant changes to the design of the system as well as rigorous testing.

Software testing is treated as a distinct process because it involves a variety of unique activities, techniques, strategies, and policies as explained in the following. Testing is performed to achieve the following objectives:

* Reveal defects in software products.
* Show the extent to which a software system possesses different quality attributes, such as reliability, performance, stability, and scalability.
* Testing begins almost at the same time as a project is conceptualized, and it lasts as long as the system is in existence.
* Testing is carried out by people with differing responsibilities in different phases of software development. Those different levels of testing are commonly known as unit testing, integration testing, system testing, and acceptance testing.
* A number of techniques can be applied at each level of testing for generating test cases.
* A number of different testing strategies can be applied at different levels of testing.
* A number of metrics can be monitored to gauge the progress of testing.
* A number of metrics can be monitored to accurately reflect the different quality levels of a system.
* Testing is influenced by organizational policies, such as the level of quality to be achieved in a product, the fraction of the total budget for a project to be allocated to testing, and the duration of testing.
* Testing can be performed in a combination of manual and automated modes of execution of test cases.

To be able to improve a defined process, organizations need to evaluate its capabilities and limitations. For example, the *capability maturity model* (CMM), developed by the Software Engineering Institute (SEI) at Carnegie Mellon University, allows an organization to evaluate its software development processes. The model supports incremental process improvement. For an organization, it is more practical to develop a continuously improving process than find one which is “just right.” By recognizing the importance of testing as a distinct process—much different from an overall process for software development—a separate model, known as a *testing maturity model* (TMM), has been developed to evaluate a testing process. In addition, for an organization to be able to improve its testing process, the *test process improvement* (TPI) model has been developed.

## **4.2 Capability Maturity Model (CMM)**

The maturity level of a development process tells us to what extent the organization is capable of producing low-cost, high-quality software. Therefore, the evaluation framework is the CMM. After evaluating the current maturity level of a development process, organizations can work on improving the process to achieve the next higher level of process maturity. In the CMM framework, a process has five maturity levels. Before going into the details of different maturity levels, it is useful to have a glimpse of an immature process—an organization can be considered to be immature if it follows immature processes. The five levels of process maturity and their KPAs are explained as follows:

***Level 1: Initial*** At this level, software is developed by following no process model. There is not much planning involved. Even if a plan is prepared, it may not be followed. Individuals make decisions based on their own capabilities and skills. There is no KPA associated with level 1. An organization reaches level 1 without making any effort.

***Level 2: Repeatable*** At this level, the concept of a process exists so that success can be repeated for similar projects. Performance of the proven activities of past projects is used to prepare plans for future projects. This level can be summarized as being *disciplined* because processes are used for repeatability. All the processes are under the effective control of a project management system. The KPAs at level 2 are as follows:

* **Requirements Management:** It is important to establish a common understanding between the customer and developers. Details of a project, such as planning and management, are guided by a common view of customer requirements.
* **Software Project Planning:** This means creating and following a reasonable plan for realizing and managing a project.
* **Software Project Tracking and Oversight:** This means making the progress of a project visible so that management is aware of the status of the project. Corrective actions can be taken if the actual progress of a project significantly deviates from the planned progress.
* **Software Subcontract Management:** This means evaluating, selecting, and managing suppliers or subcontractors.
* **Software Quality Assurance:** This means evaluating processes and products to understand their effectiveness and quality.
* **Software Configuration Management:** This means ensuring the integrity of the products of a project as long as the project continues to exist.

***Level 3: Defined*** At this level, documentation plays a key role. Processes that are related to project management and software development activities are documented, reviewed, standardized, and integrated with organizational processes. In other words, there is organization wide acceptance of standard processes. Software development is carried out by following an approved process. Functionalities and the associated qualities are tracked. Cost and schedule are monitored to keep them under control. The KPAs at level 3 are as follows:

* **Organization Process Focus:** This means putting in place an organization wide role and responsibility to ensure that activities concerning process improvement are in fact followed.
* **Organization Process Definition:** Certain practices are useful irrespective of projects. Thus, it is important to identify and document those practices.
* **Training Program:** Individuals need to be trained on an on-going basis to make them knowledgeable in application domains and new developments in software techniques and tools. Training is expected to make them effective and efficient.
* **Integrated Software Management:** This means integrating an organization’s software engineering and management activities into a common, defined process. Integration is based on commercial and technological needs of individual projects.
* **Software Product Engineering:** This means following a defined process in a consistent manner by integrating the technical activities to produce software with desired attributes. The activities include requirements elicitation, functional design, detailed design, coding, and testing.
* **Intergroup Coordination:** This means that a software development group coordinates with other groups, such as customers, the marketing group, the and software quality assurance (SQA) group, to understand their needs and expectations.
* **Peer Review:** Work products, such as requirements, design, and code, are reviewed by peers to find defects at an early stage. Peer reviews can be performed by means of inspection and walkthrough.

***Level 4: Managed*** At this level, *metrics* play a key role. Metrics concerning processes and products are collected and analyzed. Those metrics are used to gain quantitative insight into both process and product qualities. When the metrics show that limits are being exceeded, corrective actions are triggered. For example, if too many test cases fail during system testing, it is useful to start a process of root cause analysis to understand why so many tests are failing. The KPAs at level 4 are as follows:

* **Quantitative Process Management:** Process data indicate how well a process is performing. If a process does not perform as expected, the process is improved by considering the measured data.
* **Software Quality Management:** The quality attributes of products are measured in quantitative form to have a better insight into the processes and products. Improvements are incorporated into the processes, and their effectiveness is evaluated by measuring product quality attributes.

***Level 5: Optimizing*** At this level, organizations strive to improve their processes on a continual basis. This is achieved in two steps: (i) Observe the effects of processes, by measuring a few key metrics, on the quality, cost, and lead time of software products and (ii) effect changes to the processes by introducing new techniques, methods, tools, and strategies. The following are the KPAs at level 5:

* **Defect Prevention:** This means analyzing the root causes of different classes of defects and taking preventive measures to ensure that similar defects do not recur.
* **Technology Change Management:** This means identifying useful techniques, tools, and methodologies and gradually introducing those into software processes. The key idea is to take advantage of new developments in technologies.
* **Process Change Management:** This means improving an organization’s processes to have a positive impact on quality, productivity, and development time.

## **4.3 Test Process Improvement (TPI)**

A *test process* is a certain way of performing activities related to defect detection. Some typically desired test activities in software development are as follows:

* Identifying test goals
* Preparing a test plan
* Identifying different kinds of tests
* Hiring test personnel
* Designing test cases
* Setting up test benches
* Procuring test tools
* Assigning test cases to test engineers
* Prioritizing test cases for execution
* Organizing the execution of test cases into multiple test cycles
* Preparing a schedule for executing test cases
* Executing test cases
* Reporting defects
* Tracking defects while resolving them
* Measuring the progress of testing
* Measuring the quality attributes of the software under test
* Evaluating the effectiveness of a test process
* Identifying steps to improve the effectiveness of test activities
* Identifying steps to reduce the cost of testing

Therefore, it is important to improve test processes, and it is advantageous to follow a defined model for their improvement. The idea of improving test processes by following a model, namely the TPI model, was first studied by Tim Koomen and Martin Pol. They have extensively discussed the TPI model in a well-written book. A test process needs to be improved for three reasons as explained in the following:

**Quality:** A better test process should give more insight into the quality characteristics of a system being tested. For example, after running all the test cases twice, we should have a good idea about the reliability level of the system. Here, the intention is not to have a better-quality product—though that is always desirable. Rather, the intention is to have an improved test process that gives us better insight into the quality of the system being tested.

**Lead Time:** A better test process saves testing time, and thereby gives more time to other areas of system development. For example, one can use the idea of prioritizing the execution of test cases so that difficult-to-fix defects are detected as early as possible. Early detection of difficult-to-fix defects gives more time to the developers to address those defects, thereby shortening the development time.

**Cost:** A better test process is expected to be carried out with a lower cost and thereby reduces the overall cost of system development. For example, one can judiciously select a subset of the test cases, rather than the entire test suite, for regression testing toward the closing stage of system-level testing during development. By carefully choosing a subset of the test suite, we can reduce the cost of system testing without compromising its effectiveness. Test cases can have a positive impact on development cost by influencing the processes of requirements gathering and software design.

It is apparent that a better test process gives us more insight into the quality attributes of a system, and it contributes to the development of software products at a lower *cost* in less time. An intuitive approach to improving a test process is as follows:

**Step 1. Determine Area for Improvement:** We identify an area of testing where we would like to see immediate improvements. For guidelines, we look at the three major areas for improvement, namely, quality, time, and cost, and focus on one precise aspect. Suppose that we want to have a good idea about the performance quality of the system as early as possible.

**Step 2. Evaluate Current State of Test Process:** It is important to know where we stand with respect to the chosen area for improvement. This is because it is likely to influence where we go next and how to achieve that. Referring to the example of having a good idea about the performance of the system as early as possible, we evaluate *when* the current test process lets us know about the system performance. Such an evaluation can be done by identifying the time when performance-related test cases are executed. If such test cases are executed toward the closing phase of system testing, there is a possibility that such tests are performed earlier.

**Identify Next Desired State and Means:** Once we know the current state of the testing process with respect to the desired area for improvement, we are in a better position to evaluate the extent of possible improvement. One must carefully identify the amount of improvement to be seen. Too much improvement in one state may require significant changes in the test process. Therefore, incremental improvements are preferred. Sometimes, a certain improvement may call for improvements in other, unselected areas. The type and extent of improvements one wants to effect must be carefully selected. For example, one can evaluate the performance of a system under different execution environments, such as a lightly loaded system, moderately loaded system, and heavily loaded system. Therefore, if we are interested in the early performance information in a lightly loaded environment, we can prioritize the performance-related tests to be executed before performing load tests. In the above example, *prioritization* of test execution is an example of the means necessary to affect an improvement.

**Step 4. Implement Necessary Changes to Process:** Improvements, as identified above, are affected in the test process in a planned manner. Then, the effectiveness of the new test process is evaluated to verify whether or not the desired results have been obtained.

## **4.4 Test Maturity Model (TMM)**

Similar to the concept of evaluating and improving software development processes, there is a need for a framework to assess and improve testing processes. Continuous improvement of testing processes is an ideal goal of organizations, and evaluation plays a key role in process improvement. This is because an organization must know its present level of maturity before it takes actions to move to the next level. Ilene Burnstein and her colleagues at the Illinois Institute of Technology pioneered the concept of the TMM to help organizations evaluate and improve their testing processes.

The TMM describes an evolutionary path of test process maturity in five *levels*, or *stages.* The TMM gives guidance concerning how to improve a test process. Each stage is characterized by the concepts of *maturity goals*, *supporting maturity goals*, and *activities, tasks, and responsibilities* (ATRs), as explained in the following:

**Maturity Goals:** Each maturity level, except level 1, contains certain maturity goals. For an organization to achieve a certain level of maturity, the corresponding maturity goals must be met by the organization. The maturity goals are specified in terms of testing improvement goals.

**Maturity Sub goals:** Maturity goals are supported by maturity sub-goals. To achieve a maturity goal, it may be necessary to meet several, fine-grained sub-goals.

**Activities, Tasks, and Responsibilities:** Maturity sub-goals are achieved by means of ATRs that address issues concerning implementation of activities and tasks. ATRs also address how an organization can adapt its practices so that it can move in-line with the TMM model, that is, move from one level to the next. ATRs are further refined into “views,” known as critical views, from the perspectives of three different groups of people: managers, developers and test engineers, and customers (users/clients).

The maturity goals associated with the five levels of the TMM model will be explained in the following.

***Level 1. Initial*** No maturity goals are specified at this level. The TMM level 1 is called the *initial* level. For an organization to be at level 1, nothing special needs to be done. Level 1 represents a scenario where testing is not performed in a planned manner. Testing begins after code is written. At this level, an organization often performs testing to demonstrate that the system works. No serious effort is made to track the progress of testing and the quality level of a product. Test cases are designed and executed in an *ad hoc manner*. Testing resources, such as trained testers, testing tools, and test environments, are not available. In summary, testing is not viewed as a critical, distinct phase in software development.

***Level 2. Phase Definition*** At level 2, the maturity goals are as follows: Develop testing and debugging, goal, initiate a test planning process and, Institutionalize basic testing techniques and methods.

1. Develop Testing and Debugging Goals Separation of testing from debugging is an important growth in the maturity of a testing process. Unit testing and debugging may have some common features such as those being performed by individual programmers. However, their separation becomes more evident when we consider higher levels of testing, for example, integration testing, system-level testing, and acceptance testing. As we move away from unit-level testing to system-level testing, for example, test engineers are more interested in examining the higher-level features of the system, and they do not focus on code-level details.
2. Initiate a Test Planning Process Planning is an indication of a certain level of maturity of any process. The exact extent of maturity depends upon the scope and execution of a plan. Test planning addresses the following: Identify Test Objectives, Analyze Risks, Devise Strategies, Develop Test Specifications, Allocate Resources.
3. Institutionalize Basic Testing and Methods A number of basic testing techniques and methods are widely known in the industry. For example, unit-level testing can be carried out by focusing on the control flow and data flow aspects in a program unit, and a number of code coverage metrics are associated with those testing techniques. Coverage metrics associated with those techniques allow testers to quantify unit-level testing.

***Level 3. Integration*** At level 3, as the name suggests, testing is fully integrated with the development process right from the project planning. Testing is not limited to an activity that starts after coding is over. Neither is testing considered to be solely execution based. Rather, different kinds of test activities are performed throughout the life cycle of a system, and some of those test activities are performed without executing code. At level 3, an organization creates a separate test group consisting of test specialists and test managers, their own resources, and a schedule to be able to inject a sense of quality into software products from project conception.

***Level 4. Management and Measurement*** At level 4 of the TMM, testing acquires a much larger scope and is not just another phase in a software development life cycle. The following are the maturity goals at level 4:

* Establish an organization wide review program.
* Establish a test measurement program.
* Evaluate software quality.

***Level 5. Optimization, Defect Prevention, and Quality Control*** Optimization and defect prevention are the concepts at the highest level, namely level 5. Intuitively, optimization means spending less resources to achieve higher quality products, and defect prevention means taking measures throughout the development process so that products are largely defect free. At level 5, the maturity goals are as follows:

* Application of process data for defect prevention
* Statistical quality control
* Test process optimization

# **LECTURE 5: Quality Assurance**

The primary purpose of the measurement and analysis activities is to provide feedback and useful information to manage software quality, the quality engineering process, and the overall software development/maintenance process and activities. The feedback and information provided are based on the analysis results using various models on the data collected from the quality assurance (QA) and the general development activities. This chapter explain the activities regarding quality assurance in software development project management.

## **5.1 Defect Prevention**

*Defect prevention through error blocking or error source removal:* These QA activities prevent certain types of faults from being injected into the software. Since errors are the missing or incorrect human actions that lead to the injection of faults into software systems, we can directly correct or block these actions, or remove the underlying causes for them. Therefore, defect prevention can be done in two generic ways:

* *Eliminating certain error sources,* such as eliminating ambiguities or correcting human misconceptions, which are the root causes for the errors.
* *Fault prevention or blocking* by directly correcting or blocking these missing or incorrect human actions. This group of techniques breaks the causal relation between error sources and faults through the use of certain tools and technologies, enforcement of certain process and product standards, etc.

Most defect prevention activities assume that there are known error sources or missing/incorrect actions that result in fault injections:

* + If human misconceptions are the error sources, education and training can help us remove these error sources.
  + If imprecise designs and implementations that deviate from product specifications or design intensions are the causes for faults, formal methods can help prevent such deviations
  + If non-conformance to selected processes or standards is the problem that leads to fault injections, then process conformance/standard enforcement can help prevent the injection of related faults
  + If certain tools/technologies can reduce fault injections under similar environments, they should be adopted

## **5.2 Defect Reduction**

*Defect reduction through fault detection and removal:* These QA alternatives detect and remove certain faults once they have been injected into the software systems. In fact, most traditional QA activities fall into this category. For example,

* Inspection directly detects and removes faults from the software code, design, etc.
* Testing removes faults based on related failure observations during program execution.

Various other means, based on either static analyses or observations of dynamic executions, can be applied to reduce the number of faults in a software system. For most large software systems in use today, it is unrealistic to expect the defect prevention activities surveyed above to be 100% effective in preventing accidental fault injections. Need effective techniques to remove as many of the injected faults as possible under project constraints.

* + Inspection
  + Testing
  + Other techniques & risk identification

**Inspection**: Direct fault detection and removal Inspection is the most commonly used static technique for defect detection and removal. Inspections are critical reading & analysis of software code or other software artifacts, such as designs, product specifications, test plans. Inspection are typically conducted by multiple human inspectors, through some coordination process. Faults are detected directly in inspection by inspectors *Formality* and *structures* of Inspections vary Informal reviews/ Walkthroughs. Inspection is most commonly applied to code, but it could be used throughout the development process (particularly early in s/w dev.)

**Testing**: Failure observation and fault removal is one of the most important activities of QA and Most commonly performed QA activity. It involves the execution of software and the observation of the program behavior/outcome. If a failure is observed, the execution record is then analyzed to locate and fix the fault(s) that caused the failure

* *When can a specific testing activity be performed & related faults be detected?*
* *What to test, and what kind of faults are found?*
* *When, or at what defect level , to stop testing?*

## **5.3 Defect Containment**

*Defect containment through failure prevention and containment:* These containment measures focus on the failures by either containing them to local areas so that there are no global failures observable to users, or limiting the damage caused by software system failures. Therefore, defect containment can be done in two generic ways:

* Some QA alternatives, such as the use of fault-tolerance techniques, break the causal relation between faults and failures so that local faults will not cause global failures, thus “tolerating” these local faults.
* A related extension to fault-tolerance is containment measures to avoid catastrophic consequences, such as death, personal injury, and severe property or environmental damages, in case of failures. For example, failure containment for real-time control software used in nuclear reactors may include concrete walls to encircle and contain radioactive material in case of reactor melt-down due to software failures, in order to prevent damage to environment and people’s health.

Defect Containment can be done in two generic ways: some QA alternatives, such as the use of fault-tolerance techniques, break the causal relation between faults and failures so that local faults will not cause global failures, thus “*tolerating*” these local faults. A related extension to fault-tolerance is containment measures to avoid catastrophic consequences, such as death, personal injury, and severe property or environmental damages, in case of failures.

# **LECTURE 6: QA in Context**

## **6.1 QA Activities in Software Development Process**

In the software maintenance process, the focus of QA is on defect handling, to make sure that each problem reported by customers from field operations is logged, analyzed, and resolved, and a complete tracking record is kept so that we can learn from past problems for future quality improvement. In addition, such defect information can be used as additional input in planning for future releases of the same product or for replacement products. Among the different QA activities, defect containment activities play an important role in post-release product operations and maintenance support. For example, fault tolerance using recovery blocks can keep systems operational even in face of problems caused by environmental disturbances. However, repeated use of recovery blocks for the same situations may be an indication of software problems instead of environmental disturbances asthe primary cause of some dynamic problems. Therefore, the systems need to be taken off-line and fixed in order for recovery blocks to work consistently in the future. Even for these techniques, most of the implementation activities need to be carried out during software development, not after product release, similar to the implementation of other product functions or features.

Most of the core QA activities, including defect prevention and defect reduction, are performed during software development instead of during in-field software support after product release. Therefore, we focus on the software development processes in our examination of how different QA activities fit into software processes. In what follows, we examine different QA activities in the general context of several commonly used software development processes, including waterfall, spiral, incremental and iterative development processes. We first examine the process characteristics and the organization of different development activities, and then relate these activities in the process to specific QA activities.

**QA in the waterfall process:** In the most commonly used waterfall process for many large software projects, development activities are typically grouped into different sequential stages to form a waterfall, although overlaps are common among successive pairs of stages. A typical sequence includes, in chronological order: product planning, requirement analysis, specification, design, coding, testing, release, and post-release product support. As a central part of QA activities, testing is an integral part of the waterfall development process, forming an important link in the overall development chain. Other QA activities, although not explicitly stated in the process description, can be carried out throughout other phases and in the transition from one phase to another. For example, part of the criteria to move on from each phase to the next is quality, typically in the form of checking to see if certain quality plans or standards have been completed or followed, as demonstrated by the results from various forms or reviews or inspections.

**QA in other software processes:** In incremental and iterative processes, the overall process can be viewed as consisting of several increments or iterations, with each of them following more or less the same mini stages corresponding to those in the waterfall process. What is more, at the end of each increment or each iteration, the newly developed part needs to be integrated into the existing part. Therefore, integration testing plays a very important role, to make sure that different parts can inter-operate seamlessly to fulfill the intended functionalities correctly together.

The QA activities performed in the spiral process are similar to those performed in incremental and iterative processes. The minor difference is typically in the risk focus adopted in spiral process, where risk identification and analysis play an important role on the decision as to which part to work on next in the subsequent spiral iteration. This risk focus leads naturally to selective QA with a non-uniform effort applied to different parts of the software systems, with high-risk parts receiving more attention than other parts. In terms of testing techniques, usage-based statistical testing according to user operational profiles may fit this process better than other testing techniques.

The agile development method and extreme programming that have become popular recently, especially in the Internet-based and open-source development projects, can be treated as special cases of incremental, iterative, or spiral process models where many of their elements are used or adapted. In fact, QA activities, particularly testing and inspection, play an even more important role than in the traditional software development processes. For example, test-driven development is an integral part of extreme programming*,* and inspection in the form of two persons inspection, or programmer pairs, is extensively used.

## **6.2 V&V in QA Software Process**

QAactivities can also be classified by the binary grouping of verification vs. validation activities. Validation checks the conformance to quality expectations of customers and users in the form of whether the expected functions or features are present or not. On the other hand, verification checks the conformance of software product implementation against its specifications to see if it is implemented correctly. Therefore, validation deals directly with users and their requirements, while verification deals with internal product specifications. In the software development process perspective, different processes may involve customers and users in different ways. Therefore, verification and validation activities may be distributed in these different processes differently.

In the waterfall process, direct involvement of users and user requirement is at the very beginning and the very end of the development process. These phases include project planning, market analysis, requirement analysis, specification, acceptance testing, product release, and post-release product support and maintenance. Therefore, these are the phases where validation activities may be the focus. For example, overall product specifications need to be validated through inspections or reviews to make sure they conform to customer requirements. Various user-oriented testing, such as system, integration, and acceptance testing focus on the validation of user requirement in the form of checking if the functions and features expected by users are present in the software product scheduled to be delivered. Similarly, beta testing and operational support make sure the software product is validated, that is, it is doing what it is supposed to do under the application environment of the target customers.

# **LECTURE 7: Software Quality Engineering**

## **7.1 Quality Engineering: Activity and Process**

Different customers and users have different quality expectations under different market environments. Therefore, we need to move beyond just performing QA activities toward quality engineering by managing these quality expectations as an engineering problem: Our goal is to meet or exceed these quality expectations through the selection and execution of appropriate QA activities while minimizing the cost and other project risks under the project constraints.

In order to ensure that these quality goals are met through the selected QA activities, various measurements need to be taken parallel to the QA activities themselves. Postmortem data often need to be collected as well. Both in-process and post-mortem data need to be analyzed using various models to provide an objective quality assessment. Such quality assessments not only help us determine if the preset quality goals have been achieved, but also provide us with information to improve the overall product quality.

There are three major groups of activities in the quality engineering process, they are labeled in roughly chronological order as pre-QA activities, in-QA activities, and post-QA activities:

## **7.2 QA Activities**

**Pre-QA Activities**

This is quality planning. These are the activities that should be carried out before carrying out the regular QA activities. There are two major types of pre-QA activities in quality planning, including:

1. Set specific quality goals.
2. Form an overall QA strategy, which includes two sub-activities:
3. Select appropriate QA activities to perform.
4. Choose appropriate quality measurements and models to provide feedback, quality assessment and improvement.

Pre-QA quality planning activities play a leading role in this quality engineering process, although the execution of selected QA activities usually consumes the most resources. Quality goals need to be set so that we can manage the QA activities and stop them when the quality goals are met. QA strategies need to be selected, before we can carry out specific QA activities, collect data, perform analysis, and provide feedback.

There are two kinds of feedback in this quality engineering process, both the short-term direct feedback to the QA activities and the long-term feedback to the overall quality engineering process. The short-term feedback to QA activities typically provides information for progress tracking, activity scheduling, and identification of areas that need special attentions. For example, various models and tools were used to provide test effort tracking, reliability monitoring, and identification of low-reliability areas for various software products developed in the IBM Software Solutions Toronto Lab to manage their testing process.

The long-term feedback to the overall quality engineering process comes in two forms:

* Feedback to quality planning so that necessary adjustment can be made to quality goals and QA strategies. For example, if the current quality goals are unachievable, alternative goals need to be negotiated. If the selected QA strategy is inappropriate, a new or modified strategy needs to be selected. Similarly, such adjustments may also be applied to future projects instead of the current project.
* Feedback to the quality assessment and improvement activities. For example, the modeling results may be highly unstable, which may well be an indication of the model inappropriateness. In this case, new or modified models need to be used, probably on screened or pre-processed data.

**In-QA Activities**

Executing planned QA activities and handling discovered defects. In addition to performing selected QA activities, an important part of this normal execution is to deal with the discovered problems.

**Post-QA Activities**

This is Quality measurement, assessment and improvement. These are the activities that are carried out after normal QA activities have started but not as part of these normal activities. The primary purpose of these activities is to provide quality assessment and feedback so that various management decisions can be made, and possible quality improvement initiatives can be carried out. Notice here that “post-QA” does not mean after the finish of QA activities. In fact, many of the measurement and analysis activities are carried out parallel to QA activities after they are started. In addition, pre-QA activities may overlap with the normal QA activities as well.

# **LECTURE 8: Software Testing**

## **8.1 Software Testing**

Similar to the situation for many physical systems and products, the purpose of software testing is to ensure that the software systems would work as expected when they are used by their target customers and users. The most natural way to show this fulfillment of expectations is to demonstrate their operation through some “dry-runs” or controlled experimentation in laboratory settings before the products are released or delivered. In the case of software products, such controlled experimentation through program execution is generally called testing.

## **8.2 Principle of Software Testing**

A number of testing principles have been suggested over the past 40 years and offer general  
guidelines common for all testing.

**Principle 1 – Testing shows presence of defects**: Testing can prove the presence of defects but cannot prove the absence of defects. Even after testing the application or product thoroughly we cannot say that the product is 100% defect free. Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, it is not a proof of correctness.

**Principle 2 – Complete/Exhaustive testing is impossible:** Testing everything (all combinations of inputs and preconditions) is not feasible. Instead of exhaustive testing, risk analysis, time & cost and priorities should be used to focus testing efforts.

**Principle 3 – Early testing:** To find defects early, testing activities should be started as early as possible in the software development life cycle and should be focused on defined objectives. When defects are found earlier in the lifecycle, they are much easier and cheaper to fix

**Principle 4 – Defect clustering:** A small number of modules usually contains most of the defects discovered during pre-release testing or is responsible for most of the operational failures. There is NO equal distribution of defects within one test object. The place where defect occurs, it’s likely to find some more. The testing process must be flexible and respond to this behavior.

**Principle 5 – Pesticide paradox:** If the same tests are repeated over and over again, eventually the same set of test cases will no longer find any new defects. To overcome this “pesticide paradox”, test cases need to be regularly reviewed and revised, and new and different tests need to be written to exercise different parts of the software or system to find potentially more defects.

**Principle 6 – Testing is context dependent:** Testing is done differently in different contexts. For example, safety-critical software is tested differently from an e-commerce site

**Principle 7 – Absence-of-errors fallacy:** Finding and fixing defects does not help if the system built is unusable and does not fulfill the users’ needs and expectations. Just because testing didn’t find any defects in the software, it does not mean that the software is ready to be shipped.

## **Test Case, Test Plan and Test Suite**

A test plan is a document that describes the objectives, scope, approach, resources, schedule and focus of software testing activities. A test plan gives detailed testing information regarding an upcoming testing effort. In other words, a test plan is a systematic approach to testing a system and typically contains a detailed understanding of what the eventual workflow will be. Organizations may follow some standard test plan outlines, or they can have their own customized test plan outlines.

A test case is a document that describes an input, action, or event, and its expected results, in order to determine if a feature of an application is working correctly. In other words, a test case is a document specifying inputs, predicted results and a set of execution conditions for a test item. Different organizations may use different test case formats.

Test suit is the collection of individual test cases that will be run in a test sequence until some stopping criteria are satisfied is called a test suite. It involves the construction and allocation of individual test cases in some systematic way based on the specific testing techniques used. Another way to obtain a test suite is through reuse of test cases for earlier versions of the same product. This kind of testing is commonly referred to as regression testing. In general, all the test cases should form an integrated suite, regardless of their origin, how they are derived, and what models are used to derive them.

## **8.4 Testing Levels**

Testing is performed at different levels involving the complete system or parts of it throughout the life cycle of a software product. A software system goes through four stages of testing before it is actually deployed. These four stages are known as *unit*, *integration*, *system*, and *acceptance* level testing. The first three levels of testing are performed by a number of different stakeholders in the development organization, whereas acceptance testing is performed by the customers.

In unit testing, programmers test individual program units, such as a procedures, functions, methods, or classes, in isolation. After ensuring that individual units work to a satisfactory extent, modules are assembled to construct larger subsystems by following integration testing techniques. Integration testing is jointly performed by software developers and integration test engineers. The objective of integration testing is to construct a reasonably stable system that can withstand the rigor of system-level testing. System-level testing includes a wide spectrum of testing, such as functionality testing, security testing, robustness testing, load testing, stability testing, stress testing, performance testing, and reliability testing. System testing is a critical phase in a software development process because of the need to meet a tight schedule close to delivery date, to discover most of the faults, and to verify that fixes are working and have not resulted in new faults. System testing comprises a number of distinct activities: creating a test plan, designing a test suite, preparing test environments, executing the tests by following a clear strategy, and monitoring the process of test execution.

*Regression testing* is another level of testing that is performed throughout the life cycle of a system. Regression testing is performed whenever a component of the system is modified. The key idea in regression testing is to ascertain that the modification has not introduced any new faults in the portion that was not subject to modification. To be precise, regression testing is not a distinct level of testing. Rather, it is considered as a subphase of unit, integration, and system-level testing

In regression testing, new tests are not designed. Instead, tests are selected, prioritized, and executed from the existing pool of test cases to ensure that nothing is broken in the new version of the software. Regression testing is an expensive process and accounts for a predominant portion of testing effort in the industry. It is desirable to select a subset of the test cases from the existing pool to reduce the cost. A key question is how many and which test cases should be selected so that the selected test cases are more likely to uncover new faults.

After the completion of system-level testing, the product is delivered to the customer. The customer performs their own series of tests, commonly known as *acceptance testing*. The objective of acceptance testing is to measure the quality of the product, rather than searching for the defects, which is objective of system testing. A key notion in acceptance testing is the customer’s *expectations* from the system. By the time of acceptance testing, the customer should have developed their acceptance criteria based on their own expectations from the system.

**Adequacy of Testing**

In the absence of test adequacy, developers will be forced to use ad hoc measures to decide when to stop testing. Some examples of ad hoc measures for stopping testing are as follows

* Stop when the allocated time for testing expires.
* Stop when it is time to release the product.
* Stop when all the test cases execute without revealing faults.

# **LECTURE 9: Testing Overview**

## **9.1 Debugging Techniques**

The programmer, after a program failure, identifies the corresponding fault and fixes it. The process of determining the cause of a failure is known as *debugging*. Debugging occurs as a consequence of a test revealing a failure.

**Brute Force:** The brute-force approach to debugging is preferred by many programmers. Here, “let the computer find the error” philosophy is used. Print statements are scattered throughout the source code. These print statements provide a crude trace of the way the source code has executed. The availability of a good debugging tool makes these print statements redundant. A dynamic debugger allows the software engineer to navigate by stepping through the code, observe which paths have executed, and observe how values of variables change during the controlled execution. A good tool allows the programmer to assign values to several variables and navigate step by step through the code. Instrumentation code can be built into the source code to detect problems and to log intermediate values of variables for problem diagnosis. One may use a memory dump after a failure has occurred to understand the final state of the code being debugged. The log and memory dump are reviewed to understand what happened and how the failure occurred.

**Cause Elimination:** The cause elimination approach can be best described as a process involving *induction* and *deduction*. In the induction part, first, all pertinent data related to the failure are collected , such as what happened and what the symptoms are. Next, the collected data are organized in terms of behavior and symptoms, and their relationship is studied to find a pattern to isolate the causes. A cause hypothesis is devised, and the above data are used to prove or disprove the hypothesis. In the deduction part, a list of all possible causes is developed in order of their likelihoods, and tests are conducted to eliminate or substantiate each cause in decreasing order of their likelihoods. If the initial tests indicate that a particular hypothesis shows promise, test data are refined in an attempt to isolate the problem as needed.

**Backtracking:** In this approach, the programmer starts at a point in the code where a failure was observed and traces back the execution to the point where it occurred. This technique is frequently used by programmers, and this is useful in small programs. However, the probability of tracing back to the fault decreases as the program size increases, because the number of potential backward paths may become too large.

## **9.2 Testing Planning and Preparation**

**Testing: Why?**

Similar to the situation for many physical systems and products, the purpose of software testing is to ensure that the software systems would work as expected when they are used by their target customers and users. The most natural way to show this fulfillment of expectations is to demonstrate their operation through some “dry-runs” or controlled experimentation in laboratory settings before the products are released or delivered. In the case of software products, such controlled experimentation through program execution is generally called testing.

Because of the relatively defect-free manufacturing process for software as compared to the development process, we focus on testing in the development process. We run or execute the implemented software systems or components to demonstrate that they work as expected. Therefore, “demonstration of proper behavior” is a primary purpose of testing, which can also be interpreted as providing evidence of quality in the context of software QA, or as meeting certain quality goals.

However, because of the ultimate flexibility of software, where problems can be corrected and fixed much more easily than traditional manufacturing of physical products and systems, we can benefit much more from testing by fixing the observed problems within the development process and deliver software products that are as defect-free as our budget or environment allows. As a result, testing has become a primary means to detect and fix software defects under most development environments, to the degree that “detecting and fixing defects” has eclipsed quality demonstration as the primary purpose of testing for many people and organizations.

To summarize, testing fulfills two primary purposes:

* to demonstrate quality or proper behavior
* to detect and fix problems

## **9.3 White-Box and Black-Box Testing**

Test cases need to be designed by considering information from several sources, such as the specification, source code, and special properties of the program’s input and output domains. This is because all those sources provide complementary information to test designers. Two broad concepts in testing, based on the sources of information for test design, are *white-box* and *black-box* testing. White-box testing techniques are also called *structural testing t*echniques, whereas black-box testing techniques are called *functional testing* techniques.

In structural testing, one primarily examines *source code* with a focus on control flow and data flow. Control flow refers to flow of control from one instruction to another. Control passes from one instruction to another instruction in a number of ways, such as one instruction appearing after another, function call, message passing, and interrupts. Conditional statements alter the normal, sequential flow of control in a program. Data flow refers to the propagation of values from one variable or constant to another variable. Definitions and uses of variables determine the data flow aspect in a program.

In functional testing, one does not have access to the internal details of a program and the program is treated as a black box. A test engineer is concerned only with the part that is accessible outside the program, that is, just the input and the externally visible outcome. A test engineer applies input to a program, observes the externally visible outcome of the program, and determines whether or not the program outcome is the expected outcome. Inputs are selected from the program’s requirements specification and properties of the program’s input and output domains. A test engineer is concerned only with the functionality and the features found in the program’s specification.

At this point it is useful to identify a distinction between the scopes of structural testing and functional testing. One applies structural testing techniques to individual units of a program, whereas functional testing techniques can be applied to both an entire system and the individual program units. Since individual programmers know the details of the source code they write, they themselves perform structural testing on the individual program units they write. On the other hand, functional testing is performed at the external interface level of a system, and it is conducted by a separate software quality assurance group.

Let us consider a program unit *U* which is a part of a larger program *P*. A program unit is just a piece of source code with a well-defined objective and well-defined input and output domains. Now, if a programmer derives test cases for testing *U* from a knowledge of the internal details of *U*, then the programmer is said to be performing structural testing. On the other hand, if the programmer designs test cases from the stated objective of the unit *U* and from his or her knowledge of the special properties of the input and output domains of *U*, then he or she is said to be performing functional testing on the same unit *U*.

The ideas of structural testing and functional testing do not give programmers and test engineers a choice of whether to design test cases from the source code or from the requirements specification of a program. However, these strategies are used by different groups of people at different times during a software’s life cycle. For example, individual programmers use both the structural and functional testing techniques to test their own code, whereas quality assurance engineers apply the idea of functional testing.

Neither structural testing nor functional testing is by itself good enough to detect most of the faults. Even if one selects all possible inputs, a structural testing technique cannot detect all faults if there are *missing paths* in a program. Intuitively, a path is said to be missing if there is no code to handle a possible condition. Similarly, without knowledge of the structural details of a program, many faults will go undetected. Therefore, a combination of both structural and functional testing techniques must be used in program testing.

# **LECTURE 10: Generic Testing Process**

## **10.1 Generic Testing Process**

The basic concepts of testing can be best described in the context of the major activities involved in testing.

* Test planning and preparation, which set the goals for testing, select an overall testing strategy, and prepare specific test cases and the general test procedure.
* Test execution and related activities, which also include related observation and measurement of product behavior.
* Analysis and follow-up, which include result checking and analysis to determine if a failure has been observed, and if so, follow-up activities are initiated and monitored to ensure removal of the underlying causes, or faults, that led to the observed failures in the first place.

The major test activities are centered around test execution or performing the actual tests. At a minimum, testing involves executing the software and communicating the related observations. In fact, many forms of informal testing include just this middle group of activities related to test execution, with some informal ways to communicate the results and fix the defects, but without much planning and preparation. In all forms of systematic testing, the other two activity groups, particularly test planning and preparation activities, play a much more important role in the overall testing process and activities.

## **10.2 Major Activities in Generic Testing Process**

Because of the increasing size and complexity of today’s software products, informal testing without much planning and preparation becomes inadequate. Important functions, features, and related software components and implementation details could be easily overlooked in such informal testing. Therefore, there is a strong need for planned, monitored, managed and optimized testing strategies based on1 systematic considerations for quality, formal models, and related techniques. Test cases can be planned and prepared using such testing strategies, and test procedures need to be prepared and followed. Test planning and preparation include the following sub-activities:

***Goal setting:*** This is similar to the goal setting for the overall quality engineering process described in Chapter *5.* However, it is generally more concrete here, because the quality views and attributes have been decided by the overall quality engineering process. What remains to be done is the specific testing goals, such as reliability or coverage goals, to be used as the exit criteria.

***Test case preparation:*** This is the activity most people naturally associate with test preparation. It includes constructing new test cases or generating them automatically, selecting from existing ones for legacy products, and organizing them in some systematic ways for easy execution and management. In most systematic testing, these test cases need to be constructed, generated, or selected based on some formal models associated with formal testing techniques.

***Test procedure preparation:*** This is an important activity for test preparation. For systematic testing on a large scale for most of today’s software products and software intensive systems, a formal procedure is more of a necessity than a luxury. It can be defined and followed to ensure effective test execution, problem handling and resolution, and the overall test process management.

**Testing Suite Preparation and Management**

The collection of individual test cases that will be run in a test sequence until some stopping criteria are satisfied is called a *test suite.* Test suite preparation involves the construction and allocation of individual test cases in some systematic way based on the specific testing techniques used. For example, when usage-based statistical testing is planned, the test case allocation will be determined by the operational profiles (OPs) constructed as the testing models, in proportion to individual usage probabilities. Similarly, when coverage-based testing is planned, the specific coverage criteria would dictate the allocation of test cases. For example, in control flow testing not involving loops, the exact number of test cases is determined by the number of paths for all-path coverage.

Another way to obtain a test suite is through reuse of test cases for earlier versions of the same product. This kind of testing is commonly referred to as *regression testing.* It ensures that common functionalities are still supported satisfactorily in addition to satisfactory performance of new functionalities. Special types of formal models are typically used to make the selection from existing test cases.

In general, all the test cases should form an integrated suite, regardless of their origins, how they are derived, and what models are used to derive them. Sometimes, the test suite may evolve over time and its formation may overlap with the actual testing. In fact, in some testing techniques, test cases can be constructed dynamically, or “on-the-fly”, during test execution. But even for such testing, some planning of the test cases and test suite is still necessary, at least to determine the specific method for dynamic test case construction and the precise stopping criteria. For most of the testing techniques we cover in this book, a significant part of test preparation must be done before actual testing starts.

In general, test cases cost time, effort, and expertise to be obtained, and are too valuable to be thrown away. It is worthwhile to spend some addition effort and resource to save them, organize them, and manage them as a test suite for easy reuse in the future. Test suite management includes managing the collection of both the existing test cases and the newly constructed ones. At a minimum, some consistent database for the test suite needs to be kept and shared by people who are working on similar areas. Some personnel information can also be kept in the test suite, such as the testers who designed specific test cases, to better supported future use of this test suite. The information contained in the test suite constitutes an indexed database with important information about individual test cases in the test suite, as well as pointers to actual test cases. The actual test cases, in turn, contains more detailed information about the exact scenario, test input, expected output and behavior, etc.

# **LECTURE 11: Test Management and Test Automation**

## **11.1 Organization and management**

The test activities need to be managed by people with a good understanding of the testing techniques and processes. The feedback derived from analyses of measurement data needs to be used to help with various management decisions, such as product release, and to help quality improvement. Test managers are involved in these activities. Testers and testing teams can be organized into various different structures, but basically following either a horizontal or a vertical model:

* ***A vertical model*** would organize around a product, where dedicated people perform one or more testing tasks for the product. For example, one or more teams can perform all the different types of testing for the product, from unit testing up to acceptance testing.
* ***A horizontal model*** is used in some large organizations so that a testing team only performs one kind of testing for many different products within the organization. For example, different products may share the same system testing team.

Depending on the demand for testers by different projects, staffing level may vary over time. In the vertical model, as the product development shifts from one phase to another or as the development focus shifts from one area to another, project personnel could be reassigned to perform different tasks. One common practice in industry is to use programmers to perform various testing tasks when testing phase peaks. This practice may create various problems related to staffing management: If not done carefully, it may also lead to project delays, as in Brooks’ famous observation that adding people to a late project will make it later. The mismatch between people’s expertise and their assignments may also result in more defects passing through the testing phase to cause additional in-field problems. This fact is part of the reason for people to adopt the horizontal model where staffing level variations can generally be better managed due to the different schedules and demands by different projects.

In reality, a mixture of the two is often used in large software organizations with low-level testing performed by dedicated testers or testing teams, system testing shared across similar products, and general project support provided by a centralized support unit for the entire organization. The general project support includes process, technology, and tool support necessary for formal development and testing. This centralized support unit resembles the so-called experience factory that also packages experience and lessons learned from development for more effective future use.

## **11.2 Test Execution, Result Checking, and Measurement**

The key to the overall test execution is the smooth transition from one test run to another, which also requires us to allocate all the required resources to ensure that individual test runs can be started, executed, and finished, and related problems can be handled seamlessly. General steps in test execution include:

1. Allocating test time and resources.
2. Invoking and running tests and collecting execution information and measurements.
3. Checking testing results and identifying system failures.

The step to invoke and run tests is fairly straightforward with well-prepared test cases or already sensitized test cases. We can simply provide the input variable values over the whole execution duration as required and as already precisely specified in these test cases. The sequence of test runs can follow the pre-planned sequence described in test procedure.

Test time and resources allocation is most closely related to the test planning and preparation activities described in the previous section. Although the allocation could be planned or even carried out at the previous stage, the monitoring, adjustment, and management of these resources need to be carried out during test execution. Test time allocation and management are closely related to people’s roles and responsibility in carrying out specific testing activities. Managing other test resources primarily involves the environmental set up and related facility management. For pure software systems, this is fairly straightforward, with the environment setup to include the hardware configuration and software environment that the finished product will operate within. Sometimes, limited number of simulation programs or hardware simulators can be used for testing some product components, but the overall system testing would very much resemble the actual operational environment. Once the general system configuration is decided, the facility management is mainly the allocation and monitoring of testing time on these facilities.

## **11.3 Test Automation**

Test automation aims to automate some manual tasks with the use of some software tools. The demand for test automation is strong, because purely manual testing from start to finish can be tedious and error prone. On the other hand, long standing theoretical results tell us that no fully automated testing is possible. However, some level of automation for individual activities is possible, and can be supported by various commercial tools or tools developed within large organizations. The key in the use of test automation to relieve people of tedious and repetitive tasks and to improve overall testing productivity is to first examine what is possible, feasible, and economical, and then to set the right expectations and goals. Various issues related to test automation include:

* specific needs and potential for automation
* selection of existing testing tools, if available
* possibility and cost of constructing specific test automation tools
* availability of user training for these tools and time/effort needed
* overall cost, including costs for tool acquisition, support, training, and usage
* impact on resource, schedule, and project management.

**Automation for test execution**

Among the three major test activities, preparation, execution, and follow-up, execution is a prime candidate for automation. In fact, this is the area in which the earliest test automation tools found some unequivocal successes. For example, various semi-automatic debugging tools or debuggers allow testers to set and reset variable values and execution states during execution and observe the dynamic execution behavior at different observation points. These tools are semi-automatic because testers are still involved in test execution intervention.

Many of the modern test automation tools can be considered as enhanced debuggers that work for larger products, automate more individual testing activities, and are generally more flexible and more tailorable than earlier debuggers. Various automated task sequencing tools for job transfer from one test run to another work in much the same way as job dispatcher/scheduler in various operating systems. In fact, most such test run sequencing tools are platform-specific and are often constructed within testing organizations using some system utilities or APIs (application program interfaces).

**Automation for test planning and preparation**

In test planning and preparation, the potential for automation is different for different sub activities. The overall planning part can only be carried out by experienced personal with expertise in planning and management as well as a good understanding of testing and development technologies. Not much automation can be achieved in these sub-activities, nor is there a high demand for automation here. Similarly, test procedure planning is primarily done by experts, although the planned procedure can be later enforced and automated during actual test execution with the help of various test execution automation tools we discussed above.

Test case preparation is the area where there is some realistic potential for automation. For example, in testing of legacy products, various automated analysis can be performed to compare the current version of the product with its previous versions, and to screen the existing test suites to select the ones for regression testing. For construction of new test cases, automation is also possible. For example, in the T3 tool we mentioned above for test execution support, a script can be provided to generate different workload for testing, which effectively generates test cases and related test runs dynamically from test script. However, the test scripts, which are high-level descriptions of what to test, need to be constructed in the first place by the experienced testers. These test scripts are usually much simpler and shorter, thus much less costly to generate than actual test cases. Consequently, a semi-automated test case generation is supported in this case.

In general, test scripts or test cases are based on some formal models. The model construction for different test techniques requires high levels of human intelligence and expertise and is therefore not easily automated. However, some individual steps in model construction can be automated, such as some automated data gathering, graphical or other aids for modeling, etc. For small-scale programs, some tools can be used to generate certain models and test cases directly, much like using compilers to generate object code from source code. However, these tools cannot scale up to large software systems. In addition, in most of the models, various decisions need to be made and parameters need to be selected for specific model variations, which can only be carried out by people with proper expertise.

# **LECTURE 12: Review Lectures**

# **LECTURE 13: Unit Testing**

## **13.1 Basic Concepts of Unit Testing**

In this lesion we consider the first level of testing, that is, unit testing. Unit testing refers to testing program units in isolation. However, there is no consensus on the definition of a unit. Some examples of commonly understood units are functions, procedures, or methods. Even a class in an object-oriented programming language can be considered as a program unit. Syntactically, a program unit is a piece of code, such as a function or method of class, that is invoked from outside the unit and that can invoke other program units. Moreover, a program unit is assumed to implement a well-defined function providing a certain level of abstraction to the implementation of higher-level functions. The function performed by a program unit may not have a direct association with a system-level function. Thus, a program unit may be viewed as a piece of code implementing a “low”-level function. In this chapter, we use the terms unit and module interchangeably.

Now, given that a program unit implements a function, it is only natural to test the unit before it is integrated with other units. Thus, a program unit is tested in isolation, that is, in a stand-alone manner. There are two reasons for testing a unit in a stand-alone manner. First, errors found during testing can be attributed to a specific unit so that it can be easily fixed. Moreover, unit testing removes dependencies on other program units. Second, during unit testing it is desirable to verify that each distinct execution of a program unit produces the expected result. In terms of code details, a distinct execution refers to a distinct path in the unit. Ideally, all possible—or as much as possible—distinct executions are to be considered during unit testing. This requires careful selection of input data for each distinct execution. A programmer has direct access to the input vector of the unit by executing a program unit in isolation. This direct access makes it easier to execute as many distinct paths as desirable or possible. If multiple units are put together for testing, then a programmer needs to generate test input with indirect relationship with the input vectors of several units under test. The said indirect relationship makes it difficult to control the execution of distinct paths in a chosen unit.

Unit testing has a limited scope. A programmer will need to verify whether or not a code works correctly by performing unit-level testing. Intuitively, a programmer needs to test a unit as follows:

* Execute every line of code. This is desirable because the programmer needs to know what happens when a line of code is executed. In the absence of such basic observations, surprises at a later stage can be expensive.
* Execute every predicate in the unit to evaluate them to true and false separately.
* Observe that the unit performs its intended function and ensure that it contains no known errors.

Unit testing is performed by the programmer who writes the program unit because the programmer is intimately familiar with the internal details of the unit. The objective for the programmer is to be satisfied that the unit works as expected. Since a programmer is supposed to construct a unit with no errors in it, a unit test is performed by him or her to their satisfaction in the beginning and to the satisfaction of other programmers when the unit is integrated with other units. This means that all programmers are accountable for the quality of their own work, which may include both new code and modifications to the existing code. The idea here is to push the quality concept down to the lowest level of the organization and empower each programmer to be responsible for his or her own quality. Therefore, it is in the best interest of the programmer to take preventive actions to minimize the number of defects in the code. The defects found during unit testing are internal to the software development group and are not reported up the personnel hierarchy to be counted in quality measurement metrics. The source code of a unit is not used for interfacing by other group members until the programmer completes unit testing and checks in the unit to the version control system. Unit testing is conducted in two complementary phases:

* Static unit testing
* Dynamic unit testing

## **13.2 Static Unit Testing**

Static unit testing is conducted as a part of a larger philosophical belief that a software product should undergo a phase of inspection and correction at each milestone in its life cycle. At a certain milestone, the product need not be in its final form. For example, completion of coding is a milestone, even though coding of all the units may not make the desired product. After coding, the next milestone is testing all or a substantial number of units forming the major components of the product. Thus, before units are individually tested by actually executing them, those are subject to usual review and correction as it is commonly understood. The idea behind review is to find the defects as close to their points of origin as possible so that those defects are eliminated with less effort, and the interim product contains fewer defects before the next task is undertaken. In static unit testing, code is reviewed by applying techniques commonly known as *inspection* and *walkthrough*.

* **Inspection:** It is a step-by-step peer group review of a work product, with each step checked against predetermined criteria.
* **Walkthrough:** It is a review where the author leads the team through a manual or simulated execution of the product using predefined scenarios.

Regardless of whether a review is called an inspection or a walkthrough, it is a systematic approach to examining source code in detail. The goal of such an exercise is to assess the quality of the software in question, *not* the quality of the process used to develop the product. Reviews of this type are characterized by significant preparation by groups of designers and programmers with varying degree of interest in the software development project. Code examination can be time consuming. Moreover, no examination process is perfect. Examiners may take shortcuts, may not have adequate understanding of the product, and may accept a product which should not be accepted. Nonetheless, a well-designed code review process can find faults that may be missed by execution-based testing. The key to the success of code review is to divide and conquer, that is, having an examiner inspect small parts of the unit in isolation, while making sure of the following: (i) nothing is overlooked and (ii) the correctness of all examined parts of the module implies the correctness of the whole module. The decomposition of the review into discrete steps must assure that each step is simple enough that it can be carried out without detailed knowledge of the others.

The objective of code review is to review the code, not to evaluate the author of the code. A clash may occur between the author of the code and the reviewers, and this may make the meetings unproductive. Therefore, code review must be planned and managed in a professional manner. There is a need for mutual respect, openness, trust, and sharing of expertise in the group. The general guidelines for performing code review consists of six steps: readiness, preparation, examination, rework, validation, and exit. The input to the readiness step is the criteria that must be satisfied before the start of the code review process, and the process produces two types of documents, a change request (CR) and a report. These steps and documents are explained in the following.

**Step 1: Readiness** The author of the unit ensures that the unit under test is ready for review. A unit is said to be ready if it satisfies the following criteria:

* **Completeness:** All the code relating to the unit to be reviewed must be available. This is because the reviewers are going to read the code and try to understand it. It is unproductive to review partially written code or code that is going to be significantly modified by the programmer.
* **Minimal Functionality:** The code must compile and link. Moreover, the code must have been tested to some extent to make sure that it performs its basic functionalities.
* **Readability:** Since code review involves actual reading of code by other programmers, it is essential that the code is highly readable. Some code characteristics that enhance readability are proper formatting, using meaningful identifier names, straightforward use of programming language constructs, and an appropriate level of abstraction using function calls. In the absence of readability, the reviewers are likely to be discouraged from performing the task effectively.
* **Complexity:** There is no need to schedule a group meeting to review straightforward code which can be easily reviewed by the programmer. The code to be reviewed must be of sufficient complexity to warrant group review. Here, complexity is a composite term referring to the number of conditional statements in the code, the number of input data elements of the unit, the number of output data elements produced by the unit, real-time processing of the code, and the number of other units with which the code communicates.
* **Requirements and Design Documents:** The latest approved version of the low-level design specification or other appropriate descriptions of program requirements should be available. These documents help the reviewers in verifying whether or not the code under review implements the expected functionalities. If the low-level design document is available, it helps the reviewers in assessing whether or not the code appropriately implements the design.

All the people involved in the review process are informed of the group review meeting schedule two or three days before the meeting. They are also given a copy of the work package for their perusal. Reviews are conducted in bursts of 1–2 hours. Longer meetings are less and less productive because of the limited attention span of human beings. The rate of code review is restricted to about 125 lines of code (in a high-level language) per hour. Reviewing complex code at a higher rate will result in just glossing over the code, thereby defeating the fundamental purpose of code review. The composition of the review group involves a number of people with different roles. These roles are explained as follows:

* **Moderator:** A review meeting is chaired by the moderator. The moderator is a trained individual who guides the pace of the review process. The moderator selects the reviewers and schedules the review meetings. Myers suggests that the moderator be a member of a group from an unrelated project to preserve objectivity.
* **Author:** This is the person who has written the code to be reviewed.
* **Presenter:** A presenter is someone other than the author of the code. The presenter reads the code beforehand to understand it. It is the presenter who presents the author’s code in the review meeting for the following reasons: (i) an additional software developer will understand the work within the software organization; (ii) if the original programmer leaves the company with a short notice, at least one other programmer in the company knows what is being done; and (iii) the original programmer will have a good feeling about his or her work, if someone else appreciates their work. Usually, the presenter appreciates the author’s work.
* **Recordkeeper:** The recordkeeper documents the problems found during the review process and the follow-up actions suggested. The person should be different than the author and the moderator.
* **Reviewers:** These are experts in the subject area of the code under review. The group size depends on the content of the material under review. As a rule of thumb, the group size is between 3 and 7. Usually this group does not have manager to whom the author reports. This is because it is the author’s ongoing work that is under review, and neither a completed work nor the author himself is being reviewed.
* **Observers:** These are people who want to learn about the code under review. These people do not participate in the review process but are simply passive observers.

**Step 2: Preparation** Before the meeting, each reviewer carefully reviews the work package. It is expected that the reviewers read the code and understand its organization and operation before the review meeting. Each reviewer develops the following:

* **List of Questions:** A reviewer prepares a list of questions to be asked, if needed, of the author to clarify issues arising from his or her reading. A general guideline of what to examine while reading the code.
* **Potential CR:** A reviewer may make a formal request to make a change. These are called change requests rather than defect reports. At this stage, since the programmer has not yet made the code public, it is more appropriate to make suggestions to the author to make changes, rather than report a defect. Though CRs focus on defects in the code, these reports are not included in defect statistics related to the product.
* **Suggested Improvement Opportunities:** The reviewers may suggest how to fix the problems, if there are any, in the code under review. Since reviewers are experts in the subject area of the code, it is not unusual for them to make suggestions for improvements.

**Step 3: Examination** The examination process consists of the following activities:

* The author makes a presentation of the procedural logic used in the code, the paths denoting major computations, and the dependency of the unit under review on other units.
* The presenter reads the code line by line. The reviewers may raise questions if the code is seen to have defects. However, problems are not resolved in the meeting. The reviewers may make general suggestions on how to fix the defects, but it is up to the author of the code to take corrective measures after the meeting ends.
* The recordkeeper documents the change requests and the suggestions for fixing the problems, if there are any. A CR includes the following details:

1. Give a brief description of the issue or action item.
2. Assign a priority level (major or minor) to a CR.
3. Assign a person to follow up the issue. Since a CR documents a potential problem, there is a need for interaction between the author of the code and one of the reviewers, possibly the reviewer who made the CR.
4. Set a deadline for addressing a CR.

* The moderator ensures that the meeting remains focused on the review process. The moderator makes sure that the meeting makes progress at a certain rate so that the objective of the meeting is achieved. At the end of the meeting, a decision is taken regarding whether or not to call another meeting to further review the code. If the review process leads to extensive rework of the code or critical issues are identified in the process, then another meeting is generally convened. Otherwise, a second meeting is not scheduled, and the author is given the responsibility of fixing the CRs.

**Step 4: Rework** At the end of the meeting, the recordkeeper produces a summary of the meeting that includes the following information:

* A list of all the CRs, the dates by which those will be fixed, and the names of the persons responsible for validating the CRs
* A list of improvement opportunities
* The minutes of the meeting (optional) A copy of the report is distributed to all the members of the review group. After the meeting, the author works on the CRs to fix the problems. The author documents the improvements made to the code in the CRs. The author makes an attempt to address the issues within the agreed-upon time frame using the prevailing coding conventions.

**Step 5: Validation** the CRs are independently validated by the moderator or another person designated for this purpose. The validation process involves checking the modified code as documented in the CRs and ensuring that the suggested improvements have been implemented correctly. The revised and final version of the outcome of the review meeting is distributed to all the group members.

**Step 6: Exit** Summarizing the review process, it is said to be complete if all of the following actions have been taken:

* Every line of code in the unit has been inspected.
* If too many defects are found in a module, the module is once again reviewed after corrections are applied by the author. As a rule of thumb, if more than 5% of the total lines of code are thought to be contentious, then a second review is scheduled.
* The author and the reviewers reach a consensus that when corrections have been applied the code will be potentially free of defects.
* All the CRs are documented and validated by the moderator or someone else. The author’s follow-up actions are documented.
* A summary report of the meeting including the CRs is distributed to all the members of the review group.

## **13.4 Code Review Metrics**

*Code Review Metrics* It is important to collect measurement data pertinent to a review process, so that the review process can be evaluated, made visible to the upper management as a testing strategy, and improved to be more effective. Moreover, collecting metrics during code review facilitates estimation of review time and resources for future projects. Thus, code review is a viable testing strategy that can be effectively used to improve the quality of products at an early stage. The following metrics can be collected from a code review:

* Number of lines of code (LOC) reviewed per hour
* Number of CRs generated per thousand lines of code (KLOC)
* Number of CRs generated per hour
* Total number of CRs generated per project
* Total number of hours spent on code review per project

## **13.5 Defect Prevention**

It is in the best interest of the programmers in particular and the company in general to reduce the number of CRs generated during code review. This is because CRs are indications of potential problems in the code, and those problems must be resolved before different program units are integrated. Addressing CRs means spending more resources and potentially delaying the project. Therefore, it is essential to adopt the concept of defect prevention during code development. In practice, defects are inadvertently introduced by programmers. Those accidents can be reduced by taking preventive measures. It is useful to develop a set of guidelines to construct code for defect minimization as explained in the following. These guidelines focus on incorporating suitable mechanisms into the code:

* Build internal diagnostic tools, also known as *instrumentation code*, into the units. Instrumentation codes are useful in providing information about the internal states of the units. These codes allow programmers to realize built-in tracking and tracing mechanisms. Instrumentation plays a passive role in dynamic unit testing. The role is passive in the sense of observing and recording the internal behavior without actively testing a unit.
* Use standard controls to detect possible occurrences of error conditions. Some examples of error detection in the code are divides by zero and array index out of bounds.
* Ensure that code exists for all return values, some of which may be invalid. Appropriate follow-up actions need to be taken to handle invalid return values.
* Ensure that counter data fields and buffer overflow and underflow are appropriately handled.

# **LECTURE 14: Unit Testing**

## **14.1 Dynamic Unit Testing**

Execution-based unit testing is referred to as dynamic unit testing. In this testing, a program unit is actually executed in isolation, as we commonly understand it. However, this execution differs from ordinary execution in the following way:

1. A unit under test is taken out of its actual execution environment.
2. The actual execution environment is emulated by writing more code (explained later in this section) so that the unit and the emulated environment can be compiled together.
3. The above compiled aggregate is executed with selected inputs. The outcome of such an execution is collected in a variety of ways, such as straightforward observation on a screen, logging on files, and software instrumentation of the code to reveal run time behavior. The result is compared with the expected outcome. Any difference between the actual and expected outcome implies a failure and the fault is in the code.

An environment for dynamic unit testing is created by emulating the context of the unit under test. The context of a unit test consists of two parts: (i) a caller of the unit and (ii) all the units called by the unit. The environment of a unit is emulated because the unit is to be tested in isolation and the emulating environment must be a simple one so that any fault found as a result of running the unit can be solely attributed to the unit under test. The caller unit is known as a *test driver*, and all the emulations of the units called by the unit under test are called *stubs*. The test driver and the stubs are together called *scaffolding*. The functions of a test driver and a stub are explained as follows:

* **Test Driver:** A test driver is a program that invokes the unit under test. The unit under test executes with input values received from the driver and, upon termination, returns a value to the driver. The driver compares the actual outcome, that is, the actual value returned by the unit under test, with the expected outcome from the unit and reports the ensuing test result. The test driver functions as the *main* unit in the execution process. The driver not only facilitates compilation, but also provides input data to the unit under test in the expected format.
* **Stubs:** A stub is a “dummy subprogram” that replaces a unit that is called by the unit under test. Stubs replace the units called by the unit under test. A stub performs two tasks. First, it shows an evidence that the stub was, in fact, called. Such evidence can be shown by merely printing a message. Second, the stub returns a precomputed value to the caller so that the unit under test can continue its execution.

The low-level design document provides guidance for the selection of input test data that are likely to uncover faults. Selection of test data is broadly based on the following techniques:

* **Control Flow Testing:** The following is an outline of control flow testing: (i) draw a control flow graph from a program unit; (ii) select a few control flow testing criteria; (iii) identify paths in the control flow graph to satisfy the selection criteria; (iv) derive path predicate expressions from the selected paths; and (v) by solving the path predicate expression for a path, generate values of the inputs to the program unit that are considered as a test case to exercise the corresponding path.
* **Data Flow Testing:** The following is an outline of data flow testing: (i) draw a data flow graph from a program unit; (ii) select a few data flow testing criteria; (iii) identify paths in the data flow graph to satisfy the selection criteria; (iv) derive path predicate expressions from the selected paths; and (v) by solving the path predicate expression, generate values of the inputs to the program unit that are considered as a test case to exercise the corresponding path. Chapter 5 discusses data flow testing in greater detail.
* **Domain Testing:** In control flow and data flow testing, no specific types of faults are explicitly considered for detection. However, domain testing takes a new approach to fault detection. In this approach, a category of faults called *domain errors* are defined and then test data are selected to catch those faults.
* **Functional Program Testing:** In functional program testing one performs the following steps: (i) identify the input and output domains of a program; (ii) for a given input domain, select some *special* values and compute the expected outcome; (iii) for a given output domain, select some *special* values and compute the input values that will cause the unit to produce those output values; and (iv) consider various combinations of the input values chosen above.

## **14.2 Mutation Testing**

Mutation testing has a rich and long history. It can be traced back to the late 1970s. Mutation testing is a technique that focuses on measuring the adequacy of test data (or test cases). The original intention behind mutation testing was to expose and locate weaknesses in test cases. Thus, mutation testing is a way to measure the quality of test cases, and the actual testing of program units is an added benefit. Mutation testing is not a testing strategy like control flow or data flow testing. It should be used to supplement traditional unit testing techniques.

A mutation of a program is a modification of the program created by introducing a single, small, legal syntactic change in the code. A modified program so obtained is called a *mutant*. The term mutant has been borrowed from biology. Some of these mutants are equivalent to the original program, whereas others are faulty. A mutant is said to be *killed* when the execution of a test case causes it to fail and the mutant is considered to be *dead*.

Some mutants are *equivalent* to the given program, that is, such mutants always produce the same output as the original program. In the real world, large programs are generally faulty, and test cases too contain faults. The result of executing a mutant may be different from the expected result, but a test suite does not detect the failure because it does not have the right test case. In this scenario the mutant is called *killable* or *stubborn*, that is, the existing set of test cases is insufficient to kill it. A *mutation score* for a set of test cases is the percentage of nonequivalent mutants killed by the test suite. The test suite is said to be *mutation adequate* if its mutation score is 100%. Mutation analysis is a two-step process:

1. The adequacy of an existing test suite is determined to distinguish the given program from its mutants. A given test suite may not be adequate to distinguish all the nonequivalent mutants. As explained above, those nonequivalent mutants that could not be identified by the given test suite are called stubborn mutants.
2. New test cases are added to the existing test suite to kill the stubborn mutants. The test suite enhancement process iterates until the test suite has reached a desired level of mutation score.

## **14.3 Debugging Techniques**

The programmer, after a program failure, identifies the corresponding fault and fixes it. The process of determining the cause of a failure is known as *debugging*. Debugging occurs as a consequence of a test revealing a failure.

**Brute Force:** The brute-force approach to debugging is preferred by many programmers. Here, “let the computer find the error” philosophy is used. Print statements are scattered throughout the source code. These print statements provide a crude trace of the way the source code has executed. The availability of a good debugging tool makes these print statements redundant. A dynamic debugger allows the software engineer to navigate by stepping through the code, observe which paths have executed, and observe how values of variables change during the controlled execution. A good tool allows the programmer to assign values to several variables and navigate step by step through the code. Instrumentation code can be built into the source code to detect problems and to log intermediate values of variables for problem diagnosis. One may use a memory dump after a failure has occurred to understand the final state of the code being debugged. The log and memory dump are reviewed to understand what happened and how the failure occurred.

**Cause Elimination:** The cause elimination approach can be best described as a process involving *induction* and *deduction*. In the induction part, first, all pertinent data related to the failure are collected , such as what happened and what the symptoms are. Next, the collected data are organized in terms of behavior and symptoms, and their relationship is studied to find a pattern to isolate the causes. A cause hypothesis is devised, and the above data are used to prove or disprove the hypothesis. In the deduction part, a list of all possible causes is developed in order of their likelihoods, and tests are conducted to eliminate or substantiate each cause in decreasing order of their likelihoods. If the initial tests indicate that a particular hypothesis shows promise, test data are refined in an attempt to isolate the problem as needed.

**Backtracking:** In this approach, the programmer starts at a point in the code where a failure was observed and traces back the execution to the point where it occurred. This technique is frequently used by programmers, and this is useful in small programs. However, the probability of tracing back to the fault decreases as the program size increases, because the number of potential backward paths may become too large.

## **14.4 Unit Testing in Extreme Programming**

TDD approach to code development is used in the XP methodology. The key aspect of the TDD approach is that a programmer writes low-level tests before writing production code. This is referred to as *test first* in software development. Writing test-driven units is an important concept in the XP methodology. In XP, a few unit tests are coded first, then a simple, partial system is implemented to pass the tests. Then, one more new unit test is created, and additional code is written to pass the new test, but not more, until a new unit test is created. The process is continued until nothing is left to test. The process outlined below:

**Step 1:** Pick a requirement, that is, a story.

**Step 2:** Write a test case that will verify a small part of the story and assign a fail verdict to it.

**Step 3:** Write the code that implements a particular part of the story to pass the test.

**Step 4:** Execute all tests.

**Step 5:** Rework the code and test the code until all tests pass.

**Step 6:** Repeat steps 2–5 until the story is fully implemented.

**JUNIT: Framework for Unit Testing**

The JUnit is a unit testing framework for the Java programming language designed by Kent Beck and Erich Gamma. Experience gained with JUnit has motivated the development of the TDD [22] methodology. The idea in the JUnit framework has been ported to other languages, including C# (NUnit), Python (PyUnit), Fortran (fUnit) and C++ (CPPUnit). This family of unit testing frameworks is collectively referred to as xUnit. This section will introduce the fundamental concepts of Junit to the reader.

## **14.5 Tools for Unit Testing**

Programmers can benefit from using tools in unit testing by reducing testing time without sacrificing thoroughness. The well-known tools in everyday life are an editor, a compiler, an operating system, and a debugger. However, in some cases, the real execution environment of a unit may not be available to a programmer while the code is being developed. In such cases, an emulator of the environment is useful in testing and debugging the code. Other kinds of tools that facilitate effective unit testing are as follows:

***Code Auditor:***This tool is used to check the quality of software to ensure that it meets some minimum coding standards. It detects violations of programming, naming, and style guidelines. It can identify portions of code that cannot be ported between different operating systems and processors. Moreover, it can suggest improvements to the structure and style of the source code. In addition, it counts the number of LOC which can be used to measure productivity, that is, LOC produced per unit time, and calculate defect density, that is, number of defects per KLOC.

***Bound Checker:***This tool can check for accidental writes into the instruction areas of memory or to any other memory location outside the data storage area of the application. This fills unused memory space with a signature pattern (distinct binary pattern) as a way of determining at a later time whether any of this memory space has been overwritten. The tool can issue diagnostic messages when boundary violations on data items occur. It can detect violation of the boundaries of array, for example, when the array index or pointer is outside its allowed range. For example, if an array *z* is declared to have a range from *z* [0] to *z* [99], it can detect reads and writes outside this range of storage, for example, *z* [−3] or *z* [10].

***Documenters:***These tools read source code and automatically generate descriptions and caller/callee tree diagram or data model from the source code. 4. *Interactive Debuggers:* These tools assist software developers in implementing different debugging approaches discussed in this chapter. These tools should have the trace-back and breakpoint capabilities to enable the programmers to understand the dynamics of program execution and to identify problem areas in the code. Breakpoint debuggers are based on deductive logic. Breakpoints are placed according to a heuristic analysis of code. Another popular kind of debugger is known as omniscient debugger (ODB), in which there is no deduction. It simply follows the trail of “bad” values back to their source—no “guessing” where to put the breakpoints. An ODB is like “the snake in the grass,” that is, if you see a snake in the grass and you pull its tail, sooner or later you get to its head. In contrast, breakpoint debuggers suffer from the “lizard in the grass” problem, that is, when you see the lizard and grab its tail, the lizard breaks off its tail and gets away.

# **LECTURE 15: Control Flow Testing**

## **15.1 Basic of Control Flow Testing Concepts**

Two kinds of basic statements in a program unit are *assignment* statements and *conditional* statements. An assignment statement is explicitly represented by using an assignment symbol, “ = ”, such as x = 2\*y;, where *x* and *y* are variables. Program *conditions* are at the core of conditional statements, such as if(), for() loop, while() loop, and goto. As an example, in if(x! = y), we are testing for the inequality of *x* and *y*. In the absence of conditional statements, program instructions are executed in the sequence they appear. The idea of successive execution of instructions gives rise to the concept of *control flow* in a program unit. Conditional statements alter the default, sequential control flow in a program unit. In fact, even a small number of conditional statements can lead to a complex control flow structure in a program.

Function calls are a mechanism to provide *abstraction* in program design. A call to a program function leads to control entering the called function. Similarly, when the called function executes its *return* statement, we say that control exits from the function. Though a function can have many return statements, for simplicity, one can restructure the function to have exactly one return. A program unit can be viewed as having a well-defined entry point and a well-defined exit point. The execution of a sequence of instructions from the entry point to the exit point of a program unit is called a program *path*. There can be a large, even infinite, number of paths in a program unit. Each program path can be characterized by an input and an expected output. A specific input value causes a specific program path to be executed; it is expected that the program path performs the desired computation, thereby producing the expected output value. Therefore, it may seem natural to execute as many program paths as possible. Mere execution of a large number of paths, at a higher cost, may not be effective in revealing defects. Ideally, one must strive to execute fewer paths for better effectiveness.

## **15.2 Outline of Control Flow Testing**

The overall idea of generating test input data for performing control flow testing has been depicted in Figure 4.1. The activities performed, the intermediate results produced by those activities, and programmer preferences in the test generation process are explained below.

***Inputs*:** The *source code* of a program unit and a set of *path selection criteria* are the inputs to a process for generating test data. In the following, two examples of path selection criteria are given.

**Example.** Select paths such that every statement is executed at least once.

**Example.** Select paths such that every conditional statement, for example, an if() statement, evaluates to *true* and *false* at least once on different occasions. A conditional statement may evaluate to true in one path and false in a second path.

*Generation of a Control Flow Graph*: A control flow graph (CFG) is a detailed graphical representation of a program unit. The idea behind drawing a CFG is to be able to visualize all the paths in a program unit. The process of drawing a CFG from a program unit will be explained in the following section. If the process of test generation is automated, a compiler can be modified to produce a CFG.

***Selection of Paths*:** Paths are selected from the CFG to satisfy the path selection criteria, and it is done by considering the structure of the CFG.

***Generation of Test Input Data*:** A path can be executed if and only if a certain instance of the inputs to the program unit causes all the conditional statements along the path to evaluate to true or false as dictated by the control flow. Such a path is called a *feasible* path. Otherwise, the path is said to be *infeasible*. It is essential to identify certain values of the inputs from a given path for the path to execute.

***Feasibility Test of a Path*:** The idea behind checking the feasibility of a selected path is to meet the path selection criteria. If some chosen paths are found to be infeasible, then new paths are selected to meet the criteria.

## **15.3 Control Flow Graph**

A CFG is a graphical representation of a program unit. A rectangle represents a sequential computation. A maximal sequential computation can be represented either by a single rectangle or by many rectangles, each corresponding to one statement in the source code. We label each computation and decision box with a unique integer. The two branches of a decision box are labeled with **T** and **F** to represent the true and false evaluations, respectively, of the condition within the box. We will not label a merge node, because one can easily identify the paths in a CFG even without explicitly considering the merge nodes. Moreover, not mentioning the merge nodes in a path will make a path description shorter.

## **15.4 Control-Flow Path Selection Criteria**

We assume that a control flow graph has exactly one *entry* node and exactly one *exit* node for the convenience of discussion. Each node is labeled with a unique integer value. Also, the two branches of a decision node are appropriately labeled with true (T) or false (F). We are interested in identifying entry–exit paths in a CFG. A path is represented as a sequence of computation and decision nodes from the entry node to the exit node. We also specify whether control exits a decision node via its true or false branch while including it in a path.

A CFG, can have a large number of different paths. One may be tempted to test the execution of each and every path in a program unit. For a program unit with a small number of paths, executing all the paths may be desirable and achievable as well. On the other hand, for a program unit with a large number of paths, executing every distinct path may not be practical. Thus, it is more productive for programmers to select a small number of program paths in an effort to reveal defects in the code. Given the set of all paths, one is faced with a question “What paths do I select for testing?” The concept of path selection criteria is useful is answering the above question. In the following, we state the advantages of selecting paths based on defined criteria:

* All program constructs are exercised at least once. The programmer needs to observe the outcome of executing each program construct, for example, statements, Boolean conditions, and returns.
* We do not generate test inputs which execute the same path repeatedly. Executing the same path several times is a waste of resources. However, if each execution of a program path potentially updates the *state* of the system, for example, the database state, then multiple executions of the same path may not be identical.
* We know the program features that have been tested and those not tested. For example, we may execute an if statement only once so that it evaluates to true. If we do not execute it once again for its false evaluation, we are, at least, aware that we have not observed the outcome of the program with a false evaluation of the if statement.

Now we explain the following well-known path selection criteria:

1. Select *all* paths.
2. Select paths to achieve complete *statement* coverage.
3. Select paths to achieve complete *branch* coverage.
4. Select paths to achieve *predicate* coverage.

**All-Path Coverage Criterion:** If *all* the paths in a CFG are selected, then one can detect all faults, except those due to *missing path* errors. However, a program may contain a large number of paths, or even an infinite number of paths.

**Statement Coverage Criterion:** *Statement coverage* refers to executing individual program statements and observing the outcome. We say that 100% statement coverage has been achieved if *all* the statements have been executed at least once. Complete statement coverage is the *weakest* coverage criterion in program testing. Any test suite that achieves less than statement coverage for new software is considered to be unacceptable.

**Branch Coverage Criterion:** Syntactically, a branch is an outgoing edge from a node. All the rectangle nodes have at most one outgoing branch (edge). The exit node of a CFG does not have an outgoing branch. All the diamond nodes have two outgoing branches. Covering a branch means selecting a path that includes the branch. Complete branch coverage means selecting a number of paths such that every branch is included in at least one path.

**Predicate Coverage Criterion:** We refer to the partial CFG of Figure 4.9*a* to explain the concept of predicate coverage. OB1, OB2, OB3, and OB are four Boolean variables. The program computes the values of the individual variables OB1, OB2, and OB3— details of their computation are irrelevant to our discussion and have been omitted. Next, OB is computed as shown in the CFG. The CFG checks the value of OB and executes either OBlock1 or OBlock2 depending on whether OB evaluates to true or false, respectively.

## **15.5 Containing Infeasible Path**

There are some practical problems in applying the idea of path testing. First, a CFG may contain a very large number of paths; therefore, the immediate challenge is to decide which paths to select to derive test cases. Second, it may not be feasible to execute many of the selected paths. Thus, it is useful to apply a path selection strategy: First, select as many short paths as feasible; next choose longer paths to achieve better coverage of statements, branches, and predicates. A large number of infeasible paths in a CFG complicate the process of test selection. To simplify path-based unit testing, it is useful to reduce the number of infeasible paths in a program unit through language design, program design, and program transformation.

# **LECTURE 16: Data Flow Testing**

## **16.1 Basic Concepts of Data Flow Testing**

A program unit, such as a function, accepts input values, performs computations while assigning new values to local and global variables, and, finally, produces output values. Therefore, one can imagine a kind of “flow” of data values between variables along a path of program execution. A data value computed in a certain step of program execution is expected to be used in a later step. For example, a program may open a file, thereby obtaining a value for a file pointer; in a later step, the file pointer is expected to be used. Intuitively, if the later use of the file pointer is never verified, then we do not know whether or not the earlier assignment of value to the file pointer variable is all right. Sometimes, a variable may be defined twice without a use of the variable in between. One may wonder why the first definition of the variable is never used. There are two motivations for data flow testing as follows. First, a memory location corresponding to a program variable is accessed in a desirable way. For example, a memory location may not be read before writing into the location. Second, it is desirable to verify the correctness of a data value generated for a variable—this is performed by observing that all the uses of the value produce the desired results.

## **16.2 Data Flow Anomaly**

An anomaly is a deviant or abnormal way of doing something. For example, it is an abnormal situation to successively assign two values to a variable without using the first value. Similarly, it is abnormal to use a value of a variable before assigning a value to the variable. Another abnormal situation is to generate a data value and never use it. In the following, we explain three types of abnormal situations concerning the generation and use of data values. The three abnormal situations are called **type 1**, **type 2**, and **type 3** anomalies. These anomalies could be manifestations of potential programming errors. We will explain why program anomalies need not lead to program failures.

***Defined and Then Defined Again (Type 1)*:** Consider the partial sequence of computations shown in Figure 5.1, where f1(y) and f2(z) denote functions with the inputs *y* and *z* , respectively. We can interpret the two statements in Figure 5.1 in several ways as follows:

* The computation performed by the first statement is redundant if the second statement performs the intended computation.
* The first statement has a fault. For example, the intended first computation might be w = f1(y).
* The second statement has a fault. For example, the intended second computation might be v = f2(z).
* A fourth kind of fault can be present in the given sequence in the form of a missing statement between the two. For example, v = f3(x) may be the desired statement that should go in between the two given statements. It is for the programmer to make the desired interpretation, though one can interpret the given two statements in several ways, However, it can be said that there is a data flow anomaly in those two statements, indicating that those need to be examined to eliminate any confusion in the mind of a code reader.

***Undefined but Referenced (Type 2)*:** A second form of data flow anomaly is to use an undefined variable in a computation, such as *x* = *x* −*y* −*w*, where the variable *w* has not been *initialized* by the programmer. Here, too, one may argue that though *w* has not been initialized, the programmer intended to use another initialized variable, say *y*, in place of *w*. Whatever may be the real intention of the programmer, there exists an anomaly in the use of the variable *w*, and one must eliminate the anomaly either by initializing *w* or replacing *w* with the intended variable.

***Defined but Not Referenced (Type 3)*:** A third kind of data flow anomaly is to define a variable and then to undefine it without using it in any subsequent computation. For example, consider the statement *x* = *f* (*x*, *y*) in which a new value is assigned to the variable *x*. If the value of *x* is not used in any subsequent computation, then we should be suspicious of the computation represented by *x* = *f* (*x*, *y*). Hence, this form of anomaly is called “defined but not referenced.”

# **LECTURE 17: Data Flow Testing**

## **17.1 Data Flow Graph**

In this section, we explain the main ideas in a data flow graph and a method to draw it. In practice, programmers may not draw data flow graphs by hand. Instead, language translators are modified to produce data flow graphs from program units. A data flow graph is drawn with the objective of identifying data definitions and their uses as motivated in the preceding section. Each occurrence of a data variable is classified as follows:

***Definition*:** This occurs when a value is moved into the memory location of the variable. Referring to the C function VarTypes(), the assignment statement i = x; is an example of definition of the variable *i* .

***Undefinition or Kill* :** This occurs when the value and the location become unbound. Referring to the C function VarTypes(), the first statement initializes the integer pointer variable iptr and iptr = i + x; initializes the value of the location pointed to by iptr. The second iptr = malloc(sizeof(int)); statement redefines variable iptr, thereby undefining the location previously pointed to by iptr.

***Use*:** This occurs when the value is fetched from the memory location of the variable. There are two forms of *uses* of a variable as explained below.

**Computation use (c-use):** This directly affects the computation being performed. In a c-use, a potentially new value of another variable or of the same variable is produced. Referring to the C function VarTypes(), the statement \*iptr = i + x; gives examples of c-use of variables *i* and *x*.

**Predicate use (p-use):** This refers to the use of a variable in a predicate controlling the flow of execution. Referring to the C function VarTypes(), the statement if (\*iptr > y) ... gives examples of p-use of variables *y* and iptr.

**Comparison of Testing Techniques**

So far we have discussed two major techniques for generating test data from source code, namely control flow–based path selection and data flow–based path selection. We also explained a few criteria to select paths from a control flow graph and data flow graph of a program. Programmers often randomly select test data based on their own understanding of the code they have written. Therefore, it is natural to compare the effectiveness of the three test generation techniques, namely random test selection, test selection based on control flow, and test selection based on data flow. Comparing those techniques does not seem to be an easy task. An acceptable, straightforward way of comparing them is to apply those techniques to the same set of programs with known faults and express their effectiveness in terms of the following two metrics: Number of test cases produced, Percentage of known faults detected.

# **LECTURE 18: Domain Testing**

## **18.1 Basic Concepts of Domain Testing**

Two fundamental elements of a computer program are *input domain* and *program paths*. The input domain of a program is the set of all input data to the program. A program path is a sequence of instructions from the start of the program to some point of *interest* in the program. For example, the end of the program is a point of interest. Another point of interest is when the program waits to receive another input from its environment so that it can continue its execution. In other words, a program path, or simply *path*, corresponds to some flow of control in the program. A path is said to be *feasible* if there exists an input data which causes the program to execute the path. Otherwise, the path is said to be *infeasible*. There are two broad classes of errors, namely, *computation error* and *domain error*, by combining the concepts of *input data* and program path. The two kinds of errors have been explained in the following.

***Computation Error*:** A computation error occurs when a specific input data causes the program to execute the correct, i.e., desired path, but the output value is wrong. Note that the output value can be wrong even if the desired path has been executed. This can happen due to a wrong function being executed in an *assignment* statement. For example, consider a desired path containing the statement result = f(a, b), where *a* and *b* are input values. A computation error may occur if the statement is replaced by a faulty one, such as result = f(b, a). Therefore, the result of executing the path can be erroneous because of a fault in the assignment statement, and this can happen in spite of executing a correct path.

***Domain Error*:** A domain error occurs when a specific input data causes the program to execute a *wrong*, that is, undesired, path in the program. An incorrect path can be selected by a program if there is a fault in one or more of the *conditional* statements in the program. Let us consider a conditional statement of the form *if (p) then f1() else f2()*. If there is a fault in the formulation of the predicate *p*, then the wrong function call is invoked, thereby causing an incorrect path to be executed.

## **18.2 Testing Domain Errors**

There is a fundamental difference between flow graph–based testing techniques and domain testing. By flow graph we mean control flow graph and data flow graph. The difference is explained as follows:

* Select paths from a control flow graph or a data flow graph to satisfy certain *coverage criteria*. To remind the reader, the *control flow* coverage criteria are *statement coverage*, *branch coverage*, and *predicate coverage*. Similarly, the criteria studied to cover the *definition* and *use* aspects of variables in a program are all-defs, all-c-uses, all-p-uses, and all-uses, to name a few. The path predicates were analyzed to derive test data. While selecting paths and the corresponding test data, *no assumption is made regarding the actual type of faults that the selected test cases could potentially uncover*, that is, no specific types of faults are explicitly considered for detection.
* Domain testing takes an entirely new approach to fault detection. One defines a category of faults, called *domain errors*, and selects test data to detect those faults. If a program has domain errors, those will be revealed by the test cases.

## **18.3 Source of Domain**

Domains can be identified from both specifications and programs. We explain a method to identify domains from source code using the following steps:

* Draw a control flow graph from the given source code.
* Find all possible interpretations of the predicates. In other words, express the predicates solely in terms of the input vector and, possibly, a vector of constants. The reader may note that a predicate in a program may have multiple interpretations, because control may arrive at a predicate node via different paths.
* Analyze the interpreted predicates to identify domains.

## **18.4 Type of Domain**

A domain is a set of values for which the program performs identical computations. A domain can be represented by a set of predicates. Individual elements of the domain satisfy the predicates of the domain. A domain is defined, from a geometric perspective, by a set of constraints called *boundary inequalities*. Properties of a domain are discussed in terms of the properties of its *boundaries* as follows:

***Closed Boundary*:** A boundary is said to be closed if the points on the boundary are included in the domain of interest.

*P*2 : *x* ≤ −4

***Open Boundary*:** A boundary is said to be open if the points on the boundary do not belong to the domain of interest.

*P*1 : *x* +*y >*5

The above boundary is an open boundary of the domain. The reader may notice that it is the *equality symbol* ( = ) in a relational operator that determines whether or not a boundary is closed. If the relational operator in a boundary inequality has the equality symbol in it, then the boundary is a closed boundary; otherwise it is an open boundary.

***Closed Domain*:** A domain is said to be closed if all of its boundaries are closed.

***Open Domain*:** A domain is said to be open if some of its boundaries are open.

***Extreme Point*:** An extreme point is a point where two or more boundaries cross.

***Adjacent Domains*:** Two domains are said to be adjacent if they have a boundary inequality in common.

A program path will have a *domain error* if there is incorrect formulation of a path predicate. After an interpretation of an incorrect path predicate, the path predicate expression causes a boundary segment to

* be shifted from its correct position or
* have an incorrect relational operator.

A domain error can be caused by

* an incorrectly specified predicate or
* an incorrect assignment which affects a variable used in the predicate.

Now we discuss different types of domain errors:

***Closure Error*:** A closure error occurs if a boundary is open when the intention is to have a closed boundary, or vice versa. Some examples of closure error are:

* The relational operator ≤ is implemented as*<*.
* The relational operator*<*is implemented as≤.

***Shifted-Boundary Error*:** A shifted-boundary error occurs when the implemented boundary is parallel to the intended boundary. This happens when the *constant term* of the inequality defining the boundary takes up a value different from the intended value. In concrete terms, a shifted boundary error occurs due to a change in the *magnitude* or the *sign* of the constant term of the inequality.

## **18.5 ON OFF Points**

In domain testing a programmer targets domain errors where test cases are designed with the objective of revealing the domain errors. Therefore, it is essential that we consider an important characteristic of domain errors, stated as follows: *Data points on or near a boundary are most sensitive to domain errors*. In this observation, by *sensitive* we mean data points falling in the wrong domains. Therefore, the objective is to identify the data points that are most sensitive to domain errors so that errors can be detected by executing the program with those input values. In the following, we define two kinds of data points *near* domain boundaries, namely, ON point and OFF point:

***ON Point*:** Given a boundary, an ON point is a point *on* the boundary or “very close” to the boundary. This definition suggests that we can choose an ON point in two ways. Therefore, one must know when to choose an ON point in which way: If a point can be chosen to lie exactly on the boundary, then choose such a point as an ON point. If the boundary inequality leads to an *exact* solution, choose such an exact solution as an ON point. If a boundary inequality leads to an *approximate* solution, choose a point very close to the boundary.

***OFF Point*:** An OFF point of a boundary lies *away* from the boundary. However, while choosing an OFF point, we must consider whether a boundary is *open* or *closed* with respect to a domain: If the domain is *open* with respect to the boundary, then an OFF point of that boundary is an *interior* point inside the domain within a distance from the boundary. If the domain is *closed* with respect to the boundary, then an OFF point of that boundary is an *exterior* point outside the boundary within a distance.

# **LECTURE 19: System Integration Testing**

## **19.1 Basic Concepts of Integration Testing**

A software module, or component, is a self-contained element of a system. Modules have well-defined interfaces with other modules. A module can be a subroutine, function, procedure, class, or collection of those basic elements put together to deliver a higher-level service. A system is a collection of modules interconnected in a certain way to accomplish a tangible objective. A subsystem is an interim system that is not fully integrated with all the modules. It is also known as a subassembly.

In moderate to large projects, from tens to hundreds of programmers implement their share of the code in the form of modules. Modules are individually tested, which is commonly known as *unit testing*, by their respective programmers using white-box testing techniques. At the unit testing level, the system exists in pieces under the control of the programmers. The next major task is to put the modules, that is, pieces, together to construct the complete system. Constructing a working system from the pieces is not a straightforward task, because of numerous interface errors. Even constructing a reasonably stable system from the components involves much testing. The path from tested components to constructing a deliverable system contains two major testing phases, namely, integration testing and system testing. The primary objective of integration testing is to assemble a reasonably stable system in a laboratory environment such that the integrated system can withstand the rigor of a full-blown system testing in the actual environment of the system. The importance of integration testing stems from three reasons as outlined below.

* Different modules are generally created by groups of different developers. The developers may be working at different sites. In spite of our best effort in system design and documentation, misinterpretation, mistakes, and oversights do occur in reality. Interface errors between modules created by different programmers and even by the same programmers are rampant.
* Unit testing of individual modules is carried out in a controlled environment by using test drivers and stubs. Stubs are dummy modules which merely return predefined values. If a module under unit test invokes several other modules, the effectiveness of unit testing is constrained by the programmer’s ability to effectively test all the paths. Therefore, with the inherent limitations of unit testing, it is difficult to predict the behavior of a module in its actual environment after the unit testing is performed.
* Some modules are more error prone than other modules, because of their inherent complexity. It is essential to identify the ones causing most failures.

The objective of system integration is to build a “working” version of the system by (i) putting the modules together in an incremental manner and (ii) ensuring that the additional modules work as expected without disturbing the functionalities of the modules already put together. In other words, system integration testing is a systematic technique for assembling a software system while conducting tests to uncover errors associated with interfacing. We ensure that unit-tested modules operate correctly when they are combined together as dictated by the design. Integration testing usually proceeds from small subassemblies containing a few modules to larger ones containing more and more modules. Large, complex software products can go through several iterations of build-and-test cycles before they are fully integrated.

Integration testing is said to be complete when the system is fully integrated together, all the test cases have been executed, all the severe and moderate defects found have been fixed, and the system is retested.

## **19.2 Different Types of Interface**

Modularization is an important principle in software design, and modules are interfaced with other modules to realize the system’s functional requirements. An interface between two modules allows one module to access the service provided by the other. It implements a mechanism for passing control and data between modules. Three common paradigms for interfacing modules are as follows:

**Procedure Call Interface:** A procedure in one module calls a procedure in another module. The caller passes on control to the called module. The caller can pass data to the called procedure, and the called procedure can pass data to the caller while returning control back to the caller.

**Shared Memory Interface:** A block of memory is shared between two modules. The memory block may be allocated by one of the two modules or a third module. Data are written into the memory block by one module and are read from the block by the other.

**Message Passing Interface:** One module prepares a message by initializing the fields of a data structure and sending the message to another module. This form of module interaction is common in client–server-based systems and web-based systems.

Programmers test modules to their satisfaction. The question is: If all the unit-tested modules work individually, why can these modules not work when put together? The problem arises when we “put them together” because of rampant interface errors. Interface errors are those that are associated with structures existing outside the local environment of a module but which the module uses. The interface errors accounted for up to a quarter of all errors in the systems they examined. They found that of all errors that required a fix within one module, more than half were caused by interface errors. Perry and Evangelist have categorized interface errors as follows:

***Construction*:** Some programming languages, such as C, generally separate the interface specification from the implementation code. In a C program, programmers can write a statement #include header.h, where header.h contains an interface specification. Since the interface specification lies somewhere away from the actual code, programmers overlook the interface specification while writing code. Therefore, inappropriate use of #include statements cause construction errors.

***Inadequate Functionality*:** These are errors caused by implicit assumptions in one part of a system that another part of the system would perform a function. However, in reality, the “other part” does not provide the expected functionality—intentionally or unintentionally by the programmer who coded the other part.

***Location of Functionality*:** Disagreement on or misunderstanding about the location of a functional capability within the software leads to this sort of error. The problem arises due to the design methodology, since these disputes should not occur at the code level. It is also possible that inexperienced personnel contribute to the problem.

***Changes in Functionality*:** Changing one module without correctly adjusting for that change in other related modules affects the functionality of the program.

***Added Functionality*:** A completely new functional module, or capability, was added as a system modification. Any added functionality after the module is checked in to the version control system without a CR is considered to be an error.

***Misuse of Interface*:** One module makes an error in using the interface of a called module. This is likely to occur in a procedure–call interface. Interface misuse can take the form of wrong parameter type, wrong parameter order, or wrong number of parameters passed.

***Misunderstanding of Interface*:** A calling module may misunderstand the interface specification of a called module. The called module may assume that some parameters passed to it satisfy a certain condition, whereas the caller does not ensure that the condition holds. For example, assume that a called module is expected to return the index of an element in an array of integers. The called module may choose to implement binary search with an assumption that the calling module gives it a sorted array. If the caller fails to sort the array before invoking the second module, we will have an instance of interface misunderstanding.

***Data Structure Alteration*:** These are similar in nature to the functionality problems discussed above, but they are likely to occur at the detailed design level. The problem arises when the size of a data structure is inadequate, or it fails to contain a sufficient number of information fields. The problem has its genesis in the failure of the high-level design to fully specify the capability requirements of the data structure. Let us consider an example in which a module reads the data and keeps it in a record structure. Each record holds the person name followed by their employee number and salary. Now, if the data structure is defined for 1000 records, then as the number of records grows beyond 1000, the program is bound to fail. In addition, if management decides to award bonuses to a few outstanding employees, there may not be any storage space allocated for additional information.

***Inadequate Error Processing*:** A called module may return an error code to the calling module. However, the calling module may fail to handle the error properly.

***Additions to Error Processing*:** These errors are caused by changes to other modules which dictated changes in a module error handling. In this case either necessary functionality is missing from the current error processing that would help trace errors or current techniques of error processing require modification.

***Inadequate Postprocessing*:** These errors are caused by a general failure to release resources no longer required, for example, failure to deallocate memory.

***Inadequate Interface Support*:** The actual functionality supplied was inadequate to support the specified capabilities of the interface. For example, a module passes a temperature value in Celsius to a module which interprets the value in Fahrenheit.

***Initialization/Value Errors*:** A failure to initialize, or assign, the appropriate value to a variable data structure leads to this kind of error. Problems of this kind are usually caused by simple oversight. For example, the value of a pointer can change; it might point to the first character in a string, then to the second character, after that to the third character, and so on. If the programmer forgets to reinitialize the pointer before using that function once again, the pointer may eventually point to code.

## **19.3 Granularity of System Integration Testing**

System integration testing is performed at different levels of granularity. Integration testing includes both white- and black-box testing approaches. *Black-box* testing ignores the internal mechanisms of a system and focuses solely on the outputs generated in response to selected inputs and execution conditions. The code is considered to be a big black box by the tester who cannot examine the internal details of the system. The tester knows the input to the black box and observes the expected outcome of the execution. *White-box* testing uses information about the structure of the system to test its correctness. It takes into account the internal mechanisms of the system and the modules. In the following, we explain the ideas of *intra-system* testing, *intersystem* testing, and *pairwise* testing.

***Intra-system Testing:***This form of testing constitutes low-level integration testing with the objective of combining the modules together to build a cohesive system. The process of combining modules can progress in an incremental manner akin to constructing and testing successive builds. For example, in a client–server-based system both the client and the server are distinct entities running at different locations. Before the interactions of clients with a server are tested, it is essential to individually construct the client and the server systems from their respective sets of modules in an incremental fashion. The low-level design document, which details the specification of the modules within the architecture, is the source of test cases.

***Intersystem Testing:***Intersystem testing is a high-level testing phase which requires interfacing independently tested systems. In this phase, all the systems are connected together, and testing is conducted from end to end. The term *end to end* is used in communication protocol systems, and end-to-end testing means initiating a test between two access terminals interconnected by a network. The purpose in this case is to ensure that the interaction between the systems work together, but not to conduct a comprehensive test. Only one feature is tested at a time and on a limited basis. Later, at the time of system testing, a comprehensive test is conducted based on the requirements, and this includes functional, interoperability, stress, performance, and so on. Integrating a client–server system, after integrating the client module and the server module separately, is an example of intersystem testing. Integrating a call control system and a billing system in a telephone network is another example of intersystem testing. The test cases are derived from the high-level design document, which details the overall system architecture.

***Pairwise Testing:***There can be many intermediate levels of system integration testing between the above two extreme levels, namely intrasystem testing and intersystem testing. Pairwise testing is a kind of intermediate level of integration testing. In pairwise integration, only two interconnected systems in an overall system are tested at a time. The purpose of pairwise testing is to ensure that two systems under consideration can function together, assuming that the other systems within the overall environment behave as expected.

## **19.4 System Integration Technique**

One of the objectives of integration testing is to combine the software modules into a working system so that system-level tests can be performed on the complete system. Integration testing need not wait until all the modules of a system are coded and unit tested. Instead, it can begin as soon as the relevant modules are available. A module is said to be available for combining with other modules when the module’s *check-in request form*, to be discussed in this section, is ready. Some common approaches to performing system integration are as follows:

**Incremental Approach:** In this approach, integration testing is conducted in an incremental manner as a series of test cycles. In each test cycle, a few more modules are integrated with an existing and tested build to generate a larger build. The idea is to complete one cycle of testing, let the developers fix all the errors found, and continue the next cycle of testing. The complete system is built incrementally, cycle by cycle, until the whole system is operational and ready for system-level testing. The system is built as a succession of layers, beginning with some core modules. In each cycle, a new layer is added to the core and tested to form a new core. The new core is intended to be self-contained and stable. Here, “self-contained” means containing all the necessary code to support a set of functions, and “stable” means that the subsystem (i.e., the new, partial system) can stay up for 24 hours without any anomalies.

**Top Down Approach:** systems with hierarchical structures easily lend themselves to top-down and bottom-up approaches to integration. In a hierarchical system, there is a first, top-level module which is decomposed into a few second-level modules. Some of the second-level modules may be further decomposed into third-level modules, and so on. Some or all the modules at any level may be terminal modules, where a terminal module is one that is no more decomposed. An internal module, also known as a nonterminal module, performs some computations, invokes its subordinate modules, and returns control and results to its caller. In top-down and bottom-up approaches, a design document giving the module hierarchy is used as a reference for integrating modules.

**Bottom Up Approach:** In the bottom-up approach, system integration begins with the integration of lowest level modules. A module is said to be at the lowest level if it does not invoke another module. It is assumed that all the modules have been individually tested before. To integrate a set of lower level modules in this approach, we need to construct a test driver module that invokes the modules to be integrated. Once the integration of a desired group of lower level modules is found to be satisfactory, the driver is replaced with the actual module and one more test driver is used to integrate more modules with the set of modules already integrated. The process of bottom-up integration continues until all the modules have been integrated.

**Sandwich and Big Bang Approach:** In the sandwich approach, a system is integrated by using a mix of the top-down and bottom-up approaches. A hierarchical system is viewed as consisting of three layers. The bottom layer contains all the modules that are often invoked. The bottom-up approach is applied to integrate the modules in the bottom layer. The top layer contains modules implementing major design decisions. These modules are integrated by using the top-down approach. The rest of the modules are put in the middle layer. We have the advantages of the top-down approach where writing stubs for the low-level module is not required. As a special case, the middle layer may not exist, in which case a module falls either in the top layer or in the bottom layer. On the other hand, if the middle layer exists, then this layer can be integrated by using the big-bang approach after the top and the bottom layers have been integrated.

In the big-bang approach, first all the modules are individually tested. Next, all those modules are put together to construct the entire system which is tested as a whole. Sometimes developers use the big-bang approach to integrate small systems. However, for large systems, this approach is not recommended for the following reasons: In a system with a large number of modules, there may be many interface defects. It is difficult to determine whether or not the cause of a failure is due to interface errors in a large and complex system. In large systems, the presence of a large number of interface errors is not an unlikely scenario in software development. Thus, it is not cost effective to be optimistic by putting the modules together and hoping it will work.

## **19.5 Test Plan for System Integration**

System integration requires a controlled execution environment, much communication between the developers and the test engineers, judicious decision making along the way, and much time, on the order of months, in addition to the fundamental tasks of test design and test execution. Integrating a large system is a challenging task, which is handled with much planning in the form of developing a SIT plan.

An important decision to be made for integration testing is establishing the start and stop dates of each phase of integration testing. The start date and stop date for a phase are specified in terms of *entry criteria* and *exit criteria*, respectively. These criteria are described in the third section of the plan. Test configuration and integration techniques (e.g., top down or bottom up) to be used in each of these phases are described in this section. The test specification section describes the test procedure to be followed in each integration phase. The detailed test cases, including the input and expected outcome for each case, are documented in the test specification section. The history of actual test results, problems, or peculiarities is recorded in the fifth section of the SIT plan. Finally, references and an appendix, if any, are included in the test plan. System integration testing is performed in phases of increasing complexity for better efficiency and effectiveness. In the first phase interface integrity and functional validity within a system are tested. In the second phase end-to-end and pairwise tests are conducted. Finally, in the third phase, stress and endurance tests are performed. Each of the system integration phases identified in the SIT plan delineates a broad functionality category within the software structure, and it can be related to a specific domain of the system software structure.

## **19.6 OFF-THE-SHELF Component Integration**

Instead of developing a software component from scratch, organizations occasionally purchase off-the-shelf (OTS) components form third-party vendors and integrate them with their own components. In this process, organizations create less expensive software systems. A major issue that can arise while integrating different components is mismatches among code pieces developed by different parties usually unaware of each other.

Buyer organizations perform two types of testing on an OTS component before purchasing: (i) acceptance testing of the OTS component based on the criteria discussed in Chapter 14 and (ii) integration of the component with other components developed in-house or purchased from a third party. The most common cause of problems in the integration phase is inadequate acceptance testing of the OTS component. A lack of clear documentation of the system interface and less cooperation from the vendor may create an ordeal in integration testing, debugging, and fixing defects. Acceptance testing of an OTS component requires the development and execution of an acceptance test plan based on the acceptance criteria for the candidate component. All the issues are resolved before the system integration process begins. During integration testing, additional software components, such as a glue or a wrapper, can be developed to bind an OTS component with other components for proper functioning. These new software components are also tested during the integration phase.

# **LECTURE 20: System Testing**

## **20.1 Basic Concepts of System Testing**

Preparing for and executing system-level tests are a critical phase in a software development process because of the following: (i) There is pressure to meet a tight schedule close to the delivery date; (ii) there is a need to discover most of the defects before delivering the product; and (iii) it is essential to verify that defect fixes are working and have not resulted in new defects. It is important to monitor the processes of test execution and defect fixing. To be able to monitor those test processes, we identify two key categories of metrics: (i) *system test execution status* and (ii) *defects status*. We provide a detailed *defect schema*, which includes a general FSM model of defects for ease of collecting those metrics. Analyzing defects is a crucial activity performed by the software development team while fixing defects. Therefore, in this chapter, we describe three types of defect analysis techniques: *causal, orthogonal*, and *Pareto methodology*. In addition, we provide three metrics, namely, *defect removal efficiency, spoilage*, and *fault seeding*, to measure test effectiveness. The objective of a test effectiveness metric is to evaluate the effectiveness of a system testing effort in the development of a product in terms of the number of defects escaped from the system testing effort.

The product is ready for beta testing at the customer site during the system testing. We provide a framework for beta testing and discuss how beta testing is conducted at the customer’s site. Moreover, we provide a detailed structure of the system test execution report, which is generated before a product’s *general availability* is declared.

## **20.2 Modelling Defects**

The key to a successful defect tracking system lies in properly modeling defects to capture the viewpoints of their many stakeholders, called cross-functionality groups. The cross-functionality groups in an organization are those groups that have different stakes in a product. For example, a marketing group, a customer support group, a development group, a system test group, and a product sustaining group are collectively referred to as cross-functionality groups in an organization. It is not enough to merely report a defect from the viewpoint of software development and product management and seek to understand it by means of reproduction before fixing it. In reality, a reported defect is an evolving entity that can be appropriately represented by giving it a life-cycle model in the form of a state transition diagram.

The state transition model allows us to represent each phase in the life cycle of a defect by a distinct state. The model represents the life cycle of a defect from its initial reporting to its final closing through the following states: new, assigned, open, resolved, wait, FAD, hold, duplicate, shelved, irreproducible, postponed, and closed. When a defect moves to a new state, certain actions are taken by the owner of the state. By “owner” of a state of a defect we mean the person or group of people who are responsible for taking the required actions in that state. Once the associated actions are taken in a state, the defect is moved to a new state. Two key concepts involved in modeling defects are the levels of *priority* and *severity*. On one hand, a priority level is a measure of how soon the defect needs to be fixed, that is, urgency. On the other hand, a severity level is a measure of the extent of the detrimental effect of the defect on the operation of the product. Therefore, priority and severity assignments are separately done.

## **20.3 Metrics for Tracking System Test**

The system test execution brings forth (three) different facets of software development. The developers would like to know the degree to which the system meets the explicit as well as implicit requirements. The delivery date cannot be precisely predicted due to the uncertainty in fixing the problems. The customer is excited to take the delivery of the product. It is therefore a highly visible and exciting activity. At this stage, it is desirable to monitor certain metrics which truly represent the progress of system testing and reveal the quality level of the system. Based on those metrics, the management can trigger actions for corrective and preventive measures. By putting a small but critical set of metrics in place executive management is in a position to know whether they are on the right track. We make a difference between *statistics* and *in-process* metrics: Statistics are used for post project analysis to gather experience and use it in future projects, whereas in-process metrics let us monitor the progress of the project while we still have an opportunity to steer the course of the project. By considering three large, real-life test projects, we categorize execution metrics into two classes:

* For monitoring test execution
* For monitoring defects

The first class of metrics concerns the process of executing test cases, whereas the second class concerns the defects found as a result of test execution. These metrics need to be tracked and analyzed on a periodic basis, say, daily or weekly. It is important to gather valid and accurate information about the project for the leader of a system test group to effectively control a test project. One example of effective control of a test project is to precisely determine the time to trigger the revert criteria for a test cycle and initiate root cause analysis of the problems before more tests are performed. A test manager can effectively utilize the time of test engineers by triggering such a revert criterion, and possibly money, by suspending a test cycle on a product with too many defects to carry out a meaningful system test. Therefore, our objective is to identify and monitor the metrics while system testing is in progress so that the nimble decisions can be taken by the management team.

## **20.4 Metrics for Monitoring Test Execution**

It is desirable to monitor the execution of system test cases of a large project involving tens of test engineers and taking several months. The system test execution for large projects is monitored weekly in the beginning and daily toward the finish. It is strongly recommended to use automated tools, such as a test factory, for monitoring and reporting the test case execution status. Different kinds of queries can be generated to obtain the *in-process* metrics after a database is in place. The following metrics are useful in successfully tracking test projects:

* **Test Case Escapes (TCE):** The test engineers may find the need to design new test cases during the testing, called test case escapes, as testing continues. The number of test case escapes is monitored as it rises. A significant increase in the number of test case escapes implies the deficiencies in the test design process, and this may adversely affect the project schedule.
* **Planned versus Actual Execution (PAE) Rate:** The actual number of test cases executed every week is compared with the planned number of test cases. This metric is useful in representing the productivity of the test team in executing test cases. If the actual rate of execution falls far short of the planned rate, managers may have to take preventive measures so that the time required for system testing does not adversely affect the project schedule.
* **Execution Status of Test (EST) Cases:** It is useful to periodically monitor the number of test cases lying in different states—failed, passed, blocked, invalid, and untested—after their execution. It is also useful to further subdivide those numbers by test categories, such as basic, functionality, and robustness.

## **20.5 Defect Analysis Techniques**

Orthogonal defect classification (ODC) is a methodology for rapid capturing of the semantics of each software defect. The ODC methodology provides both a classification scheme for software defects and a set of concepts that provide guidance in the analysis of the classified aggregate defect data. Here, orthogonality refers to the nonredundant nature of the information captured by the defect attributes and their values that are used to classify defects. The classification of defects occurs at two different points in time during the life cycle of a defect. First, a defect is put in its initial new state when it is discovered, where the circumstances leading up to the exposure of the defect and the likely impact are typically known. Second, a defect is moved to the resolved state, where the exact nature of the defect and the scope of the fix are known. ODC categories capture the semantics of a defect from these two perspectives.

## **20.6 Defect Casual Analysis (DCA)**

The idea of defect causal analysis (DCA) in software development is effectively used to raise the quality level of products at a lower cost. Causal analysis can be traced back to the quality control literature as one of the quality circle activities in the manufacturing sector. The causes of manufacturing defects are analyzed by using the idea of a quality circle, which uses *cause–effect* diagrams and Pareto diagrams. A cause–effect diagram is also called an *Ishikawa* diagram or a *fishbone* diagram. A case study of an organization called “HPA Appliance” involving the use of causal analysis to prevent defects on manufacturing lines. Causal analysis of software defects is practiced in a number of Japanese companies, usually in the context of quality circle activities

# **LECTURE 21: System Testing**

## **21.1 Basic Concepts of Beta Testing**

Beta testing is conducted by the potential buyers prior to the official release of the product. The purpose of beta testing is not to find defects but to obtain feedback from the field about the usability of the product. There are three kinds of beta tests performed based on the relationships between the potential buyers and the sellers:

**Marketing Beta:** The purpose here is to build early awareness and interest in the product among potential buyers.

**Technical Beta:** The goal here is to obtain feedback about the usability of the product in a real environment with different configurations from a small number of friendly customers. The idea is to obtain feedback from a limited number of customers who commit a considerable amount of time and thought to their evaluation.

**Acceptance Beta:** The idea of this test is to ensure that the product meets its acceptance criteria. It is the fulfillment of the contract between a buyer and a seller. The system is released to a specific buyer, who has a contractual original equipment manufacturer (OEM) agreement with the supplier. Acceptance beta includes the objective of technical beta as well.

## **21.2 First Customer Shipment**

The exit criterion of the final test cycle must be satisfied before the FCS which is established in the test execution strategy section of Chapter 13. An FCS readiness review meeting is called to ensure that the product meets the shipment criteria. The shipment criteria are more than just the exit criteria of the final test cycle. This review should include representatives from the key function groups responsible for delivery and support of the product, such as engineering, operation, quality, customer support, and product marketing. A set of generic FCS readiness criteria is as follows:

* All the test cases from the test suite should have been executed. If any of the test cases is unexecuted, then an explanation for not executing the test case should have been provided in the test factory database.
* Test case results are updated in the test factory database with passed, failed, blocked, or invalid status. Usually, this is done during the system test cycles.
* The requirement database is updated by moving each requirement from the verification state to either the closed or the decline state, so that compliance statistics can be generated from the database. All the issues related to the EC must be resolved with the development group.
* The pass rate of test cases is very high, say, 98%.
* No crash in the past two weeks of testing has been observed.
* No known defect with *critical* or *high* severity exists in the product.
* Not more than a certain number of known defects with *medium* and *low* levels of severity exist in the product. The threshold number may be determined by the software project team members.
* All the identified FCS blocker defects are in the closed state.
* All the resolved defects must be in the closed state.
* All the outstanding defects that are still in either the open or assigned state are included in the release note along with the workaround, if there is one.
* The user guides are in place.
* A troubleshooting guide is available.
* The test report is completed and approved. Details of a test report are explained in the following section.

## **21.3 Product Sustaining**

Once a product is shipped to one of the paying customers and if there is no major issue reported by the customer within a time frame of, say, three weeks, then the product’s general availability (GA) is declared. After that the software project is moved to a new phase called the sustaining phase. The goal of this phase is to maintain the software quality throughout the product’s market life. Software maintenance activities occur because software testing cannot uncover all the defects in a large software system. The following three software maintenance activities were coined by Swanson:

* **Corrective:** The process that includes isolation and correction of one or more defects in the software.
* **Adaptive:** The process that modifies the software to properly interface with a changing environment such as a new version of hardware or third-party software.
* **Perfective:** The process that improves the software by the addition of new functionalities, enhancements, and/or modifications to the existing functions.

The first major task is to determine the type of maintenance task to be conducted when a defect report comes in from a user through the customer support group. The sustaining team, which includes developers and testers, is assigned immediately to work on the defect if the defect reported by the customer is considered as *corrective* in nature. The status of the progress is updated to the customer within 24 hours. The group continues to work until a patch with the fix is released to the customer. If the defect reported is considered as either *adaptive* or *perfective* in nature, then it is entered in the requirement database, and it goes though the usual software development phases.

## **21.4 Measuring Test Effectiveness**

It is useful to evaluate the effectiveness of the testing effort in the development of a product. After a product is deployed in the customer environment, a common measure of test effectives is the number of defects found by the customers that were not found by the test engineers prior to the release of the product. These defects had escaped from our testing effort. A metric commonly used in the industry to measure test effectiveness is the defect removal efficiency (DRE), defined as



***Spoilage Metric*** Defects are injected and removed at different phases of a software development cycle. Defects get introduced during requirements analysis, high-level design, detailed design, and coding phases, whereas these defects are removed during unit testing, integration testing, system testing, and acceptance testing phases. The cost of each defect injected in phase *X* and removed in phase *Y* is not uniformly distributed, instead the cost increases with the increase in the distance between *X* and *Y*. The delay in finding the dormant defects cause greater harms, and it costs more to fix because the dormant defects may trigger the injection of other related defects, which need to be fixed in addition to the original dormant defects. Therefore, an effective testing method would find defects earlier than a less effective testing method would. Hence a useful measure of test effectiveness is defect age, known as *PhAge*. One can modify the table to accommodate different phases of the software development life cycle followed within an organization, including the PhAge numbers.



# **LECTURE 22: System Testing**

## **22.1 Different Types of System Testing**

**Boot Tests:** Boot tests are designed to verify that the system can boot up its software image (or build) from the supported boot options. The boot options include booting from ROM, FLASH card, and PCMCIA (Personal Computer Memory Card International Association) card. The minimum and maximum configurations of the system must be tried while booting. For example, the minimum configuration of a router consists of one-line card in its slots, whereas the maximum configuration of a router means that all slots contains line cards.

**Robustness Test:** Robustness means how sensitive a system is to erroneous input and changes in its operational environment. Tests in this category are designed to verify how gracefully the system behaves in error situations and in a changed operational environment. The purpose is to deliberately break the system, not as an end in itself, but as a means to find error. It is difficult to test for every combination of different operational states of the system or undesirable behavior of the environment.

**Interoperability Test:** In this category, tests are designed to verify the ability of the system to interoperate with third-party products. An interoperability test typically combines different network elements in one test environment to ensure that they work together. In other words, tests are designed to ensure that the software can be connected with other systems and operated. In many cases, during interoperability tests, users may require the hardware devices to be interchangeable, removable, or reconfigurable. Often, a system will have a set of commands or menus that allow users to make the configuration changes. The reconfiguration activities during interoperability tests are known as *configuration testing*. Another kind of interoperability test is called a (*backward*) *compatibility test* . Compatibility tests verify that the system works the same way across different platforms, operating systems, and database management systems. Backward compatibility tests verify that the current software build flawlessly works with older version of platforms. As an example, let us consider a 1xEV-DO radio access network as shown in Figure 8.5. In this scenario, tests are designed to ensure the interoperability of the RNCs with the following products from different vendors: (i) PDSN, (ii) PDA with 1xEV-DO card, (iii) AAA server, (iv) PC with 1xEV-DO card, (v) laptop with 1xEV-DO card, (vi) routers from different vendors, (vii) BTS or RNC, and (viii) switches.

**Performance Test:** Performance tests are designed to determine the performance of the actual system compared to the expected one. The performance metrics needed to be measured vary from application to application. An example of expected performance is: The response time should be less than 1 millisecond 90% of the time in an application of the “push-to-talk” type. Another example of expected performance is: A transaction in an on-line system requires a response of less than 1 second 90% of the time. One of the goals of router performance testing is to determine the system resource utilization, for maximum aggregation throughput rate considering zero drop packets. In this category, tests are designed to verify response time, execution time, throughput, resource utilization, and traffic rate.

Scalability Test: All man-made artifacts have engineering limits. For example, a car can move at a certain maximum speed in the best of road conditions, a telephone switch can handle a certain maximum number of calls at any given moment, a router has a certain maximum number of interfaces, and so on. In this group, tests are designed to verify that the system can scale up to its engineering limits. A system may work in a limited-use scenario but may not scale up. The run time of a system may grow exponentially with demand and may eventually fail after a certain limit. The idea is to test the limit of the system, that is, the magnitude of demand that can be placed on the system while continuing to meet latency and throughput requirements. A system which works acceptably at one level of demand may not scale up to another level. Scaling tests are conducted to ensure that the system response time remains the same or increases by a small amount as the number of users are increased. Systems may scale until they reach one or more engineering limits.

**Stress Test:** The goal of stress testing is to evaluate and determine the behavior of a software component while the offered load is in excess of its designed capacity. The system is deliberately stressed by pushing it to and beyond its specified limits. Stress tests include deliberate contention for scarce resources and testing for incompatibilities. It ensures that the system can perform acceptably under worst-case conditions under an expected peak load. If the limit is exceeded and the system does fail, then the recovery mechanism should be invoked.

**Load Test:** Load and stability tests are designed to ensure that the system remains stable for a long period of time under full load. A system might function flawlessly when tested by a few careful testers who exercise it in the intended manner. However, when a large number of users are introduced with incompatible systems and applications that run for months without restarting, a number of problems are likely to occur: (i) the system slows down, (ii) the system encounters functionality problems, (iii) the system silently fails over, and (iv) the system crashes altogether. Load and stability testing typically involve exercising the system with virtual users and measuring the performance to verify whether the system can support the anticipated load. This kind of testing helps one to understand the ways the system will fare in real-life situations. With such an understanding, one can anticipate and even prevent load-related problems. Often, operational profiles are used to guide load and stability testing.

**Regulatory Test:** In this category, the final system is shipped to the regulatory bodies in those countries where the product is expected to be marketed. The idea is to obtain compliance marks on the product from those bodies. The regulatory approval bodies of various countries have been shown in Table 8.2. Most of these regulatory bodies issue safety and EMC (electromagnetic compatibility)/EMI (electromagnetic interference) compliance certificates (emission and immunity). The regulatory agencies are interested in identifying flaws in software that have potential safety consequences. The safety requirements are primarily based on their own published standards. For example, the CSA (Canadian Standards Association) mark is one of the most recognized, accepted, and trusted symbols in the world. The CSA mark on a product means

that the CSA has tested a representative sample of the product and determined that the product meets the CSA’s requirements. Safety-conscious and concerned consumers look for the CSA mark on products they buy. Similarly, the CE mark on a product indicates conformity to the European Union directive with respect to safety, health, environment, and consumer protection. In order for a product to be sold in the United States, the product needs to pass certain regulatory requirements of the Federal Communications Commission (FCC).

# **LECTURE 23: Acceptance Testing**

## **23.1 Basic Concepts of Acceptance Testing**

A product is ready to be delivered to the customer after the system test group is satisfied with the product by performing system-level tests. Customers execute acceptance tests based on their expectations from the product. The services offered by a software product may be used by millions of users. For example, the service provider of a cellular phone network is a customer of the software systems running the phone network, whereas the general public forms the user base by subscribing to the phone services. It is not uncommon for someone to have a dual role as a customer and a user. The service provider needs to ensure that the product meets certain criteria before the provider makes the services available to the general public. Acceptance testing is a formal testing conducted to determine whether a system satisfies its acceptance criteria—the criteria the system must satisfy to be accepted by the customer. It helps the customer to determine whether or not to accept the system. The customer generally reserves the right to refuse to take delivery of the product if the acceptance test cases do not pass. There are two categories of acceptance testing:

* User acceptance testing.
* Business acceptance testing.

The UAT is conducted by the customer to ensure that system satisfies the contractual acceptance criteria before being signed off as meeting user needs. Actual planning and execution of the acceptance tests do not have to be undertaken directly by the customer. Often third-party consulting firms offer their services to do this task. However, the customer must specify the acceptance criteria for the third party to seek in the product. The BAT is undertaken within the development organization of the supplier to ensure that the system will eventually pass the UAT. It is a rehearsal of UAT at the premises of the supplier. The development organization of the supplier derives and executes test cases from the client’s contract, which include the acceptance criteria.

## **23.2 Acceptance Criteria**

At the core of any contractual agreement is a set of acceptance criteria. A key question is what criteria must the system meet in order to be acceptable? The acceptance criteria must be measurable and, preferably, quantifiable. The basic principle of designing the acceptance criteria is to ensure that the quality of the system is acceptable. One must understand the meaning of the quality of a system, which is a complex concept. It means different things to different people, and it is highly context dependent. Even though different persons may have a different view about quality, it is the customer’s opinion that prevails. The concept of quality is, in fact, complex and multifaceted. Five views of quality, namely, *transcendental view*, *user view*, *manufacturing view*, *product view*, and *value-based view*. The five views were originally presented in the context of production and manufacturing in general and subsequently in a software development context. The five views are presented below in a concise form:

1. The transcendental view sees quality as something that can be recognized but is difficult to describe or define.
2. The user view sees quality as satisfying the purpose.
3. The manufacturing view sees quality as conforming to the specification.
4. The product view ties quality with the inherent characteristics of the product.
5. The value-based view puts a cost figure on quality—the amount a customer is willing to pay for it.

Acceptance criteria are defined on the basis of these multiple facets of quality attributes. These attributes determine the presence or absence of quality in a system. Buyers, or Customers, should think through the relevance and relative importance of these attributes in their unique situation at the time of formulating the acceptance criteria. The attributes of quality are discussed below and examples of acceptance criteria for each quality attribute are given.

***Functional Correctness and Completeness*** One can ask the question: Does the system do what we want it to do? All the features which are described in the requirements specification must be present in the delivered system. It is important to show that the system works correctly under at least two to three conditions for each feature as a part of acceptance.

***Accuracy*** The question is: Does the system provide correct results? Accuracy measures the extent to which a computed value stays close to the expected value. Accuracy is generally defined in terms of the magnitude of the error. A small gap—also called an error in numerical analysis, for example—between the actual value computed by a system and the expected value is generally tolerated in a continuous space. The accuracy problem is different in discrete space, leading to false-positive and false-negative results. False positives and false negatives are serious drawbacks in any diagnostic and monitoring software tools.

***Data Integrity*** Data integrity refers to the preservation of the data while it is transmitted or stored such that the value of data remains unchanged when the corresponding receives or retrieve operations are executed at a later time. Thus, data must not be compromised by performing update, restore, retrieve, transmit, and receive operations. The requirement of data integrity is included in the acceptance test criteria to uncover design flaws that may result in data corruption. In communication systems, an intruder can change the data without the sender and receiver detecting the change. If integrity check mechanisms are in place, the data may be changed, but the mechanism will detect the tampering. Data integrity mechanisms detect changes in a data set.

***Data Conversion*** Data conversion is the conversion of one form of computer data to another. For example, conversion of a file from one version of Microsoft Word to an earlier version for the sake of those who do not have the latest version of Word installed. Data conversion testing is testing of programs or procedures that are used to convert data from existing systems for use in replacement systems. Data may be converted into an invalid format that cannot be processed by the new system if this is not performed properly; thus, the data will have no value.

## **23.3 Selection of Acceptance Criteria**

The acceptance criteria discussed above provide a broad idea about customer needs and expectations, but those are too many and very general. The customer needs to select a subset of the quality attributes and prioritize them to suit their specific situation. Next, the customer identifies the acceptance criteria for each of the selected quality attributes. When the customer and the software vendor reach an agreement on the acceptance criteria, both parties must keep in mind that satisfaction of the acceptance criteria is a trade-off between time, cost, and quality. As Ed Yourdon opined, sometimes less than perfect is good enough. Only business goals and priority can determine the degree of “less than perfect” that is acceptable to both the parties. Ultimately, the acceptance criteria must be related to the business goals of the customer’s organization.

Many organizations associated with different application domains have selected and customized existing quality attributes to define quality for themselves, taking into consideration their specific business and market situation. For example, IBM used the quality attribute list CUPRIMDS—capability, usability, performance, reliability, installation, maintenance, documentation, and service—for its products. Similarly, for web-based applications, a set of quality attributes are identified in decreasing order of priority: reliability, usability, security, availability, scalability, maintainability, and time to market. Such a prioritization scheme is often used in specific application domains. For example, usability and maintainability take precedence over performance and reliability for a word processor software. On the other hand, it might be the other way around for a real-time operating system or telecommunication software.

## **23.4 Acceptance Test Plan**

Planning for acceptance testing begins as soon as the acceptance criteria are known. Early development of an acceptance test plan (ATP) gives us a good picture of the final product. The purpose of an ATP is to develop a detailed outline of the process to test the system prior to making a transition to the actual business use of the system. Often, the ATP is delivered by the vendor as a contractual agreement, so that the business acceptance testing can be undertaken within the vendor’s development organization to ensure that the system eventually passes the acceptance test.

In developing an ATP, emphasis is put on demonstrating that the system works according to the customer’s expectation, rather than passing a set of comprehensive tests. In any case, the system is expected to have already passed a set of comprehensive tests during system-level testing. The ATP must be kept very simple because the audience of this plan may include people from diverse backgrounds, such as marketing and business managers. Some people argue that the ATP is redundant and unnecessary if a comprehensive system test plan is developed. We believe that even if a system test plan is adequate, acceptance tests usually uncover additional significant problems. Moreover, user’s concerns are not addressed during system-level testing.

An ATP needs to be written and executed by the customer’s special user group. The user group consists of people from different backgrounds, such as software quality assurance engineers, business associates, and customer support engineers. In addition, the acceptance test cases are executed at the user’s operational environment, whereas the system-level test cases are executed in a laboratory environment. An overall test plan for acceptance testing and description of specific tests are documented in the ATP.

The introduction section of the ATP describes the structure of the test plan and what we intend to accomplish with this test plan. This section typically includes (i) test project name, (ii) revision history, (iii) terminology and definitions, (iv) names of the approvers and the date of approval, (v) an overview of the plan, and (vi) references.

For each quality category from the acceptance criteria signed-off document two subsections are created: operational environment and test case specification. The operational environment deals with discussion on site preparation for the execution of the acceptance test cases. Test cases are specified for each acceptance criteria within the quality category.

## **23.5 Acceptance Test Execution**

The acceptance test cases are divided into two subgroups. The first subgroup consists of basic test cases, and the second consists of test cases that are more complex to execute. The acceptance tests are executed in two phases. In the first phase, the test cases from the basic test group are executed. If the test results are satisfactory, then the second phase, in which the complex test cases are executed, is taken up. In addition to the basic test cases, a subset of the system-level test cases is executed by the acceptance test engineers to independently confirm the test results. Obviously, a key question is: Which subset of the system-level test cases are selected? It is recommended to randomly select 5–10 test cases from each test category. If a very large fraction, say more than 0.95, of the basic test cases pass, then the second phase can proceed. It may be counterproductive to carry out the execution of the more complex tests if a significant fraction of the basic tests fails.

Acceptance test execution is an important activity performed by the customer with much support from the developers. The activity includes the following detailed actions:

* The developers train the customer on the usage of the system.
* The developers and the customer coordinate the fixing of any problem discovered during acceptance testing.
* The developers and the customer resolve the issues arising out of any acceptance criteria discrepancy.

System-level test personnel from the development organization travel to the customer location where the acceptance tests are to be conducted. They assist the customer in preparing a test site and train the acceptance test engineers on how to use the system. They provide the earlier system-level test results to the customer’s test engineers in order to make informal decisions about the direction and focus of the acceptance testing effort. In addition, the on-site system test engineers answer the customer’s questions about the system and assist the acceptance test engineers in executing the acceptance tests.

Any defect encountered during acceptance testing are reported to the software development organization through the on-site system test engineers. The defects are submitted through the defect tracking system. The software build is retested by the supplier and a satisfactory software image is made available to the customer for continuation of acceptance testing when the defects are fixed. The failed tests are repeated after the system is upgraded with a new software image. An agreement must be reached between the on-site system test engineers and the acceptance test engineers when to accept a new software image for acceptance testing. The number of times the system can be upgraded to a new software image during acceptance testing is negotiated between the customer and the supplier. Multiple failures of a system during acceptance testing are an indication of poor system quality.

It is possible that an acceptance test engineer may encounter issues related to acceptance criteria during the execution of acceptance test cases. The system may not provide services to the users as described in the acceptance criteria. Any deviation from the acceptance criteria discovered at this stage may not be fixed immediately. The acceptance test engineer may create an acceptance criteria change (ACC) document to communicate the deficiency in the acceptance criteria to the supplier. A representative format of an ACC document. An ACC report is generally given to the supplier’s marketing department through the on-site system test engineers.

## **23.6 Acceptance Test Report**

Acceptance test activities are designed to reach one of three conclusions: Accept the system as delivered, accept the system after the requested modifications have been made, or do not accept the system. Usually some useful intermediate decisions are made before making the final decision:

* A decision is made about the continuation of acceptance testing if the results of the first phase of acceptance testing are not promising. One may recall that the basic tests are executed in the first phase.
* If the test results are unsatisfactory, changes will be made to the system before acceptance testing can proceed to the next phase.

The intermediate decisions are made based on evaluation of the results of the first phase of testing. Moreover, during the execution of acceptance tests, the status of testing is reviewed at the end of every working day by the leader of the acceptance test team, on-site system test engineers, and project managers of the customer and the supplier. The acceptance team prepares a test report which forms the basis of discussion at the review meeting before they meet for a review.

# **LECTURE 24: Testing Tools and Review Class**

## **24.1 Acceptance Test Report**

There are different types of software testing tools. Many tools available – commercial and open source. No single classification of testing tools. For example:

* + Functional and Regression Testing Tools
  + Performance Testing Tools
  + Defect/Bug Tracking Tools
  + Test Management Tools

Functional and Regression Testing Tools are used for functional and regression testing. Many tools available – commercial & open source, for example:

* + Selenium
  + HP-UFT (Unified Functional Testing) former QTP (Quick Test Professional)
  + Ranorex Studio
  + Katalon Studio
  + TestComplete
  + IBM Rational Functional Tester
  + Sahi Pro

Performance Testing Tools are used to test performance of a system.Evaluate the compliance of a system with specified performance requirements.

* + - * **Speed** – It determines whether the application responds quickly.
      * **Scalability** – It determines maximum user load the software application can handle.
      * **Stability** – It determines if the application is stable under varying loads.

Many tools available – commercial and open source.

* + - * HP Performance Tester (former LoadRunner)
      * Rational Performance Tester
      * WebLOAD
      * LoadNinja
      * Silk Performer
      * NeoLoad
      * Apache Jmeter

**Test management tools** are used to store information on how testing is to be done, plan testing activities and report the status of quality assurance activities. The tools have different approaches to testing and thus have different sets of features. Generally, they are used to maintain and plan manual testing, run or gather execution data from automated tests, manage multiple environments and to enter information about found defects. Many test management tools incorporate requirements managements and defect tracking. Many tools available

* + Kualitee
  + JIRA
  + HP ALM (former Quality Center)
  + Zephyr
  + SpiraTest
  + qTest

# **TEST BOOKS AND REFERENCES**

* Software Testing And Quality Assurance – Theory and Practice - Kshirasagar Naik & Priyadarshi Tripathy
* Software Quality Engineering: Testing, Quality Assurance and Quantifiable Improvement - Jeff Tian
* R.S. Pressman & Associates, Inc. (2010). *Software Engineering: A Practitioner’s Approach.*
* Kelly, J. C., Sherif, J. S., & Hops, J. (1992). An analysis of defect densities found during software inspections. *Journal of Systems and Software*, *17*(2), 111-117.

# **QUESTION BANK**