Software Quality Assurance

Presented by:

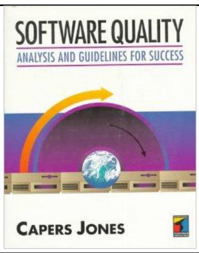
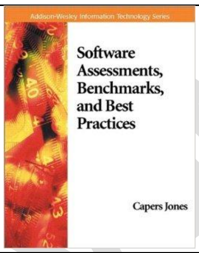
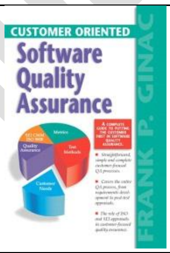
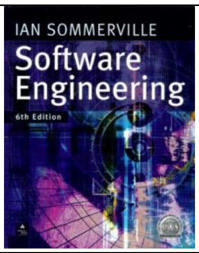
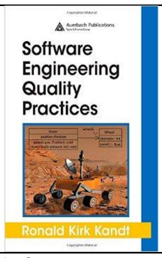
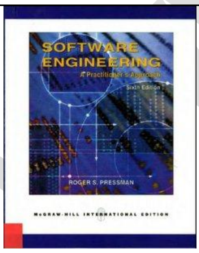
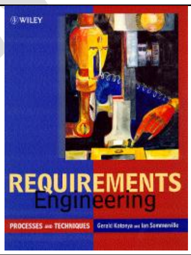
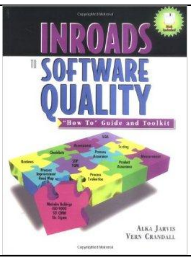
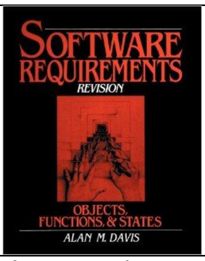
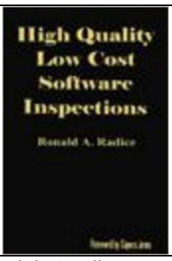
Dr. Ghulam Ahmad Farrukh

Virtual University of Pakistan

Published by:

Safi Khan

www.vumultan.com

				
Software Quality: Analysis and Guidelines for Success by Capers Jones	Software Assessments, Benchmarks, and Best Practices by Capers Jones	Customer-oriented Software Quality Assurance by Frank P. Ginac	Software Engineering by Sommerville	Software Engineering Quality Practices by Ronald K. Kandt
				
Software Engineering: A Practitioner's Approach by Roger S. Pressman	Requirements Engineering: Processes and Techniques by Kotonya and Sommerville	Inroads to Software Quality by Alka Jarvis and Vern Crandell	Software Requirements: Objects, States, and Functions by Alan M. Davis	High Quality Low Cost Software Inspections by Ronald A. Radice

Introduction to Software Quality Assurance

Introduction

- This course deals with a very important aspect of software engineering: quality assurance of software products and services
- We'll learn different aspects of software quality assurance in this course
- In the first few lectures, we will discuss what software quality is and how it impacts the development of the software development and maintenance and other basic concepts in SQA
- In the second phase of this course, we'll discuss in detail the activities in each phase of the software development lifecycle, as they relate to software quality assurance
- In the third part of this course, we'll discuss different topics related to software quality assurance. We'll look at quality assurance processes, some of the major process improvement programs from the quality assurance' perspective
- We'll also study some other topics, given our time constraints

What is Quality?

- Can you define quality?
- You must be thinking, what kind of question is that. It is very easy to define quality, but if you think really hard, it is not that easy to define quality
- Have you come with a definition? Let's see what I have in store for you

Synonyms of Quality

- Excellence
- Superiority
- Class
- Eminence
- Value
- Worth

Antonym of Quality

- Inferiority

Marketability of Quality

- Everyone claims to manufacture / develop / sell / market "good" quality products / services
- You will never come across a person or company selling products or services as low or poor quality products, even when they are

Software Quality

- Quality as it relates to all aspects of software (requirements / design / code / tests / documents / training)
- Difficult to define
    - Software quality is somewhat like the concept of beauty. Each of us has a strong opinion about what constitutes beauty, and we recognize it when we see it. But when asked to explain exactly why we regard an object as beautiful, it is hard to

put the factors into words
- Good software quality characteristics can be identified
- Bad or undesirable characteristics can also be identified

Software Quality Definitions
- Now we'll discuss six key factors, which are considered as definitions of software quality, and we'll use them throughout this course

Software Quality
- Low levels of defects when deployed, ideally approaching zero
- Low levels of defects when deployed, ideally approaching zero
- A majority of clients with high user-satisfaction when surveyed
- A structure that can minimize "bad fixes" or insertion of new defects during repairs
- Effective customer support when problems do occur
- Rapid repairs for defects, especially for high-severity defects

Beyond Absence of Defects
- Sense of beauty
- Sense of fitness for purpose
- Sense of elegance that goes beyond the simple absence of overt flaws
- Has well-formed requirements
- Robust

Why Software Quality?
- Reduces time to market for new products
- Enhances market share compared to direct competitors
- Minimizes "scrap and rework" expenses
- Attracts and keeps "top-gun" personnel
- Minimizes the risk of serious litigation
- Minimizes the risk of serious operating failures and delays
- Minimizes the risk of bankruptcy or business failures, which may be attributed directly to poor quality or poor software quality

Software Quality Assurance
- So the term software quality assurance would mean that the software guarantees high quality
- In this course, we'll learn the different processes, techniques, and activities, which enables us – the software professionals – to provide that guarantee to ourselves and our clients

Achieving Software Quality
- "For a software application to achieve high quality levels, it is necessary to begin upstream and ensure that intermediate deliverables and work products are also of high quality levels. This means that the entire process of software development must itself be

focused on quality"
  - Capers Jones

What is a Software Defect?

- A software defect is an error, flaw, mistake, failure, or fault in software that prevents it from behaving as intended (e.g., producing an incorrect or unexpected result)
- Software defects are also known as software errors or software bugs

Effects of Software Defects

- Bugs can have a wide variety of effects, with varying levels of inconvenience to the user of the software. Some bugs have only a subtle effect on the program's functionality, and may thus lie undetected for a long time. More serious bugs may cause the software to crash or freeze leading to a denial of service
- Others qualify as security bugs and might for example enable a malicious user to bypass access controls in order to obtain unauthorized privileges
- The results of bugs may be extremely serious
- In 1996, the European Space Agency's US $1 billion prototype Arian 5 rocket was destroyed less than a minute after launch, due a bug in the on-board guidance computer program
- In June 1994, a Royal Air Force Chinook crashed into the Mull of Kintyre, killing 29 people. An investigation uncovered sufficient evidence to convince that it may have been caused by a software bug in the aircraft's engine control computer
- In 2002, a study commissioned by the US Department of Commerce' National Institute of Standards and Technology concluded that software bugs are so prevalent and detrimental that they cost the US economy and estimated US $59 billion annually, or about 0.6 percent of the gross domestic product

Software Defects in Six Application Size Ranges

- 1 function point or 125 C statements
- 10 function points or 1,250 C statements
- 100 function points or 12,500 C statements
- 1,000 function points or 125,000 C statements
- 10,000 function points or 1,250,000 C statements
- 100,000 function points or 12,500,000 C statements

Categories of Software Defects

- Errors of commission
- Errors of omission
- Errors of clarity and ambiguity
- Errors of speed or capacity

Errors of Commission

- Something wrong is done
- A classic example at the code level would be going through a loop one time too many or branching on the wrong address

Errors of Omission

- Something left out by accident
- For example, omitting a parentheses in nested expressions

Errors of Clarity and Ambiguity

- Different interpretations of the same statement
- This kind of error is common with all natural language requirements and specification documents and user manuals, too.

Errors of Speed and Capacity

- Application works, but not fast enough
- Software defects can be found in any of the documents and work products including very serious ones in cost estimates and development plans
- However, there are seven major classes of software work products where defects have a strong probability of triggering some kind of request for warranty repair if they reach the field

Software Defect Origins

- Errors in Requirements
- Errors in Design
- Errors in Source code
- Errors in User Documentation
- Errors due to "Bad fixes"
- Errors in Data and Tables
- Errors in Test Cases
- We'll discuss all of them in detail, when we talk about different processes of software development life cycle

Defect Discovery

- Defects are discovered by developers & testers (usually) before release
- Defects are discovered by customers and users (usually) after release
- Defects discovered after release can be embarrassing for the development team

Defect Discovery by Customers

- Rule 1: Defect discovery is directly related to the number of users
- Rule 2: Defect discovery is inversely related to the number of defects

Software Defect Elimination Strategies

- Effective defect prevention
- High levels of defect removal efficiency
- Accurate defect prediction before the project begins
- Accurate defect tracking during development
- Useful quality measurements
- Ensuring high levels of user-satisfaction

Defect Prevention and Removal

- Both defect prevention and removal techniques are used by the “best-in-the-class” companies
- Defect prevention is very difficult to understand, study, and quantify. We’ll talk about defect prevent in a later lecture
- Both non-test and testing defect removal techniques must be applied

Typical Defect Removal

- Inspections
    - Direct fault detection and removal
- Testing
    - Failure observation and fault removal

Inspections

- Inspections are critical examinations of software artifacts by human inspectors aimed at discovering and fixing faults in the software systems
- Inspections are critical reading and analysis of software code or other software artifacts, such as designs, product specifications, test plans, etc
- Inspections are typically conducted by multiple human inspectors, through some coordination process. Multiple inspection phases or sessions may be used
- Faults are detected directly in inspection by human inspectors, either during their individual inspections or various types of group sessions
- Identified faults need to be removed as a result of the inspection process, and their removal also needs to be verified
- The inspection processes vary, but typically include some planning and follow-up activities in addition to the core inspection activity
- The formality and structure of inspections may vary, from very informal reviews and walkthroughs, to fairly formal variations of Fagan inspection, to correctness inspections approaching the rigor and formality of formal methods

Non-Test Defect Removal Methods

- Requirement inspections
- Design inspections
- Code inspections
- Test plan reviews
- Test-case inspections
- User documentation editing or reviews

Testing Defect Removal Methods

- Unit test by individual programmers
- New function testing
- Regression testing
- Performance testing
- Integration testing
- System testing
- Field test (external beta test)

Defect Removal

- Not all defects are equal when it comes to removal
- Requirements errors, design problems, and “bad fixes” are particularly difficult

Software Defect Origins & Defect Removal Effectiveness

	Req. Defects	Design Defects	Code Defects	Doc. Defects	Pref. Defects
Reviews / Inspections	Fair	Excellent	Excellent	Good	Fair
Prototypes	Good	Fair	Fair	N/A	Good
Testing (all forms)	Poor	Poor	Good	Fair	Excellent
Correctness Proofs	Poor	Poor	Good	Fair	Poor

Defect Removal Efficiency

- Accumulation of defect statistics for errors found prior to delivery, and then for a predetermined period after deployment (usually one year)
- US averages: 85%
- Best projects in best US companies: 99%

Defect Repair Rates

- Reported data on defect repair rates is not consistent
- Defect repair rates usually decline as cyclomatic and essential complexity increases
- Defect repair rates increase with experience in application, language, inspections, structured design and coding methods
- Defect repair rates are higher for maintenance specialists than others during maintenance phase
- For coding errors, they correlate with comment density. IBM’s study concluded that 18% comment density is ideal
- It also found, flow charts had no impact, but good error messages had great impact

Defect Seeding

- Willful insertion of errors into a software deliverable prior to a review, inspection, or testing activity
- It is the quickest way of determining defect removal efficiency
- Considered unpleasant by many

Defect Severity Levels

- Most software defect tracking systems include a multi-tier “severity level” scale
- For example,

- Severity level 1: total failure of application
- Severity level 2: failure of major function(s)
- Severity level 3: minor problem
- Severity level 4: cosmetic problem

Defect Tracking

- It is important to use an accurate and automated defect tracking system
- Defect tracking tools
  - Tracking defects by severity level and by origin
  - Routing defects to appropriate repair facility
  - Keeping records of duplicate defects
  - Invalid defects
  - Repair information against defects

Defect Prevention

- We do not want defects or faults to enter our work products, requirements, design, code, or other documents
- We try to eliminate the error sources in defect prevention
- Defect prevention is very difficult to understand, study, and quantify

Philosophy of Defect Prevention

- 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 intentions are the causes for faults, formal methods can help us prevent such deviations
- If non-conformance to selected processes or standards is the problem that leads to fault injections, then process conformance or standard enforcement can help use prevent the injection of related faults
- If certain tools or technologies can reduce fault injections under similar environments, they should be adopted

Education and Training

- Education and training provide people-based solutions for error source elimination
- The people factor is the most important factor that determines the quality and, ultimately, the success or failure of most software projects
- Education and training of software professionals can help them control, manage, and improve the way they work

Focus of Education & Training

- Product and domain specific knowledge
- Software development knowledge and expertise
- Knowledge about Development methodology, technology, and tools
- Development process knowledge

Product and Domain Specific Knowledge

- If the people involved are not familiar with the product type or application domain, there is a good chance that wrong solutions will be implemented

Software Development Knowledge and Expertise

Knowledge about Development Methodology, Technology, and Tools

Development Process Knowledge

Formal Methods

- Formal methods provide a way to eliminate certain error sources and to verify the absence of related faults
- Formal methods include
  - Formal specification
  - Formal verification

Formal Specification

- Formal specification is concerned with producing an unambiguous set of product specifications so that customer requirements, as well as environmental constraints and design intentions, are correctly reflected, thus reducing the chances of accidental fault injections

Formal Verifications

- Formal verification checks the conformance of software design or code against these formal specifications, thus ensuring that the software is fault free with respect to its formal specifications
- There are a number of formal method approaches. The oldest and most influential formal method is the so-called axiomatic approach
- The research in this area is on-going and depending on the real need of the software applications, formal methods are being used
- The biggest obstacle to formal methods is the high cost associated with the difficult task of performing these human intensive activities correctly without adequate automated support
- This fact also explains, to a degree, the increasing popularity of limited scope and semi-formal approaches

Other Defect Prevention Approaches

- Formal requirements analysis, i.e., JAD
- Formal risk-analysis early in the development
- Prototyping
- Structured programming methods
- Certified reusable design and code components

Software Defect Prevention

	Req. Defect	Design Defects	Code Defects	Document Defects	Perf. Defects
JAD	Excellent	Good	N/A	Fair	Poor
Prototypes	Excellent	Excellent	Fair	N/A	Excellent
Structured Methods	Fair	Good	Excellent	Fair	Fair
CASE Tools	Fair	Good	Fair	Fair	Fair
Blueprints / Reusable Code	Excellent	Excellent	Excellent	Excellent	Good
QFD	Good	Excellent	Fair	Poor	Good

Root Causes of Poor Software Quality

- Inadequate training of managers and staff
- Inadequate defect and cost measurement
- Excessive schedule pressure
- Insufficient defect removal
- High complexity levels
- Ambiguous and creeping requirements and design (feature race & gimmicks)
- We have just finished the discussion on root causes of poor software quality
- Now let us look at the status of the software industry's seriousness of the software industry with respect to the software quality assurance

Quality Assurance Organizations

• No quality assurance	60%
• Token quality assurance	20%
• Passive quality assurance	15%
• Active quality assurance	5%

- There is another point that must be remembered that software varies from industry to industry
- The focus on software quality naturally is dependent on the industry, as well as the importance of the software application. More critical applications, naturally, need to have higher software quality than others

Software Quality in Six Sub- Industries

- Systems software that controls physical devices
- Information systems that companies build for their own use
- Outsource or contract software built for clients
- Commercial software built by vendors for lease or sale
- Military software built following various military standards
- End-user software built for private use by computer literate workers or managers

Characteristics of Quality Laggards

- We'll now discuss the characteristics of companies, which produce poor quality software

Quality Laggards

- No software quality measurement program of any kind
- No usage of formal design and code inspections
- No knowledge of the concepts of defect potentials and defect removal efficiency
- Either no quality assurance group or a group that is severely understaffed
- No trained testing specialists available
- Few or no automated quality assurance tools
- No quality and reliability estimation capability
- Minimal or no software project management tools available
- No automated risk assessment or avoidance capability
- From a low of one to a high of perhaps four distinct testing stages
- No test library or test-case management tools available
- No complexity analysis tools utilized
- Defect potentials averaging more than 6 defects per function point
- Defect removal efficiency averaging less than 80%
- Executive and managerial indifference (and ignorance) of quality matters

Other Related Issues

- Staff morale and voluntary attrition
- Market shares and competitive positioning
- Litigation and product recalls
- Quality laggards are the very companies with highest probability of cancelled projects, several schedule overruns, and severe overruns
- It is no coincidence that software groups among the quality laggards also tend to be candidates for immediate replacement by outsource organizations
- Let's now discuss the reverse of the practices of the quality laggards
- Let's discuss the quality attributes, which when exhibited will result in high quality software

Quality Attributes

- Quality attributes set is a way to represent customer quality requirements
- Ask your current and prospective customers about their definition of quality
- Develop a quality assurance program based on the requirements of your customers

Categories of Quality Attributes

- Product-specific quality attributes
- Organization-specific quality attributes

Product-Specific Attributes

- Ease of use
- Documentation
- Defect tolerance
- Defect frequency
- Defect impact
- Packaging
- Price versus reliability
- Performance

Organization-Specific Attributes

- Service and support
- Internal processes

Achieving High Levels of Software Quality

- Enterprise-wide quality programs
- Quality awareness and training methods
- Quality standards and guidelines
- Quality analysis methods
- Quality measurement methods
- Defect prevention methods
- Non-test defect removal methods
- Testing methods
- User-satisfaction methods
- Post-release quality control

Best in Class Quality Results

- Quality measurements
- Defect prevention
- Defect and quality estimation automation
- Defect tracking automation
- Complexity analysis tools
- Test coverage analysis tools
- Formal inspections
- Formal testing by test specialists
- Formal quality assurance group
- Executive and managerial understanding of quality

Two Components of Software Quality Improvement

- Reductions in total defect potentials using methods of defect prevention
- Improvements in cumulative removal efficiency levels

Project Management Approaches and High Software Quality

- Use of automated project estimation methods
- Use of automated project planning methods
- Use of early and automated estimates of software defect potentials
- Use of early and automated estimates of software defect removal efficiency
- Formal risk-analysis
- Provision of adequate time for pre-test inspections
- Historical quality data from similar projects available
- Milestone tracking automated and thorough
- Defect tracking automated and thorough
- Management focus concentrated on achieving excellent results

Project Management Approaches and Poor Software Quality

- Exact opposite of the project management approaches correlating with high software quality
- In the previous lecture we talked about a software quality assurance (or SQA) group. Let's now see what is an SQA group, why we need it, and what are it's activities

SQA Group

- Every company, which wants to establish a reputation for producing high quality software, must establish a Software Quality Assurance (SQA) Group within the company
- This groups must be funded properly and management must pay attention to the reports and presentations made by this group
- The SQA group report directly to the line- management and not to the project management
- The personnel of the SQA group must work with the project management team, and vice versa to produce high quality software for the company – which is the ultimate goal
- The SQA group is needed to monitor the quality assurance-related activities in a company

SQA Group's Activities

- Preparation of an SQA plan for a project
- Participation in the development of the project's software process description
- Review of software engineering activities to verify compliance with the defined software process
- Audit of designed software work products to verify compliance with those defined as part of the software process
- Ensure that deviations in software work and work products are documented and handled according to a documented procedure
- Record any noncompliance and reports to senior management

SQA Plan

- Evaluations to be performed
- Audits and reviews to be performed
- Standards that are applicable to the project
- Procedures for error reporting and tracking
- Documents to be produced by the SQA group
- Amount of feedback provided to the software project team
- (we'll discuss SQA Plan in detail later in the course)

Software Quality Personnel

- Unfortunately are under-paid
- Usually are let go first in times of crisis
- "Top-gun" SQA personnel and managers with proven track record are in high demand from companies that have active QA programs

Costs of Software Quality

- Defects prevention costs
- User satisfaction optimization costs
- Data quality defect prevention costs
- Data quality defect removal costs
- Quality awareness/training costs
- Non-test defect removal costs
- Testing defect removal costs
- Post-release customer support costs
- Litigation and damage award costs
- Quality savings from reduced

scrap/rework
- Quality savings from reduced user downtime
- Quality value from reduced time-to-market intervals
- Quality value from enhanced competitiveness
- Quality value from enhanced employee morale
- Quality return on investment

Economics of Software Quality

- High quality software applications have shorter development schedules than low quality applications because they do not get hung up in integration and testing due to excessive defect levels
- High quality software applications have lower development and maintenance costs than low quality applications. This is because the cumulative costs of finding and fixing bugs is often the major cost driver for software projects
- High quality software applications have better reliability levels and longer mean times to failure than low quality applications
- High quality commercial software packages have larger market shares than low quality commercial software packages
- High quality software achieves better user-satisfaction ratings than low quality software
- High quality software projects score better on employee morale surveys than do low quality software projects
- High quality software produced under contract or an outsource agreement has a much lower probability of ending up in court for breach of contract or malpractice litigation than low quality software
- High quality software benefits or augments the performance levels of users, while poor quality tends to degrade worker performance
- Poor quality software can trigger truly massive unplanned expense levels. Denver airport example

Quality Measurement Questions

- What should be measured for quality?
- How often should quality measurement be taken and reported?

Quality Measurement Categories

- Measurement of defects or bugs in software
  - 100% of software projects
- Measurement of user-satisfaction levels
  - Only for software projects where clients can be queried and actually use the software consciously

Software Defect Quality Measurements

- Defect volumes (by product, by time period, by geographic region)
- Defect severity levels
- Special categories (invalid defects, duplicates, un-duplicatable problems)
- Defect origins (i.e., requirements, design, code, documents, or bad fixes)

- Defect discovery points (i.e., inspections, tests, customer reports, etc.)
- Defect removal efficiency levels
- Normalized data (i.e., defects per function point or per KLOC)
- Causative factors (i.e., complexity, creeping requirements, etc.)
- Defect repair speeds or intervals from the first report to the release of the fix

Software User-Satisfaction Quality Measurements

- User perception of quality and reliability
- User perception of features in the software product
- User perception of ease of learning
- User perception of ease of use
- User perception of customer support
- User perception of speed of defect repairs
- User perception of speed of adding new features
- User perception of virtues of competitive products
- User perception of the value versus the cost of the package

Who Measures User-Satisfaction?

- Marketing or sales organization of the software company
- User associations
- Software magazines
- Direct competitors
- User groups on the internet, etc.
- Third-party survey groups

Gathering User-Satisfaction Data

- Focus groups of customers
- Formal usability laboratories
- External beta tests
- Requests from user associations for improvements in usability
- Imitation of usability features of competitive or similar products by other vendors

Barriers to Software Quality Measurement

- Lack of understanding of need to measure quality
- Often technical staff shies away from getting their work measured
- Historically, “lines of code” or LOC and “cost per defect” metrics have been used, which are a poor way of measuring software quality

Object-Oriented Quality Levels

- OO technology is being adopted world-wide with a claim that it produces better quality software products
- OO technology has a steep learning curve, and as a result it may be difficult to achiever high quality software
- More data needs to be reported
- UML may play a significant role

Orthogonal Defect Reporting

- The traditional way of dealing with software defects is to simply aggregate the total volumes of bug reports, sometimes augmented by severity levels and assertions as to whether defects originated in requirements, design, code, user documents, or as "bad fixes"
- In the orthogonal defect classification (ODC) method, defects are identified using the following criteria
  - Detection method (design/code inspection, or any of a variety of testing steps)
  - Symptom (system completely down, performance downgraded, or questionable data integrity)
  - Type (interface problems, algorithm errors, missing function, or documentation error)
  - Trigger (start-up / heavy utilization / termination of application, or installation)
  - Source (version of of the projects using normal configuration control identification)

Outsourcing and Software Quality

- Outsourcing in software industry is done in a variety of ways
- Every situation introduces new challenges for development of high quality software
- Software quality metrics must be mentioned in the outsourcing contract

Quality Estimating Tools

- Estimating defect potentials for bugs in five categories (requirements, design, coding, documentation, and bad fixes)
- Estimating defect severity levels into four categories, ranging from 1 (total or catastrophic failure) to severity 4 (minor or cosmetic problem)
- Estimating the defect removal efficiency levels of various kinds of design reviews, inspections, and a dozen kinds of testing against each kind and severity of defects
- Estimating the number and severity of latent defects present in a software application when it is delivered to users
- Estimating the number of user-reported defects on an annual basis for up to 20 years
- Estimating the reliability of software at various intervals using mean-time to failure (MTTF) and/or mean-time between failures (MTBF) metrics
- Estimating the "stabilization period" or number of calendar months of production before users can execute the application without encountering severe errors.
- Estimating the efforts and costs devoted to various kinds of quality and defect removal efforts such as inspections, test-case preparation, defect removal, etc.
- Estimating the number of test cases and test runs for all testing stages
- Estimating maintenance costs for up to 20 years for fixing bugs (also for additions)
- Estimating special kinds of defect reports including duplicates and invalid reports which trigger investigative costs but no repair costs

Quality Process Metrics

- Defect arrival rate
- Test effectiveness
- Defects by phase
- Defect removal effectiveness
- Defect backlog
- Backlog management index
- Fix response time
- Percent delinquent fixes
- Defective fixes

Product Metrics

- Defect density
- Defects by severity
- Mean time between failures
- Customer-reported problems
- Customer satisfaction

Function Point Metric

- It was developed at IBM and reported to public in 1979
- It is a way of determining the size of a software application by enumerating and adjusting five visible aspects that are of significance to both users and developers
- Inputs that enter the application (i.e., Input screens, forms, commands, etc.)
- Outputs that leave the application (i.e., Output screens, reports, etc.)
- Inquiries that can be made to the application (i.e., Queries for information)
- Logical files maintained by the application (i.e., Tables, text files, etc.)
- Interfaces between the application and others (i.e., shared data, messages, etc.)
- Once the raw total of these five factors has been enumerated, then an additional set of 14 influential factors are evaluated for impact using a scale that runs from 0 (no impact) to 5 (major impact)

Litigation and Quality

- Relevant factors for software quality
  - Correctness, reliability, integrity, usability, maintainability, testability, understandability
- Irrelevant factors for software quality
  - Efficiency, flexibility, portability, reusability, interoperability, security
- It is important to narrow down the scope of quality definition similar to hardware warranties

Schedule Pressure and Quality

- Healthy pressure
  - Motivates and keeps morale of the personnel high
- Excessive pressure
  - Has serious negative impact on the morale of personnel
  - Can lead to low quality software
- Project life cycle quality assurance activities are process oriented, in other words, linked to completion of a project phase, accomplishment of a project milestone, and so forth
- The quality assurance activities will be integrated into the development plan that implements one ore more software development models – waterfall, prototyping, spiral.

- The SQA planners for a project are required to determine
    - The list of quality assurance activities needed for a project
    - For each quality assurance activity
        - ✓ Timing
        - ✓ Who performs the activity and the resources needed
        - ✓ Resources required for removal of defects and introduction of changes

A Word of Caution

- Some development plans, QA activities are spread throughout the process, but without any time allocated for their performance or for the subsequent removal of defects. As nothing is achieved without time, the almost guaranteed result is delay, caused by "unexpectedly" long duration of the QA process
- Hence, the time allocated for QA activities and the defects corrections work that follow should be examined
- The intensity of the quality assurance activities planned, indicated by the number of required activities, is affected by project and team factors

Project Factors

- Magnitude of the project
- Technical complexity and difficulty
- Extent of reusable software component
- Severity of failure outcomes if the project fails

Team Factors

- Professional qualification of the team members
- Team acquaintance with the project and its experience in the area
- Availability of staff members who can professionally support the team
- Familiarity with team members, in other words the percentage of new staff members in the team

Why Error-Prone Modules?

- Excessive schedule pressure on the programmers
- Poor training or lack of experience in structured methods
- Rapidly creeping requirements which trigger late changes
- High complexity levels with cyclomatic ranges greater than 15

"Good Enough" Software Quality

- Rather than striving for zero-defect levels or striving to exceed in 99% in defect removal efficiency, it is better to ship software with some defects still present in order to speed up or shorten time to market intervals
- Developed by the fact that major commercial software companies have latent software bugs in their released products
- Major commercial software companies have cumulative defect removal efficiency of 95% (and 99% on their best projects)

- This concept is very hazardous for ordinary companies, which usually have their defect removal efficiency level between 80%-85%
- Quality will be decrease for these companies

Data Quality

- Extremely important to understand issues of data quality
- Data results in (useful | useless) information
- Usually, governments are holders of largest data banks (are they consistent?)
- Companies are increasingly using data to their advantage over competitors
- Data warehouses present a unique challenge to keep data consistent
- Another problem is the interpretation of data

Software Requirements

Introduction

- Requirements form the basis for all software products
- Requirements engineering is the process, which enables us to systematically determine the requirements for a software product

Requirement

- Something required, something wanted or needed
    - Webster's dictionary
- Difference between *wanted* and *needed*

An Important Point

- We are discussing requirements with reference to quality assurance and management
- Every item we discuss, try to associate issues of quality assurance with it

Importance of Requirements

- The hardest single part of building a software system is deciding what to build...No other part of the work so cripples the resulting system if done wrong. No other part is difficult to rectify later
    - Fred Brooks

Software Requirements

- A complete description of *what* the software system will do without describing *how* it will do it
- External behavior
- Software requirements may be:
    - Abstract statements of services and/or constraints
    - Detailed mathematical functions
    - Part of the bid of contract
    - The contract itself
    - Part of the technical document, which describes a product

IEEE Definition

- A condition or capability that must be met or possessed by a system...to satisfy a contract, standard, specification, or other formally imposed document
    - IEEE Std 729

Sources of Requirements

- Stakeholders
    - People affected in some way by the system
    - Stakeholders describe requirements at different levels of detail
- Documents

- Existing system
- Domain/business area

Examples of Requirements

- The system shall maintain records of all payments made to employees on accounts of salaries, bonuses, travel/daily allowances, medical allowances, etc.
- The system shall interface with the central computer to send daily sales and inventory data from every retail store
- The system shall maintain records of all library materials including books, serials, newspapers and magazines, video and audio tapes, reports, collections of transparencies, CD-ROMs, DVDs, etc.

Kinds of Software Requirements

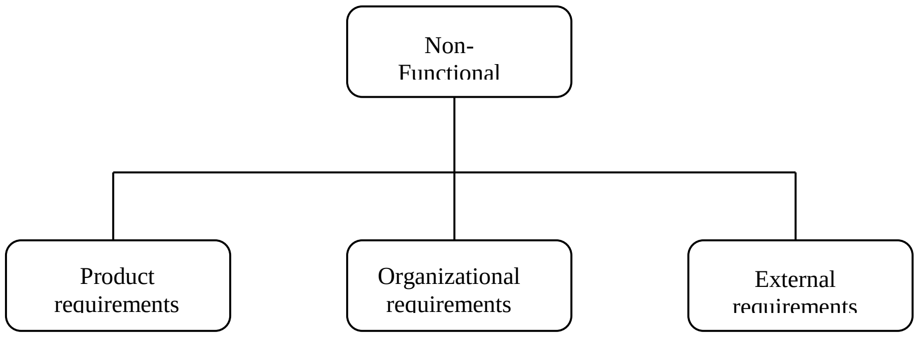
- Functional requirements
- Non-functional requirements
- Domain requirements
- Inverse requirements
- Design and implementation constraints

Functional Requirements

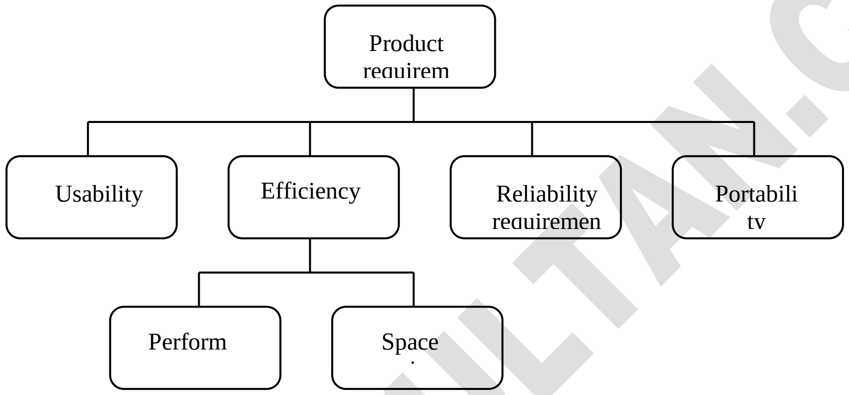
- Statements describing what the system does
- Functionality of the system
- Statements of services the system should provide
    - Reaction to particular inputs
    - Behavior in particular situations
- Sequencing and parallelism are also captured by functional requirements
- Abnormal behavior is also documented as functional requirements in the form of exception handling

Non-Functional Requirements

- Most non-functional requirements relate to the system as a whole. They include constraints on timing, performance, reliability, security, maintainability, accuracy, the development process, standards, etc.
- They are often more critical than individual functional requirements
- Capture the emergent behavior of the system, that is they relate to system as a whole
- Many non-functional requirements describe the quality attributes of the software product
- For example, if an aircraft system does not meet reliability requirements, it will not be certified as 'safe'
- If a real-time control system fails to meet its performance requirements, the control functions will not operate correctly

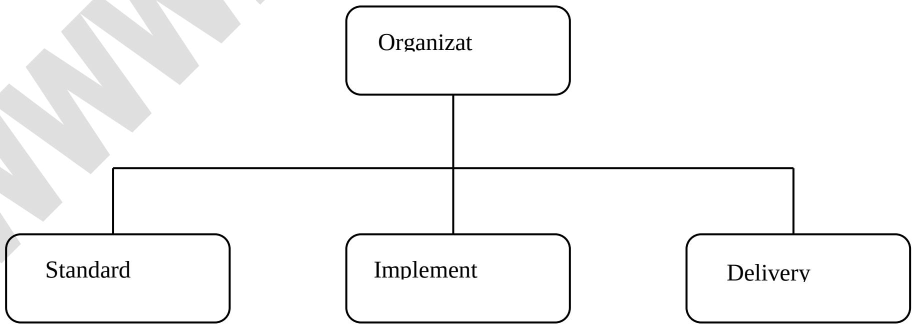


Product Requirements

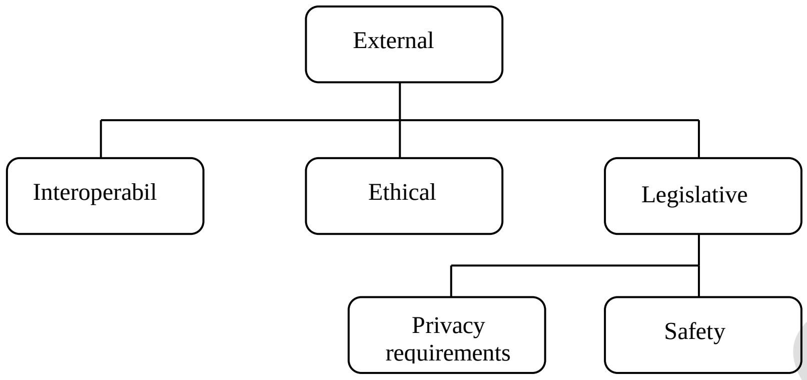


- The system shall allow one hundred thousand hits per minute on the website
- The system shall not have down time of more than one second for continuous execution of one thousand hours

Organizational Requirements



External Requirements



- The system shall not disclose any personal information about members of the library system to other members except system administrators
- The system shall comply with the local and national laws regarding the use of software tools

Observations on Non-Functional Requirements

- Non-functional requirements are written to reflect general goals for the system. Examples include:
  - Ease of use
  - Recovery from failure
  - Rapid user response
- Goals are open to misinterpretation
- Objective verification is difficult
- Distinction between functional and non-functional is not always very clear
- Non-functional requirements should be written in a quantitative manner as much as possible, which is not always easy for customers
- For some goals, there are no quantitative measures, e.g., maintainability
- Goals can be useful to designers and developers, as they give clues to them about priorities of the customers
- Chances of conflicts within non-functional requirements are fairly high, because information is coming from different stakeholders. For example, different stakeholders can give different response times or failure tolerance levels, etc.
- Some negotiations must be done among different stakeholders, to achieve an agreement in these situations
- Non-functional requirements should be highlighted in the requirements document, so that they can be used to build the architecture of the software product

Domain Requirements

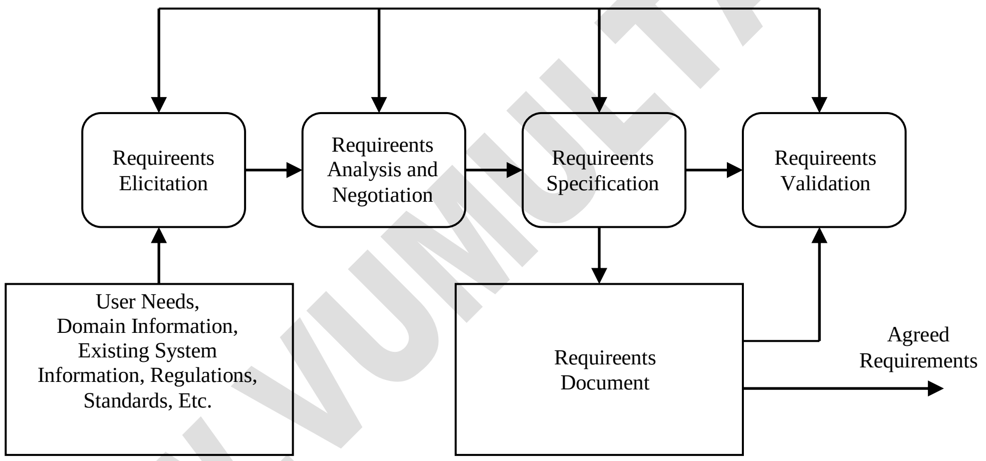
- Requirements that come from the application domain and reflect fundamental characteristics of that application domain
- Can be functional or non-functional
- These requirements, sometimes, are not explicitly mentioned, as domain experts find it

difficult to convey them. However, their absence can cause significant dissatisfaction
- Domain requirements can impose strict constraints on solutions. This is particularly true for scientific and engineering domains
- Domain-specific terminology can also cause confusion

Inverse Requirements
- They explain what the system shall **not** do. Many people find it convenient to describe their needs in this manner
- These requirements indicate the indecisive nature of customers about certain aspects of a new software product

Design and Implementation Constraints
- They are development guidelines within which the designer must work, which can seriously limit design and implementation options
- Can also have impact on human resources

Requirements Engineering Process (Requirements Engineering Activities)


Requireents Elicitation
Requirements Analysis and Negotiation
Requirents Specification
Requireents Validation
User Needs, Domain Information, Existing System Information, Regulations, Standards, Etc.
Requirents Document
Agreed Requirements

Requirements Elicitation
- Determining the system requirements through consultation with stakeholders, from system documents, domain knowledge, and market studies
- Requirements acquisition or requirements discovery

Requirements Analysis and Negotiation
- Understanding the relationships among various customer requirements and shaping those relationships to achieve a successful result
- Negotiations among different stakeholders and requirements engineers
- Incomplete and inconsistent information needs to be tackled here
- Some analysis and negotiation needs to be done on account of budgetary constraints

Requirements Specification

- Building a tangible model of requirements using natural language and diagrams
- Building a representation of requirements that can be assessed for correctness, completeness, and consistency

Requirements Document

- Detailed descriptions of the required software system in form of requirements is captured in the requirements document
- Software designers, developers and testers are the primary users of the document

Requirements Validation

- Reviewing the requirements model for consistency and completeness
- This process is intended to detect problems in the requirements document, before they are used to as a basis for the system development
- Requirements need to be validated for
  - consistency
  - testability
  - performance issues
  - conformance to local/national laws
  - conformance to ethical issues
  - conformance to company policies
  - availability of technology

Requirements Management

- Identify, control and track requirements and the changes that will be made to them
- It is important to trace requirements both ways
  - origin of a requirement
  - how is it implemented
- This is a continuous process

Requirements Problems

- The requirements don't reflect the real needs of the customer for the system
- Requirements are inconsistent and/or incomplete
- It is expensive to make changes to requirements after they have been agreed upon
- There are misunderstandings between customers, those developing the system requirements, and software engineers developing or maintaining the system
- The requirements are written using complex conditional clauses (if A then B then C…), which are confusing
- Terminology is used in a sloppy and inconsistent way
- The writers of the requirement assume that the reader has a specific knowledge of the domain or the system and they leave essential information out of the requirements document

Impact of Wrong Requirements

- Difficult to check the requirements for errors and omissions
- Different interpretations of the requirements may lead to contractual disagreements between customer and the system developer

- When requirements are wrong, systems are late, unreliable and don’t meet customers needs
- This results in enormous loss of time, revenue, market share, and trust of customers
- Conformance to wrong requirements will result in a wrong system

Requirements Defects

- All four categories of defects are found in requirements
  - Errors of commission
  - Errors of omission
  - Errors of clarity and ambiguity
  - Errors of speed and capacity
- If not prevented or removed, requirements defects usually flow downstream into design, code, and user manuals
- Historically, requirements defects are most expensive and troublesome to eliminate
- Every effort should be made to eliminate requirements defects
- For requirements errors, prevention usually works better than removal
- Errors of omission are most common among requirements defects
- Famous example is the Y2K problem
- Second most common errors are those of clarity and ambiguity
- Primarily, because natural languages (like English) are used to state requirements, while such languages are themselves ambiguous
- For example: object
- Errors of commission can also find their way into the requirements documents
- Performance errors can also be found in requirements documents

Prevention Vs Removal

- For requirements errors, prevention is usually more effective than removal
- Joint application development (JAD), Quality Function Deployment (QFD), and prototyping are more effective in defect prevention
- Requirements inspections and prototyping play an important role defect removal

Changing/Creeping Requirements

- Requirements will change, no matter what
- A major issue in requirements engineering is the rate at which requirements change once the requirements phase has “officially” ended
- This rate is on average 3% per month in the subsequent design phase, and will go down after that
- This rate should come down to 1% per month during coding
- Ideally, this should come down to no changes in testing

Defects and Creeping Requirements

- Studies have shown that very significant percentage of delivered defects can be traced back to creeping user requirements
- This realization can only be made, if defect tracking, requirements traceability, defect

removal efficiency, and defect rates are all monitored for software projects

Damage Control of Creeping Requirements

- Following quality assurance mechanisms can limit the damage done by creeping requirements
  - Formal change management procedures
  - State-of-the-art configuration control tools
  - Formal design and code inspections

Problems with Natural Languages

- Lack of clarity
- Requirements confusion
- Requirements amalgamation
- Natural language understanding relies on the specification readers and writers using the same words for same concept
- A natural language requirements specification is over-flexible. You can say the same thing in completely different ways
- It is not possible to modularize natural language requirements. It may be difficult to find all related requirements
  - To discover the impact of a change, every requirement have to be examined

Writing Requirements

- Requirements specification should establish an understanding between customers and suppliers about what a system is supposed to do, and provide a basis for validation and verification
- Typically, requirements documents are written in natural languages (like, English, Japanese, French, etc.)
- Natural languages are ambiguous
- Structured languages can be used with the natural languages to specify requirements
  - These languages cannot completely define requirements
  - They are not understandable by all stakeholders

Essentials for Writing Requirements

- Requirements are read more often than they are written. Investing effort in writing requirements, which are easy to read and understand is almost always cost-effective
- Readers of requirements come from diverse backgrounds. Requirements writers should not assume that readers have the same background and knowledge as them
- Writing clearly and concisely is not easy. If you don't allow sufficient time for requirements descriptions to be drafted, reviewed and improved, you will inevitably end up with poorly written requirements
- Different organizations write requirements at different levels of abstraction from deliberately vague product specifications to detailed and precise descriptions of all aspects of a system
- Level of detail needed is dependent on

    - Type of requirements (stakeholder or process requirements)
    - Customer expectations
    - Organizational procedures
    - External standards or regulations
- Writing good requirements requires a lot of analytic thought
- Specifying rationale of requirement is one way to encourage such thought

Guidelines for Writing Requirements

- Define standard templates for describing requirements
- Use language simply, consistently, and concisely
- Use diagrams appropriately

Use of Standard Templates

- Define a set of standard format for different types of requirements and ensure that all requirement definitions adhere to that format
- Standardization means that omissions are less likely and makes requirements easier to read and check

Using Simple Language

- Use language consistently. In particular, distinguish between mandatory and desirable requirements. It is usual practice to define mandatory requirements using '*shall*' and desirable requirements using '*should*'. Use '*will*' to state facts or declare purpose
- Use short sentences and paragraphs, using lists and table
- Use text highlighting to pick out key parts of the requirements

Using Appropriate Diagrams

- Use diagrams to present broad overviews and show relationships between entities
- Avoid complex diagrams

Using Other Descriptions of Requirements

- If readers are familiar with other types of descriptions of requirements (like equations, etc.) then use those
- Particularly applicable to scientific and engineering domains
- Don't try to write everything in natural language

Specify Requirements Quantitatively

- Specify requirements quantitatively wherever possible
- This is applicable to properties of system, such as reliability or performance
- Recollect our discussion on metrics for non-functional requirements

Additional Guidelines for Writing Requirements

- State only requirement per requirement statement
- State requirements as active sentences
- Always use a noun or a definite pronoun when referring to a thing

- Do not use more than one conjunction when writing requirements statements
- Avoid using weak words and phrases. Such words and phrases re generally imprecise and allow the expansion or contraction of requirements beyond their intent
- State the needed requirements without specifying how to fulfill them
- Write complete statements
- Write statements that clearly convey intent

Examples of Words to be avoided

- About, adequate, and/or, appropriate, as applicable, as appropriate, desirable, efficient, etc., if practical, suitable, timely, typical, when necessary

Quality Attributes of Requirements Document

Requirements Document

- The requirements document is a formal document used to communicate the requirements to customers, engineers and managers
- It is also known as software requirements specifications or SRS
- Requirements documents are usually written in natural languages (like, English, Japanese, French, etc.), which are ambiguous in themselves
- Functional requirements
- Non-functional requirements
    - Some of these are quality attributes of a software product
- Definitions of other systems which the system must integrate with
- Information about the application domain of the system, e.g., how to carry out particular types of computation
- Constraints on the process used to develop the system
- It should include both the user requirements for a system and a detailed specification of the system requirements
- In some cases, the user and system requirements may be integrated into one description, while in other cases user requirements are described before (as introduction to) system requirements
- For software systems, the requirements document may include a description of the hardware on which the system is to run
- The document should always include an introductory chapter which provides an overview of the system and the business needs
- A glossary should also be included to document technical terms
- And because multiple stakeholders will be reading documents and they need to understand meanings of different terms
- Also because stakeholders have different educational backgrounds
- Structure of requirements document is also very important and is developed on the basis of following information
    - Type of the system
    - Level of detail included in requirements

- Organizational practice
- Budget and schedule for RE process

What Should Not be Included in SRS?

- Project requirements (for example, staffing, schedules, costs, milestones, activities, phases, reporting procedures)
- Designs
- Product assurance plans (for example, configuration management plans, verification and validation plans, test plans, quality assurance plans)

Users of Requirements Documents

- System customers
- Managers
- System engineers
- System test engineers
- System maintenance engineers

Requirements for Requirements Document

- It should specify only external behavior
- It should specify constraints on the implementation
- It should be easy to change
- It should serve as a reference tool for system maintainers
- It should record forethought about the lifecycle of the system
- It should characterize acceptable responses to undesired events

Organization of Requirements Document

- Clients/developers may have there own way of organizing an SRS
- US Department of Defense
- NASA

IEEE/ANSI 830-1993 Standard

- Introduction
- General description
- Specific requirements
- Appendices
- Index

Quality Attributes of Requirements Document

- Correct
- Unambiguous
- Complete
- Verifiable
- Consistent
- Understandable by customer
- Modifiable
- Traced
- Traceable
- Design independent
- Annotated
- Concise
- Organized

Correct

- An SRS is correct if and only if every requirement stated therein represents something required of the system to be built

Unambiguous

- An SRS is unambiguous if and only if every requirement stated therein has only one interpretation
- At a minimum all terms with multiple meanings must appear in a glossary
- All natural languages invite ambiguity

Example of Ambiguity

- "Aircraft that are non-friendly and have an unknown mission or the potential to enter restricted airspace within 5 minutes shall raise an alert"
- Combination of "and" and "or" make this an ambiguous requirement

Complete

- An SRS is complete if it possesses the following four qualities
  - Everything that the software is supposed to do is included in the SRS
  - Definitions of the responses of the software to all realizable classes of input data in all realizable classes of situations is included
  - All pages are numbered; all figures and tables are numbered, named, and referenced; all terms and units of measure are provided; and all referenced material and sections are present
  - No sections are marked "To Be Determined (TBD)

Verifiable

- An SRS is verifiable if and only if every requirement stated therein is verifiable. A requirement is verifiable if and only if there exists some finite cost effective process with which a person or machine can check that the actual as-built software product meets the requirement

Consistent

- An SRS is consistent if and only if:
  - No requirement stated therein is in conflict with other preceding documents, such as specification or a statement of work
  - No subset of individual requirements stated therein conflict.
- Conflicts can be any of the following
  - Conflicting behavior
  - Conflicting terms
  - Conflicting characteristics
  - Temporal inconsistency

Understandable by Customers

- Primary readers of SRS in many cases are customers or users, who tend to be experts in an application area but are not necessarily trained in computer science

Modifiable

- An SRS is modifiable if its structure and style are such that any necessary changes to the requirements can be made easily, completely, and consistently
- Existence of index, table of contents, cross-referencing, and appropriate page-numbering
- This attribute deals with format and style of SRS

Traced

- An SRS is traced if the origin of its requirements is clear. That means that the SRS includes references to earlier supportive documents

Traceable

- An SRS is traceable if it written in a manner that facilitates the referencing of each individual requirement stated therein

Techniques for Traceability

- Number every paragraph hierarchically
- Number every paragraph hierarchically and never include more than one requirement in any paragraph
- Number every requirement with a unique number in parentheses immediately after the requirement appears in the SRS
- Use a convention for indicating a requirement, e.g., use *shall* statement

Traced and Traceability

- Backward-from-requirements traceability implies that we know why every requirement in the SRS exists.
- Forward-from-requirements traceability implies that we understand which components of the software satisfy each requirement
- Backward-to-requirements traceability implies that every software component explicitly references those requirements that it helps to satisfy
- Forward-to-requirements traceability implies that all documents that preceded the SRS can reference the SRS

Design Independent

- An SRS is design independent if it does not imply a specific software architecture or algorithm

Annotated

- The purpose of annotating requirements contained in an SRS is to provide guidance to the development organization
- Relative necessity (E/D/O)
- Relative stability

Concise

- The SRS that is shorter is better, given that it meets all characteristics

Organized

- An SRS is organized if requirements contained therein are easy to locate. This implies that requirements are arranged so that requirements that are related are co-related

Phrases to Look for in an SRS

- Always, Every, All, None, Never
- Certainly, Therefore, Clearly, Obviously, Evidently
- Some, Sometimes, Often, Usually, Ordinarily, Customarily, Most, Mostly
- Etc., And So Forth, And So On, Such As

Phrases to Look for in an SRS

- Good, Fast, Cheap, Efficient, Small, Stable
- Handled, Processed, Rejected, Skipped, Eliminated
- If…Then…(but missing Else)

The Balancing Act

- Achieving all the preceding attributes in an SRS is impossible
- Once you become involved in writing an SRS, you will gain insight and experience necessary to do the balancing act
- There is no such thing as a perfect SRS

Software Design

Design

- Synonyms: plan, arrangement, lay out, map, scheme
- Antonyms: accident, fluke, chance, guess
- Design is an activity of creating a solution that satisfies a specific goal or need
- Design is the backbone of all products and services
- Considered an artistic and heuristic activity

Software Design

- Software design is an artifact that represents a solution, showing its main features and behavior, and is the basis for implementing a program or collection of programs
- Design is a meaningful representation of something that is to be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of pre-defined criteria of "good" design
- Software design is different from other forms of design because it is not constrained by physical objects, structures, or laws
- As a result, software design tends to be much more complex than other forms of design because it is conceptually unbounded, whereas the capabilities of the human mind are bounded
- Tasks are generally ill-defined and suffer from incomplete and inaccurate specifications
- There is seldom a predefined solution, although many solutions tend to satisfy the defined task
- Viable solutions usually require broad and interdisciplinary knowledge and skill, some of which is based on rapidly changing technology
- Solutions often involve many components that have numerous interactions
- Expert designers use a mostly breadth-first approach because it allows them to mentally simulate the execution of an evolving system to detect unwanted interaction, inconsistencies, weaknesses, and incompleteness of their designs
- Thus, design is driven by known solutions, which increases performance by allowing a user to dynamically shift goals and activities
- Good designers structure problem formulations by discovering missing information, such as problem goals and evaluation criteria, and resolving many open-ended constraints
- Hence the challenge of design necessitates the use of a methodical approach based on key principles and practices to effectively and efficiently produce high quality software designs
- However, designers must occasionally deviate from a defined method in response to newly acquired information of insights
- Software design principles identify strategic approaches to the production of quality software designs
- Software design practices identify tactical methods for producing quality software designs
- Software design procedures provide an organizational framework for designing software

An Important Point

- Try to associate quality attributes with every aspect of software design

Design and Quality

- Design is the place where quality is fostered in software engineering
- Design provides us with representation of software which can be assessed for quality
- Design is the only way that we can accurately translate a customer's requirements into a finished software product or system

Without Software Design

- We risk building an unstable system
    - one that will fail when small changes are made
    - one that may be difficult to test
    - one whose quality cannot be assessed until late in the software process
    - one that will be of no or very little use for similar projects (not reusable

Software design is both a process and a model

Design Process

- It is a sequence of steps that enables a designer to describe all aspects of the software to be built
- During the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs
- Needs creative skills, past experience, sense of what makes "good" software, and an overall commitment to quality

Design Model

- Equivalent to an architect's plan for a house
- Represents the totality of the thing to be built
- Provides a variety of different views of the computer software

Design Defects

- Defects introduced during preliminary design phase are usually not discovered until integration testing, which is too late in most cases
- Defects introduced during detailed design phase are usually discovered during unit testing
- All four categories of defects are found in design models
    - Errors of commission
    - Errors of omission
    - Errors of clarity and ambiguity
    - Errors of speed and capacity
- Most common defects are errors of omission, followed by errors of commission
- Errors of clarity and ambiguity are also common, and many performance related problems originate in design process also
- Overall design ranks next to requirements as a source of very troublesome and expensive errors
- A combination of defect prevention and defect removal is needed for dealing with design defects
- Formal design inspections are one of the most powerful and successful software quality approaches of all times
- Software professionals should incorporate inspections in their software development process

Defects in Fundamental Design Topics

- Functions performed
- Function installation, invocation, control, and termination
- Data elements
- Data relationships
- Structure of the application
- Sequences or concurrency of execution
- Interfaces

Functions Performed

- Errors in descriptions of functions the application will perform, are often errors of omission
- Often omitted functions are those which, are implied functions, rather than the explicitly demanded functions

Function Installation, Invocation, Control, and Termination

- Defects in information on how t start-up a feature, control its behavior, and safely turn off a feature when finished are common in commercial and in-house software applications
- Fifty percent of the problems reported to commercial software vendors are of this class

Data Elements

- Errors in describing the data used by the application are a major source of problems downstream during coding and testing
- A minor example of errors due to inadequate design of data elements can be seen in many programs that record addresses and telephone numbers
- Often insufficient space is reserved for names, etc.

Data Relationships

- Errors in describing data relationships are very common and a source of much trouble later

Structure of the Application

- Complex software structures with convoluted control flow tend to have higher error rates
- Poor structural design is fairly common, and is often due to haste or poor training and preparation
- Tools can measure cyclomatic and essential complexity
- Prevention is often better than attempting to simplify an already complex software structure

Sequences or Concurrency of Execution

- Many errors of speed and capacity have their origin in failing to design for optimum performance
- Performance errors are a result of complex control flow, excessive branching, or too many sequential processing (use parallel processing)
- Minimize I/O operations

Interfaces

- Chronic design problem
- Incompatible data types in message communication

Errors in Eight Secondary Design Topics

- Security
- Reliability
- Maintainability
- Performance
- Human factors
- Hardware dependencies
- Software dependencies
- Packaging

Addressing Design Problems

- Continuously evaluate your design model and design process
- Use design inspections or formal technical reviews, which have proven to be the most valuable mechanism to improve quality of software ever, and especially for design
- Develop software design by following design principles and guidelines

Data Design

- The data design transforms the information domain model created during analysis into the data structures that will be required to implement the software

Architectural Design

- It defines the relationship between major structural elements of the software. Architectural design representation is derived from system specification, analysis model, and interaction of subsystems

Interface Design

- Interface design describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information

Component Design

- Component-level design transforms structural elements of software architecture into a procedural description of software components
- Think the Right Way
- To achieve a good design, people have to think the right way about how to conduct the design activity
  - Katharine Whitehead

Let' now discuss the design principles, which help us to follow the design process accurately

Design Process Principles

- The design process should not suffer from "tunnel vision"
- The design should not reinvent the wheel
- The design should "minimize the intellectual" distance between the software and the problem as it exists in the real world
- The design should exhibit uniformity and integration
- The design should be assessed for quality as it is being created, not after the fact
- The design should be reviewed to minimize conceptual (semantic) errors

When applying them

- Plan for change, because it is inevitable
- Plan for failure, because no nontrivial software system is free of defects
- How do we know if the design we have developed is of high quality?
- Now let's discuss three guidelines for evaluating software design process

Design Process Evaluation Guide

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer
- The design must be readable and understandable guide for those who generate code, write test cases, and test the software
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective

Design Model Principles

- Separation of concerns
- Modeling real-world objects
- Minimizing the interactions among cohesive design components
- The design should be traceable to the analysis model
- The design should be structured to accommodate change
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered
- Design is not coding, coding is not design
- Let's look at some design model guidelines, before we discuss the design quality attributes

Guidelines for Good Design Model

- A design should exhibit an architectural structure that
- Has been created using recognizable design patterns
- Is composed of components that exhibit good design characteristics
- Can be implemented in an evolutionary fashion, facilitating implementation and testing
- A design should be modular; that is software should be logically partitioned into elements that perform specific functions and sub-functions
- The design should contain distinct representations of data, architecture, interfaces, and components (modules)
- A design should lead to data structures that are appropriate for the objects to be implemented and are drawn from recognizable data patterns
- A design should lead to components that exhibit independent functional characteristics
- A design should lead to interfaces that reduce the complexity of connections between modules and with external environment
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis
- The guidelines we have just discussed help designers enormously in developing high quality software designs
- Designers apply these guidelines with the help of fundamental design concepts, which improve the internal and external quality of software design

Questions Answered by Design Concepts

- What criteria can be used to partition software into individual components?
- How is function or data structure detail separated from a conceptual representation of software?
- What uniform criteria define the technical quality of a software design?

Quality Design Concepts

- The beginning of wisdom for a software engineer is to recognize the difference between getting a program to work, and getting it right. M. A. Jackson
- These design concepts provide the necessary framework for "getting it right" or to produce highest possible software designs

Abstraction

- Abstraction permits one to concentrate on a problem at some level of generalization without regard to irrelevant low-level details
- Abstraction is one of the fundamental ways that we as humans cope with complexity
    - Grady Booch

Levels of Abstraction

- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment
- At lower levels of abstraction, a more procedural orientation is taken.
- Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution
- At the lowest level of abstraction, the solution is stated in a manner that can be directly implemented

Types of Abstraction

- Procedural abstraction
    - Named sequence of instructions that has a specific and limited function
    - Example: Open door
- Data abstraction
    - Named collection of data that describes a data object
    - Example: any object (ADT)
- Control abstraction
    - Implies a program control mechanism without specifying internal details
    - Example: synchronization semaphore

Refinement

- A program is developed by successively refining levels of procedural detail
- A hierarchy is developed decomposing a macroscopic statement of function in a step-wise fashion until programming language statements are reached
- Refinement is actually a process of elaboration
- There is a tendency to move immediately to full detail, skipping the refinement steps.
- This leads to errors and omissions and makes the design much more difficult to review. Perform stepwise refinement
- Abstraction and refinement are complementary concepts

Modularity

- One of the oldest concepts in software design
- Software is divided into separately named and addressable components, often called, modules, that are integrated to satisfy problem requirements
- Modularity is the single attribute of software that allows a program to be intellectually manageable
- Don't over modularize. The simplicity of each module will be overshadowed by the complexity of integration

Information Hiding

- Modules should be specified and designed so that information (procedures and data) contained within a module is inaccessible to other modules that have no need for such information
- IH means that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve a software function
- Abstraction helps to define the procedural (or informational) entities that make up the software
- IH defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module
- The greatest benefits of IH are achieved when modifications are required during testing and later, during software maintenance
- Both data and procedures are hidden from other parts of the software, inadvertent errors introduced modifications are less likely to propagate to other locations within the software

Cohesion

- Cohesion is the qualitative indication of the degree to which a module focuses on just one thing
- In other words, cohesion is a measure of the relative functional strength of a module
- A cohesive module performs one single task or is focused on one thing
- Highly cohesive modules are better, however, mid-range cohesion is acceptable
- Low-end cohesion is very bad

Types of Cohesion

- Coincidental cohesion occurs when unrelated components of a system are bundled together
- Logical cohesion occurs when components performing similar functions are put together (e.g., error handling)
- Temporal cohesion occurs when components that are activated at a single time or time span are put together
- Procedural cohesion occurs when the elements in a component form a single control sequence
- Communicational cohesion occurs when all elements of a component operate on the same input data, produce the same output data, or operate on one area of a data structure
- Sequential cohesion occurs when the output of one element in a component serves an input for another element
- Functional cohesion occurs when each part of a component is necessary for the execution of a single function
- Object cohesion occurs when the functions of a common class of objects are aggregated with the class. In this scheme, interfaces are specified to reveal as little as possible about the inner workings of the object

Coupling

- Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world
- In other words, coupling is a measure of interconnection among modules in a software structure

- Loose coupling is better. Simple connectivity is easier to understand and less prone to "ripple effect"

Type of Coupling

- Indirect coupling occurs when one object interacts with another object through a third component, usually a controller. Another form of indirect coupling occurs when using a data-driven style of computation
- Data coupling occurs when a portion of a data structure is passed among modules. When using data coupling, shorter parameter lists are preferred to longer ones
- Stamp coupling occurs when a portion of data structure is passed among modules. When using stamp coupling, a smaller number of actual arguments are preferable to a larger number because the only data that should be passed to a module is what it requires
- Control coupling occurs when information is passed to a module that affects its internal control. This is undesirable because it requires the calling module to know the internal operation of the module being called
- External coupling occurs when a module depends on the external environment
- Common coupling occurs when modules access common areas of global or shared data. This form of coupling can cause one module to unintentionally interfere with the operation of another module
- Content coupling occurs when one module uses information contained in another module

Other Design Concepts

- Software Architecture
- Control Hierarchy
- Structured Partitioning
- Data Structures
- Software Procedure
- Functional Independence
- Encapsulation
- Inheritance
- Polymorphism

A Word of Advice

- Adopt a core set of ideas for each system design, because they will improve their conceptual integrity

Design Methods

- Use a design method, which is most suitable for the problem at hand. Don't just use the latest or the most popular design method
- There are many structured design and object-oriented design methods to choose from
- Follow the design method's representation scheme. It helps in understanding design

Characteristics of Design Methods

- A mechanism for the translation of analysis model into design representation
- A notation for representing functional components and their interfaces
- Heuristics for refinement and partitioning
  Guidelines for quality assessment

B. Meyer

- Modular decomposability
- Modular compensability
- Modular understanding
- Modular continuity
- Modular protection

Modular Decomposability

- If a design method provides a systematic mechanism for decomposing the problem into sub problems, it will reduce the complexity of the overall problem

Modular Compensability

- If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel

Modular Understandability

- If a module can be understood as a standalone unit (without reference to other modules), it will be easier to build and easier to change

Modular Continuity

- If small changes to the system requirements result in changes to individual modules, rather than system-wide changes, the impact of change-induced side effects will be minimized

Modular Protection

- If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized

Programming

- The act of programming, also known as coding, produces the primary products – executable – of a software development effort
- All prior activities culminate in their development
- Programming is done in a programming language

Coding Defects

- All four categories of defects are found in source code
  - Errors of commission
  - Errors of omission
  - Errors of ambiguity and clarity
  - Errors of speed and capacity
- Errors of commission are the most common when the code is underdevelopment
- The most surprising aspect of coding defects is that more than fifty (50) percent of the serious bugs or errors found in the source code did not truly originate in the source code
- A majority of the so-called programming errors are really due to the programmer not understanding the design or a design not correctly interpreting a requirement
- Software is one of the most difficult products in human history to visualize prior to having to build it, although complex electronic circuits have the same characteristic
- Built-in syntax checkers and editors with modern programming languages have the capacity to find many “true” programming errors such as missed parentheses or looping problems

- They also have the capacity to measure and correct poor structure and excessive branching
- The kinds of errors that are not easily found are deeper problems in algorithms or those associated with misinterpretation of design
- At least five hundred (500) programming languages are in use, and the characteristics of the languages themselves interact with factors such as human attention spans and capacities of temporary memory
- This means that each language, or family of languages, tends to have common patterns of defects but the patterns are not the same from language-to-language
- There is no solid empirical data that strongly-typed languages have lower defect rates than weakly-typed languages, although there is no counter evidence either
- Of course for all programming languages, branching errors are endemic. That is, branching to the wrong location for execution of the next code segment

Defects in High-Level Languages
- Many high-level languages, such as Ada and Modula, were designed to minimize certain common kinds of errors, such as mixing data types or looping incorrect number of times
- Of course, typographical errors and syntactical errors can still occur, but the more troublesome errors have to do with logic problems or incorrect algorithms
- A common form of error with both non-procedural and procedural languages has to do with retrieving, storing, and validating data
- It may sometimes happen that the wrong data is requested
- Programming in any language is a complex intellectual challenge with a high probability of making mistakes from time to time
- Analogy with typos in a newspaper

Defects in Low-Level Languages
- Since low-level languages often manipulate registers and require that programmers setup their own loop controls, common errors involve failure to initialize registers or going through loops the wrong number of times, not allocating space for data and subroutines
- For weakly-typed languages, mismatched data types are common errors

Quality Practices for General-Purpose Programming
- Use the highest-level programming language possible
- Use integrated development environments
- Adopt a coding standard that prevents common types of defects
- Prototype user interfaces and high-risk components
- Define critical regions

Use the Highest-Level Programming Language
- Code written in higher-level programming languages is easier to read and maintain
- Any fool can write code that a computer can understand. Good programmers write code that humans can understand
- Several practical factors influence the selection of a programming language
    - Technology trends
    - Organizational informational technology strategies
    - Customer restrictions on programming language selection
    - Experience of the development team
    - Features of the programming language (e.g., to interoperate with external

systems)
- The complexity of software systems is growing quicker than our ability to develop software solutions
- For example, productivity of computer personnel increased about 6% per year during the 1990s, whereas the growth in NASA mission software is about 25% per year
- Productivity is constant in terms of program statement size. That is writing ten lines of code in assembly language requires as much work as writing ten lines of code in C++, but the functionality developed in ten lines of C++ is much more than the ten lines of assembly language code
- We are shrinking the size of the programs by using higher-level languages
- Fred Brooks has said that the advent of high-level programming languages had the greatest impact on software productivity because there was at least a factor of five improvement in productivity
- The use of high-level programming languages results in more reliable software

Use Integrated Development Environments
- Also known as IDEs, these suites include an editor, a compiler, a make utility, a profiler, and a debugger. Other tools may also be included
- Recent IDEs include tools to model software designs and implement graphical user interfaces
- These tools, if used properly, can improve the productivity 100%
- They also help identify many coding defects, as they are being introduced in the software

Adopt a Coding Standard to Prevent Common Types of Defects
- Coding standards are controversial because the choice among many candidate standards is subjective and somewhat arbitrary
- Standards are most useful when they support fundamental programming principles
- So, it is easier to adopt a standard for handling exceptions, than for identifying the amount of white-space to use for indentation
- An organization should always ask itself whether a coding standard improves program comprehension characteristics

Practices for Internal Documentation
- Specify the amount of white-space that should be used and where it should appear
  - Before and after loop statements and function definitions
  - At each indentation level (two or four spaces have been reported as improving comprehensibility of programs)
- Physically offset code comments from code when contained on the same line
- Use comments to explain each class, function, and variable contained in source code. (Comments can be from 10% and up)
  - Key interactions that a function has with other functions and global variables
  - Complex algorithms used by every function
  - Exception handling
  - Behavior and effect of iterative control flow statements and interior block statements
- Provide working examples in the user documentation or tutorial materials

Practices for Variable Definition

- Declare variables as specifically as possible and initialize them, preferably one declaration per line
- Do not use similarly named variables within the same lexical scope
- Consistently, use clear and easily remembered names for variables, classes, and functions
- Follow a uniform scheme when abbreviating name
- Do not use local declarations to hide declarations at greater scope
- Never use a variable for more than one purpose

Practices for Control Flow

- Do not assume a default behavior for multi-way branches
- Do not alter the value of an iteration variable within a loop
- Use recursion, when applicable

Practices for Functions

- Explicitly define input and output formal parameters
- Use assertions (e.g., pre- and post-conditions) to verify the accuracy and correctness of input and output formal parameters. The use of pre- and post-conditions helps programmers detect defects closer to their origin

Practices for Operations

- Make all conversion of data explicit, especially numeric data
- Do not use exact floating-point comparison operations
- Avoid using operators in potentially ambiguous situations

Practices for Exception Handling

- Process all exceptions so that personnel can more easily detect their cause
- Log important system events, including exceptions

Practices for Maintenance

- Isolate the use of nonstandard language functions
- Isolate complex operations to individual functions

Practices for Operational

- Do not permit any compilation to produce warnings
- Optimize software only after it works is complete, and only if required to achieve performance goals

Prototype User Interfaces and High-Risk Components

- User interface prototyping helps identify necessary features that software engineers might otherwise overlook
- Prototyping can reduce the development effort significantly
- Prototyping reduces development risk because is allows programmers to explore methods for achieving performance and other high-risk requirements

Define Critical Regions

- A task that interrupts an interdependent operational sequence before it is completed can leave a program in a vulnerable state, resulting in inconsistent and inaccurate results. We need a critical regions to run such transactions

- Critical regions help prevent deadlocks

What is a Review?

- A process or meeting during which a work product, or a set of work products, is presented to project personnel, managers, users, or other interested parties for comment or approval. Types include code review, design review, forma qualification review, requirements review, test readiness review
    - IEEE Std. 610.12-1990

Objectives of Reviews

- Identify required improvements in a product
- Assure that the deliverable is complete
- Assure that the deliverable is technically correct
- Measure the progress of the project
- Identify any defects early, thus resulting in cost and time savings
- Assure the quality of deliverable before the development process is allowed to continue
- Once a deliverable has been reviewed, revised as necessary, and approved, it can be safely used as a basis for further development

Colleagues as Critics

- There is no particular reason why your friend and colleague cannot also be your sternest critic
    - Jerry Weinberg

Benefits of Review

- A number of team members get an opportunity to provide their input
- Ownership of the work product is transferred from an individual to a group
- A (limited) training ground

Kinds of Reviews

- Business reviews
- Technical reviews
- Management reviews
- Walk-throughs
- Inspections

Objectives of Business Reviews

- The deliverable is complete
- The deliverable provides the information required for the next phase
- The deliverable is correct
- There is adherence to the procedures and policies

Objectives of Technical Reviews

- Point out needed improvements in the product of a single person or a team
- Confirm those parts of a product in which improvement is either not desired or not needed
- Achieve technical work or more uniform, or at least more predictable, quality than can be achieved without reviews, in order to make technical work more management
- Software reviews are a "filter" for software engineering process
- Reviews are applied at several points during software development and serve to uncover errors and defects that can then be removed
- Software reviews "purify" the software engineering activities
- Technical work needs reviewing for the same reason that pencils need erasers: To err is

human

- Another reason we need technical reviews is that although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape anyone else
- They also ensure that any changes to the software are implemented according to pre-defined procedures and standards

What Technical Reviews Are Not!

- A project budget summary
- A scheduling assessment
- An overall progress report
- A mechanism for reprisal or political intrigue!!

Objectives of Management Reviews

- Validate from a management perspective that the project is making progress according to the project plan
- Ensure a deliverable is ready for management approval
- Resolve issues that require management's attention
- Identify if the project needs a change of direction
- Control the project through adequate allocation of resources

Review Roles

- Facilitator
- Author
- Recorder
- Reviewer
- Observer

Responsibilities of Facilitator

- Responsible for providing the background of the work and assigning roles to attendees
- Encourages all attendees to participate
- Keeps the meeting focused and moving
- Responsible for gaining consensus on problems

Responsibilities of Author

- Responsible for the readiness and distribution of material to be reviewed
- During the meeting, the author paraphrases the document a section at a time
- Responsible for
  - scheduling the review
  - selecting the review participants
  - determining if the entry criteria for the review are met
  - providing information about the product during all stages
  - clarifying any unclear issues
  - correcting any problems identified
  - providing dates for rework and resolution

Responsibilities of Recorder

- Collects and records each defect uncovered during the review meeting
- Develops an issues list and identifies whose responsibility it is to resolve each issue
- Records meeting decisions on issues; prepares the minutes; and publishes the minutes, and continually tracks the action items

Responsibilities of Reviewer

- Spends time prior to the meeting reviewing information
- Makes notes of defects and becomes familiar with the product to be reviewed
- Identifies strengths of the product
- Verifies that the rework is done
- Insists upon clarifying any issues that are not clear

Responsibilities of Observer

- A new member to the project team, who learns the product and observes the review techniques

Review Guidelines

- Preparation
- Discussions
- Respect
- Agenda
- Review Records
- Resources
- Attendees

Review Frequency

- At the beginning/end of the requirements phase
- At the beginning/end of the design phase
- At the beginning/end of the code phase
- At the beginning/end of the test phase
- Approval of the test plan

Review Planning

- Distribute review package one week in advance
    - Document to be reviewed
    - Review agenda
    - Identification of the individual who will manage the agenda and schedule
    - Exit and entrance criteria for the review
    - Objective of the review
    - Names of attendees, their roles and responsibilities
    - Review location
    - Date and time of review
    - List of classifications that will be used for defects discovered (defect type, defect origin, and defect severity)
    - Procedures for handling issues raised during the review and escalation phase

Review Meeting

- Facilitator begins the meeting with an introduction of agenda, people, and description of their roles
- Author of the document proceeds to explain the materials, while reviewers raise issues based on advance preparation
- When valid problems, issues, or defects are discovered, they are classified according to their origin or severity and then recorded
- These are accompanied with the names of individuals who are responsible for resolution and the time frame during which the item will be resolved
- Related recommendations are also recorded

Guidelines for Reviewers

- Be prepared - evaluate product before the review meeting
- Review the product, not the producer
- Keep your tone mild, ask questions instead of making accusations
- Stick to the review agenda
- Raise issues, don't resolve them
- Avoid discussions of style - stick to technical correctness

Decisions at the End of a Review Meeting

- All attendees must decide whether to
    - Accept the product without further modification
    - Reject the product due to severe errors
    - Accept the product provisionally
    - Hold a follow-up review session

Review Report

- Published by the recorder, with approval from all attendees, after a week of the review meeting
- Review report consists of
    - Elements reviewed
    - Names of individuals who participated in the review
    - Specific inputs to the review
    - List of unresolved items
    - List of issues that need to be escalated to management
    - Action items/ownership/status
    - Suggested recommendations

Rework

- It is the responsibility of project manager to ensure that all defects identified in the review are fixed and retested

Follow-Up

- During the follow-up, that all discrepancies identified are resolved and the exit criteria for the review have been met
- Document lessons learned during the final report also

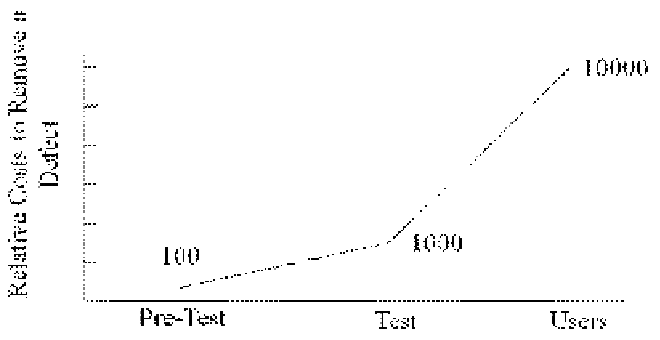
Software Inspection

Inspections:

- An inspection is a rigorous team review of a work product by peers of the producer of the work product
- The size of the team will vary with the characteristics of the work product being inspected; e.g., size, type
- The primary purpose is to find defects, recording as a basis for analysis on the current project and for historical reference and for improvement for future projects, analyzing them, and initiating rework to correct the defects
- Direct fault detection and removal
- Inspections are most effective when performed immediately after the work product is complete, but they can be held any time the work product is deemed ready for inspection
- Inspections are critical reading and analysis of software code or other software artifacts, such as designs, product specifications, test plans, etc.
- Inspections are typically conducted by multiple human inspectors, through some coordination process. Multiple inspection phases or sessions may be used
- Faults are detected directly in inspection by human inspectors, either during their individual inspections or various types of group sessions
- Identified faults need to be removed as a result of the inspection process, and their removal also needs to be verified
- The inspection processes vary, but typically include some planning and follow-up activities in addition to the core inspection activity
- Developed by Michael Fagan at IBM and were first reported in public domain in 1976
- Inspections remove software defects at reduced cost
- Inspections enable us to remove defects early in the software life cycle, and it always cheaper to remove defects earlier in than later in the software life cycle
- We know that defects are injected in every software life cycle activity
- We remove some of these defects in testing activities after code is completed
- We also know that all defects are not removed at shipment time, and these are known as latent defects
- We want to eliminate or at least minimize latent defects in the shipped software product
- It is expensive to find and remove defects in the testing phase, and even more expensive after shipment of the software
- We can use inspections to reduce these costs and improve the timelines also
- During testing, defects are found, then the programmers are notified of their presence, who will recreate the defects under the similar circumstances, fix them, re-test the software and re-integrate the software module, which were affected
- While in inspections, the inspection process is executed in the same life cycle activity, and substantial amount of rework is avoided
- This results in the reduction of costs
- If and when defects are detected after the shipment of the software, then these costs are even higher
- Many times, original development team is disbanded after the completion of the project and new staff is looking after the maintenance activity
- These people are usually not fully aware about the project
- This can result in unplanned expenses for the software development company
- On the other hand, if an effective software inspections process is in place, fewer defects enter the testing activity and the productivity of tests improve

- The costs of tests are lower and the time to complete tests is reduced
- Several studies have confirmed the reduction in project costs when defects were removed earlier

Defect Cost Relationship



- It is interesting to note that this relationship has remain consistent in the last three decades – since the earliest studies when inspections were being first reported
- In addition to the costs on project, there are additional costs to the customer for downtime, lost opportunity, etc., when defects are detected in maintenance
- Let's look at the published data from different studies of companies in which comparison of inspection costs and testing costs have been made
- These were independent studies, and so they use different units to report their results
- However, the pattern repeats that the cost of inspections is much lower than that of software testing

Reported Cost Relationship

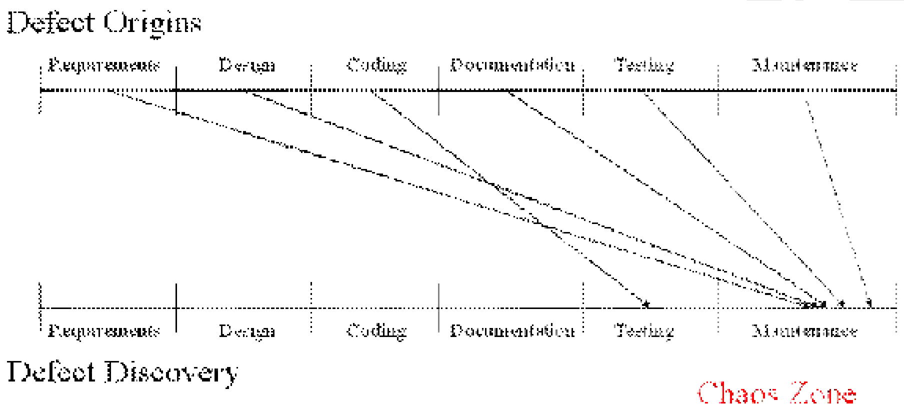
Company	Cost in Inspection	Cost in Text	Cost With Customer Discovery
IBM	$48/defect	$61 - $1030 / defect	#1770 / defect
AT&T	1 unit	20 Units	-
ICL	1.2 – 1.6 hours / defect	8.47 hours/defect	-

Company	Cost in Inspection	Cost in Text	Cost With Customer Discovery
IBM	1.4 hours	$61 - $1030 / defect	--
AT&T	$ 105/defect	$ 1700 / defect	--
ICL	1Unit	9 Times More	117 times more

Company	Cost in Inspection	Cost in Text	Cost With Customer Discovery
Shell	1Unit	30 units	--
Thom EMI	1Unit	6.8 – 26 units	96 units
Applicon, Inc	1hour	--	30 hours
Infosys	1Unit	3 – 6 units	--

- These studies clearly report data from different companies that it is cheaper to detect and remove data using software inspections as compared to software testing
- There is evidence in the literature that inspection offer significant return on investment even in their initial use
- Let' now look at inspections from another point of view
- Relating defect origin points and defect discovery
- In a project with no software inspections, defects are typically injected in the earlier activities and detected in later stages
- As a result, we get a chaos zone

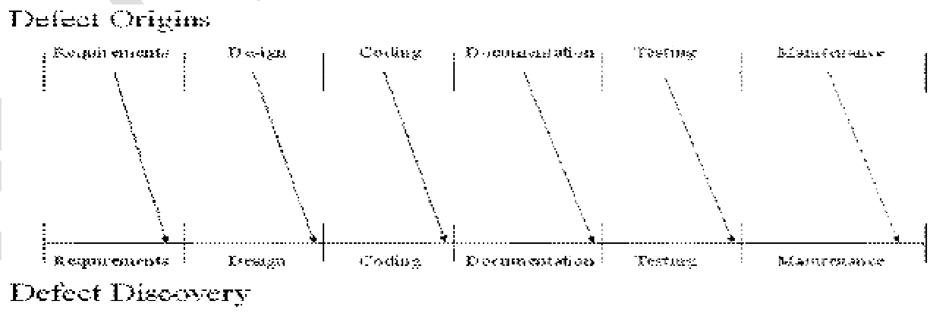
Defect Origins and Discovery Points without Usage of Formal Inspections



This situation is a mess

- If only we were able to detect defects in the same life cycle activity, we can eliminate the chaos zone, and bring some sanity back to the project team and project management
- If we introduce software inspections, we can do that

Defect Origins and Discovery Points With Usage of Formal Inspections



- Here you can see that the chaos zone has been eliminated
- This is achieved by performing inspections on work products before leaving that life cycle activity, and as a large number of requirements defects will be detected and removed during the requirements activity, design and coding defects will be detected and removed during those activities, and so on

Why Isn't Everyone Using Inspections?

- Now we are convinced that inspections have a clear value independent of any model or standard for software development, so why isn't everyone using it?

Reasons for Not Using Inspections

- There is resistance to Inspections because people view them as if they are not easy to do well
- Management often views Inspections as an added cost, when in fact Inspections will reduce cost during a project
- Development of new tools and environments
- Inspections are not the most enjoyable engineering task compared to designing and coding
- Inspections are labor intensive and low-tech
- Programmers/designers are possessive about the artifacts they create

Inspection Preconditions

- Clear and visible management support
- Defined policy
- Good training for all
- Effective procedures
- Proper planning
- Adequate resources

Success Factors

- Kept to the basics
- Trained teams rather than individuals
- Established a policy that inspections are safe
- Followed the proven method, before adapting or tailoring it
- Gave proper time for inspections to take root
- Analyzed and used the data resulting from inspections
- Built on their own successes
- Learned what was not necessary to inspect
- Rewarded the performance of inspections
- Shared the success
- Allocated budget and time for inspections
- An inspection is most successful when
  - All team members treat it as a cooperative effort to find and remove defects as early as possible during development activities
  - Inspectors with good domain knowledge of the material are available for the inspection
  - Inspectors are trained in the inspection process
- Inspections succeed to varying degrees even when these three conditions are not met
- Effectiveness may not be as good
- It must always be remembered that inspection data must not be used to evaluate the performance or capability of the work product producer
- All levels of management must accept this value, practice it, and communicate the commitment to all personnel
  - The inspection uses a checklist to prepare for the inspection and to verify the work product against historical, repetitive, and critical defect types within the domain of the

  project
  - The inspection is used to determine the suitability of the work product to continue into the next life cycle stage by either “passing” or “failing”
  - If the work product fails, it should be repaired and potentially re-inspected
  - Work Products

- Requirements specifications
- Design specifications
- Code
- User documentation
- Plans
- Test cases
- All other documents

Inspection Steps

Overview
- Provides the inspection participants a background and understanding, when warranted, of the scheduled inspection material

Preparation
- Allows time for the inspection participants to sufficiently prepare for the inspection meeting and list potential defects

Inspection meeting
- Identifies defects before work product is passed into the next project stage

Rework
- Fixes identified defects and resolves any open issues noted during the inspection

Follow-up
- Verifies that all defects and open issues have been adequately fixed, resolved, and closed out

Other Inspection Steps:

Planning and scheduling
- To ensure adequate time and resources are allocated for inspections and to establish schedules in the project for work products to be inspected, to designate the inspection team, and to ensure the entry criteria are satisfied

Data recoding
- To record the data about the defects and conduct of the inspection

Analysis meeting
- Which is held after the inspection meeting, to begin defect prevention activities

Prevention meeting
- Which is held periodically after sets of inspections have been performed to determine probable causes for selected defect types, instances, or patterns

let’s now discuss a modeling technique, which can be used to model inspections. This technique is known as Entry-Task-Validation/Verification-eXit (ETVX) technique.

ETVX Representation

The model expressed as a set of interconnected activities each of which has four sets of attributes

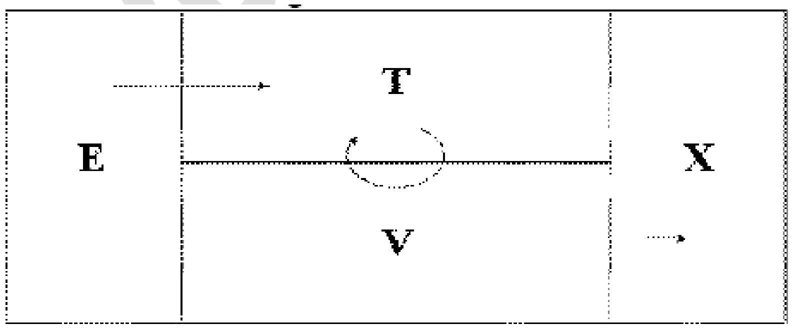
Entry (E)	The Entry section defines the entry criteria that must be satisfied for the process to be initiated, and list the work products that must be available as inputs to the process
Task (T)	The Task section defines work to be carried in performing the process. The order of the task is generally, but not strictly sequential. Some tasks may be concurrent with other tasks
Validation /Verification (V)	The validation/verification section defines steps for validating/verifying that the process has been properly executed, and that the associated work products meet project objectives
Exit (X)	The Exit section defines the exit criteria that must be satisfied for the process to be terminated. The exit criteria usually define completion and verification work products, in terms of qualitative aspects of the products

The ETVX technique indicates the relationship and flow among the four aspects of an activity and between activities. The notion of formal entry, exit, and criteria go back to the evolution of the waterfall development process. The idea is that every process step, inspection, function test, or software design has a precise entry and exit criteria

Characteristics of ETVX

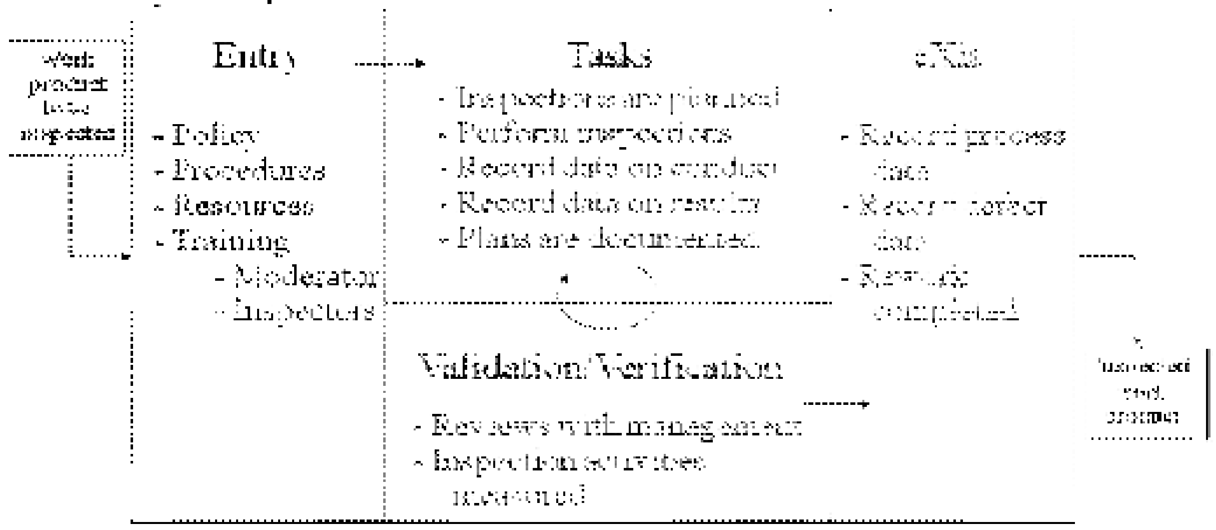
- ETVX is a task-based model
- Each task must be explicitly defined
- The basic cells can be combined to create process

The ETVX Process Definition Paradigm



The picture that you have just seen shows the ETVX model Let's now apply this model to the inspection process

Practices in the Inspection Process



Ancillary Purposes

- Improvement in productivity
- Education and increased knowledge sharing
- Developing backup/replacement capability
- Process improvement
- Early product quality visibility
- Product re-development
- Building team spirit

Productivity Improvement

- Fagan calculated a 23% productivity improvement during the VTAM study – one of the first projects in which inspections were used
- "An improvement in productivity is the most immediate effect of purging errors from the product"
  Michael Fagan
- "Inspection reduces the development cost during test by 50%" based on IBM studies
  - Norris

Education and Increased Knowledge Sharing

- Overview activity
- Preparation activity
- Inspection meeting
- Analysis meeting (causes of defect)
- Prevention meeting

Back-Up/Replacement Capability

- Many organizations have high turnover rates, and in many cases only a few people
- (or even one person) has the required knowledge of a product or key parts of a product
- Where turnover is high, knowledge can literally walking out of the door
- To mitigate this risk, some organizations have elected to inspect 100% of all work products
- Basically, they are providing backup, and this is 'dynamic backup'
- In these situation, inspections are used to spread the knowledge as fast and as far as

possible
- This education also provides a flexibility to react quicker when there are customer needs
- In some situations, maintenance of the work product may be transferred to a new organization or a subcontractor
- So new people need to be educated and trained on the work products as fast as possible
- So, inspections are used to create backups or replacement owners of work products
- The choice for when to consider these types of inspections is determined by
    - Defect backlogs
    - Change request backlogs
    - Possibilities for re-engineering
    - Risk mitigation for volatile product sections
    - Turnover rates
    - Recruitment rates
- To be successful for in these inspections, the author of the work product has to be present
- “Inspections broaden the knowledge base of the project in the group, create potential backup programmers for each module, and better inform the testers of the functions they are to test”
    - Norris

Process Improvement
- Data is gathered during the inspection and later analyzed to understand the process of doing the inspection and later to improve it

Early Product Quality Visibility
- Quality of the work product and that of the software product starts to become clear in the early stages of the software development life cycle

Product Re-Development
- Products with multiple releases can have high volumes of changes in some areas
- And, some work products with high defect rates may have to be re-engineered
- Inspections are “a very good mechanism for highlighting and prioritizing candidate areas for enhancement”

Building Team Spirit

“The review process also promotes team building. It becomes one of the first steps toward establishing a good development team, by substituting an environment where programmers work alone throughout their careers, for a programming team environment in which each individual feels free to discuss and critique everyone else’s program. Implicit in the concept of a team is the notion of working closely together, reading each other’s work, sharing responsibilities, learning each other’s idiosyncrasies both on technical and personal level, and accepting altogether as a group shared responsibility for the product where each member can expect similar rewards if the project is a success and similar penalties if the project fails”

Where Do Inspections Fit in Software Development Process?

The Cost of Inspections

- There is a cost for every activity during project life cycle. The cost is determined by many factors:
    - Capability of programmers performing the activity
    - The defined process for the activity
    - Stability of the input
- Inspections have a cost also
- Time spent on inspections is an investment. You inspect now, you invest now, and then you reap the benefits down the line
- Concern should only be when the inspections are performed for the first time
- Once the cost question is removed from management's thinking, the time needed up front in a project is no longer a concern

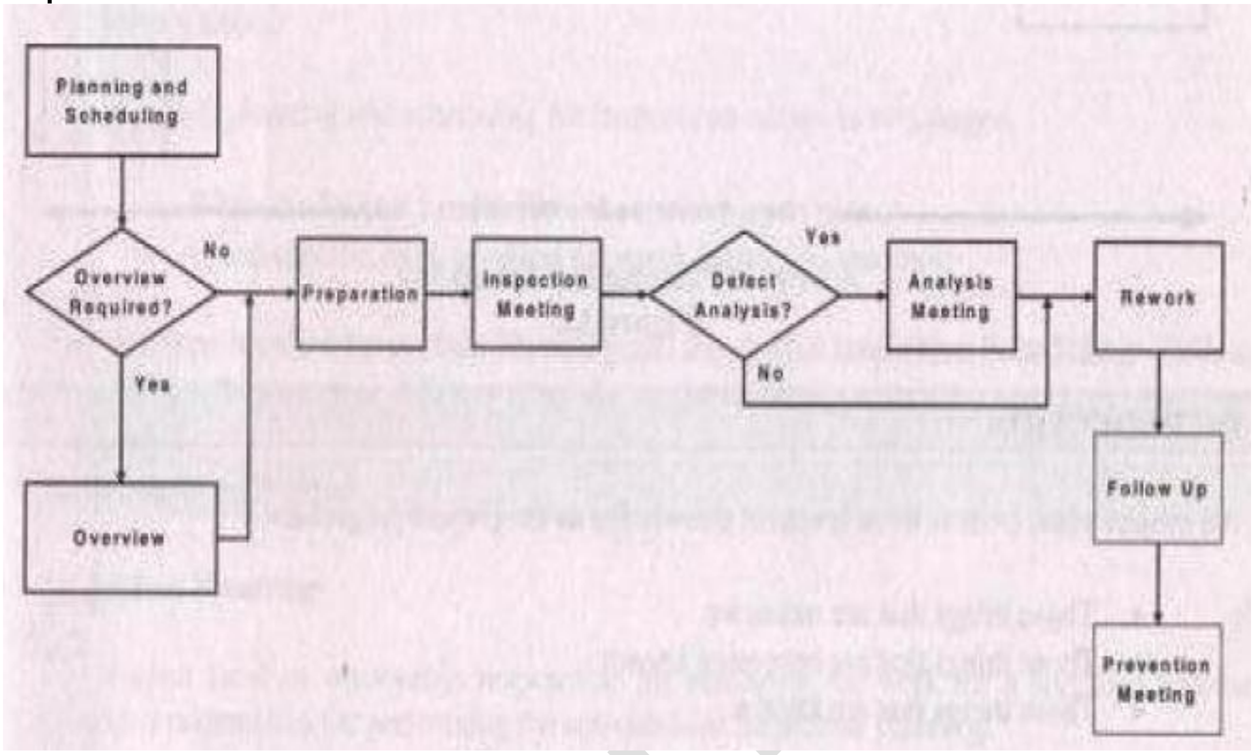
Cost in Hours/Defect During Code Inspections

COMPANY	HOURS/DEFECT
AT&T	67 – 1.4
ICL	1.2 – 1.6
JPC	1.5 – 2.1
Bell-Northem	Less than .4
HP	.2
Bull	1.43
ITT Industries	~1

What Inspections Are Not

- A review of the style of a work product
- A review of the producer, and especially not a means to evaluate the producer by management
- An impromptu meeting; it is a scheduled meeting with resource considerations to enable effectiveness
- A casual or informal meeting; there is structure and rigor for a purpose
- Typically the time or place to fix defects or discuss possible solutions
- Free! But they do yield a high return on investment
- A vehicle for shifting responsibility to inspectors for quality of the work product
- Quality assurance performed at the end of development
- It's not the products but the processes that create products that bring companies long-term success

Inspection Process Flow



Inspection Process

- Planning and scheduling
- Overview
- Preparation
- Inspection meeting
- Analysis meeting
- Rework
- Follow-up
- Prevention meeting
- Data recording and reports
- Inspection process monitoring
- We'll be using the ETVX model to describe the steps in the inspections process

Inspection Planning

- To ensure adequate time and resources are allocated for inspections and to establish schedules in the project for work products to be inspected, to designate the inspection team, and to ensure the entry criteria are satisfied
- All project plans exist at three levels of knowledge as the project progresses
    - Those things that are unknown
    - Those things that are becoming known
    - Those things that are known
- Plan details reveal themselves to the planner as a rolling wave
- The project lead must plan which inspections are to be performed at the initial stages of the project
- Unknowns become knowns
- Has two sections

  - Inspection planning
  - Inspection scheduling

Responsibility

- The project lead or whoever is responsible for managing the work for a specified software project is responsible for performing the activities for Inspection Planning

Entry Criteria

- A policy exists for inspections in the project's organization
- Planning procedures, including planning for inspections exist
- A project begins and includes the requirement to plan for inspections
- Work product types to be inspected are identified in the project plan
- Well-defined work product completion or readiness criteria are available
- Initial estimates are provided for the size of the work products to be inspected
- Expected project participants have been trained or a training plan is defined
- Goals and targets have been established for the volume or percentage of work products to be inspected

Tasks

- Determine what will be inspected
- Estimate resources for inspections and allocate budget
- Set milestones for the inspections
- Identify dependencies on other groups

Validation/Verification

- The SQA group in the organization should assure that the project plan has been documented and includes planned inspections as required by the organization policy and procedures
- Data to be gathered during this activity
- Which work products are planned for inspection
- The estimated size of work products to be inspected
- Risks
- The number of planned inspections
- Planned effort to be spent on inspections

Exit Criteria

- There is a project plan showing the inspections to be held, including resources and milestones that may be known in the early stages of the project
- Where milestones may not be known a boundary of probable dates should be noted in the plan for the inspections
- Adequate resources are allocated in the project plan for inspections

Inspection Scheduling

Responsibility

- The project lead is responsible
  - For requesting, selecting, or assigning Moderators when a work product approaches inspection readiness
  - For ensuring the work product will be ready for inspection
  - For ensuring that the participants are made available
  - For making known to a qualified Moderator that an inspection is to be scheduled

Other Roles

- The moderator with the project lead is responsible for completing inspection scheduling
    - This includes
    - Agreement on a specific date
    - Ensuring that entry criteria is met
    - Completing all logistics requirements
    - Scheduling the participants and inspection activitites

Entry Criteria

- A work product is approaching inspection readiness
- Resources are available
- The project lead makes a request to a moderator for an inspection or a set of inspections

Tasks

- Send a notification that an inspection will be needed
- Determine the inspection meeting date
- Ensure that the work product to be inspected meets entry criteria
- Schedule the inspection meeting

Validation/Verification

- The moderator remains actively involved during the inspection scheduling period and is responsible for assuring that all tasks up to completion of the inspection meeting are performed
- The SQA groups ensures that a moderator has been assigned
- Data gathered during this activity includes
    - How much in advance the project lead is sending notification to the moderator
    - How long is the period between notification and the inspection meeting
    - How many inspections required postponement
    - Actual versus planned inspections

Exit Criteria

- The inspection activities have been recorded as Performed on the scheduled dates and Closed within the dates determined at the inspection meeting or rework

Overview

- Provides the inspection participants a background and understanding, when warranted, of the scheduled inspection material
- An overview is not an inspection meeting
- If inspectors are sufficiently familiar with the work product, the overview can be skipped
- Another reason for an overview meeting, is to identify any open issues in the work product
- An open issue is an acknowledgement of the fact that a subpart of the work product is not complete for some reason
- The producer may want focus the inspectors on subparts that are problematic or of some concern

Responsibility

- The producer's primary responsibility for the success of the overview meeting is to deliver the presentation

- If overview material is provided, it is the producer’s responsibility to make sufficient copies for the meeting either directly or via the moderator

Other Roles

- The moderator determines with the project lead whether an overview is necessary, schedules the overview meeting, obtains the meeting room, and records the time of the meeting, the number of participants, and the results of the meeting
- Inspectors participate during the overview meeting and must concur that the overview met the exit criteria

Entry Criteria

- A project lead has sent notification for an inspection
- The inspection requires a mandatory overview, or criteria for an optional overview has been satisfied; e.g.,
    - Complexity of the work product solution
    - Volume of material in the work product
    - Criticality of the work product
    - Customer requirements
- The producer is ready to present the overview
- Open issues and any potential problem areas are highlighted

Tasks

- Producer prepares for the overview using a format and style that will best convey the information to the participants
- Moderator invites the participants to the overview meeting
- Producer presents the overview
- Inspection team members concur that the overview satisfies the needs for preparation and inspection meeting
- Any open issues are documented in the inspection report
- If the overview is used to familiarize the participants with their roles, the inspection process, or some other aspect key to this inspection, the moderator will provide this briefing
- Defects, if any, are documented

Validation/Verification

- The moderator uses the work product overview meeting entry criteria and procedure to determine if a meeting is necessary
- The inspection team is in concurrence with the decision taken to have an overview or not
- The inspectors have the responsibility to state that the overview, when held, is satisfactory for their preparation and subsequent inspection meeting
- The SQA group ensures that the moderator has used the overview meeting criteria and ensures an appropriate decision was made to have an overview or not. This can be done via audits of the process records or sampling of inspections
- Data gathered during this activity
- How much participant time was spent in the overview
- The clock time for the overview
- Time between notification and the overview meeting
- How many overviews required rescheduling
- How many defects were identified at the overview

Exit Criteria

- The overview meeting was determined to be satisfactory by the inspectors and SQA
- Open issues are documented
- Potential problems areas are noted to the participants for preparation and for the reader for the inspection meeting
- Defects, if any, are documented

Preparation

- Allows time for the inspection participants to sufficiently prepare for the inspection meeting and list potential defects
- During preparation the inspectors should:
    - Increase their understanding of the material
    - Inspect the work product using the checklist appropriate to the work product
    - Identify possible defects that will be discussed at the inspection meeting
    - Create a list of minor defects and provide them to the producer
    - Note the amount of time spent in this activity

Responsibility

- Primary responsibility is with the inspectors to ensure they have properly prepared for the inspection meeting
- If an inspector cannot prepare sufficiently, the moderator must be notified immediately and a backup inspector selected
- The inspection meeting may have to be cancelled in those situations if backup inspector is not available
- Decision should be recorded to learn during analysis

Other Roles

- The moderator should first estimate the preparation time needed for the inspection based on the material to be inspected. These estimates should be verified with the inspection team participants
- The moderator needs to get a commitment from each participant that enough time is allocated and that it will be sufficient for him/her to prepare

Entry Criteria

- The overview, if needed, has been satisfactorily completed
- Any open issues identified for the overview have been closed and addressed in the work product or are documented as open issues and provided as ancillary material for the preparation
- Open issues not closed are documented for tracking within the change control system used by the project
- The producer determines that the work product is ready for inspection
- The work product has reached closure and the code complies with defined standards, style guides, and templates for format
- All necessary ancillary material have been made available well in advance
- The work product includes all base-lined function and approved changes for this planned work product completion date
- The amount of time needed for preparation has been confirmed with the inspectors and is available to them
- Predecessor and dependent work products are available, have been inspected, and meet exit criteria

- The moderator and producer have defined the coverage of material to be inspected
- The work products allow easy identification of defects by location in the material
- The moderator agrees that the work product is inspectable

Tasks

- Each inspector uses the scheduled time to complete the preparation in a style and format they are comfortable with
- The material to be inspected is marked with questions, concerns, and possible defects, both major and minor, found during inspection
- The minor defects are either recorded on a separate sheet that will be delivered to the moderator at the start of the inspection meeting or they are clearly noted in the marked material that will be delivered to the moderator at the end of the inspection meeting. Each minor defect should be noted by location in the work product when using a minor list

Validation/Verification

- The moderator uses the preparation entry criteria and procedure
- The moderator uses the minor defect information to determine if all inspectors have properly performed preparation
- The inspectors have confirmed that they have prepared
- The SQA group ensures that the moderator has used the preparation procedure and that the inspectors performed sufficient preparation. This can be done via audits of the process records or sampling of inspections
- Data gathered in this activity
  - How much time was spent in preparation
  - How long a period between notification of the inspection and the preparation
  - How many inspection meetings required rescheduling due to insufficient preparation
  - The number of major and minor defects found during preparation

Exit Criteria

- Each inspector has completed sufficient preparation based on organization and project preparation time criteria
- Minor defect inputs are complete
- Preparation notes are recorded on the work product materials or defect lists
- Identifies defects before work product is passed into the next project stage
- Some discussions are held in the inspection meeting
- The identified defect is agreed to be a defect, or at least a potential defect, by the inspection team, including the producer
- If a discussion item cannot be agreed to be a defect, it should be noted as an open issue to be resolved after the meeting
- The defect can be classified by class, severity, and type
- The defect can be described crisply but sufficiently
- The inspection meeting has schedule and entry requirements. If inspectors are late by ten minutes, postponement should be considered
- The critical inspectors include the moderator, producer, and reader

Inspection Meeting

Responsibility

- The moderator is responsible for managing an effective and efficient meeting

Other Roles

- The producer is responsible for the inspected work product, answering questions, and concurring on identified defects or adequately explaining to the inspection team's agreement why the identified possible defect is not a defect
- The reader is responsible for focusing and pacing the inspection meeting by leading the team through the material
- The recorder is responsible for correctly recording all identified defects and open issues
- All inspectors, including the producer, are responsible for sharing their questions and identified defects found during preparation, and work together to find more defects in meeting

Entry Criteria

- The inspection team members are sufficiently present in number and role assignments
- Inspection materials were available for preparation with sufficient time for study and review before the inspection meeting, including necessary reference material
- Inspectors have adequately prepared
- Inspectors have submitted their minor defects list at the start of the meeting or have marked the work products that will be provided at the end of the meeting
- Scope of the inspection meeting has been defined
- Recorder and a data recording system are available
- Other roles; e.g., reader have been assigned
- The producer has identified any new potential problem areas

Tasks

- Brief introduction (moderator)
- Preparedness check (moderator)
- Read the work product (reader)
- Identify defects (inspectors)
- Record defects (recorder)
- Determine disposition of material (inspection team)
    - Accept the material
    - Accept the material after verification with follow-up inspector
    - Request the work product to be re-inspected after rework
    - Recommend re-engineering of the work product followed by a new inspection

Validation/Verification

- The moderator, using the inspection meeting entry criteria and procedure, determines if the team has properly performed the inspection
- The inspectors participated in an effective meeting
- The SQA group ensures that inspection meeting procedure and that the inspectors performed sufficient preparation. This can be done via audits of the process records or sampling of inspections
- Data gathered during this activity
    - How much time was spent in the inspection meeting
    - How long a period between the preparation and the inspection meeting

    - How many inspection meetings required rescheduling due to insufficient preparation
    - How many inspections required re-inspection
    - How many defects were found
    - How long the meeting took
    - How many inspectors were in attendance

Exit Criteria

- The inspection materials have been inspected and coverage of the work product is completed as planned
- The inspection results fall within expected tolerance of performance for
    - Time spent during preparation
    - Time spent at the inspection meeting
    - Defect density
- The defects and the conduct of the inspection have been recorded and the team concurs with the contents
- Open issues have been recorded for follow-up during rework
- The moderator or a designee has been appointed to perform follow-up with the producer
- Data is available to update the process data base
- Any associated deviations or risks are noted
- Decisions to re-inspect or not have been reviewed against criteria
- Decision on re-engineering has been addressed
- Process defects have been recorded, as appropriate, as well as product defects
- The locations of the defects of the inspected work product are clearly noted to facilitate repair
- A decision is taken on the timeframe by which defect repairs and open issues will be resolved
- The inspection satisfies the criteria to be indicated as performed

Analysis Meeting

- Which is held after the inspection meeting, to begin defect prevention activities
- This activity was not part of the original inspections

Responsibility

- The moderator with the project lead determines whether this activity will be performed

Other Roles

- The producer should be willing to accept open input from the inspection team regarding the potential causes of the identified defects

Entry Criteria

- The project lead and moderator have chosen this activity to be performed
- The inspection team has been trained in causal analysis techniques. Training in team dynamics and group behavior can be helpful
- Major defects have been found during the inspection
- A defect taxonomy or set of cause categories has been defined

Tasks

- Select the defects to discuss
- Determine the potential causes of the defects discussed
- The recorder will record the analysis meeting results and provide them to the inspection coordinator or SEPG as input for process improvement consideration within the organization at the prevention meeting

Validation/Verification

- The moderator uses the analysis meeting entry criteria and procedure to determine if all inspectors have properly participated and the meeting was effective
- The inspectors have participated
    - If they cannot participate, they must notify the moderator at the start of the inspection meeting
- The SQA group ensures that the moderator has used the Analysis meeting checklist and reviews the recorder's report for sufficiency. Audits
- Data gathered during this activity
    - How much time was spent in the analysis meeting
    - How many defects were discussed
    - How many defects were assigned causes

Exit Criteria

- The analysis meeting records have been completed
- Data is provided to the SEPG or inspections coordinator

Rework

- Fixes identified defects and resolves any open issues noted during the inspection
- In some cases, the repair may require a Change request to be written because of the nature or impact of the defect

Open Issues

- The open issue is accepted as a defect and repaired
- The open issue becomes a change request for further investigation
- The open issue was incorrectly identified as potential defect and is closed as a non-defect

Responsibility

- The producer is responsible for all open issues, fixing all defects, and writing any change requests

Other Roles

- The moderator or designee is assigned to discuss open issues with the producer during rework and to come to concurrence with the producer

Entry Criteria

- The list of defects and open issues is provided to the producer for resolution
- The moderator or someone assigned meets with the producer to review rework and open issues
- The inspection report is completed, is on file, and available

Tasks

- The producer repairs accepted defects identified during the inspection meeting
- The producer resolves any open issues
- The moderator meets with the producer to discuss resolutions of open issues
- Change requests are written for any open issues or defects not resolved during the rework activity
- Either the minor defect list or marked work products with minor defects noted are used to repair the minor defects

Validation/Verification

- The follow-up activity is scheduled; where the rework will be verified by the moderator or assigned designee
- SQA has reviewed sample results of this activity in the project
- Data gathered during this activity
  - How much time was spent in rework
  - How many open issues were resolved and accepted as defects
  - How many open issues became submitted change requests

Exit Criteria

- The producer resolves all defects and open issues
- Inspected work product materials are updated to account for repairs

Follow-Up

- Verifies that all defects and open issues have been adequately fixed, resolved, and closed out

Responsibility

- The moderator is the individual primarily responsible for reviewing repairs. The moderator will also review the producer's decisions on repairs and change requests. The moderator may delegate some or all of this responsibility

Other Roles

- The producer is to provide an explanation of the repairs and closures made

Entry Criteria

- Rework of defects has been completed; i.e., fixed or identified with a decision to not fix
- The producer has completed the rework for defects and open issues resolved to be defects
- Change requests are written for any defects or open issues not resolved
- The moderator concurs with the producer's decisions on defects, open issues, and change requests

Tasks

- The moderator and producer discuss and agree on compliance with respect to defects and open issues
- In case of disagreement, the issue would be resolved by the project lead
- The producer updates the work product to reflect the fixes to defects found and open issues accepted as defects
- The producer writes any change requests that may be required
- The moderator completes the inspection report and marks the inspection as closed

Validation/Verification

- The moderator concurs with the defect repairs and open issue closures
- The producer reviews the final inspection report
- SQA group reviews the final inspection report
- Data gathered during this activity
    - How much time was spent in follow-up
    - How many open issues were disputed

Exit Criteria

- Any change requests resulting from unresolved open issues have been submitted to the change approval process for handling
- The inspection report is completed and the producer agrees
- If necessary, a re-inspection is scheduled
- If necessary, issues are escalated to the project lead for resolution
- The inspection is noted as closed

Prevention Meeting

- Which is held periodically after sets of inspections have been performed to determine probable causes for selected defect types, instances, or patterns
- Required data about defects

Responsibility

- The prevention team leader for the prevention meeting will record the results of the meeting and deliver proposals for actions to the organization management

Other Roles

- The members of the prevention meeting team will participate to determine actions for probable causes of selected defect types

Entry Criteria

- An inspection meeting was held
- The analysis meeting was held
- Defect data including causes are available to the prevention meeting team

Tasks

- Record data from the prevention meeting
- Record proposed actions to be taken for defect prevention
- Initial preparation for the proposals to be presented to management for decision

Prevention Meeting: Validation/Verification

- The prevention team has met based on the defined cycles for the meetings
- SQA reviews sampled reports
- The SEPG reviews proposed actions and resultant actions taken
- Data gathered during this activity
    - Time in the prevention meeting
    - Effort invested at the meeting
    - Number of proposals brought forward to management
    - Number of actions taken from proposals

Exit Criteria
- The data is complete and agreed to by the prevention meeting participants

Data Recording and Reports
- To record the data about the defects and conduct of the inspection
- This activity is held concurrently with other activities, including at the end of the inspection process

Responsibility
- The recorder during the overview, inspection meeting, and optional analysis meeting records data about the defects and the conduct of the inspection
- Alternatively the moderator can enter the data

Other Roles
- The moderator after the overview and during the follow-up activity ensures that the data has been entered correctly and completely

Entry Criteria
- The optional overview meeting was held
- The inspection meeting was held
- The optional analysis meeting was held

Tasks
- Record data from overview, if held
- Record data at the inspection meeting, including preparation data
- Record data at the optional analysis meeting
- Record data during the follow-up activity, including sign-off to close the inspection

Validation/Verification
- The inspection verifies the data at the end of the inspection meeting and optional analysis meeting
- SQA review sampled reports
- The producer reviews the report completed by the moderator
- Data should be considered for this activity; e.g., how much effort is used for recording and reporting

Data Recording and Reports: Exit Criteria
- The data are complete and agreed to by the inspection meeting and analysis
- meeting participants
- The data gathered during the follow-up activity are complete and agreed to by the
- producer and moderator

Inspection Process Monitoring
- This activity is held concurrently with other activities and after inspections
- The purpose is to evaluate the results of the inspection process as performed in the organization and to propose suggestions for improvement

Responsibility
- The inspection process coordinator or SEPG is responsible for monitoring and suggesting improvements

Other Roles

- Management ensures that inspection process monitoring is integrated into the inspection process
- The inspection process improvement team proposes actions for inspection process improvements based on the monitoring and analysis of the inspection coordinator

Entry Criteria

- Reports and results from inspections over a period of performance are available
- A coordinator is assigned
- Resources are allocated for inspection process improvement team

Tasks

- Gather the inspection process data provided since the last monitoring report
- Review inspection reports and related data for trends and results against objectives
- Interview inspection participants to ensure understanding of results and to gather other inputs
- Perform analysis using data from the inspection reports, interviews, and surveys
- Provide the analysis to the inspection process improvement team for review and proposal to management for inspection process management

Validation/Verification

- The inspection coordinator performs monitoring actions per agreed periods for analysis
- The inspection process action improvement team meets per agreed periods for recommendations
- SQA reviews monitoring activity on a random basis to ensure it is being performed
- Data gathered during this activity
    - How much effort is expended
    - How many proposals for improvement are made
    - How many improvements are put into action

Exit Criteria

- Reports of analysis are developed
- Actions for improvements are proposed
- Actions are implemented for inspection process improvements

Moderator

- The moderator is a key role in successful inspections
- He/she ensures that the inspection procedures are followed, that the inspectors fulfill their responsibilities within their defined roles, and that the inspection is performed successfully
- A moderator must be trained in the process, principles, and practices of software inspections
- It is not a full-time job, but a part-time assignment, mostly given to senior and experienced programmers, designers, analysts, and writers who have active roles in a project
- They should be recognized for the extra time it takes to moderate, whether on their own project or other projects
- The moderator has to play many other tasks including working as a coordinator, facilitator, coach, mediator, manager

- He/she is not a representative of the management
- Moderator best serve when they are objective and do not have a vested interest in the work product
- It should be rare to have such a situation Moderators best serve when they have technical or domain knowledge of the work product under inspection
- The moderator should pace the inspection meeting to ensure the participants are not overtaxed, working too long without breaks
- Inspection meetings should not be scheduled for more than two hours
- Moderators must also help in finding defects effectively and efficient

Qualities of Good Moderators

- Independent and objective
- Leader
- Coach
- Technically astute
- Communication skills
- Trained

Independent and Objective

- Moderator should not be the part of the team that worked on the work product under inspection
- Sometimes, this cannot be avoided

Leader

- Moderators serve best when they have management and leadership abilities
- They will manage the inspection once it has been scheduled
- Some organizations have viewed how well a moderator leads as an indication of management ability on future projects

Coach

- Good manager/leaders are also good coaches also

Technically Astute

- The moderator does not have to be an expert in the domain of the work product, but the moderator should be able to understand the technical aspects
- When the moderator is not technically knowledgeable, the team may discount them and they are less able to control the technical discussions

Communication Skills

- The moderator must listen and hear; the moderator must give directions and explain so the participants understand the value of inspections

Trained

- The moderator must be trained
- Never never have someone serve as a moderator who has not been trained in inspections and the requirements of a moderator
- A moderator should have sense of humor, because that helps when situation gets tense during inspection meetings

Problems with Moderators

- Is aggressive
- Cannot control the meeting
- Moderator is treated as a secretary
- Biased moderator

Activities to be performed by the Moderator

- Inspection scheduling
- Overview
- Preparation
- Inspection meeting
- Data recording
- Analysis meeting
- Rework
- Follow-up

Inspection Scheduling

- Determine the need for an overview
- Determine the inspection team
- Remember that the primary purpose of an inspection is to find the maximum number of defects that may exist in the work product, so pick team members who have the best knowledge and skill to help find defects
- Ensuring availability of materials
- Assigning roles
- Chunking the materials
    - In situations where multiple meetings are required, split the work product to be inspected into reasonable chunks. This can be done in two ways:
        - ✓ By form
        - ✓ By function
    - Function chunking, when it is obvious, is easier to do
    - Form chunking is not so obvious, we seek different points of view within documents
        - ✓ Standards
        - ✓ Code versus other documentation
        - ✓ Efficiency
        - ✓ User interfaces
        - ✓ Maintainability
        - ✓ Operating convenience
- Defining the inspection activities schedule
    - Overview, when required
    - Preparation effort
    - Inspection meeting duration
    - Analysis meeting
    - Logistics

Overview

- Introducing the producer and material for the overview
- Guiding, facilitating, and managing the meeting
- Ensure identified defects that were discovered at the overview are recorded
- Ensure any open issues are recorded
- Concluding the meeting and asking the participants if the meeting met the objectives
- Overview is conducted by the producer

Preparation

- The moderator as inspector prepares for the inspection meeting just as any other inspection participant

Inspection Meeting

- The moderator has two roles during all inspection meetings
    - Moderator
    - Inspector

- The moderator must always maintain objectivity when serving as an inspector, and there is ample evidence that it can be done
- If moderators hold a mini lessons-learned session at the end of each inspection meeting and ask these questions
    - What worked well
    - What could have been improved
- The inspections process can be improved for future inspection meetings

Data Recording

- The moderator must review the defect report created by the recorder and then complete this report during the follow-up activity for the required contents of the inspection report

Analysis Meeting

- The moderator is both a facilitator and participant in this meeting, in which a causal analysis is done on the identified defects

Rework

- The moderator works with the producer during the rework activity to address any open issues or to help in defect classification

Follow-Up

- Complete the defect report as provided by the recorder to show that the inspection is closed
- Verify all rework (defects and open issues)
- Schedule a re-inspection, if warranted

Code of Conduct for Moderators

- Always remain professional and objective
- Prepare well in advance for all meetings
- Enable the team members for a successful inspection
- Keep each meeting focused to its specific objectives; e.g.,
    - Learning at the overview
    - Finding and agreeing to defects at the inspection meeting
    - Performing causal analysis at the analysis meeting
- Ensure all data is captured and recorded
- Always maintain confidentiality
- Use effective meeting practices: e.g.,
    - Properly notify all participants well in advance
    - Restate the purpose of the meeting, especially for first time participants
    - Monitor time and keep the meeting moving forward

  - Allow discussions that help meet the objectives
  - Solicit input at the end of the meeting
- Be a team player; participate as another inspector
- Remember that the moderator is accountable for the quality of the inspection
- Ensure appropriate behavior by all attendees
- Enforce and adhere to inspection entry and exit criteria
- Get the consent of the participants to continue the meeting, if it is clear that the inspection meeting will take longer than the scheduled time

Other Roles & Producer

- The individual who produced or modified the work product to be inspected
- Also known as author
- Producer should be the person who will make changes to the work product as a result of the inspection
- Producer participates in planning, overview, preparation, inspection meeting, rework, and follow-up for an inspection process

Possible Problems with Producers

- Is defensive
- Does not participate
- Was not the producer
- Responds in a hostile manner to identified defects
- Begins to make repairs at the meeting
- Biased producer
- Unprepared

Producer Types to Watch For

- The Gamesman
- The Controller
- The Intimidator
- The Debater
- The Unbeliever
- The Elitist

Reader

- The reader is the inspector who will lead the inspection team through the material during the inspection meeting. The purpose of reading is to focus on the inspection material and to ensure an orderly flow for the inspectors
- Reader participates in preparation and inspection meeting during the inspection process

Ways to Read

- Verbatim
- Paraphrase
- Mixed styles
- section-by-section enumeration
- Perspective-based
- Not read

Possible Problems with Readers
- Reads too fast for the team
- Reads as if the material is right
- The reader is not used

Recorder
- The recorder is the inspector who will record the data for defects found and data about the conduct of the inspection
- Recorder participates in the preparation and inspection meeting activities during the inspection process

Possible Problems with Recorders
- Records too slowly
- Interprets the defect or records incorrectly
- Records something not understandable
- Does not record

Inspector
- All participants are trained to be inspectors
- An inspector participates in preparation, inspection meeting, and analysis meeting activities during the inspection process

Possible Problems with Inspectors
- Is not prepared
- Does not actively participate
- Comes late to meetings
- Not focused

Criteria for Selecting Inspectors
- Domain knowledge in the work product under inspection
- Experience and expertise
- Language knowledge
- Assignment of inspector with work product
- Time availability
- Trained in inspections
- Team player

Timing Issues
- Preparation and inspection meetings
- Reasonable length
- Subsets
- Enforcing time limits
- Breaks during inspections
- Scheduled times

When is an Inspection Finished?

Planned
- Includes any time during the project's life cycle where the schedule has been defined for a required inspection; it concludes with the inspection meeting start

Performed

- Includes all times from the start of the inspection meeting through rework

Closed

- Is only after follow-up when closure has been achieved and signed-off

Best Place to Start First

- Requirements specifications
- Design
- Code
- User documentation

Can Some Work Products Not be Inspected?

- Yes, but the decision requires data that demonstrates minimal risk and good data requires time in practice
- For safety-critical or life-critical software, you should not take the risk lightly, if at all

Who are the Right Inspectors?

- A domain expert will be far more effective in finding defects than a novice. Experts, however, are not always available when we want then
- Less capable inspectors may only be able to find a certain class of defects, while experts can find deeper defects, but they all can contribute to finding defects
- The decision should be based on risk and criticality of the work product. Here criticality is not just safety-critical or life-critical situations, but work products critical to the success of the project
- Re-inspection may have to be done when experts are available, in case they were not available for the inspection meeting

Inspections Don't Make People Warm and Fuzzies

- Inspections are rarely the most exciting task for programmers, but they are necessary and useful. They should be made as comfortable as possible

Helping Programmers to Learn from Their Errors

- If we allow the programmers to learn in a safe environment, they generally will learn
- Not all programmers are equal in capability, but all can contribute to the project's success
- We may have to provide additional training to some
- As programmers learn, they will become more effective, and this will show in their work products
- The will take pride in their work and work environment

Competence Paying for Incompetence

Small Teams

- Inspections can be applied in small teams
- Such small teams do not have the formality of inspections, as we have discussed so far
- However, when another set of eyes look at the work products, defects can be identified quickly

Do We Really Need All that Data with Inspections?

- In the beginning, data may not be collected to get familiar with inspections
- However, in order to have highly effective inspections, you'll need the data and analysis on that data

Get Off to a Good Start

- Not all people will accept the use of inspections initially
- Data and experience should win some of the doubters over, but holding the hands of some will be necessary
- Learn to understand the concerns of the people who are being asked to use inspections and you will be more successful
- The doubters will ask questions like:
    - Is the overhead worth it?
    - Does management really support this?
    - How will the data be used?
- These doubts will not be limited to practitioners. Management may have even deeper doubts and issues
- This is the time the inspection coordinator or SEPG must stay the course
- Responses must be given to hold the doubters' focus, and eventually the data should win them over too
- Today it is far easier with the volume of inspection data and experiences rerported in the literature

Inspecting Changes

- Some argue that when a change is made to a previously tested work product that it is cheaper and less labor intensive to retest rather than inspect or re-inspect
- This argument is false for three reasons
    - If a fix or change is made and previous test cases did not flush out the defect, then rerunning them does not prove the changes work. It only proves the change did not regress the work product. This is not verification of the change or fix! The test cases need to be changed or modified before rerunning. Otherwise they would have found the defect
    - When a re-inspection is performed it is not required on the whole work product. The change plus surrounding code may be sufficient to verify the change or fix
    - Changes and fixes have defects in them. In fact, there is data that suggests that small changes are more defective than large changes. Therefore, there is a good likelihood that a defect will be found in test, and now we are back to the very reason why inspections are better; it is cost effective to find the defect before test versus test
- Inspections are looking at change in the context of the work product and relationships it has with other work products. This means that inspectors should ask what else could be affected by the change and try to ensure that the change does not ripple into another defect
- No one ever claimed that tests would go away because of inspections; it just will be less costly. Therefore, we should expect that some regression testing should occur even after inspections are finished
- Test will validate that the product was not regressed by the fix; inspections will verify that the fix is correct

"Yes, Of Course, We Do Inspections!"

- It sometimes happens that the checkmark of having done an inspection is tracked with a higher level of importance than performing an effective inspection
- For example, inspections are indicated as having been completed regardless of when they might have occurred
- This results in ineffective inspections and an organization attitude that inspections are a bureaucratic quality approach
- Situations where this checkmark misuse is seen
  - Unit test is performed before the code inspection, and in some cases the inspection becomes a non-event
  - Resources are not available for a scheduled inspection, so the work continues without an inspection and then sometime later a minimal inspection in name is held, e.g., a requirement inspection is held after or while the design is being complete
  - Some people believe an inspection will go faster when bugs have already been taken out via test or some other approach
- One wonders why inspections are held in the first place
- Satisfy the management's decision without buying into them. People suffer
- If You Find a Defect, Fix It
- Inspections find defects but producer do not fix them correctly
- Most common reasons for this sort of unprofessional behavior are:
  - The inspection data was lost
  - The author didn't understand the inspection data
  - There wasn't time in the schedule

Bureaucratic Data Collection

- Not all people will see value in collecting data on inspections

Inspections are too Formalized

- Formality is not itself a problem
- The question is whether the process is repeatedly producing positive business results
- If so, formality is justified
- If the process is onerous, inefficient, or ineffective then something should be done

Inspections Waste Time

Redundancy with the Test Processes

- Inspections are intended to remove defects as early as possible. They are quality control mechanism
- Testing is intended to
  - Find the defects that leaked through the inspection process
  - Revalidate that the delivered solution satisfies the needs of the customer
- Tests are both quality control and quality assurance mechanism
- With analysis of data from the inspection and test results, both processes can be tuned to maximize efficiency and minimize redundancy, while maintaining effectiveness

Arbitrary Style Viewpoints during Inspections

- If an organization has not defined or agreed to an accepted style for specifications, design, code, or for other documents, then it is possible that inspection participants may have different viewpoints. This can lead to useless discussions
- Agreements on these issues is a prerequisite for effective and efficient inspections

Less Efficient than Debugging during Execution

- Some people believe that debugging within a test execution environment is faster and cheaper. The literature consistently has suggested otherwise
- Try a controlled study
- There may be languages where execution of the code during the inspection would be more beneficial. For example, the visual type of languages may be best inspected in permutation of the traditional inspection using both the code and viewing the performance of the code by looking at the screens during the execution

Design Knowledge is Required to do Code Inspections

- Absolutely, but they do not have to have as much domain knowledge as the producer

Know code and design sufficiently

- Design and specification documents can be helpful in understanding

Only Superficial Errors are found

- There is a relationship between effectiveness and the domain knowledge of the inspectors

Inspecting too much

- It is possible
- Performance Characteristics are Not Addressed
- If it is true, then these may not have been the focus of inspections
- The "Not the Way I Would have Done It" Attitude
- This issue must be controlled
  - Proper and consistent training across the inspection team members
  - Documented methods, styles, and notations to be used by the producer in creating the artifact

Can We Do Less Rigorous Inspections?

- Can be done with forethought that this is a good business decision from an effectiveness perspective (i.e., very low defects)
- The informal approach is noted as such and tracked as a potential risk
- Review is made after test and customer use to learn if the decision was good

Customers Accept "Good Enough" Software Solutions

- Yes, they do. However, do not accept defect in their software products

Many Successful Companies do Not Use Inspections

- How much more successful could they have been?
- What was the cost of success?
- What is the definition of success?

Modified Code is too Costly to Inspect

- There is need to inspect the additions, deletions, and modifications plus some "surround code" to understand the context of the additions, deletions, and modifications

Some Inspectors Don't Contribute

- This will happen and can be for a number of reasons

We Tried Inspections and They Didn't Work

- What does the data show?
- Did they really use inspections?
- What did management did to show commitment?
- Did management keep the data safe?
- Was the project not ready for inspections?
- Were inspections treated as a silver bullet in a chaotic environment?
- Were people trained?
- Was the process followed?

Deadly Sins of Inspections

- Superficial commitment from management
- Not enough time in the schedule
- Resources are not allocated
- Insufficient preparation
- Wrong people assigned
- Not using the data
- Treating inspections as a rubber stamp
- Using the checklist without thinking beyond it
- Not training the inspectors
- No entry/exit criteria
- Wrong pace
- Believing all review types are the same

Software Testing

Basics of Software Testing

- Synonyms: checking, assessing, examination
- Testing is one of the most important parts of quality assurance and the most performed QA activity
- Testing has been a basic form of defect removal since software industry began
- For majority of software projects, it is the *only* form of defect removal utilized
- 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 faults that caused the failure
- Otherwise, we gain some confidence that the software under testing is more likely to fulfill its designated functions

Why Software Testing?

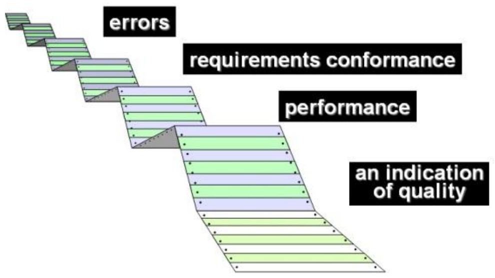
- The purpose of software testing is to ensure that the software systems would work as expected when they are used by their target customers and users
- For software products, software testing provides a mechanism to demonstrate their operation through controlled execution

Software Testing

- "The dynamic execution of software and the comparison of results of that execution against a set of known, pre-determined criteria"
- This demonstration of proper behavior is a primary purpose of software testing, which can also be interpreted as providing evidence of quality in the context of software quality assurance, or as meeting certain quality goals
- In comparison with the manufacturing systems, problems in software products can be corrected much more easily within the development process to remove as many defects from the product as possible
- Therefore, software testing has two primary purposes

Primary Purposes of Software Testing

- To demonstrate quality or proper behavior
- To detect and fix problems
- What Testing Shows?



Major Activities and Generic Testing Process

- Test planning and preparation
- Test execution and related activities
- Analysis and follow-up

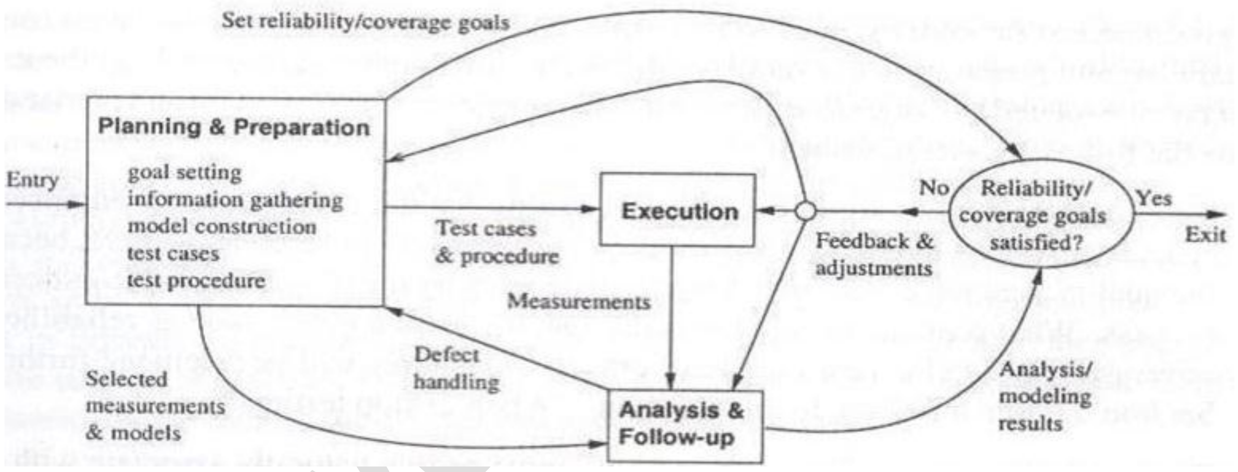
Test Execution

- Execution of software tests
- Observation and measurement of product behavior

Analysis and Follow-Up

- Includes 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, of faults, that led to the observed failures in the first place

Generic Testing Process



- Due to 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 overlook in such informal testing
- Therefore, there is a strong need for planned, monitored, managed and optimized testing strategies based on systematic considerations for quality, formal models, and related techniques
- Test planning and preparation has these sub-activities

Sub-Activities in Test Planning and Preparation

- Goal setting
  - Reliability and coverage goals
- Test case preparation
  - Constructing new test cases or generating automatically, selecting them from existing ones for legacy products, organizing them in some systematic manner

- Test procedure preparation
  - It is defined and followed to ensure effective test execution, problem handling and resolution, and overall test process management

Testing as a Part of QA in Overall Software Process

- Testing and inspections often find different kinds of problems, and may be more effective under different circumstances
- Therefore, inspections and testing should be viewed more as complementary QA alternatives instead of competing ones
- Testing is an integral part of different software development processes

Testing and Defect Removal Efficiency

- Most forms of testing are less than 30% efficient in finding bugs
- Many forms of testing should be used in combination with pre-test design and code inspections to achieve high defect removal efficiencies

Questions about Testing

Basic, testing techniques, testing activities and management

Basic Questions about Testing

- Our basic questions about testing are related to the objects being tested, perspectives and views used in testing, and overall management and organization of test activities
- What artifacts are tested?
- What to test, and what kind of faults is found?
- When, or at what defect level, to stop testing?
  - Coverage information, product reliability goals,

Questions about Testing Techniques

- Many different testing techniques can be applied to perform testing in different sub-phases, for different types of products, and under different environments
- Various questions regarding these testing techniques can help us get a better understanding of many related issues
- Are techniques for testing in other domains applicable to software testing?
  - This defect seeding, mutation, immunization used in physical domains and comes under specialized testing in software
- If multiple testing techniques are available, can they be combined or integrated for better effectiveness or efficiency?

Questions about Test Activities and Management

- Various other questions can also be used to help us analyze and classify different test activities
- Some key questions are about initiators and participants of these activities, organization and management of specific activities
- (Eight questions)

- Who performs specific activities?
- When can specific test activities be performed?
- What process is followed for these test activities?
- Is test automation possible? And if so what kind of automated testing tools are available and usable for specific applications?
- What artifacts are used to manage the testing process and related activities?
- What is the relationship between testing and various defect-related concepts?
- What is the general hardware /software / organizational environment for testing?
- What is the product type or market segment for product under testing?
- Software test planners and testers need to answer those question which are relevant for their project
- In the series of lectures on software testing, as a mechanism for quality control and quality assurance, we will also address them

Testing Objectives

- Testing is a process of executing a program with the intent of finding an error
- A good test case is one that has a high probability of finding an as-yet-undiscovered error
- A successful test is one that uncovers an as- yet-undiscovered error
- These objectives imply a dramatic change in viewpoint. They move counter to the commonly held view that a successful test is one in which no errors are found. Our objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort
- If testing is conducted successfully, it will uncover errors in the software, and as a secondary benefit, testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met
- Data collected as a result of testing can provide a good indication of software reliability and some indication of the software quality as a whole
- Testing cannot show absence of errors and defects, it can show only that software errors and defects are present
- Let us now discuss the basic principles that guide software testing

Testing Principles

- All tests should be traceable to customer requirements
    - The objective of software testing is to find defects. It follows that the most severe defects are those that cause systems to fail to meet their requirements
- Tests should be planned long before testing begins
    - Test planning can begin as soon as the requirements model is complete
- The Pareto (80-20) principle applies to software testing
    - 80% of all defects found will likely be traced to 20% of modules. These error-prone modules should be isolated and tested thoroughly
- The testing should begin “in the small” and progress toward testing “in the large”
    - The first tests planned and executed focus on individual components. As testing

progresses, focus shifts to integrated clusters of components

- Exhaustive testing is not possible
    - The number of path permutations for even a moderately sized program is exceptionally large. It is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised
- To be most effective, testing should be conducted by an independent third party
    - There are many testing techniques, as we will discuss them, and the software engineer who created the software system is not the best person to conducts all tests for the software. For certain kinds of testing, the best results are obtained by conducting these tests by professional testers
- We can say that software testing is a destructive process (psychologically speaking) and not a constructive process, because a software test will result in the breaking of software under test
- Programming on the other hand is a constructive process
- This change in thinking can help test planners and test designers to create effective test cases
- This concept also helps software designers and programmers to design and code software systems, which are testable. So, what is this testability?

Testability

- Software testability is simply how easily [a computer program] can be tested
- Since testing is so difficult, it pays to know what can be done to streamline it
- It occurs as a result of good design. Data design, architecture, interfaces, and component-level detail can either facilitate testing or make it difficult
- There are certain metrics that could be used to measure testability of a software product
- We can also call them the characteristics or attributes of testable software. These characteristics or attributes are important from quality assurance's point of view

Characteristics of Testable Software

- Operability
- Observability
- Controllability
- Decomposability
- Simplicity
- Stability
- Understandability

Operability

- "The better it works, the more efficiently it can be tested"
- The system has few bugs (bugs add analysis and reporting overhead to the test process)
- No bugs block the execution of tests
- The product evolves in functional stages (allows simultaneous development and testing)
- "What you see is what you test"
- Distinct output is generated for each input
- System states and variables are visible or queriable (e.g., transaction logs)

- All factors affecting the output are visible
- Incorrect output is easily identified
- Source code is accessible

Controllability

- "The better we can control the software, the more testing can be automated and optimized"
- All possible outputs can be generated through some combination of input
- All code is executable through some combination of input
- Software and hardware states and variable can be controlled directly by the test engineer
- Input and output formats are consistent and structured
- Tests can be conveniently specified, automated, and reproduced

Decomposability

- "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting"
- The software system is built from independent modules
- Software modules can be tested independently

Simplicity

- "The less there is to test, the more quickly we can test it"
- Functional simplicity (e.g., the feature set is the minimum necessary to meet requirements)
- Structural simplicity (e.g., architecture is modularized to limit the propagation of faults)
- Code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance)

Stability

- "The fewer the changes, the fewer the disruptions to testing"
- Changes to the software are infrequent
- Changes to the software are controlled
- Changes to the software do not invalidate existing tests
- The software recovers well from failures

Understandability

- "The more information we have, the smarter we will test"
- The design is well understood
- Dependencies between internal, external, and shared components is well understood
- Changes to the design are communicated
- Technical documentation is instantly accessible
- Technical documentation is well organized
- Technical documentation is specified and detailed
- Technical documentation is accurate

Discussion of Testability

- The characteristics we have discussed just are fundamental for software testers to efficiently and effectively perform software testing
- The all belong to the different developmental stages of software engineering process
- These characteristics clearly indicate that in order to have good software testing, best practices should be used in all activities of software engineering life cycle to develop software product, otherwise, we will end up with a product which is difficult to test

Attributes of a Good Test

- A good test has a high probability of finding an error
- A good test is not redundant
- A good test should be "best of breed"
- A good test should be neither too simple nor too complex

A good test has a high probability of finding an error

- The tester must understand the software and attempt to develop a mental picture of how the software might fail
- Ideally, the classes of failure are probed and a set of tests should be designed to show that software fails in a particular situation

A good test is not redundant

- Testing time and resources are limited and there is no point in conducting a test that has the same purpose as another test
- Every test should have a different purpose, even if it subtly different

A good test should be 'best of breed'

- In a group of tests that have a similar intent, time and resource limitations may force us to execute only a subset of these tests. In such cases, the tests that has the highest likelihood of uncovering a whole class of errors should be used

A good test should be neither too simple nor too complex

- Ideally, each test should be executed separately, instead of being combined with many tests
- If tests are combined into one test, this can result in masking of errors

Test Case Design

- The design of tests for software and other engineered products can be as challenging as the initial design of the product itself
- However, many software engineers treat testing as an afterthought, developing test cases that may "feel right" but have little assurance of being complete
- A rich variety of test case design methods have evolved for software, which provide the developer with a systematic approach to testing
- More important, methods provide a mechanism that can help to ensure the completeness of tests and provide the highest likelihood for uncovering errors in

software

- Any engineered product can be tested in one of two ways
  - Knowing the specified function that a product has been designed to perform
  - Knowing the internal workings of a product
- In the first case, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function
- In the second case, tests can be conducted to ensure that internal operations are performed according to the specifications and all internal components have been adequately exercised

Broad Categories of Testing

- General testing
- Specialized testing
- Testing that involves users or clients

General Forms of Testing

- Concerned with almost any kind of software and seek to eliminate common kinds of bugs such as branching errors, looping errors, incorrect outputs, and the like

Examples of General Forms of Testing

- Subroutine testing
- Unit testing
- System testing of full application
- New function testing
- Regression testing
- Integration testing

Specialized Forms of Testing

- More narrow in focus and seek specific kinds of errors such as problems that only occur under full load, or problems that might slow down performance

Examples of Specialized Forms of Testing

- Viral protection testing
- Stress or capacity testing
- Performance testing
- Security testing
- Platform testing
- Year 2000 testing was also an example of this
- Independent testing

Forms of Testing Involving Users

- The forms of testing involving users are aimed at usability problems and ensuring that all requirements have been in fact implemented

Examples of Forms of Testing Involving Users

- Customer acceptance testing
- Field (Beta) testing
- Usability testing
- Lab testing
- Clean-room statistical testing

Test Case Design

- Any engineered product can be tested in one of two ways
  - Knowing the specified function that a product has been designed to perform
  - Knowing the internal workings of a product
- In the first case, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function
- In the second case, tests can be conducted to ensure that internal operations are performed according to the specifications and all internal components have been adequately exercised
- In the first case, testing is focused on the external behavior of a software system or its various components, and we cannot see inside the components
- While in the second case, testing is focused on the internal implementation, and we must see inside the component

Testing Techniques

- Black-box testing (BBT)
  - aka functional/behavioral testing
- White-box testing (WBT)
  - aka structural/glass-box testing

Black-Box Testing

- Black-box testing alludes to tests that are conducted at the software interface
- Although they are designed to uncover errors, they are also used to demonstrate that software functions are operational, that input is properly accepted and output is correctly produced, and that the integrity of external information is maintained
- A block-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software
- The inner structure or control flow of the application is not known or viewed as irrelevant for constructing test cases. The application is tested against external specifications and/or requirements in order to ensure that a specific set of input parameters will in fact yield the correct set of output values
- It is useful for ensuring that the software more or less is in concordance with the written specifications and written requirements
- The simplest form of BBT is to start running the software and make observations in the hope that it is easy to distinguish between expected and unexpected behavior
- This is ad-hoc testing and it is easy to identify some unexpected behavior, like system crash

- With repeated executions, we can determine the cause to be related to software and then pass that information to the people responsible for repairs

Black-Box Testing Approaches

- System testing of full application
- New function testing
- Lab testing
- Usability testing
- Customer acceptance testing
- Field (Beta) testing
- Clean-room statistical testing

White-Box Testing

- White-box testing of software is predicated on close examination of procedural detail
- Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops
- The "status of the programs" may be examined at various points to determine if the expected/asserted status corresponds to the actual status
- The test developer is privy to inner structure of the application and knows the control flow through the application, or at least knows the control if the software works correctly
- It is useful for ensuring that all or at least most paths through the applications have been executed in the course of testing
- Using white-box testing methods, software engineers can derive test cases that
    - Guarantee that all independent paths within a module have been exercised at least once
    - Exercise all logical decisions on their true and false sides
    - Execute all loops at their boundaries and within their operational bounds
    - Exercise internal data structures to ensure their validity
- The simplest form of WBT is statement coverage testing through the use of various debugging tools, or debuggers, which help us in tracing through program executions
- By doing so, the tester can see if a specific statement has been executed, and if the result or behavior is expected
- One of the advantages is that once a problem is detected, it is also located
- However, problems of omission or design problems cannot be easily detected through white-box testing, because only what is present in the code is tested
- Another important point is that the tester needs to be very familiar with the code under testing to trace through it executions
- Therefore, typically white-box testing is performed by the programmers themselves
- We'll have some discussion on this topic, that is who is most productive for which kind of testing at a later stage in this course

White-Box Testing Approaches

- Subroutine testing
- Unit testing
- Viral protection testing
- Stress or capacity testing
- Performance testing
- Security testing
- Year 2000 testing

Mixed Testing Approaches

- Independent testing
- Regression testing
- Integration testing
- Platform testing

Comparing BBT with WBT

Comparing BBT with WBT with Perspective

- BBT view the objects of testing as a black-box while focusing on testing the input-output relations or external functional behavior; while WBT views the objects as a glass-box where internal implementation details are visible and tested
- Although the objects tested may overlap occasionally, WBT is generally used to test small objects, such as small software products or small units of large software products; while BBT is generally more suitable for large software systems or substantial parts of them as a whole
- WBT is used more in early sub-phases of testing for large software systems, such as unit and component testing; while BBT is used more in late sub-phases, such as system and acceptance testing

Comparing BBT with WBT with Defect Focus

- In BBT, failures related to specific external functions can be observed, leading to corresponding faults being detected and removed. The emphasis is on reducing the chances if encountering functional problems by tagret customers
- In the WBT, failures related to internal implementation can be observed, leading to corresponding faults being detected and removed directly. The emphasis is on reducing internal faults so that there is less chance for failures later on no matter what kind of application environment the software is subjected to

Comparing BBT with WBT with Defect Detection & Fixing

- Defects detected through WBT are easier to fix than those through BBT because of the direct connection between the observed failures and program units and implementation details in WBT. However, WBT may miss certain type of defects, such as omission and design problems, which could be detected by BBT
- In general, BBT is effective in detecting and fixing problems of interfaces and interactions, while WBT is effective for problems localized within a small unit

Comparing BBT with WBT with Techniques

- Various techniques can be used to build models and generate test cases to perform systematic BBT, and others can be used for WBT, with some of the same techniques being able to be used for both WBT and BBT. A specific technique is a BBT one if external functions are modeled; while the same technique can be a WBT one if internal implementations are modeled

Comparing BBT with WBT with Tester

- BBT is typically performed by dedicated professional testers, and could also be performed by third-party personnel in a setting of IV&V (independent verification and validation); while WBT is often performed by developers themselves

As you know, white-box testing deals with that form of software testing, where code or programs are available for review, understanding, and flow of control and data are visible. Let's first ask a question about white-box testing

White-box Testing Question

- Why spend time and energy worrying about (and testing) logical details when we might better expend effort ensuring requirements have been met? or
- Why don't we spend all of our energy on black-box tests?

Answers to that Question

- Logical errors and incorrect assumptions are inversely proportional to probability that a program path will be executed
- We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis
- Typographical errors are random, it's likely that untested paths will contain some
- White-box testing uses the control structure of the procedural design to derive test cases

WBT Methods Derive Test Cases

- Guarantee that all independent paths within a module have been exercised at least once
- Exercise all logical decisions on their true and false sides
- Execute all loops at their boundaries and within their operational bounds
- Exercise internal data structures to ensure their validity

White-Box Testing Techniques

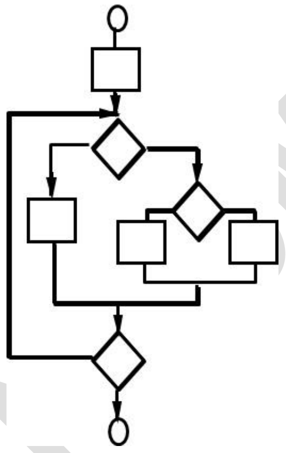
- Basis path testing
- Condition testing
- Data flow testing
- Loop testing

Basis Path Testing

- Basis path testing is a white-box testing technique first proposed by Tom McCabe
- It enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths
- Test cases derived to exercise basis set are guaranteed to execute every statement in the program at least once
- Basis path testing uses cyclomatic complexity, which is a software metric that provides a quantitative measure of the logical complexity of a program
- When the cyclomatic complexity used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once

- An independent path is any path through the program that introduces at least one new set of processing statement or a new condition
- The cyclomatic complexity can be calculated in a number of ways

<u>Cyclomatic Complexity</u>
Number of simple decisions + 1
or
Number of enclosed areas + 1
In this case, V(G) = 4

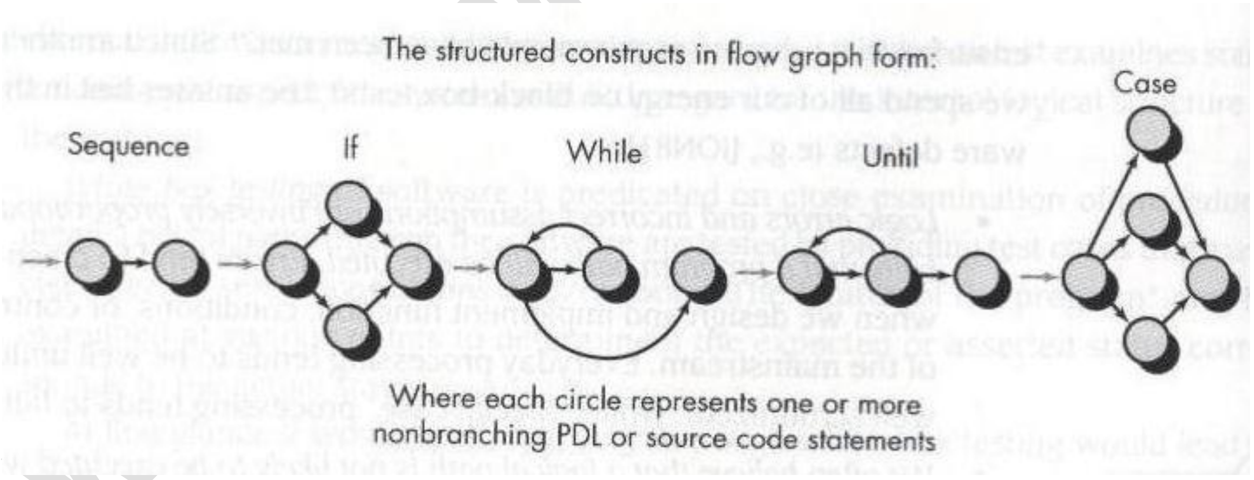


Cyclomatic complexity can also be calculated by developing flow graph (or program graph), which depicts the logical control flow

A flow graph primarily consist of edges and nodes

- A flow graph depicts logical control flow of a program, and contains notation for
  - Sequences
  - If conditions
  - While conditions
  - Until statements
  - Case statements

Flow Graph Notation


The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

- If we are using flow graph graphs, the cyclomatic complexity is calculated as
  - V(G) = Edges – Nodes + 2

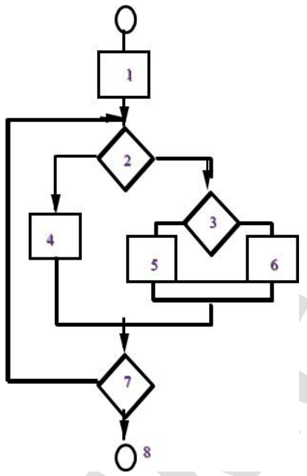
Steps in Basis Path Testing to Derive Test Cases
- Using the design or code as a foundation, draw a corresponding flow graph
- Determine the cyclomatic complexity of the resultant flow graph

- Determine a basis set of linearly independent paths
- Prepare test cases that will force execution of each path in the basis set

Finding Independent Paths

Since V(G) = 4,
there are four paths

Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8
Path 3: 1,2,4,7,8
Path 4: 1,2,4,7,2,4,...7,8
Finally, we derive test cases to exercise these paths.



Condition Testing

- Condition testing is a test case design method that exercises the logical conditions contained in a program module
- Conditions can be
  - Simple conditions
  - Compound conditions
  - Relational expressions
- Errors are much more common in the neighborhood of logical conditions than they are in the locus of sequential procession statements
- The condition testing method focuses on testing each condition in the program
- If a condition is incorrect, then at least one component of the condition is incorrect, therefore, types of errors in a condition include the following
  - Boolean operator error
  - Relational operator error
  - Boolean variable error
  - Arithmetic expression error
- The purpose is to detect errors in the conditions of a program but other errors also
- There are two advantages of condition testing
  - Measurement of test coverage of a condition is simple
  - Test coverage of conditions in a program provides guidance for the generation of additional tests for the program

Condition Testing Strategies

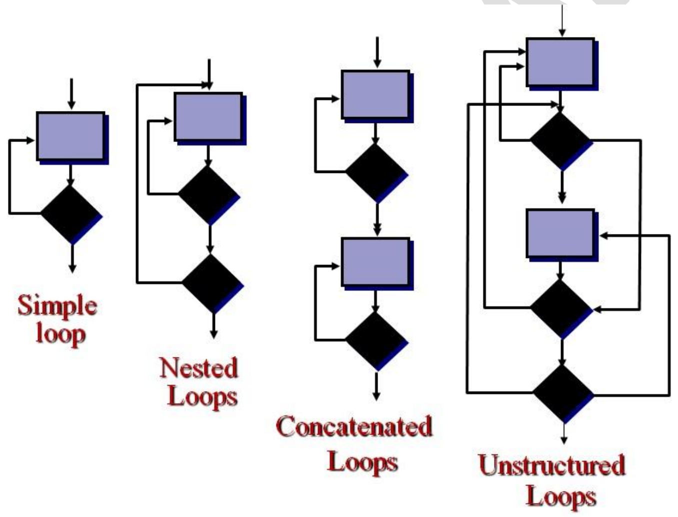
- Branch testing
  - For a compound condition, C, the true and false branches of C and every simple condition in C need to be executed at least once
- Domain testing
  - Testing related to relational operators
- BRO (branch and relational operator) testing

Data Flow Testing

- The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program
- Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements

Loop Testing

- Loops are the cornerstone for the vast majority of all algorithms implemented in software, and yet, we often them little heed while conducting software tests
- Complex loop structures are another hiding place for bugs. It's well worth spending time designing tests that fully exercise loop structures
- It is a white-box testing technique that focuses exclusively on the validity of loop constructs
- Four different classes of loops can be defined
    - Simple loop
    - Nested loops
    - Concatenated loops
    - Unstructured loops



Minimum conditions—Simple Loops

- Skip the loop entirely
- Only one pass through the loop
- Two passes through the loop
- m passes through the loop m < n
- (n-1), n, and (n+1) passes through the loop

where n is the maximum number of allowable passes

Nested Loops

- Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.
- Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.

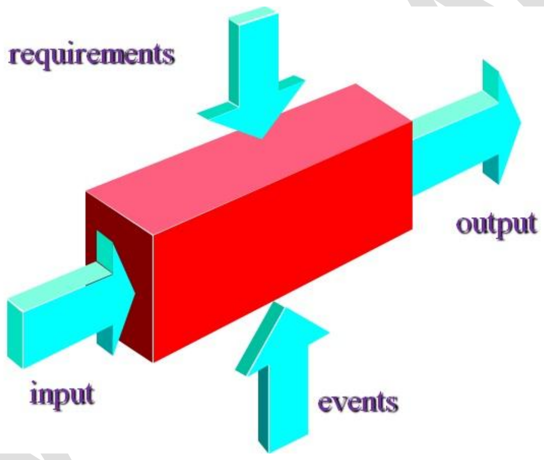
- Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

Concatenated Loops

- If the loops are independent of one another
- then treat each as a simple loop
- else* treat as nested loops
- endif*
- for example, the final loop counter value of loop 1 is used to initialize loop 2.

Black-box Testing

- BBT focuses on the functional requirements of the software
- It enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program
- It is not an alternative to WBT, rather a it is a complementary approach that is likely to uncover a different class of errors



- BBT attempts to find errors in the following categories
  - Incorrect or missing functions
  - Interface errors
  - Errors in data structures or external data base access
  - Behavior or performance errors
  - Initialization and termination errors
- As BBT purposely disregards control structure, attention is focused on the information domain
- Tests are designed to answer the following questions
  - How is functional validity tested?
  - How is system behavior and performance tested?
  - What classes of input will make good test cases?
  - Is the system particularly sensitive to certain input values?
  - How are the boundaries of a data class isolated?

  - What data rates and data volume can the system tolerate?
  - What effect will specific combinations of data have on system operation?
- By applying BBT, we derive a set of test cases that satisfy the following criteria
  - Test cases that reduce the number of additional test cases that must be designed to achieve reasonable testing
  - Test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand

BBT Techniques

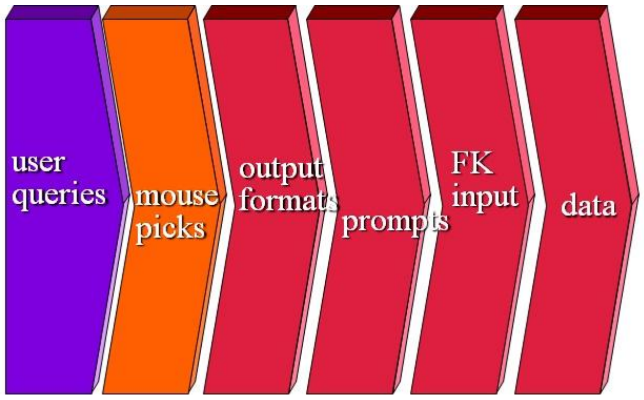
- Equivalence partitioning
- Boundary value analysis

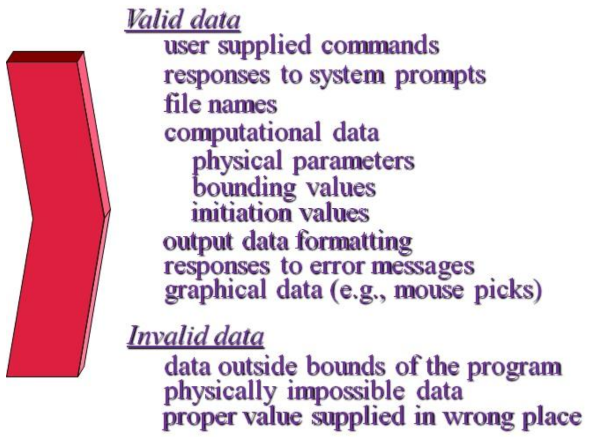
Equivalence Partitioning

- Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived
- Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed
- An equivalence class represents a set of valid or invalid states for input conditions
- Typically, an input condition is
  - A specific numeric value
  - A range of values
  - A set of related values
  - A boolean condition

Guidelines for Equivalence Classes

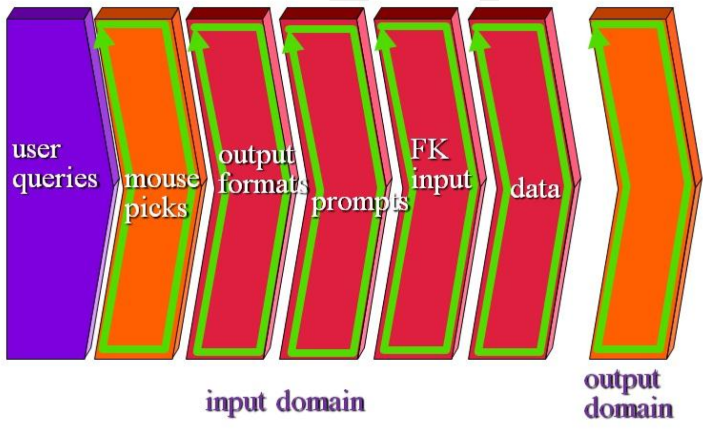
- If an input condition specifies a range, one valid and two invalid equivalence classes are defined
- If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
- If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined
- If an input condition is boolean, one valid and one invalid class are defined




Valid data
user supplied commands
responses to system prompts
file names
computational data
physical parameters
bounding values
initiation values
output data formatting
responses to error messages
graphical data (e.g., mouse picks)
Invalid data
data outside bounds of the program
physically impossible data
proper value supplied in wrong place

Boundary Value Analysis

- Boundary value analysis (BVA) is a testing technique, which leads to a selection of test cases that exercise bounding values
- This is because that for reasons not clearly completely clear, a greater number of errors tends to occur at the boundaries of the input domain rather than in the “center
- BVA complements equivalence partitioning testing technique



Testing Strategies

- Software testing needs to be planned in advance and conducted systematically
- A number of software testing strategies have been proposed in the literature
- All of them have these four generic characteristics

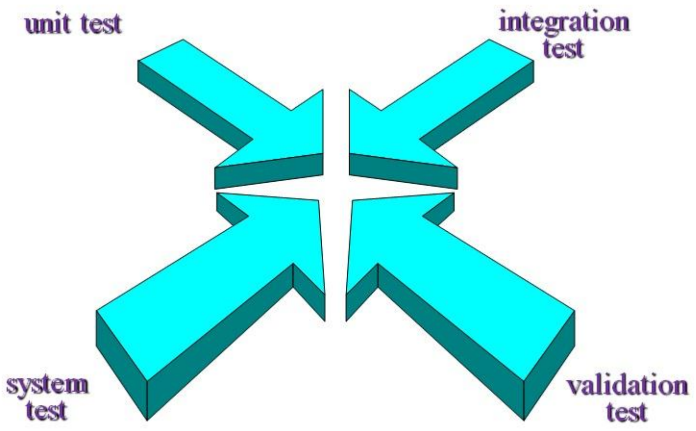
Characteristics of Testing Strategies

- Testing begins at the component level and works “outward” toward the integration of the entire computer-based system
- Different testing techniques are appropriate at different points in time
- Testing is conducted by the developer of the software and (for large projects) an independent test group

- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy
- A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements
- A strategy must provide guidance for the practitioner and a set of milestones for the manager
- Because the steps of the test strategy occur at a time when deadline pressure begins to rise, progress must be measurable and problems must surface as early as possible

Testing Strategy

- Unit testing
- Integration testing
- Validation testing
- System testing



- Unit testing concentrates on each unit (i.e., component) of the software as implemented in source code
- Integration testing is next, where the focus is on design and the construction of the software architecture
- Next is the validation testing, where requirements established as part of software requirements analysis are validated against the software that has been constructed
- Finally, we have system testing, where the software and other system elements are tested as a whole

Unit testing

- Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module’s control structure to ensure complete coverage and maximum error detection

Integration Testing

- Next, components must be assembled or integrated to form the complete software package

- Integration testing addresses the issues associated with the dual problems of verification and program construction
- BB test case design techniques are the most prevalent during integration

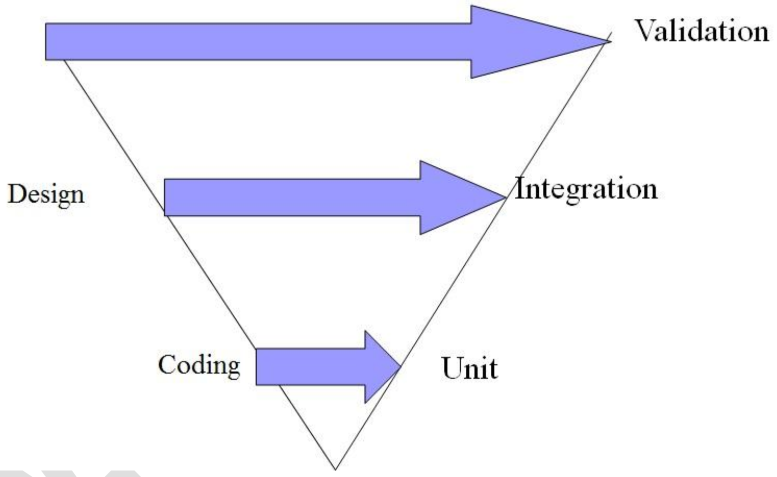
Validation Testing

- After software has been integrated, a set of high-order tests are conducted
- Validation criteria (established during requirements analysis) must be tested
- Validation testing provides final assurance that software meets all functional, behavioral, and performance requirements
- BBT techniques are used exclusively here

System Testing

- System testing is the last high-order testing and it falls outside the boundaries of software engineering and into the broader context of computer system engineering
- System testing verifies that all elements (hardware, people, databases) mesh properly and that overall system function/performance is achieved

The "V" Model of Software Testing

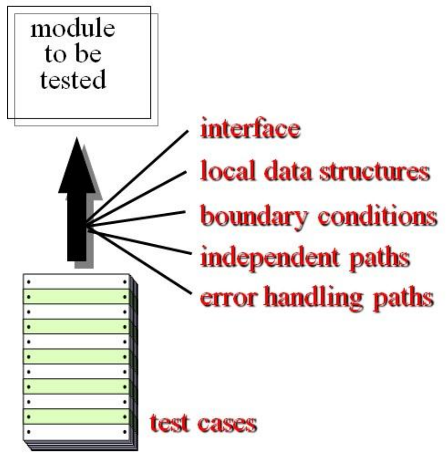


Strategic Issues

- Specify product requirements in a quantifiable manner long before testing commences
  - NFRs
- State testing objectives explicitly
- Understand the users of the software and develop a profile for each user category
  - Actors/users of use-cases
- Develop a testing plan that emphasizes "rapid cycle testing"
- Build "robust" software that is designed to test itself
  - Software should be capable of diagnosing certain classes of errors
- Use effective formal technical reviews as a filter prior to testing
- Conduct formal technical reviews to assess the test strategy and test cases themselves
- Develop a continuous improvement approach for the testing process
- Let's have some detailed discussions on each of the testing strategies introduced

Unit Testing

- Unit testing focuses verification effort on the smallest unit of software design – the software component or module
- The relative complexity of tests and uncovered errors is limited by the constrained scope established for unit testing
- Different units can be tested in parallel



- The module interface is tested to ensure that information properly flows into and out of the program unit under test
- The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution
- Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed
- All error handling paths are tested
- Tests of data flow across a module interface are required before any other test is initiated

Common Errors Found

- Among the more common errors in computation are
  - Misunderstood or incorrect arithmetic precedence
  - Mixed mode operations
  - Incorrect initialization
  - Precision inaccuracy
  - Incorrect symbolic representation of expression

Test Cases

- Test cases should uncover errors such as
  - Comparison of different data types
  - Incorrect logical operators or precedence

- Expectation of equality when precision error makes equality unlikely
- Incorrect comparison of variables
- Improper or non-existent loop termination
- Failure to exit when divergent iteration is encountered
- Improperly modified loop variables

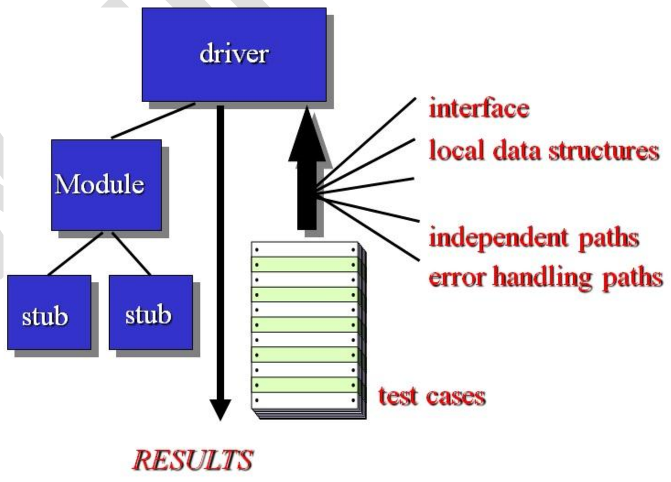
Error Handling

- Among the potential errors that should be tested when error handling is evaluated are
  - Error description is unintelligible
  - Error noted does not correspond to error encountered
  - Error condition causes system intervention prior to error handling
  - Exception-condition processing is incorrect
  - Error description does not provide enough information to assist in the location of the cause of the error

Boundary Testing

- Boundary testing is the last and probably the most important task of unit testing step
- Software often fails at its boundaries
- Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors
- Unit testing is simplified when a component with high cohesion is designed
- When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered
- Let's see a unit test environment

Unit Test Environment



Integration Testing

- "If they all work individually, why do you doubt that they'll work when we put them together?"
- The problem, of course, is "putting them together" – interfacing

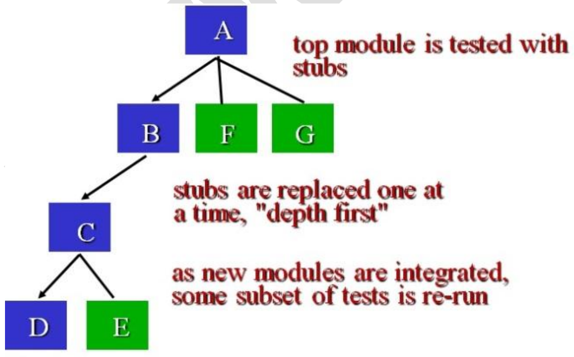
- Data can be lost across an interface
- One module can have an inadvertent, adverse affect on another
- Sub-functions, when combined, may not produce the desired major function
- Individually acceptable imprecision may be magnified to unacceptable levels
- Global data structures can present problems
- Software Integration (as following diagram)

- Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing
- Individual components have already been unit tested and the structure is dictated by design
- There is tendency, sometimes, to attempt non-incremental integration – a sort of "big bang" approach
- All components are combined in advance
- The entire software is tested as a whole
- And chaos usually results!
- We should rather use incremental integration

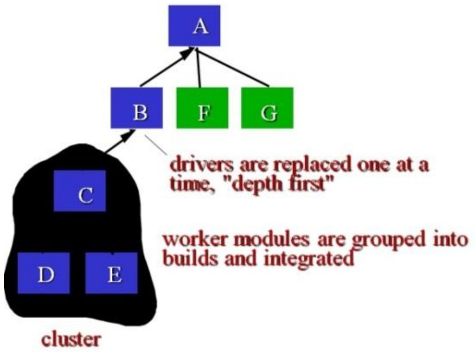
Integration Testing Approaches

- Top-down integration
- Bottom-up integration
- Sandwich testing and integration

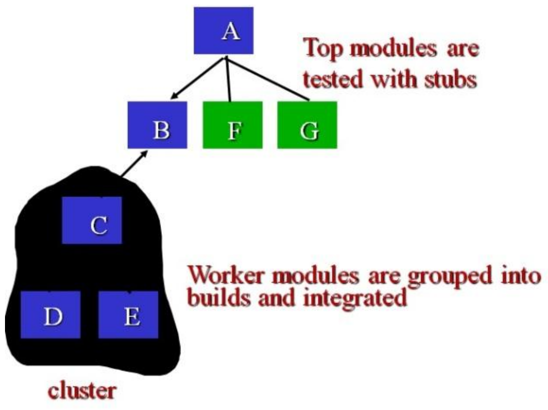
Top Down Integration


A
top module is tested with stubs
B
F
G
stubs are replaced one at a time, "depth first"
C
as new modules are integrated, some subset of tests is re-run
D
E

Bottom-Up Integration


A
B
F
G
drivers are replaced one at a time, "depth first"
C
worker modules are grouped into builds and integrated
D
E
cluster

Sandwich Testing and Integration


A
Top modules are tested with stubs
B
F
G
C
Worker modules are grouped into builds and integrated
D
E
cluster

Validation Testing

- Validation succeeds when software functions in a manner that can be reasonably expected by the customer
- Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements
- A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity to requirements
- Both test plan and procedure are designed to ensure that
    - All requirements are satisfied
    - All behavioral characteristics are achieved
    - All performance requirements are attained
    - Documentation is correct
    - Human engineered and other requirements are met
- After each validation test case has been conducted, one of two possible conditions exist
    - (1) The function or performance characteristics conform to specification and are accepted
    - (2) A deviation from specification is uncovered and a deficiency list is created
- Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery
- It is often necessary to negotiate with customer to establish a method for resolving deficiencies

Other High-Order Testing
Acceptance testing (bespoke software)
Alpha testing (developer's site by customer)
Beta testing (at customer site without developer's presence)
System testing

Specialized Testing

- Recovery testing
- Security testing
- Stress testing
    - Sensitivity testing
- Performance testing

Testing and Litigation of Poor Software Quality

	Reliable Software	Software Involved in Litigation for Poor Quality
Subroutine testing	Used	Used
Unit testing	Used	Used
New function testing	Used	Rushed or omitted
Regression testing	Used	Rushed or omitted
Integration testing	Used	Used
System testing	Used	Rushed or omitted
Performance testing	Used	Rushed or omitted
Capacity testing	Used	Rushed or omitted

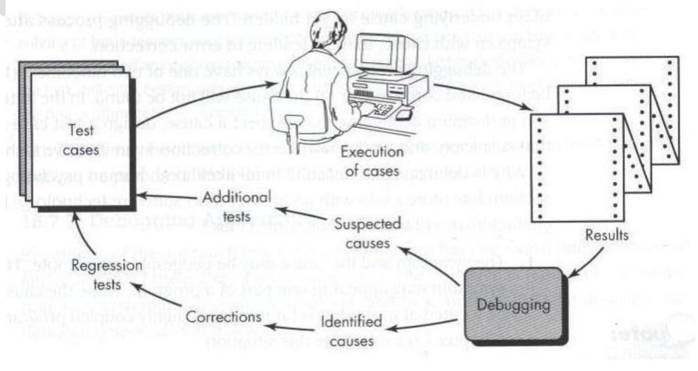
The Art of Debugging

- Debugging occurs as a consequence of successful testing
- That is, when a test case uncovers an error
- Debugging is the process that results in the removal of the error
- Although debugging can and should be a process, it is still very much an art

Debugging

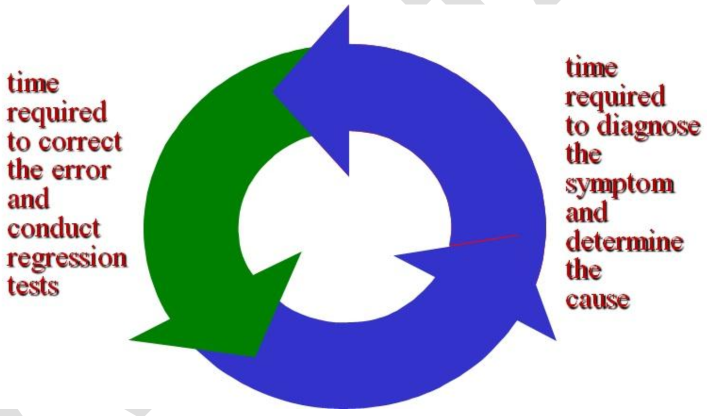
- Debugging occurs as a consequence of successful testing
- That is, when a test case uncovers an error
- Debugging is the process that results in the removal of the error
- Although debugging can and should be a process, it is still very much an art
- A software engineer, evaluating the results of a test, is often confronted with a "symptomatic" indication of a software problem
- That is, the external manifestation of the error and the internal cause of the error may have no obvious relationship to one another
- The poorly understood mental process that connects a symptom to a cause is debugging
- Debugging is not testing but always occurs as a consequence of testing

The Debugging Process



- Begins with the execution of a test case
- Results are assessed and a lack of correspondence between expected and actual performance is encountered
- In many cases, the non-correspondence data are a symptom of an underlying cause as yet hidden
- The debugging process attempts to match symptom with cause, thereby leading to error correction
- The debugging process will always have one of two outcomes
    - The cause will be found and corrected
    - The cause will not be found
- In the second case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion
- The debugging process is a human-oriented process
- Some people are good at it and others aren't
- Large variances are reported in debugging ability for programmers with the same education and experience

Debugging Effort


time required to correct the error and conduct regression tests
time required to diagnose the symptom and determine the cause

Debugging

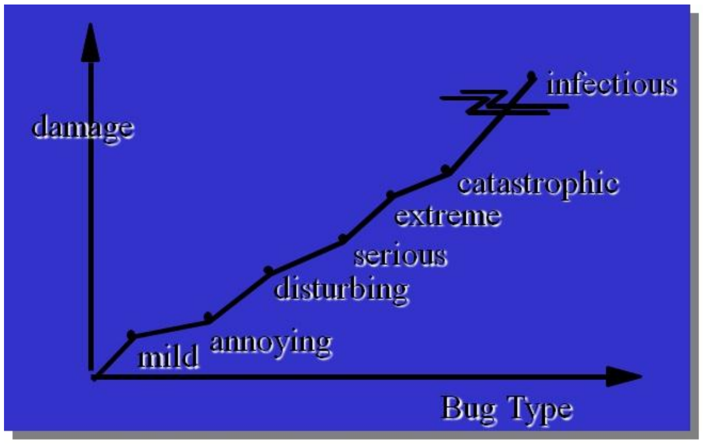
- Debugging is one the more frustrating parts of programming. It has elements of problem solving or brain teasers, coupled with the annoying recognition that you have made a mistake. Heightened anxiety and unwillingness to accept the possibility of errors increases the task difficulty. Fortunately, there is a great sigh of relief and a lessening of tension when the bug is ultimately corrected

Characteristics of Bugs

- The symptom and cause may be geographically remote. That is, the symptom may appear in one part of the program, while the cause may actually be located at a site that is far removed. Highly coupled program structures exacerbate this situation
- The symptom may disappear (temporarily) when another error is created
- The symptom may actually be caused by non-errors (e.g., round-off inaccuracies)
- The symptom may be caused by human error that is not easily traced

- The symptom may be a result of timing problems, rather than processing problems
- It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate)
- The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably
- The symptom may be due to causes that are distributed across a number of tasks running on different processors

Consequences of Bugs



Bug Categories: **function-related bugs, system-related bugs, data bugs, coding bugs, design bugs, documentation bugs, standards violations, etc.**

Debugging Approaches

- Regardless of the approach that is taken, debugging has one overriding objective: to find and correct the cause of a software error
- The objective is realized by a combination of systematic evaluation, intuition, and luck
- Regardless of the approach that is taken, debugging has one overriding objective: to find and correct the cause of a software error
- The objective is realized by a combination of systematic evaluation, intuition, and luck
- Brute force
- Backtracking
- Cause elimination

Brute Force Debugging

- The brute force debugging is probably the most common and least efficient method for isolating the cause of a software error
- We apply brute force debugging methods when all else fails, using a “let the computer find the errors” philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements
- We hope that somewhere in the morass of information that is produced we will find a clue that can lead us to the cause of an error

- Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time
- Thought must be expended first

Backtracking Debugging

- Backtracking is a fairly common debugging approach that can be used successfully in small programs
- Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found
- As the number of source lines increases, the number of potential backward paths may become unmanageably large

Cause Elimination Debugging

- Cause elimination is manifested by induction or deduction and introduces the concept of binary partitioning
- Data related to the error occurrence are organized to isolate potential causes
- A “cause hypothesis” is devised and the aforementioned data are used to prove or disprove the hypothesis
- Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each
- If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug

Fixing a Bug

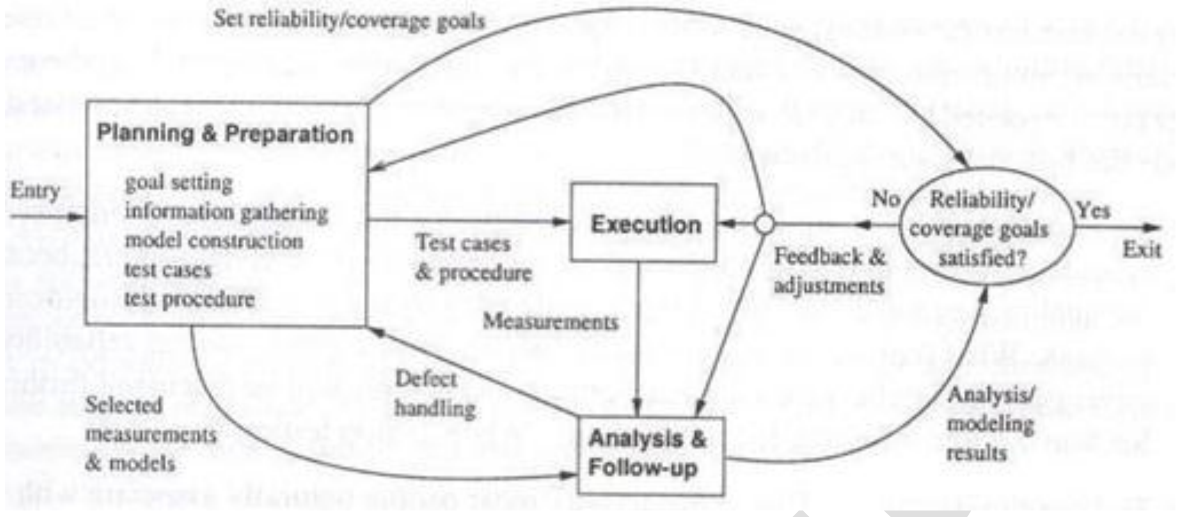
- Once a bug has been found, it must be corrected
- We know that at this time, new defects can be introduced
- So, we need to ask three questions before we fix a bug or repair a software defect

Questions before Correction

- Is the cause of the bug reproduced in another part of the program?
- What “next bug” might be introduced by the fix I’m about to make
- What could we have done to prevent this bug in the first place?

Generic Testing Process

- Test planning and preparation
- Test execution and related activities
- Analysis and follow-up



Test Planning and Preparation

- As you can see from the picture just shown to you that test planning and preparation is the most important activity in the generic testing process for systematic testing
- Most of the key decisions about testing are made during this stage
- Setting of goals for testing
- Selecting an overall testing strategy
- Preparing specific test cases
- Preparing general test procedure

Test Planning

- It is difficult to manage the testing of a large system, considering numerous test cases, problem reporting, and problem fixes
- Just as we plan and manage a project by estimating resources, schedule, cost, and so forth, we must plan the testing of a software product
- The planning activities enable us to do strategic thinking on what types of tests should be executed within the constraints of time and budget
- During the preparation of the test plan, as each test scenario is planned, we must estimate the time it would take to test the system
- The mistake most testers make is not planning the test schedule upfront
- They fail to allocate adequate time to test the product and conduct regression testing
- If one of the test dependencies identified upfront in the planning stage is receiving the code on time for testing, the management will realize that if the code does not arrive in testing on time, the entire project schedule will slip
- It’s too late to begin test planning when testing actually begins. All that you can do then is hope that baseline designing and unit testing have been so well that things fall into place easily
- This sort of luck is seldom achieved in real projects
- Planning is the only reliable way to mitigate such problems
- Beizer claims that test planning can uncover at least as many problems as the actual

tests themselves
- This is a very powerful statement based on the observations of a very seasoned and senior researchers in software engineering community
- The test plan acts as a service level agreement between the test department and other significantly impacted departments
- The testing efficiency can be monitored and improved by developing a through test plan
- A test plan identifies the test deliverables, functions to be tested, and risks
- A test plan communicates to the audience
    - The methodology which will be used to test the product
    - The resources
        - Hardware
        - Software
        - Manpower
    - The schedule during which tests will be accomplished
    - The process used to manage the testing project
- Before writing a test plan, the following information should be obtained
    - Names of all test analysts
    - Automated test tools to be used for this project
    - Environment required for conducting tests
- When developing a test plan, ensure that it is kept
    - Simple and complete
    - Current
    - Accessible
- The test plan should be frequently routed to the appropriate people for feedback and sign-offs
- A test plan can either be written by the test leader, test coordinator, or test manager

Guidelines for Developing Test Plan
- Depending on the features of the system, determine which tests you need to perform
- Once you have identified the types of tests needed to test the system, plan how each of these tests will be performed. Develop test cases
- Use a test plan template (we'll discuss one today)
- When the contents of the test plan are finalized, conduct a review for completeness of the test plan
- Incorporate the feedback from the review of the test plan
- Get approvals on the contents of the test plan from managers of all the departments that will be impacted by this new product, for example, the managers of Development, Customer Support, Marketing, Operations, Integrated Technology, Product Assurance, Quality and User Management
- Follow the test plan to ensure that everyone on the test team uses the process outlined in the test plan and if there are exceptions to this, the deviations are logged as "addenda" to the test plan. For consistency, these addenda should be approved by the same group of individuals who approved the original test plan

- A well-thought test plan will save hours of frustration at the end of test cycle, reduce risks, and provide a tool for management reporting
- As many members of the project team should contribute to the test plan document through reviews and discussion as possible
- All the participants will feel committed to the plan since they helped to create it and this fosters a high degree of dedication among participants to follow what is being said in the test plan
- If participants accept responsibilities and later do not fulfill them, the test plan provides evidence of the initial agreement
- A test plan provide criteria for measurement, deadlines for test deliverables, and describes procedures for documenting and resolving problems

Table of Contents of a Sample Test Plan

- Introduction
- Scope
- Test plan strategy
- Test environment
- Schedule
- Control procedures
- Control activities
- Functions to be tested
- Functions not to be tested
- Risks and assumptions
- Deliverable
- Test tools
- Approvals
- Exit criteria

Introduction

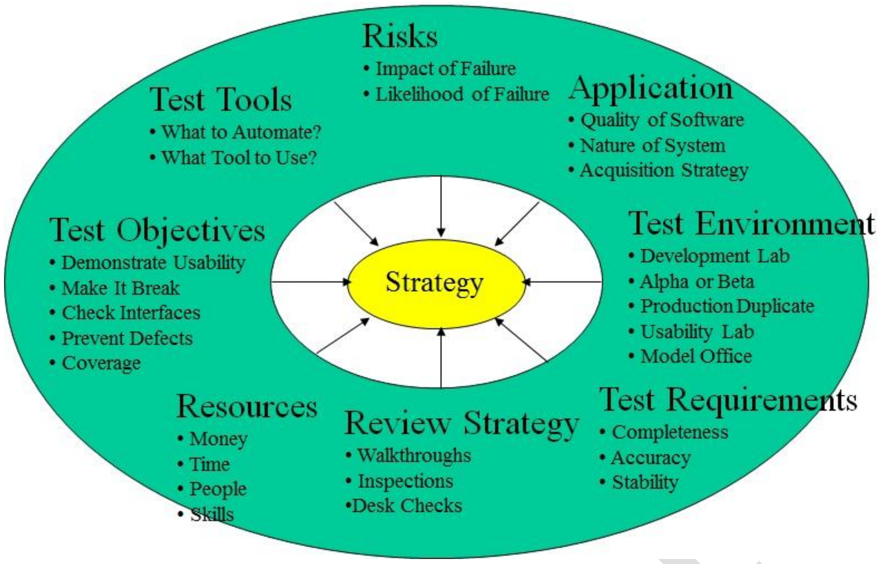
- The introduction gives a brief description of the system being tested so anyone reading the test plan would get an overview of the features of the product being tested

Scope

- The scope describes at a high level what will be tested and what will not be tested in order to avoid ambiguity and identifies what is going to be addressed by the testers

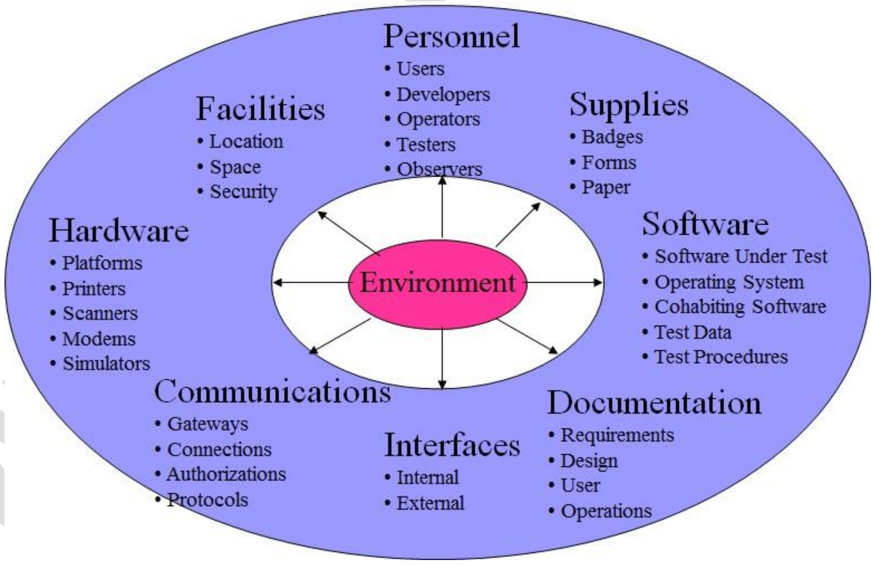
Test plan strategy

- Test plan strategy outlines the objectives for developing the test plan and establishes the procedures for conducting the tests
- The strategy should also identify all the tests that will be performed to check the robustness of the product
- A brief description of each test should be given



Test Environment

- The test environment section lists the hardware and software configurations required to perform tests
- It determines the type of help required from other departments to support the environment tests



Schedule

- The overall test schedule is identified in the project plan indicating the start date and end date of test activities. However, detailed schedule of testing should be developed during test planning process, showing start and end dates for
  - Development of the test plan
  - Designing test cases
  - Executing test cases
  - Problem reporting

  - Developing the test summary report

Control Procedures

- This section should have detailed instructions for problem reporting
- A procedure for communicating defects to the development team should be established and published in this section
- This should include
  - Whether an automated on-line system will be used to record and report problems
  - Whether the problem fixes will be accepted daily or once a week
  - How the problems should be prioritized, for example 1, 2, or 3

Control Activities

- The control activities section indicates when walkthroughs or reviews will be conducted and list the individuals who would participate in these activities

Functions to Be Tested

- This section lists all the functions to be tested
- What you test is more important than how much you test

Functions Not to Be Tested

- This section lists all functions not to be tested, along with the reason for not testing, for example, “low risk” code reused and previously tested with 100% satisfaction

Risks and Assumptions

- This section lists assumptions taken into consideration when developing the test schedule, such as
  - Unit testing will be done prior to receiving code for system testing
  - All testing personnel will have the knowledge of the product and will be experienced testers
  - The computers will be available during all work hours

Deliverable

- The deliverable section lists what will be delivered at the end of test phase, such as,
  - Test plan
  - Test cases
  - Test incident report
  - Test summary report

Test Tools

- This section identifies all test tools which will be used during testing, whether they are in-house or need to be purchased
- Test coverage analyzers might be used to provide information on the thoroughness of one or more test runs of the program
- At the end of the analysis process, the analyzer produces a visual listing of the program with all unexecuted code flagged

Approvals

- This section ensures that the plan is reviewed and approved by the individuals who will either be affected by it, or on whom you are dependent to either receive listing of program with all unexecuted code flagged

Exit Criteria

- This section identifies the exit criteria for the test phase. Some examples of the exit criteria:
  - No priority-one open problems
  - All functions identified in the requirements document are present and working
  - No more than three priority-two problems open

Automated software testing

Introduction

- Studies have shown that companies are missing shipping deadlines for their software products
- Overall 90% of developers have missed ship dates, and missing deadlines is a routine occurrence for 67% of developers
- These numbers are very alarming for our industry
- Today's software managers and developers are being asked to turn around their products within ever-shrinking schedules and with minimal resources
- Getting a product to market as early as possible may mean the difference between product survival and product death – and therefore company survival and death
- In an attempt to do more with less, organizations want to test their software adequately, but within a minimal schedule
- To accomplish this, organizations are turning to automated testing
- So, what is automated testing?

Automated Testing

- The management and performance of test activities, to include the development and execution of test scripts so as to verify test requirements, using an automated test tool
- Automated software testing addresses the challenge for today's software professionals who are faced with real schedule deadlines
- The automation of test activities provides its greatest value in instances where test scripts are repeated or where test script sub-routines are created and then invoked repeatedly by a number of test scripts
- The performance of integration test using an automated test tool for subsequent incremental software builds provide great value
- Each new build brings a considerable number of new tests, but also reuses previously developed test scripts
- Regression testing at the system test level represents another example of the efficient use of automated testing
- Regression tests seek to verify that the functions provided by the modified system or software product perform as specified and that no unintended change has occurred in

the operation of the system or product

- Automated testing can provide several benefits when implemented correctly and follows a rigorous process
- The test engineer must evaluate whether the potential benefits fit the required improvement criteria and whether the pursuit of automated testing on a project is still a logical fit, given the organizational needs

Benefits of Automated Testing

- Production of a reliable system
- Improvement of the quality of the test effort
- Reduction of the test effort and minimization of the schedule

Production of a Reliable System

- Improved performance testing
- Improved load/stress testing
- Quality measurements and test optimization
- Improved partnership with development team
- Improved system development life cycle

Improved Quality of the Test Effort

- Improved build verification testing
- Improved regression testing
- Improved multiplatform compatibility testing
- Improved software compatibility testing
- Improved execution of mundane tests
- Improved focus on advanced test issues
- Execution of tests that manual testing can't accomplish
- Ability to reproduce software defects
- Documentation of business knowledge
- After-hour testing

Reduction of Test Effort and Minimization of Schedule

- Test plan development
- Test procedure development
- Test execution
- Test result analysis
- Error status/correction monitoring
- Report creation

Manual Versus Automated Testing

Test Steps	Manual Testing (Hrs)	Automated Testing (Hrs)	Percentage Improvement
Test Plan Development	32	40	-25 %
Test Procedure Development	262	117	55 %
Test Execution	466	23	95 %
Test Result Analysis	117	58	50 %
Error Status / Correction Monitoring	117	23	80 %
Report Creation	96	16	83 %
Total Duration	1090	277	75 %

False Expectations for Automated Testing

- Automatic test plan generation
- Test tool fits all
- Immediate test effort reduction
- Immediate schedule reduction
- Tool ease of use
- Universal application of test automation
- One hundred percent coverage

Automated Testing in Rapid Software Development

- As rapid application development has become popular in the last few years, automated testing has also become an integral part
- Particularly, with the increase in the number of projects using agile manifesto, automated software testing is being used extensively
- Many agile software development processes include automated software testing as an integral part of the way of they develop software – a sort of modus operandi
- This is an area that is open for more research and certainly productivity and efficiency of software testing can increase substantially with automated software testing

Automated Test Tools

- Usually support automated execution of black-box tests
- An automated test tool reads a test script, plays the script, and compare the actual results to the original or expected responses
- They enable repeatability and consistency of test execution as well as productivity
- Some tool suites also provide the ability to distribute or simulate the execution of tests over multiple processors, thus providing performance and stress tests to measure the system's ability to handle high volumes of transactions and users

Evaluation Criteria for Test Tools

- Compatibility
  - Does the tool support the platforms and environments of the system under test?
  - Client-Level Compatibility
  - Communications-Level Compatibility
  - Host-Level Compatibility
  - Portability
- Usability
  - Does it provide the necessary functionality, and are the testers able to make effective use of it without a prohibitive learning curve?
  - Functionality
  - Extensibility
  - Learning Curve
- Maintainability
  - Can changes to the system under test be accommodated in the automated tests without undue effort?
- Manageability
  - Does the tool provide sufficient and meaningful information to enable management of the test library and measurement of the test results?
  - Test Library Management

Test Results

- Test results should be meaningful to inform management of the status of the system under test and its quality relative to expectations
- Test tools should be reviewed to determine the amount and quality of test result information; analysis and reporting capabilities are either included or can be made available

Test Case Generation

- Programmatic creation of test case data, as opposed to sampling or other techniques
- Advantage: High volumes of test conditions can be created
- Disadvantage: Volume alone is not guarantor of coverage or efficacy
- Equivalence classes
  - Large number of test cases
  - More than one test case that exercises the same path, or too few to cover all the paths
- Cause-effect node graphing

Vendor Selection

- Experience
  - What level of experience does the vendor offer in the implementation of automated test tools?
- Service
  - What services does the vendor offer for training as well as implementation to assure that the tool is properly deployed?

- Support
  - How readily available and responsive is the vendor to technical questions and issues?
- Commitment
  - Is the vendor committed to the automated test tools market or is it only one of several offerings?

Test Cases

- A test case describes how each test is to be conducted and also describes input/output details
- Development of test cases assist in keeping track of what is tested, when it is tested, and the outcome of the test
- If a defect is found in testing, a documented test case makes it easy for the developer to re-create the problem so that proper analysis can be done to fix it
- Selecting test cases is the single most important task that software testers do. Improper selection can result in testing too much, testing too little, or testing the wrong things
- Intelligently weighing the risks and reducing the infinite possibilities to a manageable effective set is where the magic is
- When designing and running your test cases, always run the test-to-pass cases first. It is important to see if the software fundamentally works before you throw the kitchen sink at it. You may be surprised how many bugs you find just by using the software normally

Design of Test Cases

- There are no standards or simple rules for designing test cases
- Test cases should be designed for testing each function, with expected results
- In order to determine what will break the system, the tester has to be creative and want to break the system
- The test cases should be repeatable, i.e., when a test case is rerun, it should give the same results each time, except in real-time environment
- Test cases should be organized and developed around system requirements and for key functions of the system
- The test design emphasizes
  - The type of test needed to test the requirements
  - How a test will run
  - How all requirements will be addressed by the tests
- Good test cases catch errors
- Develop test cases that cover as many related functions as possible
- A test case contains
  - Test case identifier
  - The name of the person responsible for executing the test
  - The date the test was executed and the version of the system tested
  - Description of the functions to be tested and a list of actions to be carried out
- Always, every, all, none, never
  - If you see words such as these that denote something as certain or absolute,

make sure that it is, indeed, certain. Think of cases that violate them

- Certainly, therefore, clearly, obviously, evidently
  - These words tend to persuade you into accepting something as given. Don't fall into the trap
- Some, sometimes, often, usually, ordinarily, customarily, most, mostly
  - These words are too vague. It's impossible to test a feature that operates "sometimes"
- Etc., And so forth, and so on, such as
  - Lists that finish with words such as these aren't testable. More explanation is needed
- Good, fast, cheap, efficient, small, stable
  - These are unquantifiable terms. They are not testable
- Handled, processed, rejected, skipped, eliminated
  - These terms can handle large amounts of functionality that need to be specified
- If…then…(but missing else)
  - Look for statements that have "if…then" clauses but don't have a matching "else." Ask yourself what will happen if the "if" doesn't happen

Characteristics of a Test Case

Repeatable	↔	Consistent
Destructive	↔	Detects Discrepancy
Manageable	↔	Results Can Be Traced

Types of Test Cases

- Functional Test Cases
- Computation and Boundary Analysis Test Cases
- Cause-Effect and Logic Cases
- Security Test Cases

Test Case Development Considerations

- All valid system input must be accepted
- All invalid system input must be rejected with clear error messages
- All functions – including generation of reports – should be tested
- Any reports generated by the system should be tested and verified
- Any interfacing systems must be invoked and verified that the data is passed to and from the interfacing system

Bad Test Cases

- The error density in software test cases is often higher than the error density in the software product
- Sometimes more than 15% of total number of test cases created can have errors themselves
- 50% of repair effort is applied towards test cases themselves

- Running test cases that are flawed or in error is of no value in terms of quality
- There is an implied need for greater rigor in building and validating test cases and test libraries

Common Problems in Test Cases

- Testing limits and ranges where those included in the test-case are themselves incorrect
- Testing for numeric values where the test data contains wrong data
- Test cases derived from specifications or user requirements which contained undetected errors that were accidentally passed on to the test cases
- Accidental redundancy of test cases or test libraries (30%)
- Gaps, or portions of the application for which no test case exist (70%)
  - Result of gaps in requirements
- Formal inspections on test cases are very effective in eliminating bad test cases

Interesting Observations

- Test cases created by the programmers themselves have the lowest probability of being duplicated, but the highest probability of containing errors or bugs
- Test cases created later by test or quality assurance personnel have the greatest probability of redundancy but lower probability of containing errors

Software Testers

Who Should Do Testing?

- The developers themselves
- Professional test personnel
- Professional quality assurance personnel
- Some combination of the above three

Some Observations About Testers

- The defect removal efficiency of black-box testing is higher when performed by test personnel or by quality assurance personnel rather than developers themselves
- Black-box testing performed by clients (Beta and acceptance testing) varies widely, but efficiency rises with the numbers of clients involved
- For usability problems, testing by clients themselves outranks all other forms of testing
- The defect removal efficiency of white-box subroutine and unit testing stages is highest when performed by developers themselves
- The defect removal efficiency of specialized kinds of white-box testing such as Year 2000 testing or viral protection testing is highest when performed by professional test personnel rather than by the developers themselves

Distribution of Testers

- On average
  - Developers do 31% of testing
  - Professional test personnel do 31% of testing
  - Quality assurance personnel do 18% of testing
  - Clients do 20% of testing

Criteria for Completion of Testing

- You're never done testing, the burden simply shifts from you to your customer
- You're done testing when you run out of time or you run out of money
- There is no definitive answer

When to Stop Testing?

- On a small or local scale, we can ask: "When to stop testing for a specific test activity?"
- On a global scale, we can ask: "When to stop all the major test activities?" because the testing phase is usually the last major development phase before product release, this question is equivalent to: "When to stop testing and release the product?"
- Resource-based criteria, where decision is made based on resource consumptions. The most commonly used such stopping criteria are
  - "Stop when you run out of time"
  - "Stop when you run out of money"
- Such criteria are irresponsible, as far as product quality is concerned, although they may be employed if product schedule or cost are the dominant concerns for the product in question
- Activity-based criteria, commonly in the form:
  - "Stop when you complete planned test activities"
- This criterion implicitly assumes the effectiveness of the test activities in ensuring the quality of the software product. However, this assumption could be questionable without strong historical evidence based on actual data from the project concerned
- No we cannot be absolutely certain that the software will never fail, but relative to a theoretically sound and experimentally validated statistical model, we have done sufficient testing to say with 95% confidence that the probability of 1000 CPU hours of failure free operation in a probabilistically defined environment is at least 0.995
  - Musa and Ackerman

Software Configuration Management

- This area is software configuration management
- In today's lecture, we'll introduce the basic concepts of software configuration management and develop a foundation for further discussions in later lectures
- Change is inevitable when software is built
- Changes will happen in all work products and during all processes during software development and maintenance
- Change increases the level of confusion among software engineers who are working on a software project

Confusion

- Confusion arises when changes are not
  - Analyzed before they are made
  - Recorded before they are implemented
  - Reported to those who need to know
  - Controlled in a manner that will improve quality and reduce error
- We need to minimize this confusion, or else our projects will get out of control

Configuration Management

- The art of coordinating software development to minimize …confusion is called configuration management
- Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team
- The goal is to maximize productivity by minimizing mistakes

Software Configuration Management

- Software configuration management (or SCM) is an umbrella activity that is applied throughout the software process
- This means that SCM is not tied to one process model or another, instead it used in all instances of software development processes
- SCM is a set of activities designed to control change by identifying the work products that are likely to change, establish relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes
- So, SCM provides a cover against
  - Lack of visibility
  - Lack of control
  - Lack of traceability
  -
  - Lack of monitoring
  - Uncontrolled change
- Software configuration management provides a means for visibility, traceability, and formally controlling the evolution of software

- SCM should be viewed as a software quality assurance activity that is applied throughout the software process
- Let's now have more detailed discussion on software configuration management
- The items that comprise all information produced as part of the software process are collectively called a software configuration
  - Computer programs (source and executable)
  - Documents that describe the computer programs
  - Data
- Software configuration items will grow

Purpose of SCM Activities

- Identify change
- Control change
- Ensure that the change is being properly implemented
- Report changes to others who may be interested
- If we don't control change, it will control us
- It's very easy for a stream of uncontrolled changes to turn a well-run software project into chaos
- For that reason, SCM is an essential part of good project management and is a solid software engineering practice

Change - A Constant

- The number of configuration items in a software project will grow continuously and the changes within each configuration item will also occur on a frequent basis
- So, we can say that
  - There is nothing permanent except change
    - Heraclitus (500 B.C.)
- No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle
- Software is like a sponge due to its susceptibility to change
- We emphasize this point, because this is the first step in planning and implementing a good software configuration management process

Sources of Change

- New business or market conditions dictate changes in product requirements or business rules
- New customer needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by computer-based system
- Reorganization or business growth / downsizing causes changes in project priorities or software engineering team structure
- Budgetary or scheduling constraints cause a redefinition of the system or product

Change is Everywhere

- Customers want to modify requirements
- Developers want to modify the technical approach
- Managers want to modify the project strategy

Why All This Modification?

- As time passes, all constituencies know more
  - About what they need
  - Which approach would be best
  - How to get it done and still make money
- Most changes are justified!
- We, in the software industry need to be ready to manage and implement changes in configuration items
- We need to have mechanisms in place to manage changes in all work products, in all processes of software development
- We can apply the concept of baselines

How to Manage Change?

- A baseline is a software configuration management concept that helps us to control change without seriously impeding justifiable change

Baseline

- A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures
  - IEEE Std. No. 610.12-1990
- Before a software configuration item becomes a baseline, change may be made quickly and informally
- However, once a baseline is established, changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change request
- In the context of software engineering, a baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and approval of these software configuration items is obtained through a formal technical review or inspection
- Typical, work products that are base-lined are
  - System specification
  - Software requirements
  - Design specification
  - Source code
  - Test plans/procedures/data
  - Operational system
- SCM is an important element of SQA program
- Its primary responsibility is the control of change
- Any discussion of SCM introduces a set of following complex questions. Listen carefully

to these questions. Each one of them is related to one or more aspect of software quality

SCM Questions

- How does an organization identify and manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
- How does an organization control changes before and after software is released to a customer?
- Who has the responsibility for approving and ranking changes?
- How can we ensure that changes have been made properly?
- What mechanism is used to appraise others of changes that are made?
- These questions lead us to definition of five SCM functions

SCM Functions

- Identification of software configuration items
  - To control and manage software configuration items, each item must be separately named or numbered
  - The identification scheme is documented in the software configuration management plan
  - The unique identification of each SCI helps in organization and retrieval of configuration items
- Version control
  - Version control combines procedures and tools to manage different versions of configuration items that are created during the software process
  - The naming scheme for SCIs should incorporate the version number
  - Configuration management allows a user to specify alternative configurations of the software system through the selection of appropriate versions
  - This is supported by associating attributes with each software version, and then allowing a configuration to be specified [and constructed] by describing the set of attributes
- Change control
  - Change control is vital
  - Too much change control and we create problems. Too little, and we create other problems (elaborate this point more)
  - The art of progress is to preserve order amid change and to preserve change amid order
  - Alfred North Whitehead
  - For large projects, uncontrolled change rapidly leads to chaos
  - For medium to large projects, change control combines human procedures and automated tools to provide a mechanism for the control of change
- Configuration audit
  - How can we ensure that the approved changes have been implemented?
  - Formal technical reviews/inspections

  - Focuses on the technical correctness of the modified item. The reviewers/inspectors assess the SCI to determine consistency with other SCIs, omissions, or potential side effects
  - Software configuration audit
  - A software configuration audit complements the formal technical reviews/inspections by assessing a configuration item for characteristics that are generally not considered during review
  - The SCM audit is conducted by the quality assurance group
- Status accounting/reporting
  - The status accounting function provides a corporate memory of project events that supports accomplishment of other configuration management items
    - What happened?
    - Who did it?
    - When did it happen?
    - What else will be affected?
- Audit Questions
  - Has the change specified in the ECO been made? Have any additional modifications been incorporated?
  - Has a formal technical review/inspection been conducted to assess technical correctness?
  - Has the software process been followed and have software engineering standards been properly applied?
  - Has the change been "highlighted" in the SCI? Have the change date and change author been specified? Do the attributes of the configuration item reflect the change?
  - Have SCM procedures for noting the change, recording it, and reporting it been followed?
  - Have all related related SCIs been properly updated?
- Status Accounting/Reporting
  - The status accounting function provides a corporate memory of project events that supports accomplishment of other configuration management items
    - What happened?
    - Who did it?
    - When did it happen?
    - What else will be affected?
  - Each time an SCI is assigned a new or updated identification, a configuration status reporting (or CSR) entry is made
  - Each time a change is approved, a CSR entry is made
  - Each time a configuration audit is conducted, the results are reported as part of CSR task
  - Output from CSR may be placed in an on-line database for easy access, and CSR reports are issued on a regular basis to keep management and practitioners informed about important changes
  - Configuration status reporting plays a vital role in the success of a large software

development project
- When many people are involved, it is likely that "the left hand not knowing what the right hand is doing" syndrome will occur
- Two developers may attempt to modify the same SCI with different and conflicting intents
- A software engineering team may spend months of effort building software to an obsolete hardware specification
- A person who would recognize serious side effects for a proposed change is not aware that the change is being made
- Configuration status reporting helps to eliminate these problems by improving communication among all people involved

Change Control Process

- Without proper safeguards, change control can retard progress and create unnecessary red tape
- It is relatively easy to incorporate changes before a work-product has been base-lined – the author has the authority to incorporate changes based on the organization's policy, project management's guidelines, and according to the technical needs of the project. We only need informal change control
- However, when a work-product has been base-lined after conducting a formal technical review or inspection, we need a more formal process to incorporate changes
- Before the release of software, project level change control is implemented
- However, when the software product is released to the customer, strict formal change control is instituted
- Proposed changes to software work-products are reviewed, then subjected to the agreement of project participants, and finally incorporated into the currently approved software configuration
- This requires that the separate authority, which reviews and approves all change requests be established for every project from among the project participants

Change Control Authority/Board

- This authority is known as change control authority (or CCA) or change control board (or CCB)
- We'll use these two terms interchangeably during this course
- A CCA/CCB plays an active role in the project level change control and formal change control activities
- Typically, a CCA consists of representatives from software, hardware, database engineering, support, and marketing, etc., depending on the size and nature of the project
- For every change request, the change control authority/board assesses the
    - Technical merit
    - Potential side effects
    - Overall impact on other configuration items and system functions
    - Projected cost of the change

- In addition, change control authority/board may assess the impact of change beyond the SCI in question
    - How will the change affect hardware?
    - How will the change affect performance?
    - How will the change modify customer's perception of the product?
    - How will the change affect product quality and reliability?
- The CCA/CCB has the authority to approve or reject a change request
- It can delay the request for consideration and inclusion in the next iteration also
- For every approved change request, an engineering change order (or ECO) is generated, which describes
    - The change to be made
    - The constraints that must be respected
    - The criteria of review and audit
- At this time, we have to implement the approved changes
- First, we need to "check-out" the configuration item that has to be changed from the project database
- Now we have access and authorization to make modifications to that specific SCI
- Then the approved change are made and necessary SQA and testing activities are applied
- That SCI is then "checked-in" to the project data base
- The changes are now part of the new version of the base-lined work-product

The Change Control Process

Need for change is recognized

↓

Change request from user

↓

Developer evaluates

↓

Change report is generated

↓

Change control authority decides → Change request is denied, User is informed

↓

Request is queued for action

↓

Assign people to SCIs

↓

Check-out SCIs

↓

Make the change

↓

Review/audit the change

↓

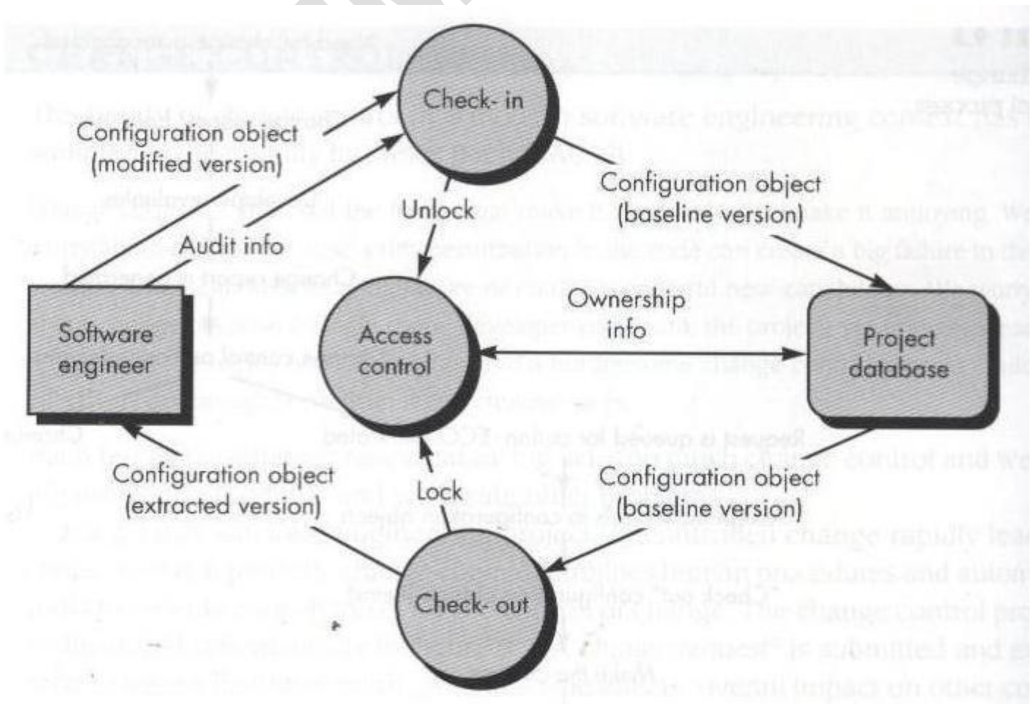
Establish a "baseline" for testing

↓
Perform SQA and testing activities
↓
Check-in the changed SCIs
↓
Promote SCI for inclusion in next release
↓
Rebuild appropriate version
↓
Review/audit the change
↓
Include all changes in release

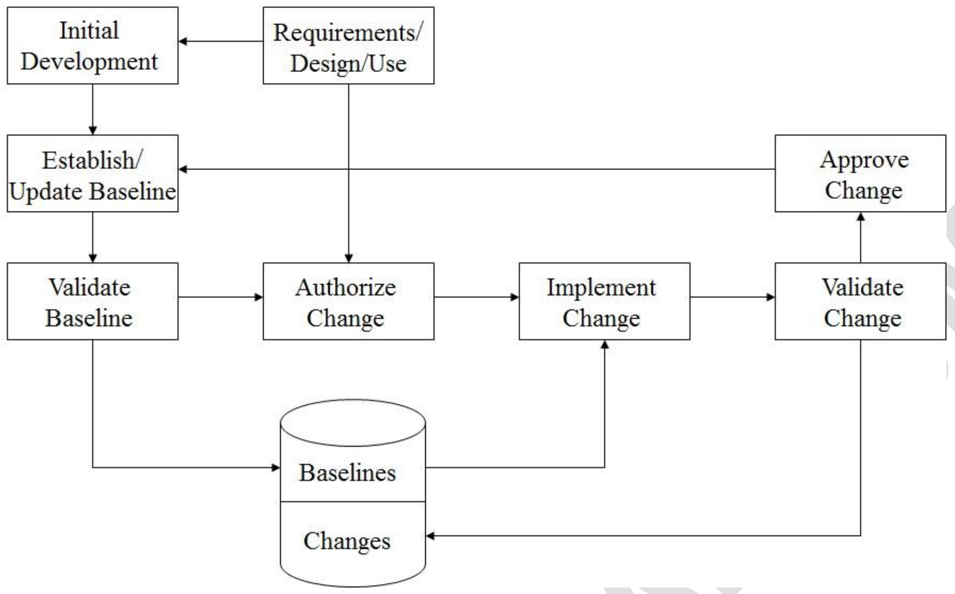
- As you can notice, that formal change control process is very elaborate and comprehensive process
- The "check-in" and "check-out" activities implement two important elements of change control – access control and synchronization control

Access and Synchronization Control

- Access control governs which software engineers have the authority to access and modify a particular configuration item
- Synchronization control helps to ensure that parallel changes, performed by two different people, don't overwrite one another
- We need to implement both



Software Configuration Management Overview



SCM Standards

- MIL-STD-483
- DOD-STD-480A
- MIL-STD-1521A
- ANSI/IEEE Std. No. 828-1983
- ANSI/IEEE Std. No. 1042-1987
- ANSI/IEEE Std. No. 1028-1988

It was mentioned in the first lecture on software configuration management that SCM provides a cover against lack of visibility and lack of traceability. So, now we'll discuss how SCM infuses visibility and traceability through its main functions

SCM Functions Infuse

- Visibility
  - Identification
    - User/buyer/seller can see what is being/has been built/is to be modified
    - Management can see what is embodied in a product
    - All project participants can communicate with a common frame of reference
  - Control
    - Current and planned configuration generally known
    - Management can see impact of change
    - Management has option of getting involved with technical detail of project
  - Auditing
    - Inconsistencies and discrepancies manifest
    - State of product known to management and product developers
    - Potential problems identified early

  - Accounting/Reporting
    - Reports inform as to status
    - Actions/decisions made explicit (e.g., through CCB meeting minutes)
    - Database of events is project history
- Traceability
  - Identification
    - ✓ Provides pointers to software parts in software products for use in referencing
    - ✓ Make software parts and their relationships more visible, thus facilitating the linking of parts in different representations of the same product
  - Control
    - ✓ Makes baselines and changes to them manifest, thus providing the links in a traceability chain
    - ✓ Provides the forum for avoiding unwanted excursions and maintaining convergence with requirements
- Traceability
  - Auditing
    - ✓ Checks that parts in one software product are carried through to the subsequent software product
    - ✓ Checks that parts in a software product have antecedents/roots in requirements documentation
- Accounting/Reporting
  - Provides history of what happened and when
  - Provides explicit linkages between change control forms

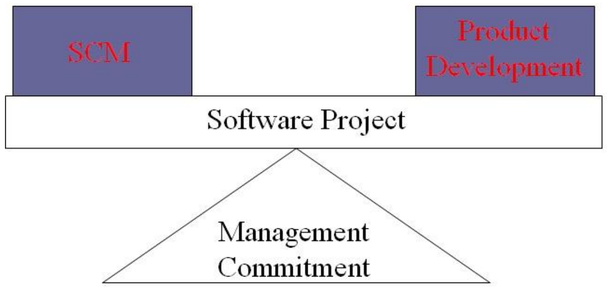
Real-World Considerations

- Management Commitment
- SCM Staffing
- Establishment of a CCB
- SCM During the Acceptance Testing Cycle
- Justification and Practicality of Auditing
- Avoiding the Paperwork Nightmare
- Allocating Resources among SCM Activities

Management Commitment

- Management commitment to the establishment of checks and balances is essential to achieving benefits from SCM

Management Commitment for SCM



SCM Staffing

- Initial staffing by a few experienced people quickly gains the confidence and respect of other project team members
- It is important to build the image that the SCM team has the objective of helping the other project team members achieve the overall team goals, that they are not a group of obstructionists and criticizers
- An SCM organization requires
  - Auditors
  - Configuration control specialists
  - Status accountants
- These positions require hard work and dedication
- Important qualification of these people are the ability to see congruence (similarity) between software products and the ability to perceive what is missing from a software product
- With these abilities, the SCM team member can observe how change to the software system is visibly and traceably being controlled
- Should all SCM personnel be skilled programmers and analysts?
- No, these particular skills are not a necessity by any means, although personnel performing software configuration auditing should be technically skilled

SCM Staffing Skills

- Identification
  - Ability to see partitions
  - Ability to see relationships
  - Some technical ability desirable
    - System engineering orientation
    - Programming
- Control
  - Ability to evaluate benefits versus costs
  - System viewpoint (balance of technical / managerial, user / buyer / seller)
  - An appreciation of what is involved in engineering a software change
- Auditing
  - Extreme attention to detail
  - Ability to see congruence
  - Ability to perceive what is missing
  - Extensive experience with technical aspects of system engineering and/or software engineering
- Status Accounting/Reporting
  - Ability to take notes and record data
  - Ability to organize data
  - Some technical familiarity desirable but not required
    - System engineering orientation
    - Programming

Establishment of a CCB

- As a starting point in instituting SCM, periodic CCB meetings provide change control, visibility, and traceability
- The CCB meeting is a mechanism for controlling change during the development and maintenance of software
- CCB membership should be drawn from all organizations on the project
- Decision mechanism
- CCB chairperson
- CCB minutes
- When changes are necessary, a CCB meets to evaluate and manage the impact of change on the software development process
- The impact of change can be countered in only three ways
    - Add more people to the project to reduce the impact of the change
    - Extend the time to completion
    - Eliminate other nonessential or less essential functionality
- If a small amount of code is changed, it is redesigned into the old code; if a large amount is changed, a complete subsystem is redesigned as though it were a new product
- From a maintenance point of view, IBM followed this rule of thumb; “If 20% of the code must be modified, then the module should be redesigned and rewritten”

SCM During the Acceptance Testing Cycle

SCM integrated within the acceptance testing cycle maintains a visible and traceable product ready for delivery to the customer

- Identification
    - Preparation of release notes (lists of changed software)
    - Identification of development baseline
    - Identification of incident reports
    - Identification of operational baseline
- Control
    - CCB meetings
        - Establishment of development baseline
        - Assignment of testing and incident resolution priorities
        - Establishment of turnover dates
        - Approval of audit and test reports
        - Approval of incident report resolutions
        - Establishment of operational baseline
- Auditing
    - Comparison of new baseline to previous baseline
    - Assurance that standards have been met
    - Testing (verification and validation) of software system
- Accounting/Reporting
    - Logging and tracking of incident reports
    - Publication of CCB minutes

Justification and Practicality of Auditing

- Although the auditing consumes the greater part of the SCM budget, it has the potential of preventing the waste of much greater resources

Avoiding the Paperwork Nightmare

- The buyer/user and seller should agree on the paperwork needed to achieve a mutually desirable level of visibility and traceability

Allocating Resources among SCM Activities

- Cost versus benefits must be evaluated for each individual project in determining the allocation of limited SCM resources

Important Issues Relating to SCM

- Following issues should be examined and evaluated in terms of corporate return on investment and employee leverage
  - How long to support a particular version of the software?
  - What upgrade paths should be allowed?
  - How many variations (not versions) of the product should be produced and supported?

Practices for Managing Versions of Software Artifacts

- All source artifacts should be under configuration control
- All artifacts used to produce an artifact of a delivery should be under configuration control
- Work within managed, private workspaces
- Save artifacts at the completion of intermediate steps of a larger change
- Regularly synchronize development with the work of others
- Define policies of branches, codelines, and workspaces
- Codelines
  - Identify how many development codelines are avilable and name each one – typically one
  - Identify how often development codelines must be integrated
  - Identify who can make changes to a codeline, and under what circumstances
  - Identify if parallel or non-parallel development is permitted for each codeline
- Branches
  - Identify the necessary conditions for creating a branch
  - Identify the necessary conditions for merging a branch back into it source codeline
  - Identify the maximum period that an artifact can undergo change without being saved within the configuration management system
- Workspaces
  - Identify who can read and write artifacts of a workspace
  - Identify who can add artifacts to a workspace or delete them from a workspace

Practices for Controlling Changes to Software Artifacts

- Document identified software defects
- Create a defined process for requesting and approving changes
  - Use a change control board
  - Use change packages (aggregate collection of related changes)
- Apply defect repairs to existing releases and ongoing development efforts

Practices for Building Software Systems

- Use shared, static build processes and tools
- Build software on a regular, preferably daily, basis

Practices for Releasing Software Systems

- Maintain a unique read-only copy of each release
- A version manifest should describe each software release
- Software artifacts that comprise a release should adhere to defined acceptance criteria
- Configuration management tools should provide release updates

Practices for Maintaining the Integrity of Software Artifacts

- Use a software tool to perform configuration management functions
- Repositories should exist on reliable physical storage elements
- Configuration management repositories should undergo periodic backups
- Test and confirm the backup process (backup and restore repositories)

Procedure for Creating a CMS

- Acquire highly reliable and redundant physical storage and processing elements for the software repository
- Identify configuration management administrator, who is responsible for numerous tasks. The key tasks include
  - The creation of configuration management accounts and the assignment of capabilities to them
  - The enforcement of defined configuration management policies and procedures
  - The building of internal and external deliveries
- Define a backup procedure to regularly back up configuration management repositories to nonvolatile storage and periodically purge them of redundant or useless data. This procedure should identify when incremental and full backups are done
- Define a procedure that verifies that the backup process functions correctly
- Determine whether work must be authorized. If work must be authorized, then:
  - Establish a change control board
  - Assign people to the change control board
  - Define rules for approving changes to artifacts
- Identify the number of development lines. Typically, one is sufficient. If more than one development line is needed, then:
  - Specify the frequency of the integration of each development line
- Identify the number of new tasks that an individual can work on simultaneously

- Determine whether parallel development is permitted
- Determine if branches can be created for tasks other than parallel development or the development of releases. If branches can be created then:
  - Identify who can create the branches
  - Specify under what conditions the branches can be created
  - Establish the criteria for determining when merges are performed
- Determine who can create workspaces
- Specify standard workspaces
- Identify what information should be specified for each new development task, change request, and anomaly report. Consider performing the following as required actions for each change request and anomaly report
  - Estimate the size of the change
  - Identify any alternative solutions
  - Identify the complexity of the change and the impact on other systems
  - Identify when the need exists
  - Identify the effect the change will have on subsequent work
  - Estimate the cost of the change
  - Identify the criticality of the change request
  - Identify if another change request will solve this problem
  - Identify the effort to verify the change
  - Identify who will verify the change
  - Identify whether the right people are available to work on the request
  - Identify the impact on critical system resources, if this is an issue
  - Identify the length of time that the change request has been pending
- Select metrics to gather
  - Number of change requests submitted
  - Number of change requests reviewed and approved for resolution
  - Number of change requests resolved and length of resolution
  - Number of anomaly reports submitted
  - Number of anomaly reports reviewed and approved for correction
  - Number of anomaly reports corrected and length of correction
  - Number of artifacts changed
  - Number of artifacts changed more than once (these should be characterized by the number of changes and frequency of the changes)
- Acquire a configuration management tool that is able to manage software configurations, document identified software defects, and produce software releases
- Automate the policies, practices, and procedures as much as possible

Best Change Control Practices Industry-Wise

- MIS software projects
- Outsourced software projects
- System software projects
- Commercial software projects
- Military software projects

MIS Software Projects

- Change Control Boards
  - For all projects larger that 5,000 function points
  - CCB usually has three to seven people
- Automated Change Control
  - For all deliverables, which include requirements, specifications, design documents, source code, test plans, user documentation, and training material
  - Package should flag recommended changes
- Function Point Metrics for Changes
  - For projects larger than 15 function points in size, estimate and measure the function totals of all changes to software project
  - These changes include requirements creep, removal (or deferral) of feature, and requirements churn

Outsourced Software Projects

- Larger outsource vendors are often quite expert in implementing change control mechanisms
- Once an application has been deployed, new features and modifications average approximately 7% per year for several years in a row (i.e., new and changed features will approximate 7% of function points)
- Change Estimation and Costing in Contracts
  - Specific clauses for change control are included in the outsource agreements
  - The forms of clauses vary with specific needs of the client, but are often based on the predicted volumes of the changes
  - Initial set of requirements have a fixed price, but new requirements will be included at a higher price
- Function Point Metrics for Changes
  - Estimate and measure the function point totals of all changes to software projects larger than 15 function points
- Change Control Boards
  - For all projects larger that 5,000 function points
  - CCB usually has three to seven people
- Automated Change Control
  - For all deliverables, which include requirements, specifications, design documents, source code, test plans, user documentation, and training material
  - Package should flag recommended changes
  - Automated change control tools that support only source code are adequate for projects larger than 100 function points

Systems Software Projects

- For these kinds of projects, changes during development can occur for a much wider variety of reasons than those found with internal information systems
- Systems software control physical devices
- Change Control Boards

  - For all projects larger that 10,000 function points
  - CCB usually has three to seven people
    - Primary client
    - Project office
    - Development team
    - Hardware portion of the application
- Automated Change Control
  - For all deliverables, which include requirements, specifications, design documents, source code, test plans, user documentation, and training material
  - Package should flag recommended changes
  - Automated change control tools that support only source code are adequate for projects larger than 100 function points
- Function Point Metrics for Changes
  - Estimate and measure the function point totals of all changes to software projects
  - This data can be used for charge-backs and billing, and to ascertain monthly rate of requirements creep
- Function Point Metrics for Changes
  - Estimate and measure the function point totals of all changes to software projects
  - This data can be used for charge-backs and billing, and to ascertain monthly rate of requirements creep
- Cost Estimates for Changes
  - Cost-estimating changes and cost measurement of changes are both difficult
  - Use automated estimation tools and function point metrics
- Requirements Tracing and Changes
  - Each design feature and even each code module is traced back to specific requirement
  - Requirements tracking requires fairly sophisticated automation, and also demand a formal change control board

Commercial Software Projects

- Commercial software vendors may market the same application on different hardware platforms
- They may offer the same application in different national languages
- When major changes occur, they affect dozens of versions at the same time
- Change control is a key technology for commercial software vendors
- Automated Change Control

Military Software Projects

- The military software community was an early adopter of change control packages
- Change control starts during initial development and continues until an application is retired
- Change Control Boards
  - For all projects larger that 10,000 function points
  - CCB usually has three to seven people

    - Primary client
    - Project office
    - Development team
    - Hardware portion of the application for hybrid projects
- Automated Change Control
    - For all deliverables, which include requirements, specifications, design documents, source code, test plans, user documentation, and training material
    - Package should flag recommended changes
    - This is one of the 16 best practices identified by the Airlie Council
- Function Point Metrics for Changes
    - Estimate and measure the function point totals of all changes to military projects
    - This data can be used to ascertain the monthly rate of requirements creep
- Cost Estimates for Changes
    - Cost-estimating changes and cost measurement of changes are both difficult
    - Use automated estimation tools and function point metrics
- Requirements Tracing and Changes
    - Each design feature and even each code module is traced back to specific requirement
    - Requirements tracking requires fairly sophisticated automation, and also demand a formal change control board
    - Change control in the military domain is not limited only to software changes

SCM Plan and SQA Plan - IEEE SCM Plan

- Introduction
- Reference Documents, Definitions, Acronyms
- Management
- Activities
- Resources
- Plan Maintenance
- Plan Approval

Introduction

- Introduction information provides a simplified overview of the configuration management activities so that those approving, performing, or interacting with the Software Configuration Management Plan can obtain a clear understanding of the SCM Plan.
- Explains the purpose and content of the Configuration Management Plan (i.e., the methodical storage and recording of all software components and deliverables during development)
- The SCM Plan documents methods to be used for the identification of software items, control and implementation of change, and recording and reporting change implementation status
- The plan should describe methods used for:
    - Identification of software configuration items,
    - Control and implementation of change
    - Recording and reporting change and problem report implementation status
    - Conducting configuration audits

- Review and approval cycles as well as approval authority, and
- Identification of personnel responsible for configuration management

Reference Documents, Definitions, Acronyms

- This section provides a complete list of documents referenced elsewhere in the text of the SCM Plan. By definition, these documents originate outside the project. Also included in this section is a glossary of project specific terms and their definitions and a list of project-specific abbreviations and acronyms and their meaning
- Reference Documents
- Glossary of Terms
- Abbreviations and Acronyms

Management

- This section provides information describing the allocation of responsibilities and authorities for software configuration management activities to organizations and individuals within the project structure
- Organization
  - This section depicts the organizational context, both technical and managerial, within which the prescribed software configuration management activities are to be implemented
- Responsibilities
  - Describes the allocation of software configuration management activities to organizational units
- Policies, Directives and Procedures
  - Any external constraints, or requirements, placed on the SCM Plan by other policies, directives, or procedures must be identified here. A detailed impact analysis should accompany the identification of external constraints

Activities

- Identifies all functions and tasks required to manage the configuration of the software system as specified in the scope of the SCM Plan. Both technical and managerial activities must be identified
- Configuration Identification
  - Identify, name, and describe the documented physical and functional characteristics of the code, specification, design, and data elements to be controlled for the project. The Plan must identify the items to be maintained in configuration management control
  - Identifying configuration items
  - Naming configuration items
  - Acquiring configuration items
- Configuration Control
  - Configuration control activities request, evaluate, approve or disapprove, and implement changes to the software configuration items. Changes include both error correction and enhancement. This section shall identify the records to be used for tracking and documenting the sequence of steps for each change

- Configuration Control (cont'd)
  - Requesting changes
  - Evaluating changes
  - Approving or disapproving changes
  - Implementing changes
- Configuration Status Accounting
  - Record and report the status of configuration items. The following minimum data elements should be tracked and reported for each configuration management item:
    - Approved version
    - Status of requested changes
    - Implementation status of approved changes
- Configuration Audits and Reviews
  - Configuration audits determine the extent to which the actual configuration management items reflect the required physical and functional characteristics. Configuration reviews may also be designed a management tool used to ensure that a software configuration management baseline is established
- Interface Control
  - Coordinates changes to the project's configuration management items with changes to interfacing items outside the scope of the SCM Plan
- Subcontractor/Vendor Control
  - For acquired software, the Software Configuration Management Plan shall describe how the vendor software will be received, tested, and placed under software configuration management control
  - For both subcontracted and acquired software, the SCM Plan must define the activities to be performed to incorporate the externally developed items into project configuration management and to coordinate changes to these items

Resources

- Establishes the sequence and coordination for all the software configuration management activities and all the events affecting the Plan's implementation
- Schedules
  - Schedule information shall be expressed as absolute dates, as dates relative to other project activities, as project milestones, or as a simple sequence of events. Graphic representation can be particularly appropriate for conveying this information
- Resources
  - Identifies the software tools, techniques, equipment, personnel, and training necessary for the implementation of software configuration management activities

Plan Maintenance

- Identifies and describes the activities and responsibilities necessary to ensure continued software configuration management planning during the life cycle of the project. This

section of the SCM Plan should state the following:
- Who is responsible for monitoring the SCM Plan
- How frequently updates are to be applied
- How changes to the SCM Plan are to be evaluated and approved
- How changes to the SCM Plan are to be made and communicated

References for SCM Plan

- SCM Plan based on IEEE Standard for Software Configuration Management Plans (Std 828-1990) and the IEEE Guide to Software Configuration Management (Std 1042-1987)
- Managing the Software Process by Watts S. Process, AW, 1989, (Chapter 12.1) [pp.228-232]

Software Quality Assurance Plan (SQAP)

- Every development and maintenance project should have a software quality assurance plan (SQAP) that specifies:
  - Its goals
  - The SQA tasks to be performed
  - The standards against which the development work is to be measured
  - The procedures and organizational structure

IEEE SQAP Standard

- Purpose
- Reference documents
- Management
- Documentation
- Standards, practices, and conventions
- Reviews and audits
- Software configuration management
- Test
- Problem reporting and corrective action
- Tools, techniques, and methodologies
- Media control
- Supplier control
- Records collection, maintenance, and retention

Purpose

- This section documents the purpose of this Software Quality Assurance (SQA) Plan
- It documents the goals, processes, and responsibilities required to implement effective quality assurance functions for the current project
- Scope
  - Defines the scope of the SQAP in different activities of the software life cycle and even through maintenance

Reference Documents

- Lists the documents referenced in this software quality assurance plan

Management

- This section describes the management organizational structure, its roles and responsibilities, and the software quality tasks to be performed
- Management organization
- Tasks
- Roles and responsibilities
- Software assurance estimated resources
- Management Organization
  - Describes the support of entities, organizations and personnel
  - Relevant entities/roles that are of interest and applicable to this SQA Plan and the software assurance effort specially include
    - Project office
    - Assurance management office
- Tasks
  - This section summarizes the tasks (product and process assessments) to be performed during the development, operations, and maintenance of software
  - These tasks are selected based on the developer's Project Schedule, Software Management Plan (SMP) (and/or Software Maintenance Plan) and planned deliverables, contractual deliverables, and identified reviews
- Tasks (cont'd)
  - Product assessments
    - Peer Review packages
    - Document Reviews
    - Software Development Folders
    - Software Configuration Management
    - Test results
  - Process assessments
    - Project Planning
    - Project Monitoring and Control
    - Measurement and Analysis
    - System/Subsystem Reviews
    - Peer Reviews
    - Requirements Management
    - Software Configuration Management and Configuration Audits
    - Test Management (Verification & Validation)
    - Software Problem Reporting and Corrective Action
    - Risk Management
    - Supplier Agreement Management
- Roles and Responsibilities
  - This section describes the roles and responsibilities for each assurance person assigned to a project
- Software Assurance Manager
- Software Quality Personnel
- Software Assurance Estimated Resources

  - Staffing to support software assurance (i.e., quality, safety, and reliability) activities must be balanced against various project characteristics and constraints, including cost, schedule, maturity level of the providers, criticality of the software being developed, return on investment, perceived risk, etc.

Documentation

- This section identifies the minimum documentation governing the requirements, development, verification, validation, and maintenance of software that falls within the scope of this software quality plan
- Each document below shall be assessed (reviewed) by SQ personnel
  - The software requirements specifications
  - The software design description
  - The software verification and validation report
  - User documentation

Standards, Practices, and Conventions

- This section highlights the standards, practices, quality requirements, and metrics to be applied to ensure a successful software quality program
- This section specifies a minimum content of:
  - Documentation standards
  - Logic structure standards
  - Coding standards
  - Commentary standards
- Software Quality Program
- Standard Metrics

Reviews and Audits

- This section discusses major project reviews conducted by SQA staff and software team members
- Generic review guidelines
  - A set of guidelines for all formal technical reviews or inspections is presented in this section
  - Conducting a review
    - General guidelines for conducting a review
  - Roles and responsibilities
    - The roles people play during a FTR or inspection and the responsibilities of each player
  - Review work products
    - Documents, forms, lists produced as a consequence of the FTR/inspection
- Formal technical reviews/inspections
  - A description of the specific character and the intent of each major FTR/inspection conducted during the software process

- SQA audits
  - A description of audits performed by the SQA group with the intent of assessing how well SQA and software engineering activities are being conducted on a project

Software Configuration Management

- A brief overview of the content of the software configuration management (SCM) plan is presented here
- Alternatively, the SCM plan is referenced

Test

- SQA personnel will assure that the test management processes and products are being implemented per the Software Management Plan and /or Test Plans
- This includes all types of testing of software system components as described in the test plan, specifically during integration testing (verification) and acceptance testing (validation).

Problem Reporting and Corrective Action

- This section describes problem reporting mechanisms that occur as a consequence of the formal technical reviews' or inspections' that are conducted and the means for corrective action and follow-up
- Reporting mechanisms
  - Describes how and to whom problems are reported
- Responsibilities
  - Describes who has responsibility for corrective actions and follow-up
- Data collection and evaluation
  - Describes the manner in which error/defect data are collected and stored for future or real-time evaluation
- Statistical SQA
  - Describes the quantitative techniques that will be applied to error/defect data in an effort to discern trends and improvement

Tools, Techniques, and Methodologies

- Specialized tools, techniques, and methods to be used by the SQA group are described in this section

Media Control

Supplier Control

- SQA personnel will conduct off-site surveillance activities at supplier sites on software development activities
- SQA personnel will conduct a baseline assessment of the suppliers' Quality Management Systems (QMS) to ensure that the suppliers' have quality processes in place. This initial assessment will help to scope the level of effort and follow-on activities

in the area of software quality assurance

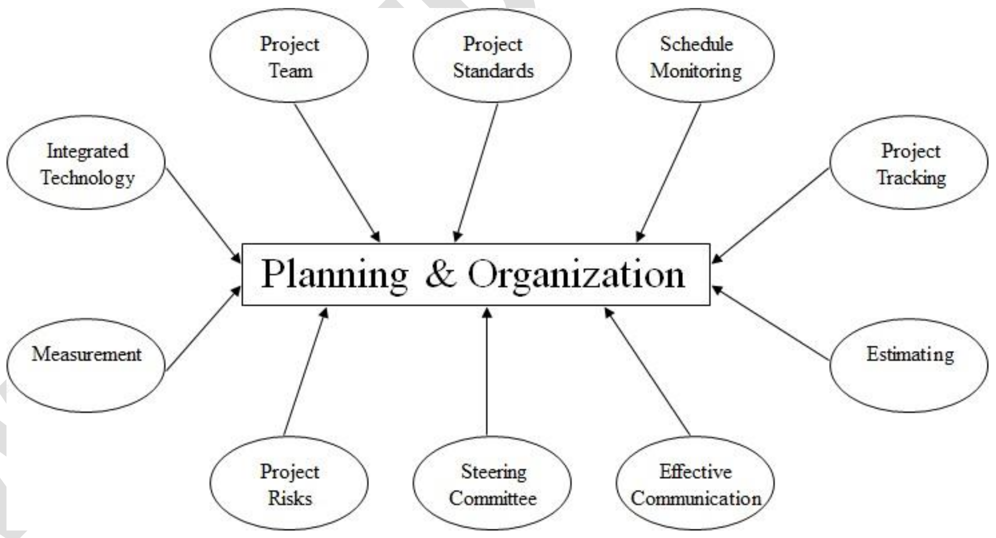
Record Collection, Maintenance, and Retention

- SQA personnel will maintain records that document assessments performed on the project. Maintaining these records will provide objective evidence and traceability of assessments performed throughout the project's life cycle
- Example records include the process and product assessments reports, completed checklists, the SQA Activity Schedule, metrics, weekly status reports, etc.

Software Process Assurance

- Process assurance makes certain that the process for building and delivering software is robust and allows for the delivery and maintenance of the products
- Process assurance consists of the collective activities carried out while developing a product to ensure that the methods and techniques used are integrated, consistent, and correctly applied
- Emphasis is given to cost, time, technical requirements, testing measurements, and prototyping
- Process assurance involves the interrelationships of several different components. Depending on how these are managed, they can have a major positive impact on the products
- Once an effective process assurance program is put in place and shown to be beneficial, then emphasis can be placed in making verification and validation strategies effective and in improving the quality of the products
- Successful process assurance is based on planning and organization
- There are several important aspects of planning and organization that must be considered before starting the project
- I'll show you a picture, which captures the components of planning and organization

Components of Planning and Organization



Project Team

- Project team is the project manager's only means of reaching the project goals
- Formation of project team is vital to success
- Size of the team depends on the size and complexity of the project
- Right mix of technical knowledge and experience
- Fostering of mutual respect within team and maintains good morals

Project Standards

- Before the project is started, standards should be established for activities like requirements gathering, design development, and unit testing
- Standards should also be developed for quality control activities, like walkthroughs, reviews, and inspections
- Many companies follow IEEE software engineering standards or they have their internally developed standards
- Standards should be flexible enough to be applied to large or small projects
- Any deviations from the standards should be approved by the project team and the reason for such deviation should be noted in the minutes of the project meetings

Schedule Monitoring

- Stringent deadlines for the project are frequently established by management, end users, a project sponsor, or a client with no regard to the reality of achievement
- The project manager is then designated to meet unrealistic expectations of the project completion date
- For this reason, the project start date, milestones, and completion date should be negotiated upfront
- If the unrealistic date is accepted and the project activities are then made to fit within this time frame, the quality of the project certainly will suffer
- The key to an “on-time” project lies in the ability to identify the critical path before starting the project
- The critical path of a project is where problems that may affect the overall schedule are faced
- Develop systematic work breakdown structures which identify task groupings (tasks that can be combined together), task sequences, and entrance/exit criteria for each task
- To define tasks, follow the guidelines of the system development methodology used by your organization
- In the absence of a development methodology, obtain copies of task lists and task dependencies from other projects and customize them to suit your needs of the current project
- Clearly defined work breakdown structures will assist in selecting the correct skilled resources
- At the same time, using the breakdown structures also ensures that no activity is forgotten
- The technique of breaking down activities into smaller tasks takes an impossibly complex project and reorganizes it into manageable pieces under the direction of the project manager
- Once you have defined the critical path, review the tasks and schedule with the project team members and other significantly impacted individuals
- Since these people are the stakeholders and are affected by the project in one or more of the following ways:
  - Their budget is charged for all or part of the project
  - The department’s resources are used by the project

    - The department has either existing projects or ongoing projects that are affected by the new project
- Avoid the most common mistake of adding another resource to shorten or meet the schedule

Project Tracking

- Project tracking is an important activity in project management
- Project tracking tools should be used to monitor the progress of project schedule, and estimate resources

Estimation

- Realistic estimates allow you to discuss alternative approaches at the start of the project
- Estimates are not foolproof
- Allow time for resource management and unforeseen events, like the illness of a team member
- Revise estimates, and update plans

Effective Communication

- Effective communication between the management and project team is a critical interpersonal skill
    - Listening
    - Observing
    - Giving guidance
- Delegation of responsibility
- Informal negotiation

Steering Committee

- A committee responsible for defining project policy, reviewing the project milestones, and evaluating risk factors must be established
- Members of the committee should represent all the impacted areas of the business. They should be knowledgeable enough to make informed technological decisions and be able to change the course if needed
- It is responsible for
    - Estimating the time that will be required to maintain the system
    - Deciding on the type of support required from the operations for the running of the system
    - Deciding when the data will be available and how it will be managed, reported, and used
    - Forming a configuration control board (CCB) that manages the impact of changes

Project Risks

- Every project has risks associated with them, some more than others
- There is need to identify and address the risk factors upfront

- All risk factors should be discussed with the project team, management, and users
- A risk mitigation policy needs to be developed
- Every project has risks associated with them, some more than others
- There is need to identify and address the risk factors upfront
- All risk factors should be discussed with the project team, management, and users
- A risk mitigation policy needs to be developed
- Risks can be minimized by
  - Implementing controls from the initiation stage and by ensuring pre-established development standards are followed
  - Providing project management training
  - Reducing the scope of the project by incremental development or phased development
- Technical risk is encountered when the project team utilizes a new technology like new hardware or new development methodology for the first time
- Technical risks can be controlled by
  - Appointing a qualified technical project leader
  - Implementing a strong, independent quality control group to evaluate the progress of the project and project deliverable
  - Getting additional technical expertise from outside consultants who have expertise and the knowledge to make a difference in the ultimate quality of the project

Measurement

- Establishing measurement criteria, against which each phase of the project will be evaluated, is vital
- When exit criteria is well defined, it is sufficient to evaluate the outcome of each phase against the exit criteria and move forward
- If the outcome of each phase does not meet the performance criteria, the project manager should be able to control the project by evaluating the problems, identifying the deviations, and implementing new processes to address the deviations
- The pre-established quality goals for the project can also serve as criteria against which the project can be measured
- Processes should be established to
  - Enable the organization to address customer complaints
  - Give the organization statistics regarding the types of customer calls
  - Incorporate reporting and handling of customer problems
  - Enable management to make staffing decisions based on the number of customer calls

Integrated Technology

- Strategy for Integrated Technology should be considered by management in relation to the other business needs
- This will empower the management to react to the operational needs of the business and, at the same time, take an inventory of the current status of various systems,

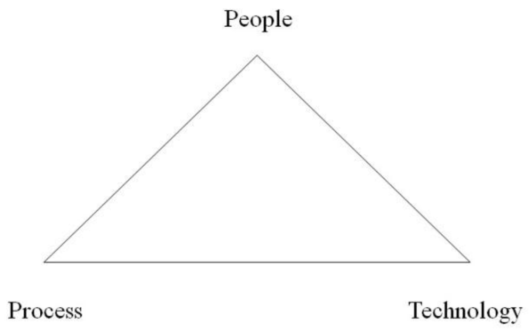
projects, and the ability of technical staff to support any future projects
- The IT trends, competitors, and demands of the customers should be visible to the management
- Parts of the new system that will be interfacing with existing system should be identified so that the impact can be evaluated
- If technology is new and not well understood, allowances to incorporate experiments should be made in the overall project plan and schedule

Causes of Failure in Process Assurance
- Lack of Management Support
- Lack of User Involvement
- Lack of Project Leadership
- Lack of Measures of Success

Symptoms of Process Failure
- Commitments consistently missed
  - Late delivery
  - Last minute crunches
  - Spiraling costs
- No management visibility into progress
  - You're always being surprised
- Quality problems
  - Too much rework
  - Functions do not work correctly
  - Customer complaints after delivery
- Poor morale
  - People frustrated
  - Is anyone in charge?

Common Misconceptions
- I don't need process, I have
  - Really good people
  - Advanced technology
  - An experience manager
- Process
  - Interfaces with creativity
  - Equals bureaucracy + regimentation
  - Isn't needed when building prototypes
  - Is only useful on large projects
  - Hinders agility in fast-moving markets
  - Costs too muc
- Everyone realizes the importance of having a motivated, quality work force but even our finest people cannot perform at their best when the process is not understood or operating at its best
- Process, people, and technology are the major determinants of product cost, schedule, and cost

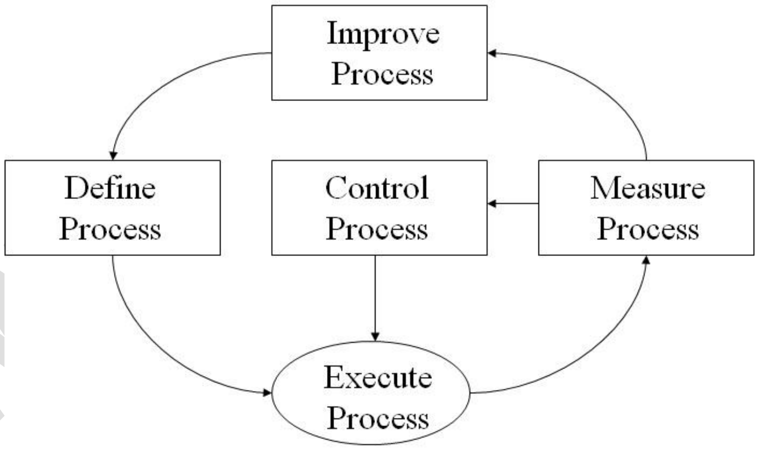
Process, People, and Technology



- The quality of a system is highly influenced by the quality of the process used to acquire, develop, and maintain it
- While process is often described as a node of the process-people-technology triad, it can also be considered the “glue” that ties the triad together

Process Management and Improvement – CMM

Process Management Responsibilities
- Define the process
- Measure the process
- Control the process
  - Ensure variability is stable. Why?
- Improve the process



Objectives of Process Definition
- Design processes that can meet or support business and technical objectives
- Identify and define the issues, models, and measures that relate to the performance of the processes
- Provide infrastructures (the set of methods, people, and practices) that are needed to support software activities
- Ensure that the software organization has the ability to execute and sustain the processes

- Skills
- Training
- Tools
- Facilities
- Funds

Objectives of Process Measurements

- Collect the data that measure the performance of each process
- Analyze the performance of each process
- Retain and use data:
  - To assess process stability and capability
  - To interpret the results of observations and analyses
  - To predict future costs and performance
  - To provide baselines and benchmarks
  - To plot trends
  - To identify opportunities for improvements

Objectives of Process Control

- Controlling a process means keeping within its normal (inherent) performance boundaries – that is, making the process behave consistently
- Measurement
  - Obtaining information about process performance
- Detection
  - Analyzing the information to identify variations in the process that are due to assignable causes
- Correction
  - Taking steps to remove variation due to assignable causes from the process and to remove the results of process drift from the product
- Determine whether or not the process is under control (is stable with respect to the inherent variability of measured performance)
- Identify performance variations that are caused by process anomalies (assignable causes)
- Estimate the sources of assignable causes so as to stabilize the process
- Once a process is under control, sustaining activities must be undertaken to forestall the effects of entropy. Without sustaining activities, processes can easily fall victim to the forces of ad hoc change or disuse and deteriorate to out-of-control states
- This requires reinforcing the use of defined processes through continuing management oversight, measurement, benchmarking, and process assessments

Objectives of Process Improvement

- Understand the characteristics of existing processes and the factors that affect process capability
- Plan, justify, and implement actions that will modify the processes so as to better meet business needs
- Assess the impacts and benefits gained, and compare these to the costs of changes made to the processes

- Process improvement should be done to help the business – not for its own sake

Identifying Process Issues

- Clarify your business goals or objectives
- Identify the critical processes
- List the objectives for each critical process
- List the potential problem areas associated with the processes
- Group the list of potential problems into common areas or topics

Common Process Issues

- Product quality
- Process duration
- Product delivery
- Process cost

Steps before Process Improvements

- Explain the problem, discuss why the change is necessary, and spell out the reasons in terms that are meaningful
- Create a comfortable environment where people will feel free to openly voice their concerns and their opinions
- Explain the details of the change, elaborate on the return on investment, how it will effect the staff, and when the change will take place
- Explain how the change will be implemented and measured
- Identify the individuals who are open-minded to accept the change more easily
- Train employees to help them acquire needed skills
- Encourage team work at all times and at all levels
- Address each concern with care so there is no fear left and value each opinion
- Make decisions based on factual data rather than opinions or gut feelings
- Enforce decisions to reinforce the change

Seven Steps of the Process Improvement

- Plan
- Gather data
- Analyze findings
- Describe the ideal process
- Implement the ideal process
- Measure progress
- Standardize the process

Useful Tips on SPI

- Proactively identify and seek support of process-improvement champions and sponsor
- Reinforce management awareness and commitment with a strong business case for each desired process improvement
- Build an infrastructure strong enough to achieve and hold software core competence
- Measure the extent of adoption of each desired process improvement until it is effectively, efficiently, and across all appropriate parts of the organization

Process Improvement Programs

- Capability Maturity Model (CMM)
- CMMI
- ISO 9000
- Tick/IT
- Spice
- Total Quality Management (TQM)

What is a Process Model?

- A process model is a structured collection of practices that describe the characteristics of effective processes
- Practices included are those proven by experience to be effective

How is a Process Model Used?

- A process model is used
  - To help set process improvement objectives and priorities
  - To help ensure stable, capable, and mature processes
  - As a guide for improvement of project and organizational processes
  - With an appraisal method to diagnose the state of an organization's current practices

Why is a Process Model Important?

- A process model provides
  - A place to start improving
  - The benefit of a community's prior experiences
  - A common language and a shared vision
  - A framework for prioritizing actions
  - A way to define what improvement means for an organization
- Process improvement should be done to help the business – not for its own sake
- In God we trust, all others bring data
  - W. Edwards Deming

Software State-of-the-Art in 1984

- More than half of the large software systems were late in excess of 12 months
- The average costs of large software systems was more than twice the initial budget
- The cancellation rate of large software systems exceeded 35%
- The quality and reliability levels of delivered software of all sizes was poor
- Software personnel were increasing by more than 10% per year
- Software was the largest known business expense which could not be managed

Software Engineering Institute

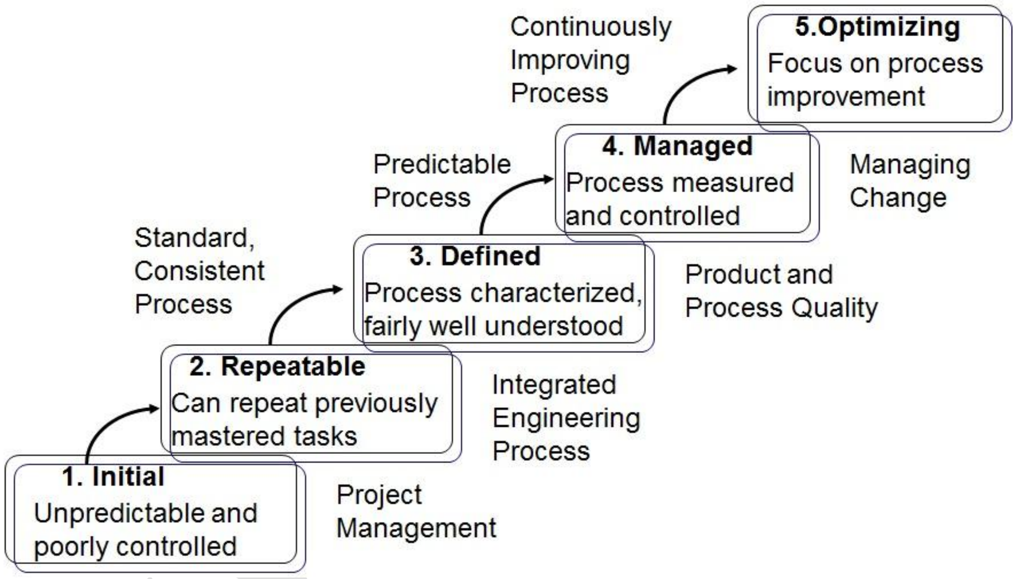
- A research facility, located in University of Carnegie Mellon, Pennsylvania
- Primarily funded by US DoD to explore software issues, and especially topics associated

with defense contracts
- US DoD is the largest producer and consumer of software in the world

Capability Maturity Model

- SEI developed a Capability Maturity Model (CMM) for software systems and an assessment mechanism
- CMM has five maturity models
  - Initial
  - Repeatable
  - Defined
  - Managed
  - Optimizing

The Five Levels of Software Process Maturity


Continuously Improving Process
5.Optimizing
Focus on process improvement
Predictable Process
4. Managed
Process measured and controlled
Managing Change
Standard, Consistent Process
3. Defined
Process characterized, fairly well understood
Product and Process Quality
2. Repeatable
Can repeat previously mastered tasks
Integrated Engineering Process
1. Initial
Unpredictable and poorly controlled
Project Management

CMM Level 1: Initial
- Organizations are characterized by random or chaotic development methods with little formality and uninformed project management
- Small projects may be successful, but larger projects are often failures
- Overall results are marginal to poor
- In terms of People CMM, level 1 organizations are deficient in training at both the technical staff and managerial levels
- SEI does not recommend any key process areas

CMM Level 2: Repeatable
- Organizations have introduced at least some rigor into project management and technical development tasks
- Approaches such as formal cost estimating are noted for project management, and formal requirements gathering are often noted during development

- Compared to initial level, a higher frequency of success and a lower incidence of overruns and cancelled projects can be observed
- In terms of People CMM, level 2 organizations have begun to provide adequate training for managers and technical staff
- Become aware of professional growth and the need for selecting and keeping capable personnel
- Key process areas
  - Requirements management
  - Software project planning
  - Software project tracking and oversight
  - Software subcontract management
  - Software quality assurance
  - Software configuration management
- Key process areas for People CMM
  - Compensation
  - Training
  - Staffing
  - Communication
  - Work environment

CMM Level 3: Defined

- Organizations have mastered a development process that can often lead to successful large systems
- Over and above the project management and technical approached found in Level 2 organizations, the Level 3 groups have a well-defined development process that can handle all sizes and kinds of projects
- In terms of People CMM, the organizations have developed skills inventories
- Capable of selecting appropriate specialists who may be needed for critical topics such as testing, quality assurance, web mastery, and the like
- Key process areas
  - Organization process focus
  - Organization process definition
  - Training
  - Software product engineering
  - Peer reviews
  - Integrated software management
  - Inter-group coordination

- Key process areas for People CMM
  - Career development
  - Competency-based practices
  - Work force planning
  - Analysis of the knowledge and the skills needed by the organization

CMM Level 4: Managed

- Organizations have established a firm quantitative basis for project management and utilize both effective measurements and also effective cost and quality estimates
- In terms of People CMM, organizations are able to not only monitor their need for specialized personnel, but are actually able to explore the productivity and quality results associated from the presence of specialists in a quantitative way
- Able to do long-range predictions of needs
- Mentoring
- Key process areas
  - Software quality management
  - Quantitative software management
- Key process areas for People CMM
  - Mentoring
  - Team building
  - Organizational competency
  - Ability to predict and measure the effect of specialists and teams in quantitative manner

CMM Level 5: Optimizing

- Organizations are assumed to have mastered the current state-of-the-art of software project management and development
- In terms of People CMM, the requirements are an extension of the Level 4 capabilities and hence different more in degree than in kind
- Stresses both coaching and rewards for innovation
- Key process areas
  - Defect prevention
  - Technology change management
  - Process change management
- Key process areas for People CMM
  - Encouragement of innovation
  - Coaching
  - Personal competency development

Levels Process Categories	Management	Organizational	Engineering
5 Optimizing		Technology Change Management Process Change Management	Defect Prevention
4 Managed	Quantitative Software Management		Software Quality Management
3 Defined	Integrated Software Management Intergroup Coordination	Organization Process Focus Organization Process Definition Training Program	Software Product Engineering Peer Reviews

2 Repeatable	Requirements Management Software Project Planning Software Project Tracking and Oversight Software Subcontract Management Software Quality Assurance Software Configuration Management		
1 Initial		Ad Hoc Processes	

Level 1 Quality

- Software defect potentials run from 3 to more than 15 defects per function points, but average is 5 defects per function point
- Defect removal efficiency runs from less than 70% to more than 95%, but average is 85%
- Average number of delivered defects is 0.75 defects per function point
- Several hundred projects surveyed

Level 2 Quality

- Software defect potentials run from 3 to more than 12 defects per function points, but average is 4.8 defects per function point
- Defect removal efficiency runs from less than 70% to more than 96%, but average is 87%
- Average number of delivered defects is 0.6 defects per function point
- Fifty (50) projects surveyed

Level 3 Quality

- Software defect potentials run from 2.5 to more than 9 defects per function points, but average is 4.3 defects per function point
- Defect removal efficiency runs from less than 75% to more than 97%, but average is 89%
- Average number of delivered defects is 0.47 defects per function point
- Thirty (30) projects surveyed

Level 4 Quality

- Software defect potentials run from 2.3 to more than 6 defects per function points, but average is 3.8 defects per function point
- Defect removal efficiency runs from less than 80% to more than 99%, but average is 94%
- Average number of delivered defects is 0.2 defects per function point
- Nine (9) projects surveyed

Level 5 Quality

- Software defect potentials run from 2 to 5 defects per function points, but average is 3.5 defects per function point
- Defect removal efficiency runs from less than 90% to more than 99%, but average is 97%
- Average number of delivered defects is 0.1 defects per function point
- Four (4) projects surveyed

Capability Maturity Model Integration (CMMI)

- Capability Maturity Model Integration (CMMI) is a process improvement approach that provides organizations with the essential elements of effective processes
- It can be used to guide process improvement across a project, a division, or an entire organization
- CMMI helps integrate traditionally separate organizational functions, set process improvement goals and priorities, provide guidance for quality processes, and provide a point of reference for appraising current processes
- The CMMI product suite is at the forefront of process improvement because it provide the latest best practices for product and service development and maintenance
- The CMMI models improve the best practices of previous models in many important ways

Benefits of CMMI

- CMMI best practices enable organizations to do the following
  - More explicitly link management and engineering activities to their business objectives
  - Expand the scope of and visibility into the product lifecycle and engineering activities to ensure that the product or service meets customer expectations
  - Incorporate lessons learned from additional areas of best practices (e.g., measurement, risk management, and supplier management)
  - Implement more robust high-maturity practices
  - Address additional organizational functions critical to their products and services
  - More fully comply with relevant ISO standards

Background of CMMI

- CMM-SW: CMM Software
- SE CMM: System Engineering CMM
- SA CMM: Software Acquisition CMM
- IPD CMM: Integrated Product Development Team CMM
- System Engineering Capability Assessment Model (SECAM)
- System Engineering Capability Model (SECM)
- Use CMMI in process improvement activities as a
  - Collection of best practices
  - Framework for organizing and prioritizing activities
  - Support for the coordination of multi-disciplined activities that might be required

    - to successfully build a product
    - Means to emphasize the alignment of the process improvement objectives with organization business objectives

- A CMMI model is not a process
- CMMI is a collection of best practices from highly functioning organizations collected to help you improve your processes by describing *what* things or activities should be done in your organization
- A CMMI model describes the characteristics of effective processes
- All of the source models for CMMI are considered capability maturity models; however, each has a different approach. Review and examination of each source model led to the discovery of two types of approaches to presenting capability maturity models. These types of approaches have been given the label "representations" in the process improvement community. A representation reflects the organization, use, and presentation of components in a model
- All capability maturity models have process areas that are defined by levels. An example of a process area is Project Planning. There are two types of CMMI model representations: staged and continuous.
- Two of the source models use other terms for the concept of a process area. The Software CMM uses the term key process areas; the SECM uses the term focus areas.

Staged Representation

- The staged representation is the approach used in the Software CMM. It is an approach that uses predefined sets of process areas to define an improvement path for an organization. This improvement path is described by a model component called a maturity level
- A maturity level is a well-defined evolutionary plateau toward achieving improved organizational processes

Continuous Representation

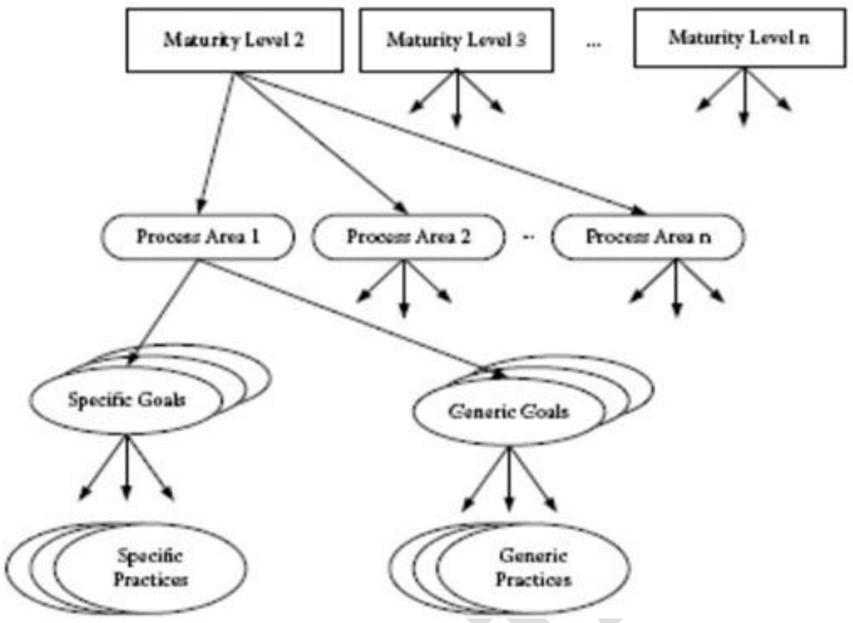
- The continuous representation is the approach used in the SECM and the IPD-CMM. This approach allows an organization to select a specific process area and improve relative to it
- The continuous representation uses capability levels to characterize improvement relative to an individual process area

CMMI Model Structure

- Maturity Levels (staged representation) or Capability Levels (continuous representation)
- Process Areas
- Goals – Generic and Specific
- Practices – Generic and Specific

Staged Representation

CMMI Model Components in the Staged Representation



- The staged representation offers a systematic, structured way to approach process improvement one step at a time. Achieving each stage ensures that an adequate improvement has been laid as a foundation for the next stage
- Process areas are organized by maturity levels that take much of the guess work out of process improvement. The staged representation prescribes the order for implementing each process area according to maturity levels, which define the improvement path for an organization from the initial level to the optimizing level. Achieving each maturity level ensures that an adequate improvement foundation has been laid for the next maturity level and allows for lasting, incremental improvement

Maturity Level

- Maturity level signifies the level of performance that can be expected from an organization
- There are five maturity levels
- Adhoc
- Managed
- Defined
- Quantitatively Managed
- Optimizing

Process Areas

- Each maturity level consists of several process area
- A process area is a group of practices or activities performed collectively in order to achieve a specific objective

Goals

- Each PA has several goals that need to be satisfied in order to satisfy the objectives of the PA. There are two types of goals:
  - Specific goals (SG): goals that relate only to the specific PA under study
  - Generic goals (GG): goals that are common to multiple process areas throughout the model. These goals help determine whether the PA has been institutionalized

Practices

- Practices are activities that must be performed to satisfy the goals for each PA. Each practice relates to only one goal. There are two types of practices:
  - Specific practices (SP): practices that relate to specific goals
  - Generic practices (GP): practices associated with the generic goals for institutionalization

Level 1: Adhoc

Level 2: Defined

- Adhering to organizational policies
- Following a documented plan and process description
- Applying adequate funding and resources
- Maintaining appropriate assignment of responsibility and authority
- Training people in their appropriate processes
- Placing work products under appropriate configuration management
- Monitoring and controlling process performance, and taking corrective action
- Objectively reviewing the process, work products, and services, and addressing noncompliance
- Reviewing the activities, status, and results of the process with appropriate levels of management, and taking corrective action
- Identifying and interacting with relevant stakeholders

Level 3: Managed

- The organization has achieved all of the goals of Level 2. There is an organizational way of doing business, with tailoring of this organizational method allowed under predefined conditions. The organization has an organization's set of standard processes (OSSP)
- The following characteristics of the process are clearly stated
  - Purpose
  - Inputs
  - Entry criteria
  - Activities
  - Roles
  - Measures
  - Verification steps
  - Outputs
  - Exit criteria
- Level 3 continues with defining a strong, meaningful, organization-wide approach to developing products. An important distinction between Level 2 and Level 3 is that at Level 3, processes are described in more detail and more rigorously than at Level 2. Processes are managed more proactively, based on a more sophisticated understanding of the interrelationships and measurements of the processes and parts of the processes.

Level 3 is more sophisticated, more organized, and establishes an organizational identity—a way of doing business particular to this organization

Level 4: Quantitatively Managed

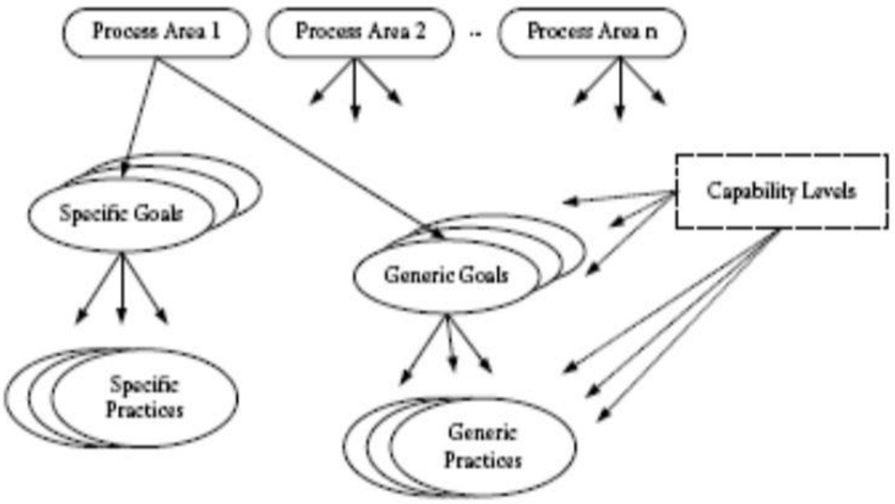
- The organization controls its processes by statistical and other quantitative techniques. Product quality, process performance, and service quality are understood in statistical terms and are managed throughout the life of the processes

Level 5: Optimizing

- Processes are continually improved based on an understanding of *common* causes of variation within the processes
- Level 5 is nirvana
- Everyone is a productive member of the team, defects are reduced, and your product is delivered on time and within the estimated budget

Continuous Representation

CMMI Model Components in the Continuous Representation



- The continuous representation offers a flexible approach to process improvement. An organization may choose to improve the performance of a single process-related trouble spot, or it can work on several areas that are closely aligned to the organization's business objectives. The continuous representation also allows an organization to improve different processes at different rates. There are some limitations on an organization's choices because of the dependencies among some process areas

Continuous Representation

- The continuous representation uses the same basic structure as the staged representation.
- However, each PA belongs to a Process Area Category. A Process Area Category is just a simple way of arranging PAs by their related, primary functions

- Capability levels are used to measure the improvement path through each process area from an unperformed process to an optimizing process. For example, an organization may wish to strive for reaching capability level 2 in one process area and capability level 4 in another. As the organization's process reaches a capability level, it sets its sights on the next capability level for that same process area or decides to widen its scope and create the same level of capability across a larger number of process areas

Process Area Categories

- Process management
  - Organizational Process Focus
  - Organizational Process Definition (with Integrated Product and Process Development—IPPD)
  - Organizational Training
  - Organizational Process Performance
  - Organizational Innovation and Deployment
- Project management
  - Project Planning
  - Project Monitoring and Control
  - Supplier Agreement Management
  - Integrated Project Management (with Integrated Product and Process Development—IPPD)
  - Risk Management
  - Quantitative Project Management
- Engineering
  - Requirements Development
  - Requirements Management
  - Technical Solution
  - Product Integration
  - Verification
  - Validation
  - (listed in increasing order of complexity)
  - Support
- Support
- Configuration Management
- Process and Product Quality Assurance
- Measurement and Analysis
- Decision Analysis and Resolution
- Causal Analysis and Resolution

Goals and Practices

- Specific goals and practices relate to specific process areas and relate to tasks that make sense for that process area only. For example, Project Planning requires a project plan. Quantitative Project Management requires a process performance baseline
- Generic goals and practices relate to multiple process areas.

- CMMI focuses on institutionalization. Goals cannot be achieved without proving institutionalization of the process. Generic goals and generic practices support institutionalization and increasing sophistication of the process. Specific goals and specific practices support implementation of the process area. Process maturity and capability evolve. Process improvement and increased capability are built in stages because some processes are ineffective when others are not stable
- The continuous representation has the same basic information as the staged representation, just arranged differently; that is, in capability levels not maturity levels, and process area categories. The continuous representation focuses process improvement on actions to be completed within process areas, yet the processes and their actions may span different levels. More sophistication in implementing the practices is expected at the different levels. These levels are called capability levels

Capability levels

- What's a capability level? Capability levels focus on maturing the organization's ability to perform, control, and improve its performance in a process area. This ability allows the organization to focus on specific areas to improve performance of that area
- There are six capability levels
  - Level 0: Incomplete
  - Level 1: Performed
  - Level 2: Managed
  - Level 3: Defined
  - Level 4: Quantitatively Managed
  - Level 5: Optimizing

Level 0: Incomplete

- An incomplete process does not implement all of the Capability Level 1 specific practices in the process area that has been selected. This is tantamount to Maturity Level 1 in the staged representation

Level 1: Performed

- A Capability Level 1 process is a process that is expected to perform all of the Capability Level 1 specific practices. Performance may not be stable and may not meet specific objectives such as quality, cost, and schedule, but useful work can be done
- This is only a start, or baby step, in process improvement. It means you are doing something, but you cannot prove that it is really working for you

Level 2: Managed

- A managed process is planned, performed, monitored, and controlled for individual projects, groups, or stand-alone processes to achieve a given purpose. Managing the process achieves both the model objectives for the process as well as other objectives, such as cost, schedule, and quality. As the title of this level states, you are actively managing the way things are done in your organization. You have some metrics that are consistently collected and applied to your management approach

Level 3: Defined

- A defined process is a managed process that is tailored from the organization's set of standard processes. Deviations beyond those allowed by the tailoring guidelines are documented, justified, reviewed, and approved. The organization's set of standard processes is just a fancy way of saying that your organization has an identity. That is, there is an organizational way of doing work that differs from the way another organization within your company may do it

Level 4: Quantitatively Managed

- A quantitatively managed process is a defined process that is controlled using statistical and other quantitative techniques. Product quality, service quality, process performance, and other business objectives are understood in statistical terms and are controlled throughout the life cycle

Level 5: Optimizing

- An optimizing process is a quantitatively managed process that is improved based on an understanding of the common causes of process variation inherent in the process. It focuses on continually improving process performance through both incremental and innovative improvements. Both the defined processes and the organization's set of standard processes are targets of improvement activities

Software Quality Metrics

- The software industry is in a constant state of change. With advances in hardware, software, graphical, and multimedia technologies, applications are constantly being rethought, redesigned, and reengineered. New development methods and techniques are continuously evolving. Because of this environment of continuous change, the software environment or software business can be extremely difficult to manage
- Effective change management techniques require consistent and meaningful measures
- The ability of an organization to effectively and efficiently manage data provides a true competitive advantage and adds value to the company's bottom line
- IEEE Standard Glossary of Software Engineering Terminology defines "measure" as an activity that ascertains or appraises by comparing to a standard
- Measurement lies at the heart of many systems that govern our lives
- Economic measurements
- Military measurements
- Medical measurements
- Atmospheric measurements
- Measurements in every day life
- Prices, weight, size, etc.
- Some aspect of a thing is assigned a descriptor that allows us to compare it with others
- Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules
- An entity is an object or an event in the real world
- An attribute is a feature or property of an entity. Typical attributes include the area or color, the cost, or the elapsed time
- We often talk about entities and their attributes interchangeably
  - 'It is cold today'
  - 'Usman is taller than Ali'

- We can make judgments about entities solely by knowing and analyzing their attributes
- Measurement is a process whose definition is far from clear-cut
- Color is an attribute of a room. How can we compare 'blue' with 'off-white'
- Can we measure intelligence of human? Is I.Q an accurate measure of the human intelligence
- Accuracy and margin of error of measurement
- Analyze and draw conclusions about measurements
- Measurement is a direct quantification of, as in measuring the height of a tree or weight of a shipment of bricks
- Calculation is indirect, where we take measurements and combine them into a quantified item that reflects some attribute whose value we are trying to understand

Measurements in SE

- We fail to set measurable targets for our software products. For example, we promise that the product will be user-friendly, reliable and maintainable without specifying clearly and objectively what these terms mean
- Projects without clear goals will not achieve their goals clearly (Gilb)
- We fail to understand and quantify the component costs of software projects. For example, most projects cannot differentiate the cost of design from the cost of coding and testing
- We do not quantify or predict the quality of the products we produce
- We allow anecdotal evidence to convince us to try yet another revolutionary new development technology, without doing a carefully controlled study to determine if the technology is efficient and effective

Objectives for Software Measurements and Measurement Programs

- Managers' Perspectives
  - What does each process cost?
  - How productive is the staff?
  - How good is the code being developed?
  - Will the user be satisfied with the product?
  - How can we improve?
- Engineers' Perspectives
  - Are the requirements testable?
  - Have we found all the faults?
  - Have we met our product or process goals?
  - What will happen in the future?

Need for Collecting Metrics

- Conduct performance appraisals to evaluate individual productivity
- Justify existence, particularly in the era of company downsizing, justification may be needed for the existence of separate test group
- Compare the quality of one system with another (similar) system
- Monitor if the quality level is affected when any changes are introduced to the software
- Estimate time and cost required to develop a system
- Evaluate the quality of service and support provided to the customer

Benefits of Metrics

- Increase customer satisfaction
- Improve productivity and quality by quantifying development and maintenance processes
- Develop, identify, and analyze trends
- Provide useful information for planning cycle
- Provide a baseline against which future efforts can be measured
- Determine the skill level and the number of resources required to support a given application

- Identify programs that require special attention or additional maintenance time
- Identify complex programs that may cause unpredictable results
- Provide constructive means of making decisions about product quality

Cost of Metrics

- There is an initial cost associated with setting up a measurement system
- However, the long-term gains derived by measuring and implementing process improvement programs certainly outweigh the cost
- Areas that require cost consideration when implementing measurement program
  - Training
  - System development methodology
  - Tools
  - Organization change

Categories of Software Metrics

- Product metrics
  - Product metrics are those that describe the characteristics of the product such as size, complexity, design features, performance, and quality level
- Process metrics
  - Process metrics are those that can be used for improving the software development and maintenance process
  - Examples include the effectiveness of defect removal during development, the pattern of testing defect arrival, and the response time of the fix process
- Project metrics
  - Project metrics are those that describe the project characteristics and execution
  - Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity

Software Quality Metrics

- Software quality metrics are a subset of software metrics that focus on quality aspects of the product, process, and project
- In general, software metrics are more closely associated with process and product metrics than with project metrics
- Nonetheless, the project parameters such as number of developers and their skill levels, the schedule, the size, and the organization structure certainly affect the quality of the product
- Before, we discuss software quality metrics, let's first discuss attributes of software quality

Attributes of Software Quality

- Correctness
  - The extent to which a program satisfies specifications and fulfills user's mission
- Reliability
  - The extent to which a program can be expected to perform the intended function

with accuracy (acceptable level of downtime)

- Efficiency
  - The extent to which a computer uses minimum resources to perform a function
- Integrity
  - The extent to which access to software by unauthorized persons can be controlled
- Usability
  - The effort required to use or learn the application
- Maintainability
  - How difficult it is find and fix the problem
- Testability
  - The effort required to test the system in order to ensure it functions according to the requirement
- Flexibility
  - The ease of updating or modifying a program
- Portability
  - The efforts required to transfer a program from one hardware configuration to another, or from one software system to another
- Reusability
  - The extent to which a code could be reused for another application

Common Measurements

- Requirements
  - Size of the document (# of words, pages, functions)
  - Number of changes to the original requirements, which were developed later in the life cycle but not specified in the original requirements document. This measure indicates how complete the original requirements document was
  - Consistency measures to ensure that the requirements are consistent with interfaces from other systems
  - Testability measures to evaluate if the requirements are written in such a way that the test cases cab be developed and traced to the requirements
- Often problems detected in the requirements are related to unclear requirement which are difficult to measure and test, such as
  - The system must be user friendly. (What does user friendly mean?)
  - The system must give speedy response time (What is speedy response time? 10 seconds, 13 seconds?)
  - The system must have state-of-the-art technology (What is considered state-of-the-art?)
  - They system must have clear management reports (What should these reports look like? What is the definition of clear?)
- Code/Design
  - No of external data items from which a module reads
  - No of external data items to which a module writes
  - No of modules specified at a later phase and not in the original design

    - No of modules which the given module calls
- Design/Code (cont'd)
    - No of lines of code
    - Data usage, measured in terms of the number of primitive data items
    - Entries/exits per module which predict the completion time of the system
- Testing
    - No of planned test cases in the test plan that ran successfully
    - Success/effectiveness of test cases against the original test plan
    - No of new unplanned test cases which are developed at a later time

Though there are several items in organization that can be tracked, measured, and improved, there are seven measures that are commonly tracked

Seven Commonly Tracked Measures

- Number of defects
- Work effort
- Schedule
- Number of changes in requirements
- Size
- Documentation defects
- Complexity

Number of Defects

- Defect count can be kept at three different stages
    - During white box testing to evaluate the quality of original code
    - During black box testing to evaluate the number of errors that escaped white box
    - After the product is released to the customer to evaluate the number of errors not found during both the unit and black box tests
- Along with the defect, the origin of the defect should be evaluated and also severity of defect should be noted

Work Effort

- Work effort constitutes the number of hours spent on development of a new system, system enhancement, or the support and maintenance of an existing system
- The hours are collected throughout the project life cycle, across all the development phases to track commitments, expectations made to the clients, and to provide historical data to improve estimating for future work efforts
- Can provide early warnings regarding budget over-runs and project delays

Schedule

- The purpose of schedule measurements is to track the performance of the project team toward meeting the committed schedule
- Planned start date versus actual date
- Planned completion date versus actual date

Number of Changes to the Requirements

- The number of additions, changes, and deletions to requirements should be measured as soon as the requirements document is checked into the formal configuration management

- This measure reflects the quality of requirements and the need to change the process of either collecting the requirements for documenting them
- The measure is also used to determine system stability, since evidence of constantly changing requirements may affect the overall quality of the product
- The changes to the requirements are counted as they occur
- Also the determination of the mean time from the requirement completion to when the first change was introduced is considered
- This data tells you how thorough the original requirement-gathering phase was
- Once the software is released, enhancement requests resulting from customer calls or updates due to problem fixes are also counted as changes are made

Size

- The size measures are important because the amount of effort required to perform most tasks is directly related to the size of the program involved
- The size is usually measured in
  - Lines of code
  - Function Points
- When the size grows larger than expected, the cost of the project as well as the estimated time for completion also grow
- The size of the software is also used for estimating the number of resources required
- The measure used to estimate program size should be easy to use early in the project life cycle
- Lines of Code
  - Empty lines
  - Comments/statements
  - Source lines
  - Reused lines
  - Lines used from other programs
- Function Points
  - It is a method of quantifying the size and complexity of a software system based on a weighted user view of the number of external inputs to the application; number of outputs from the application; inquiries end users can make; interface files; and internal logical files updated by an application
  - These five items are counted and multiplied by weight factors that adjust them for complexity
  - Function points are independent of languages, can be used to measure productivity and number of defects, are available early during functional design, and the entire product can be counted in a short time
  - They can be used for a number of productivity and quality metrics, including defects, schedules, and resource utilization
  - More on function points later

Documentation Defects

- The documentation defects are counted throughout project life cycle
- Defects of the following nature are tracked
  - Missing functionality
  - Unclear explanation
  - Spellings
  - Not user friendly

Complexity

- This metric gives data on the complexity of the software code
- As the complexity of the software increases, the required development effort increases and the probability of defects increases
- Complexity can be measured at as many successive levels as possible beginning at the individual software module level
- The purpose is to evaluate the number of resources and time that would be required to develop the application, since complex programs require more time for the programmer to code, test, and become familiar with the program
- The complexity of a module can be evaluated by
  - Number of files
  - Program size
  - Number of verbs, if statements, paragraphs, total lines, and loops
  - Architectural metrics such as
    - Go Tos
    - Entry points
    - Termination verbs
  - Diagnostics metrics which are the measure of programming language violations which may not cause compile errors or warning but which are confusing, such as
    - Dead code
    - Unresolved procedure exits
  - Design complexity which is measured by
    - Modularity (how well as design is decomposed into small, manageable modules)
    - Coupling (the interfaces between units)
- McCabe's cyclomatic complexity

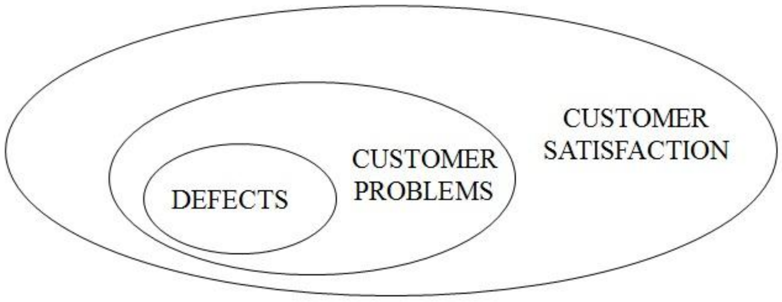
Additional Software Quality Metrics

- Mean time to failure (MTTF)
  - The mean time to failure metric measures the time between failures
  - It is often used with safety-critical systems such as the airline traffic control systems, avionics, and weapons. For example, in civilian airliners, the probability of certain catastrophic failures must be no worse than 10^(-9) per hour
  - A failure occurs when a functional unit of a software-related system can no longer perform its required function or cannot perform it within specified limits
  - Mean time between failures (MTBF)

- Defect density
  - The defect density measures the number of defects discovered per some unit of software size (lines of code, function points)
  - The defect density metric is used in many commercial software systems
  - The defect rate of a product or the expected number of defects over a certain time period is important for cost and resource estimates of the maintenance phase of the software life cycle
- Defects by severity
  - The defects by severity metric is a simple count of the number of unresolved defects listed by severity
  - Typically, this metric is measured at some regular interval and plotted to determine whether or not a trend exists
  - Ideally, a trend exists, showing progress toward the acceptable values for each severity
  - Movement away from those values should raise a flag that the project is at risk of failing to satisfy the conditions of the metric
- Customer problems
  - This metric is a simple count of the number of new (non-duplicate) problems reported by customers over some time interval
  - When measured at regular intervals and plotted, the data can be used to identify a trend. Although, a trend may be apparent, it is more useful to determine the reasons behind the trend
  - If, for example, the number of customer-reported problems increases over time, is it because more end users are using the product?
  - If you measure the number of customers who use the product at the same intervals that you measure customer-reported problems, you might identify a cause-effect or correlation between the metric and number of end users
  - For example, if you determine that as the number of end users of the system increases the number of customer-reported problems increases, a relationship may exist between the two that suggests that you may have a serious scalability flaw in your product
  - On the other hand, is the increase related to greater demands placed on the system by end users as their experience with the product matures? With the help of profiling features, you can determine the load on the product
- Customer satisfaction
  - This metric is typically measured through a customer satisfaction survey
    - Very satisfied
    - Satisfied
    - Neutral
    - Dissatisfied
    - Very dissatisfie
  - Percent of completely satisfied customers
  - Percent of satisfied customers
    - Satisfied and completely satisfied
  - Percent of dissatisfied customers
    - Dissatisfied and completely dissatisfied

- Percent of non-satisfied customers
  - Neutral, dissatisfied and completely dissatisfied

Scopes of Three Quality Metrics



Establishing a Measurement Program

- Identify the reason for establishing a measurement program
- Identify the needs of your audience and stakeholders
- Determine the level of organizational resistance and how you would address it
- Involve as many staff members as you can to overcome resistance
- Determine how much time you have
- Investigate if there are any measures already in place that you can use
- Decide which metrics to start collecting first
- Ensure there is compatibility of measurement goals with the goals of the organization
- Establish a phases implementation plan
- Establish quantifiable targets such as cost, personnel, and schedule targets for the activities and outputs of each phase of the development process
- Review the measurement plan with appropriate managers and executive management
- Obtain commitment from upper management to provide you with additional resources or tools, if the need arises
- Measure the actual values against the targets
- Determine the variances and the reasons for the variances
- Establish a plan to address the variance and standardize the plan to prevent these variances from occurring in the future
- Establish history with which to compare future data
- Integrate metrics into the development process as a part of the way you do business
- Focus on success
- Implementing by Prototyping
  - Consider prototyping the plan before complete implementation

Measurement Cycle

- After collecting measurement data
  - Analyze the data
  - Identify the problem area
  - Develop procedures to address the problem area

- Get buy-in from appropriate staff
- Train staff in the new procedures
- Implement new procedures
- Continue the measurement activity
- Change the process if need be

Causes of Failure

- Executive management not committed
- Measurement plan focused on only one metric
- Data was not utilized as intended

Process Metrics

- Process metrics are those that can be used for improving the software development and maintenance process
- Examples include the effectiveness of defect removal during development, the pattern of testing defect arrival, and the response time of the fix process
- Compared to end-product quality metrics, process quality metrics are less formally defined, and their practices vary greatly among software developers
- Some organizations pay little attention to process quality metrics, while others have well-established software metrics programs that cover various parameters in each phase of the development cycle
- Defect arrival rate
- Test effectiveness
- Defects by phase
- Defect removal effectiveness

Defect Arrival Rate

- It is the number of defects found during testing measured at regular intervals over some period of time
- Rather than a single value, a set of values is associated with this metric
- When plotted on a graph, the data may rise, indicating a positive defect arrival rate; it may stay flat, indicating a constant defect arrival rate; or decrease, indicating a negative defect arrival rate
- Interpretation of the results of this metric can be very difficult
- Intuitively, one might interpret a negative defect arrival rate to indicate that the product is improving since the number of new defects found is declining over time
- To validate this interpretation, you must eliminate certain possible causes for the decline
- For example, it could be that test effectiveness is declining over time. In other words, the tests may only be effective at uncovering certain types of problems. Once those problems have been found, the tests are no longer effective
- Another possibility is that the test organization is understaffed and consequently is unable to adequately test the product between measurement intervals. They focus their efforts during the first interval on performing stress tests that expose many problems, followed by executing system tests during the next interval where fewer problems are uncovered

Test Effectiveness

- To measure test effectiveness, take the number of defects found by formal tests and divide by the total number of formal tests
- $TE = D_n / T_n$
- When calculated at regular intervals and plotted, test effectiveness can be observed over some period of time
- If the graph rises over time, test effectiveness may be improving. On the other hand, if the graph is falling over time, test effectiveness may be waning

Defects by Phase

- It is much less expensive in terms of resources and reputation to eliminate defects early instead of fix late
- The defects by phase metric is a variation of the defect arrival rate metric
- At the conclusion of each discreet phase of the development process, a count of the new defects is taken and plotted to observe a trend
- If the graph appears to be rising, you might infer that the methods used for defect detection and removal during the earlier phases are not effective since the rate at which new defects are being discovered is increasing
- On the other hand, if the graph appears to be falling, you might conclude that early defect detection and removal is effective
- Explain the snowball effect

Defect Removal Effectiveness

- Defect removal effectiveness can be calculated by dividing the number of defects removed prior to release by the sum of defects removed prior to release and the total number of defects that remain in the product at release. When multiplied by 100, this value can be expressed as a percentage:
- $DRE = D_r/(D_r + D_t)$

Metrics for Software Maintenance Process

- When a software product has completed its development and is released to the market, it enters into the maintenance phase of its life cycle
- During this phase the defect arrivals by time interval and customer problem calls (which may or may not be defects) by time interval rate are the *de facto* metrics
- However, the number or problem arrivals is largely determined by the development process before the maintenance phase
- Not much can be done to alter the quality of the product during this phase. Therefore, the *de facto* metrics, while important, do not reflect the quality of software maintenance
- What can be done during maintenance phase is to fix the defects as soon as possible and with excellent fix quality
- Such actions, although still not able to improve the defect rate of the product, can improve customer satisfaction to a large extent

Metrics in Software Maintenance

- Defect backlog
  - The defect backlog metric is a count of the number of defects in the product following its release that require a repair
  - It is usually measured at regular intervals of time and plotted for trend analysis
  - By itself, this metric provides very little useful information
  - For example, what does a defect backlog count of 128 tells you? Can you predict the impact of those defects on customers? Can you estimate the time it would take to repair those defects? Can you recommend changes to improve the development process?
  - A more useful way to represent the defect backlog is by defect severity
  - By calculating the defect backlog by severity level, you can begin to draw useful conclusions from your measurements
- Backlog management index
  - As the backlog is worked, new problems arrive that impact the net result of your team's efforts to reduce the backlog
  - If the number of new defects exceeds the number of defects closed over some period of time, your team is losing ground to the backlog. If, on the other hand, your team closes problems faster than new ones are opened, they are gaining ground
  - The backlog management index (BMI) is calculated by dividing the number of defects closed during some period of time by the number of new defects that arrived during that same period of time
  - BMI = D_c / D_n
  - If the result is greater than 1, your team is gaining ground; otherwise it is losing
  - When measurements are taken at regular intervals and plotted, a trend can be observed indicating the rate at which the backlog is growing or shrinking
- Fix response time
  - The fix response time metric is determined by calculating the average time it takes your team to fix a defect
  - It can be measured several different ways
  - In some cases, it is the elapsed time between the discovery of the defect and the development of an unverified fix
  - In other cases, it is the elapsed time between the discovery and the development of verified fix
  - A better alternative to this metric is fix response time by severity
- Percent delinquent fixes
  - A fix is delinquent if exceeds your fix response time criteria. In other words, if you have established a maximum fix response time of 48 hours; then fix response times that exceed 48 hours are considered delinquent
  - To calculate the percent delinquent fixes; divide the number of delinquent fixes by the number of non-delinquent fixes and multiply by 100
  - PDF = $(F_d / F_n) * 100$
  - This metric is also measured better by severity since the consequences of having

a high delinquent fixes for severe defects is typically much greater than for less severe or minor defects

- Defective fixes
  - A defect for which a fix has been prepared that later turns out to be defective or worse, creates one or more additional problems, is called a defective fix
  - The defective fixes metric is a count of the number of such fixes
  - To accurately measure the number of defective fixes, your organization must not only keep track of defects that have been closed and then reopened but must also keep track of new defects that were caused by a defect fix
  - This metric is also known as fix quality

Function Points

- Historically, lines of code count have been used to measure software size
- LOC count is only one of the operational definitions of size and due to lack of standardization in LOC counting and the resulting variations in actual practices, alternative measures were investigated
- One such measure is function point
- Increasingly function points are gaining in popularity, based on the availability of industry data
- The value of using function points is in the consistency of the metric
- If we use lines of code to compare ourselves to other organizations we face the problems associated with the inconsistencies in counting lines of code
- Differences in language complexity levels and inconsistency in counting rules quickly lead us to conclude that line of code counting, even within an organization, can be problematic and ineffective
- This is not the case with function points
- Function points also serve many different measurement types
- Again, lines of code measure the value of the rate of delivery during the coding phase
- Function points measure the value of the entire deliverable from a productivity perspective, a quality perspective, and a cost perspective
- Common Industry Software Measures

Type	Metrics	Measure
Productivity	Function Points / Effort	Function points per person month
Responsiveness	Function Points / Duration	Function points per calendar month
Quality	Defects / Function Points	Defects per Function points
Business	Costs / Function Points	Cost per Function point

- A function point can be defined as a collection of executable statements that performs a certain task, together with declarations of the formal parameters and local variables manipulated by those statements

- The function points were proposed by Albrecht and his colleagues at IBM in mid-1970s
- It addresses some of the problems associated with LOC counts in size and productivity measures, especially the differences in LOC counts due to different levels of languages used
- It is a weighted total of five major components that comprise an application
- These five components are
  - Number of external inputs
    - Those items provided by the user that describe distinct application-oriented data (such as file names and menu selections). These items do not include inquiries, which are counted separately
    - For example, document filename, personal dictionary-name
  - Number of external outputs
    - Those items provided to the user that generate distinct application-oriented data (such as reports and messages, rather than the individual components of these)
    - For example, misspelled word report, number-of-words-processed message, number-of-errors-so-far message
  - Number of logical internal files
    - Logical master files in the system
    - For example, dictionary
  - Number of external interface files
    - Machine-readable interfaces to other systems
    - For example, document file, personal dictionary
  - Number of external inquiries
    - Interactive inputs requiring a response
    - For example, words processed, errors so far
- Example of a simple spell checker
- Each item/component is assigned a subjective “complexity” rating on a three point ordinal scale: “simple”, “average”, or “complex”
- Then a weight is assigned to the item according to the following table
- FP Complexity Weights

Item	Simple	Weighting Factor Average	Complex
External Inputs	3	4	6
External Outputs	4	5	7
External inquiries	3	4	6
External files	7	10	15
Internal files	5	7	10

- The complexity classification of each component is based on a set of standards that define complexity in terms of objective guidelines
- For instance, for the external output component, if the number of data element types is 20 or more and the number of file types referenced is two or more, then complexity is high
- If the number of data element types is 5 or fewer and the number of file types referenced is two to three
- With the weighting factors, the first step is to calculate the Unadjusted function counts (UFC) based on the formula, which is obtained by the by multiplying the weighting factors with the corresponding components, and adding them up
- There are fifteen (15) different varieties of items/components – in theory
- Unadjusted Function Counts
  - UFC = sum((Number of items of variety i) * weight(i))
    - For i= 1..15

- We calculate an adjusted function point count, FP, by multiplying UFC by a technical complexity factor, TCF
- This technical involves the 14 contributing factors
- Components of the Technical Complexity Factor

Reliable back-up and recovery	Data communications
Distributed functions	Performance
Heavily used configuration	Online data entry
Operational ease	Online update
Complex interface	Complex processing
Reusability	Installation ease
Multiple sites	Facilitate change

- The scores (ranging from 0 to 5) for these characteristics are then summed, based on the following formula, to form the technical complexity factor (TCF)
- Technical Complexity Factor
- TCF = 0.65 + 0.01 * (complexity of item i)
  - For i = 1 to 14

- Finally, the number of function points is obtained by multiplying unadjusted function counts and the value adjustment factor:
- FP = UFC * TCF
  - Over the years, the function point metric has gained acceptance as a key productivity measure in the application world
  - In 1986 the IFPUG was established. The IFPUG counting practices committee is the *de facto* standards organization for function point counting methods

Problems with Function Points

- Problems with subjectivity in the technology factor
- Problems with double-counting
- Problems with accuracy of TCF
- Problems with early life-cycle use
- Problems with changing requirements
- Problems with methodology dependence
- Problems with application domain
- Problems with subjective weighting

Complexity

- Ideally, we would like the complexity of the solution to be no greater than the complexity of the problem, but that is not always the case
- The complexity of a problem as the amount of resources required for an optimal solution to the problem
- The complexity of a solution can be regarded in terms of the resources needed to implement a particular solution
- Time complexity
  - Where the resource is computer time
- Space complexity
  - Where the resource is computer memory
- Problem complexity
  - Measures the complexity of the underlying problem
  - This is also known as computational complexity by computer scientists)
- Algorithmic complexity
  - Reflects the complexity of the algorithm implemented to solve the problem; in some sense, this type of complexity measures the efficiency of the software
  - We use Big-O notation to determine the complexity of algorithms
- Structural complexity
  - Measures the structure of the software used to implement the algorithm
  - For example, we look at control flow structure, hierarchical structure, and modular structure to extract this type of measure
- Cognitive complexity
  - Measures the effort required to understand the software

Structural Measures

- All other things being equal, we would like to assume that a large module takes longer to specify, design, code, and test than a small one. But experience shows us that such an assumption is not valid; the structure of the product plays a part, not only in requiring development effort but also in how the product is maintained
- We must investigate characteristics of product structure, and determine how they affect the outcomes we seek
- Structure has three parts
  - Control-flow structure
    - The control-flow addresses the sequence in which instructions are

executed in a program. This aspect of structure reflects the iterative and looping nature of programs. Whereas size counts an instruction just once, control flow makes more visible the fact an instruction may be executed many times as the program is actually run

- The control flow measures are usually modeled with directed graphs, where each node (or point) corresponds to a program statement, and each arc (or directed edge) indicates the flow of control from one statement to another
- These directed graphs are called control-flow graphs or flowgraphs
- In-degree (Fan-in)
  - ✓ A count of the number of modules that call a given module
- Out-degree (Fan-out)
  - ✓ A count of the number of modules that are called by a given module
- Path
  - ✓ A sequence of consecutive (directed) edges, some of which may be traversed more than once during the sequence
- Simple path
- Modules that have large fan-in or large fan-out may indicate a poor design. Such modules have probably not been decomposed correctly and are candidates for redesign

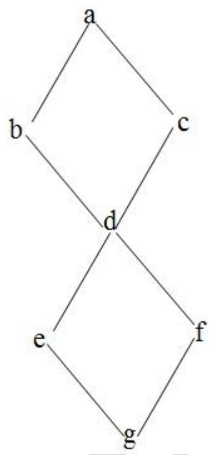
– Data-flow structure
  - Data flow follows the trail of a data item as it is created or handled by a program. Many times, the transactions applied to data are more complex than the instructions that implement them; data-flow measures depict the behavior of the data as it interacts with the program

– Data structure
  - Data structure is the organization of the data itself, independent of the program. When data elements are arranged as lists, queues, stacks, or other well-defined structure, the algorithms for creating, modifying, or deleting them are more likely to be well-defined, too. So the structure of the data tells us a great deal about the difficulty involved in writing programs to handle the data, and in defining test cases for verifying that the programs are correct
  - Sometimes a program is complex due to a complex data structure rather than complex control or data flow

Cyclomatic Complexity

- The measurement of cyclomatic complexity by McCabe was designed to indicate a program's testability and understandability (maintainability)
- It is a classical graph theory cyclomatic number, indicating the number of regions in a graph
- As applied to software, it is the number of linearly independent paths comprising the program

- As such it can be used to indicate the effort required to test a program
- To determine the paths, the program procedure is represented as a strongly connected graph with a unique entry and exit point
- M = V(G) = e – n + 2
  - where
  - V(G) = cyclomatic number of G
  - e = number of edges
  - n = number of nodes
- What we have seen is a control graph of a simple program that might contain two if statements. If we count the edges, nodes, and disconnected part of the graph, we see that



  - e = 8, n = 7
  - and that M = 8 – 7 + 2 = 3
- Note that M is equal to number of binary decisions plus one (1)
- The cyclomatic complexity metric is additive. The complexity of several graphs considered as a group is equal to the sum of the individual graphs' complexities
- However, it ignores the complexity of sequential statements
- The metric also does not distinguish between different kinds of control flow complexity such as loops versus IF-THEN-ELSE statements or cases verses nested IF-THEN-ELSE statements
- To have good testability and maintainability, McCabe recommended that no program should exceed a cyclomatic complexity of 10. Because the complexity metric is based on decisions and branches, which is consistent with the logic pattern of design and programming, it appeals to software professionals
- Many experts in software testing recommend use of the cyclomatic representation to ensure adequate test coverage; the use of McCabe's complexity measure has been gaining acceptance in practitioners

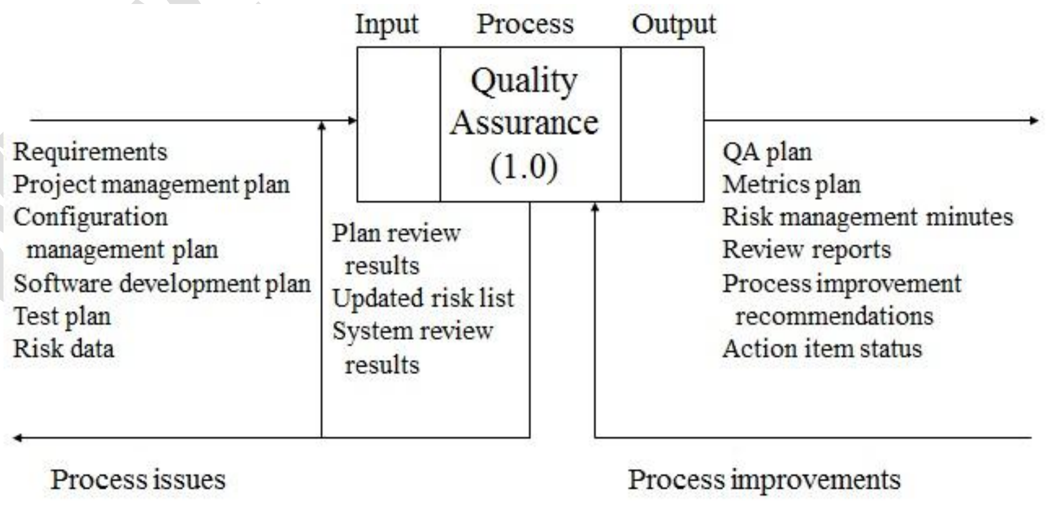
Essential Complexity

- This term is similar to "cyclomatic complexity" with an important difference. Before calculating the essential complexity level the graph of program flow is analyzed and duplicate segments are removed. Thus, if a module or code segment is utilized several times in a program, its complexity would only be counted once rather than at each instance
- Thus, essential complexity eliminates duplicate counts of reusable material and provides a firmer basis for secondary calculations such as how many test cases might be needed. This form of complexity is usually derived by tools which parse source code directly

Process Model FOR Software Quality Assurance

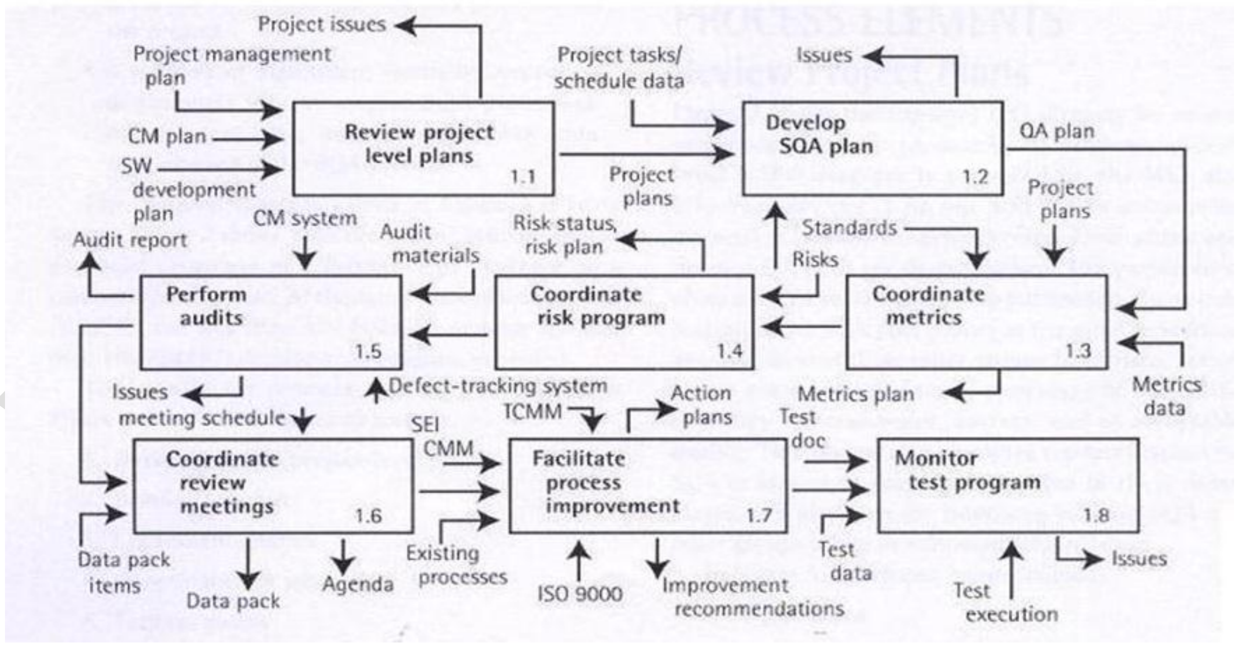
- Throughout this course, we have been talking about software quality, software quality assurance, and related activities in the software development life cycle
- Today, we'll talk about a process model of software quality assurance
- All managers and personnel who have worked in a software quality assurance have a process that represents the way an SQA organization operates
- Some mature software organizations have an SQA process that is defined by a set of standards for documentation and operations. Other smaller firms may have a less well-defined and more ad-hoc process; the primary repository for information regarding this process is in the minds of its SQA personnel
- In both cases, a process does exist
- Lacking any better definition, the process is "the way SQA is done here"
- What would an organization do with a process model for software quality assurance?
  - A tool to train new SQA engineers
  - A model to use in estimating
  - A model to use in the process improvement
  - A model to use to select or develop a set of standards and templates for SQA documentation
  - A tool to show the scope of SQA activities to new developers
  - A graphic for showing managers and developers the tasks SQA engineers perform
- The top-level of the SQA process is shown graphically, using input-process-output (IPO) technique

Software Quality Assurance Level 0 IPO Diagram



- The SQA process is based on relationships, interfaces, and tasks that defined in other project plans (for example, project management plan, configuration management plan)
- A significant task to be addressed by SQA is coordination of risk management to assist project management. This involves scheduling and convening regular meetings to address risk management, maintaining lists of active risks, and tracking status of risk mitigation plans for project management
- SQA must be actively involved in review meetings (including, but not limited to, peer reviews and formal design reviews with the customer)
- The SQA organization must actively involved in monitoring test program on the project
- The number of significant formally controlled documents (for example, SQA plan, risk management plan, metrics plan) result from performance of the SQA
- Such a model is vital for effectively identifying, estimating, and performing SQA tasks
- The key to effective SQA is the development of an SQA plan to serve as a contract between the project team and the SQA organization
- The SQA plan must be based on the process model that identifies the specific tasks SQA needs to perform to effectively support the project team
- This SQA process model has eight sub- processes
  - Review program/project-level plans
  - Develop quality assurance plan
  - Coordinate metrics
  - Coordinate risk program
  - Perform audits
  - Coordinate review meetings
  - Facilitate process improvement
  - Monitor test program

Software Quality Assurance Level 1 IPO Diagram



1. Review Project Plans
- The project-level plans must be reviewed prior to publication of SQA plan
- Since the publication of the SQA plan occurs in the same time frame as publication of these other project-level plans, review of the project-level plans is necessary to determine that they are consistent, correct, and of acceptable quality
- This review also identifies top-level tasks that SQA is tasked to performed (specified in these other plans), and identifies the interfaces between SQA and other groups in the development organization
- The plans to be reviewed are configuration management plan, project management plan, software development plan, configuration system, and standards/templates
- The test plan is not reviewed at this point in the project; it is not normally written until after the requirements are generated. Thus, the test plan is developed later than the document review that is taking place in this process
- There are three sub-processes that are components of the review project-plans process
  - Review the project management plan
  - Review the configuration management plan
  - Review the software development plan
- Members of the SQA organization should review each of these plans and report on the results of the review
- The review should determine that the project plans are complete, consistent, and correct
- Outputs from these processes are simple
  - Documented list of issues found while reviewing the various plans
  - SQA-approved project plans

2. Develop QA Plan
- As a result of developing SQA plan, SQA tasks to be performed on the program or project are formally documented
- The SQA plan provides a road map to the activities that the SQA organization will perform
- Essentially, the SQA plan serves as a formal contract between the SQA organization and the other groups on the project, specifying what tasks will be performed, according to what schedule
- Inputs for this activities are
  - Software development life cycle, as defined in the project management plan
  - Project plans
    - ✓ Project management plan
    - ✓ CM plan
    - ✓ Software development plan
  - Schedules
  - Standards (specifically, IEEE-Standard-730, Standard for Software Quality Assurance Plans)
- There are thirteen (13) sub-processes associated with SQA planning
- The first eleven (11) sections that make up an SQA plan, defined in IEEE-Standard-730
- The final two sub-processes identify the tasks required to document, assemble, and

review the SQA plan
- Sub-Processes
  - Develop QA plan introduction
  - Create management section
  - Identify documentation requirements
  - Identify standards
  - Specify reviews and audits
  - Review CM interface
  - Review defect reporting
  - Develop metrics strategy
  - Identify tools and techniques
  - Define supplier control
  - Define records approach
  - Document SQA plan
  - Review and approve SQA plan

3. Coordinate Metrics
- This sub-process defines the activities and tasks that SQA performs while coordinating and collecting metrics
- Inputs are
  - Project management plan
  - Defect tracking system
  - Metrics database
  - Product metrics data
  - Process metrics data
  - Standards
- There are nine (9) sub-processes associated with coordinate metrics
- Both process and products metrics are addressed
- Metrics reports must be published in a timely manner so that any course of corrections indicated by the metrics can be made quickly
- Sub-Processes
  - Develop metrics strategy
  - Create metrics database schema
  - Document metrics plan
  - Review metrics plan
  - Collect measurement data
  - Compute metrics
  - Evaluate trends
  - Issue metrics report
  - Update metrics process and plans
  - Output from the process include
    - ✓ Approved metrics plan
    - ✓ Metrics database
    - ✓ Regular metrics status reports

4. Coordinate Risk Program
- A significant activity performed by an SQA organization is coordination of a formal risk management program on the project
- One must remember that the ownership of risk management belongs to upper-level project management
- SQA can provide invaluable assistance to management, however, through coordination and day-to-day administration of risk management
- This coordination can involve a number of activities, including but not limited to:
  - Scheduling regular meetings to address risk issues
  - Chairing the risk meetings
  - Performing surveys of the development team (including the SQA and test engineering team) to identify areas of risk

- Maintaining a database of risk issues
- Collecting and tracking risk reduction plans

- The inputs to the Coordinate Risk Program are
  - Project-level plans
    - ✓ Project management plan
    - ✓ CM plan
    - ✓ Software QA plan
    - ✓ Test plan
  - Process risk
  - Product risk
  - Risk database
- Risk types are defined as follow
  - Known unknowns
  - Unknown knowns
  - Unknown unknowns
- There are nine (9) sub-processes associated with Coordinate Risk Program
- Sub-Processes
  - Develop risk plan
  - Review risk plan
  - Evaluate plans, schedules for risk
  - Collect process and product risks
  - Establish risk database
  - Perform risk assessment
  - Coordinate risk control
  - Coordinate risk meetings
  - Issue risk reports
- Outputs from the coordinate risk program process include
  - Risk process
  - Risk status
  - Risk management plan
  - Minutes from risk meetings
  - Risk action-item lists

5. Perform Audits

- The process for perform audits is one of the few activities remaining in modern quality assurance that can be described as a classic “policeman” activity
- Inputs include
  - Audit materials (including requirements, criteria, checklists)
  - CM system
  - Defect tracking system
  - Process templates
  - Project plans, including the test plan(s)
  - Risks
  - Standards and templates
- A fully engaged SQA organization will also perform a number of less formal audits. These should be performed according to a documented and approved audit plan, and will include audits of the CM database
- Product audits can include review of operator manuals
- Sub-Processes
  - Review project plans
  - Develop audit plan
  - Review audit plan
  - Establish audit database
  - Perform process audits
  - Perform product audits
  - Perform physical configuration audit (PCA)
  - Perform functional configuration audit (FCA)

- Update audit process
- Outputs from this include
  - ✓ Audit checklists
  - ✓ Audit issues
  - ✓ Audit reports
  - ✓ Action items
  - ✓ Updated audit process

6. Coordinate Review Meetings

- An SQA organization is usually chartered with coordinating review meetings, on a project/program where these meetings are formally controlled
- During reviews, a specific task for SQA personnel is to evaluate the process used in performing the reviews and report on that evaluation to management
- Inputs include
  - Checklists
  - Project management plan
  - Test plan
  - Review data packs
  - Review schedules
  - Standards
  - Templates

- Sub-Processes
  - Verify peer review schedule
  - Develop (peer review) agenda template
  - Support peer review meetings
  - Track (peer review) action items
  - Verify design review schedule
  - Develop design review agenda template
  - Coordinate design review data packs
  - Support design review meetings
  - Track design review action items

7. Facilitate Process Improvement

- A critical function of SQA organization in the current age is to facilitate process improvement
- This sub-process is only applicable for an SQA group when the program/project organization does not have, or does not have access to, an SEPG
- Even if a corporate-level SEPG exists, it can be advantageous for the program/project dedicated SQA group to be responsible for some of the process improvement functions
- That way, more general corporate-level initiatives can be tailored and adjusted to the needs of specific projects
- Inputs include
  - Assessment checklists
  - Current processes
  - ISO 9000
  - SEI Capability Maturity Model
  - IT Business Group (ITBG) Testing Capability Maturity Model
  - Software Process Improvement Capability Determination (SPICE) model
- Sub-Processes

- Review project plans
- Identify process improvement opportunities
- Develop process improvement plan
- Prepare for assessment
- Perform assessment
- Process assessment result
- Monitor action plan progress
- Update action plans

• Outputs from this process include
  - Action items
  - Assessment reports
  - Process improvement recommendations
  - Periodic status reports

8. Monitor Test Program

- The final sub-process within the QA process is monitor test program
- This sub-process is designed around an SQA group that does not have responsibility for testing
- It is logical for the SQA organization to take responsibility for assuring that the testing process is performed according to the documented and approved test documentation (test plans, test designs, test cases, and test procedures)
- Inputs include
  - Standards
  - Test documentation
    - ✓ Test plan
    - ✓ Test design
    - ✓ Test cases
    - ✓ Test procedures (if applicable)
    - ✓ Test summary report
  - Test output data
  - Test process
- Sub-Processes
  - Establish test metrics database
  - Collect test metrics
  - Report test metrics
  - Review test documentation
  - Monitor test execution
  - Plan test process improvement
  - Assess test process
  - Develop test assessment report
- Outputs from this process include
  - Test assessment report (including process improvement recommendation
  - Issues

Summary Starts

- This top-level model of the SQA/SQE process provides a generic model
- A specific SQA organization should tailor this model to fit its own environment and organization
- Use of this SQA process model depends on the maturity of a specific SQA group
- For a mature SQA group that has a fully documented functional process, this process can be used as a sanity check
- If there is anything in this process that the SQA group is not doing but which seems valuable, the group should add that process to its operations
- For a mature group with a functional process that is not documented, these diagrams could be tailored to match the existing process. Then the process could be formally

documented

- This process model can be used as a training aid for junior SQA engineers
- Scheduling of different SQA activities
- The SQA process model provides a tool for constructing a work breakdown structure in estimating the cost and schedule for SQA activities
- The model establishes the framework and list of SQA tasks. Then completing the estimate becomes an exercise in using existing metrics data or engineering judgment to determine the elapsed time and staffing levels to complete each task
- Finally, each task can be entered into a schedule tracking package so activities can be scheduled and tracked on an ongoing basis
- The SQA process model can be used as a guide to effective implementation of process improvement
- The process model as presented is “ideal”; a process model can also be developed that represents the current process in use by the SQA organization

Frequently Asked Questions

What is 'Software Quality Assurance'?
Software QA involves the entire software development PROCESS - monitoring and improving the process, making sure that any agreed-upon standards and procedures are followed, and ensuring that problems are found and dealt with.

What is 'Software Testing'?
Testing involves operation of a system or application under controlled conditions and evaluating the results. Testing should intentionally attempt to make things go wrong to determine if things happen when they shouldn't or things don't happen when they should.

Does every software project need testers?
It depends on the size and context of the project, the risks, the development methodology, the skill and experience of the developers. If the project is a short-term, small, low risk project, with highly experienced programmers utilizing thorough unit testing or test-first development, then test engineers may not be required for the project to succeed. For non-trivial-size projects or projects with non-trivial risks, a testing staff is usually necessary. The use of personnel with specialized skills enhances an organization's ability to be successful in large, complex, or difficult tasks. It allows for both a) deeper and stronger skills and b) the contribution of differing perspectives.

What is Regression testing?
Retesting of a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made.

Why does software have bugs?

- Some of the reasons are:
- Miscommunication or no communication.
- Programming errors
- Changing requirements
- Time pressures

How can new Software QA processes be introduced in an existing Organization?

- It depends on the size of the organization and the risks involved.
- For small groups or projects, a more ad-hoc process may be appropriate, depending on the type of customers and projects.
- By incremental self managed team approaches.

What is verification and Validation?
Verification typically involves reviews and meetings to evaluate documents, plans, code, requirements, and specifications. This can be done with checklists, issues lists, walkthroughs, and inspection meetings. Validation typically involves actual testing and takes place after verifications are completed.

What is a 'walkthrough'? What's an 'inspection'?
A 'walkthrough' is an informal meeting for evaluation or informational purposes. Little or no preparation is usually required. An inspection is more formalized than a 'walkthrough', typically with 3-8 people including a moderator, reader, and a recorder to take notes. The subject of the

inspection is typically a document such as a requirements spec or a test plan, and the purpose is to find problems and see what's missing, not to fix anything.

What kinds of testing should be considered?
Some of the basic kinds of testing involve:
Blackbox testing, Whitebox testing, Integration testing, Functional testing, smoke testing, Acceptance testing, Load testing, Performance testing, User acceptance testing.

What are 5 common problems in the software development process?

- Poor requirements
- Unrealistic Schedule
- Inadequate testing
- Changing requirements
- Miscommunication

What are 5 common solutions to software development problems?

- Solid requirements
- Realistic Schedule
- Adequate testing
- Clarity of requirements
- Good communication among the Project team

What is software 'quality'?
Quality software is reasonably bug-free, delivered on time and within budget, meets requirements and/or expectations, and is maintainable

What are some recent major computer system failures caused by software bugs?
Trading on a major Asian stock exchange was brought to a halt in November of 2005, reportedly due to an error in a system software upgrade. A May 2005 newspaper article reported that a major hybrid car manufacturer had to install a software fix on 20,000 vehicles due to problems with invalid engine warning lights and occasional stalling. Media reports in January of 2005 detailed severe problems with a $170 million high-profile U.S. government IT systems project. Software testing was one of the five major problem areas according to a report of the commission reviewing the project.

What is 'good code'? What is 'good design'?
'Good code' is code that works, is bug free, and is readable and maintainable. Good internal design is indicated by software code whose overall structure is clear, understandable, easily modifiable, and maintainable; is robust with sufficient error-handling and status logging capability; and works correctly when implemented. Good functional design is indicated by an application whose functionality can be traced back to customer and end-user requirements.

What is SEI? CMM? CMMI? ISO? Will it help?
These are all standards that determine effectiveness in delivering quality software. It helps organizations to identify best practices useful in helping them increase the maturity of their processes.

What steps are needed to develop and run software tests?

- Obtain requirements, functional design, and internal design specifications and other necessary documents

- Obtain budget and schedule requirements.
- Determine Project context.
- Identify risks.
- Determine testing approaches, methods, test environment, test data.
- Set Schedules, testing documents.
- Perform tests.
- Perform reviews and evaluations
- Maintain and update documents

What's a 'test plan'? What's a 'test case'?
A software project test plan is a document that describes the objectives, scope, approach, and focus of a software testing effort. A test case is a document that describes an input, action, or event and an expected response, to determine if a feature of an application is working correctly.

What should be done after a bug is found?
The bug needs to be communicated and assigned to developers that can fix it. After the problem is resolved, fixes should be re-tested, and determinations made regarding requirements for regression testing to check that fixes didn't create problems elsewhere

Will automated testing tools make testing easier?
It depends on the Project size. For small projects, the time needed to learn and implement them may not be worth it unless personnel are already familiar with the tools. For larger projects, or on-going long-term projects they can be valuable.

What's the best way to choose a test automation tool? Some of the points that can be noted before choosing a tool would be:
- Analyze the non-automated testing situation to determine the testing activity that is being performed.
- Testing procedures that are time consuming and repetition.
- Cost/Budget of tool, Training and implementation factors.
- Evaluation of the chosen tool to explore the benefits.

How can it be determined if a test environment is appropriate?
Test environment should match exactly all possible hardware, software, network, data, and usage characteristics of the expected live environments in which the software will be used.

What's the best approach to software test estimation?
- The 'best approach' is highly dependent on the particular organization and project and the experience of the personnel involved. Some of the following approaches to be considered are:
- Implicit Risk Context Approach
- Metrics-Based Approach
- Test Work Breakdown Approach
- Iterative Approach
- Percentage-of-Development Approach

What if the software is so buggy it can't really be tested at all?
The best bet in this situation is for the testers to go through the process of reporting whatever bugs or blocking-type problems initially show up, with the focus being on critical bugs.

How can it be known when to stop testing?
Common factors in deciding when to stop are:

- Deadlines (release deadlines, testing deadlines, etc.)
- Test cases completed with certain percentage passed
- Test budget depleted
- Coverage of code/functionality/requirements reaches a specified point
- Bug rate falls below a certain level
- Beta or alpha testing period ends

What if there isn't enough time for thorough testing?

- Use risk analysis to determine where testing should be focused.
- Determine the important functionalities to be tested.
- Determine the high risk aspects of the project.
- Prioritize the kinds of testing that need to be performed.
- Determine the tests that will have the best high-risk-coverage to time-required ratio.

What if the project isn't big enough to justify extensive testing?
Consider the impact of project errors, not the size of the project. The tester might then do ad hoc testing, or write up a limited test plan based on the risk analysis.

How does a client/server environment affect testing?
Client/server applications can be quite complex due to the multiple dependencies among clients, data communications, hardware, and servers, especially in multi-tier systems. Load/stress/performance testing may be useful in determining client/server application limitations and capabilities.

How can World Wide Web sites be tested?
Some of the considerations might include:

- Testing the expected loads on the server
- Performance expected on the client side
- Testing the required securities to be implemented and verified.
- Testing the HTML specification, external and internal links
- cgi programs, applets, javascripts, ActiveX components, etc. to be maintained, tracked, controlled

How is testing affected by object-oriented designs?
Well-engineered object-oriented design can make it easier to trace from code to internal design to functional design to requirements. If the application was well-designed this can simplify test design.

What is Extreme Programming and what's it got to do with testing?
Extreme Programming (XP) is a software development approach for small teams on risk-prone projects with unstable requirements. For testing ('extreme testing', programmers are expected to write unit and functional test code first - before writing the application code. Customers are expected to be an integral part of the project team and to help develop scenarios for acceptance/black box testing.

What makes a good Software Test engineer?
A good test engineer has a 'test to break' attitude, an ability to take the point of view of the customer, a strong desire for quality, and an attention to detail. Tact and diplomacy are useful in

maintaining a cooperative relationship with developers, and an ability to communicate with both technical (developers) and non-technical (customers, management) people is useful.

What makes a good Software QA engineer?
They must be able to understand the entire software development process and how it can fit into the business approach and goals of the organization. Communication skills and the ability to understand various sides of issues are important. In organizations in the early stages of implementing QA processes, patience and diplomacy are especially needed. An ability to find problems as well as to see 'what's missing' is important for inspections and reviews

What's the role of documentation in QA?
QA practices should be documented such that they are repeatable. Specifications, designs, business rules, inspection reports, configurations, code changes, test plans, test cases, bug reports, user manuals, etc. should all be documented. Change management for documentation should be used.

What is a test strategy? What is the purpose of a test strategy?
It is a plan for conducting the test effort against one or more aspects of the target system. A test strategy needs to be able to convince management and other stakeholders that the approach is sound and achievable, and it also needs to be appropriate both in terms of the software product to be tested and the skills of the test team.

What information does a test strategy captures?
It captures an explanation of the general approach that will be used and the specific types, techniques, styles of testing

What is test data?
It is a collection of test input values that are consumed during the execution of a test, and expected results referenced for comparative purposes during the execution of a test

What is Unit testing?
It is implemented against the smallest testable element (units) of the software, and involves testing the internal structure such as logic and dataflow, and the unit's function and observable behaviors

How can the test results be used in testing?
Test Results are used to record the detailed findings of the test effort and to subsequently calculate the different key measures of testing

What is Developer testing?
Developer testing denotes the aspects of test design and implementation most appropriate for the team of developers to undertake.

What is independent testing?
Independent testing denotes the test design and implementation most appropriately performed by someone who is independent from the team of developers.

What is Integration testing?
Integration testing is performed to ensure that the components in the implementation model operate properly when combined to execute a use case

What is System testing?
A series of tests designed to ensure that the modified program interacts correctly with other system components. These test procedures typically are performed by the system maintenance staff in their development library.

What is Acceptance testing?
User acceptance testing is the final test action taken before deploying the software. The goal of acceptance testing is to verify that the software is ready, and that it can be used by end users to perform those functions and tasks for which the software was built

What is the role of a Test Manager?
The Test Manager role is tasked with the overall responsibility for the test effort's success. The role involves quality and test advocacy, resource planning and management, and resolution of issues that impede the test effort

What is the role of a Test Analyst?
The Test Analyst role is responsible for identifying and defining the required tests, monitoring detailed testing progress and results in each test cycle and evaluating the overall quality experienced as a result of testing activities. The role typically carries the responsibility for appropriately representing the needs of stakeholders that do not have direct or regular representation on the project.

What is the role of a Test Designer?
The Test Designer role is responsible for defining the test approach and ensuring its successful implementation. The role involves identifying the appropriate techniques, tools and guidelines to implement the required tests, and to give guidance on the corresponding resources requirements for the test effort

What are the roles and responsibilities of a Tester?
The Tester role is responsible for the core activities of the test effort, which involves conducting the necessary tests and logging the outcomes of that testing. The tester is responsible for identifying the most appropriate implementation approach for a given test, implementing individual tests, setting up and executing the tests, logging outcomes and verifying test execution, analyzing and recovering from execution errors.

What are the skills required to be a good tester?
A tester should have knowledge of testing approaches and techniques, diagnostic and problem-solving skills, knowledge of the system or application being tested, and knowledge of networking and system architecture

What is test coverage?
Test coverage is the measurement of testing completeness, and it's based on the coverage of testing expressed by the coverage of test requirements and test cases or by the coverage of executed code.

What is a test script?
The step-by-step instructions that realize a test, enabling its execution. Test Scripts may take the form of either documented textual instructions that are executed manually or computer readable instructions that enable automated test execution.

What is 'Software Quality Assurance'?
Software QA involves the entire software development PROCESS - monitoring and improving the process, making sure that any agreed-upon processes, standards and procedures are followed, and ensuring that problems are found and dealt with. It is oriented to 'prevention'.

What is 'Software Testing'?
Testing involves operation of a system or application under controlled conditions and evaluating the results (e.g., 'if the user is in interface A of the application while using hardware B, and does C, then D should happen'). The controlled conditions should include both normal and abnormal conditions. Testing should intentionally attempt to make things go wrong to determine if things happen when they shouldn't or things don't happen when they should. It is oriented to 'detection'.

- Organizations vary considerably in how they assign responsibility for QA and testing. Sometimes they're the combined responsibility of one group or individual. Also common are project teams that include a mix of testers and developers who work closely together, with overall testing and QA processes monitored by project managers. It will depend on what best fits an organization's size and business structure.
- Note that testing can be done by machines or people. When done by machines (computers usually) it's often called 'automated testing

Does every software project need testers?
While all projects will benefit from testing, some projects may not require independent test staff to succeed.

Which projects may not need independent test staff? The answer depends on the size and context of the project, the risks, the development methodology, the skill and experience of the developers, and other factors. For instance, if the project is a short-term, small, low risk project, with highly experienced programmers utilizing thorough unit testing or test-first development, then test engineers may not be required for the project to succeed.

In some cases an IT organization may be too small or new to have a testing staff even if the situation calls for it. In these circumstances it may be appropriate to instead use contractors or outsourcing, or adjust the project management and development approach (by switching to more senior developers and agile test-first development, for example). Inexperienced managers sometimes gamble on the success of a project by skipping thorough testing or having programmers do post-development functional testing of their own work, a decidedly high risk gamble.

For non-trivial-size projects or projects with non-trivial risks, a testing staff is usually necessary. As in any business, the use of personnel with specialized skills enhances an organization's ability to be successful in large, complex, or difficult tasks. It allows for both a) deeper and stronger skills and b) the contribution of differing perspectives. For example, programmers typically have the perspective of 'what are the technical issues in making this functionality work?'. A test engineer typically has the perspective of 'what might go wrong with this functionality, and how can we ensure it meets expectations?'. A technical person who can be highly effective in approaching tasks from both of those perspectives is rare, which is why, sooner or later, organizations bring in test specialists.

Why does software have bugs?
Miscommunication or no communication - as to specifics of what an application should or shouldn't do (the application's requirements).

Software complexity - the complexity of current software applications can be difficult to comprehend for anyone without experience in modern-day software development. Multi-tier distributed systems, applications utilizing multiple local and remote web services applications, use of cloud infrastructure, data communications, enormous databases, security complexities, and sheer size of applications have all contributed to the exponential growth in software/system complexity.

Programming errors - programmers, like anyone else, can make mistakes.

Changing requirements (whether documented or undocumented) - the end-user may not understand the effects of changes, or may understand and request them anyway - redesign, rescheduling of engineers, effects on other projects, work already completed that may have to be redone or thrown out, hardware requirements that may be affected, etc. If there are many minor changes or any major changes, known and unknown dependencies among parts of the project are likely to interact and cause problems, and the complexity of coordinating changes may result in errors. Enthusiasm of engineering staff may be affected. In some fast-changing business environments, continuously modified requirements may be a fact of life. In this case, management must understand the resulting risks, and QA and test engineers must adapt and plan for continuous extensive testing to keep the inevitable bugs from running out of control

Time pressures - scheduling of software projects is difficult at best, often requiring a lot of guesswork. When deadlines loom and the crunch comes, mistakes will be made

Poorly designed/documented code - it's tough to maintain and modify code that is badly written or poorly documented; the result is bugs. In many organizations management provides no incentive for programmers to write clear, understandable, maintainable code. In fact, it's usually the opposite: they get points mostly for quickly turning out code, and there's job security if nobody else can understand it ('if it was hard to write, it should be hard to read').

Software development tools - visual tools, class libraries, compilers, scripting tools, etc. often introduce their own bugs or are poorly documented, resulting in added bugs.

How can new Software QA processes be introduced in an existing organization?
A lot depends on the size of the organization and the risks involved. For large organizations with high-risk (in terms of lives or property) projects, serious management buy-in is required and a more formalized QA process is necessary.

Where the risk is lower, management and organizational buy-in and QA implementation may be a slower, step-at-a-time process. QA processes should be balanced with productivity so as to keep bureaucracy from getting out of hand.

For small groups or projects, a more ad-hoc process may be appropriate, depending on the type of customers and projects. A lot will depend on team leads or managers, feedback to developers, and ensuring adequate communications among customers, managers, developers, and testers.

The most value for effort will often be in (a) requirements management processes, with a goal of clear, complete, testable requirement specifications embodied in requirements or design documentation, or in agile-type environments extensive continuous coordination with end-users, (b) design reviews and code reviews, and (c) post-mortems/retrospectives. Agile approaches utilizing extensive regular communication can coordinate well with improved QA processes.

Other possibilities include incremental self-managed team approaches such as 'Kaizen' methods of continuous process improvement, the Deming-Shewhart Plan-Do-Check-Act cycle, and others.

What is verification & validation?
Verification typically involves reviews and meetings to evaluate documents, plans, code, requirements, and specifications. This can be done with checklists, issues lists, walkthroughs, and inspection meetings. Validation typically involves actual testing and takes place after verifications are completed. The term 'IV & V' refers to Independent Verification and Validation.

What is a 'walkthrough'?
A 'walkthrough' is an informal meeting for evaluation or informational purposes. Little or no preparation is usually required.

What's an 'inspection'?
An inspection is more formalized than a 'walkthrough', typically with 3-8 people including a moderator, reader, and a recorder to take notes. The subject of the inspection is typically a document such as a requirements spec or a test plan, and the purpose is to find problems and see what's missing, not to fix anything. Attendees should prepare for this type of meeting by reading thru the document; most problems will be found during this preparation. The result of the inspection meeting should be a written report. Thorough preparation for inspections is difficult, painstaking work, but is one of the most cost effective methods of ensuring quality. Employees who are most skilled at inspections are like the 'eldest brother' in the parable in 'Why is it often hard for organizations to get serious about quality assurance?'. Their skill may have low visibility but they are extremely valuable to any software development organization, since bug prevention is far more cost-effective than bug detection.

What kinds of testing should be considered?

- black box testing - not based on any knowledge of internal design or code. Tests are based on requirements and functionality.
- white box testing - based on knowledge of the internal logic of an application's code. Tests are based on coverage of code statements, branches, paths, conditions.
- unit testing - the most 'micro' scale of testing; to test particular functions or code modules. Typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code. Not always easily done unless the application has a well-designed architecture with tight code; may require developing test driver modules or test harnesses.
- API testing - testing of messaging/data exchange among systems or components of systems. Such testing usually does not involve GUI's (graphical user interfaces). It is often considered a type of 'mid-level' testing.
- incremental integration testing - continuous testing of an application as new functionality is added; requires that various aspects of an application's functionality be independent enough to work separately before all parts of the program are completed, or that test drivers be developed as needed; done by programmers or by testers.

- integration testing - testing of combined parts of an application to determine if they function together correctly. The 'parts' can be code modules, services, individual applications, client and server applications on a network, etc. This type of testing is especially relevant to multi-tier and distributed systems.
- functional testing - black-box type testing geared to functional requirements of an application; this type of testing should be done by testers. This doesn't mean that the programmers shouldn't check that their code works before releasing it (which of course applies to any stage of testing.)
- system testing - black-box type testing that is based on overall requirements specifications; covers all combined parts of a system.
- end-to-end testing - similar to system testing; the 'macro' end of the test scale; involves testing of a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.
- sanity testing or smoke testing - typically an initial testing effort to determine if a new software version is performing well enough to accept it for a major testing effort. For example, if the new software is crashing systems every 5 minutes, bogging down systems to a crawl, or corrupting databases, the software may not be in a 'sane' enough condition to warrant further testing in its current state.
- regression testing - re-testing after fixes or modifications of the software or its environment. It can be difficult to determine how much re-testing is needed, especially near the end of the development cycle. Automated testing approaches can be especially useful for this type of testing.
- acceptance testing - final testing based on specifications of the end-user or customer, or based on use by end-users/customers over some limited period of time.
- load testing - testing an application under heavy loads, such as testing of a web site under a range of loads to determine at what point the system's response time degrades or fails.
- stress testing - term often used interchangeably with 'load' and 'performance' testing. Also used to describe such tests as system functional testing while under unusually heavy loads, heavy repetition of certain actions or inputs, input of large numerical values, large complex queries to a database system, etc.
- performance testing - term often used interchangeably with 'stress' and 'load' testing. Ideally 'performance' testing (and any other 'type' of testing) is defined in requirements documentation or QA or Test Plans.
- usability testing - testing for 'user-friendliness'. Clearly this is subjective, and will depend on the targeted end-user or customer. User interviews, surveys, video recording of user sessions, and other techniques can be used. Programmers and testers are usually not appropriate as usability testers.
- install/uninstall testing - testing of full, partial, or upgrade install/uninstall processes.
- recovery testing - testing how well a system recovers from crashes, hardware failures, or other catastrophic problems.
- failover testing - typically used interchangeably with 'recovery testing'
- security testing - testing how well the system protects against unauthorized internal or external access, willful damage, etc; may require sophisticated testing techniques.
- compatibility testing - testing how well software performs in a particular hardware/software/operating system/network/etc. environment.
- exploratory testing - often taken to mean a creative, informal software test that is not based on formal test plans or test cases; testers may be learning the software as they test it.

- ad-hoc testing - similar to exploratory testing, but often taken to mean that the testers have significant understanding of the software before testing it.
- context-driven testing - testing driven by an understanding of the environment, culture, and intended use of software. For example, the testing approach for life-critical medical equipment software would be completely different than that for a low-cost computer game.
- user acceptance testing - determining if software is satisfactory to an end-user or customer.
- comparison testing - comparing software weaknesses and strengths to competing products.
- alpha testing - testing of an application when development is nearing completion; minor design changes may still be made as a result of such testing. Typically done by end-users or others, not by programmers or testers.
- beta testing - testing when development and testing are essentially completed and final bugs and problems need to be found before final release. Typically done by end-users or others, not by programmers or testers.
- mutation testing - a method for determining if a set of test data or test cases is useful, by deliberately introducing various code changes ('bugs') and retesting with the original test data/cases to determine if the 'bugs' are detected. Proper implementation requires large computational resources.

What are 5 common problems in the software development process?

- poor requirements or user stories - if these are unclear, incomplete, too general, or not testable, there may be problems.
- unrealistic schedule - if too much work is crammed in too little time, problems are inevitable.
- inadequate testing - no one will know whether or not the software is any good until customers complain or systems crash.
- featuritis - requests to add on new features after development goals are agreed on.
- miscommunication - if developers don't know what's needed or customers have erroneous expectations, problems can be expected.

In agile projects, problems often occur when the project diverges from agile principles (such as forgetting that 'Business people and developers must work together daily throughout the project.'

What are 5 common solutions to software development problems?

- solid requirements - clear, complete, detailed, cohesive, attainable, testable requirements that are agreed to by all players. In 'agile'-type environments, continuous close coordination with customers/end-users is necessary to ensure that changing/emerging requirements are understood.
- realistic schedules - allow adequate time for planning, design, testing, bug fixing, re-testing, changes, and documentation; personnel should be able to complete the project without burning out.
- adequate testing - start testing early on, re-test after fixes or changes, plan for adequate time for testing and bug-fixing. 'Early' testing could include static code analysis/testing, test-first development, unit testing by developers, built-in testing and diagnostic capabilities, automated post-build testing, etc.
- stick to initial requirements where feasible - be prepared to defend against excessive changes and additions once development has begun, and be prepared to explain consequences. If changes are necessary, they should be adequately reflected in related

schedule changes. If possible, work closely with customers/end-users to manage expectations. In 'agile'-type environments, initial requirements may be expected to change significantly, requiring that true agile processes be in place and followed.

- communication - require walkthroughs and inspections when appropriate; make extensive use of group communication tools - groupware, wiki's, bug-tracking tools and change management tools, etc.; ensure that information/documentation is available and up-to-date; promote teamwork and cooperation; use prototypes, frequent deliveries, and/or continuous communication with end-users if possible to clarify expectations.

What is software 'quality'?

Quality software is reasonably bug-free, delivered on time and within budget, meets requirements and/or expectations, and is maintainable. However, quality is obviously a subjective term. It will depend on who the 'customer' is and their overall influence in the scheme of things. A wide-angle view of the 'customers' of a software development project might include end-users, customer acceptance testers, customer contract officers, customer management, the development organization's management/accountants/testers/salespeople, future software maintenance engineers, stockholders, magazine columnists, etc. Each type of 'customer' will have their own slant on 'quality' - the accounting department might define quality in terms of profits while an end-user might define quality as user-friendly and bug-free.

What is 'good code'?

'Good code' is code that works, is reasonably bug free, secure, and is readable and maintainable. Some organizations have coding 'standards' that all developers are supposed to adhere to, but everyone has different ideas about what's best, or what is too many or too few rules. There are also various theories and metrics, such as McCabe Complexity metrics. It should be kept in mind that excessive use of standards and rules can stifle productivity and creativity. 'Peer reviews', 'buddy checks' pair programming, code analysis tools, etc. can be used to check for problems and enforce standards. For example, in C/C++ coding, here are some typical ideas to consider in setting rules/standards; these may or may not apply to a particular situation:

- minimize or eliminate use of global variables.
- use descriptive function and method names - use both upper and lower case, avoid abbreviations, use as many characters as necessary to be adequately descriptive (use of more than 20 characters is not out of line); be consistent in naming conventions.
- use descriptive variable names - use both upper and lower case, avoid abbreviations, use as many characters as necessary to be adequately descriptive (use of more than 20 characters is not out of line); be consistent in naming conventions.
- function and method sizes should be minimized; less than 100 lines of code is good, less than 50 lines is preferable.
- function descriptions should be clearly spelled out in comments preceding a function's code.
- organize code for readability.
- use whitespace generously - vertically and horizontally
- each line of code should contain 70 characters max.
- one code statement per line.
- coding style should be consistent throughout a program (e.g., use of brackets, indentations, naming conventions, etc.)
- in adding comments, err on the side of too many rather than too few comments; a common rule of thumb is that there should be at least as many lines of comments (including header blocks) as lines of code.

- no matter how small, an application should include documentation of the overall program function and flow (even a few paragraphs is better than nothing); or if possible a separate flow chart and detailed program documentation.
- make extensive use of error handling procedures and status and error logging.
- for C++, to minimize complexity and increase maintainability, avoid too many levels of inheritance in class hierarchies (relative to the size and complexity of the application). Minimize use of multiple inheritance, and minimize use of operator overloading (note that the Java programming language eliminates multiple inheritance and operator overloading.)
- for C++, keep class methods small, less than 50 lines of code per method is preferable.
- for C++, make liberal use of exception handlers

What is 'good design'?
'Design' could refer to many things, but often refers to 'functional design' or 'internal design'. Good internal design is indicated by software code whose overall structure is clear, understandable, easily modifiable, and maintainable; is robust with sufficient error-handling and status logging capability; and works as expected when implemented. Good functional design is indicated by an application whose functionality can be traced back to customer and end-user requirements or user stories. (See further discussion of functional and internal design in FAQ 'What's the big deal about requirements?'). For programs that have a user interface, it's often a good idea to assume that the end user will have little computer knowledge and may not read a user manual or even the on-line help; some common rules-of-thumb include:

- the program should act in a way that least surprises the user
- it should always be evident to the user what can be done next and how to exit
- the program shouldn't let the users do something stupid without warning them

What is SEI? CMM? CMMI? ISO? Will it help?

SEI = 'Software Engineering Institute' at Carnegie-Mellon University; initiated by the U.S. Defense Department to help improve software development processes.

CMM = 'Capability Maturity Model', now called the CMMI ('Capability Maturity Model Integration'), developed by the SEI and as of January 2013 overseen by the CMMI Institute at Carnegie Mellon University. In the 'staged' version, it's a model of 5 levels of process 'maturity' that help determine effectiveness in delivering quality software. CMMI models are "collections of best practices that help organizations to improve their processes." It is geared to larger organizations such as large U.S. Defense Department contractors. However, many of the QA processes involved are appropriate to any organization, and if reasonably applied can be helpful. Organizations can receive CMMI ratings by undergoing assessments by qualified auditors. CMMI V1.3 (2010) supports Agile development processes.

- Level 1 - 'Initial': characterized by chaos, periodic panics, and heroic efforts required by individuals to successfully complete projects. Few if any processes in place; successes may not be repeatable.
- Level 2 - 'Managed': projects carried out in accordance with policies and employ skilled personnel with sufficient resources. Project tracking and reporting is in place. Schedules and budgets are set and revised as needed. Work products are appropriately controlled.
- Level 3 - 'Defined': standard development and maintenance processes are established, integrated consistently throughout an organization,
- Level 4 - 'Quantitatively Managed': metrics are used to track process performance.

Project performance is controlled and predictable.
- Level 5 - 'Optimizing': the focus is on continuous process improvement. The impact of new processes and technologies can be predicted and effectively implemented when required. Quality and process objectives are established and regularly revised to reflect changing objectives and organizational performance, and used as criteria in managing process improvement.

What is the 'software life cycle'?
The life cycle begins when an application is first conceived and ends when it is no longer in use. It includes aspects such as initial concept, requirements analysis, functional design, internal design, documentation planning, test planning, coding, document preparation, integration, testing, maintenance, updates, retesting, phase-out, and other aspects.

What makes a good Software Test engineer?
A good test engineer has a 'test to break' attitude, an ability to take the point of view of the customer, a strong desire for quality, and an attention to detail. Tact and diplomacy are useful in maintaining a cooperative relationship with developers, and an ability to communicate with both technical (developers) and non-technical (customers, management) people is useful. Previous software development experience can be helpful as it provides a deeper understanding of the software development process, gives the tester an appreciation for the developers' point of view, and reduce the learning curve in automated test programming. Judgement skills are needed to assess high-risk or critical areas of an application on which to focus testing efforts when time is limited.

What makes a good Software QA engineer?
The same qualities a good tester has are useful for a QA engineer. Additionally, they must be able to understand the entire software development process and how it can fit into the business approach and goals of the organization. Communication skills and the ability to understand various sides of issues are important. In organizations in the early stages of implementing QA processes, patience and diplomacy are especially needed. An ability to find problems as well as to see 'what's missing' is important for inspections and reviews.

What makes a good QA or Test manager?
A good QA, test, or QA/Test(combined) manager should:
- be familiar with the software development process
- be able to maintain enthusiasm of their team and promote a positive atmosphere, despite what is a somewhat 'negative' process (e.g., looking for or preventing problems)
- be able to promote teamwork to increase productivity
- be able to promote cooperation between software, test, and QA engineers
- have the diplomatic skills needed to promote improvements in QA processes
- have the ability to withstand pressures and provide appropriate feedback to other managers when there are issues with quality/processes/schedules/risk
- have people judgement skills for hiring and keeping skilled personnel
- be able to communicate with technical and non-technical people, engineers, managers, and customers.
- be able to run meetings and keep them focused

What's the role of documentation in QA?

Generally, the larger the team/organization, the more useful it will be to stress documentation, in order to manage and communicate more efficiently. (Note that documentation may be electronic, not necessarily in printable form, and may be embedded in code comments, may be embodied in well-written test cases, user stories, etc.) QA practices may be documented to enhance their repeatability. Specifications, designs, business rules, configurations, code changes, test plans, test cases, bug reports, user manuals, etc. may be documented in some form. There would ideally be a system for easily finding and obtaining information and determining what documentation will have a particular piece of information. Change management for documentation can be used where appropriate. For agile software projects, it should be kept in mind that one of the agile values is "Working software over comprehensive documentation", which does not mean 'no' documentation. Agile projects tend to stress the short term view of project needs; documentation often becomes more important in a project's long-term context.

What's the big deal about 'requirements'?

Depending on the project, it may or may not be a 'big deal'. For agile projects, requirements are expected to change and evolve, and detailed documented requirements may not be needed. However some requirements, in the form of user stories or something similar, are useful. For non-agile types of projects detailed documented requirements are usually needed. (Note that requirements documentation can be electronic, not necessarily in the form of printable documents, and may be embedded in code comments, or may be embodied in well-written test cases, wiki's, user stories, etc.) Requirements are the details describing an application's externally-perceived functionality and properties. Requirements are ideally clear, complete, reasonably detailed, cohesive, attainable, and testable. A non-testable requirement would be, for example, 'user-friendly' (too subjective). A more testable requirement would be something like 'the user must enter their previously-assigned password to access the application'. Determining and organizing requirements details in a useful and efficient way can be a difficult effort; different methods and software tools are available depending on the particular project. Many books are available that describe various approaches to this task.

Care should be taken to involve ALL of a project's significant 'customers' in the requirements process. 'Customers' could be in-house personnel or outside personnel, and could include end-users, customer acceptance testers, customer contract officers, customer management, future software maintenance engineers, salespeople, etc. Anyone who could later derail the success of the project if their expectations aren't met should be included if possible.

Organizations vary considerably in their handling of requirements specifications. Often the requirements are spelled out in a document with statements such as 'The product shall.....'. 'Design' specifications should not be confused with 'requirements'. It can be helpful to have design specifications traceable back to the requirements.

In some organizations requirements may end up in high level project plans, functional specification documents, in design documents, or in other documents at various levels of detail. No matter what they are called, some type of documentation with requirements, user stories, and related information will be useful to testers in order to properly plan and execute tests (manual or automated). Without such documentation, there will be no clear-cut way to determine if software is performing correctly.

If testable requirements are not available or are only partially available, useful testing can still be performed. In this situation test results may be more oriented to providing information about the state of the software and risk levels, rather than providing pass/fail results. A relevant testing approach in this situation may include approaches such as 'exploratory testing'. Many software projects have a mix of documented testable requirements, poorly documented requirements, undocumented requirements, and changing requirements. In such projects a mix of scripted and exploratory testing approaches may be useful.

'Agile' approaches use methods requiring close interaction and cooperation between programmers and stakeholders/customers/end-users to iteratively develop requirements, user stories, etc. In the XP 'test first' approach developers create automated unit testing code before the application code, and these automated unit tests can essentially embody the requirements.

What steps are needed to develop and run software tests?

The following are some of the steps to consider:

- Obtain requirements, functional design, internal design specifications, user stories, or other available/necessary information
- Obtain budget and schedule requirements
- Determine project-related personnel and their responsibilities, reporting requirements, required standards and processes (such as release processes, change processes, etc.)
- Determine project context, relative to the existing quality culture of the product/organization/business, and how it might impact testing scope, approaches, and methods.
- Identify the application's higher-risk and more important aspects, set priorities, and determine scope and limitations of tests.
- Determine test approaches and methods - unit, integration, functional, system, security, load, usability tests, whichever are in scope.
- Determine test environment requirements (hardware, software, configuration, versions, communications, etc.)
- Determine testware requirements (automation tools, coverage analyzers, test tracking, problem/bug tracking, etc.)
- Determine test input data requirements
- Identify tasks, those responsible for tasks, and labor requirements
- Set initial schedule estimates, timelines, milestones where feasible.
- Determine, where appropriate, input equivalence classes, boundary value analyses, error classes
- Prepare test plan document(s) and have needed reviews/approvals
- Write test cases or test scenarios as needed.
- Have needed reviews/inspections/approvals of test cases/scenarios/approaches.
- Prepare test environment and testware, obtain needed user manuals/reference documents/configuration guides/installation guides, set up test tracking processes, set up logging and archiving processes, set up or obtain test input data
- Obtain/install/configure software releases
- Perform tests
- Evaluate and report results
- Track problems/bugs and fixes

- Retest as needed
- Maintain and update test plans, test cases, test environment, and testware through life cycle

What's a 'test plan'?
A software project test plan is a document that describes the objectives, scope, approach, and focus of a software testing effort. The process of preparing a test plan is a useful way to think through the efforts needed to validate the acceptability of a software product. The completed document will help people outside the test group understand the 'why' and 'how' of product validation. It should be thorough enough to be useful but not so overly detailed that no one outside the test group will read it. The following are some of the items that might be included in a test plan, depending on the particular project:

- Title
- Identification of software including version/release numbers
- Revision history of document including authors, dates, approvals
- Table of Contents
- Purpose of document, intended audience
- Objective of testing effort
- Software product overview
- Relevant related document list, such as requirements, design documents, other test plans, etc.
- Relevant standards or legal requirements
- Traceability requirements
- Relevant naming conventions and identifier conventions
- Overall software project organization and personnel/contact-info/responsibilities
- Test organization and personnel/contact-info/responsibilities
- Assumptions and dependencies
- Project risk analysis
- Testing priorities and focus
- Scope and limitations of testing
- Test outline - a decomposition of the test approach by test type, feature, functionality, process, system, module, etc. as applicable
- Outline of data input equivalence classes, boundary value analysis, error classes
- Test environment - hardware, operating systems, other required software, data configurations, interfaces to other systems
- Test environment validity analysis - differences between the test and production systems and their impact on test validity.
- Test environment setup and configuration issues
- Software migration processes
- Software CM processes
- Test data setup requirements
- Database setup requirements
- Outline of system-logging/error-logging/other capabilities, and tools such as screen capture software, that will be used to help describe and report bugs
- Discussion of any specialized software or hardware tools that will be used by testers to help track the cause or source of bugs
- Test automation - justification and overview
- Test tools to be used, including versions, patches, etc.
- Test script/test code maintenance processes and version control
- Problem tracking and resolution - tools and processes

- Project test metrics to be used
- Reporting requirements and testing deliverables
- Software entrance and exit criteria
- Initial sanity testing period and criteria
- Test suspension and restart criteria
- Personnel allocation
- Personnel pre-training needs
- Test site/location
- Outside test organizations to be utilized and their purpose, responsibilties, deliverables, contact persons, and coordination issues
- Relevant proprietary, classified, security, and licensing issues.
- Open issues
- Appendix - glossary, acronyms, etc.

What's a 'test case'?
A test case describes an input, action, or event and an expected response, to determine if a feature of a software application is working correctly. A test case may contain particulars such as test case identifier, test case name, objective, test conditions/setup, input data requirements, steps, and expected results. The level of detail may vary significantly depending on the organization and project context. Note that organizations vary considerably in their handling of test cases; many utilize less-detailed 'test scenarios' that allow for simpler and more adaptable/maintainable test documentation.

Note that the process of developing test cases can help find problems in the requirements or design of an application, since it requires completely thinking through the operation of the application. For this reason, it's useful to prepare test cases early in the development cycle if possible.

What should be done after a bug is found?
The bug needs to be communicated and assigned to developers that can fix it. After the problem is resolved, fixes should be re-tested, and determinations made regarding requirements for regression testing to check that fixes didn't create problems elsewhere. If a problem-tracking system is in place, it should encapsulate these processes. A variety of commercial problem-tracking/management software tools are available (see the 'Tools' section for web resources with listings of such tools). The following are items to consider in the tracking process:

- Complete information such that developers can understand the bug, get an idea of it's severity, and reproduce it if necessary.
- Bug identifier (number, ID, etc.)
- Current bug status (e.g., 'Released for Retest', 'New', etc.)
- The application name or identifier and version
- The function, module, feature, object, screen, etc. where the bug occurred
- Environment specifics, system, platform, relevant hardware specifics
- Test case or scenario information/name/number/identifier
- One-line bug description
- Full bug description
- Description of steps needed to reproduce the bug if not covered by a test case or automated test or if the developer doesn't have easy access to the test case/test script/test tool
- Names and/or descriptions of file/data/messages/etc. used in test

- File excerpts/error messages/log file excerpts/screen shots/test tool logs that would be helpful in finding the cause of the problem
- Severity estimate (a 5-level range such as 1-5 or 'critical'-to-'low' is common)
- Was the bug reproducible?
- Tester name
- Test date
- Bug reporting date
- Name of developer/group/organization the problem is assigned to
- Description of problem cause
- Description of fix
- Code section/file/module/class/method that was fixed
- Date of fix
- Application version that contains the fix
- Tester responsible for retest
- Retest date
- Retest results
- Regression testing requirements
- Tester responsible for regression tests
- Regression testing results

A reporting or tracking process should enable notification of appropriate personnel at various stages. For instance, testers need to know when retesting is needed, developers need to know when bugs are found and how to get the needed information, and reporting/summary capabilities are needed for managers.

What is 'configuration management'?
Configuration management covers the processes used to control, coordinate, and track: code, requirements, documentation, problems, change requests, designs, tools/compilers / libraries / patches, changes made to them, and who makes the changes.

What if the software is so buggy it can't really be tested at all?
The best bet in this situation is for the testers to go through the process of reporting whatever bugs or blocking-type problems initially show up, with the focus being on critical bugs. Since this type of problem can significantly affect schedules, and indicates deeper problems in the software development process (such as insufficient unit testing or insufficient integration testing, poor design, improper build or release procedures, etc.) managers should be notified, and provided with some documentation as evidence of the problem.

How can it be known when to stop testing?
This can be difficult to determine. Most modern software applications are so complex, and run in such an interdependent environment, that complete testing can never be done. Common factors in deciding when to stop are:

- Deadlines (release deadlines, testing deadlines, etc.)
- Test cases completed with certain percentage passed
- Test budget depleted
- Coverage of code/functionality/requirements reaches a specified point
- Bug rate falls below a certain level
- Beta or alpha testing period ends

What if there isn't enough time for thorough testing?
Use risk analysis, along with discussion with project stakeholders, to determine where testing should be focused. Since it's rarely possible to test every possible aspect of an application, every possible combination of events, every dependency, or everything that could go wrong, risk analysis is appropriate to most software development projects. This requires judgement skills, common sense, and experience. (If warranted, formal methods are also available.) Considerations can include:

- Which functionality is most important to the project's intended purpose?
- Which functionality is most visible to the user?
- Which functionality has the largest safety impact?
- Which functionality has the largest financial impact on users?
- Which aspects of the application are most important to the customer?
- Which aspects of the application can be tested early in the development cycle?
- Which parts of the code are most complex, and thus most subject to errors?
- Which parts of the application were developed in rush or panic mode?
- Which aspects of similar/related previous projects caused problems?
- Which aspects of similar/related previous projects had large maintenance expenses?
- Which parts of the requirements and design are unclear or poorly thought out?
- What do the developers think are the highest-risk aspects of the application?
- What kinds of problems would cause the worst publicity?
- What kinds of problems would cause the most customer service complaints?
- What kinds of tests could easily cover multiple functionalities?
- Which tests will have the best high-risk-coverage to time-required ratio?

What if the project isn't big enough to justify extensive testing?
Consider the impact of project errors, not the size of the project. However, if extensive testing is still not justified, risk analysis is again needed and the same considerations as described previously in 'What if there isn't enough time for thorough testing?' apply. The tester might then do ad hoc or exploratory testing, or write up a limited test plan based on the risk analysis.

How do distributed multi-tier environments affect testing?
Most current software being tested involves multi-tier and distributed applications which can be highly complex due to the multiple dependencies among systems, services, data communications, hardware, and servers. Thus testing requirements can be extensive. When time is limited (as it usually is) a focus on integration and system testing can be considered. Additionally, load/stress/performance testing may be useful in determining distributed application limitations and capabilities and where the limitations are. There are commercial and open source tools to assist with such testing

How should Web sites be tested?
Many modern web sites are essentially complex distributed systems with html, css, web services, encrypted communications, browser-side scripts/apps/libraries (such as javascript, flash, etc), the wide variety of applications/libraries/datastores that could run on the server side, load balancers, etc. Additionally, there are a wide variety of servers and browsers, mobile and other platforms, various versions of each, small but sometimes significant differences between them, variations in connection speeds, rapidly changing technologies, and multiple standards and protocols. Although web site testing was initially relatively simple years ago, testing of modern web site front ends, back end systems, mid-level tiers, web services, databases,

security, performance, etc, can be as complex as or more complex than any other type of application.

To assist in testing of web sites via their GUI's, many popular web browsers normally include a set of 'Developer Tools' that are helpful in testing and debugging, and in developing test automation scripts:

How is testing affected by object-oriented designs?
Well-engineered object-oriented design can make it easier to trace from code to internal design to functional design to requirements. While there will be little effect on black box testing (where an understanding of the internal design of the application is unnecessary), white-box testing can be oriented to the application's objects, methods, etc. If the application was well-designed this can simplify test design and test automation design.

What is Agile Software Development and how does it impact testing?
Agile Software Development generally refers to incremental, collaborative software development approaches that provide alternatives to 'heavyweight', documentation-driven, waterfall-type development practices. It grew out of such approaches as Extreme Programming, SCRUM, DSDM, Crystal, and other 'lightweight' methodologies. In 2001 a group of software development and test practitioners gathered to discuss lightweight methods and created the 'Agile Manifesto' which describes the Agile approach values and lists 12 principles that describe Agile software development. In reality many organizations implement these principles to widely varying degrees (and with widely varying degrees of success) and still call their approach 'Agile'. The impact of Agile approaches on software testing can also vary widely but often includes the following:

- Requirements and documentation are often minimal and when present are often in the form of high-level 'user stories' and 'acceptance tests'.
- Requirements can be added or changed often
- Iterative development/test cycles ('sprints') are often in the range of 1-3 weeks. Both new functionality testing and regression testing (preferably automated) may occur within each iterative cycle.
- Close collaboration between testers and developers, product owners and other team members
- Short daily project status 'standup' meetings that include testers.
- Common testing-related practices in agile projects may include test-driven development, extensive unit testing and unit test automation, API-level test automation, exploratory and session-based testing, UI test automation.
- Testers may be heavily involved in fleshing out requirements details, including both functional and non-functional requirements.

Why is it sometimes hard for organizations to get serious about quality assurance?
Solving problems is a high-visibility process; preventing problems is low-visibility. This is illustrated by an old parable:

In ancient China there was a family of healers, one of whom was known throughout the land and employed as a physician to a great lord. The physician was asked which of his family was the most skillful healer. He replied, "I tend to the sick and dying with drastic and dramatic treatments, and on occasion someone is cured and my name gets out among the lords."

"My elder brother cures sickness when it just begins to take root, and his skills are known among the local peasants and neighbors."

"My eldest brother is able to sense the spirit of sickness and eradicate it before it takes form. His name is unknown outside our home."

This is a problem in any business, but it's a particularly difficult problem in the software industry. Software quality problems are often not as readily apparent as they might be in the case of an industry with more physical products, such as auto manufacturing or home construction.

Additionally: Many organizations are able to determine who is skilled at fixing problems, and then reward such people. However, determining who has a talent for preventing problems in the first place, and figuring out how to incentivize such behavior, is a significant challenge.

Who is responsible for risk management?
Risk management means the actions taken to avoid things going wrong on a software development project, things that might negatively impact the scope, quality, timeliness, or cost of a project. This is, of course, a shared responsibility among everyone involved in a project. However, there needs to be a 'buck stops here' person who can consider the relevant tradeoffs when decisions are required, and who can ensure that everyone is handling their risk management responsibilities.

It is not unusual for the term 'risk management' to never come up at all in a software organization or project. If it does come up, it's often assumed to be the responsibility of QA or test personnel. Or there may be a 'risks' or 'issues' section of a project, QA, or test plan, and it's assumed that this means that risk management has taken place.

The issues here are similar to those for the LFAQ question "Who should decide when software is ready to be released?" It's generally NOT a good idea for a test lead, test manager, or QA manager to be the 'buck stops here' person for risk management. Typically QA/Test personnel or managers are not managers of developers, analysts, designers and many other project personnel, and so it would be difficult for them to ensure that everyone on a project is handling their risk management responsibilities. Additionally, knowledge of all the considerations that go into risk management mitigation and tradeoff decisions is rarely the province of QA/Test personnel or managers. Based on these factors, the project manager is usually the most appropriate 'buck stops here' risk management person. QA/Test personnel can, however, provide input to the project manager. Such input could include analysis of quality-related risks, risk monitoring, process adherence reporting, defect reporting, and other information.

Who should decide when software is ready to be released?
In many projects this depends on the release criteria for the software. Such criteria are often in turn based on the decision to end testing, discussed in FAQ #2 item "How can it be known when to stop testing?" Unfortunately, for any but the simplest software projects, it is nearly impossible to adequately specify useful criteria without a significant amount of assumptions and subjectivity. For example, if the release criteria are based on passing a certain set of tests, there is likely an assumption that the tests have adequately addressed all appropriate software risks.

For most software projects, this would of course be impossible without enormous expense, so this assumption would be a large leap of faith. Additionally, since most software projects involve a balance of quality, timeliness, and cost, testing alone cannot address how to balance all three of these competing factors when release decisions are needed.

A typical approach is for a lead tester or QA or Test manager to be the release decision maker. This again involves significant assumptions - such as an assumption that the test manager understands the spectrum of considerations that are important in determining whether software quality is 'sufficient' for release, or the assumption that quality does not have to be balanced with timeliness and cost. In many organizations, 'sufficient quality' is not well defined, is extremely subjective, may have never been usefully discussed, or may vary from project to project or even from day to day.

Release criteria considerations can include deadlines, sales goals, business/market/competitive considerations, business segment quality norms, legal requirements, technical and programming considerations, end-user expectations, internal budgets, impacts on other organization projects or goals, and a variety of other factors. Knowledge of all these factors is often shared among a number of personnel in a large organization, such as the project manager, director, customer service manager, technical lead or manager, marketing manager, QA manager, etc. In smaller organizations or projects it may be appropriate for one person to be knowledgeable in all these areas, but that person is typically a project manager, not a test lead or QA manager.

For these reasons, it's generally not a good idea for a test lead, test manager, or QA manager to decide when software is ready to be released. Their responsibility should be to provide input to the appropriate person or group that makes a release decision. For small organizations and projects that person could be a product manager, a project manager, or similar manager. For larger organizations and projects, release decisions might be made by a committee of personnel with sufficient collective knowledge of the relevant considerations.

What can be done if requirements are changing continuously?
This is a common problem for organizations where there are expectations that requirements can be pre-determined and remain stable. If these expectations are reasonable, here are some approaches:

- Work with the project's stakeholders early on to understand how requirements might change so that alternate test plans and strategies can be worked out in advance, if possible.
- It's helpful if the application's initial design allows for some adaptability so that later changes do not require redoing the application from scratch.
- If the code is well-commented and well-documented this makes changes easier for the developers.
- Use some type of rapid prototyping whenever possible to help customers feel sure of their requirements and minimize changes.
- The project's initial schedule should allow for some extra time commensurate with the possibility of changes.
- Try to move new requirements to a 'Phase 2' version of an application, while using the original requirements for the 'Phase 1' version.
- Negotiate to allow only easily-implemented new requirements into the project, while moving more difficult new requirements into future versions of the application.
- Be sure that customers and management understand the scheduling impacts, inherent

risks, and costs of significant requirements changes. Then let management or the customers (not the developers or testers) decide if the changes are warranted - after all, that's their job.

- Balance the effort put into setting up automated testing with the expected effort required to refactor them to deal with changes.
- Try to design some flexibility into automated test scripts.
- Focus initial automated testing on application aspects that are most likely to remain unchanged.
- Devote appropriate effort to risk analysis of changes to minimize regression testing needs.
- Design some flexibility into test cases (this is not easily done; the best bet might be to minimize the detail in the test cases, or set up only higher-level generic-type test plans)
- Focus less on detailed test plans and test cases and more on ad hoc testing (with an understanding of the added risk that this entails).

If this is a continuing problem, and the expectation that requirements can be pre-determined and remain stable is NOT reasonable, it may be a good idea to figure out why the expectations are not aligned with reality, and to refactor an organization's or project's software development process to take this into account. It may be appropriate to consider agile development approaches.

What if the application has functionality that wasn't in the requirements?
It may take serious effort to determine if an application has significant unexpected or hidden functionality, and it could indicate deeper problems in the software development process. If the functionality isn't necessary to the purpose of the application, it should be removed, as it may have unknown impacts or dependencies that were not taken into account by the designer or the customer. (If the functionality is minor and low risk then no action may be necessary.) If not removed, information will be needed to determine risks and to determine any added testing needs or regression testing needs. Management should be made aware of any significant added risks as a result of the unexpected functionality.

This problem is a standard aspect of projects that include COTS (Commercial Off-The-Shelf) software or modified COTS software. The COTS part of the project will typically have a large amount of functionality that is not included in project requirements, or may be simply undetermined. Depending on the situation, it may be appropriate to perform in-depth analysis of the COTS software and work closely with the end user to determine which pre-existing COTS functionality is important and which functionality may interact with or be affected by the non-COTS aspects of the project. A significant regression testing effort may be needed (again, depending on the situation), and automated regression testing may be useful.

How can Software QA processes be implemented without reducing productivity?
By implementing QA processes slowly over time, using consensus to reach agreement on processes, focusing on processes that align tightly with organizational goals, and adjusting/experimenting/refactoring as an organization matures, productivity can be improved instead of stifled. Problem prevention will lessen the need for problem detection, panics and burn-out will decrease, and there will be improved focus and less wasted effort. At the same time, attempts should be made to keep processes simple and efficient, avoid a 'Process Police' mentality, minimize paperwork, promote computer-based processes and automated tracking and reporting, minimize time required in meetings, and promote training as part of the QA process. However, no one - especially talented technical types - likes rules or bureaucracy, and in the short run things may slow down a bit. A typical scenario would be that more days of

planning, reviews, and inspections will be needed, but less time will be required for late-night bug-fixing and handling of irate customers.

Other possibilities include incremental self-managed team approaches such as 'Kaizen' methods of continuous process improvement, the Deming-Shewhart Plan-Do-Check-Act cycle, and others.

What if an organization is growing so fast that fixed QA processes are impossible?
This is a common problem in the software industry, especially in new technology areas. There is generally no easy solution in this situation. One approach is:

- Hire good people
- Management should 'ruthlessly prioritize' quality issues and maintain focus on the customer
- Everyone in the organization should be clear on what 'quality' means to the customer

Depending on the growth rate, it is possible that incremental self-managed team approaches may be applicable, such as 'Kaizen' methods of continuous process improvement, or the Deming-Shewhart Plan-Do-Check-Act cycle, and others.

Will automated testing tools make testing easier?

- Possibly. For small projects, the time needed to learn and implement them may not be worth it unless personnel are already familiar with the tools. For larger projects, or ongoing long-term projects they can be valuable.
- Most test automation tools utilize a standard coding language such as ruby, python, Java, etc or a proprietary scripting language specific to the tool. Sometimes initial tests can be 'recorded' such that the test scripts are automatically generated. and then modified as needed. One of the challenges of automated testing is that if there are continual changes to the system being tested, the test automation code may have to be changed so often that it becomes very time-consuming (thus expensive) to continuously update the scripts. Additionally, interpretation and analysis of results (screens, reports, data, logs, etc.) can be a difficult task. Note that there are test automation tools and frameworks for web and UI interfaces as well as text-based and back-end interfaces, and for all types of platforms.
- A common type of approach for automation of functional testing is 'data-driven' or 'keyword-driven' automated testing, in which the test drivers are separated from the data and/or actions utilized in testing (an 'action' would be something like 'enter a value in a text box'). Test drivers can be in the form of automated test tools or frameworks or custom-written testing software. The data and actions can be more easily maintained - such as via a spreadsheet - since they are separate from the test drivers. The test drivers 'read' the data/action information to perform specified tests. This approach can enable more efficient control, development, documentation, and maintenance of automated tests/test cases.
- Other automated tools can include:
  - code analyzers - monitor code complexity, adherence to standards, etc.
  - coverage analyzers - these tools check which parts of the code have been exercised by a test, and may be oriented to code statement coverage, condition coverage, path coverage, etc.
  - memory analyzers - such as bounds-checkers and leak detectors.
  - load/performance test tools - for testing client/server and web applications under various load levels.

- web test tools - to check that links are valid, HTML code usage is correct, client-side and server-side programs work, a web site's interactions are secure.
- other tools - for test case management, BDT (behavior-driven testing), documentation, management, bug reporting, and configuration management, file and database comparisons, screen captures, security testing, macro recorders, etc.

Test automation is, of course, possible without COTS tools. Many successful automation efforts utilize open source tools, or custom automation software that is targeted for specific projects, specific software applications, or a specific organization's software development environment. In test-driven agile software development environments, automated tests are often built into the software during (or preceding) coding of the application.

What's the best way to choose a test automation tool?
It's easy to get caught up in enthusiasm for the 'silver bullet' of test automation, where the dream is that a single mouse click can initialize thorough unattended testing of an entire software application, bugs will be automatically reported, and easy-to-understand summary reports will be waiting in the manager's in-box in the morning.
Although that may in fact be possible in some situations, it is not the way things generally play out.

In manual testing, the test engineer exercises software functionality to determine if the software is behaving in an expected way. This means that the tester must be able to judge what the expected outcome of a test should be, such as expected data outputs, screen messages, changes in the appearance of a User Interface, XML files, database changes, etc. In an automated test, the computer does not have human-like 'judgement' capabilities to determine whether or not a test outcome was correct. This means there must be a mechanism by which the computer can do an automatic comparison between actual and expected results for every automated test scenario and unambiguously make a pass or fail determination. This factor may require a significant change in the entire approach to testing, since in manual testing a human is involved and can:

- make mental adjustments to expected test results based on variations in the pre-test state of the software system
- often make on-the-fly adjustments, if needed, to data used in the test
- make pass/fail judgements about results of each test
- make quick judgements and adjustments for changes to requirements.
- make a wide variety of other types of judgements and adjustments as needed.

For those new to test automation, it might be a good idea to do some reading or training first. There are a variety of ways to go about doing this; some example approaches are:

- Read through information on the web about test automation such as general information available on some test tool vendor sites or some of the automated testing articles listed in the 'Other Resources' Automation section.
- Read some books on test automation such as those listed in the Softwareqatest.com Bookstore
- Obtain some test tool trial versions or low cost or open source test tools and experiment with them

- Attend software testing conferences or training courses related to test automation

As in anything else, proper planning and analysis are critical to success in choosing and utilizing an automated test tool. Choosing a test tool just for the purpose of 'automating testing' is not useful; useful purposes might include: testing more thoroughly, testing in ways that were not previously feasible via manual methods (such as load testing), testing faster, enabling continuous integration processes, or reducing excessively tedious manual testing. Automated testing rarely enables savings in the cost of testing, although it may result in software lifecycle savings (or increased sales) just as with any other quality-related initiative.
With the proper background and understanding of test automation, the following considerations can be helpful in choosing a test tool (automated testing will not necessarily resolve them, they are only considerations for automation potential):

- Analyze the current non-automated testing situation to determine where testing is not being done or does not appear to be sufficient
- Where is current testing excessively time-consuming?
- Where is current testing excessively tedious?
- What kinds of problems are repeatedly missed with current testing?
- What testing procedures are carried out repeatedly (such as regression testing or security testing)?
- What testing procedures are not being carried out repeatedly but should be?
- What test tracking and management processes can be implemented or made more effective through the use of an automated test tool?

Taking into account the testing needs determined by analysis of these considerations and other appropriate factors, the types of desired test tools can be determined. For each type of test tool (such as functional test tool, load test tool, etc.) the choices can be further narrowed based on the characteristics of the software application. The relevant characteristics will depend, of course, on the situation and the type of test tool and other factors. Such characteristics could include the operating system, GUI components, development languages, web server type, etc. Other factors affecting a choice could include experience level and capabilities of test personnel, advantages/disadvantages in developing a custom automated test tool, tool costs, tool quality and ease of use, usefulness of the tool on other projects, etc.

Once a short list of potential test tools is selected, several can be utilized on a trial basis for a final determination. Any expensive test tool should be thoroughly analyzed during its trial period to ensure that it is appropriate and that it's capabilities and limitations are well understood. This may require significant time or training, but the alternative is to take a major risk of a mistaken investment (in terms of time, resources, and/or purchase price).

How can it be determined if a test environment is appropriate?
This is a difficult question in that it typically involves tradeoffs between 'better' test environments and cost. The ultimate situation would be a collection of test environments that mimic exactly all possible hardware, software, network, data, and usage characteristics of the expected live environments in which the software will be used. For many software applications, this would involve a nearly infinite number of variations, and would clearly be impossible. And for new software applications, it may also be impossible to predict all the variations in environments in which the application will run. For very large, complex systems, duplication of a 'live' type of environment may be prohibitively expensive.

In reality judgements must be made as to which characteristics of a software application environment are important, and test environments can be selected on that basis after taking into account time, budget, and logistical constraints. Such judgements are preferably made by those who have the most appropriate technical knowledge and experience, along with an understanding of risks and constraints.

For smaller or low risk projects, an informal approach is common, but for larger or higher risk projects (in terms of money, property, or lives) a more formalized process involving multiple personnel and significant effort and expense may be appropriate.

In some situations it may be possible to mitigate the need for maintenance of large numbers of varied test environments. One approach might be to coordinate internal testing with beta testing efforts. Another possible mitigation approach is to provide built-in automated tests that run automatically upon installation of the application by end-users. These tests might then automatically report back information, via the internet, about the application environment and problems encountered. Another possibility is the use of virtual environments instead of physical test environments, using such tools as VMWare or VirtualBox.

What's the best approach to software test estimation?
There is no simple answer for this. The 'best approach' is highly dependent on the particular organization and project and the experience of the personnel involved.
For example, given two software projects of similar complexity and size, the appropriate test effort for one project might be very large if it was for life-critical medical equipment software, but might be much smaller for the other project if it was for a low-cost computer game. A test estimation approach that only considered size and complexity might be appropriate for one project but not for the other. Following are some approaches to consider:

Implicit Risk Context Approach:
A typical approach to test estimation is for a project manager or QA manager to implicitly use risk context, in combination with past personal experiences in the organization, to choose a level of resources to allocate to testing. In many organizations, the 'risk context' is assumed to be similar from one project to the next, so there is no explicit consideration of risk context. (Risk context might include factors such as the organization's typical software quality levels, the software's intended use, the experience level of developers and testers, etc.) This is essentially an intuitive guess based on experience.

Metrics-Based Approach:
A useful approach is to track past experience of an organization's various projects and the associated test effort that worked well for projects. Once there is a set of data covering characteristics for a reasonable number of projects, then this 'past experience' information can be used for future test project planning. (Determining and collecting useful project metrics over time can be an extremely difficult task.) For each particular new project, the 'expected' required test time can be adjusted based on whatever metrics or other information is available, such as function point count, number of external system interfaces, unit testing done by developers, risk levels of the project, etc. In the end, this is essentially 'judgement based on documented experience', and is not easy to do successfully.

Test Work Breakdown Approach:
Another common approach is to decompose the expected testing tasks into a collection of small tasks for which estimates can, at least in theory, be made with reasonable accuracy. This of

course assumes that an accurate and predictable breakdown of testing tasks and their estimated effort is feasible. In many large projects, this is not the case. For example, if a large number of bugs are being found in a project, this will add to the time required for testing, retesting, bug analysis and reporting. It will also add to the time required for development, and if development schedules and efforts do not go as planned, this will further impact testing.

Iterative Approach:
In this approach for large test efforts, an initial rough testing estimate is made. Once testing begins, a more refined estimate is made after a small percentage (e.g., 1%) of the first estimate's work is done. At this point testers have obtained additional test project knowledge and a better understanding of issues, general software quality, and risk. Test plans and schedules can be refactored if necessary and a new estimate provided. Then a yet-more-refined estimate is made after a somewhat larger percentage (e.g., 2%) of the new work estimate is done. Repeat the cycle as necessary/appropriate.

Percentage-of-Development Approach:
Some organizations utilize a quick estimation method for testing based on the estimated programming effort. For example, if a project is estimated to require 1000 hours of programming effort, and the organization normally finds that a 40% ratio for testing is appropriate, then an estimate of 400 hours for testing would be used. This approach may or may not be useful depending on the project-to-project variations in risk, personnel, types of applications, levels of complexity, etc.

Successful test estimation is a challenge for most organizations, since few can accurately estimate software project development efforts, much less the testing effort of a project. It is also difficult to attempt testing estimates without first having detailed information about a project, including detailed requirements, the organization's experience with similar projects in the past, and an understanding of what should be included in a 'testing' estimation for a project (functional testing? unit testing? reviews? inspections? load testing? security testing?)

With agile software development approaches, test effort estimations may be unnecessary if pure test-driven development is utilized. However, it is not uncommon to have a mix of some automated positive-type unit tests, along with some type of separate manual or automated functional testing. In general, agile-based projects by their nature will not be heavily dependent on large one-shot testing efforts, since they emphasize the construction of releasable software in short iteration cycles. Test estimates are often focused on individual 'stories' and the testing associated with each of these. These smaller multiple test effort estimates may not be as difficult to estimate and the impact of inaccurate estimates will be less severe, and expectations are that estimates will improve with each sprint.

CS 706 - Final Term 2015

1. Explain unit testing
2. Briefly explain defect arrival rate
3. Explain testing strategies
4. Explain step before process improvement
5. Explain mean time failure, defect density, defect severity
6. Given a pictorial view of planning and management, explain schedule monitoring
7. Establishment of CCB and SCM in accepting test cycle
8. What is structural, problem and algorithm complexity? (5)
9. Explain steering committee, project risks. (10)
10. black box testing.(5)
11. What are the board categories of software testing.(10)
12. What is the integration testing (10)
13. Explain work effort, changing in requirements. (10)
14. Explain review and audits in planning and organization. (5)
15. How to audit. (5)
16. White box testing and approaches
17. Attributes of Software Quality
18. Equivalence Partitioning
19. SCM Function Status Accounting/Reporting
20. SCM Staffing
21. Project Tracking, Estimation, Effective Communication, Steering Committee, Project Risks.
22. Problem complexity, Algorithmic complexity, Structural complexity ,Cognitive complexity