UNIT-V

Metrics for Process and Products: Software measurement, metrics for software quality.
Risk management: Reactive Vs proactive risk strategies, software risks, risk identification, risk projection, risk refinement, RMMM, RMMM plan.
Quality Management: Quality concepts, software quality assurance, software reviews, formal technical reviews, statistical software quality assurance, software reliability, the ISO 9000 quality standards.

METRICS FOR PROCESS AND PROJECTS

SOFTWARE MEASUREMENT

Software measurement can be categorized in two ways.

(1) ***Direct measures*** of the software engineering process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time.

(2) ***Indirect measures*** of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities"

Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the *size* of the software that has been produced. To develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size- oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code).
- Defects per KLOC.
- $ per LOC.
- Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

- Errors per person-month.
- LOC per person-month.
- $ per page of documentation.

Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since ‘functionality’ cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht, who suggested a measure called the ***function point.*** **Function points** are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

Proponents claim that FP is programming language independent, making it ideal for application using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach.

Opponents claim that the method requires some "sleight of hand " in that computation is basedsubjective rather than objective data, that counts of the information domain can be difficult to collect after the fact, and that FP has no direct physical meaning- it's just a number.

Typical Function-Oriented Metrics:

- errors per FP (thousand lines of code)
- defects per FP
- $ per FP
- pages of documentation per FP
- FP per person-month

Reconciling Different Metrics Approaches

The relationship between lines of code and function points depend upon the programming language that is used to implement the software and the quality of the design.

Function points and LOC based metrics have been found to be relatively accurate predictors of software development effort and cost.

Object Oriented Metrics:

Conventional software project metrics (LOC or FP) can be used to estimate object oriented software projects. Lorenz and Kidd suggest the following set of metrics for OO projects:

- **Number of scenario scripts**: A scenario script is a detailed sequence of steps that describes the interaction between the user and the application.
- **Number of key classes**: Key classes are the "highly independent components that are defined early in object-oriented analysis.
- **Number of support classes**: Support classes are required to implement the system but are not immediately related to the problem domain.
- **Average number of support classes per key class**: Of the average number of support classes per key class were known for a given problem domain estimation would be much simplified. Lorenz and Kidd suggest that applications with a GUI have between two and three times the number of support classes as key classes.
- **Number of subsystems**: A subsystem is an aggregation of classes that support a function that is visible to the end-user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in ehic work on subsystems is partitioned among project staff.

Use-Case Oriented Metrics

Use-cases describe user-visible functions and features that are basic requirements for a system. The use-cases is directly proportional to the size of the application in LOC and to the number of use-cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

Because use-cases can be created at vastly different levels of abstraction, there is no standard size for a use-case. Without a standard measure of what a use-case is, its application as a normalization measure is suspect.

Web Engineering Project Metrics

The objective of all web engineering projects is to build a Web application that delivers a combination of content and functionality to the end-user.

- Number of static Web pages: These pages represent low relative complexity and generally require less effort to construct than dynamic pages. This measures provides an indication of the overall size of the application and the effort required to develop it.
- Number of dynamic Web pages: : Web pages with dynamic content are essential in all e- commerce applications, search engines, financial application, and many other Web App categories. These pages represent higher relative complexity and require more effort to construct than static pages. This measure provides an indication of the overall size of the application and the effort required to develop it.
- Number of internal page link: Internal page links are pointers that provide an indication of the degree of architectural coupling within the Web App.
- Number of persistent data objects: As the number of persistent data objects grows, the complexity of the Web App also grows, and effort to implement it increases proportionally.
- Number of external systems interfaced: As the requirement for interfacing grows, system complexity and development effort also increase.
- Number of static content objects: Static content objects encompass static text-based, graphical, video, animation, and audio information that are incorporated within the Web App.
- Number of dynamic content objects: Dynamic content objects are generated based on end-user actions and encompass internally generated text-based, graphical, video, animation, and audio information that are incorporated within the Web App.
- Number of executable functions: An executable function provides some computational service to the end-user. As the number of executable functions increases, modeling and construction effort also increase.

METRICS FOR SOFTWARE QUALITY

The overriding goal of software engineering is to produce a high-quality system, application, or product within a timeframe that satisfies a market need. To achieve this goal, software engineers must apply effective methods coupled with modern tools within the context of a mature software process.

Measuring Quality

The measures of software quality are correctness, maintainability, integrity, and usability. These measures will provide useful indicators for the project team.

- **Correctness.** Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.
- **Maintainability.** Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. A simple timeoriented metric is ***mean-time-tochange*** (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users.
- **Integrity.** Attacks can be made on all three components of software: programs, data, and documents.

  To measure integrity, two additional attributes must be defined: threat and security. *Threat* is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. *Security* is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as integrity = $\sum$ [1 –(threat × (1 – security))]
- **Usability:** Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics:

Defect Removal Efficiency

A quality metric that provides benefit at both the project and process level is defect removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities. When considered for a project as a whole, DRE is defined in the following manner:

DRE = $E/(E + D)$ where E is the number of errors found before delivery of the software to the end-user and D is the number of defects found after delivery.

Those errors that are not found during the review of the analysis model are passed on to the design task (where they may or may not be found). When used in this context, we redefine DRE as

DREi = Ei/(Ei+Ei+1)

E_i is the number of errors found during software engineering activity *i* and E_{i+1} is the number of errors found during software engineering activity *i+1* that are traceable to errors that were not discovered in software engineering activity *i.*

A quality objective for a software team (or an individual software engineer) is to achieve DRE that approaches 1. That is, errors should be filtered out before they are passed on to the next activity.

RISK MANAGEMENT

1) REACTIVE VS. PROACTIVE RISK STRATEGIES

At best, a **reactive strategy** monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a *fire fighting mode.*

- project team reacts to risks when they occur
- mitigation—plan for additional resources in anticipation of fire fighting
- fix on failure—resource are found and applied when the risk strikes
- crisis management—failure does not respond to applied resources and project is in jeopardy

A **proactive strategy** begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk.

- formal risk analysis is performed
- organization corrects the root causes of risk o examining risk sources that lie beyond the bounds of the software o developing the skill to manage change

Risk Management Paradigm

SOFTWARE RISK

Risk always involves two characteristics

Uncertainty—the risk may or may not happen; that is, there are no 100% probable risks ***Loss***—if the risk becomes a reality, unwanted consequences or losses will occur.

When risks are analyzed, it is important to quantify the level of uncertainty in the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

Project risks threaten the project plan. That is, if project risks become real, it is likely that project schedule will slip and that costs will increase.

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems.

Business risks threaten the viability of the software to be built. Business risks often jeopardize the project or the product. Candidates for the top five business risks are

(1) Building a excellent product or system that no one really wants (market risk),
(2) Building a product that no longer fits into the overall business strategy for the company (strategic risk),
(3) Building a product that the sales force doesn't understand how to sell,
(4) Losing the support of senior management due to a change in focus or a change in people (management risk), and
(5) Losing budgetary or personnel commitment (budget risks).

Known risks are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources.

Predictable risks are extrapolated from past project experience.

Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

2) RISK IDENTIFICATION

Risk identification is a systematic attempt to specify threats to the project plan. There are two distinct types of risks.

1) Generic risks and
2) product-specific risks.

Generic risks are a potential threat to every software project.

Product-specific risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the project that is to be built. Known and predictable risks in the following generic subcategories:

- ***Product size***—risks associated with the overall size of the software to be built or modified.

- ***Business impact***—risks associated with constraints imposed by management or the marketplace.
- ***Customer characteristics***—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
- ***Process definition***—risks associated with the degree to which the software process has been defined and is followed by the development organization.
- ***Development environment***—risks associated with the availability and quality of the tools to be used to build the product.
- ***Technology to be built***—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- ***Staff size and experience***—risks associated with the overall technical and project experience of the software engineers who will do the work.

Assessing Overall Project Risk

The questions are ordered by their relate importance to the success of a project.

1. Have top software and customer managers formally committed to support the project?
2. Are end-users enthusiastically committed to the project and the system/product to be built?
3. Are requirements fully understood by the software engineering team and their customers?
4. Have customers been involved fully in the definition of requirements?
5. Do end-users have realistic expectations?
6. Is project scope stable?
7. Does the software engineering team have the right mix of skills?
8. Are project requirements stable?
9. Does the project team have experience with the technology to be Implemented?
10. Is the number of people on the project team adequate to do the job?
11. Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

3.2 Risk Components and Drivers

The risk components are defined in the following manner:

- ***Performance risk***—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- ***Cost risk***—the degree of uncertainty that the project budget will be maintained.
- ***Support risk***—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
- ***Schedule risk***—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

The impact of each risk driver on the risk component is divided into one of four impact categories— negligible, marginal, critical, or catastrophic.

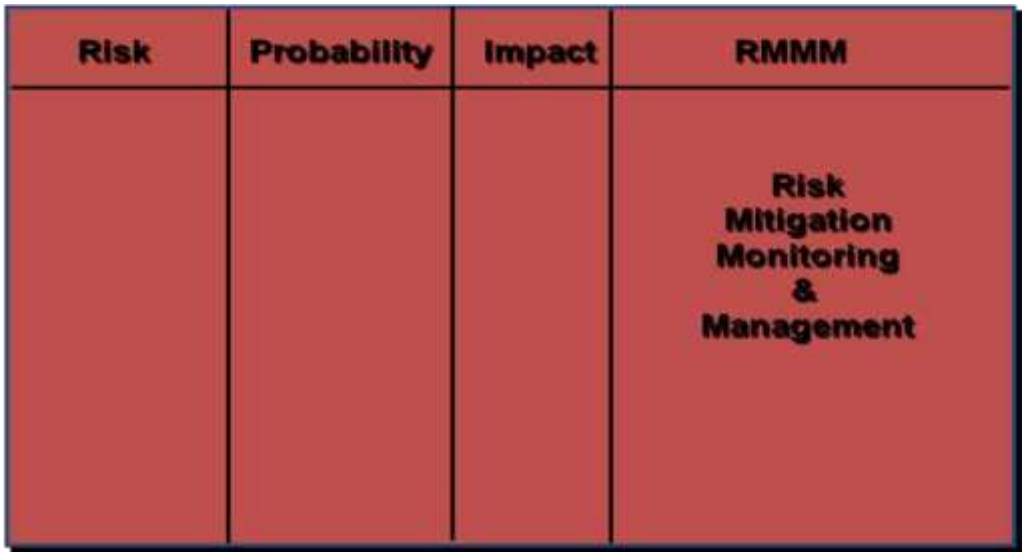
3) RISK PROJECTION

Risk projection, also called ***risk estimation,*** attempts to rate each risk in two ways—the likelihood or probability that the risk is real and the consequences of the problems associated with the risk, should it occur.

The project planner, along with other managers and technical staff, performs four risk projection activities:

(1) establish a scale that reflects the perceived likelihood of a risk,
(2) delineate the consequences of the risk,
(3) estimate the impact of the risk on the project and the product, and
(4) note the overall accuracy of the risk projection so that there will be no misunderstandings.

Developing a Risk Table Building a Ris



- A project team begins by listing all risks (no matter how remote) in the first column of the table.
- Each risk is categorized in Next; the impact of each risk is assessed.
- The categories for each of the four risk components—performance, support, cost, and schedule— are averaged to determine an overall impact value.
- High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom. This accomplishes first-order risk prioritization.

The project manager studies the resultant sorted table and defines a cutoff line.

The ***cutoff line*** (drawn horizontally at some point in the table) implies that only risks that lie above the line will be given further attention. Risks that fall below the line are re-evaluated to accomplish second-order prioritization.

Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing.

The ***nature*** of the risk indicates the problems that are likely if it occurs.
The ***scope*** of a risk combines the severity (just how serious is it?) with its overall distribution. ✓ Finally, the ***timing*** of a risk considers when and for how long the impact will be felt.

The overall ***risk exposure,*** RE*,* is determined using the following relationship
RE = *P* x *C*
Where *P* is the probability of occurrence for a risk, and *C* is the cost to the project should the risk occur.

Risk identification. Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Risk probability. 80% (likely).

Risk impact. 60 reusable software components were planned.

Risk exposure. *RE* = 0.80 x 25,200 ~ $20,200.

The total risk exposure for all risks (above the cutoff in the risk table) can provide a means for adjusting the final cost estimate for a project etc.

4) RISK REFINEMENT

One way for risk refinement is to represent the risk in *condition-transition-consequence(CTC)* format.
This general condition can be refined in the following manner:
Sub condition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.
Sub condition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.
Sub condition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

5) RISK MITIGATION, MONITORING, AND MANAGEMENT

An effective strategy must consider three issues:

- Risk avoidance
- Risk monitoring

- Risk management and contingency planning

If a software team adopts a proactive approach to risk, avoidance is always the best strategy. To mitigate this risk, project management must develop a strategy for reducing turnover. Among the possible steps to be taken are

- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
- Mitigate those causes that are under our control before the project starts.
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.
- Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner.
- Conduct peer reviews of all work (so that more than one person is "up to speed"). • Assign a backup staff member for every critical technologist.

As the project proceeds, risk monitoring activities commence. The following factors can be monitored:

- General attitude of team members based on project pressures.
- The degree to which the team has jelled.
- Interpersonal relationships among team members.
- Potential problems with compensation and benefits
- The availability of jobs within the company and outside it. •

Software safety **and** ***hazard analysis*** are software quality assurance activities that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

6) THE RMMM PLAN

A risk management strategy can be included in the software project plan or the risk management steps can be organized into a separate *Risk Mitigation, Monitoring and Management Plan.*

The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan.

Risk monitoring is a project tracking activity with three primary objectives:

1) to assess whether predicted risks do, in fact, occur;
2) to ensure that risk aversion steps defined for the risk are being properly applied; and 3) to collect information that can be used for future risk analysis.

QUALITY MANAGEMENT

1) QUALITY CONCEPTS:

Quality management encompasses
(1) a quality management approach,
(2) effective software engineering technology (methods and tools),
(3) formal technical reviews that are applied throughout the software process,
(4) a multitiered testing strategy,
(5) control of software documentation and the changes made to it,
(6) a procedure to ensure compliance with software development standards (when applicable), and
(7) measurement and reporting mechanisms.

Variation control is the heart of quality control.

Quality

The *American Heritage Dictionary* defines *quality* as "a characteristic or attribute of something." ✓ ***Quality of design*** refers to the characteristics that designers specify for an item.

Quality of conformance is the degree to which the design specifications are followed during manufacturing.

In software development, quality of design encompasses requirements, specifications, and the design of the system. Quality of conformance is an issue focused primarily on implementation. If the implementation follows the design and the resulting system meets its requirements and performance goals, conformance quality is high.
Robert Glass argues that a more "intuitive" relationship is in order:

User satisfaction = compliant product + good quality + delivery within budget and schedule

Quality Control

Quality control involves the series of inspections, reviews, and tests used throughout the software process to ensure each work product meets the requirements placed upon it.
A key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process. The feedback loop is essential to minimize the defects produced.

Quality Assurance

Quality assurance consists of the auditing and reporting functions that assess the effectiveness and completeness of quality control activities. The **goal of quality** assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals.

Cost of Quality

The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities.

Quality costs may be divided into costs associated with prevention, appraisal, and failure.

Prevention costs include

- quality planning
- formal technical reviews
- test equipment
- training

Appraisal costs include activities to gain insight into product condition the "first time through" each process. Examples of appraisal costs include

- in-process and interprocess inspection
- equipment calibration and maintenance
- testing

Failure costs are those that would disappear if no defects appeared before shipping a product to customers.

Failure costs may be subdivided into internal failure costs and external failure costs.

Internal failure costs are incurred when we detect a defect in our product prior to shipment. Internal failure costs include

- rework
- repair
- failure mode analysis

External failure costs are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are

- complaint resolution
- product return and replacement
- help line support
- warranty work

2) SOFTWARE QUALITY ASSURANCE

Software quality is defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

The definition serves to emphasize three important points:

1) Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2) Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3) A set of implicit requirements often goes unmentioned (e.g., the desire for ease of use and good maintainability). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

Background Issues

The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world. During the 1940s, more formal approaches to quality control were suggested. These relied on measurement and continuous process improvement as key elements of quality management.Today, every company has mechanisms to ensure quality in its products.

During the early days of computing (1950s and 1960s), quality was the sole responsibility of the programmer. Standards for quality assurance for software were introduced in military contract software development during the 1970s.

Extending the definition presented earlier, software quality assurance is a "planned and systematic pattern of actions" that are required to ensure high quality in software. The scope of quality assurance responsibility might best be characterized by paraphrasing a once-popular automobile commercial: "Quality Is Job #1." The implication for software is that many different constituencies have software quality assurance responsibility—software engineers, project managers, customers, salespeople, and the individuals who serve within an SQA group.

The SQA group serves as the customer's in-house representative. That is, the people who perform

SQA must look at the software from the customer's point of view

SQA Activities

Software quality assurance is composed of a variety of tasks associated with two different constituencies—

the software engineers who do technical work and

an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

The Software Engineering Institute recommends a set of SQA activities that address quality assurance planning, oversight, record keeping, analysis, and reporting. These activities are performed (or facilitated) by an independent SQA group that conducts the following activities.

Prepares an SQA plan for a project. The plan is developed during project planning and is reviewed by all interested parties. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies

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

Participates in the development of the project's software process description. The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

Reviews software engineering activities to verify compliance with the defined software process. The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process. The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure. Deviations may be encountered in the project plan, process description, applicable standards, or technical work products.

Records any noncompliance and reports to senior management. Noncompliance items are tracked until they are resolved.

3) SOFTWARE REVIEWS

Software reviews are a "filter" for the software engineering process. That is, reviews are applied at various points during software development and serve to uncover errors and defects that can then be removed. Software reviews "purify" the software engineering activities that we have called *analysis, design,* and *coding.*

Many different types of reviews can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review, if technical problems are discussed. A formal presentation of software design to an audience of customers, management, and technical staff is also a form of review A formal technical review is the most effective filter from a quality assurance standpoint. Conducted by software engineers (and others) for software engineers, the FTR is an effective means for improving software quality.

Cost Impact of Software Defects:

The primary objective of formal technical reviews is to find errors during the process so that they do not become defects after release of the software.

A number of industry studies indicate that design activities introduce between 50 and 65 percent of all errors during the software process. However, formal review techniques have been shown to be up to 75 percent effective] in uncovering design errors. By detecting and removing a large percentage of these errors, the review process

substantially reduces the cost of subsequent steps in the development and support phases.

To illustrate the cost impact of early error detection, we consider a series of relative costs that are based on actual cost data collected for large software projects Assume that an error uncovered ✓ during design will cost 1.0 monetary unit to correct.

just before testing commences will cost 6.5 units;
during testing, 15 units;
and after release, between 60 and 100 units.

3.2) Defect Amplification and Removal:

(*This topic I will tell you later)*

4) FORMAL TECHNICAL REVIEWS

A formal technical review is a software quality assurance activity performed by software engineers
(and others). The objectives of the FTR are

(1) to uncover errors in function, logic, or implementation for any representation of the software;
(2) to verifythat the software under review meets its requirements;
(3) to ensure that the software has been represented according to predefined standards; (4) to achieve software that is developed in a uniform manner; and
(5) to make projects more manageable.

The Review Meeting

Every review meeting should abide by the following constraints:

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.

The focus of the FTR is on a work product.
The individual who has developed the work product—the *producer*—informs the project leader that the work product is complete and that a review is required.

The project leader contacts a *review leader,* who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation.
Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.
The review meeting is attended by the review leader, all reviewers, and the producer. One of the reviewers takes on the role of the *recorder;* that is, the individual who records (in writing) all important issues raised during the review.

At the end of the review, all attendees of the FTR must decide whether to
(1) accept the product without further modification,

(2) reject the product due to severe errors (once corrected, another review must be performed), or
(3) accept the product provisionally.
The decision made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

Review Reporting and Record Keeping

At the end of the review meeting and a review issues list is produced. In addition, a formal technical review summary report is completed. A ***review summary report*** answers three questions:

1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?

The review summary report is a single page form.
It is important to establish a follow-up procedure to ensure that items on the issues list have been properly corrected.

Review Guidelines

The following represents a minimum set of guidelines for formal technical reviews:

1. ***Review the product, not the producer.*** An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of accomplishment.
2. ***Set an agenda and maintain it.*** An FTR must be kept on track and on schedule. The review leader is chartered with the responsibility for maintaining the meeting schedule and should not be afraid to nudge people when drift sets in.
3. ***Limit debate and rebuttal.*** When an issue is raised by a reviewer, there may not be universal agreement on its impact.
4. ***Enunciate problem areas, but don't attempt to solve every problem noted.*** A review is not a problem-solving session. The solution of a problem can often be accomplished by the producer alone or with the help of only one other individual. Problem solving should be postponed until after the review meeting.
5. ***Take written notes.*** It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded.
6. ***Limit the number of participants and insist upon advance preparation.*** Keep the number of people involved to the necessary minimum.
7. ***Develop a checklist for each product that is likely to be reviewed.*** A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues. Checklists should be developed for analysis, design, code, and even test documents.
8. ***Allocate resources and schedule time for FTRs.*** For reviews to be effective, they should be scheduled as a task during the software engineering process

9. ***Conduct meaningful training for all reviewers.*** To be effective all review participants should receive some formal training.
10. ***Review your early reviews***. Debriefing can be beneficial in uncovering problems with the review process itself.

Sample-Driven Reviews (SDRs):

SDRs attempt to quantify those work products that are primary targets for full FTRs.To accomplish this the following steps are suggested…

- Inspect a fraction a_i of each software work product, *i.* Record the number of faults, f_i found within a_i.
- Develop a gross estimate of the number of faults within work product *i* by multiplying f_i by $1/a_i$.
- Sort the work products in descending order according to the gross estimate of the number of faults in each.
- Focus available review resources on those work products that have the highest estimated number of faults.

The fraction of the work product that is sampled must

- Be representative of the work product as a whole and
- Large enough to be meaningful to the reviewer(s) who does the sampling.

5) STATISTICAL SOFTWARE QUALITY ASSURANCE

For software, statistical quality assurance implies the following steps:

1. Information about software defects is collected and categorized.
2. An attempt is made to trace each defect to its underlying cause (e.g., non-conformance to specifications, design error, violation of standards, poor communication with the customer).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the "vital few").
4. Once the vital few causes have been identified, move to correct the problems that have caused the For software, statistical quality assurance implies the following steps:

.

The application of the statistical SQA and the pareto principle can be summarized in a single sentence: *spend your time focusing on things that really matter, but first be sure that you understand what really matters*.

Six Sigma for software Engineering:

Six Sigma is the most widely used strategy for statistical quality assurance in industry today.

The term “six sigma” is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard. The Six Sigma methodology defines three core steps:

- ***Define*** customer requirements and deliverables and project goals via well-defined methods of customer communication
- ***Measure*** the existing process and its output to determine current quality performance (collect defect metrics)
- ***Analyze*** defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps.

- ***Improve*** the process by eliminating the root causes of defects.
- ***Control*** the process to ensure that future work does not reintroduce the causes of defects These core and additional steps are sometimes referred to as the DMAIC (define, measure, analyze, improve, and control) method.

If any organization is developing a software process (rather than improving and existing process), the core steps are augmented as follows:

- ***Design*** the process to o avoid the root causes of defects and o to meet customer requirements
- ***Verify*** that the process model will, in fact, avoid defects and meet customer requirements.

This variation is sometimes called the DMADV (define, measure, analyze, design and verify) method.

6) THE ISO 9000 QUALITY STANDARDS

A *quality assurance system* may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management

ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered.

ISO 9001:2000 is the quality assurance standard that applies to software engineering. The standard contains 20 requirements that must be present for an effective quality assurance system. Because the ISO 9001:2000 standard is applicable to all engineering disciplines, a special set of ISO guidelines have been developed to help interpret the standard for use in the software process.

The requirements delineated by ISO 9001 address topics such as

- management responsibility,
- quality system, contract review,
- design control,
- document and data control,
- product identification and traceability,
- process control,
- inspection and testing,
- corrective and preventive action,
- control of quality records,
- internal quality audits,
- training,

- servicing and
- statistical techniques.

In order for a software organization to become registered to ISO 9001, it must establish policies and procedures to address each of the requirements just noted (and others) and then be able to demonstrate that these policies and procedures are being followed.

SOFTWARE RELIABILITY

Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time".

7.1 Measures of Reliability and Availability:

Most hardware-related reliability models are predicated on failure due to wear rather than failure due to design defects. In hardware, failures due to physical wear (e.g., the effects of temperature, corrosion, shock) are more likely than a design-related failure. Unfortunately, the opposite is true for software. In fact, all software failures can be traced to design or implementation problems; wear does not enter into the picture. A simple measure of reliability is *meantime-between-failure* (MTBF), where

MTBF = MTTF + MTTR

The acronyms MTTF and MTTR are mean-time-to-failure and mean-time-to-repair, respectively.

In addition to a reliability measure, we must develop a measure of availability. *Software availability* is the

probability that a program is operating according to requirements at a given point in time and is defined as **Availability = [MTTF/(MTTF + MTTR)] 100%**

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR, an indirect measure of the maintainability of software.

Software Safety

Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

For example, some of the hazards associated with a computer-based cruise control for an automobile might be

- causes uncontrolled acceleration that cannot be stopped
- does not respond to depression of brake pedal (by turning off)
- does not engage when switch is activated
- slowly loses or gains speed

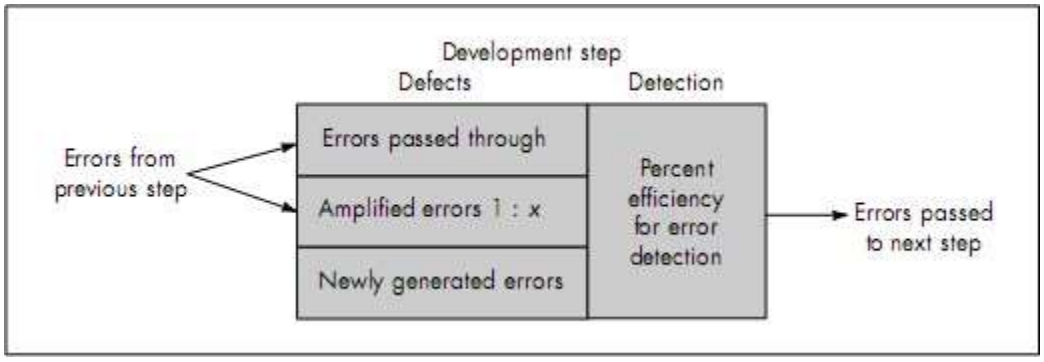
Once these system-level hazards are identified, analysis techniques are used to assign severity and probability of occurrence.To be effective, software must be analyzed in the context of the entire system. If a set of external environmental conditions are met

(and only if they are met), the improper position of the mechanical device will cause a disastrous failure. Analysis techniques such as *fault tree analysis* [VES81], *real-time logic* [JAN86], or *petri net models* [LEV87] can be used to predict the chain of events that can cause hazards and the probability that each of the events will occur to create the chain.

Once hazards are identified and analyzed, safety-related requirements can be specified for the software. That is, the specification can contain a list of undesirable events and the desired system responses to these events. The role of software in managing undesirable events is then indicated.

Although software reliability and software safety are closely related to one another, it is important to understand the subtle difference between them. Software reliability uses statistical analysis to determine the likelihood that a software failure will occur. However, the occurrence of a failure does not necessarily result in a hazard or mishap. Software safety examines the ways in which failures result in conditions that can lead to a mishap.

Defect Amplification and Removal:



Defect Amplification Model

A defect amplification model can be used to illustrate the generation and detection of errors during the preliminary design, detail design, and coding steps of the software engineering process.

A box represents a software development step. During the step, errors may be inadvertently generated. Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through. In some cases, errors passed through from previous steps are amplified (amplification factor, *x*) by current work. The box subdivisions represent each of these characteristics and the percent of efficiency for detecting errors, a function of the thoroughness of the review.

Referring to the figure8.3 each test step is assumed to uncover and correct 50 percent of all incoming errors without introducing any new errors (an optimistic

assumption). Ten preliminary design defects are amplified to 94 errors before testing commences. Twelve latent errors are released to the field.

Figure8.4 considers the same conditions except that design and code reviews are conducted as part of each development step. In this case, ten initial preliminary design errors are amplified to 24 errors before testing commences. Only three latent errors exist.

Recalling the relative costs associated with the discovery and correction of errors, overall cost (with and without review for our hypothetical example) can be established. The number of errors uncovered during each of the steps noted in Figures 8.3 and 8.4 is multiplied by the cost to remove an error (1.5 cost units for design, 6.5 cost units before test, 15 cost units during test, and 67 cost units after release).

Using these data, the total cost for development and maintenance when reviews are conducted is 783 cost units.

When no reviews are conducted, total cost is 2177 units—nearly three times more costly.

To conduct reviews, a software engineer must expend time and effort and the development organization must spend money. Formal technical reviews (for design and other technical activities) provide a demonstrable cost benefit. They should be conducted.

FIGURE 8.3

Defect amplification, no reviews

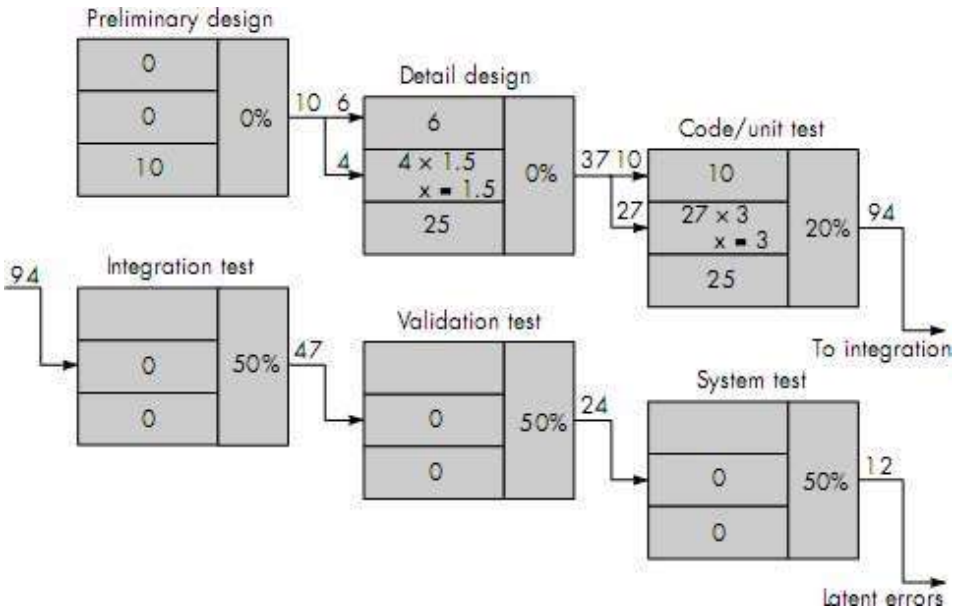
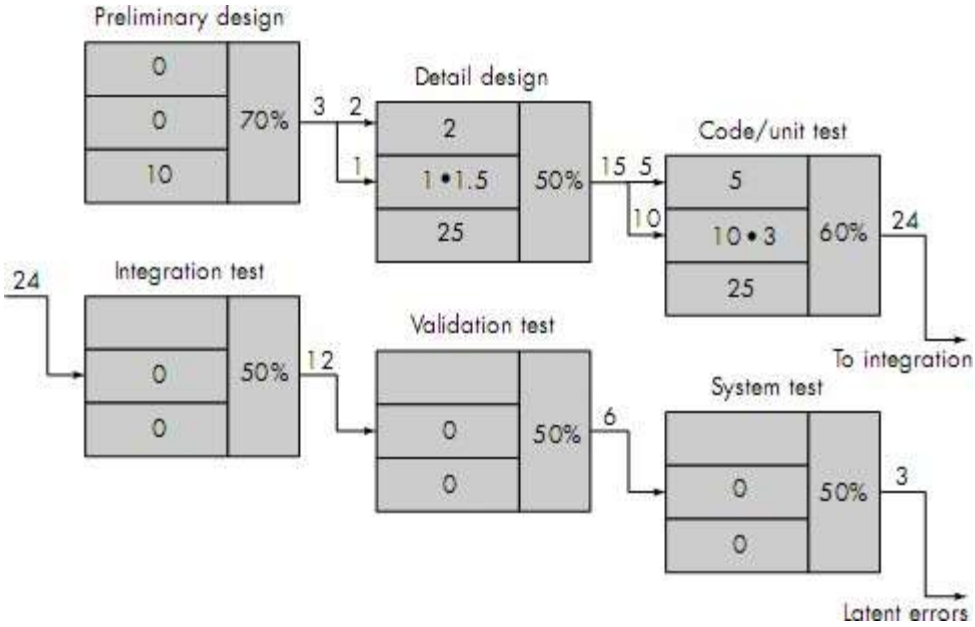


FIGURE 8.4

Defect amplification, reviews conducted


Preliminary design
0
0
10
70%
3
2
1
Detail design
2
1 • 1.5
25
50%
15
5
10
Code/unit test
5
10 • 3
25
60%
24
To integration
24
Integration test
0
0
50%
12
Validation test
0
0
50%
6
System test
0
0
50%
3
Latent errors