Fundamentals of Software Engineering

What is Software Engineering?

- **Engineering approach to develop software.**
  - **Building Construction Analogy.**

- **Systematic collection of past experience:**
  - **Techniques,**
  - **Methodologies,**
  - **Guidelines.**

Engineering Practice
Neither Science nor Art

- **Heavy use of past experience:**
  - **Past experience is systematically arranged.**
- **Theoretical basis and quantitative techniques provided.**
- **Many are just thumb rules.**
- **Tradeoff between alternatives.**
- **Pragmatic approach to cost-effectiveness.**

Technology Development Pattern

Evolution of an Art into an Engineering Discipline

- **The early programmers used an exploratory (also called build and fix) style.**
  - **In the build and fix (exploratory) style, normally a `dirty' program is quickly developed.**
  - **The different imperfections that are subsequently noticed are fixed.**

What is Wrong with the Exploratory Style?

- **Can successfully be used for very small programs only.**

What is Wrong with the Exploratory Style? Cont…

- **Besides the exponential growth of effort, cost, and time with problem size:**
  - **Exploratory style usually results in unmaintainable code.**
  - **It becomes very difficult to use the exploratory style in a team development environment.**

Implication in Program Development

- **A small program having just a few variables:**
  - **Is within the easy grasp of an individual.**
- **As the number of independent variables in the program increases:**
  - **It quickly exceeds the grasping power of an individual:**
    - **Requires an unduly large effort to master the problem.**

Implication in Program Development

- **Instead of a human, if a machine could be writing (generating) a program,**
  - **The slope of the curve would be linear.**
- **But, why does the effort-size curve become almost linear when software engineering principles are deployed?**
  - **Software engineering principles extensively use techniques specifically to overcome the human cognitive limitations.**

Principles Deployed by Software Engineering to Overcome Human Cognitive Limitations

- **Mainly two important principles are deployed:**
  - **Abstraction**
  - **Decomposition**

Abstraction

- **Simplify a problem by omitting unnecessary details.**
  - **Focus attention on only one aspect of the problem and ignore irrelevant details.**
- **Suppose you are asked to develop an overall understanding of some country.**
  - **No one in his right mind would meet all the citizens of the country, visit every house, and examine every tree of the country, etc.**
  - **You would possibly refer to various types of maps for that country.**
- **A map, in fact, is an abstract representation of a country.**

Decomposition

- **Decompose a problem into many small independent parts.**
  - **The small parts are then taken up one by one and solved separately.**
  - **The idea is that each small part would be easy to grasp and can be easily solved.**
  - **The full problem is solved when all the parts are solved.**

Decomposition

- **A popular way to demonstrate the decomposition principle:**
  - **Try to break a bunch of sticks tied together versus breaking them individually.**
- **Example use of decomposition principle:**
  - **You understand a book better when the contents are organized into independent chapters**
  - **Compared to when everything is mixed up.**

Why Study Software Engineering? (1)

- To acquire skills to develop large programs.
  - Exponential growth in complexity and difficulty level with size.
  - The ad hoc approach breaks down when size of software increases.

Why Study Software Engineering? (2)

- **Ability to solve complex programming problems:**
  - **How to break large projects into smaller and manageable parts?**
  - **How to use abstraction?**
- **Also learn techniques of:**
  - **Specification, design, user interface development, testing, project management, etc.**

Why Study Software Engineering? (3)

- **To acquire skills to be a better programmer:**
  - **Higher Productivity**
  - **Better Quality Programs**

Software Crisis

- **Software products:**
  - **Fail to meet user requirements.**
  - **Frequently crash.**
  - **Expensive.**
  - **Difficult to alter, debug, and enhance.**
  - **Often delivered late.**
  - **Use resources non-optimally.**

Software Crisis (cont.)

Relative Cost of Hardware and Software

Factors Contributing to the Software Crisis

- **Larger problems,**
- **Lack of adequate training in software engineering,**
- **Increasing skill shortage,**
- **Low productivity improvements.**

Programs versus Software Products

• Usually small in size	• Large
• Author himself is sole user	• Large number of users
• Single developer	• Team of developers
• Lacks proper user interface	• Well-designed interface
• Lacks proper documentation	• Well documented & user-manual prepared
• Ad hoc development.	• Systematic development

Types of Software Projects

- **Software products**
- **Outsourced projects**
- **Indian companies have focused on outsourced projects.**

Evolution of Design Techniques

Emergence of Software Engineering

- **Early Computer Programming (1950s):**
  - **Programs were being written in assembly language.**
  - **Programs were limited to about a few hundreds of lines of assembly code.**

High-Level Language Programming (Early 60s)

- **High-level languages such as FORTRAN, ALGOL, and COBOL were introduced:**
  - This reduced software development efforts greatly.

High-Level Language Programming (Early 60s)

- Software development style was still exploratory.
  - Typical program sizes were limited to a few thousands of lines of source code.

Control Flow-Based Design (late 60s)

- **Size and complexity of programs increased further:**
  - **Exploratory programming style proved to be insufficient.**
- **Programmers found:**
  - **Very difficult to write cost-effective and correct programs.**

Control Flow-Based Design (late 60s)

- **Programmers found:**
  - **programs written by others very difficult to understand and maintain.**
- **To cope up with this problem, experienced programmers advised: ``Pay particular attention to the design of the program's control structure.''**

Control Flow-Based Design (late 60s)

- A program's control structure indicates:
  - The sequence in which the program's instructions are executed.
- To help design programs having good control structure:
  - Flow charting technique was developed.

Control Flow-Based Design (late 60s)

- Using flow charting technique:
  - One can represent and design a program's control structure.
  - Usually one understands a program:
    - By mentally simulating the program's execution sequence.

Control Flow-Based Design

(Late 60s)

- **A program having a messy flow chart representation:**
  - **Difficult to understand and debug.**

Control Flow-Based Design (Late 60s)

- It was found:
  - GO TO statements makes control structure of a program messy.
  - GO TO statements alter the flow of control arbitrarily.
  - The need to restrict use of GO TO statements was recognized.

Control Flow-Based Design (Late 60s)

- **Many programmers had extensively used assembly languages.**
  - **JUMP instructions are frequently used for program branching in assembly languages.**
  - **Programmers considered use of GO TO statements unavoidable.**

Control-flow Based Design (Late 60s)

- At that time, Dijkstra published his article:
  - "Goto Statement Considered Harmful" Comm. of ACM, 1969.
- Many programmers were unhappy to read his article.

Control Flow-Based Design (Late 60s)

- **They published several counter articles:**
  - **Highlighting the advantages and inevitability of GO TO statements.**

Control Flow-Based Design (Late 60s)

- **But, soon it was conclusively proved:**
  - **Only three programming constructs are sufficient to express any programming logic:**
    - **sequence (e.g. a=0;b=5;)**
    - **selection (e.g.if(c=true) k=5 else m=5;)**
    - **iteration (e.g. while(k>0) k=j-k;)**

Control-flow Based Design (Late 60s)

- **Everyone accepted:**
  - **It is possible to solve any programming problem without using GO TO statements.**
  - **This formed the basis of Structured Programming methodology.**

Structured Programming

- A program is called structured
  - When it uses only the following types of constructs:
    - sequence,
    - selection,
    - iteration

Structured Programs

- **Unstructured control flows are avoided.**
- **Consist of a neat set of modules.**
- **Use single-entry, single-exit program constructs.**

Structured Programs

- However, violations to this feature are permitted:
  - Due to practical considerations such as:
    - Premature loop exit to support exception handling.

Structured programs

- **Structured programs are:**
  - **Easier to read and understand,**
  - **Easier to maintain,**
  - **Require less effort and time for development.**

Structured Programming

- **Research experience shows:**
  - **Programmers commit less number of errors:**
    - **While using structured if-then-else and do-while statements.**

Data Structure-Oriented Design

(Early 70s)

- **Soon it was discovered:**
  - **It is important to pay more attention to the design of data structures of a program**
    - **Than to the design of its control structure.**

Data Structure-Oriented Design (Early 70s)

- Techniques which emphasize designing the data structure:
  - Derive program structure from it:
    - Are called `data structure-oriented design techniques.`

Data Structure Oriented Design

(Early 70s)

- **Example of data structure-oriented design technique:**
  - **Jackson's Structured Programming(JSP) methodology**
    - **Developed by Michael Jackson in 1970s.**

Data Structure Oriented Design

(Early 70s)

- **JSP technique:**
  - Program code structure should correspond to the data structure.

Data Structure Oriented Design

(Early 70s)

- In JSP methodology:
  - A program's data structures are first designed using notations for
    - sequence, selection, and iteration.
  - Then data structure design is used :
    - To derive the program structure.

Data Structure Oriented Design

(Early 70s)

- **Several other data structure-oriented Methodologies also exist:**
  - e.g., Warnier-Orr Methodology.

Data Flow-Oriented Design (Late 70s)

- **Data flow-oriented techniques advocate:**
  - **The data items input to a system must first be identified,**
  - **Processing required on the data items to produce the required outputs should be determined.**

Data Flow-Oriented Design (Late 70s)

- **Data flow technique identifies:**
  - **Different processing stations (functions) in a system.**
  - **The items (data) that flow between processing stations.**

Data Flow-Oriented Design (Late 70s)

- **Data flow technique is a generic technique:**
  - **Can be used to model the working of any system.**
    - **not just software systems.**
- **A major advantage of the data flow technique is its simplicity.**

Data Flow Model of a Car Assembly Unit

Object-Oriented Design (80s)

- **Object-oriented technique:**
  - **An intuitively appealing design approach:**
  - **Natural objects (such as employees, pay-roll-register, etc.) occurring in a problem are first identified.**

Object-Oriented Design (80s)

- **Relationships among objects:**
  - **Such as composition, reference, and inheritance are determined.**
- **Each object essentially acts as**
  - **A data hiding (or data abstraction) entity.**

Object-Oriented Design (80s)

- Object-Oriented Techniques have gained wide acceptance:
  - Simplicity
  - Reuse possibilities
  - Lower development time and cost
  - More robust code
  - Easy maintenance

Evolution of Other Software Engineering Techniques

- **The improvements to the software design methodologies**
  - **are indeed very noticeable.**
- **In additions to the software design techniques:**
  - **Several other techniques evolved.**

Evolution of Other Software Engineering Techniques

- life cycle models,
- specification techniques,
- project management techniques,
- testing techniques,
- debugging techniques,
- quality assurance techniques,
- software measurement techniques,
- CASE tools, etc.

Differences between the exploratory style and modern software development practices

- Use of Life Cycle Models
- Software is developed through several well-defined stages:
  - requirements analysis and specification,
  - design,
  - coding,
  - testing, etc.

Differences between the exploratory style and modern software development practices

- **Emphasis has shifted**
  - **from error correction to error prevention.**
- **Modern practices emphasize:**
  - **detection of errors as close to their point of introduction as possible.**

Differences between the exploratory style and modern software development practices (CONT.)

- In exploratory style,
  - errors are detected only during testing,
- Now,
  - focus is on detecting as many errors as possible in each phase of development.

Differences between the exploratory style and modern software development practices (CONT.)

- In exploratory style,
  - coding is synonymous with program development.
- Now,
  - coding is considered only a small part of program development effort.

Differences between the exploratory style and modern software development practices (CONT.)

- A lot of effort and attention is now being paid to:
  - Requirements specification.
- Also, now there is a distinct design phase:
  - Standard design techniques are being used.

Differences between the exploratory style and modern software development practices (CONT.)

- **During all stages of development process:**
  - **Periodic reviews are being carried out**
- **Software testing has become systematic:**
  - **Standard testing techniques are available.**

Differences between the exploratory style and modern software development practices (CONT.)

- **There is better visibility of design and code:**
  - **Visibility means production of good quality, consistent and standard documents.**
  - **In the past, very little attention was being given to producing good quality and consistent documents.**
  - **increased visibility makes software project management easier.**

Differences between the exploratory style and modern software development practices (CONT.)

- **Because of good documentation:**
  - **fault diagnosis and maintenance are smoother now.**
- **Several metrics are being used:**
  - **help in software project management, quality assurance, etc.**

Differences between the exploratory style and modern software development practices (CONT.)

- **Projects are being thoroughly planned:**
  - **estimation,**
  - **scheduling,**
  - **monitoring mechanisms.**
- **Use of CASE tools.**

Software Life Cycle

- **Software life cycle (or software process):**
  - Series of identifiable stages that a software product undergoes during its life time:
    - **Feasibility study**
    - **Requirements analysis and specification,**
    - **Design,**
    - **Coding,**
    - **Testing**
    - **maintenance.**

Life Cycle Model

- **A software life cycle model (or process model):**
  - **a descriptive and diagrammatic model of software life cycle:**
  - **identifies all the activities required for product development,**
  - **establishes a precedence ordering among the different activities,**
  - **Divides life cycle into phases.**

Life Cycle Model (CONT.)

- **Several different activities may be carried out in each life cycle phase.**
  - **For example, the design stage might consist of:**
    - **structured analysis activity followed by**
    - **structured design activity.**

Why Model Life Cycle ?

- **A written description:**
  - **Forms a common understanding of activities among the software developers.**
  - **Helps in identifying inconsistencies, redundancies, and omissions in the development process.**
  - **Helps in customization of a process model for specific projects.**

Why Model Life Cycle ?

- **Processes are customized for special projects.**
  - **A documented process model**
    - **Helps to identify where the customization is needed.**

Life Cycle Model (CONT.)

- **The development team must identify a suitable life cycle model:**
  - **and then adhere to it.**
  - **Primary advantage of adhering to a life cycle model:**
    - **Helps development of software in a systematic and disciplined manner.**

Life Cycle Model (CONT.)

- **When a program is developed by a single programmer ---**
  - **he has the freedom to decide his exact steps.**

Life Cycle Model (CONT.)

- **When a software product is being developed by a team:**
  - **there must be a precise understanding among team members as to when to do what,**
  - **otherwise it would lead to chaos and project failure.**

Life Cycle Model (CONT.)

- A software project will never succeed if:
  - one engineer starts writing code,
  - another concentrates on writing the test document first,
  - yet another engineer first defines the file structure
  - another defines the I/O for his portion first.

Life Cycle Model (CONT.)

- A life cycle model:
  - defines entry and exit criteria for every phase.
  - A phase is considered to be complete:
    - only when all its exit criteria are satisfied.

Life Cycle Model (CONT.)

- **The phase exit criteria for the software requirements specification phase:**
  - **Software Requirements Specification (SRS) document is complete, reviewed, and approved by the customer.**
- **A phase can start:**
  - **only if its phase-entry criteria have been satisfied.**

Life Cycle Model (CONT.)

- **It becomes easier for software project managers:**
  - **to monitor the progress of the project.**

Life Cycle Model (CONT.)

- When a life cycle model is adhered to,
  - the project manager can at any time fairly accurately tell,
    - at which stage (e.g., design, code, test, etc.) of the project is.
  - Otherwise, it becomes very difficult to track the progress of the project
    - the project manager would have to depend on the guesses of the team members.

Life Cycle Model (CONT.)

- **This usually leads to a problem:**
  - **known as the 99% complete syndrome.**

Life Cycle Model (CONT.)

- Many life cycle models have been proposed.
- We will confine our attention to a few important and commonly used models.
  - Classical waterfall model
  - Iterative waterfall,
  - Evolutionary,
  - Prototyping, and
  - Spiral model

Software Life Cycle

- **Software life cycle (or software process):**
  - Series of identifiable stages that a software product undergoes during its life time:
    - **Feasibility study**
    - **Requirements analysis and specification,**
    - **Design,**
    - **Coding,**
    - **Testing**
    - **maintenance.**

Life Cycle Model

- **A software life cycle model (or process model):**
  - **a descriptive and diagrammatic model of software life cycle:**
  - **identifies all the activities required for product development,**
  - **establishes a precedence ordering among the different activities,**
  - **Divides life cycle into phases.**

Life Cycle Model (CONT.)

- **Several different activities may be carried out in each life cycle phase.**
  - **For example, the design stage might consist of:**
    - **structured analysis activity followed by**
    - **structured design activity.**

Why Model Life Cycle ?

- **A written description:**
  - **Forms a common understanding of activities among the software developers.**
  - **Helps in identifying inconsistencies, redundancies, and omissions in the development process.**
  - **Helps in customization of a process model for specific projects.**

Why Model Life Cycle ?

- **Processes are customized for special projects.**
  - **A documented process model**
    - **Helps to identify where the customization is needed.**

Life Cycle Model (CONT.)

- **The development team must identify a suitable life cycle model:**
  - **and then adhere to it.**
  - **Primary advantage of adhering to a life cycle model:**
    - **Helps development of software in a systematic and disciplined manner.**

Life Cycle Model (CONT.)

- **When a program is developed by a single programmer ---**
  - **he has the freedom to decide his exact steps.**

Life Cycle Model (CONT.)

- **When a software product is being developed by a team:**
  - **there must be a precise understanding among team members as to when to do what,**
  - **otherwise it would lead to chaos and project failure.**

Life Cycle Model (CONT.)

- A software project will never succeed if:
  - one engineer starts writing code,
  - another concentrates on writing the test document first,
  - yet another engineer first defines the file structure
  - another defines the I/O for his portion first.

Life Cycle Model (CONT.)

- A life cycle model:
  - defines entry and exit criteria for every phase.
  - A phase is considered to be complete:
    - only when all its exit criteria are satisfied.

Life Cycle Model (CONT.)

- **The phase exit criteria for the software requirements specification phase:**
  - **Software Requirements Specification (SRS) document is complete, reviewed, and approved by the customer.**
- **A phase can start:**
  - **only if its phase-entry criteria have been satisfied.**

Life Cycle Model (CONT.)

- **It becomes easier for software project managers:**
  - **to monitor the progress of the project.**

Life Cycle Model (CONT.)

- When a life cycle model is adhered to,
  - the project manager can at any time fairly accurately tell,
    - at which stage (e.g., design, code, test, etc.) of the project is.
  - Otherwise, it becomes very difficult to track the progress of the project
    - the project manager would have to depend on the guesses of the team members.

Life Cycle Model (CONT.)

- **This usually leads to a problem:**
  - **known as the 99% complete syndrome.**

Life Cycle Model (CONT.)

- Many life cycle models have been proposed.
- We will confine our attention to a few important and commonly used models.
  - Classical waterfall model
  - Iterative waterfall,
  - Evolutionary,
  - Prototyping, and
  - Spiral model

Life Cycle Models

- Classical Waterfall Model
- Iterative Waterfall Model
- Prototyping Model
- Evolutionary Model
- Spiral Model

Classical Waterfall Model

- Classical waterfall model divides life cycle into phases:
  - feasibility study,
  - requirements analysis and specification,
  - design,
  - coding and unit testing,
  - integration and system testing,
  - maintenance.

Classical Waterfall Model

Relative Effort for Phases

- Phases between feasibility study and testing
  - known as development phases.
- Among all life cycle phases
  - maintenance phase consumes maximum effort.
- Among development phases,
  - testing phase consumes the maximum effort.

Classical Waterfall Model (CONT.)

- Most organizations usually define:
  - **standards on the outputs (deliverables) produced at the end of every phase**
  - **entry and exit criteria for every phase.**
- They also prescribe specific methodologies for:
  - **specification,**
  - **design,**
  - **testing,**
  - **project management, etc.**

Classical Waterfall Model (CONT.)

- The guidelines and methodologies of an organization:
  - **called the organization's software development methodology.**
- Software development organizations:
  - **expect fresh engineers to master the organization's software development methodology.**

Feasibility Study

- Main aim of feasibility study:determine whether developing the product
  - financially worthwhile
  - technically feasible.
- First roughly understand what the customer wants:
  - different data which would be input to the system,
  - processing needed on these data,
  - output data to be produced by the system,
  - various constraints on the behavior of the system.

Activities during Feasibility Study

- Work out an overall understanding of the problem.
- Formulate different solution strategies.
- Examine alternate solution strategies in terms of:
  - resources required,
  - cost of development, and
  - development time.

Activities during Feasibility Study

- Perform a cost/benefit analysis:
  - to determine which solution is the best.
  - you may determine that none of the solutions is feasible due to:
    - high cost,
    - resource constraints,
    - technical reasons.

Requirements Analysis and Specification

- Aim of this phase:
  - understand the exact requirements of the customer,
  - document them properly.
- Consists of two distinct activities:
  - requirements gathering and analysis
  - requirements specification.

Goals of Requirements Analysis

- Collect all related data from the customer:
  - analyze the collected data to clearly understand what the customer wants,
  - find out any inconsistencies and incompleteness in the requirements,
  - resolve all inconsistencies and incompleteness.

Requirements Gathering

- Gathering relevant data:
  - usually collected from the end-users through interviews and discussions.
  - For example, for a business accounting software:
    - interview all the accountants of the organization to find out their requirements.

Requirements Analysis (CONT.)

- The data you initially collect from the users:
  - would usually contain several contradictions and ambiguities:
  - each user typically has only a partial and incomplete view of the system.

Requirements Analysis (CONT.)

- Ambiguities and contradictions:
  - must be identified
  - resolved by discussions with the customers.
- Next, requirements are organized:
  - into a Software Requirements Specification (SRS) document.

Requirements Analysis (CONT.)

- Engineers doing requirements analysis and specification:
  - are designated as analysts.

Design

- Design phase transforms requirements specification:
  - into a form suitable for implementation in some programming language.

Design

- In technical terms:
  - during design phase, <u>software architecture</u> is derived from the SRS document.
- Two design approaches:
  - traditional approach,
  - object oriented approach.

Traditional Design Approach

- **Consists of two activities:**
  - **Structured analysis**
  - **Structured design**

Structured Analysis Activity

- Identify all the functions to be performed.
- Identify data flow among the functions.
- Decompose each function recursively into sub-functions.
  - Identify data flow among the subfunctions as well.

Structured Analysis (CONT.)

- Carried out using Data flow diagrams (DFDs).
- After structured analysis, carry out structured design:
  - architectural design (or high-level design)
  - detailed design (or low-level design).

Structured Design

- High-level design:
  - decompose the system into *modules*,
  - represent invocation relationships among the modules.
- Detailed design:
  - different modules designed in greater detail:
    - data structures and algorithms for each module are designed.

Object Oriented Design

- First identify various objects (real world entities) occurring in the problem:
  - identify the relationships among the objects.
  - For example, the objects in a pay-roll software may be:
    - employees,
    - managers,
    - pay-roll register,
    - Departments, etc.

Object Oriented Design (CONT.)

- Object structure
  - further refined to obtain the detailed design.
- OOD has several advantages:
  - lower development effort,
  - lower development time,
  - better maintainability.

Implementation

- Purpose of implementation phase (aka coding and unit testing phase):
  - translate software design into source code.
  - Coding Standards
    - Standard way of laying out the program codes
    - Template for laying out the function and module headers
    - Commenting guidelines
    - Variable and function naming conventions
    - Maximum number of source lines permitted in each module

Implementation

- During the implementation phase:
  - each module of the design is coded,
  - each module is unit tested
    - tested independently as a stand alone unit, and debugged,
  - each module is documented.

Implementation (CONT.)

- The purpose of unit testing:
  - test if individual modules work correctly.
- The end product of implementation phase:
  - a set of program modules that have been tested individually.

Integration and System Testing

- Different modules are integrated in a planned manner:
  - modules are almost never integrated in one shot.
  - Normally integration is carried out through a number of steps.
- During each integration step,
  - the partially integrated system is tested.

Integration and System Testing

System Testing

- After all the modules have been successfully integrated and tested:
  - system testing is carried out.
- Goal of system testing:
  - ensure that the developed system functions according to its requirements as specified in the SRS document.

- **α- testing: performed by development team**
- **β-testing: performed by friendly set of customers**
- **Acceptance testing: performed by customer after product delivery (whether to accept the product or reject)**
- **System test plan: prepared after SRS, just based upon SRS.**
  - **List testing related activities, schedule of testing, allocate resources, list all test cases and expected outcome**
- **Test Report**

Maintenance

- Maintenance of any software product:
  - **requires much more effort than the effort to develop the product itself.**
  - **development effort to maintenance effort is typically 40:60.**

Maintenance (CONT.)

- Corrective maintenance:
  - Correct errors which were not discovered during the product development phases.
- Perfective maintenance:
  - Improve implementation of the system
  - enhance functionalities of the system.
- Adaptive maintenance:
  - Port software to a new environment,
    - e.g. to a new computer or to a new operating system.

Iterative Waterfall Model

- Classical waterfall model is idealistic:
  - assumes that no defect is introduced during any development activity.
  - in practice:
    - defects do get introduced in almost every phase of the life cycle.

Iterative Waterfall Model (CONT.)

- Defects usually get detected much later in the life cycle:
  - **For example, a design defect might go unnoticed till the coding or testing phase.**

Iterative Waterfall Model (CONT.)

- Once a defect is detected:
  - we need to go back to the phase where it was introduced
  - redo some of the work done during that and all subsequent phases.
- Therefore we need feedback paths in the classical waterfall model.
- It assumes that requirements are defined correctly at the beginning of the project
- It assumes that all the phases are sequential

Iterative Waterfall Model (CONT.)

Phase Containment of Errors

- Errors should be detected
  - **in the same phase in which they are introduced.**
- For example:
  - if a design problem is detected in the design phase itself,
    - **the problem can be taken care of much more easily**
    - **than say if it is identified at the end of the integration and system testing phase.**

Phase containment of errors

- Reason: rework must be carried out not only to the design but also to code and test phases.
- The principle of detecting errors as close to its point of introduction as possible:
  - is known as **phase containment of errors.**
- Iterative waterfall model is by far the most widely used model.
  - Almost every other model is derived from the waterfall model.

- **Iterative waterfall Shortcomings:**
  - **Can’t satisfactorily handle risks, uncertainty in the requirements**
  - **It is difficult to follow the rigid phase sequence prescribed by it, can lead to blocking states.**

Prototyping Model

- Before starting actual development,
  - a working prototype of the system should first be built.
- A prototype is a toy implementation of a system:
  - limited functional capabilities,
  - low reliability,
  - inefficient performance.

Reasons for developing a prototype

- **1. Illustrate to the customer:**
  - **input data formats, messages, reports, or interactive dialogs.**
- **2. Examine technical issues associated with product development:**
  - **Often major design decisions depend on issues like:**
    - **response time of a hardware controller,**
    - **efficiency of a sorting algorithm, etc.**

Prototyping Model (CONT.)

- 3. The third reason for developing a prototype is:
  - it is impossible to ``get it right'' the first time,
  - we must plan to throw away the first product
    - if we want to develop a good product.

Prototyping Model (CONT.)

- STEPS
- Start with approximate requirements.
- Carry out a quick design.
- Prototype model is built using several short-cuts:
  - **Short-cuts might involve using inefficient, inaccurate, or dummy functions.**
    - **A function may use a table look-up rather than performing the actual computations.**
    - **Some variable may use initialized values rather than taking input from user**

Prototyping Model (CONT.)

- The developed prototype is submitted to the customer for his evaluation:
  - **Based on the user feedback, requirements are refined.**
  - **This cycle continues until the user approves the prototype.**
- The actual system is developed using the iterative waterfall approach.

Prototyping Model (CONT.)

Prototyping Model (CONT.)

- Requirements analysis and specification phase becomes redundant:
  - final working prototype (with all user feedbacks incorporated) serves as an animated requirements specification.
- Design and code for the prototype is usually thrown away:
  - However, the experience gathered from developing the prototype helps a great deal while developing the actual product.

Prototyping Model (CONT.)

- Even though construction of a working prototype model involves additional cost --- overall development cost might be lower for:
  - **systems with unclear user requirements,**
  - **systems with unresolved technical issues.**
- Many user requirements get properly defined and technical issues get resolved:
  - **these would have appeared later as change requests and resulted in incurring massive redesign costs**.

Evolutionary Model

- Evolutionary model (aka successive versions or incremental model):
  - The system is broken down into several modules which can be incrementally implemented and delivered.
- First develop the core modules of the system. (Core module: do't need services from other modules, non-core need services from other)
- The initial product skeleton is refined into increasing levels of capability:
  - by adding new functionalities in successive versions.

Evolutionary Model (CONT.)

- Successive version of the product:
  - functioning systems capable of performing some useful work.
  - A new release may include new functionality:
    - also existing functionality in the current release might have been enhanced.

Evolutionary Model (CONT.)

Advantages of Evolutionary Model

- Users get a chance to experiment with a partially developed system:
  - much before the full working version is released,
- Helps finding exact user requirements:
  - much before fully working system is developed.
- Core modules get tested thoroughly:
  - reduces chances of errors in final product.

Disadvantages of Evolutionary Model

- Often, difficult to subdivide problems into functional units:
  - which can be incrementally implemented and delivered.
  - evolutionary model is useful for very large problems,
    - where it is easier to find modules for incremental implementation.

Evolutionary Model with Iteration

- Many organizations use a combination of iterative and incremental development:
  - a new release may include new functionality
  - existing functionality from the current release may also have been modified.

Evolutionary Model with iteration

- Several advantages:
  - Training can start on an earlier release
    - customer feedback taken into account
  - Markets can be created:
    - for functionality that has never been offered.
  - Frequent releases allow developers to fix unanticipated problems quickly.

Spiral Model

- Proposed by Boehm in 1988.
- Each loop of the spiral represents a phase of the software process:
  - the innermost loop might be concerned with system feasibility,
  - the next loop with system requirements definition,
  - the next one with system design, and so on.
- There are no fixed phases in this model, the phases shown in the figure are just examples.

Spiral Model (CONT.)

- The team must decide:
  - how to structure the project into phases.
- Start work using some generic model:
  - add extra phases
    - for specific projects or when problems are identified during a project.
- Each loop in the spiral is split into four sectors (quadrants).

Spiral Model (CONT.)

Objective Setting (First Quadrant)

- Identify objectives of the phase,
- Examine the risks associated with these objectives.
  - Risk:
    - any adverse circumstance that might hamper successful completion of a software project.
- Find alternate solutions possible.

Risk Assessment and Reduction (Second Quadrant)

- For each identified project risk,
  - a detailed analysis is carried out.
- Steps are taken to reduce the risk.
- For example, if there is a risk that the requirements are inappropriate:
  - a prototype system may be developed.

Spiral Model (CONT.)

- Development and Validation (Third quadrant):
  - develop and validate the next level of the product.
- Review and Planning (Fourth quadrant):
  - review the results achieved so far with the customer and plan the next iteration around the spiral.
- With each iteration around the spiral:
  - progressively more complete version of the software gets built.
- Radius of spiral at any point represent the cost incurred in the project so far.
- Angular dimension represent the progress made so far in the current phase

Spiral Model as a meta model

- Subsumes all discussed models:
  - **a single loop spiral represents waterfall model.**
  - **uses an evolutionary approach --**
    - **iterations through the spiral are evolutionary levels.**
  - **enables understanding and reacting to risks during each iteration along the spiral.**
  - **uses:**
    - prototyping as a risk reduction mechanism
    - **retains the step-wise approach of the waterfall model.**

Comparison of Different Life Cycle Models

- Iterative waterfall model
  - **most widely used model.**
  - **But, suitable only for well-understood problems**.
- Prototype model is suitable for projects not well understood:
  - user requirements
  - technical aspects

Comparison of Different Life Cycle Models (CONT.)

- Evolutionary model is suitable for large problems:
  - **can be decomposed into a set of modules that can be incrementally implemented,**
  - **incremental delivery of the system is acceptable to the customer.**
- The spiral model:
  - **suitable for development of technically challenging software products that are subject to several kinds of risks.**

SEI Capability Maturity Model (CMM)

- Developed by Software Engineering Institute (SEI) of the Carnegie Mellon University, USA:
  - To assist the U.S. Department of Defense (DoD) in software acquisition.
  - The rationale was to include:
    - Likely contractor performance as a factor in contract awards.

SEI Capability Maturity Model

- Major DoD contractors began CMM-based process improvement initiatives:
  - As they vied for DoD contracts.
- SEI CMM helped organizations:
  - Helped Improve quality of software they developed
  - Realized adoption of SEI CMM model had significant business benefits.
- Other organizations adopted CMM.

SEI Capability Maturity Model

- In simple words:
  - CMM is a model for apprising the software process maturity of a contractor into different levels.
  - Can be used to predict the most likely outcome to be expected:
    - from the next project that the organization undertakes.

SEI Capability Maturity Model

- Can be used in two ways:
  - Capability evaluation
  - Software process assessment.

Capability Evaluation

- Provides a way to **assess** the software process capability of an organization:
  - Helps in selecting a contractor
  - Indicates the likely contractor performance.

Software Process Assessment

- Used by an organization to assess its current process:
  - Suggests ways to improve the process capability.
  - This type of assessment is for purely internal use.

SEI Capability Maturity Model

- The SEI CMM classifies software development industries into:
  - Five maturity levels.
  - Stages are ordered so that improvements at one stage provide foundations for the next.
  - Based on the pioneering work of Philip Crosby.

SEI Capability Maturity Model

Level 1: (Initial)

- Organization operates
  - Without any formalized process or project plans
- An organization at this level is characterized by
  - Ad hoc and often chaotic activities.

Level 1: (Initial)

- Software production processes are not defined,
  - Different engineers follow their own process
  - Development efforts become chaotic.
  - The success of projects depend on individual efforts and heroics.

Level 2: (Repeatable)

- Basic project management practices
  - Tracking cost, schedule, and functionality are followed.
- Size and cost estimation techniques:
  - Function point analysis, COCOMO, etc. used.
- Production process is ad hoc:
  - Not formally defined
  - Also not documented.

Level 2: (Repeatable)

- Process used for different projects might vary between projects:
  - Earlier success on projects with similar applications can be repeated.
  - Opportunity to repeat process exist when a company produces a family of products.

Level 3: (Defined)

- Management and development activities:
  - Defined and documented.
  - Common organization-wide understanding of activities, roles, and responsibilities.

Level 3: (Defined)

- The process though defined:
  - Process and product qualities are not measured.
- ISO 9001 aims at achieving this level.

Level 4: (Managed)

- Quantitative quality goals for products are set.
- Software process and product quality are measured:
  - The measured values are used to control the product quality.
  - Results of measurement used to evaluate project performance:
    - Rather than improve process.

Level 4: (Managed)

- Organization sets quantitative quality goals.
- World-wide about 100 organizations assessed at this level.

Level 5: (Optimizing)

- Statistics collected from process and product measurements are analyzed:
  - Continuous process improvement based on the measurements.
    - Known types of defects are prevented from recurring by tuning the process
    - Lessons learned from specific projects incorporated into the process

Level 5: (Optimizing)

- Identify best software engineering practices and innovations:
  - Tools, methods, or process are identified.
  - Transferred throughout the organization.
- World-wide about 500 organizations have been assessed at this level.

Key Process Areas

- Each level is associated with a key process area (KPA) identifies:
  - Where an organization at the previous level must focus to reach this level.

Level 2 KPAs

- Software project planning:
  - Size, cost, schedule.
  - Project monitoring
- Configuration management
- Subcontract management

Level 3 KPAs

- Process definition and documentation.
- Reviews
- Training program

Level 4 KPAs

- Quantitative measurements.
- Process management.

Level 5 KPAs

- Defect prevention.
- Technology change management.
- Process change management.

Software Project Management

Introduction

- Many software projects fail:
  - due to faulty project management practices:
    - It is important to learn different aspects of software project management.

Introduction

- Goal of software project management:
  - enable a group of engineers to work efficiently towards successful completion of a software project.

Responsibility of project managers

- Project proposal writing,
- Project cost estimation,
- Scheduling,
- Project staffing,
- Project monitoring and control,
- Software configuration management,
- Risk management,
- Managerial report writing and presentations, etc.

Introduction

- A project manager's activities are varied.
  - can be broadly classified into:
    - project planning,
    - project monitoring and control activities.

Project Planning

- Once a project is found to be feasible,
  - project managers undertake project planning.

Project Planning Activities

- Estimation:
  - Effort, cost, resource, and project duration
- Project scheduling:
- Staff organization:
  - staffing plans
- Risk handling:
  - identification, analysis, and abatement procedures
- Miscellaneous plans:
  - quality assurance plan, configuration management plan, etc.

Project planning

- Requires utmost care and attention --- commitments to unrealistic time and resource estimates result in:
  - irritating delays.
  - customer dissatisfaction
  - adverse affect on team morale
    - poor quality work
  - project failure.

Sliding Window Planning

- Involves project planning over several stages:
  - protects managers from making big commitments too early.
  - More information becomes available as project progresses.
    - Facilitates accurate planning

SPMP Document

- After planning is complete:
  - Document the plans:
  - in a Software Project Management Plan(SPMP) document.

Organization of SPMP Document

- Introduction (Objectives,Major Functions,Performance Issues,Management and Technical Constraints)
- Project Estimates (Historical Data,Estimation Techniques,Effort, Cost, and Project Duration Estimates)
- Project Resources Plan (People,Hardware and Software,Special Resources)
- Schedules (Work Breakdown Structure,Task Network, Gantt Chart Representation,PERT Chart Representation)
- Risk Management Plan (Risk Analysis,Risk Identification,Risk Estimation, Abatement Procedures)
- Project Tracking and Control Plan
- Miscellaneous Plans(Process Tailoring,Quality Assurance)

Software Cost Estimation

- Determine size of the product.
- From the size estimate,
  - determine the effort needed.
- From the effort estimate,
  - determine project duration, and cost.

Software Cost Estimation

Software Cost Estimation

- Three main approaches to estimation:
  - Empirical
  - Heuristic
  - Analytical

Software Cost Estimation Techniques

- Empirical techniques:
  - an educated guess based on past experience.
- Heuristic techniques:
  - assume that the characteristics to be estimated can be expressed in terms of some mathematical expression.
- Analytical techniques:
  - derive the required results starting from certain simple assumptions.

Software Size Metrics

- LOC (Lines of Code):
  - Simplest and most widely used metric.
  - Comments and blank lines should not be counted.

Disadvantages of Using LOC

- Size can vary with coding style.
- Focuses on coding activity alone.
- Correlates poorly with quality and efficiency of code.
- Penalizes higher level programming languages, code reuse, etc.

Disadvantages of Using LOC (cont…)

- Measures lexical/textual complexity only.
  - does not address the issues of structural or logical complexity.
- Difficult to estimate LOC from problem description.
  - So not useful for project planning

Function Point Metric

- Overcomes some of the shortcomings of the LOC metric
- Proposed by Albrecht in early 80's:
- Input:
  - A set of related inputs is counted as one input.

Function Point Metric

- Output:
  - A set of related outputs is counted as one output.
- Inquiries:
  - Each user query type is counted.
- Files:
  - Files are logically related data and thus can be data structures or physical files.
- Interface:
  - Data transfer to other systems.

- Unadjusted Function Point = #inputs *4 + #outputs * 5 + #inquiries*4 + #files*10 + #interfaces*10
- Technical complexity factor (such as high transaction rates, response time requirements, scope of reuse etc.)
  - 14 factors- value b/w 0-6
  - Degree of influence = sum of 14 factors
  - TCF = 0.65 + 0.01 *DI
- FP = UFP * TCF

- Simple, Average, Complex
- Input 3 4 6
- Output 4 5 7
- Inquiry 3 4 6
- Files 7 10 15
- Interface 5 7 10

Function Point Metric (CONT.)

- Suffers from a major drawback:
  - the size of a function is considered to be independent of its complexity.
- Extend function point metric:
  - Feature Point metric:
  - considers an extra parameter:
    - Algorithm Complexity.

Function Point Metric (CONT.)

- Proponents claim:
  - FP is language independent.
  - Size can be easily derived from problem description
- Opponents claim:
  - it is subjective --- Different people can come up with different estimates for the same problem.

Empirical Size Estimation Techniques

- Expert Judgement:
  - An euphemism for guess made by an expert.
  - Suffers from individual bias.
- Delphi Estimation:
  - overcomes some of the problems of expert judgement.

Expert judgement

- Experts divide a software product into component units:
  - e.g. GUI, database module, data communication module, billing module, etc.
- Add up the guesses for each of the components.

Delphi Estimation:

- Team of Experts and a coordinator.
- Experts carry out estimation independently:
  - mention the rationale behind their estimation.
  - coordinator notes down any extraordinary rationale:
    - circulates among experts.

Delphi Estimation:

- Experts re-estimate.
- Experts never meet each other
  - to discuss their viewpoints.

Heuristic Estimation Techniques

- Single Variable Model:
  - Parameter to be Estimated=C1(Estimated Characteristic)d1
- Multivariable Model:
  - Assumes that the parameter to be estimated depends on more than one characteristic.
  - Parameter to be Estimated=C1(Estimated Characteristic)d1+ C2(Estimated Characteristic)d2+…
  - Usually more accurate than single variable models.

COCOMO Model

- COCOMO (COnstructive COst MOdel) proposed by Boehm.
- Divides software product developments into 3 categories:
  - Organic
  - Semidetached
  - Embedded

COCOMO Product classes

- Roughly correspond to:
  - application, utility and system programs respectively.
    - Data processing and scientific programs are considered to be application programs.
    - Compilers, linkers, editors, etc., are utility programs.
    - Operating systems and real-time system programs, etc. are system programs.

Elaboration of Product classes

- Organic:
  - Relatively small groups
    - working to develop well-understood applications.
- Semidetached:
  - Project team consists of a mixture of experienced and inexperienced staff.
- Embedded:
  - The software is strongly coupled to complex hardware, or real-time systems.

COCOMO Model (CONT.)

- For each of the three product categories:
  - From size estimation (in KLOC), Boehm provides equations to predict:
    - project duration in months
    - effort in programmer-months
- Boehm obtained these equations:
  - examined historical data collected from a large number of actual projects.

COCOMO Model (CONT.)

- Software cost estimation is done through three stages:
  - Basic COCOMO,
  - Intermediate COCOMO,
  - Complete COCOMO.

Basic COCOMO Model (CONT.)

- Gives only an approximate estimation:
  - Effort = a1 (KLOC)a2
  - Tdev = b1 (Effort)b2
    - KLOC is the estimated kilo lines of source code,
    - a1,a2,b1,b2 are constants for different categories of software products,
    - Tdev is the estimated time to develop the software in months,
    - Effort estimation is obtained in terms of person months (PMs).

Development Effort Estimation

- Organic :
  - Effort = 2.4 (KLOC)1.05 PM
- Semi-detached:
  - Effort = 3.0(KLOC)1.12 PM
- Embedded:
  - Effort = 3.6 (KLOC)1.20PM

Development Time Estimation

- Organic:
  - Tdev = 2.5 (Effort)0.38 Months
- Semi-detached:
  - Tdev = 2.5 (Effort)0.35 Months
- Embedded:
  - Tdev = 2.5 (Effort)0.32 Months

Basic COCOMO Model (CONT.)

- Effort is somewhat super-linear in problem size.

Basic COCOMO Model (CONT.)

- Development time
  - sublinear function of product size.
- When product size increases two times,
  - development time does not double.
- Time taken:
  - almost same for all the three product categories.

Basic COCOMO Model (CONT.)

- Development time does not increase linearly with product size:
  - For larger products more parallel activities can be identified:
    - can be carried out simultaneously by a number of engineers.

Basic COCOMO Model (CONT.)

- Development time is roughly the same for all the three categories of products:
  - For example, a 60 KLOC program can be developed in approximately 18 months
    - regardless of whether it is of organic, semi-detached, or embedded type.
  - There is more scope for parallel activities for system and application programs,
    - than utility programs.

Example

- The size of an organic software product has been estimated to be 32,000 lines of source code.

.....

- Effort = $2.4*(32)^{1.05}$ = 91 PM
- Nominal development time = $2.5*(91)^{0.38}$ = 14 months

Intermediate COCOMO

- Basic COCOMO model assumes
  - effort and development time depend on product size alone.
- However, several parameters affect effort and development time:
  - Reliability requirements
  - Availability of CASE tools and modern facilities to the developers
  - Size of data to be handled

Intermediate COCOMO

- For accurate estimation,
  - the effect of all relevant parameters must be considered:
  - Intermediate COCOMO model recognizes this fact:
    - refines the initial estimate obtained by the basic COCOMO by using a set of 15 cost drivers (multipliers).

Intermediate COCOMO (CONT.)

- If modern programming practices are used,
  - initial estimates are scaled downwards.
- If there are stringent reliability requirements on the product :
  - initial estimate is scaled upwards.

Intermediate COCOMO
(CONT.)

- Rate different parameters on a scale of one to three:
  - Depending on these ratings,
    - multiply cost driver values with the estimate obtained using the basic COCOMO.

Intermediate COCOMO (CONT.)

- Cost driver classes:
  - Product: Inherent complexity of the product, reliability requirements of the product, etc.
  - Computer: Execution time, storage requirements, etc.
  - Personnel: Experience of personnel, etc.
  - Development Environment: Sophistication of the tools used for software development.

Shortcoming of basic and intermediate COCOMO models

- Both models:
  - consider a software product as a single homogeneous entity:
  - However, most large systems are made up of several smaller sub-systems.
    - Some sub-systems may be considered as organic type, some may be considered embedded, etc.
    - for some the reliability requirements may be high, and so on.

Complete COCOMO

- Cost of each sub-system is estimated separately.
- Costs of the sub-systems are added to obtain total cost.
- Reduces the margin of error in the final estimate.

Complete COCOMO Example

- A Management Information System (MIS) for an organization having offices at several places across the country:
  - Database part (semi-detached)
  - Graphical User Interface (GUI) part (organic)
  - Communication part (embedded)
- Costs of the components are estimated separately:
  - summed up to give the overall cost of the system.

Halstead's Software Science

- An analytical technique to estimate:
  - size,
  - development effort,
  - development time.

Halstead's Software Science

- Halstead used a few primitive program parameters
  - number of operators and operands
- Derived expressions for:
  - over all program length,
  - potential minimum volume
  - actual volume,
  - language level,
  - effort, and
  - development time.

- Study Halstead Software Science in Detail?

- Take any example of code, and try to find the value of these parameters using Halstead software science?

Organization Structure

- <u>Functional Organization:</u>
  - Engineers are organized into functional groups, e.g.
    - specification, design, coding, testing, maintenance, etc.
  - Engineers from functional groups get assigned to different projects

Advantages of Functional Organization

- Specialization
- Ease of staffing
- Good documentation is produced
  - different phases are carried out by different teams of engineers.
- Helps identify errors earlier.

Project Organization

- Engineers get assigned to a project for the entire duration of the project
  - Same set of engineers carry out all the phases
- Advantages:
  - Engineers save time on learning details of every project.
  - Leads to job rotation

- Compare functional organization and project organization?

Team Structure

- Problems of different complexities and sizes require different team structures:
  - Chief-programmer team
  - Democratic team
  - Mixed organization

Democratic Teams

- Suitable for:
  - small projects requiring less than five or six engineers
  - research-oriented projects
- A manager provides administrative leadership:
  - at different times different members of the group provide technical leadership.

Democratic Teams

- Democratic organization provides
  - higher morale and job satisfaction to the engineers
  - therefore leads to less employee turnover.
- Suitable for less understood problems,
  - a group of engineers can invent better solutions than a single individual.

Democratic Teams

- Disadvantage:
  - team members may waste a lot time arguing about trivial points:
    - absence of any authority in the team.

Chief Programmer Team

- A senior engineer provides technical leadership:
  - partitions the task among the team members.
  - verifies and integrates the products developed by the members.

Chief Programmer Team

- Works well when
  - the task is well understood
    - also within the intellectual grasp of a single individual,
  - importance of early completion outweighs other factors
    - team morale, personal development, etc.

Chief Programmer Team

- Chief programmer team is subject to single point failure:
  - too much responsibility and authority is assigned to the chief programmer.

Mixed Control Team Organization

- Draws upon ideas from both:
  - democratic organization and
  - chief-programmer team organization.
- Communication is limited
  - to a small group that is most likely to benefit from it.
- Suitable for large organizations.

Team Organization

Chief Programmer team

Democratic Team

Mixed team organization

Take any case-study of project and find the best possible team structure and organization structure for that ?

Staffing Level Estimation

- Number of personnel required during any development project:
  - not constant.
- Norden in 1958 analyzed many R&D projects, and observed:
  - Rayleigh curve represents the number of full-time personnel required at any time.

Rayleigh Curve

- Rayleigh curve is specified by two parameters:
  - td the time at which the curve reaches its maximum
  - K the total area under the curve.
- L=f(K, td)

Putnam's Work:

- In 1976, Putnam studied the problem of staffing of software projects:
  - observed that the level of effort required in software development efforts has a similar envelope.
  - found that the Rayleigh-Norden curve
    - relates the number of delivered lines of code to effort and development time.

Putnam's Work (CONT.):

- Putnam analyzed a large number of army projects, and derived the expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

  - K is the effort expended and L is the size in KLOC.
  - td is the time to develop the software.
  - Ck is the state of technology constant
    - reflects factors that affect programmer productivity.

Putnam's Work (CONT.):

- Ck=2 for poor development environment
  - no methodology, poor documentation, and review, etc.
- Ck=8 for good software development environment
  - software engineering principles used
- Ck=11 for an excellent environment

Rayleigh Curve

- Very small number of engineers are needed at the beginning of a project
  - carry out planning and specification.
- As the project progresses:
  - more detailed work is required,
  - number of engineers slowly increases and reaches a peak.

Rayleigh Curve

- Putnam observed that:
  - the time at which the Rayleigh curve reaches its maximum value
    - corresponds to system testing and product release.
  - After system testing,
    - the number of project staff falls till product installation and delivery.

Rayleigh Curve

- From the Rayleigh curve observe that:
  - approximately 40% of the area under the Rayleigh curve is to the left of td
  - and 60% to the right.

Effect of Schedule Change on Cost

- Using the Putnam's expression for L,

  $$K=L^3/C_k^3 t_d^4$$

  Or, $K=C/t_d^4$

- For the same product size, $C=L^3/C_k^3$ is a constant.

- Or, $K1/K2 = t_{d2}^4/t_{d1}^4$

Effect of Schedule Change on Cost (CONT.)

- Observe:
  - a relatively small compression in delivery schedule
    - can result in substantial penalty on human effort.
- Also, observe:
  - benefits can be gained by using fewer people over a somewhat longer time span.

Example

- If the estimated development time is 1 year, then in order to develop the product in 6 months,
  - the total effort and hence the cost increases 16 times.
  - In other words,
    - the relationship between effort and the chronological delivery time is highly nonlinear.

Effect of Schedule Change on Cost (CONT.)

- Putnam model indicates extreme penalty for schedule compression
  - and extreme reward for expanding the schedule.
- Putnam estimation model works reasonably well for very large systems,
  - but seriously overestimates the effort for medium and small systems.

Effect of Schedule Change on Cost (CONT.)

- Boehm observed:
  - "There is a limit beyond which the schedule of a software project cannot be reduced by buying any more personnel or equipment."
  - This limit occurs roughly at 75% of the nominal time estimate.

Effect of Schedule Change on Cost (CONT.)

- If a project manager accepts a customer demand to compress the development time by more than 25%
  - very unlikely to succeed.
    - every project has only a limited amount of parallel activities
    - sequential activities cannot be speeded up by hiring any number of additional engineers.
    - many engineers have to sit idle.

Jensen Model

- Jensen model is very similar to Putnam model.
  - attempts to soften the effect of schedule compression on effort
  - makes it applicable to smaller and medium sized projects.

Jensen Model

- Jensen proposed the equation:
  - $L = C_{te} t_d K^{1/2}$
  - Where,
    - Cte is the effective technology constant,
    - td is the time to develop the software, and
    - K is the effort needed to develop the software.
  - $K1/K2 = t_{d2}^2 / t_{d1}^2$

Quality Management

Introduction

- Traditional definition of quality:
  - Fitness of purpose:
    - A quality product does exactly what the users want it to do.

Fitness of Purpose

- Fitness of purpose for software products:
  - Satisfaction of the requirements specified in SRS document.

Fitness of Purpose

- A satisfactory definition of quality for many products:
  - A car, a table fan, a food mixer, microwave oven, etc.
- But, not satisfactory for software products.
  - Why?

Quality for Software Products

- Consider a software product:
  - Functionally correct:
    - Performs all functions as specified in the SRS document.
  - But, has an almost unusable user interface.
    - Cannot be considered as a quality product.

Quality for Software Products

- Consider another example:
  - A product which does everything that users want.
  - But has an almost incomprehensible and unmaintainable code.
  - Will you call it a quality product?

Quality

- The *American Heritage Dictionary* defines *quality* as
  - "a characteristic or attribute of something."
- For software, two kinds of quality may be encountered:
  - Quality of design encompasses requirements, specifications, and the design of the system.
  - Quality of conformance is an issue focused primarily on implementation.
  - User satisfaction = compliant product + good quality + delivery within budget and schedule

Quality—A Pragmatic View

- The *user view* sees quality in terms of an end-user's specific goals. If a product meets those goals, it exhibits quality.
- The *manufacturer's view* defines quality in terms of the original specification of the product. If the product conforms to the spec, it exhibits quality.
- The *product view* suggests that quality can be tied to inherent characteristics (e.g., functions and features) of a product.
- Finally, the *value-based view* measures quality based on how much a customer is willing to pay for a product. In reality, quality encompasses all of these views and more.

Software Quality

- Software quality can be defined as:
  - *An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.*
-

Modern View of Quality/Quality Dimensions

- Several quality factors are associated with a software product :
  - Correctness
  - Reliability
  - Efficiency (includes efficiency of resource utilization)
  - Portability
  - Usability
  - Reusability
  - Maintainability

Correctness

- A software product is correct:
  - If different requirements as specified in the SRS document have been correctly implemented.
  - Results are accurate.

Portability

- A software product is said to be portable:
  - If it can be easily made to work
    - In different operating systems.
    - In different machines,
    - With other software products, etc.

Reusability

- A software product has good reusability:
  - If different modules of the product can easily be reused to develop new products.

Usability

- A software product has good usability:
  - If different categories of users (i.e. both expert and novice users) can easily invoke the functions of the product.

Maintainability

- A software product is maintainable:
  - If errors can be easily corrected as and when they show up,
  - New functions can be easily added to the product,
  - Functionalities of the product can be easily modified.

Quality Dimensions

- **Performance Quality.** Does the software deliver all content, functions, and features that are specified as part of the requirements model in a way that provides value to the end-user?
- **Feature quality.** Does the software provide features that surprise and delight first-time end-users?
- **Reliability.** Does the software deliver all features and capability without failure? Is it available when it is needed? Does it deliver functionality that is error free?
- **Conformance.** Does the software conform to local and external software standards that are relevant to the application? Does it conform to de facto design and coding conventions? For example, does the user interface conform to accepted design rules for menu selection or data input?

Quality Dimensions

- **Durability.** Can the software be maintained (changed) or corrected (debugged) without the inadvertent generation of unintended side effects? Will changes cause the error rate or reliability to degrade with time?
- **Serviceability.** Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period. Can support staff acquire all information they need to make changes or correct defects?
- **Aesthetics.** Most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious "presence" that are hard to quantify but evident nonetheless.

Software Quality Management System

- Quality management system (or quality system):
  - Principal methodology used by organizations to ensure that the products have desired quality.

Quality System

- A quality system consists of the following:
  - Managerial Structure
  - Individual Responsibilities.
- Responsibility of the organization as a whole.

Quality System

- Every quality conscious organization has an independent quality department:
  - Performs several quality system activities.
  - Needs support of top management.
  - Without support at a high level in a company:
    - Many employees may not take the quality system seriously.

Quality System Activities

- Auditing of projects
- Development of:
  - standards, procedures, and guidelines.
- Production of reports for the top management:
  - Summarizing the effectiveness of the quality system in the organization.
- Review of the quality system itself.

Quality System

- A good quality system must be well documented.
  - Without a properly documented quality system,
    - Application of quality procedures become ad hoc,
    - Results in large variations in the quality of the products delivered.

Quality System

- An undocumented quality system:
  - Sends clear messages to the staff about the attitude of the organization towards quality assurance.
- International standards such as ISO 9000 provide:
  - Guidance on how to organize a quality system.

Evolution of Quality Systems

- Quality systems have evolved:
  - Over the last six decades.
- Prior to World War II:
  - Accepted way to produce quality products:
    - Inspect the finished products
    - Eliminate defective products.

Evolution of Quality Systems

- Since World war II,
  - Quality systems of organizations have undergone:
    - Four stages of evolution.
- Many advances came from Japanese:
  - Helped resurrect Japanese economy.

Evolution of Quality Systems

Inspection

Quality control (QC)

Quality assurance (QA)

Total Quality Management (TQM)

?

Product assurance

Process assurance

Evolution of Quality Systems

- Initial product inspection method:
  - Gave way to quality control (QC).
- Quality control:
  - Not only detect the defective products and eliminate them
  - But also determine the causes behind the defects.

Quality Control (QC)

- Quality control aims at correcting the causes of errors:
  - Not just rejecting defective products.
- Statistical quality control (SQC):
  - Quality of the output of the process is inferred using statistical methods.
  - Instead of inspection or testing of all products.

Quality Control (QC)

- The next breakthrough:
  - Development of quality assurance principles.

Quality Assurance

- Basic premise of modern quality assurance:
  - If an organization's processes are good and are followed rigorously:
    - The products are bound to be of good quality.

Quality Assurance

- All modern quality paradigms include:
  - Guidance for recognizing, defining, analyzing, and improving the production process.

Total Quality Management (TQM)

- TQM advocates:
  - Continuous process improvements through process measurements.

Business Process Reengineering

- BPR:A term related to TQM.
- Process reengineering goes a step further than quality assurance:
  - Aims at continuous process improvement.

Business Process Reengineering

- TQM focuses on reengineering of the software process.
  - Whereas BPR aims at reengineering the way business is carried out in any organization:
    - Not just software development.

Total Quality Management (TQM)

- TQM goes beyond documenting processes
  - Optimizes them through redesign.
- Over the years the quality paradigm has shifted:
  - From product assurance to process assurance.

Software Configuration Management (SCM)

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.

Bersoff, et al, 1980

What Are These Changes?

The Software Configuration

Baselines

- The IEEE (IEEE Std. No. 610.12-1990) defines a baseline as:
  - 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.
- a baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs (S/W Config. Indicators) that is obtained through a formal technical review

- Various objects: source, code, design document, SRS, manual etc.
- Configuration: state of all these objects at any point of time
- SCM: effectively tracking and controlling configu. of s/w product during its life cycle.

Version vs. Revision vs. Release

- Version:
  - An *initial* release or re-release of a configuration item associated with a *complete compilation* or recompilation of the item. Different versions have different functionality.
- Revision:
  - *Change* to a version that corrects only errors in the design/code, but does not affect the documented functionality.
- Release:
  - The *formal distribution* of an approved version.
  - Version m, Release n……

Necessity

- Inconsistency when objects are replicated
- Problem with concurrent access
- Provide stable development environment
- System accounting: who made changes and when
- Handing variants

Change management

- Change management is the handling of change requests
  - A change request leads to the creation of a new release
- General change process
  - The change is requested (this can be done by anyone including users and developers)
  - The change request is assessed against project goals
  - Following the assessment, the change is accepted or rejected
  - If it is accepted, the change is assigned to a developer and implemented
  - The implemented change is audited.

SCM Activities

- Config. Identification
- Configu. Control

Identification

- Decide the parts/objects on which control should be applies i.e should be kept track of
  - Controlled (should be under config control)
  - Pre-controlled (still not, but controllable)
  - Uncontrolled (not controlled)

Configuration Control

- Allow only authorized changes to the controlled objects
  - Existing baseline- 1, 2, 3, 4
  - Need changes in 1
  - ‘Reserve Private copy’ of 1- other can’t change
  - Apply changes
  - Get permission from change control board (CCB)
  - Not permitted: discard changes and cancel Reserve
  - Permitted: Restore

Tools for Software Configuration Management

- Software configuration management is normally supported by tools with different functionality.

- Examples:
  - RCS
    - very old but still in use; only version control system
  - CVS
    - based on RCS, allows concurrent working without locking
  - Perforce
    - Repository server; keeps track of developer's activities
  - ClearCase
    - Multiple servers, process modeling, policy check mechanisms

Computer Aided Software Engineering (CASE)

- CASE tools help in software development and maintenance.
- CASE is a much talked about topic in software industries.

CASE and Its Scope

- CASE tool is a generic term:
  - denotes any form of automated support for software engineering.
- In a more restrictive sense:
  - a CASE tool automates some software development activity.

CASE and Its Scope

- Some CASE tools assist in phase-related tasks:
  - specification, structured analysis, design, coding, testing, etc.
- Other tools help non-phase activities:
  - project management and configuration management.

Objectives of CASE

- To increase productivity
- To help produce better quality software at lower cost.

CASE Environment

- Although individual CASE tools are useful:
  - true power of a tool set can be realized only when:
    - all CASE tools are integrated together.

CASE Environment

- Tools covering different stages of life cycle share information (data):
  - they should integrate through some central repository (store)
  - consistent view of development information.

CASE Environment

- The central repository is the data dictionary:
  - contains definition of all composite and elementary data items.
  - through this repository all CASE tools share information.

Programming Environment

- A CASE environment helps:
  - automate step-by-step methodologies.
- In contrast to CASE environment:
  - a programming environment denotes tools supporting coding phase alone.

Schematic representation of architecture of CASE environment/Taxonomy

CASE Environment Model

An Integration Framework

- Object Management Layer
  - Case tools were represented as set of entities in Case-Tool Set Layer (spec, design, project plan etc.)
  - This layer maps these logical entities into underlying storage management system (repository)

Benefits of CASE

- A key benefit of using CASE environment:
  - cost saving through all developmental phases.
- Studies carried out to measure the impact of CASE usage:
  - cost saving between 30% to 40%.

Benefits of CASE

- Use of CASE tools leads to improvements in quality:
  - becomes easy to iterate through different software development phases.
  - chances of human error is reduced.
  - CASE tools help produce higher quality and consistent documents.

Benefits of CASE

- Data relating to a software product are maintained in a central repository:
  - redundancy in the stored data is reduced.
  - chances of inconsistent documentation is reduced.

Benefits of CASE

- CASE tools helps in software engineers' work:
  - engineers need not manually check balancing of the DFDs
  - easily draw diagrams and produce documentation, etc.

Benefits of CASE

- CASE tools lead to cost saving in software maintenance effort:
  - traceability and consistency checks,
  - systematic information capture during various development phases.

Benefits of CASE

- Introduction of a CASE environment:
  - impacts the style of working of engineers.
  - makes them oriented towards structured and orderly approach.

Prototyping Support

- Prototyping CASE tool:
  - often used in graphical user interface (GUI) development,
  - supports creating a GUI using a graphics editor.

Prototyping Support

- The user should be allowed to define:
  - data entry forms, menus and controls.
- It should integrate with the data dictionary of a CASE environment.

Structured Analysis and Design

- A CASE tool should:
  - support some standard structured analysis and design technique.
  - support easy creation of analysis and design diagrams.
  - should provide easy navigation through different levels of design and analysis diagrams.

Structured Analysis and Design

- The tool must support completeness and consistency checking.
- The tool should disallow inconsistent operations:
  - but, it is difficult to implement such a feature.

Code Generation

- As far as code generation is concerned:
  - expectations from a CASE tool is low.
- The CASE tool should support:
  - generation of module skeletons in one or more popular languages.
  - Another reasonable requirement is traceability from source code to design.

Code Generation

- It should automatically generate header information:
  - copyright messages,
  - brief description of the module,
  - author name and date of creation, etc.

Code Generation

- The tool should generate data records or structures automatically:
  - using data dictionary definitions.
  - It should generate database tables for relational database management systems.

Code Generation

- The tool should generate code for user interface from the prototype:

Testing Support

- Static and dynamic program analysis of programs.
- Should help in test case generation
- It should generate test reports in ASCII format:
  - which can be directly imported into the test plan document.

Desirable Features

- The tool should work satisfactorily
  - when many users work simultaneously.
- The tool should support windowing interface:
  - Enable the users to see more than one diagram at a time.
  - Facilitate navigation and switching from one part to the other.

Documentation Support

- The deliverable documents:
  - should be able to incorporate text and diagrams from the central repository.
  - help in producing up-to-date documentation.

Desirable Features

- The CASE tool should integrate
  - with commercially available desk-top publishing packages.
- It should be possible to export text, graphics, tables, data dictionary reports:

Project Management

- It should support collecting, storing, and analyzing information on the software project's progress:
  - such as the estimated task duration,
  - scheduled and actual task start, completion date, dates and results of the reviews, etc.

External Interface

- The tool should allow exchange of information for reusability of design.
  - The information exported by the tool should preferably be in ASCII format.
- The data dictionary should provide
  - a programming interface to access information.

Reverse Engineering Support

- The tool should support:
  - generating structure chart, DFD, and data dictionary from source code.
  - should populate the data dictionary from source code.
  - Support re-engineering: hierarchical/network data base to relational database

Data Dictionary Interface

- Data dictionary interface should provide
  - viewing and updating the data definitions.
  - print facility to obtain hard copy of the viewed screens.
  - analysis reports like cross-referencing, impact analysis, etc.
  - it should support a query language.

- Tutorial and Help Support

- 2nd Generation Case tools
  - Intelligent Diagramming Tech: would provide help in aesthetically and automatically lay out diagrams
  - Integration with implementation environment: integration b/w design and implementation
  - Data Dictionary standards: It will be required to integrate many tools from different vendors. So need to have standard on Data dictionary
  - Customization support: user should be allowed to define new types of objects and connections.

Requirements Analysis and Specification

Requirements Analysis and Specification

- **Many projects fail:**
  - **Because they start implementing the system.**
  - **Without determining whether they are building what the customer really wants.**

Requirements Analysis and Specification

- **It is important to learn:**
  - **Requirements analysis and specification techniques carefully.**

Requirements Analysis and Specification

- **Goals of requirements analysis and specification phase:**
  - **Fully understand the user requirements.**
  - **Remove inconsistencies, anomalies, etc. from requirements.**
  - **Document requirements properly in an SRS document.**

Requirements Analysis and Specification

- Consists of two distinct activities:
  - Requirements Gathering and Analysis
  - Specification

Requirements Engineering

- **Elicitation** — determining what the customer requires
- **Analysis & negotiation** — understanding the relationships among various customer requirements and shaping those relationships to achieve a successful result
- **Requirements specification** — building a tangible model of requirements

Requirements Engineering

- **System Modeling —** building a representation of requirements that can be assessed for correctness, completeness, and consistency
- **Validation** — reviewing the model
- **Management —** identify, control and track requirements and the changes that will be made to them

Who Carries Out Requirements Analysis and Specification?

- **The person who undertakes requirements analysis and specification:**
  - **Known as systems analyst:**
  - **Collects data pertaining to the product**
  - **Analyzes collected data:**
    - **To understand what exactly needs to be done.**
  - **Writes the Software Requirements Specification (SRS) document.**

Requirements Analysis and Specification

- **Final output of this phase:**
  - **Software Requirements Specification (SRS) Document.**
- **The SRS document is reviewed by the customer.**
  - **Reviewed SRS document forms the basis of all future development activities.**

Requirements Analysis

- **Requirements analysis consists of two main activities:**
  - **Requirements gathering**
  - **Analysis of the gathered requirements**

Requirements Gathering

- **Analyst gathers requirements through:**
  - **Observation of existing systems,**
  - **Studying existing procedures,**
  - **Discussion with the customer and end-users,**
  - **Analysis of what needs to be done, etc.**

Requirements Gathering

- **Also known as requirements elicitation.**
- **If the project is to automate some existing procedures**
  - **e.g., automating existing manual accounting activities,**
  - **The task of the system analyst is a little easier**
  - **Analyst can immediately obtain:**
    - **input and output formats**
    - **accurate details of the operational procedures**

Requirements Gathering Methods

- **1. Studying the existing documentation**
  - **Before visiting customer**

- **Interview**
  - **Delphi Technique**
    - **Gather information from different users' view**
    - **Analyst consolidate requirements as understood by him**
    - **Circulate for comments of all users**
    - **Feedback and refine**
    - **Repeat till user agree**

- **Task analysis**
  - **Each functionality as task**
  - **Analyze sub-tasks with help of customer**

- **Scenario analysis**
  - **A task can be performed in diff scenario**
  - **Each scenario is analyzed with help of user**
  - **Such as book issue task at Fresh issue, re-issue etc.**
- **Form analysis**
  - **Different input and output forms**
  - **Tasks to convert input to output**
- **Interview, site-visit, questionnaire distribution etc.**

Requirements Gathering (CONT.)

- In the absence of a working system,
  - Lot of imagination and creativity are required.
- Interacting with the customer to gather relevant data:
  - Requires a lot of experience.

Requirements Gathering (CONT.)

- **Some desirable attributes of a good system analyst:**
  - **Good interaction skills,**
  - **Imagination and creativity,**
  - **Experience.**

Requirements Gathering

Facilitated Application Specification Techniques

FAST Guidelines

- participants must attend entire meeting
- all participants are equal
- preparation is as important as meeting
- all pre-meeting documents are to be viewed as "proposed"
- off-site meeting location is preferred
- set an agenda and maintain it
- don't get mired in technical detail

Analysis of the Gathered Requirements

- **Main purpose of requirements analysis:**
  - **Clearly understand the user requirements,**
  - **Detect inconsistencies, ambiguities, and incompleteness.**
- **Incompleteness and inconsistencies:**
  - **Resolved through further discussions with the end-users and the customers.**

Ambiguity in Req.

- **Not properly mentioned**
  - **Such as ‘very high temp’, ‘sufficiently low marks’**

Inconsistent Requirement

- **Some part of the requirement:**
  - **contradicts with some other part.**
- **<u>Example:</u>**
  - **One customer says turn off heater and open water shower when temperature > 100 C**
  - **Another customer says turn off heater and turn ON cooler when temperature > 100 C**

Incomplete Requirement

- **Some requirements have been omitted:**
  - **Possibly due to oversight.**
- **<u>Example:</u>**
  - **The analyst has not recorded:**
    when temperature falls below 90 C
    - **heater should be turned ON**
    - **water shower turned OFF.**

Analysis of the Gathered Requirements (CONT.)

- **Requirements analysis involves:**
  - **Obtaining a clear, in-depth understanding of the product to be developed,**
  - **Remove all ambiguities and inconsistencies from the initial customer perception of the problem.**

Analysis of the Gathered Requirements (CONT.)

- **It is quite difficult to obtain:**
  - **A clear, in-depth understanding of the problem:**
    - **Especially if there is no working model of the problem.**

Analysis of the Gathered Requirements (CONT.)

- **Experienced analysts take considerable time:**
  - **To understand the exact requirements the customer has in his mind.**

Analysis of the Gathered Requirements (CONT.)

- **Without a clear understanding of the problem, it is impossible to develop a satisfactory system.**

Analysis of the Gathered Requirements(CONT.)

- **Several things about the project should be clearly understood by the analyst:**
  - **What is the problem?**
  - **Why is it important to solve the problem?**
  - **What are the possible solutions to the problem?**
  - **What complexities might arise while solving the problem?**

Analysis of the Gathered Requirements(CONT.)

- **Some anomalies and inconsistencies can be very subtle:**
  - **Escape even most experienced eyes.**
  - **If a formal model of the system is constructed,**
    - **Many of the subtle anomalies and inconsistencies get detected.**

Analysis of the Gathered Requirements(CONT.)

- **After collecting all data regarding the system to be developed,**
  - **Remove all inconsistencies and anomalies from the requirements,**
  - **Systematically organize requirements into a Software Requirements Specification (SRS) document.**

The Analysis Process

Software Requirements Specification

- **Main aim of requirements specification:**
  - **Systematically organize the requirements arrived during requirements analysis.**
  - **Document requirements properly.**

Software Requirements Specification

- **The SRS document is useful in various contexts:**
  - **Statement of user needs**
  - **Contract document**
  - **Reference document**
  - **Definition for implementation**

Software Requirements Specification: A Contract Document

- **Requirements document is a reference document.**
- **SRS document is a contract between the development team and the customer.**
  - **Once the SRS document is approved by the customer,**
    - **Any subsequent controversies are settled by referring the SRS document.**

Software Requirements Specification: A Contract Document

- **Once customer agrees to the SRS document:**
  - **Development team starts to develop the product according to the requirements recorded in the SRS document.**
- **The final product will be acceptable to the customer:**
  - **As long as it satisfies all the requirements recorded in the SRS document.**

- **SRS document used by:**
  - **User/customer**
  - **Marketing personnel**
  - **Software developer**
  - **Test engineer**
  - **User manual writer**
  - **Project manager**
  - **Service and maintenance engineer**

SRS Document (CONT.)

- **The SRS document is known as <u>black-box specification:</u>**
  - **The system is considered as a black box whose internal details are not known.**
  - **Only its visible external (i.e. input/output) behavior is documented.**

SRS Document (CONT.)

- **SRS document concentrates on:**
  - **What needs to be done**
  - **Carefully avoids the solution ("how to do") aspects.**
- **The SRS document serves as a contract**
  - **Between development team and the customer.**
  - **Should be carefully written**

Properties of a Good SRS Document

- **It should be concise**
  - **and at the same time should not be ambiguous.**
- **It should specify what the system must do**
  - **and not say how to do it.**
- **Easy to change.,**
  - **i.e. it should be well-structured.**
- **It should be consistent.**
- **It should be complete.**

Properties of a Good SRS Document (cont...)

- **It should be traceable**
  - **You should be able to trace which part of the specification corresponds to which part of the design, code, etc and vice versa.**
- **It should be verifiable. i.e. One can verify the statements written in it with the product**

SRS Document

- **Introduction.**
- **Functional Requirements**
- **Nonfunctional Requirements**
  - **External interface requirements**
  - **Performance requirements**
- **Goals of implementation**

SRS Document (CONT.)

- **SRS document, normally contains three important parts:**
  - **Functional requirements,**
  - **Non-functional requirements,**
  - **Goals of Implementation.**

Functional Req. (CONT.)

- **It is desirable to consider every system:**
  - **Performing a set of functions {fi}.**
  - **Each function fi considered as:**
  - **Transforming a set of input data to corresponding output data.**

Example: Functional Requirement

- **F1: Search Book**
  - **Input:**
    - **an author's name:**
  - **Output:**
    - **details of the author's books and the locations of these books in the library.**

Functional Requirements

- **Functional requirements describe:**
  - **A set of high-level requirements (from which the user can get some useful piece of work)**
  - **Each high-level requirement:**
    - **takes in some data from the user**
    - **outputs some data to the user**
  - **Each high-level requirement:**
    - **might consist of a set of identifiable functions**

Functional Requirements

- **For each high-level requirement:**
  - **Every function is described in terms of:**
    - **Input data set**
    - **Output data set**
    - **Crude Processing required to obtain the output data set from the input data set.**

Functional Requirements

- **A high-level function is one:**
  - **Using which the user can get some useful piece of work done.**
- **A high-level requirement typically involves:**
  - **Accepting some data from the user,**
  - **Transforming it to the required response, and then**
  - **Outputting the system response to the user.**

Example Functional Requirements

- **Req. 1:**
  - **Once the user selects the "search" option,**
    - **he is asked to enter the key words.**
  - **The system should output details of all books**
    - **whose title or author name matches any of the key words entered.**
    - **Details include: Title, Author Name, Publisher name, Year of Publication, ISBN Number, Catalog Number, Location in the Library.**

Req. 1:

- **R.1.1:**
  - **Input:** "search" option,
  - **Output:** user prompted to enter the key words.
- **R1.2:**
  - **Input:** key words
  - **Output:** Details of all books whose title or author name matches any of the key words.
    - Details include: Title, Author Name, Publisher name, Year of Publication, ISBN Number, Catalog Number, Location in the Library.
  - **Processing:** Search the book list for the keywords

Example Functional Requirements

- **Req. 2:**
  - **When the "renew" option is selected,**
    - **The user is asked to enter his membership number and password.**
  - **After password validation,**
    - **The list of the books borrowed by him are displayed.**
  - **The user can renew any of the books:**
    - **By clicking in the corresponding renew box.**

Req. 2:

- **R2.1:**
  - **Input:** "renew" option selected,
  - **Output:** user prompted to enter his membership number and password.
- **R2.2:**
  - **Input:** membership number and password
  - **Output:**
    - list of the books borrowed by user are displayed. User prompted to enter books to be renewed or
    - user informed about bad password
  - **Processing:** Password validation, search books issued to the user from borrower list and display.

Req. 2:

- **R2.3:**
  - **Input:** user choice for renewal of the books issued to him through mouse clicks in the corresponding renew box.
  - **Output:** Confirmation of the books renewed
  - **Processing:** Renew the books selected by them in the borrower list.

Nonfunctional Requirements

- **Characteristics of the system which can not be expressed as functions:**
  - **Maintainability,**
  - **Portability,**
  - **Usability, etc.**

Nonfunctional Requirements

- **Nonfunctional requirements include:**
  - **Reliability issues,**
  - **Performance issues:**
    - **Example: How fast the system can produce results**
      - **so that it does not overload another system to which it supplies data, etc.**
  - **Human-computer interface issues,**
  - **Interface with other external systems,**
  - **Security, maintainability, etc.**

Non-Functional Requirements

- **Hardware to be used,**
- **Operating system**
  - **or DBMS to be used: specific recommendation of user**
- **Capabilities of I/O devices**
- **Standards compliance**
- **Data representations**
  - **by the interfaced system**

- IEEE 870: Non-Func Req.
  - External Interface Req.
  - Performance Req.
  - Constraints
  - S/w system attributes

Goals of Implementation

- **Goals describe things that are desirable of the system:**
  - **But, would not be checked for compliance.**
  - **For example,**
    - **Reusability issues**
    - **Functionalities to be developed in future**

Properties of Bad SRS Documents

- **<u>Unstructured Specifications:</u>**
  - **Narrative essay --- one of the worst types of specification document:**
    - **Difficult to change,**
    - **Difficult to be precise,**
    - **Difficult to be unambiguous,**
    - **Scope for contradictions, etc.**

- **<u>Noise:</u>**
  - **Presence of text containing information irrelevant to the problem.**
- **<u>Silence:</u>**
  - **aspects important to proper solution of the problem are omitted.**

- **<u>Overspecification:</u>**
  - **Addressing "how to" aspects**
  - **For example, "Library member names should be stored in a sorted descending order"**
  - **Overspecification restricts the solution space for the designer.**
- **<u>Contradictions:</u>**
  - **Contradictions might arise**
    - **if the same thing described at several places in different ways.**

- **<u>Ambiguity/Uncertainty/ Vagueness:</u>**

- **<u>Forward References:</u>**
  - **References to aspects of problem**
    - **defined only later on in the text.**
- **<u>Wishful Thinking:</u>**
  - **Descriptions of aspects**
    - **for which realistic solutions will be hard to find.**

Org. of SRS

- IEEE 830 standard
- Introduction: Purpose, hardware, people, peripherals etc.
- Functional Req.: User class1 Req, User class 2 Req.
- Non-Fun Req
- Behavioral Description: system state, events & actions

Design

Introduction

- **Design phase transforms SRS document:**
  - **To a form easily implementable in some programming language.**

Analysis to Design

THE ANALYSIS MODEL

THE DESIGN MODEL

Fundamental Concepts

- abstraction—**data, procedure, control**
- refinement—**elaboration of detail for all abstractions**
- modularity—**compartmentalization of data and function**
- architecture—**overall structure of the software**
  - **Structural properties**
  - **Extra-structural properties**
  - **Styles and patterns**
- procedure—**the algorithms that achieve function**
- hiding—**controlled interfaces**

Data Abstraction

Stepwise Refinement

Modular Design

easier to build, easier to change, easier to fix ...

Modularity: Trade-offs

What is the "right" number of modules for a specific software design?

Sizing Modules: Two Views

Functional Independence

COHESION - the degree to which a module performs one and only one function.

COUPLING - the degree to which a module is "connected" to other modules in the system.

Architecture

"The overall structure of the software and the ways in which that structure provides conceptual integrity for a system." [SHA95a]

Information Hiding

Why Information Hiding?

- reduces the likelihood of “side effects”
- limits the global impact of local design decisions
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software

Items Designed During Design Phase

- **Module structure,**
- **Control relationship among the modules**
  - **call relationship or invocation relationship**
- **Interface among different modules,**
  - **Data items exchanged among different modules,**
- **Data structures of individual modules,**
- **Algorithms for individual modules.**

Module Structure

Introduction

- A module consists of:
  - Several functions
  - Associated data structures.

Introduction

- **Good software designs:**
  - **Seldom arrived through a single step procedure:**
  - **But through a series of steps and iterations.**

Introduction

- **Design activities are usually classified into two stages:**
  - **Preliminary (or high-level) design.**
  - **Detailed design.**
- **Meaning and scope of the two stages:**
  - **Vary considerably from one methodology to another.**

High-Level Design

- **Identify:**
  - **Modules**
  - **Control relationships among modules**
  - **Interfaces among modules.**

High-Level Design

- **The outcome of high-level design:**
  - **Program structure (or software architecture).**

High-Level Design

- **Several notations are available to represent high-level design:**
  - **Usually a tree-like diagram called structure chart is used.**
  - **Other notations:**
    - **Jackson diagram or Warnier-Orr diagram can also be used.**

Detailed Design

- For each module, design:
  - Data structure
  - Algorithms
- Outcome of detailed design:
  - Module specification.

A Classification of Design Methodologies

- Procedural (aka Function-oriented)
- Object-oriented
- More recent:
  - Aspect-oriented
  - Component-based (Client-Server)

Does a Design Technique Lead to a Unique Solution?

- **No:**
  - **Several subjective decisions need to be made to trade off among different parameters.**
  - **Even the same designer can come up with several alternate design solutions.**

Analysis versus Design

- **An analysis technique helps elaborate the customer requirements through careful thinking:**
  - **And at the same time consciously avoids making any decisions regarding implementation.**
- **The design model is obtained from the analysis model through transformations over a series of steps:**
  - **Decisions regarding implementation are consciously made.**

Choose b/w two alternate design

- How to distinguish between the superior of two alternate design solutions?
  - Unless we know what a good software design is:
    - We can not possibly design one.

Good and Bad Designs

- **There is no unique way to design a system.**
- **Even using the same design methodology:**
  - **Different designers can arrive at very different design solutions.**
- **We need to distinguish between good and bad designs.**

Which of Two is a Better Design?

- **Should implement all functionalities of the system correctly.**
- **Should be easily understandable.**
- **Should be efficient.**
- **Should be easily amenable to change,**
  - **i.e. easily maintainable.**

Which of Two is a Better Design?

- **Understandability of a design is a major issue:**
  - **Determines goodness of design:**
  - **A design that is easy to understand:**
    - **Also easy to maintain and change.**

Which of Two is a Better Design?

- **Unless a design is easy to understand,**
  - **Tremendous effort needed to maintain it**
  - **We already know that about 60% effort is spent in maintenance.**
- **If the software is not easy to understand:**
  - **Maintenance effort would increase many times.**

Understandability

- **Use consistent and meaningful names:**
  - **For various design components.**
- **Should make use of abstraction and decomposition principles in ample measure.**

How are Abstraction and Decomposition Principles Used in Design?

- **Two principal ways:**
  - **Modular Design**
  - **Layered Design**

Modularity

- **Modularity is a fundamental attributes of any good design.**
  - **Decomposition of a problem cleanly into modules:**
  - **Modules are almost independent of each other**
  - **Divide and conquer principle.**

Modularity

- If modules are independent:
  - Modules can be understood separately,
    - Reduces the complexity greatly.
  - To understand why this is so,
    - Remember that it is very difficult to break a bunch of sticks but very easy to break the sticks individually.

Layered Design

Layered Design

- **Neat arrangement of modules in a hierarchy means:**
  - **Low fan-out**
  - **Control abstraction**

Modularity

- **In technical terms, modules should display:**
  - **High cohesion**
  - **Low coupling.**

Cohesion and Coupling

- **Cohesion is a measure of:**
  - **functional strength of a module.**
  - **A cohesive module performs a single task or function.**
- **Coupling between two modules:**
  - **A measure of the degree of the interdependence or interaction between the two modules.**

- **Unfortunately, there are no ways:**
  - **To quantitatively measure the degree of cohesion and coupling.**
  - **Classification of different kinds of cohesion and coupling:**
    - **Can give us some idea regarding the degree of cohesiveness of a module.**

Classification of Cohesiveness

- **Classification is often subjective:**
  - **Yet gives us some idea about cohesiveness of a module.**
- **By examining the type of cohesion exhibited by a module:**
  - **We can roughly tell whether it displays high cohesion or low cohesion.**

Classification of Cohesiveness

Coincidental Cohesion

- **The module performs a set of tasks:**
  - **Which relate to each other very loosely, if at all.**
    - **The module contains a random collection of functions.**
    - **Functions have been put in the module out of pure coincidence without any thought or design.**

Logical Cohesion

- **All elements of the module perform similar operations:**
  - **e.g. error handling, data input, data output, etc.**
- **An example of logical cohesion:**
  - **A set of print functions to generate an output report arranged into a single module.**

Temporal Cohesion

- **The module contains tasks that are related by the fact:**
  - **All the tasks must be executed in the same time span.**
- **Example:**
  - **The set of functions responsible for**
    - **initialization,**
    - **start-up, shut-down of some process, etc.**
    - **Module for computer booting**

Procedural Cohesion

- **The set of functions of the module:**
  - **All part of a procedure (algorithm)**
  - **Certain sequence of steps have to be carried out in a certain order for achieving an objective,**
    - **e.g. the algorithm for decoding a message.**
    - **Functions executed in a sequence, but may be for different purposes and operate on different data.**

Communicational Cohesion

- **All functions of the module:**
  - **Reference or update the same data structure,**
- **Example:**
  - **The set of functions defined on an array or a stack.**

Sequential Cohesion

- **Elements of a module form different parts of a sequence,**
  - **Output from one element of the sequence is input to the next.**
  - **Example:**

Functional Cohesion

- **Different elements of a module cooperate:**
  - **To achieve a single function,**
  - **e.g. managing an employee's pay-roll.**
- **When a module displays functional cohesion,**
  - **We can describe the function using a single sentence.**

Determining Cohesiveness

- **Write down a sentence to describe the function of the module**
  - **If the sentence is compound,**
    - **It has a sequential or communicational cohesion.**
  - **If it has words like "first", "next", "after", "then", etc.**
    - **It has sequential or temporal cohesion.**
  - **If it has words like initialize,**
    - **It probably has temporal cohesion.**

Coupling

- **Coupling indicates:**
  - **How closely two modules interact or how interdependent they are.**
  - **The degree of coupling between two modules depends on their interface complexity.**

Coupling

- **There are no ways to precisely determine coupling between two modules:**
  - **Classification of different types of coupling will help us to approximately estimate the degree of coupling between two modules.**
- **Five types of coupling can exist between any two modules.**

Classes of coupling

Data coupling

- **Two modules are data coupled,**
  - **If they communicate via a parameter:**
    - **an elementary data item,**
    - **e.g an integer, a float, a character, etc.**
  - **The data item should be problem related:**
    - **Not used for control purpose.**

Stamp Coupling

- **Two modules are <u>stamp coupled</u>,**
  - **If they communicate via a composite data item**
    - **such as a record in PASCAL**
    - **or a structure in C.**

Control Coupling

- **Data from one module is used to direct:**
  - **Order of instruction execution in another.**
- **Example of control coupling:**
  - **A flag set in one module and tested in another module.**

Common Coupling

- **Two modules are <u>common coupled</u>,**
  - **If they share some global data.**

Content Coupling

- **Content coupling exists between two modules:**
  - **If they share code,**
  - **e.g, branching from one module into another module.**
- **The degree of coupling increases**
  - **from data coupling to content coupling.**

Functional Independence

- **A module having high cohesion and low coupling:**
  - **functionally independent of other modules:**
    - **A functionally independent module has minimal interaction with other modules.**

Advantages of Functional Independence

- **Better understandability and good design:**
- **Complexity of design is reduced,**
- **Different modules easily understood in isolation:**
  - **Modules are independent**

Advantages of Functional Independence

- **Functional independence reduces error propagation.**
  - **Degree of interaction between modules is low.**
  - **An error existing in one module does not directly affect other modules.**
- **Reuse of modules is possible.**

Advantages of Functional Independence

- A functionally independent module:
  - Can be easily taken out and reused in a different program.
    - Each module does some well-defined and precise function
    - The interfaces of a module with other modules is simple and minimal.

Neat Hierarchy

- **Control hierarchy represents:**
  - **Organization of modules.**
  - **Control hierarchy is also called <u>program structure.</u>**
- **Most common notation:**
  - **A tree-like diagram called <u>structure chart.</u>**

Layered Design

- **Essentially means:**
  - **Low fan-out**
  - **Control abstraction**

Characteristics of Module Hierarchy

- **Depth:**
  - **Number of levels of control**
- **Width:**
  - **Overall span of control.**
- **Fan-out:**
  - **A measure of the number of modules directly controlled by given module.**

Characteristics of Module Structure

- **Fan-in:**
  - **Indicates how many modules directly invoke a given module.**
  - **High fan-in represents code reuse and is in general encouraged.**

Module Structure

Layered Design

- **A design having modules:**
  - **With high fan-out numbers is not a good design:**
  - **A module having high fan-out lacks cohesion.**

Goodness of Design

- **A module that invokes a large number of other modules:**
  - **Likely to implement several different functions:**
  - **Not likely to perform a single cohesive function.**

Control Relationships

- **A module that controls another module:**
  - **Said to be superordinate to it.**
- **Conversely, a module controlled by another module:**
  - **Said to be subordinate to it.**

Visibility and Layering

- **A module A is said to be visible by another module B,**
  - **If A directly or indirectly calls B.**
- **The layering principle requires**
  - **Modules at a layer can call only the modules immediately below it.**

Bad Design

Abstraction

- **A module is unaware (how to invoke etc.) of the higher level modules.**
- **Lower-level modules:**
  - **Do input/output and other low-level functions.**
- **Upper-level modules:**
  - **Do more managerial functions.**

Abstraction

- **The principle of abstraction requires:**
  - **Lower-level modules do not invoke functions of higher level modules.**
  - **Also known as <u>layered design</u>.**

High-level Design

- **High-level design maps functions into modules {fi} {mj} such that:**
  - **Each module has high cohesion**
  - **Coupling among modules is as low as possible**
  - **Modules are organized in a neat hierarchy**

High-level Design

Design Approaches

- **Two fundamentally different software design approaches:**
  - **Function-oriented design**
  - **Object-oriented design**

Design Approaches

- These two design approaches are radically different.
  - However, are complementary
    - Rather than competing techniques.
  - Each technique is applicable at
    - Different stages of the design process.

Function-Oriented Design

- A system is looked upon as something
  - That performs a set of functions.
- Starting at this high-level view of the system:
  - Each function is successively refined into more detailed functions.
  - Functions are mapped to a module structure.

Example

- **The function create-new-library-member:**
  - **Creates the record for a new member,**
  - **Assigns a unique membership number**
  - **Prints a bill towards the membership**

Example

- **Create-library-member function consists of the following sub-functions:**
  - **Assign-membership-number**
  - **Create-member-record**
  - **Print-bill**

Function-Oriented Design

- **Each subfunction:**
  - **Split into more detailed subfunctions and so on.**

Function-Oriented Design

- **The system state is centralized:**
  - **Accessible to different functions,**
  - **Member-records:**
    - **Available for reference and updation to several functions:**
      - **Create-new-member**
      - **Delete-member**
      - **Update-member-record**

Function-Oriented Design

- **Several function-oriented design approaches have been developed:**
  - **Structured design (Constantine and Yourdon, 1979)**
  - **Jackson's structured design (Jackson, 1975)**
  - **Warnier-Orr methodology**
  - **Wirth's step-wise refinement**
  - **Hatley and Pirbhai's Methodology**

Object-Oriented Design

- System is viewed as a collection of objects (i.e. entities).
- System state is decentralized among the objects:
  - Each object manages its own state information.

Object-Oriented Design Example

- **Library Automation Software:**
  - **Each library member is a separate object**
    - **With its own data and functions.**
  - **Functions defined for one object:**
    - **Cannot directly refer to or change data of other objects.**

Object-Oriented Design

- **Objects have their own internal data:**
  - **Defines their state.**
- **Similar objects constitute a class.**
  - **Each object is a member of some class.**
- **Classes may inherit features**
  - **From a super class.**
- **Conceptually, objects communicate by message passing.**

Object-Oriented versus Function-Oriented Design

- Unlike function-oriented design,
  - In OOD the basic abstraction is not functions such as "sort", "display", "track", etc.,
  - But real-world entities such as "employee", "picture", "machine", "radar system", etc.

Object-Oriented versus Function-Oriented Design

- In OOD:
  - Software is not developed by designing functions such as:
    - update-employee-record,
    - get-employee-address, etc.
  - But by designing objects such as:
    - employees,
    - departments, etc.

Object-Oriented versus Function-Oriented Design

- **Grady Booch sums up this fundamental difference saying:**
  - **"Identify verbs if you are after procedural design and nouns if you are after object-oriented design."**

Object-Oriented versus Function-Oriented Design

- **In OOD:**
  - **State information is not shared in a centralized data.**
  - **But is distributed among the objects of the system.**

Object-Oriented versus Function-Oriented Design

- Objects communicate by message passing.
  - One object may discover the state information of another object by interrogating it.

Object-Oriented versus Function-Oriented Design

- **Of course, somewhere or other the functions must be implemented:**
  - **The functions are usually associated with specific real-world entities (objects)**
  - **Directly access only part of the system state information.**

Object-Oriented versus Function-Oriented Design

- **Function-oriented techniques group functions together if:**
  - **As a group, they constitute a higher level function.**
- **On the other hand, object-oriented techniques group functions together:**
  - **On the basis of the data they operate on.**

Object-Oriented versus Function-Oriented Design

- To illustrate the differences between object-oriented and function-oriented design approaches,
  - let us consider an example ---
  - An automated fire-alarm system for a large building.

Fire-Alarm System

- **We need to develop a computerized fire alarm system for a large multi-storied building:**
  - **There are 80 floors and 1000 rooms in the building.**

Fire-Alarm System

- **Different rooms of the building:**
  - **Fitted with smoke detectors and fire alarms.**
- **The fire alarm system would monitor:**
  - **Status of the smoke detectors.**

Fire-Alarm System

- **Whenever a fire condition is reported by any smoke detector:**
  - **the fire alarm system should:**
    - **Determine the location from which the fire condition was reported**
    - **Sound the alarms in the neighboring locations.**

Fire-Alarm System

- **The fire alarm system should:**
  - **Flash an alarm message on the computer console:**
    - **Fire fighting personnel see the console round the clock.**

Fire-Alarm System

- **After a fire condition has been successfully handled,**
  - **The fire alarm system should let fire fighting personnel reset the alarms.**

Function-Oriented Approach:

- /* Global data (system state) accessible by various functions */

```
BOOL  detector_status[1000];
int   detector_locs[1000];
BOOL  alarm-status[1000]; /* alarm activated when status set */
int   alarm_locs[1000]; /* room number where alarm is located */
int   neighbor-alarms[1000][10];/*each detector has at most*/
                        /* 10 neighboring alarm locations */
```

The functions which operate on the system state:

```
interrogate_detectors();
get_detector_location();
determine_neighbor();
ring_alarm();
reset_alarm();
report_fire_location();
```

Object-Oriented Approach:

- class detector
- attributes: status, location, neighbors
- operations: create, sense-status, get-location,
- find-neighbors
- class alarm
- attributes: location, status
- operations: create, ring-alarm, get_location,
- reset-alarm
- In the object oriented program,
  - appropriate number of instances of the class detector and alarm should be created.

Object-Oriented versus Function-Oriented Design

- **In the function-oriented program :**
  - **The system state is centralized**
  - **Several functions accessing these data are defined.**
- **In the object oriented program,**
  - **The state information is distributed among various sensor and alarm objects.**

Object-Oriented versus Function-Oriented Design

- Use OOD to design the classes:
  - Then applies top-down function oriented techniques
    - To design the internal methods of classes.

Object-Oriented versus Function-Oriented Design

- **Though outwardly a system may appear to have been developed in an object oriented fashion,**
  - **But inside each class there is a small hierarchy of functions designed in a top-down manner.**

Coding, Testing and Maintenance

Coding

Coding

- The input to the coding phase is the design document.
- During coding phase:
  - modules identified in the design document are coded according to the module specifications.

Coding

- At the end of the design phase we have:
  - module structure (e.g. structure chart) of the system
  - module specifications:
    - data structures and algorithms for each module.
- Objective of coding phase:
  - transform design into code
  - unit test the code.

Coding Standards

- Good software development organizations require their programmers to:
  - adhere to some standard style of coding
  - called coding standards.

Coding Standards

- Many software development organizations:
  - formulate their own coding standards that suits them most,
  - require their engineers to follow these standards rigorously.

Coding Standards

- Advantage of adhering to a standard style of coding:
  - it gives a uniform appearance to the codes written by different engineers,
  - it enhances code understanding,
  - encourages good programming practices.

Coding Standards

- A coding standard
  - sets out standard ways of doing several things:
    - the way variables are named,
    - code is laid out,
    - maximum number of source lines allowed per function, etc.

Coding guidelines

- Provide general suggestions regarding coding style to be followed:
  - leave actual implementation of the guidelines:
    - to the discretion of the individual engineers.

Coding Standards and Guidelines

- Good organizations usually develop their own coding standards and guidelines:
  - depending on what best suits their organization.

Representative Coding Standards

- Rules for limiting the use of globals:
  - what types of data can be declared global and what can not.
- Naming conventions for
  - global variables,
  - local variables, and
  - constant identifiers.

Representative Coding Standards

- Contents of headers for different modules:
  - The headers of different modules should be standard for an organization.
  - The exact format for header information is usually specified.

Representative Coding Standards

- Header data:
  - Name of the module,
  - date on which the module was created,
  - author's name,
  - modification history,
  - synopsis of the module,
  - different functions supported, along with their input/output parameters,
  - global variables accessed/modified by the module.

Representative Coding Standards

- Error return conventions and exception handling mechanisms.
  - the way error and exception conditions are handled should be standard within an organization.

Representative Coding Guidelines

- Do not use too clever and difficult to understand coding style.
  - Code should be easy to understand.
- Many inexperienced engineers actually take pride:
  - in writing cryptic and incomprehensible code.

Representative Coding Guidelines

- Clever coding can obscure meaning of the code:
  - hampers understanding.
  - makes later maintenance difficult.

Representative Coding Guidelines

- The side effects of a function call include:
  - modification of parameters passed by reference,
  - modification of global variables,
  - I/O operations.
- An obscure side effect:
  - one that is not obvious from a casual examination of the code.

Representative Coding Guidelines

- Obscure side effects make it difficult to understand a piece of code.
- For example,
  - if a global variable is changed obscurely in a called module,
  - it becomes difficult for anybody trying to understand the code.

Representative Coding Guidelines

- Do not use an identifier (variable name) for multiple purposes.
  - Programmers often use the same identifier for multiple purposes.
  - For example, some programmers use a temporary loop variable
    - also for storing the final result.

Example use of a variable for multiple purposes

- for(i=1;i<100;i++)

```
    {.....}
   i=2*p*q;
return(i);
```

Use of a variable for multiple purposes

- The rationale given by programmers for such use:
  - memory efficiency:
  - e.g. three variables use up three memory locations,
  - whereas the same variable used in three different ways uses just one memory location.

Use of a variable for multiple purposes

- There are several things wrong with this approach:
  - hence should be avoided.
- Each variable should be given a name indicating its purpose:
  - This is not possible if an identifier is used for multiple purposes.

Use of a variable for multiple purposes

- Leads to confusion and annoyance
  - for anybody trying to understand the code.
  - Also makes future maintenance difficult.

Representative Coding Guidelines

- Code should be well-documented.
- Rules of thumb:
  - on the average there must be at least one comment line
    - for every three source lines.
  - The length of any function should not exceed 10 source lines.

Representative Coding Guidelines

- Lengthy functions:
  - usually very difficult to understand
  - probably do too many different things.

Representative Coding Guidelines

- Do not use goto statements.
- Use of goto statements:
  - make a program unstructured
  - make it very difficult to understand.

Code inspection and code walk throughs

- After a module has been coded,
  - code inspection and code walk through are carried out
  - ensures that coding standards are followed
  - helps detect as many errors as possible before testing.

Code inspection and code walk throughs

- Detect as many errors as possible during inspection and walkthrough:
  - detected errors require less effort for correction
    - much higher effort needed if errors were to be detected during integration or system testing.

Code Walk Through

- An informal code analysis technique.
  - undertaken after the coding of a module is complete.
- A few members of the development team select some test cases:
  - simulate execution of the code by hand using these test cases.

Code Walk Through

- Even though an informal technique:
  - several guidelines have evolved over the years
  - making this naive but useful analysis technique more effective.
  - These guidelines are based on
    - personal experience, common sense, and several subjective factors.

Code Walk Through

- The guidelines should be considered as examples:
  - rather than accepted as rules to be applied dogmatically.
- The team performing code walk through should not be either too big or too small.
  - Ideally, it should consist of between three to seven members.

Code Walk Through

- Discussion should focus on discovery of errors:
  - and not on how to fix the discovered errors.
- To foster cooperation:
  - avoid the feeling among engineers that they are being evaluated in the code walk through meeting,
  - managers should not attend the walk through meetings.

Code Inspection

- In contrast to code walk throughs,
  - code inspection aims mainly at discovery of commonly made errors.
- During code inspection:
  - the code is examined for the presence of certain kinds of errors,
  - in contrast to the hand simulation of code execution done in code walk throughs.

Code Inspection

- It is more likely that such an error will be discovered:
  - by looking for this kind of mistakes in the code,
  - rather than by simply hand simulating execution of the procedure.

Code Inspection

- Good software development companies:
  - collect statistics of errors committed by their engineers
  - identify the types of errors most frequently committed.
- A list of common errors:
  - can be used during code inspection to look out for possible errors.

Commonly made errors

- Use of uninitialized variables.
- Nonterminating loops.
- Array indices out of bounds.
- Incompatible assignments.
- Improper storage allocation and deallocation.
- Actual and formal parameter mismatch in procedure calls.
- Jumps into loops.

- Use of incorrect logical operators
  - or incorrect precedence among operators.
- Improper modification of loop variables.
- Also during code inspection,
  - adherence to coding standards is checked.

Testing

How Do You Test a Program?

- Input test data to the program.
- Observe the output:
  - Check if the program behaved as expected.

How Do You Test a Program?

- If the program does not behave as expected:
  - Note the conditions under which it failed.
  - Later debug and correct.

Overview of Testing Activities

- Test Suite Design
- Run test cases and observe results to detect failures.
- Debug to locate errors
- Correct errors.

Error, Faults, and Failures

- A failure is a manifestation/sign of an error (aka defect or bug).
  - Mere presence of an error may not lead to a failure.

Error, Faults, and Failures

- A fault is an incorrect state entered during program execution:
  - A variable value is different from what it should be.
  - A fault may or may not not lead to a failure.

Test cases and Test suites

- Test a software using a set of carefully designed test cases:
  - The set of all test cases is called the <u>test suite</u>

Test cases and Test suites

- A test case is a triplet [I,S,O]
  - I is the data to be input to the system,
  - S is the state of the system at which the data will be input,
  - O is the expected output of the system.

Verification versus Validation

- Verification is the process of determining:
  - Whether output of one phase of development conforms to its previous phase.
- Validation is the process of determining:
  - Whether a fully developed system conforms to its SRS document.

Verification versus Validation

- Verification is concerned with phase containment of errors,
  - Whereas the aim of validation is that the final product be error free.

Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical:
  - Input data domain is extremely large.
- Design an optimal test suite:
  - Of reasonable size and
  - Uncovers as many errors as possible.

Design of Test Cases

- If test cases are selected randomly:
  - Many test cases would not contribute to the significance of the test suite,
  - Would not detect errors not already being detected by other test cases in the suite.
- Number of test cases in a randomly selected test suite:
  - Not an indication of effectiveness of testing.

Design of Test Cases

- Testing a system using a large number of randomly selected test cases:
  - Does not mean that many errors in the system will be uncovered.
- Consider foloowing example:
  - Find the maximum of two integers x and y.

Design of Test Cases

- The code has a simple programming error:
- If (x>y) max = x;
  else max = x;
- Test suite {(x=3,y=2);(x=2,y=3)} can detect the error,
- A larger test suite {(x=3,y=2);(x=4,y=3); (x=5,y=1)} does not detect the error.

Design of Test Cases

- Systematic approaches are required to design an optimal test suite:
  - Each test case in the suite should detect different errors.

Design of Test Cases

- There are essentially two main approaches to design test cases:
  - Black-box approach
  - White-box (or glass-box) approach

Black-Box Testing

- Test cases are designed using only functional specification of the software:
  - Without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as functional testing.

White-box Testing

- Designing white-box test cases:
  - Requires knowledge about the internal structure of software.
  - White-box testing is also called structural testing.

Black-Box Testing

- There are essentially two main approaches to design black box test cases:
  - Equivalence class partitioning
  - Boundary value analysis

Equivalence Class Partitioning

- Input values to a program are partitioned into equivalence classes.
- Partitioning is done such that:
  - Program behaves in similar ways to every input value belonging to an equivalence class.

Why Define Equivalence Classes?

- Test the code with just one representative value from each equivalence class:
  - As good as testing using any other values from the equivalence classes.

Equivalence Class Partitioning

- If the input data to the program is specified by a range of values:
  - e.g. numbers between 1 to 5000.
  - One valid and two invalid equivalence classes are defined.

Equivalence Class Partitioning

- If input is an enumerated set of values:
  - e.g. {a,b,c}
  - One equivalence class for valid input values.
  - Another equivalence class for invalid input values should be defined.

Example

- A program reads an input value in the range of 1 and 5000:
  - Computes the square root of the input number

Example (cont.)

- There are three equivalence classes:
  - The set of negative integers,
  - Set of integers in the range of 1 and 5000,
  - Integers larger than 5000.

Example (cont.)

- The test suite must include:
  - Representatives from each of the three equivalence classes:
  - A possible test suite can be: {-5,500,6000}.

Boundary Value Analysis

- Some typical programming errors occur:
  - At boundaries of equivalence classes
  - Might be purely due to psychological factors.
- Programmers often fail to see:
  - Special processing required at the boundaries of equivalence classes.

Boundary Value Analysis

- Programmers may improperly use < instead of <=
- Boundary value analysis:
  - Select test cases at the boundaries of different equivalence classes.

Example

- For a function that computes the square root of an integer in the range of 1 and 5000:
  - Test cases must include the values: {0,1,5000,5001}.

White-Box Testing

- There exist several popular white-box testing methodologies:
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Condition coverage
  - Mutation testing
  - Data flow-based testing

Coverage-Based Testing Versus Fault-Based Testing

- Idea behind coverage-based testing:
  - Design test cases so that certain program elements are executed (or covered).
  - Example: statement coverage, path coverage, etc.
- Idea behind fault-based testing:
  - Design test cases that focus on discovering certain types of faults.
  - Example: Mutation testing.

Statement Coverage

- Statement coverage methodology:
  - Design test cases so that every statement in the program is executed at least once.

Statement Coverage

- The principal idea:
  - Unless a statement is executed,
  - We have no way of knowing if an error exists in that statement.

Statement Coverage Criterion

- Observing that a statement behaves properly for one input value:
  - No guarantee that it will behave correctly for all input values.

Branch Coverage

- Test cases are designed such that:
  - Different branch conditions
    - Given true and false values in turn.

Branch Coverage

- Branch testing guarantees statement coverage:
  - A stronger testing compared to the statement coverage-based testing.

Stronger Testing

- Test cases are a superset of a weaker testing:
  - A stronger testing covers at least all the elements of the elements covered by a weaker testing.

Condition Coverage

- Test cases are designed such that:
  - Each component of a composite conditional expression
    - Given both true and false values.

Example

- Consider the conditional expression
  - ((c1.and.c2).or.c3):
- Each of c1, c2, and c3 are exercised at least once,
  - i.e. given true and false values.

Branch Testing

- Branch testing is the simplest condition testing strategy:
  - Compound conditions appearing in different branch statements
    - Are given true and false values.

Branch testing

- Condition testing
  - Stronger testing than branch testing.
- Branch testing
  - Stronger than statement coverage testing.

Condition coverage

- Consider a boolean expression having n components:
  - For condition coverage we require 2^n test cases.
- Condition coverage-based testing technique:
  - Practical only if n (the number of component conditions) is small.

Path Coverage

- Design test cases such that:
  - All linearly independent paths in the program are executed at least once.
- Defined in terms of
  - Control flow graph (CFG) of a program.

Path Coverage-Based Testing

- To understand the path coverage-based testing:
  - we need to learn how to draw control flow graph of a program.
- A control flow graph (CFG) describes:
  - the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

How to Draw Control Flow Graph?

- Number all the statements of a program.
- Numbered statements:
  - Represent nodes of the control flow graph.
- An edge from one node to another node exists:
  - If execution of the statement representing the first node
    - Can result in transfer of control to the other node.

Example

- int f1(int x,int y){
- 1 while (x != y){
- 2 if (x>y) then
- 3 x=x-y;
- 4 else y=y-x;
- 5 }
- 6 return x; }

Example Control Flow Graph

How to draw Control flow graph?

- Sequence:
  - 1 a=5;
  - 2 b=a*b-1;

How to draw Control flow graph?

- Selection:
  - 1 if(a>b) then
  - 2 c=3;
  - 3 else c=5;
  - 4 c=c*c;

How to draw Control flow graph?

- Iteration:
  - 1 while(a>b){
  - 2 b=b*a;
  - 3 b=b-1;}
  - 4 c=b+d;

Path

- A path through a program:
  - A node and edge sequence from the starting node to a terminal node of the control flow graph.
  - There may be several terminal nodes for program.

Linearly Independent Path

- Any path through the program:
  - Introducing at least one new edge:
    - That is not included in any other independent paths.

Independent path

- It is straight forward:
  - To identify linearly independent paths of simple programs.
- For complicated programs:
  - It is not so easy to determine the number of independent paths.

McCabe's Cyclomatic Metric

- An upper bound:
  - For the number of linearly independent paths of a program
- Provides a practical way of determining:
  - The maximum number of linearly independent paths in a program.

McCabe's Cyclomatic Metric

- Given a control flow graph G, cyclomatic complexity V(G):
  - V(G)= E-N+2
    - N is the number of nodes in G
    - E is the number of edges in G

Example Control Flow Graph

Cyclomatic complexity = 7-6+2 = 3.

Cyclomatic Complexity

- Another way of computing cyclomatic complexity:
  - inspect control flow graph
  - determine number of bounded areas in the graph
- V(G) = Total number of bounded areas + 1
  - Any region enclosed by a nodes and edge sequence.

Example Control Flow Graph

Example

- From a visual examination of the CFG:
  - the number of bounded areas is 2.
  - cyclomatic complexity = 2+1=3.

Cyclomatic complexity

- McCabe's metric provides:
  - A quantitative measure of testing difficulty and the ultimate reliability
- Intuitively,
  - Number of bounded areas increases with the number of decision nodes and loops.

Cyclomatic Complexity

- The first method of computing V(G) is amenable to automation:
  - You can write a program which determines the number of nodes and edges of a graph
  - Applies the formula to find V(G).

Cyclomatic complexity

- The cyclomatic complexity of a program provides:
  - A lower bound on the number of test cases to be designed
  - To guarantee coverage of all linearly independent paths.

Cyclomatic Complexity

- Defines the number of independent paths in a program.
- Provides a lower bound:
  - for the number of test cases for path coverage.

Cyclomatic Complexity

- Knowing the number of test cases required:
  - Does not make it any easier to derive the test cases,
  - Only gives an indication of the minimum number of test cases required.

Path Testing

- The tester proposes:
  - An initial set of test data using his experience and judgement.
- A dynamic program analyzer is used:
  - To indicate which parts of the program have been tested
  - The output of the dynamic analysis
    - used to guide the tester in selecting additional test cases.

Derivation of Test Cases

- Let us discuss the steps:
  - to derive path coverage-based test cases of a program.
- Draw control flow graph.
- Determine V(G).
- Determine the set of linearly independent paths.
- Prepare test cases:
  - to force execution along each path.

Debugging

- Once errors are identified:
  - It is necessary identify the precise location of the errors and to fix them.
- Each debugging approach has its own advantages and disadvantages:
  - Each is useful in appropriate circumstances.

Brute-Force method

- This is the most common method of debugging:
  - Least efficient method.
  - Program is loaded with print statements
  - Print the intermediate values
  - Hope that some of printed values will help identify the error.

Symbolic Debugger

- Brute force approach becomes more systematic With the use of a symbolic debugger,

Symbolic Debugger

- Using a symbolic debugger:
  - Values of different variables can be easily checked and modified
  - Single stepping to execute one instruction at a time
  - Break points and watch points can be set to test the values of variables.

Backtracking

- This is a fairly common approach.
- Beginning at the statement where an error symptom has been observed:
  - Source code is traced backwards until the error is discovered.

Backtracking

- Unfortunately, as the number of source lines to be traced back increases,
  - the number of potential backward paths increases
  - becomes unmanageably large for complex programs.

Cause-elimination method

- Determine a list of causes:
  - which could possibly have contributed to the error symptom.
  - tests are conducted to eliminate each.

Program Slicing

- This technique is similar to back tracking.
- However, the search space is reduced by defining slices.
- A slice is defined for a particular variable at a particular statement:
  - set of source lines preceding this statement which can influence the value of the variable.

Example

```
int main(){
 int i,s;
 i=1; s=1;
 while(i<=10){
        s=s+i;
        i++;}
printf("%d",s);
printf("%d",i);
}
```

Debugging Guidelines

- Be aware of the possibility:
  - an error correction may introduce new errors.
- After every round of error-fixing:
  - regression testing must be carried out.

Levels of Testing

- Software products are tested at three levels:
  - Unit testing
  - Integration testing
  - System testing

Unit testing

- During unit testing, modules are tested in isolation:
  - If all modules were to be tested together:
    - it may not be easy to determine which module has the error.

Unit testing

- Unit testing reduces debugging effort several folds.
  - Programmers carry out unit testing immediately after they complete the coding of a module.

Integration testing

- After different modules of a system have been coded and unit tested:
  - modules are integrated in steps according to an integration plan
  - partially integrated system is tested at each integration step.

System Testing

- System testing involves:
  - validating a fully developed system against its requirements.

Integration Testing

- Develop the integration plan by examining the structure chart :
  - big bang approach
  - top-down approach
  - bottom-up approach
  - mixed approach

Example Structured Design

Big Bang Integration Testing

- Big bang approach is the simplest integration testing approach:
  - all the modules are simply put together and tested.
  - this technique is used only for very small systems.

Big Bang Integration Testing

- Main problems with this approach:
  - If an error is found:
    - It is very difficult to localize the error
    - The error may potentially belong to any of the modules being integrated.
  - Debugging errors found during big bang integration testing are very expensive to fix.

Bottom-up Integration Testing

- Integrate and test the bottom level modules first.
- A disadvantage of bottom-up testing:
  - when the system is made up of a large number of small subsystems.
  - This extreme case corresponds to the big bang approach.

Top-down integration testing

- Top-down integration testing starts with the main routine:
  - and one or two subordinate routines in the system.
- After the top-level 'skeleton’ has been tested:
  - immediate subordinate modules of the 'skeleton’ are combined with it and tested.

Mixed Integration Testing

- Mixed (or sandwiched) integration testing:
  - uses both top-down and bottom-up testing approaches.
  - Most common approach

Integration Testing

- In top-down approach:
  - testing waits till all top-level modules are coded and unit tested.
- In bottom-up approach:
  - testing can start only after bottom level modules are ready.

Phased versus Incremental Integration Testing

- Integration can be incremental or phased.
- In incremental integration testing,
  - only one new module is added to the partial system each time.

Phased versus Incremental Integration Testing

- In phased integration,
  - A group of related modules are added to the partially integrated system each time.
- Big-bang testing:
  - A degenerate case of the phased integration testing.

Phased versus Incremental Integration Testing

- Phased integration requires less number of integration steps:
  - compared to the incremental integration approach.
- However, when failures are detected,
  - it is easier to debug if using incremental testing
    - since errors are very likely to be in the newly integrated module.

System Testing

- There are three main kinds of system testing:
  - Alpha Testing
  - Beta Testing
  - Acceptance Testing

Alpha Testing

- System testing is carried out by the test team within the developing organization.

Beta Testing

- System testing performed by a select group of friendly customers.

Acceptance Testing

- System testing performed by the customer himself:
  - to determine whether the system should be accepted or rejected.

System Testing

- During system testing:
  - Functional requirements are validated through functional tests.
  - Non-functional requirements validated through performance tests.

Performance Testing

- Addresses non-functional requirements.
  - May sometimes involve testing hardware and software together.
  - There are several categories of performance testing.

Stress Testing

- Stress testing (aka endurance testing):
  - impose abnormal input to stress the capabilities of the software.
  - Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity.

Stress testing

- Evaluates system performance
  - when stressed for short periods of time.
- Stress testing
  - also known as endurance testing.

Stress testing

- Stress tests are black box tests:
  - Designed to impose a range of abnormal and even illegal input conditions
  - So as to stress the capabilities of the software.

Stress Testing

- If the requirements is to handle a specified number of users, or devices:
  - Stress testing evaluates system performance when all users or devices are busy simultaneously.

Stress Testing

- If an operating system is supposed to support 15 multiprogrammed jobs,
  - The system is stressed by attempting to run 15 or more jobs simultaneously.
- A real-time system might be tested
  - To determine the effect of simultaneous arrival of several high-priority interrupts.

Stress Testing

- Stress testing usually involves an element of time or size,
  - Such as the number of records transferred per unit time,
  - The maximum number of users active at any time, input data size, etc.
- Therefore stress testing may not be applicable to many types of systems.

Volume Testing

- Addresses handling large amounts of data in the system:
  - Whether data structures (e.g. queues, stacks, arrays, etc.) are large enough to handle all possible situations.
  - Fields, records, and files are stressed to check if their size can accommodate all possible data volumes.

Configuration Testing

- Analyze system behavior:
  - in various hardware and software configurations specified in the requirements
  - sometimes systems are built in various configurations for different users
  - for instance, a minimal system may serve a single user,
    - other configurations for additional users.

Compatibility Testing

- These tests are needed when the system interfaces with other systems:
  - Check whether the interface functions as required.

Compatibility testing Example

- If a system is to communicate with a large database system to retrieve information:
  - A compatibility test examines speed and accuracy of retrieval.

Recovery Testing

- These tests check response to:
  - Presence of faults or to the loss of data, power, devices, or services
  - Subject system to loss of resources
    - Check if the system recovers properly.

Maintenance Testing

- Diagnostic tools and procedures:
  - help find source of problems.
  - It may be required to supply
    - memory maps
    - diagnostic programs
    - traces of transactions,
    - circuit diagrams, etc.

Maintenance Testing

- Verify that:
  - all required artifacts for maintenance exist
  - they function properly

Documentation tests

- Check that required documents exist and are consistent:
  - user guides,
  - maintenance guides,
  - technical documents

Documentation tests

- Sometimes requirements specify:
  - Format and audience of specific documents
  - Documents are evaluated for compliance

Usability tests

- All aspects of user interfaces are tested:
  - Display screens
  - messages
  - report formats
  - navigation and selection problems

Environmental test

- These tests check the system's ability to perform at the installation site.
- Requirements might include tolerance for
  - heat
  - humidity
  - chemical presence
  - portability
  - electrical or magnetic fields
  - disruption of power, etc.

Regression Testing

- Does not belong to either unit test, integration test, or system test.
  - In stead, it is a separate dimension to these three forms of testing.

Regression testing

- Regression testing is the running of test suite:
  - after each change to the system after each bug fix.
  - Ensures that no new bug has been introduced due to the change or the bug fix.

Regression testing

- Regression tests assure:
  - the new system's performance is at least as good as the old system.
  - Always used during incremental system development.

Data Flow-Based Testing

- Selects test paths of a program:
  - According to the locations of
    - Definitions and uses of different variables in a program.

Data Flow-Based Testing

- For a statement numbered S,
  - DEF(S) = {X/statement S contains a definition of X}
  - USES(S)= {X/statement S contains a use of X}
  - Example: 1: a=b; DEF(1)={a}, USES(1)={b}.
  - Example: 2: a=a+b; DEF(1)={a}, USES(1)={a,b}.

Data Flow-Based Testing

- A variable X is said to be live at statement S1, if
  - X is defined at a statement S:
  - There exists a path from S to S1 not containing any definition of X.

DU Chain Example

```
1 X(){
2  a=5; /* Defines variable a */
3  While(C1) {
4     if (C2)
5           b=a*a;   /*Uses variable a */
6           a=a-1; /* Defines variable a */
7     }
8   print(a); } /*Uses variable a */
```

Definition-use chain (DU chain)

- [X,S,S1],
  - S and S1 are statement numbers,
  - X in DEF(S)
  - X in USES(S1), and
  - the definition of X in the statement S is live at statement S1.

Data Flow-Based Testing

- One simple data flow testing strategy:
  - Every DU chain in a program be covered at least once.
- Data flow testing strategies:
  - Useful for selecting test paths of a program containing nested if and loop statements.

Data Flow-Based Testing

```
• 1 X(){
• 2  B1;        /* Defines variable a */
• 3  While(C1) {
• 4      if (C2)
• 5          if(C4) B4; /*Uses variable a */
• 6          else B5;
• 7      else  if (C3) B2;
• 8      else B3;     }
• 9  B6 }
```

Data Flow-Based Testing

- [a,1,5]: a DU chain.
- Assume:
  - DEF(X) = {B1, B2, B3, B4, B5}
  - USED(X) = {B2, B3, B4, B5, B6}
  - There are 25 DU chains.
- However only 5 paths are needed to cover these chains.

Mutation Testing

- The software is first tested:
  - using an initial testing method based on white-box strategies we already discussed.
- After the initial testing is complete,
  - mutation testing is taken up.
- The idea behind mutation testing:
  - make a few arbitrary small changes to a program at a time.

Mutation Testing

- Each time the program is changed,
  - it is called a mutated program
  - the change is called a mutant.

Mutation Testing

- A mutated program:
  - tested against the full test suite of the program.
- If there exists at least one test case in the test suite for which:
  - a mutant gives an incorrect result,
  - then the mutant is said to be dead.

Mutation Testing

- If a mutant remains alive:
  - even after all test cases have been exhausted,
  - the test suite is enhanced to kill the mutant.
- The process of generation and killing of mutants:
  - can be automated by predefining a set of primitive changes that can be applied to the program.

Mutation Testing

- The primitive changes can be:
  - altering an arithmetic operator,
  - changing the value of a constant,
  - changing a data type, etc.

Mutation Testing

- A major disadvantage of mutation testing:
  - computationally very expensive,
  - a large number of possible mutants can be generated.

Test Summary Report

- Generated towards the end of testing phase.
- Covers each subsystem:
  - A summary of tests which have been applied to the subsystem.

Test Summary Report

- Specifies:
  - how many tests have been applied to a subsystem,
  - how many tests have been successful,
  - how many have been unsuccessful, and the degree to which they have been unsuccessful,
    - e.g. whether a test was an outright failure
    - or whether some expected results of the test were actually observed.

How Many Errors are Still Remaining?

- Seed the code with some known errors:
  - artificial errors are introduced into the program.
  - Check how many of the seeded errors are detected during testing.

Error Seeding

- Let:
  - N be the total number of errors in the system
  - n of these errors be found by testing.
  - S be the total number of seeded errors,
  - s of the seeded errors be found during testing.

Error Seeding

- $n/N = s/S$
- $N = S\ n/s$
- remaining defects:

  $N - n = n\ ((S - s)/ s)$

Example

- 100 errors were introduced.
- 90 of these errors were found during testing
- 50 other errors were also found.
- Remaining errors=
  50 (100-90)/90 = 6

Error Seeding

- The kind of seeded errors should match closely with existing errors:
  - However, it is difficult to predict the types of errors that exist.
- Categories of remaining errors:
  - can be estimated by analyzing historical data from similar projects.

Maitenance

Introduction

- Software maintenance:
  - any modifications to a software product after it has been delivered to the customer.
- Software maintenance is an important activity for many organizations.

Introduction

- Maintenance is inevitable for almost any kind of product.
- Most products need maintenance:
  - due to wear and tear caused by use.
- Software products do not need maintenance on this count.

Introduction

- Many people think
  - only bad software products need maintenance.
- The opposite is true:
  - bad products are thrown away,
  - good products are maintained and used for a long time.

Introduction

- Software products need maintenance for three reasons:
  - corrective
  - adaptive
  - perfective

Corrective

- Corrective maintenance of a software product:
  - to correct bugs observed while the system is in use.
  - to enhance performance of the product.

Adaptive

- A software product needs maintenance (porting) when customers:
  - need the product to run on new platforms,
    - or, on new operating systems,
  - need the product to interface with new hardware or software.

Perfective

- Perfective maintenance:
  - to support new features required by users.
  - to change some functionality of the system due to customer demands.

Maintenance Effort Distribution

Causes for maintenance

- During development:
  - Software not anticipated to last very long (e.g., Y2K problem).
- Rate of hardware obsolescence:
  - immortality of software products
  - maintenance necessary for software performing low-level functions.

Causes for maintenance

- Users want existing software to run on new platforms:
  - to run in new environments,
  - and/or with enhanced features.

Causes for maintenance

- Whenever other software it works with change:
  - maintenance is needed to cope up with the newer interface.
  - For instance, a software product may need maintenance when the operating system changes.

Software evolution

- Every software product continues to evolve after its development:
  - through maintenance efforts.
- Larger software products stay in operation for longer time:
  - because of high replacement cost.

Laws of Maintenance

- There will always be a lot of old software needing maintenance.
- Good products are maintained, bad products are thrown away.

Laws of Maintenance

- Lehman's first Law:
- "Software products must change continuously, or become progressively less useful."

Laws of Maintenance

- Lehman's Second Law
- "When software is maintained, its structure degrades,
  - unless active efforts are made to avoid this phenomenon."

Laws of Maintenance

- Lehman's Third Law:
- "Over a program's life time,
  - its rate of development is approximately constant."

Other Laws of Maintenance

- All large programs will undergo significant changes during operation phase of their life cycle,
  - regardless of *apriori* intentions.

Legacy code--- Major maintenance problems

- Unstructured code (bad programs)
- Maintenance programmers have:
  - insufficient knowledge of the system or the application domain.
  - Software maintenance has a bad image.
- Documentation absent, out of date, or insufficient.

Insufficient knowledge

- Maintenance team is usually different from development team.
  - even after reading all documents
    - it is very difficult to understand why a thing was done in a certain way.
  - Also there is a limit to the rate at which a person can study documents
    - and extract relevant information

Bad image of maintenance?

- Maintainers are skilled not only in writing code:
  - proficient in understanding others' code
  - detecting problems, modifying the design, code, and documentation
  - working with end-users

Maintenance Nightmares

- Use of gotos
- Lengthy procedures
- Poor and inconsistent naming
- Poor module structure
- Weak cohesion and high coupling
- Deeply nested conditional statements
- Functions having side effects

How to do better maintenance?

- Program understanding
- Reverse engineering
- Design recovery
- Reengineering
- Maintenance process models

Maintenance activities

- Two types of activities:
  - Productive activities:
    - modification of analysis, design, coding, etc.
  - Non-productive activities:
    - understanding system design, code, etc.

Software Reverse Engineering

- By analyzing a program code, recover from it:
  - the design and the requirements specification.

Software Reverse Engineering

- Reverse engineering is an important maintenance technique:
  - several existing software products are unstructured,
  - lack proper documentation,
  - were not developed using software engineering principles.

Software Reverse Engineering

- First carry out cosmetic changes to the code to improve:
  - readability,
  - structure,
  - understandability.

Cosmetic changes

Cosmetic Changes

- Reformat the program:
  - use any pretty printer program
  - layout the program neatly.
- Give more meaningful names to:
  - Variables, data structures, and functions.

Cosmetic Changes

- Replace complex and nested conditional expressions:
  - simpler conditional statements
  - whenever appropriate use case statements.

Software Reverse Engineering

- In order to extract the design:
  - fully understand the code.
- Automatic tools can be used to help derive:
  - data flow and control flow diagrams from the code.

Software Reverse Engineering

- Extract structure chart:
  - module invocation sequence and data interchange among modules.
- Extract requirements specification:
  - after thoroughly understanding the code.
  - design has been extracted.

Software Maintenance Process Models

- Maintenance activities are not unique:
  - depend on the extent of modifications required,
  - also, depend on condition of the product:
    - how structured it is,
    - how well documented it is, etc.

Software Maintenance Process Model - 1

- When the required changes are small and simple:
  - the code can be directly modified
  - changes reflected in all relevant documents.
  - more elaborate activities are required when required changes are not trivial.

Software Maintenance Process Model - 1

Software Maintenance Process Model 1

- Start by gathering change requirements.
- Analyze change requirements
  - formulate strategies for code change.

Software Maintenance Process Model 1

- Formulating strategies for code change:
  - presence of few members of the original development team
    - helps in reducing cycle time,
    - especially for unstructured and inadequately documented code.

Software Maintenance Process Model 1

- Availability of a working old system at the maintenance site:
  - greatly helps the maintenance team
  - provides a good insight into the working of the old system
  - can compare the working of the modified system with the old system.

Software Maintenance Process Model 1

- Debugging the system under maintenance becomes easier:
  - program traces of both the systems can be compared to localize bugs.

Software Maintenance Process Model -2

- For complex maintenance projects, software reengineering needed:
  - a reverse engineering cycle followed by a forward engineering cycle.
  - with as much reuse as possible from existing code and other documents.

Maintenance Process Model 2

- Preferable when:
  - amount of rework is significant
  - software has poor structure.
- Can be represented by a reverse engineering cycle:
  - followed by a forward engineering cycle.

Software reengineering

- Many aging software products belong to this category.
- During the reverse engineering,
  - the old code is analyzed (abstracted) to extract the module specifications.

Software reengineering

- The module specifications are analyzed
  - to produce the design.
- The design is analyzed (abstracted)
  - to produce the original requirements specification.
- The change requests are then applied to the requirements specification:
  - arrive at the new requirements specification.

Software reengineering

- Forward engineering is carried out to produce the new code.
- During design, module specification, and coding:
  - substantial reuse is made from the reverse engineered products.

Process model for Software reengineering

Software reengineering

- Advantages of reengineering:
  - produces better design than the original product,
  - produces required documents,
  - often results in higher efficiency.

Software reengineering

- Efficiency improvements are brought about by better design.
  - However, this approach is more costly than the first approach.
- An empirical study indicates:
  - process 1 is preferable when amount of rework is no more than 15%.

Software reengineering

- Reengineering is preferable when:
  - amount of rework is high,
  - product exhibits high failure rate.
  - product difficult to understand.