BCA

CORE - 5

SOFTWARE ENGINEERING

BHARATHIAR UNIVERSITY
SCHOOL OF DISTANCE EDUCATION
COIMBATORE

CORE - 5 SOFTWARE ENGINEERING

Subject Description: This subject deals with Software Engineering concepts like Analysis, Design, Implementation, Testing and Maintenance.

Goal: Knowledge on how to do a software project with in-depth analysis.

Objective: To inculcate knowledge on Software engineering concepts in turn gives a roadmap to design a new software project.

- - - - -

UNIT-I : Introduction to Software Engineering: Definitions – Size Factors – Quality and Productivity Factors. **Planning a Software Project:** Planning the Development Process – Planning an Organizational Structure.

UNIT-II: Software Cost Estimation: Software cost Factors – Software Cost Estimation Techniques – Staffing-Level Estimation – Estimating Software Estimation Costs.

UNIT-III: Software Requirements Definition: The Software Requirements specification – Formal Specification Techniques. **Software Design:** Fundamental Design Concepts – Modules and Modularization Criteria.

UNIT-IV: Design Notations – Design Techniques. **Implementation Issues**: Structured Coding Techniques – Coding Style – Standards and Guidelines – Documentation Guidelines.

UNIT-V: Verification and Validation Techniques: Quality Assurance – Walkthroughs and Inspections – Unit Testing and Debugging – System Testing. **Software Maintenance:** Enhancing Maintainability during Development – Managerial Aspects of Software Maintenance – Configuration Management.

TEXTBOOK:

1. **SOFTWARE ENGINEERING CONCEPTS – Richard Fairley,** 1997, TMH.

(UNIT-I: 1.1-1.3,2.3-2.4 UNIT-II: 3.1-3.4 UNIT III: 4.1-4.2,5.1-5.2

UNIT-IV: 5.3-5.4,6.1-6.4 UNIT-V: 8.1-8.2, 8.5-8.6, 9.1-9.3)

REFERENCE BOOKS:

1. Software Engineering for Internet Applications – Eve Anderson, Philip Greenspun, Andrew Grumet, 2006, PHI.

2. Fundamentals of SOFTWARE ENGINEERING – Rajib Mall, 2nd edition, PHI

3. SOFTWARE ENGINEERING – Stephen Schach, 7th edition ,TMH.

CONTENTS

Lesson 1: Introduction to Software Engineering

Contents

1.0. AIMS AND OBJECTIVES

- To understand the meaning of software engineering
- To understand the principles of Software Engineering
- To understand the quality and productivity factors of SE
- To understand the various managerial issues in SE

1.1. INTRODUCTION

Computers find applications in a wide variety of fields ranging from rail ticket reservation systems to medical diagnosis systems. Hence it is necessary for any computer to work on a wide spectrum of software.

Software is not merely a collection of computer programs. There is a thin difference between software and program. A *program* consists of a set of instructions that a computer follows to perform a task and is complete in it. It is generally used only by the author of the program and has little or no documentation. It is difficult for others to use the program, modify it, or debug it. An example may be a program in C language to arrange a list of names in ascending order.

On the other hand, software is a collection of programs and is developed for widespread use. Hence it is expected to be very user friendly. It contains comprehensive documentation, which enables it to be used by people other than the developers also. The documentation also enables customization, i.e., modification of the software to suit specific applications by third party programmers.

In the initial stages of software development, the software was unmanaged and little attention was paid to systematic methods for its creation, until delays and costs began to go up. This necessitated the need for a structured approach to programming, laying the way for software development within a planned framework.

Software creation has accelerated in the recent past as evidenced by the boom in software employment. Software has been created to automate most aspects of every day life. But the software development has been to large extent independent, raising questions of compatibility, quality, reusability, and maintainability. Schedule and cost increases, poorly defined customer requirements, poor quality of software, little or sometimes no documentation for the software developed are some of the few problems that led to a ***software crisis (disaster)***. These problems have grown exponentially making it almost impossible to manage the large volume of software available. These problems have led to a new branch of study concerned with the development of formal methods to address these problems, called ***Software Engineering***.

1.2. DEFINITIONS

The IEEE glossary on software engineering terminology gives the following definition for software:

> ***Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, that is, the application of engineering to software.***

A systematic approach to the analysis, design, implementation and maintenance of software. It often involves the use of CASE tools. There are various models of the software life cycle and many methodologies for the different phases.

It will be clear that the term *software engineering* involves many difference issues - all are important to the success of large-scale software development. Many authors define Software Engineering as:

Boehm [Boehm 1979]:

> Software Engineering: The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them.

Dennis [Dennis 1975]:

> Software engineering is the application of principles, skills, and art to the design and construction of programs and systems of programs.

Parnas [Parnas 1974]:

Software engineering is programming under at last one of the following two conditions:

(1) More than one person is involved in the construction and/or use of the programs

(2) More than one version of the program will be produced

Fairley [Fairley 1985]:

Software engineering is the technological and managerial discipline concerned with the systematic production and maintenance of software products that are developed and modified on time and within cost estimates.

Sommerville [Sommerville 1989]:

Software Engineering is concerned with building software systems which are large than would normally be tackled by a single individual, uses engineering principles in the development of these systems and is made up of both technical and non-technical aspects.

Pomberger and Blaschek [Pomberger 1996]:

Software engineering is the practical application of scientific knowledge for the economical production and use of high-quality software.

1.2.1. Principle of Software Engineering

The basic principle of software Engineering is to use structured, formal and disciplined methods for building and using systems, just as in any other branch of engineering. Software Engineering is an attempt at providing the same, a set of methods, a variety of tools, and a collection of procedures.

- **Methods** provide the rules and steps for carrying out tasks. The tasks are project-planning and estimation, system and software requirements analysis, design of processes and data structures, algorithm and program architecture, coding, testing and maintenance.
- **Tools** provide automated or semi-automated support for methods. Tools which automate the range of Software Engineering methods can be integrated into one system called CASE (Computer Aided Software Engineering) tools. CASE tools are available to support and automate most of the software engineering methods.
- **Standard procedures** bind the methods and tools into a framework. They define the sequence in which methods will be applied and the deliverables (documents, forms, reports, etc.) that are required. These specify the controls that ensure quality, coordinate change, and the milestones that help a manager assess progress.

Software Engineering provides certain models that encompass the above methods, tools and procedures. The paradigm (theories and methodology) may be viewed as models of software development.

Check your progress

I State whether true of false:

1. There is no difference between software engineering and programming.
2. Software engineering differs from classical engineering disciplines to some extent.

II Choose the correct answer.

The concept of software engineering was born because:

a) People felt the need for more programmers.
b) Most software projects were over budget and behind schedule
c) Most development time was spent on rework.
d) B and C

1.3. SIZE FACTORS

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the "size" of the software that has been produced.

Examples:

- ***Direct measures*** - Errors per KLOC, Defects per KLOC, Cost per KLOC, pages of documentation per KLOC.
- ***Indirect measures*** – Errors/person-month, LOC per person-month, Cost/page of documentation

Advantages:

These measures are easy to use.

Disadvantage:

LOC is not universally accepted as a key measure because it is language dependant and programmer dependant.

1.4. QUALITY AND PRODUCTIVITY FACTORS

The overriding goal of software engineering is to produce a high quality system, application, or product. To achieve this goal, software engineers must do the following:

1. Apply effective methods coupled with modern tools within the context of a mature software process.
2. Measure in order to assess the quality of the process as well the product.

Measuring Quality

Although there are many measures of software quality, correctness, maintainability, integrity, and usability provide useful indicators for the project team.

a. **Correctness:** Correctness is the degree to which the software performs its required function. Defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.

b. **Maintainability:** Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. There is no direct measure for software maintenance. Hence indirect measures have to be used. A simple time-oriented metric known as a Mean-Time-To-Change (MTTC) is the time it takes to analyze a change request, design an appropriate modification, implement the change, test it, and distribute the change to all the users.

c. **Integrity:** Integrity measures a system's ability to withstand attacks on its security. Attacks can be made on programs, data, and documents. To measure integrity, two additional attributes must be defined: threat and security. Threat is the probability that an attack of a specific type will occur within a given time. Security is the probability that the attack of a specific type will be repelled. The integrity of a system can be defined as:

$$\text{Integrity} = \sum [1 - \text{threat} \times (1 - \text{security})]$$

d. **Usability:** Usability is an attempt to quantify "user friendliness" and can be measured in terms of four characteristics: (1) the physical and/or intellectual skill required to learn the system; (2) the time required to become moderately efficient in the use of the system; (3) the net increase in productivity measured when the system is used by someone who is moderately efficient, and (4) a subjective assessment of users attitude towards the system.

Defect Removal Efficiency (DRE)

A quality metric that provides benefit at both the project and process level is Defect Removal Efficiency (DRE). DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities.

$$DRE = E / (E + D)$$

Where E = number of errors found before delivery of the software to the end user,

D = number of defects found after delivery

The ideal value for DRE is 1. That is, no defects are found in the software. Realistically, D will be greater than zero, but the value of DRE can still approach 1 as E increases. In fact, as E increases, it is likely that the final value of D will decrease. On the whole, DRE encourages a software project team to institute techniques for finding as many errors as possible before delivery. DRE can also be used within the project to assess a team's ability to find errors before they are passed to the next software engineering task. This is because errors that are not found during a phase (e.g. Requirements Analysis) are passed onto the subsequent stage (e.g. Design). When used in this context, DRE can be redefined as:

$$DRE_i = E_i / (E_i + E_{i+1})$$

where,

E_i = number of errors found during software engineering activity i

E_{i+1} = number of errors found during software engineering activity i + 1 that are traceable to errors that were not discovered in software engineering activity i.

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

Measurement of software quality factors

Software Quality Factors cannot be measured because of their unclear description. It is necessary to find measures, or metrics, which can be used to quantify them as non-functional requirements

For example, reliability is a software quality factor, but cannot be evaluated in its own right. However there are related attributes to reliability, which can indeed be measured. Such attributes are mean time to failure, rate of failure occurrence, availability of the system. Similarly, an attribute of portability is the number of target dependent statements in a program.

A scheme which could be used for evaluating software quality factors is given below. For every characteristic, there are a set of questions which are relevant to that characteristic. Some type of scoring formula could be developed based on the answers to these questions, from which a measure of the characteristic may be obtained.

- **Understandibility:** Are variable names descriptive of the physical or functional property represented? Do uniquely recognisable functions contain adequate comments so that their purpose is clear? Are deviations from forward logical flow adequately commented? Are all elements of an array functionally related?

- **Completeness:** Does the program contain all referenced subprograms not available in the usual systems library? Are all parameters required by the program available? Are all inputs required by the program available?
- **Conciseness (short):** Is all code reachable? Is any code redundant? How many statements within loops could be placed outside the loop, thus reducing computation time? Are branch decisions too complex?
- **Portability (capable of being transferred*)*:** Does the program depend upon system or library routines unique to a particular installation? Have machine-dependent statements been flagged and commented? Has dependency on internal bit representation of alphanumeric or special characters been avoided?
- **Consistency (stability):** Is one variable name used to represent difficult physical entities in the program? Does the program contain only one representation for physical or mathematical constants? Are functionally similar arithmetic expressions similarly constructed? Is a consistent scheme for indentation used?
- **Maintainability:** Has some memory capacity been reserved for future expansion? Is the design cohesive, i.e., each module has recognizable functionality? Does the software allow for a change in data structures (object-oriented designs are more likely to allow for this)? If a functionally-based design (rather than object-oriented), is a change likely to require restructuring the main-program, or just a module?
- **Testability:** Are complex structures employed in the code? Does the detailed design contain clear pseudo-code? Is the pseudo-code at a higher level of abstraction than the code? If tasking is used in concurrent designs, are schemes available for providing adequate test cases?
- **Usability:** Is a GUI used? Is there adequate on-line help? Is a user manual provided? Are meaningful error messages provided?
- **Reliability:** Are loop indexes range tested? Is input data checked for range errors? Is divide-by-zero avoided? Is exception handling provided?
- **Structuredness:** Is a block-structured programming language used? Are modules limited in size? Have the rules for transfer of control between modules been established and followed?
- **Efficiency:** Have functions been optimized for speed? Have repeatedly used blocks of code been formed into sub-routines?

1.5. MANAGERIAL ISSUES

Software engineering is an application of management activities such as:

- Planning,

- Coordinating,
- Ensuring,
- Monitoring,
- Controlling and
- Reporting

to insure that the development of software is systematic, disciplined and measured.

Software Engineering Measurement

1. Goals
    1. Organizational objectives
    2. Software process improvement goals
2. Measurement selection
    1. Goal-driven measurement selection
    2. Measurement validity
3. Measuring software and its development
    1. Size measurement
    2. Structure measurement
    3. Resource measurement
    4. Quality measurement
4. Collection of data
    1. Survey techniques and form design
    2. Automated and manual data collection
5. Software measurement models
    1. Model building, calibration and evaluation
    2. Implementation, interpretation and refinement of models

Software Project Management

- *Communication*:
- *Rationale management*:
    - The problem
    - the alternatives considered
    - the criteria used to evaluate the alternatives

    - the debate
    - the decision
- *Testing*: find differences between the system and its models by executing the system with sample input data sets.
    - Unit testing: object design compared with object and subsystem.
    - Integration testing: combinations of subsystems are integrated and compared with the system design model.
    - System testing: typical and exceptional cases are compared with the requirements model
    - Acceptance testing
- *Software configuration management*: establishes baseline, monitors and controls changes in work products, versions.
- *Project management*: oversight activities that insure the delivery of a high-quality system on time and within budget including
    - planning and budgeting
    - hiring and organizing developers into teams
    - monitoring project status
    - intervening when deviation occur
- Software life cycle modeling activities

Since quantitative methods have proved so powerful in the other sciences, computer science practitioners and theoreticians have worked hard to bring similar approaches to software development

Software metrics are numerical data related to software development. Metrics strongly support software project management activities. They relate to the four functions of management as follows:

1. **Planning** - Metrics serve as a basis of cost estimating, training planning, and resource planning, scheduling, and budgeting.
2. **Organizing** - Size and schedule metrics influence a project's organization.
3. **Controlling** - Metrics are used to status and track software development activities for compliance to plans.
4. **Improving** - Metrics are used as a tool for process improvement and to identify where improvement efforts should be concentrated and measure the effects of process improvement efforts.

A metric quantifies a characteristic of a process or product. Metrics can be directly observable quantities or can be derived from one or more directly

observable quantities. Examples of raw metrics include the number of source lines of code, number of documentation pages, number of staff-hours, number of tests, number of requirements, etc. Examples of derived metrics include source lines of code per staff-hour, defects per thousand lines of code, or a cost performance index.

1.6. REVIEW QUESTIONS

1. What are the different software paradigms available? Can we combine the paradigms? Explain.
2. What are the objectives of Software Engineering?
3. Explain the evolving role of software.
4. Give a generic view of Software Engineering.
5. What is software re-engineering? Explain it.
6. Explain Software crises and software myths.
7. What is software reuse? Explain.
8. Define Software Engineering. List out is components.
9. Discuss the fundamental activities, which are common to all software processes.

1.7. LET US SUM UP

This Lesson covered the following:

- The early unplanned approaches to software development and how their inadequacy in face of increased sophisticated led to the software crises
- The realization of the need for a more structured and organized way of building software, leading to the birth of Software Engineering.
- Definition of Software Engineering
- Software Engineering methods, tools and procedures

1.8. LESSON END ACTIVITIES

i. Interview programmer and managers in the organization of your choice to determine management problem areas.

ii. How do the perceptions of managers and programmers differ?

1.9. POINTS FOR DISCUSSION

i. Which is more important the product or process? Justify your answer

ii. Identify the umbrella activities in software engineering process.

1.10. REFERENCES

1. Richard Fairley, "Software Engineering Concepts", Tata McGraw-Hill, 1997.
2. Roger S.Pressman, Software engineering- A practitioner's Approach, McGraw-Hill International Edition, 5th edition, 2001.

LESSON 2 : SOFTWARE PROJECT PLANNING

Contents

2.0. AIMS AND OBJECTIVES

- To provide a broad outlook of the steps involved in a software project planning
- To understand and define software Engineering Models
- To understand the project planning activities, scope and resources.

2.1. INTRODUCTION

The dictionary definition of Project is: a planned undertaking to present results at a specified time. ***Note***: there the word undertaking means: a. Making a new product, or b. changing the old product.

The software project management process begins with a set of activities that are collectively called project planning (*see Figure 2.1*.). The first of these activities is estimation. Estimation of resources, cost, and schedule for a software development effort requires experience, access to good historical information, and the courage to commit to quantitative measures when only qualitative data exists. Estimation carries inherent risk and risk leads to uncertainty. Hence software project estimation must be viewed as an art and science and must be done as accurately as possible.

2.2. PLANNING A SOFTWARE PROJECT

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. These estimates are made within a limited time frame at the

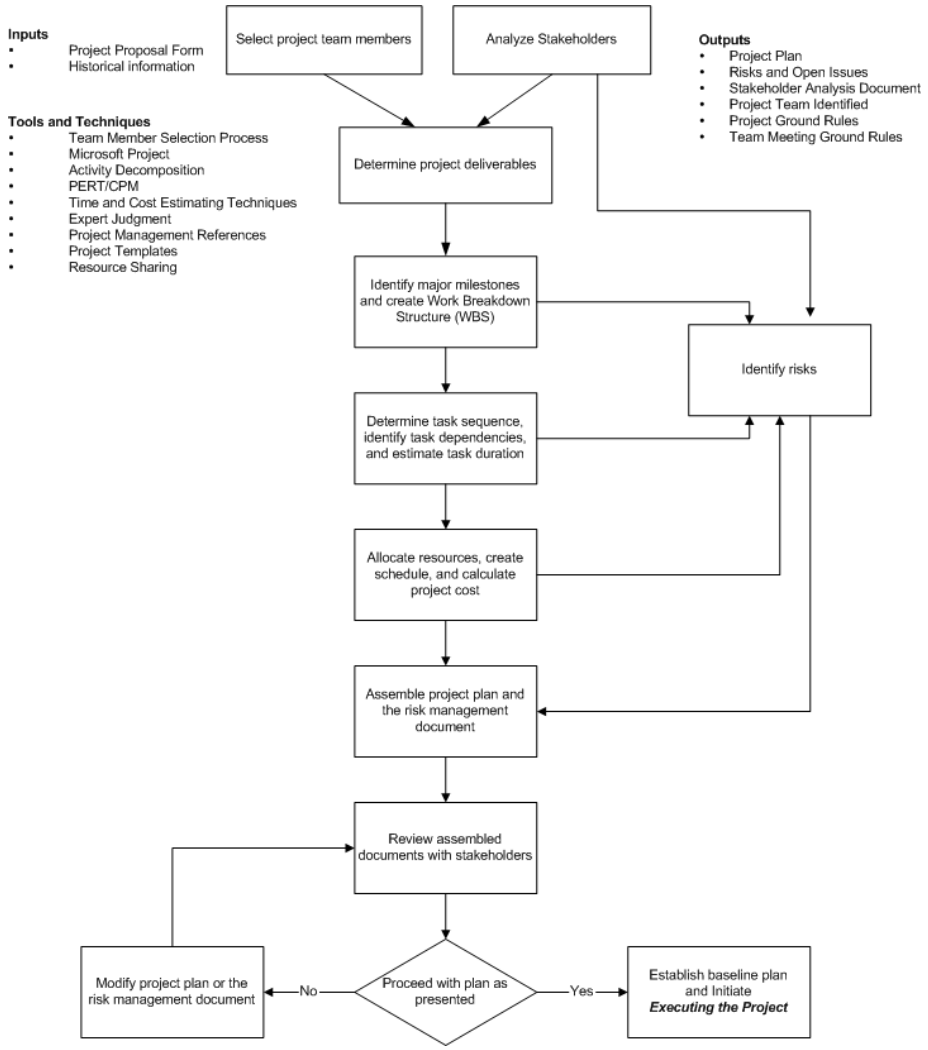


Fig. 2.1. Steps in Software Project Plan

beginning of a software project. Also they should be updated regularly as the project progresses. The planning objective is achieved through a process of information discovery that leads to reasonable estimates.

2.2.1. Software Scope

The first activity in software project planning is the determination of software scope. Software scope describes function, performance, constraints, interfaces, and reliability. Functions are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Both cost and schedule estimates are functionally oriented and hence some degree of decomposition is

often useful. Performance considerations encompass processing and response time requirements. Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.

To define scope, it is necessary to obtain the relevant information and hence get the communication process started between the customer and the developer. To accomplish this, a preliminary meeting or interview is to be conducted. The analyst may start by asking ***context free questions***. That is, a set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of the first encounter itself.

The next set of questions enable the analyst to gain a better understanding of the problem and the customer to voice his or her perceptions about a solution. The final set of questions, known as ***Meta questions***, focus on the effectiveness of the meeting.

A team-oriented approach, such as the **Facilitated Application Specification Techniques (FAST)**, helps to establish the scope of a project.

2.2.2. Resources

The development resources needed are:

1. Development Environment (Hardware/Software Tools)
2. Reusable Software Components
3. Human Resources (People)

Each resource is specified with four characteristics – description of the resource, a statement of availability, chronological time that the resource will be required, and duration of time that the resource will be applied. The last two characteristics can be viewed as a time window. Availability of the resource for a specified window must be established at the earliest practical time.

Human Resources: Both organizational positions (e.g. Manager, senior software engineer, etc.) and specialty (e.g. Telecommunications, database, etc.) are specified. The number of people required varies for every software project and it can be determined only after an estimate of development effort is made.

Reusable Software Resources: These are the software building blocks that can reduce development costs and speed up the product delivery. The four software resource categories that should be considered as planning proceeds are:

1. **Off-the-shelf components –** Existing software that can be acquired from a third-party or that has been developed internally for past project. These are ready for use and have been fully validated. Generally, the cost for acquisition and integration of such components will be less than the cost to develop equivalent software.

2. **Full-experience components –** Existing specifications, designs, code, or test data developed for past projects that are similar to the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore modifications will be relatively low risk.
3. **Partial-experience components -** Existing specifications, designs, code, or test data developed for past projects that are related to the current project, but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore modifications will have a fair degree of low risk and hence their use for the current project must be analyzed in detail.
4. **New components –** Software components that must be built by the software team specifically for the needs of the current project.

Environmental Resources: The environment that supports the software project, often called ***Software Engineering Environment*** (SEE), incorporates hardware and software. Hardware provides a platform that supports the tools required to produce the work products. A project planner must prescribe the time window required for hardware and software and verify that these resources will be available.

The phases of a software project

Software projects are divided into individual phases. These phases collectively and their chronological sequence are termed the *software life cycle (see Fig. 2.2)*.

Software life cycle: a time span in which a software product is developed and used, extending to its retirement.

The cyclical nature of the model expresses the fact that the phases can be carried out repeatedly in the development of a software product.

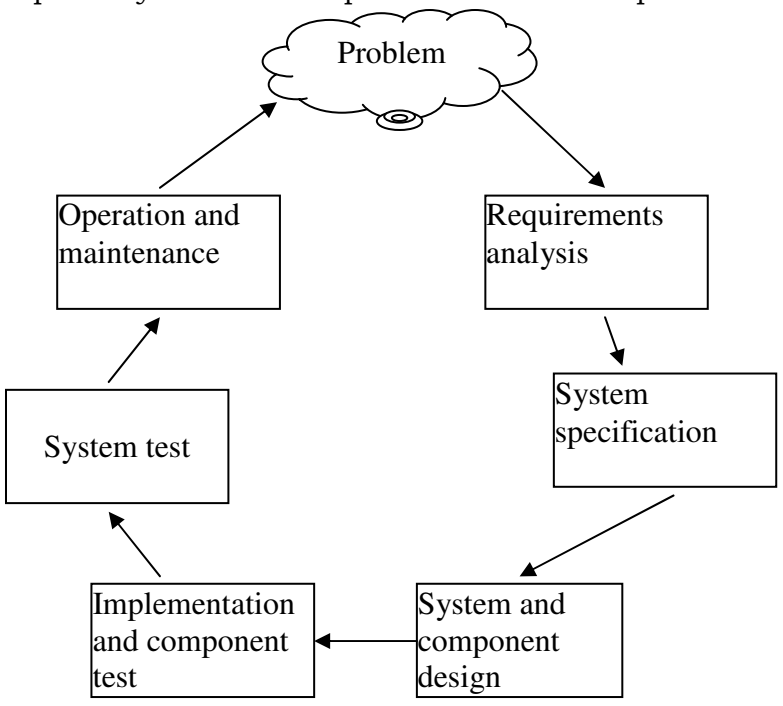


Figure 2.2. The classical sequential software life-cycle

Requirements analysis and planning phase

Goal:

- Determining and documenting:
  - Which steps need to be carried out,
  - The nature of their mutual effects,
  - Which parts are to be automated, and
  - Which recourses are available for the realization of the project.

Important activities:

- Completing the requirements analysis,
- Delimiting the problem domain,
- Roughly sketching the components of the target system,
- Making an initial estimate of the scope and the economic feasibility of the planned project, and
- Creating a rough project schedule.

Products:

- User requirements,
- Project contract, and
- Rough project schedule.

System specification phase

Goal:

- a contract between the client and the software producer (precisely specifies what the target software system must do and the premises for its realization.)

Important activities:

- Specifying the system,
- Compiling the requirements definition,
- Establishing an exact project schedule,
- Validating the system specification, and
- Justifying the economic feasibility of the project.

Products:

- Requirements definition, and
- Exact project schedule.

System and components design

Goal:

- Determining which system components will cover which requirements in the system specification, and
- How these system components will work together.

Important activities:

- Designing system architecture,
- Designing the underlying logical data model,
- Designing the algorithmic structure of the system components, and
- Validating the system architecture and the algorithms to realize the individual system components.

Products:

- Description of the logical data model,
- Description of the system architecture,
- Description of the algorithmic structure of the system components, and
- Documentation of the design decisions.

Implementation and component test

Goal:

- Transforming the products of the design phase into a form that is executable on a computer.

Important activities:

- Refining the algorithms for the individual components,
- Transferring the algorithms into a programming language (coding),
- Translating the logical data model into a physical one,
- Compiling and checking the syntactical correctness of the algorithm, and
- Testing, and syntactically and semantically correcting erroneous system components.

Products:

- Program code of the system components,
- Logs of the component tests, and
- Physical data model.

System test

Goal:

- Testing the mutual effects of system components under conditions close to reality,
- Detecting as many errors as possible in the software system, and
- Assuring that the system implementation fulfills the system specification.

Operation and maintenance

Task of software maintenance:

- Correcting errors that are detected during actual operation, and
- Carrying out system modifications and extensions.

This is normally the longest phase of the software life cycle.

Two important additional aspects:

- Documentation, and
- Quality assurance.

During the development phases the *documentation* should enable communication among the persons involved in the development; upon completion of the development phases it supports the utilization and maintenance of the software product.

Quality assurance encompasses analytical, design and organizational measures for quality planning and for fulfilling quality criteria such as correctness, reliability, user friendliness, maintainability, efficiency and portability.

2.3. DEFINING THE PROBLEM

Most software projects are undertaken to provide solution to business needs. In the beginning of a software project the business needs are often expressed informally as part of a meeting or a casual conversation. In a more formal approach, a customer could send Request For Information (RFI) to organizations to know their area of expertise and domain specifications. The customer puts up a Request For Proposal (RFP) stating the business needs. Organizations will to provide their services will send proposals and one of the proposals is accepted by the customer.

2.4. DEVELOPING A SOLUTION STRATEGY

The business needs have to be understood and the role of software in providing the solution has to be identified. Software development requires a model to be used to drive it and tract it to completion. The model will provide an effective roadmap for the software team.

2.5. PLANNING THE DEVELOPMENT PROCESS

Planning the software development process involves several important considerations. The first consideration is to define a product life-cycle model. A software project goes through various phases before it is ready to be used for practical purposes. For every project, a framework must be used to define the flow of activities such as define, develop, test, deliver, operate, and maintain a software product. There are many well define models that can be use. There could be variations to these models also, depending on the deliverables and milestones for the project. A model has to be selected and finalized to start a project.

The following section discusses the various models such as:

a. Waterfall Model

b. Prototype Model

c. Spiral Model

d. Object-oriented life-cycle model

2.5.1. The Waterfall Model

Developed in the 1970s [Royce 1970].

The Waterfall model is one of the oldest models available for software development and also most widely used. It is also called the Classic Life Cycle Model. It suggests a systematic and sequential approach to software development. The different phases are System Engineering, Requirement Analysis, Design, Coding, Testing and Deployment

Figure 2.3., describes the various stages of the waterfall model. The sequential move from one phase to the next (as illustrated in figure) gave rise to the name Waterfall model. The model mandates that each phase will be executed after the completion of the previous phase. In the figure, the forward arrow describes the flow between phases and backward arrows (unlike water fall) describe the feedback mechanism that should exist between phases to provide information for future development. Each of these phases will now be discussed in detail.

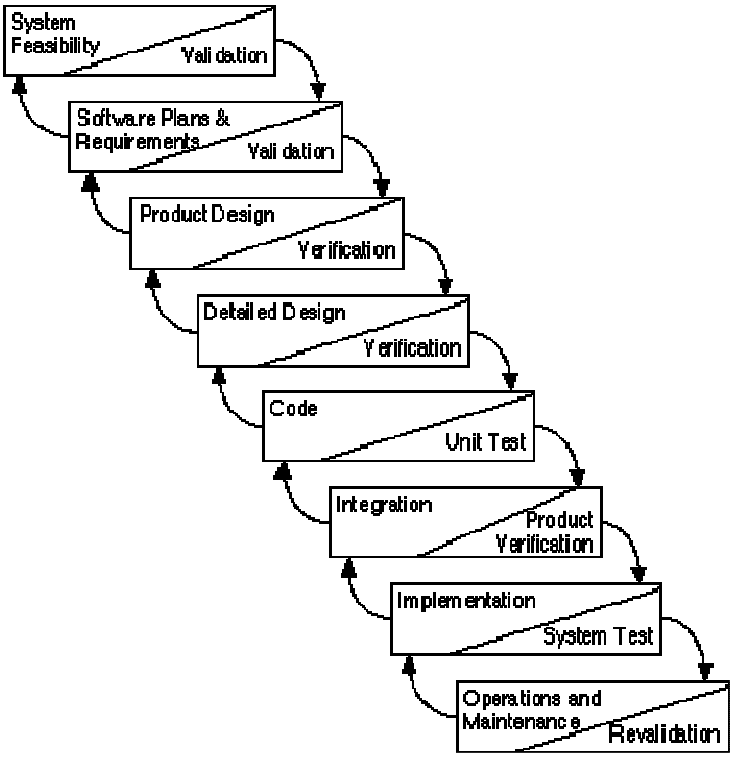


Fig. 2.3. Waterfall Model

Feasibility or Conception	Defining a preferred concept for the software product, and determining its life-cycle feasibility and superiority to alternative concepts. This phase makes clears certain issues such as: ❖ The problem perceived ❖ The goals to be achieved by the solution ❖ The benefits from the solution ❖ The scope of the project.
Requirements or Initiation	In this phase, the software engineers work with users to carry out a macro level study of the user's requirements. The software engineers define the various alternatives possible and the cost-benefit justification of these alternatives.
Product Design or Analysis	In this phase, the software engineers carry out a detailed study of the user's requirements. They then arrive at the proposed system to be built. The model of this system is used to freeze all requirements before the next phase begins. This phase generates the functional

	specifications, which contain: ❖ Outputs to be produced ❖ Inputs that need to be received ❖ Procedures that will get the output from the input ❖ Audit and control requirements that the user can carry out to ensure that the system is acceptable.
Detailed Design:	In this phase, the functional specifications are used for translating the model into a design of the desired system. The purpose of the design phase is to specify a particular software system that will meet the stated requirements. The design specifications that get generated at the end of this phase are technical in nature and typically contain: ❖ User interfaces ❖ Databases and data structures ❖ Algorithms and program structures ❖ Equipment, personnel and other facilities required ❖ Manual procedures that will be part of the implemented system
Coding or Construction	This phase produces the actual code that will be delivered to the customer as the running system. Individual modules developed in this phase are tested before being delivered to the next phase.
Integration or Testing	All the modules that have been developed before are integrated or put together in this phase, and tested as a complete system.
Implementation:	Once the system has passed all the tests, it is delivered to the customer.
Maintenance:	Modifications made after delivery are a part of this phase.
Phase out:	A clean transition of the functions performed by the product to its successors.

The waterfall model is ideal in situations where the requirements are well defined from the beginning. And undergo only minor changes. However, the

requirements for a large number of applications are less stable and not perfectly known at the very beginning. Even in cases where the initial requirements are clear, changes are still likely to occur. For example, changes in technology can cause changes in the initial requirements. This is particularly turn of interactive end user applications. Most software systems of this kind are dynamic – they are required to change over time as they acquire more users. To constrain the development of this kind of software in a rigid process, such as the Waterfall model, can prove counter-productive. Due to these deficiencies in the Waterfall model, an alternative – The Evolutionary model – was developed.

2.5.2. The prototyping-oriented life-cycle model

Developed in the 1990s [Pomberger 1991].

This model is an example of an iterative approach to software development, which is useful when either the customer or the developer is unsure of the exact requirements of the software. The developer creates a model of the system to be built.

This model may be one of the following types:

a) **Throwaway model:** Discard the model once all requirements are understood

b) **Evolving model:** Refine the model every time when the requirements are clearer

Irrespective of whether it's a Throwaway or Evolving model prototyping starts with the communication phase as described in Figure 2.4. The software engineer and the customer together define the overall objective for the software, identify the requirements that are known and outline area for further definition. The iteration is planned quickly and modeled in quick design. The design focuses only in the aspects which are visible to the user. The quick design leads to the construction of a prototype. The prototype is evaluated by the customer. The feedback is taken and used to refine the requirements for the software. The iterations continue, each time refining the software. Prototype Model is extremely useful for doing proof of concepts. The rapid development process can test out new concepts at minimal cost.

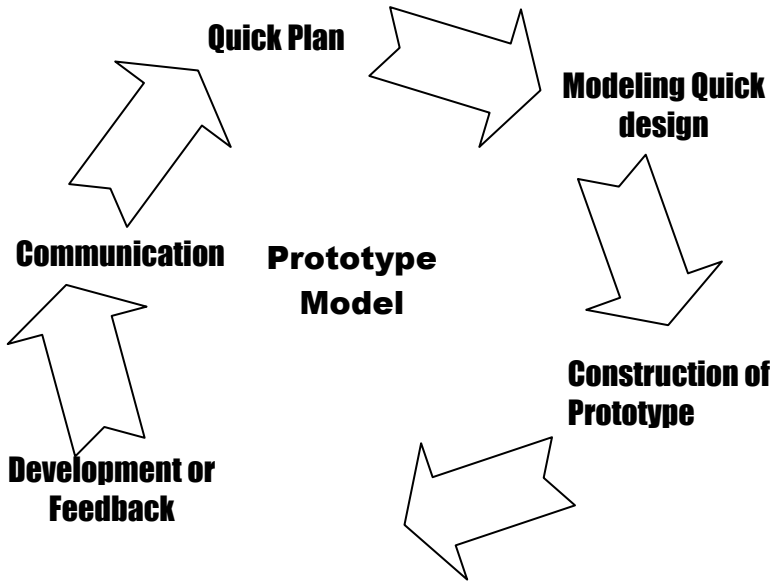


Fig. 2.4: Prototype Oriented-Life cycle Model

Advantages

- Changes can be made easily – An appreciable number of the changes in requirement trigger modifications only in the prototype. Modifying a prototype description is significantly simpler than modifying production code.
- Costs are reduced – Modifying the prototype is much faster and cheaper than modifying the production code.
- User's requirements become clearer to the developer – Usually, the user's requirements are unclear, and the developer is uncertain about what the user meant. Prototyping is the best solution in such cases. The developer can build a prototype and demonstrate to the user his or her understanding of the user's requirements. The user can then verify if the prototype represents his or her requirement.
- User involvement is higher – The user is involved in the development of the system from the very beginning. The user gets a feel of what the real system will look like and suggests changes that he or she desires.

Disadvantages

Disadvantage is that it may lead to indiscipline of development (which the classic life cycle model tries to overcome). For example, if some new requirements are received while in the construction phase, the engineer may drop construction and go back to planning or communication with customer. At some point of time the customer requirements has to be frozen in

conformance with the customer or else it becomes on unending processing, consuming project's time and resources.

When to Use Prototyping: Prototyping is fast becoming popular. It is best suited in situations where:

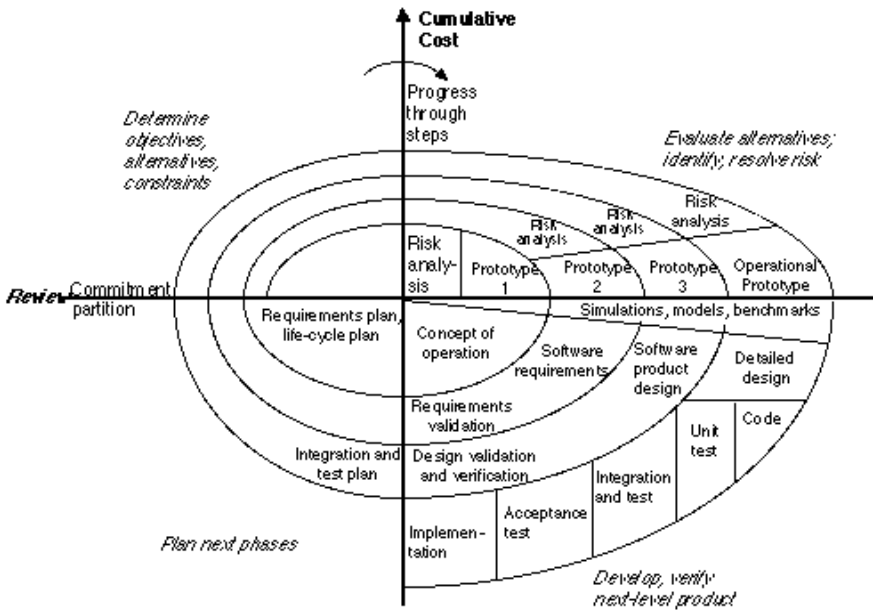
- The user is unable to articulate his or her requirements.
- The user is unwilling or unable to look at abstract models of the system, for example, data flow diagrams.
- The biggest problem area is the user interface. For example, report formats and input screens. The developer can confirm the type of reports that screens that the user wants. User interfaces are the most important aspect of end user applications.
- Good tools are available for prototyping. Tools like screen painters and fourth generation language aid the process of prototyping.

Check your Progress 1

State whether true of false:

a) The prototype is always required to be a small version of the entire system.

b) The difference between the early approach to prototyping and the new approach (Evolutionary model) is that in the latter the prototype is not discarded. It is modified and enlarged to eventually become the real system



2.5.3. The Spiral model

Developed in 1988 [Boehm 1988].

The spiral model is a software development model that combines the above models or includes them as special cases. The model makes it possible to choose the most suitable approach for a given project. Each cycle encompasses the same sequence of steps for each part of the target product and for each stage of completion. It is an evolutionary software process model (*see Fig. 2.5*). The model has a series of evolutionary releases of the software as it is being developed. Every iterative release might be a prototype. The later releases are more complete versions of the software to be produced.

The development strategy behind the Spiral model has been stated as:

- Deliver something to the user.
- Measures the added value to the user in all critical dimensions.
- Adjust both the design and objectives based on observed realities.

The next step evaluates the proposed solution variant with respect to the project goals and applicable constraints, emphasizing the detection of risks and uncertainties. If such are found, measures and strategies are considered to reduce these risks and their effects.

- ***Important aspects of the spiral model:*** Each cycle has its own validation step that includes all persons participating in the project and the affected future users or organizational unit,
- Validation encompasses all products emanating from the cycle, including the planning for the next cycle and the required resources

Spiral model contains six task regions:

- **Customer communication**: tasks required to establish effective communication between developer and customer.
- **Planning**-tasks required defining resources, timelines, and other project –related information.
- **Risk analysis** – tasks required to assess both technical and management risks.
- **Engineering** – tasks required to build one or more representations of the application.
- **Construction and release** – tasks required to construct, test, install, and provide user support (e.g., documentation and training).
- **Customer evaluation** – tasks required to obtain customer feedback based on evolution of the software representations created during the engineering.

Advantages and Disadvantages

The spiral model thus encompasses the earlier models and also introduces the element of risk analysis. It is more realistic because real-world engineering requires considerable iteration. The disadvantage of the spiral model is that it requires considerable expertise in terms of risk assessment and project management.

Check your progress

i. Compare waterfall model and spiral model of software development. Give the advantages and disadvantages of each.

.....

.....

.....

.....

.....

ii. Compare evolutionary prototyping and throw away prototyping.

2.5.4.. The object-oriented life-cycle model

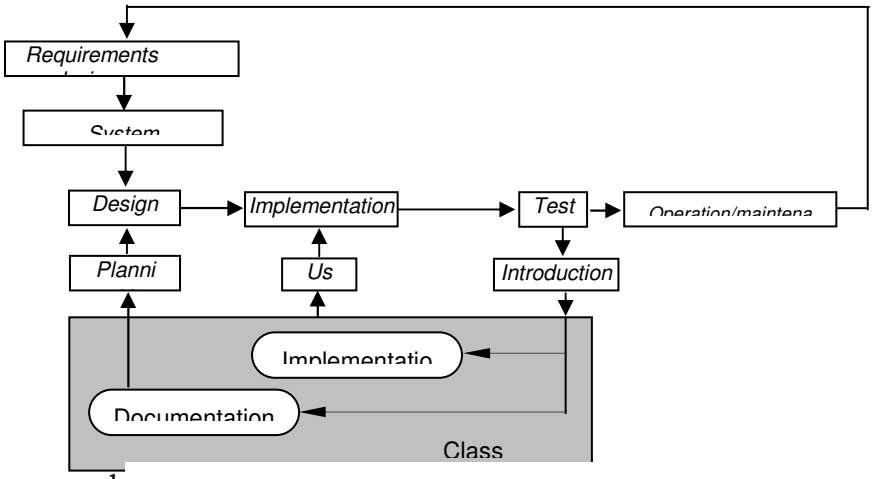
- The usual division of a software project into phases remains intact with the use of object-oriented techniques.
- The requirements analysis stage strives to achieve an understanding of the client's application domain.
- The tasks that a software solution must address emerge in the course of requirements analysis.
- The requirements analysis phase remains completely independent of an implementation technique that might be applied later.
- In the system specification phase the requirements definition describes *what* the software product must do, but not *how* this goal is to be achieved.
- One point of divergence from conventional phase models arises because implementation with object-oriented programming is marked by the assembly of already existing components.

The advantages of object-oriented life-cycle model:

- Design no longer is carried out independently of the later implementation because during the design phase we must consider which components are available for the solution of the problem. Design and implementation become more closely associated, and even the choice of a different programming language can lead to completely different program structures.
- The duration of the implementation phase is reduced. In particular, (sub) products become available much earlier to allow testing of the correctness of the design. Incorrect decisions can be recognized and corrected earlier. This makes for closer feedback coupling of the design and implementation phases.
- The class library containing the reusable components must be continuously maintained. Saving at the implementation end is partially lost as they are reinvested in this maintenance. A new job title emerges, the class librarian, who is responsible for ensuring the efficient usability of the class library.

- During the test phase, the function of not only the new product but also of the reused components is tested. Any deficiencies in the latter must be documented exactly. The resulting modifications must be handled centrally in the class library to ensure that they impact on other projects, both current and future.
- Newly created classes must be tested for their general usability. If there is a chance that a component could be used in other projects as well, it must be included in the class library and documented accordingly. This also means that the new class must be announced and made accessible to other programmers who might profit from it. This places new requirements on the in-house communication structures.

The class library serves as a tool that extends beyond the scope of an individual



project because cla Fig. 2.6: Object Oriented-Life cycle Model vity in subsequent projec

The actual software life cycle recurs when new requirements arise in the company that initiates a new requirements analysis stage.

The object and prototyping-oriented life-cycle model

The specification phase steadily creates new prototypes. Each time we are confronted with the problem of having to modify or enhance existing prototypes. If the prototypes were already implemented with object-oriented technology, then modifications and extensions are particularly easy to carry out. This allows an abbreviation of the specification phase, which is particularly important when proposed solutions are repeatedly discussed with the client. With such an approach it is not important whether the prototype serves solely for specification purposes or whether it is to be incrementally developed to the final product. If no prototyping tools are available, object-oriented programming can serve as a substitute tool for modeling user interfaces. This particularly applies if an extensive class library is available for user interface elements.

For incremental prototyping (i.e. if the product prototype is to be used as the basis for the implementation of the product), object-oriented programming also proves to be a suitable medium. Desired functionality can be added stepwise to the prototypes without having to change the prototype itself. These results in a clear distinction between the user interfaces modeled in the specification phase and the actual functionality of the program. This is particularly important for the following reasons:

- This assures that the user interface is not changed during the implementation of the program functionality. The user interface developed in collaboration with the client remains as it was defined in the specification phase.
- In the implementation of the functionality, each time a subtask is completed, a more functional prototype results, which can be tested (preferably together with the client) and compared with the specifications? During test runs situations sometimes arise that require rethinking the user interface. In such cases the software life cycle retreats one step and a new user interface prototype is constructed.

Since the user interface and the functional parts of the program are largely

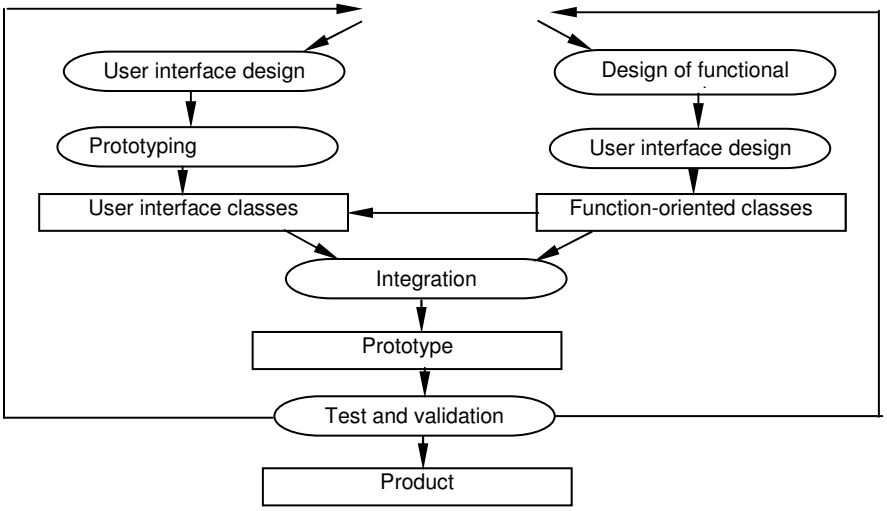


Fig. 2.7: Software development with prototyping and object-orientation

decoupled, two cycles result that shares a common core. The integration of the functional classes and the user interface classes creates a prototype that can be tested and validated. This places new requirements on the user interface and/or the functionality, so that the cycle begins.

Types of project planning:

Plan Type	Description

Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements of the system, maintenance costs and effort required.
Staff development plan	Describes how the skills and experience of the project team members will be developed. See

2.6. PLANNING AN ORGANIZATION STRUCTURE

Completing a software project is a team effort. The following options are available for applying human resources to a project that will require 'n' people working for 'K' years.

- 'n' individuals are assigned to 'm' different functional tasks.
- 'n' individuals are assigned to 'm' different functional tasks (m<n) so that informal teams are established and coordinated by project manager.
- 'n' individuals are organized into 't' teams and each team is assigned one/more functional tasks.

Even though the above three approaches have their pros and cons, option 3 is most productive.

There are several roles within each software project team. Some of the roles in a typical software project are listed below:

Designation	**Job Profile**
Project Manager	Initiates, plans, tracks and manages resources of an entire project
Module Leader	A software engineer who manages and leads the team working on a particular module of the software project. The module leader will conduct reviews and has to ensure the proper

	functionality of the module
Analyst	A software engineer who analyzes the requirements gathered. Analysis of the requirements is done to get a clear understanding of the requirements.
Domain Consultant	An expert who knows the system of which the software is a part. This would involve the technical knowledge of how the entities of the domain interface with the software being developed. For example, a banking domain consultant or a telecom domain consultant.
Reviewer	A software engineer who reviews artifacts like project documents or code. The review can be a technical review which scrutinizes the technical details of the artifact. It could be a review where the reviewer ascertains whether or not the artifact adheres to a prescribed standard
Architect	A software engineer involved in the design of the solution after the analyst has clearly specified the business requirements
Developer	A software engineer, who writes the code, tests it and delivers it error free
Tester	A software engineer who conducts tests on the completed software or a unit of the software. If any defects are found these defects are logged and a report is sent to the owner of the tested unit.

Programming Team Structure

Every programming team must have an internal structure. The best team structure for any particular project depends on the nature of the project and the product, and on the characteristics of the individual team members. Basic team structure includes:

a. **Democratic team:** Team Member participate in all decisions

b. **Chief Programmer Team:** A chief programmer is assisted and supported by other team members.

c. **Hierarchical Team:** The project leader assigns tasks attend reviews and walkthrough, detects problem areas, balances the workload and participates in technical activities.

Democratic Team

This was first described by Weinberg as the "egoless team". In an egoless team goals are set and decisions made by group consensus. Group leadership rotates from member to member based on the tasks to be performed and the differing abilities of the team members. Work products (requirements, design, source code, user manual, etc) are discussed openly and are freely examined by all team members.

Advantage:

- ❖ Opportunity for each team member to contribute to decision
- ❖ Opportunity for team members to learn from one another
- ❖ Increased Job satisfaction that results from good communication in open, non-threatening work environments.

Disadvantages

- ❖ Communication overhead required in reaching decision,
- ❖ All team members must work well together,
- ❖ Less individual responsibility and authority can result in less initiative and less personal drive from team members.

The chief programmer team

Baker's organizational model ([Baker 1972])

- ➢ Important characteristics:
  - The lack of a project manager who is not personally involved in system development
  - The use of very good specialists
  - The restriction of team size
- ➢ The chief programmer team consists of:
  - The chief programmer
  - The project assistant
  - The project secretary
  - Specialists (language specialists, programmers, test specialists).
- ➢ The chief programmer is actively involved in the planning, specification and design process and, ideally, in the implementation process as well.
- ➢ The chief programmer controls project progress, decides all important questions, and assumes overall responsibility.
- ➢ The qualifications of the chief programmer need to be accordingly high.

- The project assistant is the closest technical coworker of the chief programmer.
- The project assistant supports the chief programmer in all important activities and serves as the chief programmer's representative in the latter's absence. This team member's qualifications need to be as high as those of the chief programmer.
- The project secretary relieves the chief programmer and all other programmers of administrative tasks.
- The project secretary administrates all programs and documents and assists in project progress checks.
- The main task of the project secretary is the administration of the project library.
- The chief programmer determines the number of specialists needed.
- Specialists select the implementation language, implement individual system components, choose and employ software tools, and carry out tests.

Advantages

- The chief programmer is directly involved in system development and can better exercise the control function.
- Communication difficulties of pure hierarchical organization are ameliorated. Reporting concerning project progress is institutionalized.
- Small teams are generally more productive than large teams.

Disadvantages

- It is limited to small teams. Not every project can be handled by a small team.
- Personnel requirements can hardly be met. Few software engineers can meet the qualifications of a chief programmer or a project assistant.
- The project secretary has an extremely difficult and responsible job, although it consists primarily of routine tasks, which gives it a subordinate position. This has significant psychological disadvantages. Due to the central position, the project secretary can easily become a bottleneck.

The organizational model provides no replacement for the project secretary. The loss of the project secretary would have failing consequences for the remaining course of the project.

Hierarchical organizational model

- There are many ways to organize the staff of a project. For a long time the organization of software projects oriented itself to the hierarchical organization common to other industrial branches. Special importance is

vested in the decomposition of software development into individual phases. A responsible leader is assigned to each of the phases, which are led and controlled by the project leader and which, depending on the size of the project, are led and controlled either by a single person or by a group leader.

- The project manager normally also has a project management staff with advisory and administrative tasks.
- The larger the project, the greater is the number of hierarchy levels in the organizational schema.

The project manager's tasks and responsibilities encompass

- personnel selection,
- assignment and management,
- planning of and division of labor for the project,
- project progress checks, and
- appropriate measures in case of cost or schedule overruns.

- *The project management staff includes* personnel who advise the project manager in task-specific questions, provide support in administrative tasks concerning project progress checks, prepare project standards, provide the necessary resources, and carry out training for project team members as needed.
- *The managers at the middle management level* are responsible for planning, execution and control of phase-related activities of the software life cycle.

2.7. OTHER PLANNING ACTIVITIES

Project management is the discipline of defining and achieving targets while optimizing the use of resources (time, money, people, materials, energy, space, etc) over the course of a project (a set of activities of finite duration).

Project Management is quite often the province and responsibility of an individual project manager. This individual seldom participates directly in the activities that produce the end result, but rather strives to maintain the progress and productive mutual interaction of various parties in such a way that overall risk of failure is reduced.

In contrast to on-going, functional work, a project is "a temporary endeavor undertaken to create a unique product or service." The *duration* of a project is the time from its start to its completion, which can take days, weeks, months or even years. Typical projects include the engineering and construction of various public or consumer products, including buildings, vehicles, electronic devices, and computer software.

Project Management is composed of several different types of activities such as:

- Planning the work
- Assessing risk
- Estimating resources
- Organizing the work
- Acquiring human and material resources
- Assigning tasks
- Directing activities
- Controlling project execution
- Reporting progress
- Analyzing the results based on the facts achieved

Project control variables

Project Management tries to gain control over five variables:

Time - The amount of time required to complete the project. Typically broken down for analytical purposes into the time required to complete the components of the project, which is then further broken down into the time required to complete each task contributing to the completion of each component.

Cost - Calculated from the time variable. Cost to develop an internal project is time multiplied by the cost of the team members involved. When hiring an independent consultant for a project, cost will typically be determined by the consultant or firm's hourly rate multiplied by an estimated time to complete.

Quality - The amount of time put into individual tasks determines the overall quality of the project. Some tasks may require a given amount of time to complete adequately, but given more time could be completed exceptionally. Over the course of a large project, quality can have a significant impact on time and cost (or vice versa).

Scope - Requirements specified for the end result. The overall definition of what the project is supposed to accomplish, and a specific description of what the end result should be or accomplish.

Risk - Potential points of failure. Most risks or potential failures can be overcome or resolved, given enough time and resources.

Three of these variables can be given by external or internal customers. The value(s) of the remaining variable(s) is/are then set by project management, ideally based on solid estimation techniques. The final values have to be agreed upon in a negotiation process between project management and the customer. Usually, the values in terms of time, cost, quality and scope are contracted.

2.8. REVIEW QUESTIONS

1. Discuss project scheduling.
2. Discuss Human-resources planning.
3. What kind of team structure would you recommend for waterfall method and prototyping?

2.9. LET US SUM UP

Defining the problem

1 Develop a definitive statement of the problem to be solved. Include a description of the present situation, problem constraints, and a statement of the goals to be achieved. The problem statement should be phrased in the customer's terminology

2 Justify a computerized solution strategy for the problem.

3 Identify the functions to be provided by, and the constraints on, the hardware subsystem, the software subsystem, and the people subsystem.

4 Determine system-level goals and requirements for the development process and the work products.

5 Establish high-level acceptance criteria for the system.

Developing a solution strategy

6 Outline several solution strategies, without regard for constraints.

7 Conduct a feasibility study for each strategy.

8 Recommend a solution strategy, indicating why other strategies were rejected.

9 Develop a list of priorities for product characteristics.

Planning the development process

10 Define a life-cycle model and an organizational structure for the project

11 Plan the configuration management, quality assurance, and validation activities.

12 Determine phase-dependent tools, techniques, and notations to be used

2.10. LESSON END ACTIVITIES

Suppose we wish to computerize the activities of Department, which offer various programmes of the university. The activities are as follows:

- ❖ Forwarding the faculties bio-data to University
- ❖ Schedule the classes (both theory and practical)
- ❖ Identify Practical Centres

- ❖ Collect Assignments from students and get it evaluated by faculties
- ❖ Students enquiry
- ❖ Dispatch of grades to university

Develop a system requirement Specification, design DFDs, and identify modules and its operations. Select appropriate data structures for the various modules.

2.11. POINTS FOR DISCUSSION

1. Akkash wants to build a restaurant. He calls upon Rithanya, a famous architect and builder, for this task. Rithanya goes through the following steps:

   a) She first talks to Akkash to understand the general specifications.

   b) She then draws the complete plan on how to go about building the structure.

   c) Then, with the help of a contractor, she finishes the building in one year.

The steps that Rithanya has gone through are similar to some of the phases of the Waterfall Model. Identify them.

2. As far as the new user is concerned, which approach do you think will be more appreciated – Waterfall or Prototyping?

2.12. REFERENCES

1. Richard Fairley, "Software Engineering Concepts", Tata McGraw-Hill, 1997.
2. Roger S.Pressman, Software engineering- A practitioner's Approach, McGraw-Hill International Edition, 5th edition, 2001.
3. IEEE, Standards Collection, Software Engineering, IEEE, New York, 1994

4. J.C. Wethebe and N.P. Vitalani, System Analysis and Design: Best Approach, Ed.4, West, St. Paul 1994.

LESSON 3: SOFTWARE COST ESTIMATION

Contents

3.0. AIMS AND OBJECTIVES

- To Estimate the cost of a software product
- To understand the various factors influence software costs
- To use empirical estimation models

3.1. INTRODUCTION

Software is the most expensive element in most computer-based systems. A large cost estimation error can make the difference between profit and loss. Too many variables – human, technical, environmental, political – can affect the ultimate cost of software and effort applied to it.

3.2. SOFTWARE COST ESTIMATION

Cost estimation

[Boehm 1981], [Putnam 1978], [Albrecht 1983], [Schnupp 1976]

- The necessity of cost estimation stems from the requirements of scheduling and cost planning. For lack of more precise methods, cost estimation for software development is almost always based on a comparison of the current project with previous ones. Due to the uniqueness of software systems, the number of comparable projects is usually quite small, and empirical data are seldom available. But even if there are exact cost records of comparable projects, these data are based on the technical and organizational conditions under which the comparable project was carried out at that time.

- The technical and organizational conditions are variable parameters, which makes empirical data from comparable projects only an unreliable basis for estimates.
- Relationship between the best and worst programming experience (referring to the same task, [Schnupp 1976]):
- The time requirement for each task handled in a team consists of two basic components ([Brooks 1975]):

  (1) Productive work

  (2) Communication and mutual agreement of team members

  If no communication were necessary among team members, then the time requirement *t* for a project would decline with the number *n* of team members

  $t \approx 1/n$

  If each team member must exchange information with one other and that the average time for such communication is *k*, then the development time follows the formula:

$t \approx 1/n + k.\ n^2/2$

"Adding manpower to a late software project makes it later." ([Brooks 1975])

- Most empirical values for cost estimation are in-house and unpublished. The literature gives few specifications on empirical data, and these often deviate pronouncedly. The values also depend greatly on the techniques and tools used.
- Distribution of the time invested in the individual phases of software development (including the documentation effort by share) according to the selected approach model and implementation technique ([Pomberger 1996]):

  Approach model: classical sequential software life cycle

  Implementation technique: module-oriented

problem analysis and system specification....	25%
design.....	25%
implementation.....	15%
testing.....	35%

  Approach model: prototyping-oriented software life cycle

  Implementation technique: module-oriented

problem analysis and system specification.....	40%
design.....	25%
implementation.....	10%
testing.....	25%

  Approach model: object- and prototyping-oriented software life cycle
  Implementation technique: object-oriented

problem analysis and system specification.....	45%
design.....	20%
implementation.....	8%
testing.....	27%

The following options are useful to achieve reliable cost and effort estimates:

1. ***Delay estimation until late in the project***. The longer we wait, the less likely we are to make errors in our estimates. However this is not practical. Cost estimates must be provided "up-front".
2. ***Base estimates on similar projects that have already been completed***. This works well if the current project is quite similar to past efforts. Unfortunately, past experience has not always been a good indicator of future results.
3. Use "***decomposition techniques***" to generate project cost and effort estimates. These techniques use a "*divide and conquer*" approach to estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a step-wise fashion.

Use one or more ***empirical models*** for software cost and effort estimation. A model is based on experience (historical data) and takes the form $\boldsymbol{d = f(v_i)}$, where $\boldsymbol{d}$ is one of a number of estimated variables (eg. effort, cost, project duration) and $\boldsymbol{v_i}$ are selected independent parameters (eg. Estimated LOC or FP).

3.3. SOFTWARE COST FACTORS

3.3.1. Factors for cost and time estimates

- Experience and qualifications of the estimator
- Type of software (e.g., function-, data- or object-oriented, time-critical software with high efficiency requirements, control and monitoring software with high quality requirements)
- Anticipated complexity (e.g., number of components (modules, classes), number of component couplings, number of methods per class, complexity of methods)
- Expected size of program (number of statements)
- Experience and qualifications of personnel (e.g., project experience, programming experience, knowledge of methods, experience in use of tools, knowledge of problem domain)
- Techniques and tools to be used (e.g., planning, design and documentation techniques, test strategies and tools, programming languages)
- Number of staff members

3.3.2. LOC (Lines of Code) Based Estimation

To illustrate the LOC based estimation technique, let us consider the development of software for Computer-Aided Design (CAD) application. If we

assume that the range of LOC estimates for the 3D geometric analysis function is:

Optimistic: 4600

Most likely: 6900

Pessimistic: 8600

Applying the equation to compute EV (Earned Value), the expected value for the 3D geometric analysis function is 6800 LOC. Estimates for all the modules of the CAD application are computed in a similar fashion. By summing all the estimated values, an estimate of the lines of code is established for the CAD software system.

3.3.3. Process-Based Estimation

The process is decomposed into a relatively small set of activities or tasks and the effort required to accomplish each task is estimated. A process-based estimation involves the following steps:

1. Delineate the software functions obtained from the project scope.
2. A series of software process activities must be performed for each function.
3. Functions and related software process activities may be represented as part of a table.
4. The planner estimates the effort (eg. person-months) that will be required to accomplish each software process activity for each software function.
5. Average labor rates (eg. cost/unit effort) are applied to the effort estimated for each process activity.
6. Costs and Effort for each function and software process activity are computed.

3.4. SOFTWARE COST ESTIMATION TECHNIQUES

Cost estimation models

- *COCOMO Model,*
- *Putnam Estimation Model*
- *Function Point Model.*

COCOMO MODEL

The constructive cost model is an algorithmic cost estimating model; it is a bottom-up technique. It starts estimating at sub system level and finally sums up all the estimates.

The steps required to estimate the software cost using COCOMO are as follows:

- Identify all subsystems and modules in the product.
- Estimate the size of each module and calculate the size of each subsystem and total system.

- Specify module-level effort multipliers for each module. The module-level multipliers are: Product complexity, Programmer capability, virtual machine experience and Programming language experience.
- Compute the module effort and development time estimates for each module and its subsystems.
- Compute the total system effort and development time.
- Perform a sensitivity analysis on the estimate to establish tradeoff benefits
- Add other development costs, such as planning and analysis that are not included in the estimate.
- Compare the estimate with one developed top-down Delphi estimation. Identify and rectify the differences in the estimates.

The COCOMO Models are defined for *three classes* of software projects. They are:

1. ***organic mode*** – relatively small, simple software projects in which small teams with good application experience work to a set of less than rigid requirements (e.g. A thermal analysis program developed for a heat transfer group)
2. ***semi-detached mode*** – an intermediate software project in which teams with mixed experience levels must meet a mix of rigid and less than rigid requirements (e.g. A transaction processing system with fixed requirements for terminal hardware and database software)
3. ***embedded mode*** – a software project that must be developed with a set of tight hardware, software, and operational constraints. (eg. Flight control software for aircraft).

Software Project Estimation using COCOMO:

Nominal Effort equation

$$PM = a_i \, KLOC^{\,b_i}$$

$$TDEV = c_i \, E^{\,d_i}$$

Where PM is Programmer months

TDEV is Product Development time in months.

KLOC is number of delivered lines of code.

a_i, c_i is coefficients

b_i, d_i is exponents.

Coefficients and exponents are:

S/W Project	$\mathbf{a_i}$	$\mathbf{b_i}$	$\mathbf{c_i}$	$\mathbf{d_i}$
Organic	3.2	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35

Embedded	2.8	1.20	2.5	0.32

Example:

Consider the product to be developed in 10-KLOC embedded mode s/w product.

PM = $(2.8) * (10)^{1.20}$

= 44.4

TDEV = $(2.5) * (44)^{0.32}$

=8.4

It is predicted that 44.4. Programmer months and 8.4 elapsed months for product development.

Check your progress

i. Three different cost estimation models are _____, _____ and _____

ii. The COCOMO Models are defined for three classes of software projects. They are _____, _____ and _____

iii. Important factors of Cost estimation are _____ , _____ and _____

iv. Boehm's maintenance cost estimation is calculated in terms of a quantity called the _____

Solutions

I

Ii

Iii

Iv

3.5. STAFFING LEVEL ESTIMATION

Estimating Staff includes the processes required to make the most effective use of the people involved with the project. It includes both organizational positions (e.g. Manager, senior software engineer, etc.) and specialty (e.g. Telecommunications, database, etc.). The number of people required varies for every software project and it can be determined only after an estimate of development effort is made.

Project schedule primarily depends on effort. However, the relationship is not linear. During the earlier phases of the project like Requirement Analysis and HLD a lesser number of people would be required as against those required during the Build phase (for coding and unit testing). Once Build phase is complete, the staff requirement for the next phase i.e. the Testing phase would be lower. While staffing a project, adequate care is taken to have a blend of experienced and non-so-experienced people.

3.6. ESTIMATING SOFTWARE MAINTENANCE COSTS

Maintenance Costs

- Typical software organizations spend anywhere from 40 to 70 percent of all funds for maintenance.
- Maintenance-bound organizations result loss or postponement of development opportunities.
- Customer dissatisfaction when requests cannot be addressed.
- Reduction in overall software quality as a result of changes that introduce latent errors in the maintained software.

Maintenance cost factors

Non-technical factors

- Application domain
- Staff stability
- Program age
- External environment
- Hardware stability

1) *The application being supported.*

   If the application of the program is *clearly defined* and *well understood*, the system requirements may be definitive and maintenance due to changing requirements minimized.

   If the application is completely new, it is likely that the initial requirements will be modified frequently, as users gain experience with the system.

2) *Staff stability.* It is easier for the original writer of a program to understand and change a program rather than some other individual who must understand the program by study of its documentation and code listing.

If the programmer of a system also maintains that system, maintenance costs will be reduced.

In practice, the nature of the programming profession is such that individuals change jobs regularly. It is unusual for one person to develop and maintain a program throughout its useful life.

3) *The lifetime of the program.*

   The useful life of a program depends on its application.

Programs become obsolete when the application becomes obsolete or their original hardware is replaced and conversion costs exceed rewriting costs.

The older a program, the more it has been maintained and the more degraded its structure.

Maintenance costs tend to rise with program age.

4) *The dependence of the program on its external environment.*

If a program is dependent on its external environment it must be modified as that environment changes. For example, changes in a taxation system might require payroll, accounting, and stock control programs to be modified. Taxation changes are relatively common and maintenance costs for these programs are related to the frequency of these changes.

A program used in a mathematical application does not normally depend on humans changing the assumptions on which the program is based.

5) *Hardware stability.*

If a program is designed to operate on a particular hardware configuration and that configuration does not change during the program's lifetime, no maintenance costs due to hardware changes will be incurred. However, hardware developments are so rapid that this situation is rare. The program must be modified to use new hardware which replaces obsolete equipment.

Maintenance cost estimation

Boehm's maintenance cost estimation

- *Boehm's maintenance cost estimation* ([Boehm 1981]) is calculated in terms of a quantity called the *Annual Change Traffic* (ACT) which is defined as follows:

  The fraction of a software product's source instructions which undergo change during a (typical) year either through addition or modification.

  $ACT = (DSI_{added} + DSI_{modified}) / DSI_{total}$

  $PM = ACT * MM$

  where

  DSI is no. of source instructions

  PM is no. of Program months.

  MM is no. of months for development

A further enhancement is provided by an effort adjustment factor EAF

$PM = ACT * EAF * MM$

Where

EAF – recognize that the effort multipliers for maintenance may be different from the effort multipliers used for development.

3.7. REVIEW QUESTIONS

1. Define Estimation
2. What are the types of estimation?
3. What are the factors that affect the efficiency of estimation? Explain the factors.
4. Discuss estimation using decomposition techniques
5. Discuss empirical estimation models.
6. Write a note on automated estimation tools.
7. Specify, design and develop a program that implements COCOMO model.
8. Discuss the importance of making correct estimates project management

3.8. LET US SUM UP

- COCOMO was proposed by Barry W. Boehm in Software Engineering Economics.
- COCOMO covers a broad spectrum of software development projects:
- From application software to systems software
- From small software to large software
- From new development to modifications / enhancements
- This model is good for estimating both effort and elapsed time by project phases
- The model needs lines-of-code as input

3.9. LESSON END ACTIVITIES

i. Some experimental evidence suggests that the initial size estimate for a project affects the nature and results of the project. Consider two different managers charged with developing the same application one estimates that the size of the application will be 50,000 lines while the other estimates that will be 1,00,000 lines. Discuss how these estimates affect the project throughout its life cycle.

3.10. POINTS FOR DISCUSSION

i. Use the COCOMO equations to estimate the programmer-months and development time for your term project. Prepare a range of estimates based on probable product size.

3.11. REFERENCES

1. Richard Fairley, "Software Engineering Concepts", Tata McGraw-Hill, 1997.
2. Roger S.Pressman, Software engineering- A practitioner's Approach, McGraw-Hill International Edition, 5th edition, 2001.
3. http://sunset.usc.edu/COCOMO2.0/Cocomo.html
4. Matson, J., B. Barett, and J. Mellichamp, "Software Development Cost Estimation Using Funciton Points", IEEE Trans. Software Engineering vol. 20, no 4, April 1994, pp. 275-287

LESSON 4: REQUIREMENT SPECIFICATIONS

Contents

4.0. AIMS AND OBJECTIVES

- To understand the sate of "what" of the software product without implying "how".
- Define system elements like software, hardware, people data base, documentation, and procedures.

4.1. INTRODUCTION

A complete understanding of the software requirements is essential to the success of a software development effort. Requirements analysis task is a process of discovery, refinement, modeling, and specification. The specification produced becomes the foundation for all software engineering activities.

Requirements Analysis is a software engineering task that bridges the gap between system-level software allocation and software design (*See Figure 4.1*).

Requirements analysis enables the system engineer to,

- Specify software function & performance,
- Indicate software's interface with other system elements.
- Establish design constraints that the software must meet.

Requirements analysis allows the software engineer to,

- Refine the software allocation
- Build models of the process, data and domains

Requirements analysis provides the software engineer,

- Information of data, architectural and procedural design.
- Means to assess software quality, with the help of requirements specification.

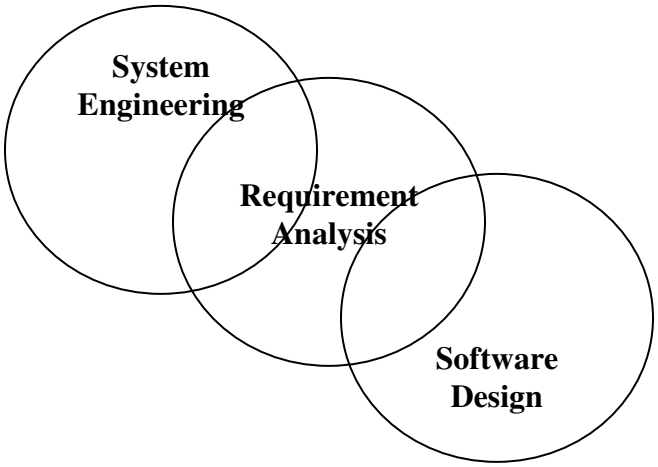


Figure 4.1.: ***Requirements Analysis - a bridge between System Engg. & Design***

Software requirements analysis may be divided into the following five areas of effort:

- Problem Recognition
- Evaluation & Synthesis
- Modeling
- Specification, and
- Review

Problem Recognition: Initially the analyst studies the system specification and the software project plan. Next communication for analysis must be established so that problem recognition is ensured. The goal of the analyst is to identify the basic problem elements as perceived by the user/customer.

Evaluation & Synthesis: The analyst must define all data objects, evaluate the flow and content of information, define software functions and establish system interface characteristics considering the context of events that affect the system and uncover additional design constraints.

Upon evaluating current problems and desired information, the analyst begins to synthesize one or more solutions. To begin, the data processing functions, and behavior of the system are defined in detail. Next basic architectures for implementation are considered. The process of evaluation and synthesis continues till both the customer and analyst feel confident that the software can adequately be specified for subsequent development steps.

Modeling: Developing a model for an industrial-strength software system prior to its construction or renovation is as essential as having a blueprint for large building. Good models are essential for communication among project teams and to assure architectural soundness. As the complexity of systems increase, so does the importance of good modeling techniques. The analyst creates models of the system to better understand data and control flow, functional processing, and behavioral operation. The model serves as a foundation for software design and as the basis for the creation of a specification.

Specification: The final output of the requirements analysis is the creation of a Software Requirements Specification (SRS) document. For simple problems the specification activity might be the end result of the entire analysis. However in most real life problems, the problem analysis and specification are done concurrently. A good SRS should be understandable, complete, consistent, and unambiguous.

Review: Requirements reviews are the most common method employed for validating the requirements specifications. Reviews are used throughout software development for quality assurance and data collection. Requirements review is a review by a group of people to find out errors and point out other matters of concern in the requirements specifications of a system.

4.1.1. Specification Principles

The list of basic specification principles given below will provide a basis for representing software requirements.

1. Separate functionality from implementation.
2. Develop a model of the desired behavior of a system that encompasses data and the functional responses of a system to various stimuli from the environment.
3. Establish the context in which software operates by specifying the manner in which other system components interact with software.
4. Define the environment in which the system operates and indicate how "a highly twisted collection of agents react to stimuli in the environment produced by those agents.
5. Create a cognitive (perception) model rather than design or implementation model. The cognitive model describes a system as perceived by its user community.
6. Recognize that "the specification must be tolerant of incompleteness and augmentable". A specification is always a model – an abstraction - of some real situation that is normally quite complex. Hence it will be incomplete and will exist at many levels of detail.
7. Establish the content and structure of a specification in a way that will enable it to be amenable to change.

Check your progress

i. Requirements Analysis is a software engineering task that bridges the gap between _____ and _____

ii. Requirements analysis enables the system engineer to _____, _____ and _____

iii. Requirements analysis provides the software engineer to _____ and _____

iv. Two types of formal specifications they are _____

Solutions

I

.....

Ii

.....

Iii

.....

Iv

.....

4.2. THE SOFTWARE REQUIREMENT SPECIFICATION

The Format of a requirements specification document is presented in the following Table.

Section	Requirement Specification
1.	Product Overview and Summary
2.	Development, Operating and Maintenance Environments
3.	External Interface and Data Flow
4.	Functional Requirements
5.	Performance Requirements
6.	Exception Handling
7.	Early subsets and Implementation Priorities

8.	Foreseeable Modifications and Enhancements
9.	Acceptance Criteria
10.	Design Hints and Guidelines
11.	Cross-Reference Index
12.	Glossary of Terms

There are number of desirable properties that a software requirement specification should processes. They are:

- ❖ Correct
- ❖ Complete
- ❖ Consistent
- ❖ Unambiguous
- ❖ Functional
- ❖ Verifiable
- ❖ Traceable
- ❖ Easily changed

4.3. FORMAL SPECIFICATION TECHNIQUES

pecifying the functional characteristics of a software product is one of the most important activities to be accomplished during requirements analysis. Formal specifications have the advantage of being brief and clear-cut, they support formal reasoning about the functional specifications, and they provide a basis for verification of the resulting software product. Formal notations are not appropriate in all situations or for all types of systems. However, our experience indicates that there is usually too little formalism in software development, rather than too much hence, our emphasis on formalism.

There are two types of formal specifications they are:

1. Relational Notations
2. State-Oriented Notations

Relational Notations

elational Notations are based on the concepts of entities and attributes. Entities are named elements in a system; the names are chosen to denote the nature of the elements (e.g., stack, queue). Attributed are specified by applying functions and relations to the name entities. Attributes specify permitted operations on entities, relationships among entities, and data flow between entities.

Example:

Implicit equations, recurrence relations, algebraic axioms and regular expressions.

State-Oriented Notations

he state of a system is the information required to summarize the status of system entities at any particular point in time; given the current state and the current stimuli, the next state can be determined. The execution history by which the current state was attained does not influence the next state; it is only dependent on the current state and current stimuli.

Example:

Decision tables, event tables, transition tables, finite-state mechanisms, and Petri nets.

4.4. Languages & Processors for Requirements Specification

A number of special-purpose languages and processors have been developed to permit concise statement and automated analysis of requirements specifications for software. Some specification languages are graphical in nature, while others are textual; all are relational in nature. Some specification languages are manually applied and other has automated processors. Some are the specification languages are:

a. PSL/PSA

b. RSL/REVS

c.SADT

d. SSA

e. GIST

a. PSL/PSA

The Problem Statement Language (PSL) was developed by Prof. Daniel Teichrow at the University of Michigan. The Problem Statement Analyzer (PSA) is the PSL processor.

The objective of PSL is to permit expression of much of the information that commonly appears in a Software Requirements Specification. In PSL, system descriptions can be divided into eight major aspects:

i. System input/output flow

ii. System structure

iii. Data structure

iv. Data derivation

v. System size and volume

vi. System dynamics

vii. System properties

viii. Project management

PSL contains a number of types of objects and relationships to permit description of these eight aspects. The system input/output flow aspect deals with the iteration between a system and its environment.

The Problem Statement Analyzer (PSA) is an automated analyzer for processing requirements stated in PSL.

b. RSL/REVS

The Requirements Statement Language (RSL) was developed by the TRW Defense and Space Systems Group to permit brief and clear-cut specification of requirements for real-time software systems. The Requirements Engineering Validation Systems (REVS) processes and analyzes RSL statements; both RSL and REVS are components of the Software Requirements Engineering Methodology. Many of the concepts in RSL are based on PSL.

c. SADT

Structured Analysis and Design Technique (SADT) was developed by D.T. Ross and Colleagues at Softech, Inc. SADT incorporates a graphical language and set of methods and management guidelines for using the language. The SADT language is called the language of Structured Analysis (SA). The SA language and the procedures for using it are similar to the engineering blueprint systems used in civil and mechanical engineering.

d. SSA

Structured System Analysis is used primarily in traditional data processing environments. Like SADT, SSA uses a graphical language to build models of systems. Unlike SADT, SSA uses graphical concepts; however, SSA does not provide the variety of structural mechanisms available in SADT. There are four basis features in SSA: data flow diagrams, data dictionaries, procedure logic representation and data store structuring techniques. SSA data flow diagrams are similar to SADT diagrams, but they do not indicate mechanism and control, and an additional notation is used to show data stores.

e. GIST

Gist is a formal specification language developed at the USC/Information Sciences Institute by R. Balzar and colleagues. Gist is a textual language based on a relational model of objects and attributes. A Gist specification is a formal description of valid behaviors of a system. A specification is composed of three parts:

i. A specification of object types and relationships between these types. This determines a set of possible states.

ii. A specification of actions and demons which define transitions between possible states.

iii. A specification of constraints on states and state transitions.

4.5. REVIEW QUESTIONS

1. Define requirements analysis.
2. List requirements analysis tasks.
3. Discuss the problems with requirements analysis.
4. Discuss Balzer and Goldman's specification principles.
5. List the different analysis methods.

4.6. LET US SUM UP

Software requirements definition is concerned with preparation of the software Requirements Specifications. The format and contents of the Software Requirements Specification have been discussed, and relational and state oriented notations for specifying the functional characteristics of a software product were presented.

Several notations are automated tools for software requirements were described. Some of the notations (SADT and SSA) do not have automated processors, but are on the other hand useful techniques. Most of the automated tools for requirements definition are in fact analysis and design tools; they incorporate notations for describing structure and processing details.

4.7. LESSON END ACTIVITIES

i. In an organization of your choice, determine whether automated tools are used for requirements analysis.

ii. If automated tools are used, what are the good and bad experiences of the tools users?

iii. If automated tools are not used, why not? Is there any plan to experiment with automated tools? Why or why not?

4.8. POINTS FOR DISCUSSION

i. Obtain, from an organization of your choice, a Software Requirement Specifications. Assess the strengths ad weakness of the document in terms of the suggested format and contents.

4.9. REFERENCES

1. Pressman R., *Software Engineering: A practitioner's Approach,* (4th ed.), McGraw-Hill, 1997
2. Sommerville I.,*Software Engineering* (5th ed.), Addison-Wesley, 1996.
3. IEEE Std 830-1988, IEEE Recommended Practice for Software Requirements Specifications. IEEE Computer Society
4. www. Imappl.org/crest/environment.html.

LESSON 5: SOFTWARE DESIGN

Contents

5.0. AIMS AND OBJECTIVES

- To understand the concept of various Internal and External Design of software development process.
- To define procedural design and its concepts-structured programming, flow charts, looping constructs and box diagram.
- To understand the concepts of modular design, Functional independence, Cohesion and its types, Coupling and its types.

5.1. INTRODUCTION

Design is the first step in the development phase for any engineered product or system. The term design is used in two ways. Used as a verb, it represents the process of design. Used as a noun, it represents the result of the design process, which is the design for the system. The goal of design process is to produce a model or representation of a system, which can be used later to build that system. The produced model is called the *design* of the system.

Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software. Each of the elements of the requirements analysis model provides information that is required to

create a design model. The flow of information during software design is illustrated in *Figure 5.1*.

Characteristics of a Good Design

The design should:

- Exhibit hierarchical representation of software components
- Be modular
- Contain district representation of data and procedure.
- Have modules exhibiting independent functional characteristics,
- Minimize the complexity of interface

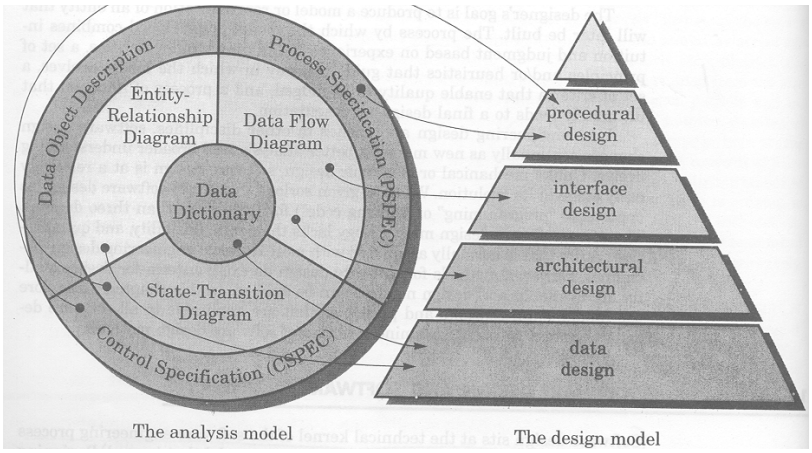


Figure 5.1. The Design Model

5.2. FUNDAMENTAL DESIGN CONCEPTS

Fundamental software design concepts provide the necessary framework for "*getting a program right*". A set of fundamental software design concepts has evolved over the years and each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each concept helps the software engineer to answer the following questions:

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

The fundamental design concepts are:

- **Abstraction** - allows designers to focus on solving a problem without being concerned about irrelevant lower level details (procedural

abstraction - named sequence of events, data abstraction - named collection of data objects)

- **Refinement** - process of elaboration where the designer provides successively more detail for each design component
- **Modularity** – Software is divided into different modules that are integrated. Since the monolithic software is difficult to grasp, we need to decompose them into modules. But care should be taken when modularising. Because at one point of time, recursive modularising will increase the total effort.
- **Software architecture** - overall structure of the software components and the ways in which that structure provides conceptual integrity for a system
- **Control hierarchy** or **program structure** - represents the module organization and implies a control hierarchy, but does not represent the procedural aspects of the software (e.g. event sequences)
- **Structural partitioning** - horizontal partitioning defines three partitions (input, data transformations, and output); vertical partitioning (factoring) distributes control in a top-down manner (control decisions in top level modules and processing work in the lower level modules).
- **Data structure** - representation of the logical relationship among individual data elements (requires at least as much attention as algorithm design)
- **Software procedure** - precise specification of processing (event sequences, decision points, repetitive operations, data organization/structure)

Information Hiding

The principle of *information hiding* suggests that modules be "characterized by design decisions that hide from all others". In other words, modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information. Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information that is necessary to achieve software function.

Procedural Design

Procedural design occurs after data, architectural, and interface designs have been established. The procedural specification required to define algorithmic details would be stated in a natural language like English. All members of a software development organization can understand this common natural language. But the disadvantage in using this natural language is that we can write a set of procedural steps in too many different ways. Because it is difficult to specify procedural detail unambiguously, a more constrained mode for representing procedural detail must be used.

5.3. MODULES AND MODULARIZATION CRITERIA

Architectural design has the goal of producing well-structured, modular software systems. In this section we consider a software module to be name entity having the following characteristics:

i. Modules contain instructions, processing logic, and data structures.

ii. Modules can be separately compiled and stored in a library.

iii. Modules can be included in a program.

iv. Module segments can be used by invoking a name and some parameters.

v. Modules can use other modules.

Examples of modules include procedures, subroutines, and functions; functional groups of related procedures, subroutines, and functions; data abstraction groups; utility groups and concurrent processes.

5.3.1. Software Architecture

Software architecture is the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. Architecture is the hierarchical structure of program components, the manner in which these components interact, and the structure of the data that are used by the components.

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to reuse design level concepts.

5.3.2. Control Hierarchy

Control hierarchy, also called *program structure*, represents the organization of program components and implies a hierarchy of control. It does not represent procedural aspects of software such as sequence of processes, occurrence/order of decisions, or repetition of operations.

A *tree-like diagram* is used to represent the hierarchy. *Depth* provides an indication of the number of levels of control. *Width* indicates the overall span of control. *Fan-out* is a measure of the number of modules that are directly controlled by another module. *Fan-in* indicates how many modules directly control a given module. The control relationship among modules is expressed in the following way: A module that controls another module is said to be *superordinate* to it; conversely, a module controlled by another is said to be *subordinate* to the controller.

The control hierarchy represents two subtly different characteristics of the software architecture: visibility, and connectivity. *Visibility* indicates the set of program components that may be invoked or used as data by a given component, even when this is accomplished indirectly. *Connectivity* indicates the set of components that are directly invoked or used as data by a given component.

5.3.3. Structural Partitioning

The program structure should be partitioned both horizontally and vertically.

Horizontal partitioning defines separate branches of the modular hierarchy for each major program function. The simplest approach to horizontal partitioning defines three partitions – input, data transformation, and output.

The benefits of horizontal partitioning are:

- Results in software that is easier to test
- Leads to software that is easier to maintain
- Results in propagation of fewer side effects
- Results in software that is easier to extend

The disadvantages of horizontal partitioning are:

- Causes more data to be passed across module interfaces and hence complicates the overall control of program flow.

Vertical partitioning, often called *factoring*, suggests that control and work should be distributed top-down in the program architecture. Top-level modules should perform control functions and relatively little processing work. Low-level modules should perform all input, computational, and output tasks.

The advantages of vertical partitioning are:

- Vertically partitioned architectures are less likely to be at risk to side effects when changes are made and hence more maintainable.

5.3.4. Data Structure

Data structure is a representation of the logical relationship among individual elements of data. Data structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information. The following are some classic data structures that form the building blocks for more sophisticated structures:

- **Scalar Item**: A scalar item represents a single element of information that may be addressed by an identifier.
- **Sequential Vector**: When scalar items are organized as a list or contiguous group, a sequential vector is formed.
- **Array**: When the sequential vector is extended to two, three, and even to arbitrary number of dimensions, an n-dimensional space is created. It is known as an array.
- **Linked List**: A linked list organizes noncontiguous scalar items, vectors, or spaces in a manner that enables them to be processed as a list.

Software Procedure

Software procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing,

including sequence of events, exact decision points, repetitive operations, and even data organization/structure.

5.4. COUPLING AND COHESION

A fundamental goal of software design is to structure the software product so that the number and complexity of interconnections between modules is minimized. An appealing set of heuristics for achieving this goal involves the concepts of coupling and cohesion.

5.4.1. Cohesion

A ***cohesive module*** performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. Cohesion is a measure of relative functional strength of a module as shown in fig. 5.2.,

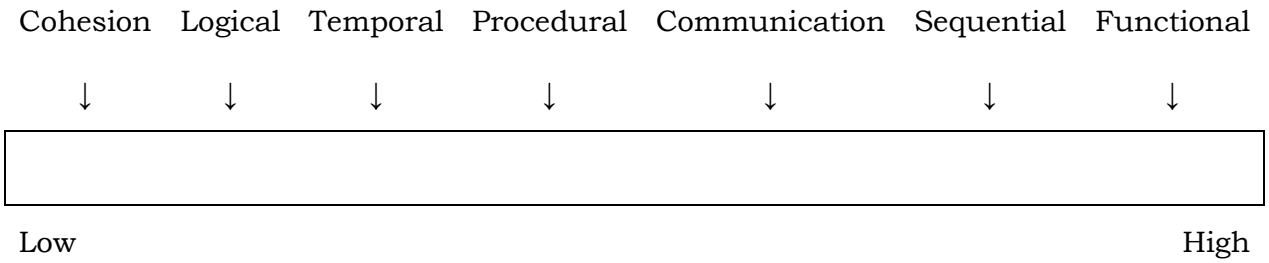


Fig. 5.2. Cohesion

A module that performs a set of tasks that relate to each other loosely, if at all, is termed ***coincidentally cohesive***. A module that performs tasks that are related logically is ***logically cohesive***. When a module contains tasks that are related by the fact that all must be executed within the same span of time, the module exhibits ***temporal cohesion***. When processing elements of a module are related and must be executed in a specific order, ***procedural cohesion*** exists. When all processing elements concentrate on one area of a data structure, ***communicational cohesion*** is present. It is unnecessary to determine the precise level of cohesion. Rather, ***it is important to strive for high cohesion.***

5.4.2. Coupling

Coupling is a measure of the interconnection among modules in a program structure. In software design, one must ***strive for low coupling***. Simple connectivity among modules results in software that is easier to understand and less liable to a "ripple effect" caused when errors occur at one location and propagate through a system. See Fig. 5.3.

No direct	Data coupling	Stamp coupling	Control coupling	External coupling	Common coupling	Content coupling

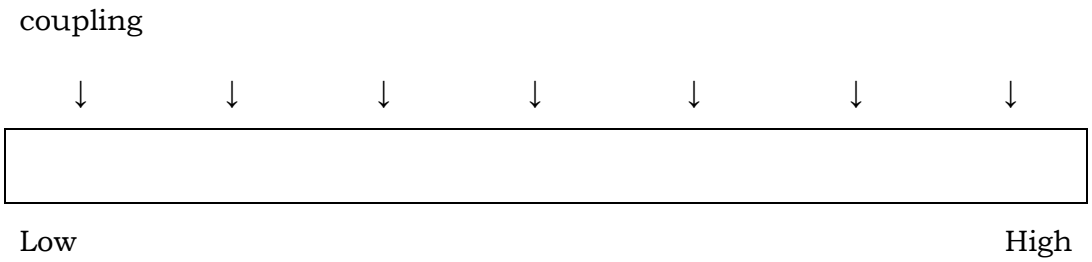


Fig. 5.3. Coupling

When there are different modules and each is unrelated, there exists ***no direct coupling***. When a simple data is passed from one module to another, and a one-to-one correspondence of items exists, ***data coupling*** is said to exist. ***Stamp coupling*** is found, when a portion of a data structure is passed via a module interface. ***Control coupling*** exists when a control flag is passed between modules. A control flag is a variable that controls decisions in a subordinate or superordinate module. When modules are tied to an environment external to the software, ***external coupling*** exists. However this form of coupling must be limited to a small number of modules with a structure. When a number of modules reference a global data area, ***common coupling*** occurs. Diagnosing problems in structures with considerable common coupling is time-consuming and difficult. ***Content coupling*** occurs, when one module makes use of data or control information maintained within the boundary of another module. Secondarily, content coupling occurs when branches are made into the middle of a module. This form of coupling should be avoided.

Data Design

Data design is the first of four design activities that are conducted during the software development. The data structure has a considerable impact on the program structure and procedural complexity. Hence data design is said to have a profound influence on software quality. The concepts of information hiding and data abstraction provide the foundation for an approach to data design.

The process of data design as summarized by Wasserman:

The primary activity during data design is to select logical representations of data objects (data structures) identified during the requirements definition and specification phase. The selection process may involve algorithmic analysis of alternative structures in order to determine the most efficient design or may simply involve the use of a set of modules that provide the desired operations upon some representation of an object. An important related activity during design is to identify those program modules that must operate directly upon the logical data structures. In this way the scope of effect of individual data design decisions can be constrained.

Wasserman has proposed a set of principles that may be used to specify and design data. The set of principles for data specification are:

1. The systematic analysis principles applied to function and behavior should also be applied to data.
2. All data structures and the operations to be performed on each should be identified.
3. A data dictionary should be established and used to define both data and program design.
4. Low-level data design decisions should be deferred until late in the design process.
5. The representation of data structures should be known only to those modules that must make direct use of the data contained within the structure.
6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract data types.

5.5. EFFECTIVE MODULAR DESIGN

A *modular design* reduces complexity, facilitates change, and results in an easier implementation by encouraging parallel development of different parts of a system. Software with effective modularity is easier to develop because function may be compartmentalized and interfaces are simplified. Independent modules are easier to maintain.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Independence is measured using two qualitative criteria: cohesion and coupling. *Cohesion* is a measure of the relative functional strength of a module. *Coupling* is a measure of the relative interdependence among modules in a program structure. One must always strive for high cohesion and low coupling.

5.5.1. Design Heuristics for Effective Modular Design

The program architecture is manipulated according to a set of heuristics given below:

1. ***Evaluate the "first iteration" of the program structure to reduce coupling and improve cohesion.***

   Once program structure has been developed, in order to improve module independence, modules may be exploded or imploded. An *exploded module* becomes two or more modules in the final program structure. An *imploded module* is the result of combining the processing implied by two or more modules. An exploded module leads to a more cohesive design, which is always sought after. When high coupling is expected, modules can sometimes be imploded.

2. ***Attempt to minimize structures with high fan-out; strive for fan-in as depth increases.***

In some structures (such as the one in Figure 5.4), a single module controls too many modules that are subordinate to it.

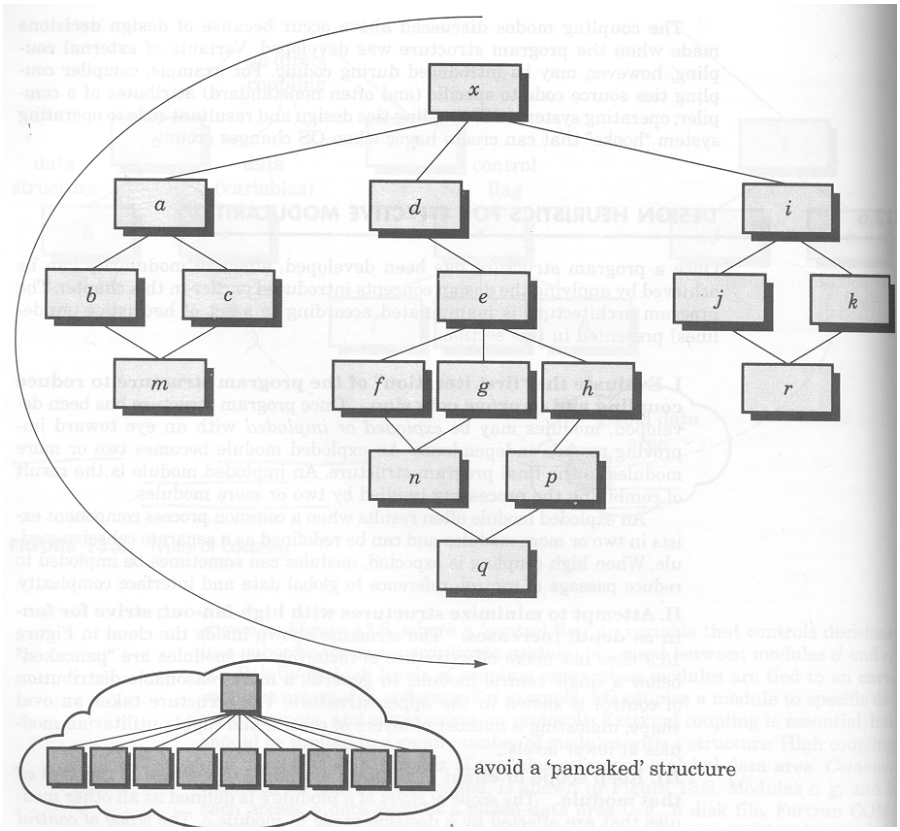


Figure 5.4.: Program Structures

Here, all the modules are "pancaked" below a single control module. This feature indicates that the structure does not make effective use of factoring. The fan-out value is high in these structures. To have an effective modular design it is better to minimize such high fan-out and strive for high fan-in. A structure with high fan-in indicates a number of layers of control and highly utilitarian modules at the lower levels.

3. ***Keep scope of effect of a module within the scope of control of that module.***

The *scope of effect* of a module *m* is defined as all other nodules that are affected by a decision made in module *m*. The *scope of control* of module *m* is all modules that are subordinate and ultimately subordinate to *m*. For example, in Fig.5.4. if module *e* makes a decision that affects module *r*, then it is said to be a violation of Heuristic 3, because module *r* lies outside the scope of control of module *e*.

4. ***Evaluate module interfaces to reduce complexity and redundancy and improve consistency.***

   Module interface complexity is a prime cause of software errors. Interface inconsistency is an indication of low cohesion.

5. ***Define modules whose function is predictable, but avoid modules that are overly restrictive.***

   A module is "predictable" when it can be treated as a black box; that is, the same external data will be produced regardless of internal processing details. Modules that have internal "memory" can be unpredictable unless care is taken in their use.

6. ***Strive for "controlled entry" modules, avoiding "pathological connections".***

   Pathological connection refers to branches or references into the middle of a module. Software is easier to understand and to maintain when modules interfaced are constrained and controlled.

7. ***Package software based on design constraints and portability requirements.***

   Packaging are to the techniques used to assemble software for a specific processing environment. When a program is said to "overlay" itself in memory, the design structure may have to be reorganized to group modules by degree of repetition, frequency of access and interval between calls. Also "one-shot" modules may be separated in the structure so that they may be overlaid.

5.6. DESIGN NOTATIONS CONSIDERATIONS

Design notation, coupled with structured programming concepts, enables the designer to represent procedural detail in a manner that facilitates translation to code. The commonly used design notations are:

a) Structured Programming

b) Graphical Design Notation (GDN)

c) Tabular Design Notation (TDN)

d) Program Design Language (PDL)

a. Structured Programming

Structured programming is an important procedural design technique. The three constructs that are fundamental to structured programming are:

- **Sequence** – Sequence implements processing steps that are essential in the specification of any algorithm
- **Condition** – Condition provides the facility for selected processing based on some logical occurrence
- **Repetition** – Repetition provides for looping

Each construct has a predictable logical structure, allowing entry at the top, and exit at the bottom. This enables a reader to follow procedural flow more easily. The use of such structured constructs enhances readability, testability, and maintainability. The structured constructs are *logical chunks* that allow a reader to recognize procedural elements of a module rather than read the design or code line by line. Understanding is enhanced when readily recognizable logical forms are encountered. Also any program can be designed and implemented only using the three structured constructs.

b. Graphical Design Notation (GDN)

Graphical tools such as flowchart or box diagram provide excellent pictorial patterns that readily depict procedural detail. However if graphical tools are misused, the wrong picture may lead to the wrong software.

i. Data Flow Diagram (DFD)

A data flow diagram is a graphical technique that depicts information flow and the transforms that applier as data move from input to output. The basic model of DFD is shown in Fig.5.5 The data flow diagram may be used to represent system or software at any levels. A level 0 DFD is partitioned into several bubbles with interconnecting arrows. Each of the processes represented at level1 are sub functions of the over all system described in the context model

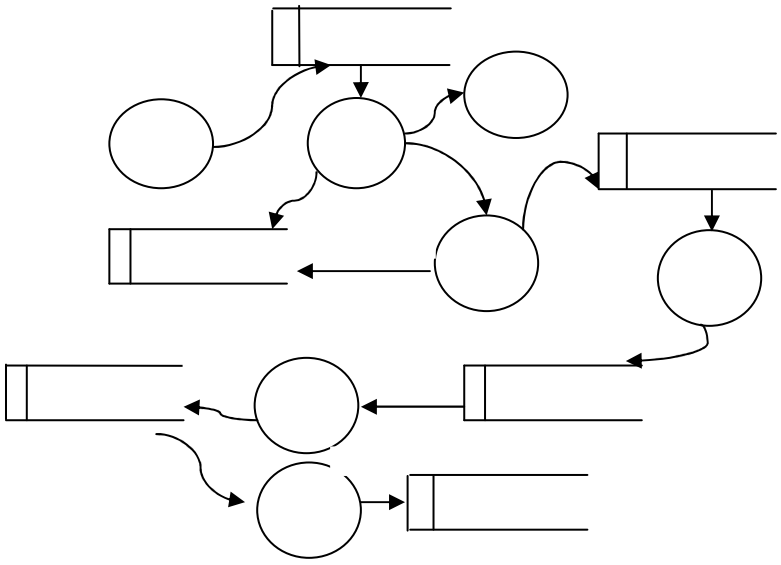


Figure 5.5: An Example of a Data Flow Diagram

ii. Flowchart

The *flowchart* is quite simple pictorially. The symbols used in a flowchart are:

- Box – to indicate a processing step
- Diamond – represents a logical condition
- Arrows – show the flow of control

The three structured constructs – sequence, condition, and repetition using the flowchart symbols are given in Figure 5.6.

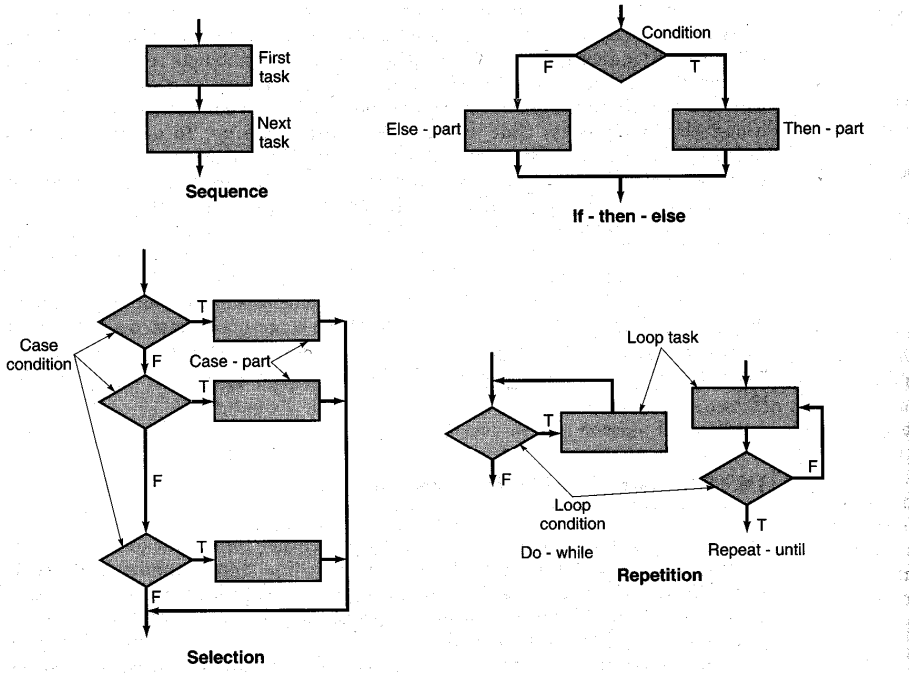


Figure 5.6: Flowchart Constructs

iii. Box Diagram

A *box diagram* has the following characteristics:

1. functional domain is well defined and clearly visible as a pictorial representation;
2. arbitrary transfer of control is impossible;
3. the scope of local and/or global data can be easily determined; and
4. Recursion is easy to represent.

The box diagram constructs are given in Figure 5.7.

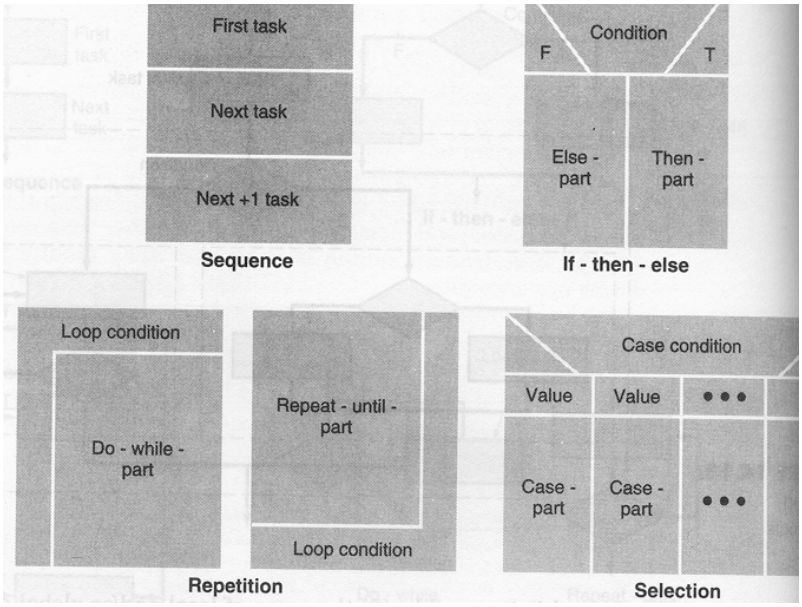


Figure 5.7. Box Diagram Constructs

c. Tabular Design Notation

Decision Tables provide a notation that translates actions and conditions into a tabular form. The table is difficult to misinterpret and may even be used as a machine readable input to a table driven algorithm. A decision table is divided into four sections. The upper left hand quadrant contains a list of all conditions. The lower left hand quadrant contains a list of all actions that are possible based on combinations of conditions. The right hand quadrants form a matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination. Therefore, each column of the matrix may be interpreted as a processing *rule*. The decision table nomenclature is given in Figure 5.8.

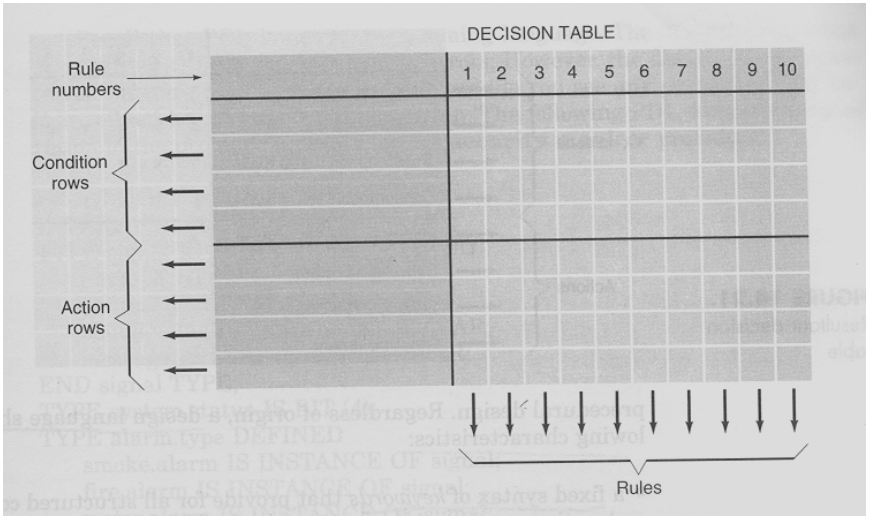


Figure 5.8. Decision Table nomenclature

The following steps are applied to develop a decision table:

1. List all actions that can be associated with a specific procedure.
2. List all conditions during execution of the procedure.
3. Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
4. Define rules by indicating what action or actions occur for a set of conditions.

d. Program Design Language

Program Design Language (PDL), also called *structured English* or *pseudocode*, is " a local language in that it uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e., a structured programming language)". PDL differs from modern programming languages in the use of narrative text embedded directly within PDL statements and hence PDL cannot be complied. However PDL processors can translate PDL into a graphical representation of design and produce nesting maps, a design operation index, cross-reference tables, and a variety of other information.

A design language should have the following features:

- A fixed syntax of *keywords* that provide for all structured constructs, data declarations, and modularity characteristics
- A free syntax of natural language that describes processing features
- Data declaration facilities that should include both simple and complex data structures, and
- Subprogram definition and calling techniques that support various modes of interface description.

A basic PDL syntax should include constructs for subprogram definition, interface description, and data declaration; and techniques for *block structuring*, condition constructs, repetition constructs, and I/O constructs.

5.7. DESIGN TECHNIQUES

The design process involves developing a conceptual view of the system, establishing system structure, identifying data streams and data stores, decomposing high level functions into sub-functions, establishing relationships and interconnections among components, developing concrete data representations, and specifying algorithmic details.

Design techniques are typically base on two approaches:

- Top-down design, and
- Bottom-up design.

5.7.1. Top-Down Design ([Dijkstra 1969], [Wirth 1971])

- The design activity must begin with the analysis of the requirements definition and should not consider implementation details at first.
- A project is decomposed into subprojects, and this procedure is repeated until the subtasks have become so simple that an algorithm can be formulated as a solution.
- Top-down design is a successive concretization of abstractly described ideas for a solution.
- Abstraction proves to be the best medium to render complex systems comprehensible; the designer is involved only with those aspects of the system that are necessary for understanding before the next step or during the search for a solution.

5.7.2. Bottom-up design

The fundamental idea:

- To perceive the hardware and layer above it as an abstract machine.

Technique:

- Bottom-up design begins with the givens of a concrete machine and successively develops one abstract machine after the other by adding needed attributes, until a machine has been achieved that provides the functionality that the user requires.

An abstract machine is a set of basic operations that permits the modeling of a system.

5.7.3. Design decomposition

- The design process is influenced not only by the design approach but also by the criteria used to decompose a system.
- Numerous decomposition principles have been proposed.

Classification of decomposition methods

1. *Function-oriented decomposition.* ([Wirth 1971], [Yourdon 1979]).
   - A function-oriented system perspective forms the core of the design.
   - Based on the functional requirements contained in the requirements definition, a task-oriented decomposition of the overall system takes place.
2. *Data-oriented decomposition.* ([Jackson 1975], [Warnier 1974], [Rechenberg 1984a])
   - The design process focuses on a data-oriented system perspective.
   - The design strategy orients itself to the data to be processed.

- The decomposition of the system stems from the analysis of the data.

3. *Object-oriented decomposition*. ([Abbott 1983], [Meyer 1988], [Wirfs-Brock 1989], [Coad 1990], [Booch 1991], [Rumbaugh 1991])
   - An object-oriented system perspective provides the focus of the design.
   - A software system is viewed as a collection of objects that communicate with one another. Each object has data structures that are invisible from outside and operations that can be executed on these structures.
   - The decomposition principle orients itself to the unity of data and operations and is based on Parnas' principle of information hiding [Parnas 1972] and the principle of inheritance derived from Dahl and Nygaard [Dahl 1966].

Check your progress

a. _____allows designers to focus on solving a problem without being concerned about irrelevant lower level details

b. _____process of elaboration where the designer provides successively more detail for each design component

Solution

a.

.....

b.

.....

5.7.4. Design Documentation

The document outlined below can be used as a template for a design specification.

I. Scope
   - A. System Objectives
   - B. Major software requirements
   - C. Design constraints, limitations

II. Data Design
   - A. Data objects and resultant data structures
   - B. File and database structures
     1. external file structure
        a. logical structure
        b. logical record description

c. access method

2. global data
3. file and data cross reference

III. Architectural Design
   A. Review of data and control flow
   B. Derived program structure

IV. Interface Design
   A. Human-machine interface specification
   B. Human-machine interface design rules
   C. External interface design
      1. Interfaces to external data
      2. Interfaces to external systems or devices
   D. Internal interface design rules

V. Procedural Design

For each module:

   A. Processing narrative
   B. Interface description
   C. Design language (or other) description
   D. Modules used
   E. Internal data structures
   F. Comments/restrictions/limitations

VI. Requirements Cross-Reference

VII. Test Provisions
   1. Test guidelines
   2. Integration strategy
   3. Special considerations

VIII. Special Notes

IX. Appendices

The numbered sections of the design specification are completed as the designer refines his or her representation of the software. The overall scope of the design effort is described in Section I. Section II presents the data design, describing the various entities that connect data objects to specific files. Section

III, the architectural design, indicates how the program architecture has been derived from the analysis model. Section IV describes the interface design, which includes the external and internal program interface and the human-machine interface. Section V explains the modules used and the processing narrative that explains the procedural function of each module. Section VI contains a requirements cross-reference, whose purpose is to establish that all requirements are satisfied by the software design and to indicate which modules are critical to the implementation of specific requirements. The first stage in the development of test documentation is contained in Section VII of the design document. It includes guidelines for testing and the requirements and considerations for software packaging. Section IX contains supplementary data. Algorithm descriptions, alternative procedures, tabular data, excerpts from other documents and other relevant information are presented as a special note or as a special appendix. A preliminary operations/installation manual may be developed and included as an appendix.

5.8. REVIEW QUESTIONS

1. Describe the architectural design of software.
2. Explain and discuss the need of user interface design.
3. What are the requirements of a software design?

5.9. LET US SUM UP

Every intellectual discipline is characterized by fundamental concepts and specific techniques. Techniques are the manifestations of the concepts as they apply to particular situations. Techniques come and go with changes in technology, intellectual dislikes, economic conditions and social concerns. By definition, fundamental principles remain the same throughout. They provide the underlying basis for development and evaluation of techniques. Fundamental concepts of software design include abstraction, structure, information hiding, modularity, concurrency, verification, and design aesthetics.

5.10. LESSON END ACTIVITIES

i. Use the structured design to produce a design for the systems described below:

ii. A system to manage patient's record to keep records of patient's number of visits to doctor, his/her progress, medicines prescribed in each visit etc.

5.11. POINTS TO DISCUSSION

i. What is meant by design for reuse? How it differs from Re-Engineering? Give the steps of reuse model widely use. Can a system ever be completely decoupled? Justify.

ii. Draw a top-level DFD for the system made up of the following

a. Selling customers request the organization to sell various items on their behalf. A record of these requests is kept.

b. Buying customers make requests to buy items.

c. A sale is arranged with a buyer if the item requested by the buyer has been previously put forward for sale by a selling customer.

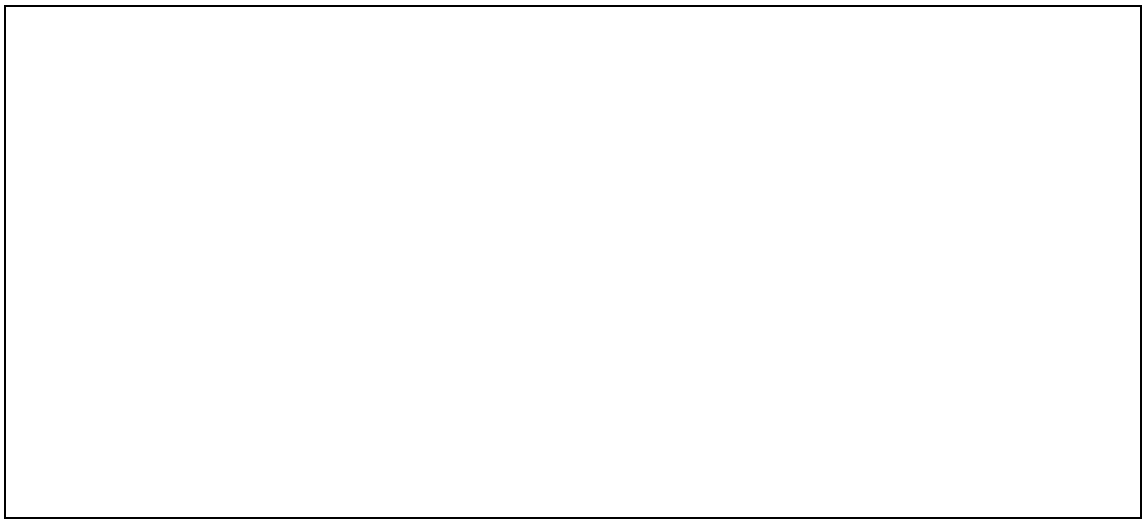
d. During a sale arrangement

i. An invoice is prepared for the buyer. A record or invoice is kept.

ii. A notification is sent to the seller whose item was sold the seller will now hold the item.

iii. A commission is computed and debited to the selling customer. This commission is subtracted from the amount sent to the selling custome3r and advice of the commission is given in the sale advice.

e. The sale is completed when payment is received from the buyer. A cheques is sent to the seller together with the dispatch request for items to be sent to the buyer.



5.12. REFERENCES

1. Dijkstra E. W., Structured Programming, Software Engineering Technique, *Report on a Conference*, Rome, 1969
2. Yourdon E., Constantine L., *Structured Design,* Prentice Hall, 1979
3. http://www.sei.cmu.edu/architecture/

LESSON 6: SOFTWARE IMPLEMENTATION

Contents

6.0. AIMS AND OBJECTIVES

- To write source code and internal documentation so that conformance of the code to its specifications can be easily verified.
- To understand psychological and the engineering view of a programming language characteristics
- To understand the connection between programming languages and the different areas in software engineering.
- To understand the coding style and efficiency of a programming language.

6.1. INTRODUCTION

The implementation phase of software development is concerned with translating design specifications into source code. The primary goal of implementation is to write source code and internal documentation so that

conformance of the code to its specifications can be easily verified, and so that debugging, testing, and modification are eased. This goal can be achieved by making the source code as clear and straightforward as possible. Simplicity, clarity and elegance are the hallmarks of good programs; unexplained cleverness and complexity are indications of inadequate design and misdirected thinking.

6.2. STRUCTURED CODING TECHNIQUES

The goal of structured coding is to sequencing control flow through a computer program so that the execution sequence follows the sequence in which the code is written. Linear flow of control can be achieved by restricting the set of allowed program constructs to single entry, single exit formats; however, strict loyalty to nested, single entry, single exit constructs leads to questions concerning efficiency, questions about "reasonable" violations of single entry, single exit, and questions about the proper role of the go to statements in structured coding.

Programming Language Characteristics

Programming languages are vehicle for communication between humans and computers. The coding process communication through a programming language is a human activity.

The Psychological characteristics of a language have an important impact on the quality of communication. The coding process may also be viewed as one step in the S/W development project.

The technical characteristics of a language can influence the quality of design. The technical characteristics can affect both human and S/W engineering concerns.

A Psychological View

A number of psychological characteristics occur as a result of programming language design. Although these characteristics are not measurable in any way, their manifestations in all the programming languages are recognized. The different characteristics are,

- Uniformity – The indicates the degree to which a language use consistent notation applies arbitrary restrictions and supports syntactic or semantic exceptions to the rule.
- Ambiguity – This is a programming language that is perceived by the programmer. A compiler will always interpret a statement in one way but the human reader may interpret the statement differently.
- Compactness – A programming language is an indication of the amount of code-oriented information that must be recalled from the human memory. The characteristics of human memory have a strong impact on the manner in which a language is used. Human memory and recognition may be divided into synthetic and sequential domains. Synesthetic memory allows remembering and recognizing the things as a whole. Sequential memory provides a means for recalling the next

element in a sequence. Each of these; memory characteristics affect programming language characteristics that are called Locality and linearity.

  - Locality- Is the synthetic characteristic of a programming language. Locality is enhanced when statements may be combines into blocks when the structured constructs may be implemented directly, and when design and resultant code are modular and cohesive.
  - Linearity – Is a psychological characteristic that is closely associated with the concept of maintenance of functional domain. Extensive branching violates the linearity of processing.

- Tradition- A software engineer with a background in FORTRAN would have little difficulty learning PASCAL or C. The latter languages have a tradition established by the former. The constructs are similar, the form is compatible and a sense of programming language format is maintained. However if the same S/W engineer is required to earn APL, LISP or Small talk, tradition would be broken and time on the learning curve would be longer.

The Psychological characteristics of a programming language have an important bearing on our ability to learn apply and maintain them.

A Syntactic / Semantic Model

When a programmer applies S/W engineering methods that are programming language-independent, semantic knowledge is used. Syntactic knowledge on other hand is language dependent, concentrating on the characteristics of a specific language.

The semantic knowledge is the more difficult to acquire and also more intellectually demanding to apply. All S/W engineering steps that precede coding make heavy use of semantic knowledge. The coding step applies syntactic knowledge that is arbitrary and instructional. When a new programming language is learn, the new syntactic information is added to memory. Potential confusion may occur when the syntax of a new programming language is similar but not equivalent to the syntax of another language.

An Engineering View

A S/W engineering view of programming language characteristics focuses on the needs of specific S/W development project. The characteristics are,

- Ease of design to code translation.
- Compiler efficiency.
- Source code portability.
- Availability of development tools.
- Availability of libraries

- Company policy
- External requirements
- Maintainability.

The **ease of design** to code translation provides an indication of how closely a programming language reflects a design representation. A language that directly implements the structured constructs, sophisticated data structures, specialized I/O, bit manipulation capabilities and object oriented constructs will make translation from design to source code much easier.

The ***quality of the compiler*** is important for the actual implementation phase. A good compiler should not only generate efficient code, but also provide support for debugging (e.g. with clear error messages and run-time checks). Although fast advances in processor speed and memory density have begun to satisfy the need for super efficient code may applications still require fast and low memory requirement programs. Particularly in the area of microcomputers, many compilers have been integrated in development systems. Here the user-friendliness of such systems must also be considered.

The **source code portability** can be as follows:

- The source code may be transported from processor to processor and compiler to compiler with little or no modification.
- The source code remains unchanged even when its environment changes.
- The source code may be integrated into different software packages with little or not modification required because of programming language characteristics.

The **availability of development tools** can shorten the time required to generate source code and can improve the quality of code. Many programming languages may e acquired with a set of tools that include debugging compilers, source code formatting aids, built-in editing facilities, tools for source code control, extensive subprogram libraries, browsers, cross-compilers for microprocessor development, microprocessor capabilities and others.

With modular programming languages, the ***availability of libraries*** for various application domains represents a significant selection criterion. For example, for practically all FORTRAN compilers, libraries are available with numerous mathematical functions, and Smalltalk class libraries contain a multitude of general classes for constructing interactive programs. The availability of libraries can also be used in C or Modular-2 if the compiler supports linking routines from different languages. On the other hand, there are libraries that are available only in compiled form and usable only in connection with a certain compiler.

Often a particular ***company policy*** influences the choice of a programming language. Frequently the language decision is made not by the implementers, but by managers that want to use only a single language company-wide for reasons of uniformity. Such a decision was made by the U.S. Department of Defense, which mandates the use of ADA for all programming in the military

sector in the U.S (and thus also in most other NATO countries). Such global decisions have also been made in the area of telecommunications, where many programmers at distributed locations work over decades on the same product.

Even in-house situations, such as the education of the employees or a module library built up over years, can force the choice of a certain language. A company might resist switching to a more modem programming language to avoid training costs, the purchase of new tools, and the re-implementation of existing software.

Sometimes ***external requirements*** force the use of a given programming language. Contracts for the European Union increasingly prescribe ADA, and the field of automation tends to require programs in FORTRAN or C. Such requirements arise when the client's interests extend beyond the finished product in compiled form and the client plans to do maintenance work and further development in an in-house software department. Then the education level of the client's programming team determines the implementation language.

Maintainability of source code is important for all nontrivial software development efforts. Maintenance cannot be accomplished until S/W is understood. The earlier elements of the S/W configuration provide a foundation for understanding, but the source code must be real and modified according to the changes in design. The self-documenting characteristics of a language have a strong influence on maintainability.

6.3. CODING STYLE

The readability of a program depends on the programming language used and on the programming style of the implementer. Writing readable programs is a creative process. Once the source code has been generated the function of a module should be apparent without reference to a design specification. The programming style of the implementer influences the readability of a program much more than the programming language used. A stylistically well-written FORTRAN or COBOL program can be more readable than a poorly written Modula-2 or Smalltlak program. Coding style encompasses a coding philosophy that stresses simplicity and clarity. The elements of style include the following:

i. Structuredness

ii. Expressiveness

iii. Data declaration

iv. Statement construction

v. Input/Output

vi. Outward form

vii. Efficiency

This refers to both the *design* and the *implementation*.

Although efficiency is an important quality attribute, we do not deal with questions of efficiency here. Only when we understood a problem and its solution correctly does it make sense to examine efficiency.

6.3.1. Structuredness

- Decomposing a software system with the goal of mastering its complexity through abstraction and striving for comprehensibility (*Structuring in the large.*)
- Selecting appropriate program components in the algorithmic formulation of subsolutions (*Structuring in the small.*)

a. Structuring in the large

- Classes and methods for object-oriented decomposition
- Modules and procedures assigned to the modules.
- During implementation the components defined in the design must be realized in such a way that they are executable on a computer. The medium for this translation process is the programming language.
- For the implementation of a software system, it is important that the decomposition defined in the design be expressible in the programming language; i.e. that all components can be represented in the chosen language.

b. Structuring in the small

- The understanding and testing of algorithms require that the algorithms be easy to read.
- Many complexity problems ensure from the freedom taken in the use of GOTO statements, i.e., from the design of unlimited control flow structures.
- The fundamental ideal behind structured programming is to use only control flow structures with one input and one output in the formulation of algorithms. This leads to a correspondence between the static written form of an algorithm and its dynamic behavior. What the source code lists sequentially tends to be executed chronologically. This makes algorithms comprehensible and easier to verify, modify or extend.
- Every algorithm can be represented by a combination of the control elements sequence, branch and loop (all of which have one input and one output) ([Böhm 1996]).
- D-diagrams (named after Dijkstra): Diagrams for algorithm consisting of only elements sequence, branch and loop.

  Note: *If we describe algorithms by a combination of these elements, no GOTO statement is necessary. However, programming without GOTO statements alone cannot guarantee structuredness. Choosing inappropriate program components produces poorly structured programs even if they contain no GOTO statements.*

6.3.2. Expressive power (Code Documentation)

- The implementation of software systems encompasses the naming of objects and the description of actions that manipulate these objects.

- The choice of names becomes particularly important in writing an algorithm.

Recommendation:

- Choose expressive names, even at the price of long identifiers. The writing effort pays off each time the program must be read, particular when it is to be corrected and extended long after its implementation. For local identifiers (where their declaration and use adjoin) shorted names suffice.
- If you use abbreviations, then use only ones that the reader of the program can understand without any explanation. Use abbreviations only in such a way as to be consistent with the context.
- Within software system assign names in only one language (e.g. do not mix English and Vietnamese).
- Use upper and lower case to distinguish different kinds of identifiers (e.g., upper case first letter for data types, classes and modules; lower case for variables) and to make long names more readable

  (e.g. CheckInputValue).
- Use nouns for values, verbs for activities, and adjectives for conditions to make the meaning of identifiers clear (e.g., width, ReadKey and valid, respectively).
- Establish your own rules and follow them consistently.

- Good programming style also finds expression in the use of comments: they contribute to the readability of a program and are thus important program components. Correct commenting of programs is not easy and requires experience, creativity and the ability to express the message concisely and precisely.

The rules for writing comments:

- Every system component (every module and every class) should begin with a detailed comment that gives the reader information about several questions regarding the system component:
  - What does the component do?
  - How (in what context) is the component used?
  - Which specialized methods, etc. are use?
  - Who is the author?
  - When was the component written?
  - Which modifications has have been make?

Example:

```
/* FUZZY SET CLASS: FSET
   FSET.CPP     2.0,     5 Sept. 2007
```

Lists operations on fuzzy sets

Written by Suresh Babu

*/

- Every procedure and method should be provided with a comment that describes its task (and possibly how it works). This applies particularly for the interface specifications.
- Explain the meaning of variables with a comment.
- Program components that are responsible for distinct subtasks should be labeled with comments.
- Statements that are difficult to understand (e.g. in tricky procedures or program components that exploit features of a specific computer) should be described in comments so that the reader can easily understand them.
- A software system should contain comments that are as few and as concise as possible, but as many adequately detailed comments as necessary.
- Ensure that program modifications not only affect declarations and statements but also are reflected in updated comments. Incorrect comments are worse than none at all.

Note: *These rules have deliberately been kept general because there are no rules that apply uniformly to all software systems and every application domain. Commenting software systems is an art, just like the design and implementation of software systems.*

6.3.3. Data declaration

The style of the data description is established when code is generated. A number of relatively simple guidelines can be established to make data more understandable and maintenance simpler. The order of data declarations should be standardized even if the programming language has no mandatory requirements. Ordering makes attributes easier to find, expediting testing, debugging and maintenance. When multiple variable names are declared with a single statement, an alphabetical ordering of names is used. Similarly, labeled global data should be ordered alphabetically. If complex data structures are used in design, the commenting should be used to explain peculiarities that are present in a programming language implementation.

6.3.4. Statement Construction

The construction of software logical flow is established during design. The construction of individual statements, is apart of the coding step. Statement construction should abide by one overriding rule. Each statement should be simple and direct and code should not be complicated to effect efficiency. Many programming languages allow multiple statements per line. The spaces saving aspects of this feature are hardly justified by poor readability that results. Individual source coder statements can be simplified by,

- Avoiding the use of complicated conditional tests.

- Eliminating tests on negative conditions.
- Avoiding heavy nesting of loops or conditions
- Use of parenthesis to clarify logical or arithmetic expressions.

6.3.5. Input / Output

The style of input and output is established during S/W requirements analysis and design and not coding Input and output style will vary with the degree of human interaction. For batch oriented I/O, logical input organization, meaningful Input/Output error checking, good I/O error recovery and rational output report formats are desirable characteristics. For Interactive I/O, a simple guided input scheme, extensive error checking and recovery, human engineered output, and consistency of I/O format are the primary concerns.

6.3.6. Outward form

- *Beyond name selection and commenting, the readability of software systems also depends on its outward form.*

Recommended rules for the outward form of programs:

- For every program component, the declarations (of data types, constants, variables, etc.) should be distinctly separated form the statement section.
- The declaration sections should have a uniform structure when possible, e.g. using the following sequence: constant, data types, classes and modules, methods and procedures.
- The interface description (parameter lists for method and procedures) should separate input, output and input/output parameters.
- Keep comments and source code distinctly separate.
- The program structure should be emphasized with indentation.

6.3.7. Efficiency

In well engineered systems, there is a natural tendency to use critical resources efficiently. Processor cycles and memory locations are viewed as critical resources. Three area of Efficiency should be taken care when a programming language is developed. They are,

a) Code Efficiency

b) Memory Efficiency

c) Input /Output Efficiency

a. Code Efficiency: The efficiency of source code is directly connected to the efficiency of algorithms defined during detailed design. However, the coding style can affect the execution speed and memory requirement. The following guidelines can be applier when detail design is translated into code.

- Simplify arithmetic and logical expressions before committing to code.

- Carefully evaluated the nested loops.
- Avoid use of multi-dimensional arrays.
- Avoid the use of pointers and complete lists.
- Don't mix data types.

b. Memory Efficiency – Memory restrictions in the large machines like mainframes and workstation are a thing of the past. Low-cost memory provides a large physical address space and virtual memory management provides application software with an enormous logical address space. Memory efficiency for such environments cannot be elated to minimum memory used. Memory efficiency must take into account the paging characteristics of an operating system. Maintenance of functional domain through the structured constructs is an excellent method for reducing paging and thereby increases efficiency.

c. Input/Output Efficiency – Two classes of I/O should be considered when efficiency is discussed. They are,

- I/O directed to human (user),
- I/O directed to another device.

Input supplied by a user and output produced for a user is efficient when the information can be supplier or understood with an economy of intellectual effort.

Efficiency of I/O to other hardware is an extremely complicated topic and the following guidelines improve I/O efficiency.

- The number of I/O requests should be minimized.
- All I/O should be buffered to reduce communication overhead
- I/O to secondary memory devices should be blocked
- I/O to terminals and printers should recognize features of the device that could improve quality or speed.

6.4. STANDARDS AND GUIDELINES

Coding standards are specifications for a preferred coding style. Give a choice of ways to achieve an effect, a preferred way is specified. Coding standards are often viewed by programmers as mechanisms to constrain and devalue the programmer's creative problem solving skills. It is desirable that all programmers on a software project adopt similar coding styles so that code of uniform quality is produced. This does not mean that all programmers must think alike, or that they must slavishly implement all algorithms in exactly the same manner. Indeed, the individual style of each programmer on a project is identical even when right adherence to standards of programming style is observed.

A programming standard might specify items such as

1. Goto Statements will not be used.

2. The nesting depth of program constructs will not exceed five levels.
3. Subroutine length will not exceed 30 lines.

A Guideline rephrases these specifications in the following manner:

1. The use of goto statements should be avoided in normal circumstances.
2. The nesting depth of program constructs should be five or less in normal circumstances.
3. The number of executable statements in a subprogram should not exceed 30 in normal circumstances.
4. Departure from normal circumstances requires approval by the project leader.

6.5. DOCUMENTATION GUIDELINES

Computer software includes the source code for a system and all the supporting documents generated during analysis, design, implementation, testing, and maintenance of the system. Internal documentation includes standards prologues for compellation units and subprograms, the self documenting aspects of the source code, and the internal comments embedded in the source code. Program unit notebooks provide mechanisms for organizing the work activities and documentation efforts of individual programmers. This section describes some aspects of supporting documents, the use of program unit notebooks, and some guidelines for internal documentation of source code.

6.5.1. Supporting Documents

Requirements specifications, design documents, test plans, user's manuals, installation instructions, and maintenance reports are examples of supporting documents. These documents are the products that result from systematic development and maintenance of computer software.

6.5.2. Program Unit Notebooks

A program unit is a unit of source code that is developed and/or maintained by one person; that person is responsible for the unit. In well-designed system a program unit is a subprogram or group of subprograms that provide a well-defined function or form a well define subsystem. A program unit is also small enough, and modular enough, that it can be thoroughly tested in isolation by the programmer who develops or modifies it. Program unit notebooks are used by individual programmers to organize their work activities, and maintain the documentation for their program units.

6.5.3. Internal Documentation

Internal documentation consists of a standard introduction for each program unit and compilation unit, the self-documenting aspects of the source code, and the internal comments embedded in the executable portion of the code.

Check your progress

i. The implementation phase of software development is concerned with translating _____ into _____.

ii. Programming languages are vehicle for communication between _____ and _____

iii. The _____ is important for the actual implementation phase.

iv. _____ of source code is important for all nontrivial software development efforts.

v. The coding style refers to both the *design* and the *implementation*.

Solutions

i

.....

ii

.....

iii

.....

iv

.....

v

.....

6.6. PROGRAMMING ENVIRONMENTS

- The question of which is the "right" programming language has always been a favorite topic of discussion in programming circles.
- The choice of a programming language for the implementation of a project frequently plays an important role.
- In the ideal case, the design should be carried out without any knowledge of the later implementation language so that it can be implemented in any language.

Quality criteria for programming languages:

- Modularity
- Documentation value
- Data structures
- Control flow

- Efficiency
- Integrity
- Portability
- Dialog support
- Specialized language elements

➢ ***Modularity of a programming language***

- Degree to which it support modularization of programs.
- The decomposition of a large program into multiple modules is a prerequisite for carrying out large software projects.
- Without modularization, division of labor in the implementation becomes impossible. Monolithic programs become unmanageable: they are difficult to maintain and document and they impede implementation with long compile times.
- Languages such as standard Pascal (which does not support modules, but compare with Turbo Pascal and Modula-2) prove unsuitable for large projects.
- If a language supports the decomposition of a program into smaller units, there must also be assurance that the components work together. If a procedure is invoked from another module, there must be a check of whether the procedure actually exists and whether it is used correctly (i.e. whether the number of parameters and their data types are correct).
- Languages may have independent compilation (e.g. C and FORTRAN), where this check takes place only upon invocation at run time (if at all)
- Alternatively, languages may have separate compilation (e.g. Ada and Modula-2), where each module has an interface description that provides the basis for checking its proper use already at compile time.

➢ ***Documentation value of a programming language***

- Affects the readability and thus the maintainability of programs.
- The importance of the documentation value rises for large programs and for software that the client continues to develop.
- High documentation value results, among other things, from explicit interface specifications with separate compilation (e.g. in Ada and Modula-2). Likewise the use of keywords instead of special characters (e.g. **begin** . . . **end** in Pascal rather than {. . .} in C) has a positive effect on readability because the greater redundancy gives less cause for careless errors in reading. Since programs are generally written only once but read repeatedly, the minimum additional effort in writing pays off no more so than in the maintenance phase. Likewise the language's scoping rules influence the readability of programs.

- Extensive languages with numerous specialized functions (e.g. Ada) are difficult to grasp in all their details, thus encouraging misinterpretations. Languages of medium size and complexity (e.g. Pascal and Modula-2) harbor significantly less such danger.

➢ ***Data structures in the programming language***

- Primarily when complex data must be processed, the availability of data structures in the programming language plays an important role.
- Older languages such as FORTRAN, BASIC, and COBOL offer solely the possibility to combine multiple homogeneous elements in array or heterogeneous elements in structures.
- Recursive data structures are difficult to implement in these languages.
- Languages like C permit the declaration of pointers to data structures. This enables data structures of any complexity, and their scope and structure can change at run time. However, the drawback of these data structures is that they are open and permit unrestricted access.
- Primarily in large projects with multiple project teams, abstract data takes on particular meaning. Although abstract data structures can be emulated in any modular language, due to better readability, preference should be given to a language with its own elements supporting this concept.
- Object-oriented languages offer the feature of extensible abstract data types that permit the realization of complex software systems with elegance and little effort. For a flexible and extensible solution, object-oriented languages provide a particularly good option.

➢ **Structuring control flow in the programming language**

- Languages like BASIC and FORTRAN include variations of a GOTO statement, which programmers can employ to create unlimited and incomprehensible control flow structures. In Pascal and C the use of the GOTO statement is encumbered because the target must be declared explicitly. In Eiffel, Modula-2 and Smalltalk there is no GOTO statement at all, which forces better structuring of the control flow.
- In technical applications additional possibilities for control flow can play an important role. These include the handling of exceptions and interrupts as well as parallel processes and their synchronization mechanisms. Many programming languages (e.g., Ada and Eiffel) support several of these concepts and thus permit simpler program structures in certain cases.

  Notes: *Every deviation from sequential flow is difficult to understand and thus has a negative effect on readability.*

➢ ***Efficiency of a programming language***

- The *efficiency* of a programming language is often overrated as a criterion. For example, the programming language C bears the reputation that it

supports the writing of very efficient programs, while object-oriented languages are accused of inefficiency. However, there are few cases where a language is in principle particularly efficient or especially inefficient.

- Optimizing compilers often generate excellent code that an experienced Assembler programmer could hardly improve upon. For time-critical operations, it pays to use a faster machine or a better compiler rather than a "more efficient" programming language.

➢ ***Integrity of a programming language***

- The integrity of a programming language flow primarily from its readability and its mechanisms for type checking (even across module boundaries).
- Independent of the application domain, therefore, a language with static typing should be preferred. Static typing means that for each expression the compiler can determine which type it will have at run time. Additional integrity risks include type conversions (type casts) and pointer arithmetic operations, which are routine for programming in C, for example.
- Run-time checks are also important for integrity, especially during the development phase. These normally belong in the domain of the compiler.
- Mechanisms for formulating assertions (e.g. in Eiffel) and features for exception handling (e.g. in Eiffel and Ada) also contribute to the integrity of a programming language.

➢ ***Portability***

- *Portability* can be a significant criterion if a software product is destined for various hardware platforms. In such a situation it makes sense to select a standardized language such as Ada or C. However, this alone does not suffice to ensure portability. For any external modules belonging to the language also need to be standardized. This is a problem in the language Modula-2 because various compiler producers offers different module libraries.
- Beyond standardization, another criterion is the availability of compilers for the language on different computers. For example, developers of software for mainframes will find a FORTRAN compiler on practically every machine.

➢ ***Dialog support***

- For interactive programs the programming language must also provide *dialog support*. For example, FORTRAN and COBOL offer only line-oriented input and output;
- Highly interactive programs (that react to every key pressed) can thus be developed only with the help of specialized libraries.

- Some languages like BASIC and LOGO are particularly designed provide dialog support for the user (and with the programmer), making these languages better suited for such applications.
- Object-oriented programming languages also prove well suited to the development of interactive programs, especially with the availability of a corresponding class library or an application framework.
- For specialized tasks, *specialized language elements* can be decisive for the selection of a programming language. For technical applications, for example, the availability of complex number arithmetic (e.g. in COBOL) can be important. For mathematical problems, matrix operations (e.g. in APL) can simplify the task, and translation and character string operations are elegantly solved in SNOBOL. The lack of such specialized language elements can be compensated with library modules in modular languages.
- Object-oriented languages prove particular suited to extending the language scope.

6.7. TYPE CHECKING

A data type specifies a set of data objects and a set of permitted operations on objects of that type. Thus, objects of type "integer" comprise an implementation dependent range of integer values and a set of relational and arithmetic operators on literals and variable of integer type.

The purpose of data typing is to permit classification of objects according to intended usage, to allow the language translator to select storage representations for objects of different types, and, the case of strong type languages, to detect and prevent operations among object of different types.

Type checking refers to restrictions and limitations imposed on the ways in which data items can be manipulated by the program. Different languages improve different restrictions, reflecting the differing philosophies of various language designers. At least five levels of type checking can be distinguished:

Level 0: Typeless
Level 1: Automatic Type coercion
Level 2: Mixed Mode
Level 3: Pseudo-Strong Type Checking
Level 4: Strong Type checking

6.8. SCOPING RULES

A declaration associates an identifier with a program entity, such as a variable, a type, a subprogram, a formal parameter, or a record component. The region of source text over which a declaration has an effect is called the scope of the declaration. The scoping rules of a programming language dictate the manner in which identifiers can be defined and used by the programmer.

Scoping rules used in various programming languages include global scope, FORTRAN scope, nested scope, and restricted scope. Global scope is provided in BASIC and COBOL. All identifiers are known in all regions of a program. In

FORTRAN, identifiers are known throughout the containing program unit, but are not known outside the unit unless they appear in a COMMON statement or as actual parameters in a subprogram invocation.

6.9. CONCURRENCY MECHANISMS

Two or more segments of a program can be executing concurrently if the effect of executing the segments is independent of the order in which they are executed. We refer to these program segments as ***tasks*** or ***processes***. On multiple-processor machines, independent code segments can be executed simultaneously to achieve increased processing efficiency. On a single-processor machine, the execution of independent code segments can be interleaved to achieve processing efficiency: One task may be able to execute while another is waiting for an external event to occur or waiting for completion of an I/O operation.

The trend towards multiple-processor machines and the increasingly sophisticated applications of computers has resulted in higher-level language constructs for specifying concurrent tasks. Two fundamental problems in concurrent programming are synchronization of tasks so that information can be transferred between tasks, and prevention of simultaneous updating of data that are accessible to more than one task. These problems are referred to as the synchronization problem and the mutual exclusion problem.

There are three fundamental approaches to concurrent programming:

a. shared variables

b. asynchronous message passing,

c. synchronous message passing.

6.10. REVIEW QUESTIONS

1. What are the psychological characteristics that occur as a result of programming language design?
2. What is a syntactic and Semantic model?
3. Explain the different characteristics in the engineering view on the software development project.
4. What are the aspects to be followed in choosing a language?
5. How is the programming languages linked with software Engineering?
6. Explain the four areas in programming language fundamentals.
7. What is a good coding style? Explain the element in coding style
8. What are the different areas in which efficiency should be taken care? Explain.

6.11. LET US SUM UP

In software, detailed design is the implementation of module internals, design of data structures and algorithms, and coding in a programming language. This chapter connects the treatment of these issues in the software technology.

6.12. LESSON END ACTIVITIES

i. Give reasons to justify restricting the size of subprograms to between 5 and 25 executable statements.

ii. Give reasons to justify subprograms of less than 5 statements and subprograms of more than 25 statements.

6.13. POINTS FOR DISCUSSION

i. Give some reasons for using global variables rather than parameters.

ii. Describe several potential problems created by use of global variables.

iii. In what situations would you definitely not use global?

6.14. REFERENCES

1. Horowitz E. and Munsen J.B., An expansive view of Reusable software, *IEEE Trans. Software Eng.*, SE-10 (5), 1984
2. Shneiderman B., *Designing the User Interface,* Reading, Addison-Wesley, 1986
3. S.R. Schach, Practical Software Engineering, Irwin-Aksen, Homewood, III, 1992

LESSON 7: SOFTWARE QUALITY ASSURANCE

Contents

7.0. AIM AND OBJECTIVES

- To introduce the various aspects involved in delivering quality software.
- To introduce different quality standards and how they affect the software industry.

7.1. INTRODUCTION

The American Heritage Dictionary defines ***quality*** as "a characteristic or attribute of something". As an attribute of an item, quality refers to measurable characteristics when we examine an item based on its measurable characteristics, two kinds of quality may be encountered: *quality of design* and *quality of conformance*.

In software development, ***quality of design*** encompasses requirements, specifications, and the design of the system. ***Quality of conformance*** is an issue focused primarily on implementation. If the implementation follows the design and the resulting system meets its requirements and performance goals, conformance quality is high.

7.2. QUALITY ASSURANCE

Quality Assurance is a "planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements. It also consists of the auditing and reporting functions of management.

Need for Quality Assurance

- Human beings cannot work in error free manner
- Human beings are blind to their own errors
- Cost of fixing errors increases exponentially with time since their occurrence
- Systems Software itself is not bug free
- Customers should not find bugs when using software
- Post-release debugging is more expensive
- An organization must learn from its mistakes i.e. not repeat its mistakes
- Appropriate resources need to be allotted e.g. people with right skills
- Error detection activity itself is not totally error free
- A group of people cannot work together without standards, processes, guidelines, etc.,

The goal of quality assurance is to provide management with the data necessary to be informed about the product quality, thereby gaining insight and confidence that product quality is meeting its goals. If the data provided through quality assurance identify problems, it is management's responsibility to address the problems and apply the necessary resources to resolve the quality issues.

7.2.1. Cost of Quality

Cost of quality refers to the total cost of all efforts to achieve product/service quality, and includes all work to conformance to requirements, as well as all work resulting from nonconformance to requirements. ***Quality costs*** may be divided into costs associated with prevention, appraisal, and failure.

- ***Prevention costs*** include quality planning, Formal Technical Reviews, test equipment, training.
- ***Appraisal costs*** include activities to gain insight into product condition the first time through each process. Examples: in-process and inert-process inspection, equipment calibration and maintenance, testing.
- ***Failure costs*** are costs that would disappear if no defects appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs.
  - ***Internal failure costs*** include rework, repair, and failure mode analysis.
  - ***External failure costs*** include complaint resolution, product return and replacement, help line support, and warranty work.

7.2.2. Need for Software Quality

- **Competitive pressure:** Today's business is extremely competitive, and the software industry is no exception. Companies must continuously make and sustain improvements in cost and quality to remain in business.
- **Customer satisfaction:** Acquiring a new customer is far more expensive than retaining a current one. Further, few unsatisfied customers might complain but the vast majority simply takes their business elsewhere.
- **Management of change:** In an industry like software where new tools and methods arrive at a faster rate than it takes to train staff in their use, it is especially important that organizations fundamentally change their management styles and their workforce's attitudes so as to effect significant improvements.
- **Cost of defects:** Defects in software may lead to huge losses to the customer. In case of mission-critical systems, defects can cause serious harm to the end-user.

7.2.3. Software quality attributes

Software quality is a broad and important field of software engineering.
Software quality is addressed by standardization bodies:

ISO, ANSI, IEEE, etc.,

Software quality attributes (see Figure 7.1)

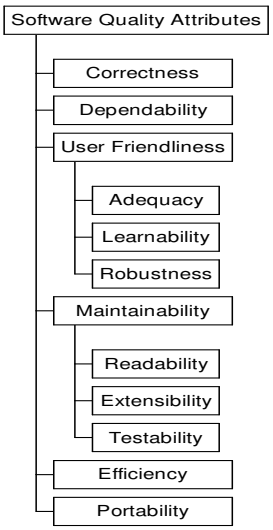


Figure 7.1 Software quality attributes

a. Correctness

The extent to which a program satisfies its specifications and fulfils its user's mission and objectives.

b. Reliability

Reliability of a software system derives from

- Correctness, and
- Availability.

The behavior over time for the fulfillment of a given specification depends on the reliability of the software system.

Reliability of a software system is defined as the probability that this system fulfills a function (determined by the specifications) for a specified number of input trials under specified input conditions in a specified time interval (assuming that hardware and input are free of errors).

A software system can be seen as reliable if this test produces a low *error rate* (i.e., the probability that an error will occur in a specified time interval.)

The error rate depends on the frequency of inputs and on the probability that an individual input will lead to an error.

c. User friendliness:

- Adequacy
- Learnability
- Robustness

Adequacy

Factors for the requirement of **Adequacy**:

1. The input required of the user should be limited to only what is necessary. The software system should expect information only if it is necessary for the functions that the user wishes to carry out. The software system should enable flexible data input on the part of the user and should carry out probable checks on the input. In dialog-driven software systems, we give particular importance in the uniformity, clarity and simplicity of the dialogs.
2. The performance offered by the software system should be adapted to the wishes of the user with the consideration given to extensibility; i.e., the functions should be limited to these in the specification.
3. The results produced by the software system:

   The results that a software system delivers should be output in a clear and well-structured form and be easy to interpret. The software system should afford the user flexibility with respect to the scope, the degree of detail, and the form of presentation of the results. Error messages must be provided in a form that is comprehensible for the user.

Learnability

Learnability of a software system depends on:

- The design of user interfaces
- The clarity and the simplicity of the user instructions (tutorial or user manual).

The user interface should present information as close to reality as possible and permit efficient utilization of the software's failures.

The user manual should be structured clearly and simply and be free of all dead weight. It should explain to the user what the software system should do, how the individual functions are activated, what relationships exist between functions, and which exceptions might arise and how they can be corrected. In addition, the user manual should serve as a reference that supports the user in quickly and comfortably finding the correct answers to questions.

Robustness (strong)

Robustness reduces the impact of operational mistakes, erroneous input data, and hardware errors.

A software system is robust if the consequences of an error in its operation, in the input, or in the hardware, in relation to a given application, are inversely proportional to the probability of the occurrence of this error in the given application.

- Frequent errors (e.g. erroneous commands, typing errors) must be handled with particular care
- Less frequent errors (e.g. power failure) can be handled more relaxly, but still must not lead to irreversible consequences.

d. Maintainability

The effort required to locate and fix an error or introduce new features in an operational program.

The maintainability of a software system depends on its:

- Readability
- Extensibility
- Testability

Readability

Readability of a software system depends on its:

- Form of representation
- Programming style

- Consistency
- Readability of the implementation programming languages
- Structuredness of the system
- Quality of the documentation
- Tools available for inspection

Extensibility

Extensibility allows required modifications at the appropriate locations to be made without undesirable side effects.

Extensibility of a software system depends on its:

- Structuredness (modularity) of the software system
- Possibilities that the implementation language provides for this purpose
- Readability (to find the appropriate location) of the code
- Availability of comprehensible program documentation

Testability

Testability: suitability for allowing the programmer to follow program execution (run-time behavior under given conditions) and for debugging.

The testability of a software system depends on its:

- Modularity
- Structuredness

Modular, well-structured programs prove more suitable for systematic, stepwise testing than monolithic, unstructured programs.

Testing tools and the possibility of formulating consistency conditions (assertions) in the source code reduce the testing effort and provide important prerequisites for the extensive, systematic testing of all system components.

e. Efficiency

Efficiency: ability of a software system to fulfill its purpose with the best possible utilization of all necessary resources (time, storage, transmission channels, and peripherals).

f. Portability

Portability: the ease with which a software system can be adapted to run on computers other than the one for which it was designed.

The portability of a software system depends on:

- Degree of hardware independence

- Implementation language
- Extent of exploitation of specialized system functions
- Hardware properties
- Structuredness: System-dependent elements are collected in easily interchangeable program components.

A software system can be said to be portable if the effort required for porting it proves significantly less than the effort necessary for a new implementation.

7.2.4. The importance of quality criteria

The quality requirements encompass all levels of software production.

Poor quality in intermediate products always proves detrimental to the quality of the final product.

- Quality attributes that affect the end product
- Quality attributes that affect intermediate products

Quality of *end products* [Bons 1982]:

- Quality attributes that affect their *application*: These influence the suitability of the product for its intended application (correctness, reliability and user friendliness).
- Quality attributes related to their *maintenance*: These affect the suitability of the product for functional modification and extensibility (readability, extensibility and testability).
- Quality attributes that influence their *portability*: These affect the suitability of the product for porting to another environment (portability and testability).

Quality attributes of *intermediate products*:

- Quality attributes that affect the transformation: These affect the suitability of an intermediate product for immediate transformation to a subsequent (high-quality) product (correctness, readability and testability).
- Quality attributes that affect the quality of the end product: These directly influence the quality of the end product (correctness, reliability, adequacy, readability, extensibility, testability, efficiency and portability).

7.2.5 The effects of quality criteria on each other

Attributes \ Effect on	Correctness	Dependability	Adequacy	Learnability	Robustness	Readability	Modifiability/extensibilit	Testability	Efficiency	Portability
Correctness		+	0	0	+	0	0	0	0	0
Dependability	0		0	0	+	0	0	0	-	0
Adequacy	0	0		+	0	0	0	0	+	-
Learnability	0	0	0		0	0	0	0	-	0
Robustness	0	+	+	0		0	0	+	-	0
Readability	+	+	0	0	+		+	+	-	+
Modifiability/extensibility	+	+	0	0	+	0		+	-	+
Testability	+	+	0	0	+	0	+		-	+
Efficiency	-	-	+	-	-	-	-	-		-
Portability	0	0	-	0	0	0	+	0	-	

Table 1.1 Mutual effects between quality criteria ("+": positive effect, "-": negative effect, "0": no effect)

7.2.6. Software Quality Assurance

- Refers to umbrella activities concerned with ensuing quality
- Covers quality activities associated with all line and staff functions.
- Scope not restricted only to fault detection and correction.

- Oriented towards both prevention and excellence.
- Involves taking care in design, production and servicing
- Assurance comes from:
    - Defining appropriate standards, procedures, guidelines, tools, and techniques.
    - Providing people with right skills, adequate supervision and guidance.

Software Quality Assurance is defined as:

- Conformance to explicitly stated functional and performance requirements,
- Conformance to explicitly documented development standards, and
- Conformance to implicit characteristics that are expected of all professionally developed software.

The above definition serves to emphasize three important points:

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3. There is a set of implicit requirements that often goes unmentioned. If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

Software Quality Assurance (SQA) Activities

The SQA group has responsibility for quality assurance planning, record keeping, analysis, and reporting. They assist the software engineering team in achieving a high quality end product.

The activities performed by an independent SQA group are:

1. Prepare a SQA Plan for a project

    The plan is developed during project planning and is reviewed by all interested parties. The plan identifies:

    - Evaluations to be performed
    - Audits and reviews to be performed
    - Standards that are applicable to the project
    - Procedures for error reporting

- Documents to be produced by the SQA group
- Amount of feedback provided to software project team

2. Participates in the development of the project's software process description.
3. Reviews software engineering activities to verify compliance with the defined software process.
4. Audits designated software work products to verify compliance with those defined as part of the software process.
5. Ensures that deviations in software work and work products are documented and handled according to a documented procedure.
6. Records any noncompliance and reports to senior management.
7. Coordinates the control and management of change.
8. Helps to collect and analyze software metrics.

7.3. WALKTHROUGHS AND INSPECTIONS

The walkthrough is a procedure that is commonly used to check the correctness of models produced by structured systems analysis, although its techniques are applicable to other design methodologies. Such checking has always been necessary in system life cycle. Walkthroughs differ from earlier methods in that they recommended a specific checking procedure and walkthrough team structure.

A Walkthrough team usually consists of a review and three to five reviewers. On one-or two-person project it may not be cost-effective to assemble a review team; however, the walkthrough technique can be beneficial with only one or two reviewers. In this case a walkthrough formalizes the processes of explaining your work to a colleague. The team must check that the model:

- ❖ Meets system objectives;
- ❖ Is a correct representation of the system;
- ❖ Has no omissions or ambiguities;
- ❖ Will do the job it is supposed to do; and
- ❖ Is easy to understand.

7.3.1. Software Reviews

Software reviews are a filter for the software engineering process. Reviews are applied at various points during software development to uncover errors, which can be removed. So Software reviews help to eliminate defects in the software work products that occur as a result of improper analysis, design,

and coding. Defect implies a quality problem that is discovered after the software has been shipped to end-users and so needs to be eliminated.

Any review employs the diversity of a group of people to

1. identify needed improvements in the product
2. conform those parts in which improvement is neither desired nor needed
3. achieve work of uniform, or at least predictable, quality to make technical work more manageable.

Reviews can either be formal or informal. Formal technical reviews are more effective from a quality assurance standpoint and effective means for improving software quality.

7.3.2. Formal Technical Reviews

A ***Formal Technical Review (FTR)*** is a software quality assurance activity that is performed by software engineers. The objectives of the FTR are:

1. To uncover errors in function, logic or implementation
2. To verify the software under review meets requirements
3. To ensure Software is as per predefined standards
4. To achieve uniform development of software
5. To make projects more manageable

The FTR is actually a class of reviews that include that include walkthroughs, inspections, round-robin reviews, and other small group technical assessment of software. Each FTR is conducted as a meeting, and will be successful only if it is properly planned, controlled and attended.

7.3.3. The Review Meeting

Every review meeting should focus on a specific part of the overall software. It should be:

1. Attended by 3-5 people
2. Should be well planned, but not require more than two hours of work per person
3. Of less than two hours duration

The focus of the FTR is on a ***work product*** – a component of the software. The individual who has developed the work product is the ***Producer***. The Producer informs the Project Leader that the work product is complete, and a review is required. The Project Leader contacts a Review Leader who evaluates the work product for readiness, generates copies, and distributes them to two

or three reviewers for advance preparation. Concurrently, the Review Leader also reviews the work product and establishes an agenda for the review meeting.

The Review Leader, all reviewers and the Producer attend the Review Meeting. The producer presents the work product and the reviewers raise issues if any. One of the reviewers takes on the role of recorder. The ***recorder*** records all important issues raised during the review, like problems and errors.

After the review, all attendees of the FTR must decide whether to

1. Accept the work product without modification,
2. Reject the work product due to severe errors, or
3. Accept the work product provisionally.

Once the decision is made, all FTR attendees complete a Sign-off, indicating their participation in the review, and their concurrence with the review team's findings.

During the FTR, it is important to summarize the issues and produce a Review Issues List, and a Review Summery Report. A Review Summery Report becomes the part of the Project Historical Record, and contains information about what was reviewed, who reviewed, and the findings and conclusions of the review. This report is distributed to the Project Leader, and other interested parties. The Review Issues List serves to identify problem areas within the product, and to serve as an action item checklist that guides the Producer, as corrections are made. It is important to establish a follow-up procedure to ensure that items on the issues list have been properly corrected.

A minimum set of guidelines for FTR is:

1. Review the product, not the Producer.
2. Set an agenda, and maintain it.
3. Limit arguments.
4. List out problem areas, but don't attempt to solve every problem noted.
5. Take written notes.
6. Limit the number of participants, and insist on advance preparation.
7. Develop a checklist for each work product that is likely to be reviewed.
8. Allocate resources and time schedules for FTRs.
9. Conduct meaningful training for all reviewers.
10. Review your earlier reviews.

7.3.4. Inspections

Inspections, like walkthroughs, can be used throughout the software life cycle to asses and improve the quality of the various work products. Inspection teams consist of one to four members (producer, inspector, moderator, reader) who are trained for their tasks. The ***producer***, whose product is under review, ***inspector*** who evaluates the product, and the ***moderator*** who controls the review process. There is also a ***reader***, who may guide inspectors through the product. Inspections are conducted in a similar manner to walkthrough, but more structure is imposed on the sessions, and each participant has a definite role to play. Each of the roles in an inspection team would have well-defined responsibilities within the inspection team. ***Fagan*** suggests a procedure made up of five steps, namely:

1. ***Overview***, where the producers of the work explain their work to inspectors.
2. ***Preparation***, where the inspectors prepare the work and the associated documentation for inspection.
3. ***Inspection***, which is meeting moderated by the moderator and guided by a reader who goes through the work with the inspectors.
4. ***Rework***, which is any work required by the producers to correct any deficiencies.
5. ***Follow-up***, where a check is made to ensure that any deficiencies have been corrected.

The important thing here is that the inspections are formal and have a report that must be acted on. It is also important that any recommendations made during inspections be acted upon and followed up to ensure that any deficiencies are corrected.

7.4. STATIC ANALYSIS

Static analysis is a technique for assessing the structural characteristics of source code, design specifications, or any notational representation that conforms to well define syntactic rules. The present discussion is restricted to static analysis of source code

Static program analysis

- *Static program analysis* seeks to detect errors without direct execution of the test object.
- The activities involved in static testing concern *syntactic, structural and semantic analysis of the test object.*
- The goal is to localize, as early as possible, error-prone parts of the test object.
- The most important activities of static program analysis are:
  - Code inspection

- Complexity analysis
- Structural analysis
- Data-flow analysis

a. Code inspection

- *Code inspection* is a useful technique for localizing design and implementation errors. Many errors prove easy to find if the author would only read the program carefully enough.
- The idea behind code inspection is to have the author of a program discuss it step by step with other software engineers.
- Structure of the four person team:

  1. An experienced software engineer who is not involved in the project to serve as moderator
  2. The designer of the test object
  3. The programmer responsible for the implementation
  4. The person responsible for testing

  The moderator notes every detected error and the inspection continues.
- The *task of the inspection team* is to *detect, not to correct,* errors. Only on completion of the inspection do the designer and the implementor begin their correction work.

b. Complexity analysis

- The *goal* of *complexity analysis* is to *establish metrics* for the complexity of a program These metrics include complexity measures for modules, nesting depths for loops, lengths of procedures and modules, import and use frequencies for modules, and the complexity of interfaces for procedures and methods.
- The results of complexity analysis permit statements about the quality of a software product (within certain limits) and localization of error-prone positions in the software system.
- Complexity is difficult to evaluate objectively; a program that a programmer perceives as simple might be viewed as complex by someone else.

c. Structural analysis

- The *goal* of *structural analysis* is to uncover structural anomalies of a test object.

c. Data-flow analysis

- Data-flow analysis is intended to help discover data-flow anomalies.

- Data-flow analysis provides information about whether a data object has a value before its use and whether a data object is used after an assignment.
- Data-flow analysis applies to both the body of a test object and the interfaces between test objects.

7.5. SYMBOLIC EXECUTION

Symbolic execution is a validation technique in which the input variables of a program unit are assigned symbolic values rather than literal values. A program is analyzed by transmitting the symbolic values of the inputs into the operands in expressions. The resulting symbolic expressions are simplified at each step in the computations so that all intermediate computations and decisions are always expressed in terms of the symbolic inputs. For instance, evaluation of an assignment statement results in association of a symbolic expression with the left-hand variable. When that variable is used in subsequent expressions, the current symbolic value is used. In this manner, all computations and decisions are expressed as symbolic values of the inputs.

Check your progress

i. _____ encompasses requirements, specifications, and the design of the system.

ii. _____ is an issue focused primarily on implementation.

iii. _____consists of the auditing and reporting functions of management.

iv. _____ refers to the total cost of all efforts to achieve product/service quality,

v. _____ reduces the impact of operational mistakes, erroneous input data, and hardware errors.

Solutions

I

.....

Ii

.....

Iii

.....

iv

.....

v

.....

7.6. REVIEW QUESTIONS

1. Explain software reliability in software quality assurance.
2. What are the SQA activities? Explain them in detail.
3. Explain in detail the formal technical review.
4. What are the Review guidelines and review checklist?
5. Explain the three S/W Quality metrics.
6. What are the formal approaches to SQA? Explain.
7. Explain a SQA approach

7.7. LET US SUM UP

This lesson gave a brief introduction to the kinds of activities that make up quality assurance. It stressed the importance of processes that are objective in order to find modeling errors in an organized way. This chapter outlined one such process, walkthroughs.

The discipline of software assurance that

1) defines the requirements for software controlled system fault/failure detection, isolation, and recovery;
2) reviews the software development processes and products for software error prevention and/ or controlled change to reduced functionality states; and
3) Defines the process for measuring and analyzing defects and defines/ derives the reliability and maintainability factors.

Software quality is important, if,

(1) explicitly define what is meant when you say 'software quality',

(2) create a set of activities that will help ensure that every software engineering work product exhibits high quality,

(3) perform quality assurance activities on every software project,

(4) use metrics to develop strategies for improving your software process, and as a consequence, improving the quality of the end product.

7.8. LEARNING ACTIVITIES

Conduct a structured walkthrough session on some segment of a software project of your choice. Include a moderator and a recording secretary plus two or three people who have knowledge off and interest in the project. Provide participants

7.9. POINTS FOR DISCUSSION

a) Compose a design inspection checklist for a software project of your choice. Apply checklist to the design specifications.

b) Design a code inspection checklist for a real-time software product

c) Design a code inspection checklist for a scientific application program.

d) Design a code inspection checklist for a data processing application program.

7.10. REFERENCES

1. Richard Fairley, "Software Engineering Concepts", Tata McGraw-Hill, 1997.
2. Roger S.Pressman, Software engineering- A practitioner's Approach, McGraw-Hill International Edition, 5th edition, 2001.
3. http://www.quality.org

LESSON 8: SOFTWARE TESTING

Contents

8.0. AIMS AND OBJECTIVES

- To understand the types of testing done on software
- To understand the different approaches to testing
- To do testing effectively by designing proper test cases

8.1. INTRODUCTION

In a software development project, errors can be injected at any stage during the development. Techniques are available for detecting and eliminating errors that originate in each phase. However, some requirement errors and design errors are likely to remain undetected. Such errors will ultimately be reflected in the code. Since code is the only product that can be executed and whose actual behavior can be observed, testing the code forms an important part of the software development activity.

8.1.1. Software testing process

Software testing is the process used to help identify the correctness, completeness, security and quality of developed computer software. With that in mind, testing can never completely establish the correctness of arbitrary computer software. In computability theory, a field of computer science and neat mathematical proof concludes that it is impossible to solve the halting problem, the question of whether an arbitrary computer program will enter an infinite loop, or halt and produce output. In other words, testing is nothing but **criticism** or **comparison** that is comparing the actual value with expected one.

Testing presents an interesting variance for the software engineer. The engineer creates a series a series of test cases that are intended to demolish the software that has been built. In fact, testing is the only activity in the software engineering process that could be viewed as "destructive" rather than "constructive". Testing performs a very critical role for quality assurance and for ensuring the reliability of the software.

Glen Meyers [1979] states a number of rules that can serve well as testing objectives:

1. Testing is the process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as-yet undiscovered error.
3. A successful test is one that uncovers an as-yet undiscovered error.

The common viewpoint of testing is that a successful test is one in which no errors are found. But from the above rules, it can be inferred that a successful test is one that systematically uncovers different classes of errors and that too with a minimum time and effort. The more the errors detected, the more successful is the test.

What testing can do?

1. It can uncover errors in the software
2. It can demonstrate that the software behaves according to the specification
3. It can show that the performance requirements have been met
4. It can prove to be a good indication of software reliability and software quality

What testing cannot do?

Testing cannot show the absence of defects, it can only show that software errors are present.

Davis [1995] suggests a set of **testing principles** as given below:

1. All tests should be traceable to the customer requirements.

   From the customer's point of view, when the program fails to meet requirements, it is considered to be a severe defect. Tests are to be designed to detect such defects.

2. Tests should be planned long before testing begins.

   It is commonly misunderstood that testing begins only after coding is complete. Testing is to be carried out all throughout the software development life cycle. Test planning can begin as soon as the requirements model is complete.

3. The Pareto principle applies to software testing.

   The Pareto principle implies that 80% of all errors uncovered during testing will likely be traceable to 20% of all program modules. Hence the idea is that those 20% suspect modules are to be isolated and thoroughly tested.

4. Testing should begin "in the small" and progress towards testing "in the large".

   Testing activity normally begins with the testing of individual program modules and then progresses towards integrated clusters (group) of modules, and ultimately the entire system.

5. Exhaustive testing is not possible.

   It is highly impossible to test all possible paths in a program because even for a moderately sized program, the number of path permutations is exceptionally large. However in practice, it is possible to adequately cover program logic.

6. To be most effective, testing should be conducted by an independent third party.

   By "most effective", we mean testing that has the highest probability of finding errors. In general, the software engineer who created the system is not the best person to conduct all tests for the software and hence an independent third party is the obvious choice.

Kaner, Falk, and Nguyen [1993] suggest the following attributes of a good test:

1. A good test has a high probability of finding an error.

2. A good test is not redundant (unnecessary).

   Since testing time and resources are limited, there is no point in conducting a test that has the same purpose as another test. For example, a library routine is developed to find the factorial of a given number. To test the routine in order to uncover errors, the test input

may be chosen to be 3. If the routine produces the correct result, then it is better to test for the input 50, a larger input when compared to 3, instead of testing with the input 4, because if the routine behaved correctly for the input 3 there is every possibility that it will also behave correctly for the input 4.

3. A good test should be the best of kind.

   In a group of tests that have a similar intent, the test that has the highest likelihood of uncovering a whole class of errors should be used. This is because the testing time and resources are limited.

4. A good test should be neither too simple nor too complex.

   Too simple tests may fail to uncover errors. At the same time, it is possible to combine a series of tests into one complex test, which may mask some errors. Hence each test should be executed separately.

In carrying out testing we will wish to address some or all of the following questions:

- How is functional validity tested?
- What *classes* of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?
- Which activities are necessary for the systematic testing of a software system?
- What is to be tested?
  - *System specifications,*
  - *Individual modules,*
  - *Connections between modules,*
  - *Integration of the modules in the overall system*
  - *Acceptance of the software product.*

8.1.2. Dynamic testing

- *For dynamic testing* the test objects are executed or simulated.
- Dynamic testing is an imperative process in the software life cycle. Every procedure, every module and class, every subsystem and the overall system must be tested dynamically, even if static tests and program verifications have been carried out.

- The activities for dynamic testing include:
  - Preparation of the test object for error localization
  - Availability of a test environment
  - Selection of appropriate test cases and data
  - Test execution and evaluation

There are many approaches to software testing, but effective testing of complex products is essentially a process of investigation, not merely a matter of creating and following rote procedure. One definition of testing is "the process of questioning a product in order to evaluate it", where the "questions" are things the tester tries to do with the product, and the product answers with its behavior in reaction to the questioning of the tester. Although most of the intellectual processes of testing are nearly identical to that of review or inspection, the word testing is imply as a consequence to mean the dynamic analysis of the product—putting the product through its paces. The quality of the application can, and normally does, vary widely from system to system but some of the common quality attributes include reliability, stability, portability, maintainability and usability.

Software testing answers questions that development testing and code reviews can't.

- Does it really work as expected?
- Does it meet the users' requirements?
- Is it what the users expect?
- Do the users like it?
- Is it compatible with our other systems?
- How does it perform?
- How does it scale when more users are added?
- Which areas need more work?
- Is it ready for release?

What can we do with the answers to these questions?

- Save time and money by identifying defects early
- Avoid or reduce development downtime
- Provide better customer service by building a better application
- Know that we've satisfied our users' requirements
- Build a list of desired modifications and enhancements for later versions

- Identify and catalog reusable modules and components
- Identify areas where programmers and developers need training

8.2. LEVELS OF TESTING

The different levels of testing are used to validate the software at different levels of the development process.

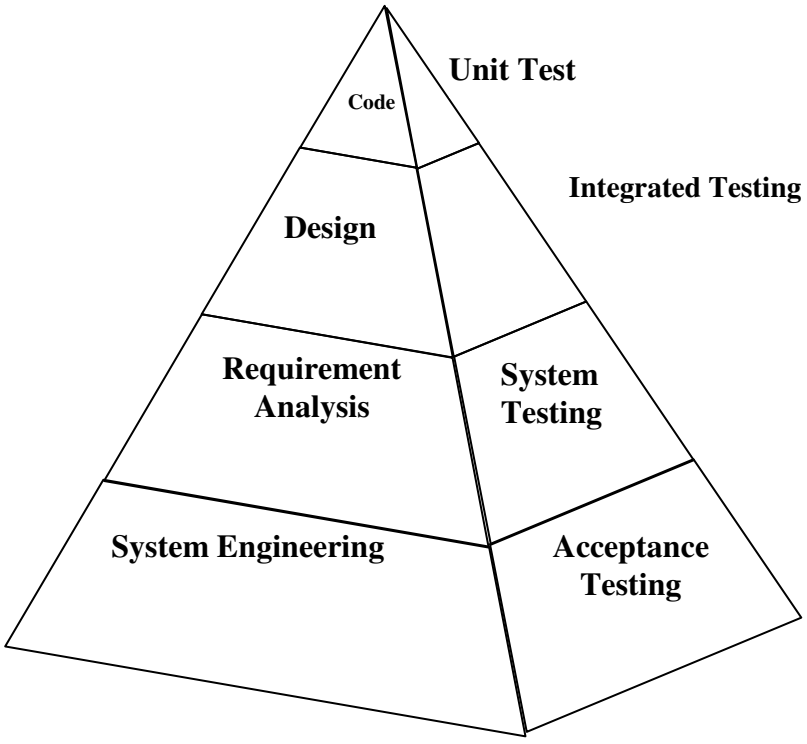


Fig. 8.1. Levels of testing

Fig. 8.1 shows the different testing phases and the corresponding development phases that it validates. Unit Testing is done to validate the code written and is usually done by the author of the code. Integration testing is done to validate the design strategies of the software. System testing is done to ensure that all the functional and non functional requirements of the software are met. Acceptance testing is then done by the customer to ensure that the software works well according to customer specification.

8.3. UNIT TESTING

Unit testing is essentially for verification of the code produced during the coding phase, and hence the goal is to test the internal logic of the modules. The unit test is normally ***white box*** oriented, and the step can be conducted in parallel for multiple modules. Unit testing is simplified when a module with high cohesion is designed. When only one function is addressed by a module, the number of test cases is reduced and errors can easily be uncovered.

8.3.1. Unit Test Considerations

The tests that occur as part of unit testing are listed below:

- **Interface** – The module interface is tested to ensure that information properly enters into and out of the program unit under test. If data does not enter or exit properly, then all other tests are unresolved.
- **Local data structures** – The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- **Boundary conditions** – Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- **Independent paths** – All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- **Error handling paths** – All error handling paths are tested.

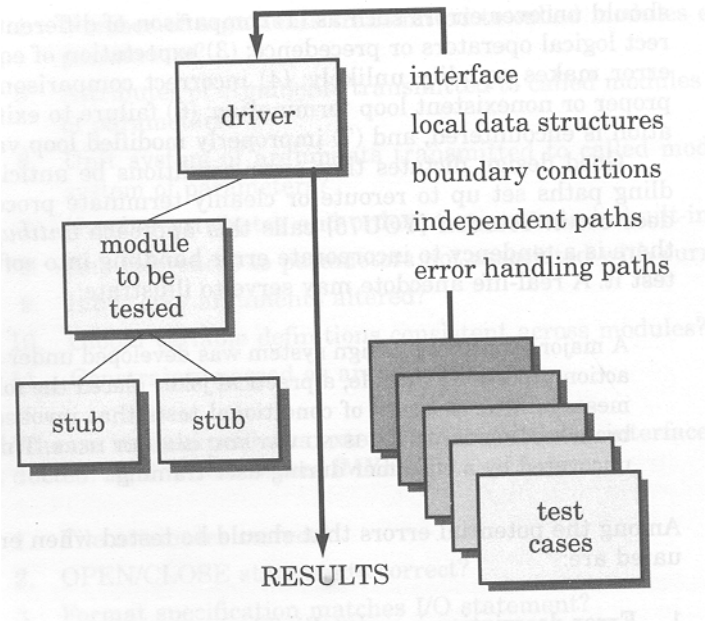


Figure 8.2. Unit Test Environment

8.3.2. Unit Test Procedures

Because a module is not a standalone program, driver and/or stub software must be developed for each unit test. A *driver* is nothing more than a "main program" that accepts test case data, passes such data to the module to be tested, and prints relevant results. *Stubs* serve to replace modules that are subordinate to the module to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints

verification of entry, and returns. Drivers and stubs represent overhead. Because, both are software modules that must be developed to aid in testing but not delivered with the final software product.

8.3.3. Integration Testing

Integration testing involves checking for errors when units are put together as described in the design specifications. While integrating, software can be thought of as a system consisting of several levels. A unit that makes a function call to another unit is considered to be one level above it. Fig. 8.3

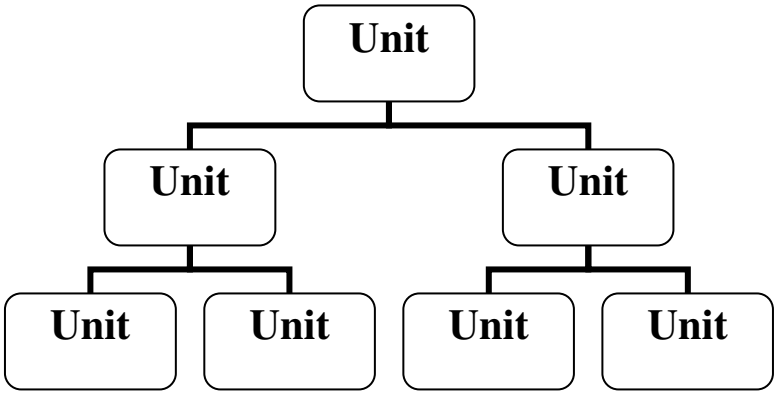


Fig. 8.3. Functional Unit

There are several approaches for integration:

a. Bottom-Up

The bottom-up approach integrates the units at the lowest level (bottom level) first, and then the units at the next level above it an so on till the topmost level is integrated. When integrating, each interface is tested to see if it works properly together.

Method

- First those operations are tested that require no other program components; then their integration to a module is tested.
- After the module test the integration of multiple (tested) modules to a subsystem is tested, until finally the integration of the subsystems, i.e., the overall system can be tested.

The advantages

- The advantages of bottom-up testing prove to be the drawbacks of top-down testing (and vice versa).
- The bottom-up test method is solid and proven. The objects to be tested are known in full detail. It is often simpler to define relevant test cases and test data.

- The bottom-up approach is psychologically more satisfying because the tester can be certain that the foundations for the test objects have been tested in full detail.

The drawbacks

- The characteristics of the finished product are only known after the completion of all implementation and testing, which means that design errors in the upper levels are detected very late.
- Testing individual levels also cause high costs for providing a suitable test environment.

b. Top-Down

Top-Down integration starts from the units at the top level first and works downwards integrating the units at a lower level. While integrating if a unit in the lower level is not available a replica of the lower level unit is created which imitates its behavior.

Method

- The control module is implemented and tested first.
- Imported modules are represented by substitute modules.
- Surrogates have the same interfaces as the imported modules and simulate their input/output behavior.
- After the test of the control module, all other modules of the software systems are tested in the same way; i.e., their operations are represented by surrogate procedures until the development has progressed enough to allow implementation and testing of the operations.
- The test advances stepwise with the implementation. Implementation and phases merge, and the integration test of subsystems become unnecessary.

The advantages

- Design errors are detected as early as possible, saving development time and costs because corrections in the module design can be made before their implementation.
- The characteristics of a software system are evident from the start, which enables a simple test of the development state and the acceptance by the user.
- The software system can be tested thoroughly from the start with test cases without providing (expensive) test environments.

The drawbacks

- Strict top-down testing proves extremely difficult because designing usable surrogate objects can prove very complicated, especially for complex operations.
- Errors in lower hierarchy levels are hard to localize.

c. Sandwich

Sandwich integration is an attempt to combine the advantages of both the above approaches. A "target" layer is identified somewhere in between and the integration converges on the layer using a top-down approach above it and a bottom-up approach below it. Identifying the target layers must be done by people with good experience in similar projects or else it might leads to serious delays.

d. Big-Bang

A different and somewhat simplistic approach is the big-bang approach, which consists of putting all unit-tested modules together and testing them in one go. Chances are that it will not work! This is not a very feasible approach as it will be very difficult to identify interfacing issues.

8.4. SYSTEM TESTING

System testing is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. Software is only one element of a larger computer-based system. The software developed is ultimately incorporated with other system elements such as new hardware, information etc., and a series of system integration and validation tests are conducted. These tests are not conducted by the software developer alone. System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that all system elements have been properly integrated and perform allocated functions.

System testing falls within the scope of Black box testing, and as such, should require no knowledge of the inner design of the code or logic.

8.4.1. Alpha and Beta Test

Alpha testing and Beta testing are sub-categories of System testing. If software is developed as a product (example: Microsoft Word) which is intended to be used by many end-users, it is not practical to perform formal acceptance tests with each end-user. In this situation most software products are tested using the process called alpha and beta testing to allow the end-user to find defects.

The ***Alpha test*** is conducted in the developer's environment by the end-users. The environment might be simulated, with the developer and the

typical end-user present for the testing. The end-user uses the software and records the errors and problems. Alpha test is conducted in a controlled environment.

The ***Beta test*** is conducted in the end-user's environment. The developer is not present for the beta testing. The beta testing is always in the real-world environment which is not controlled by the developer. The end-user records the problems and reports it back to the developer at intervals. Based on the results of the beta testing the software is made ready for the final release to the intended customer base.

As a rule, System testing takes, as its input, all of the "integrated" software components that have successfully passed Integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of Integration testing is to detect any inconsistencies between the software units that are integrated together called *assemblages* or between any of the *assemblages* and hardware. System testing is more of a limiting type of testing, where it seeks to detect both defects within the "inter-assemblages" and also the system as a whole.

8.4.2. Finger Pointing

A classic system testing problem is "finger pointing". This occurs when an error is uncovered, and each system element developer blames the other for the problem. The software engineer should anticipate potential interfacing problems and do the following:

1) Design error-handling paths that test all information coming from other elements of the system
2) Conduct a series of tests that simulate bad data or other potential errors at the software interface
3) Record the results of tests to use as "evidence" if finger pointing does occur
4) Participate in planning and design of system tests to ensure that software is adequately tested.

8.4.3. Types of System Tests

The *types of system tests* for software-based systems are:

a. Recovery Testing
b. Security Testing
c. Stress testing
d. Sensitivity Testing
e. Performance Testing

a. Recovery Testing

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatically performed by the system itself, then re-initialization, check pointing mechanisms, data recovery, and restart are each evaluated for correctness. If recovery requires human intervention, the *mean time to repair* is evaluated to determine whether it is within acceptable limits.

b. Security Testing

Security testing attempts to verify that protection mechanisms built into a system will in fact protect it from improper penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; unhappy employees who attempt to penetrate for revenge; and dishonest individuals who attempt to penetrate for illegal personal gain.

c. Stress Testing

Stress tests are designed to tackle programs with abnormal situations. Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example,

1) Special tests may be designed that generate 10 interrupts per second, when one or two is the average rate
2) Input data rates may be increased by an order of magnitude to determine how input functions will respond
3) Test cases that require maximum memory or other resources may be executed
4) Test cases that may cause thrashing in a virtual operating system may be designed
5) Test cases that may cause excessive hunting for disk resident data may be created.

d. Sensitivity Testing

A variation of stress testing is a technique called sensitivity testing. In some situations a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

e. Performance Testing

Performance testing is designed to test run time performance (speed and response time) of software within the context of an integrated system. It occurs throughout all steps in the testing process. Performance tests are often coupled with stress testing and often require both hardware and software

instrumentation. External instrumentation can monitor execution intervals, log events as they occur, and sample machine states on a regular basis. Performance testing can be categorized into the following:

- **Load Testing** is conducted to check whether the system is capable of handling an anticipated load. Here, Load refers to the number of concurrent user accessing the system. Load testing is used to determine whether the system is capable of handling various activities performed concurrently by different users.
- **Endurance testing** deals with the reliability of the system. This type of testing is conducted for a longer duration to find out the health of the system in terms of its consistency. Endurance testing is conducted on either a normal load, or stress load. However, the duration of the test is long.
- **Stress testing** helps to identify the number of users the system can handle at a time before breaking down or degrading severely. Stress testing goes one step beyond the load testing and identifies the system's capability to handle the peak load.
- **Spike testing** is conducted to stress the system suddenly for a short duration. This testing ensures whether the system will be a stable and responsive under an unexpected rise in load.

8.4.4. Regression Testing

This is an important aspect of testing-ensuring that when an error is fixed in a system, the new version of the system does not fail any test that the older version passed. Regression testing consists of running the corrected system against tests which the program had already passed successfully. This is to ensure that in the process of modifying the existing system, the original functionality of the system was not disturbed. This is particularly important in maintenance project, where the likelihood of making changes can inadvertently affect the program's behavior.

Maintenance projects require enhancement or updating of the existing system; enhancements are introduction of new features to the software and might be released in different versions. Whenever a version is released, regression testing should be done on the system to ensure that the existing features have not been disturbed.

8.5. ACCEPTANCE TEST

Acceptance testing is the process of testing the entire system, with the completed software as part of it. This is done to ensure that all the requirements that the customer specified are met. Acceptance testing (done after System testing) is similar to system testing but administered by the customer to test if the system follow to the agreed upon requirements.

- The development of a software product ends with the *inspection (acceptance test)* by the user.
- At inspection the software system is tested with *real data* under *real conditions of use.*
- The *goal* of the inspection is to *uncover all errors* that arose from such sources as misunderstandings in consultations between users and software developers, poor estimates of application-specific data quantities, and unrealistic assumptions about the real environment of the software system.

Any engineered product can be tested in one of two ways:

1. Black Box Testing
2. White Box Testing

1. **White Box Testing**: White box testing is done to ensure that internal operation performs according to the specification and all internal components have been adequately exercised.
2. **Black Box Testing**: Each product has been designed to perform certain specified functions. Tests can be conducted to demonstrate that each function is fully operational.

8.6. WHITE BOX TESTING

White box testing, **clear box testing**, **glass box testing** or **structural testing** is used in computer programming, software engineering and software testing to check that the outputs of a program, given certain inputs, conform to the structural specification of the program.

The term *white box* (or *glass box*) indicates that testing is done with knowledge of the code used to execute certain functionality. For this reason, a programmer is usually required to perform white box tests. Often, multiple programmers will write tests based on certain code, so as to gain varying perspectives on possible outcomes.

A complementary technique, black box testing or functional testing, performs testing based on previously understood requirements (or understood functionality), without knowledge of how the code executes.

- *White-box testing* is one of the most important test methods. For a limited number of program paths, which usually suffices in practice, the test permits correct manipulation of the data structures and examination of the input/output behavior of test objects.
- In white-box testing the test activities involve not only checking the input/output behavior, but also examination of the inner structure of the test object.

- The goal is to determine, for every possible path through the test object, the behavior of the test object in relation to the input data.
- Test cases are selected on the basis of knowledge of the control flow structure of the test object.
- The selection of test cases must consider the following:
  - Every module and function of the test object must be invoked at least once
  - Every branch must be taken at least once
  - As many paths as possible must be followed
- It is important to assure that every branch is actually taken. It does not suffice to merely assure that every statement is executed because errors in binary branches might not be found because not all branches were tested.
- It is important to consider that, even for well-structured programs, in practice it remains impossible to test all possible paths, i.e., all possible statement sequences of a program ([Pressman 1987], see Figure 9.4)

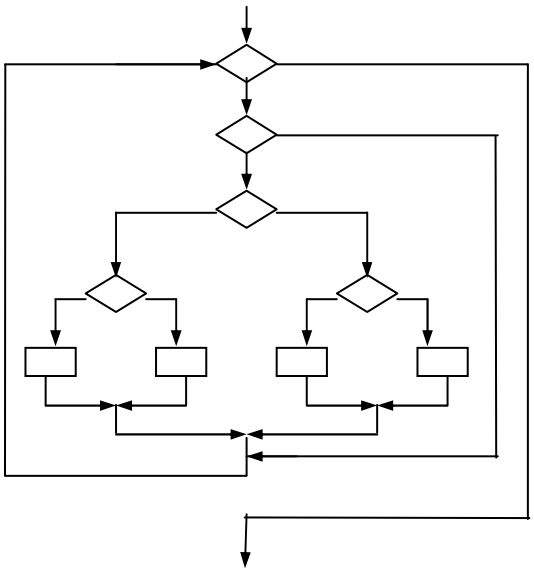


Figure 9.4. Control flowchart: for 10 iterations there are almost one million paths

It uses the control structure of the procedural design to derive test cases that can

1. guarantee that all *independent paths* within a module have been exercised at least once
2. exercise all *logical decisions* on their *true* and *false* sides

3. execute all *loops* at their boundaries and within their operational bounds
4. exercise *internal data structures* to assure their validity

The various White Box Testing techniques are:

a. Basis Path Testing

b. Condition Testing

c. Data flow Testing

d. Loop Testing

a. Basis Path Testing

The *basis path method* enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing. The steps to be followed in deriving test cases using this method are:

1. Using the design or code as a foundation, draw a corresponding flow graph.
2. Determine the cyclomatic complexity, V(G), of the resultant flow graph, using the formula

   V(G) = E – N + 2,

   where E is the number of edges, and N is the number of nodes in the flowgraph.
3. Determine a basis set of linearly independent paths. The value of V(G) provides the number of linearly independent paths through the program control structure.
4. Prepare test cases that will force execution of each path in the basis set.
5. Execute each test case and compare it with the expected results.

b. Condition Testing

Condition Testing is a test case design method that exercises the logical conditions contained in a program module. Conditions can be either simple or compound in nature.

- A ***simple condition*** is a Boolean variable or a relational expression, possibly preceded with a NOT operator.

A ***relational expression*** takes the form E_1 <relational operator> E_2, where E_1 and E_2 are arithmetic expressions and (relational operator) is one of the following: "<", "≤", ">", "≥", "=", "≠".

- A *compound condition* is composed of two simple conditions, Boolean operators, and parentheses.

A condition without relational expressions is referred to as a ***Boolean expression.***

A Boolean expression returns a value of true or false, based on a comparison. The various Boolean operators used for comparing Boolean variables are:

OR ("|"), AND ("&"), and NOT ("¬").

The possible types of ***components*** in a condition include the following: a Boolean operator, a Boolean variable, a pair of Boolean parentheses, a relational operator, or an arithmetic expression. If a condition is incorrect, then at least one component of the condition is incorrect. Thus, types of errors in a condition include the following:

1. Boolean operator error

   Example: (a<b) &&& (a>c)

2. Boolean parenthesis error

   Example: ((a<b) && (a<c)

3. Relational operator error

   Example: a<b c

4. Arithmetic expression error

   Example: a+b*

The following are the various condition testing strategies:

1. Branch Testing

For a compound condition C, the true and false branches of C and every simple condition in C need to be executed at least once.

2. Domain Testing

For a Boolean expression with n variables, all of 2^n possible tests are required (n>0). This strategy can detect Boolean operator, variable, and parenthesis errors, but is practical only if ***n*** is small.

3. Branch and Relational Operator (BRO) Testing

This technique guarantees the detection of branch and relational operator errors in a condition provided that all Boolean variables and relational operators in the condition occur only once and have no common variables.

c. Data Flow Testing

The ***data flow testing*** method selects test paths of a program according to the locations of definitions and uses of variables in the program. In a program, if we assume a statement with S as its statement number,

DEF(S) = { X | statement S contains a definition of X }

USE(S) = { X | statement S contains a use of X }

The definition of variable X at statement S is said to be *live* at statement S′ if there exists a path from statement S to statement S′ that does not contain any other definition of X.

A ***definition-use chain*** (or ***DU chain***) of variable X is of the form [X, S, S′], where S, S′ are statement numbers, X is in DEF(S) and USE(S′), and the definition of X in statement S is live at statement S′.

A simple data flow testing strategy, known as DU testing strategy, is to require that every *DU chain* be covered at least once. Data flow testing strategies are useful for selecting test paths of a program containing ***nested if*** and ***loop statements***. However the disadvantage is that it does not guarantee the coverage of all branches of a program.

d. Loop Testing

Loop Testing focuses exclusively on the validity of loop constructs. The four different classes of loop constructs are described below:

1. **Simple Loops**

   The following set of tests should be applied to simple loops, where n is the maximum number of allowable passes through the loop.

   i. Skip the loop entirely.

   ii. Only one pass through the loop.

   iii. Two passes through the loop.

   iv. m passes through the loop where m < n.

   v. n – 1, n, n + 1 passes through the loop.

2. **Nested Loops**

   i. Start at the innermost loop. Set all other loops to minimum values.

   ii. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values. Add other tests for out-of-range or excluded values.

iii. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to *typical* values.

iv. Continue until all loops have been tested.

3. **Concatenated Loops**

   Concatenated loops can be tested using the approach defined above for simple loops if each of the loops is independent of the other. When the loops are not independent, the approach applied to nested loops is recommended.

4. **Unstructured Loops**

   Whenever possible this class of loops should be redesigned in to structured loops. Then one of the above mentioned approaches can be used.

8.6.1. Limitations

There are some software defects in the white box testing. They are:

a. Logic errors and incorrect assumptions are inversely proportional to probability that a program path will be executed. Errors enter into our work when we design and implement functions, conditions, or controls that are out of the mainstream.

b. It is believed that a logical path is not likely to be executed when it may be executed on a regular basis. The assumptions about the flow of control and data may lead to make design errors that are uncovered only once when testing commences.

c. Typographical errors are random. When a program is translated into programming language source code, it is likely that some typing errors will occur. Many will be uncovered by syntax checking mechanisms, but others will go undetected until testing begins.

8.7. BLACK BOX TESTING

Black box testing is based on an analysis of the specification of a piece of software without reference to its internal workings. The term *black box* indicates that the internal implementation of the program being executed is not examined by the tester. For this reason black box testing is not normally carried out by the programmer. In most real-world engineering firms, one group does design work while a separate group does the testing.

Black Box testing also known as *Functional Testing* focuses on the functional requirements of the software. It allows the tester to derive a set of input conditions that will fully exercise all functional requirements of a program. The structure of a program (i.e., code) is not considered for the design

of test cases. Instead the test cases for the entire system are designed from the requirements specification document for the system.

Black Box Testing attempts to find errors in the following categories:

1. Incorrect or missing functions
2. Interface errors
3. Errors in data structures or external data base access
4. Performance errors
5. Initialization and termination errors

By applying black box testing techniques, we derive a set of test cases that satisfy the following criteria:

1. Test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and
2. Test cases that tell us something about the presence or absence of classes of errors, rather than errors associated only with the specific test at hand.

The various Black Box Testing techniques are as follows:

a. Graph Based Testing

b. Equivalence Partitioning

c. Boundary Value Analysis

d. Comparison Testing

a. Graph Based Testing

The steps involved in graph based testing are as follows:

1. Identify the *objects* that are modeled in the software and the *relationships* that connect these objects.
2. Create a *graph* – a collection of *nodes* that represent objects; *links* that represent the relationships between objects; *node weights* that describe the properties of a node like a specific data value or state behavior, and link weights that describe some characteristic of a link.
3. Devise a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

b. Equivalence Partitioning

A technique in black box testing is **equivalence partitioning**. Equivalence partitioning is designed to minimize the number of test cases by dividing tests in such a way that the system is expected to act the same way for

all tests of each equivalence partition. Test inputs would be selected from each partition.

Equivalence partitions are designed so that every possible input belongs to one and only one equivalence partition.

Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a *range*, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific *value*, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a *set*, one valid and one invalid equivalence class are defined.
4. If an input condition is *Boolean*, one valid and one invalid class are defined.

Disadvantages

- Doesn't test every input
- No guidelines for choosing inputs
- Heuristic based
- very limited focus

c. Boundary value analysis (BVA)

Boundary value analysis is a technique of black box testing in which input values at the boundaries of the input domain are tested. It has been widely recognized that input values at the extreme ends of, and just outside of, input domains tend to cause errors in system functionality. BVA is a test case design technique that leads to a selection of test cases that exercise bounding values. This technique complements the equivalence partitioning method. It has been observed that most programs that work correctly for a set of values in an equivalence class, fail on some special values. These values often lie on the boundary of the equivalence class. Test cases that have values on the boundaries of equivalence classes are therefore likely to be “high yield” test cases. Selecting such test cases is the aim of BVA.

Boundary value analysis is an excellent way to catch common user input errors which can interrupt proper program functionality.

Boundary value analysis complements the technique of equivalence partitioning.

Guidelines for BVA area as follows:

1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b*, just above and just below *a* and *b*, respectively.
2. If an input condition specifies a number of values, test cases should be developed that exercise the maximum and minimum numbers. Values just above and below maximum and minimum are also tested.
3. Guidelines 1 and 2 are applied to output conditions.
4. If internal program data structures have prescribed boundaries, be certain to design a test case to exercise the data structure at its boundary.

Some of the advantages of boundary value analysis are:

- Very good at exposing potential user interface/user input problems
- Very clear guidelines on determining test cases
- Very small set of test cases generated

Disadvantages to boundary value analysis:

- Does not test all possible inputs
- Does not test dependencies between combinations of inputs

d. Comparison Testing

Using lessons learned from redundant systems, researchers have suggested that independent versions of software be developed for critical applications, even when only a single version will be used in the delivered computer based system. These independent versions form the basis of black box testing technique called ***comparison testing*** or ***back-to-back testing***.

When multiple implementations of the same specification have been produced, test cases designed using other black box techniques are provided as input t o each version of the software. If the output from each version is the same, it is assumed that all implementations are correct. If the output is different, each of the applications is investigated to determine if a defect in one or more versions is more responsible for the difference. Mostly comparison is done using automated tools.

Disadvantages

Comparison testing is not foolproof. If the specification from which all versions have been developed is itself is in error, then all versions will likely reflect the error. Also, if each of the independent versions produces identical but incorrect results, then comparison testing will fail to detect the error.

8.7.1. Smoke testing

A sub-set of the black box test is the smoke test. A smoke test is a cursory examination of all of the basic components of a software system to ensure that they work. Typically, smoke testing is conducted immediately after a software build is made. The term comes from electrical engineering, where in order to test electronic equipment, power is applied and the tester ensures that the product does not spark or smoke.

8.8. TESTING FOR SPECIALIZED ENVIRONMENTS

As computer software has become more complex, the need for specialized testing approaches has also grown.

a. Testing GUIs

Although the creation of the Graphical User Interfaces (GUIs) has become less time consuming, the design and execution of test cases is more difficult. The following questions can serve as a guideline for creating a series of generic tests for GUIs:

For Windows:

> Questions relating to the opening, resizing, moving, scrolling, and closing of windows, functionality of windows, multitasking, etc. are some of the few questions.

For pull-down menus and mouse operations:

> Questions relating to the menu bar, menu functions, pull down menus, mouse operations with regard to menus, etc. need to be considered while designing tests.

Data entry:

> Questions relating to modes of data entry, proper echoing of data entry, recognition of valid/invalid data, input messages, etc. may be used in designing test cases.

b. Testing of Client/Server Architectures

The distributed nature of client/server environments, the performance issues associated with transaction processing, the complexities of network communication, the need to service multiple clients from a centralized database, and the coordination requirements imposed on the server all combine to make testing of Client/Server architectures more difficult.

c. Testing Documentation & Help Facilities

Documentation testing should be a meaningful part of every software test plan as errors in documentation are more frustrating. Documentation testing can be approached in two phases:

1. **First Phase** - a ***formal technical review*** examines the document for editorial clarity.
2. **Second Phase** - ***live test***, uses the documentation in conjunction with the use of the actual program.

Live test for documentation can be approached using black-box testing methods. For instance, Graph based testing can be used to describe the use of the program, Equivalence partitioning and Boundary Value Analysis can be used to define various classes of input and associated interactions. Several questions relating to the documentation, terminology used, guidance provided for understanding error messages and easy troubleshooting, etc. have to be answered. This can be accomplished by having an independent third party test the documentation in the context of program usage.

d. Testing for Real Time Systems

In many real-time applications, in addition to designing white and black box test cases, the test case designer must also consider event handling, the timing of data, and the parallelism of the tasks that handle the data. Most real-time systems process interrupts. Therefore, testing the handling of these Boolean events is essential and is done using the State Transition Diagram and the Control Specification.

An overall ***four-step strategy*** for design of test cases for real-time systems is as follows:

1. ***Task Testing:*** White box and Black box tests are designed and executed for each task. Each task is executed independently during these tests. Task testing uncovers errors in logic and function, but will not uncover timing or behavioral errors.
2. ***Behavioral Testing:*** Events such as interrupts, control signals, data are categorized for testing. Each of these events is tested individually and the behavior of the executable system is examined to detect errors that occur as a consequence of processing associated with these events. The behavior of the system model and the executable software can be compared for conformance. Once each class of events has been tested, events are presented to the system in a random order and with random frequency. The behavior of the software is examined to detect behavior errors.
3. ***Intertask Testing:*** Asynchronous tasks that are known to communicate with one another are tested with different data rates and processing load to determine if inter-task synchronization errors will

occur. Also, tasks that communicate via a message queue or data store are tested to uncover errors in the sizing of data storage areas.

4. ***System Testing:*** Software and hardware are integrated, and a full range of system tests is conducted so as to uncover errors at hardware/software interface.

8.9. FORMAL VERIFICATION

Software testing is very often referred to as software verification and validation (V & V). Verification refers to the activities which ensure that software correctly implements a specific function. Formal verification involves the use of rigorous, mathematical techniques to demonstrate that computer programs have certain desired properties. The methods of ***input-output declaration***, ***weakest preconditions***, and ***structural induction*** are three commonly used techniques.

Two important questions:

- Whether the chosen approach will actually lead to the required solution?
- Whether the already developed components are correct, i.e., whether they fulfill the specifications?

- Ideally we would like to be able to prove at any time that for every imaginable input the algorithms included in the design and their interplay will deliver the expected results according to the specification.
- Since today's large software systems permit the complete description of neither the input set nor the anticipated result set (due to combinatorial explosion), such systems can no longer be completely tested. Thus efforts must be made to achieve as much clarity as possible already in the design phase about the correctness of the solution.
- Proof of correctness cannot be handled without formal and mathematical tools
- The use of a verification procedure forces the software engineer to reproduce all design decisions and thus also helps in finding logical errors. Verification is an extremely useful technique for early detection of design errors and also complements the design documentation. Verification itself can be fallible; however, it cannot replace testing of a software product.
- Verification can be used successfully to prove the correctness of short and simple algorithms. For the production of larger software systems, the difficulties rise so sharply that it should be clear that verification fails as a practical test aid.

8.9.1 Mathematical program verification

- If programming language semantics are formally defined, it is possible to consider a program as a mathematical object.
- Using mathematical techniques, it is possible to demonstrate the correspondence between a program and a formal specification of that program.
- Program is proved to be correct with respect to its specification.
- Formal verification may reduce testing costs; it cannot replace testing as a means of system validation.
- Techniques for proving program correctness and axiomatic approaches

The basis of the axiomatic approach

- Assume that there are a number of points in a program where the software engineer can provide assertions concerning program variables and their relationships. At each of these points, the assertions should be invariably true. Say the points in the program are *P(1), P(2),...P(n).* The associated assertions are *a(l), a(2),...,a(n).* Assertion *a(1)* must be an assertion about the input of the program and *a(n)* an assertion about the program output.
- To prove that the program statements between points *P(i)* and *P(i+l)* are correct, it must be demonstrated that the application of the program statements separating these points causes assertion *a(i)* to be transformed to assertion *a(i+l).*
- Given that the initial assertion is true before program execution and the final assertion is true after execution, verification is carried out for adjacent program statements.

8.10. DEBUGGING

Debugging, a narrow view of software testing, is performed heavily to find out design defects by the programmer. The limitation of human nature makes it almost impossible to make a moderately complex program correct the first time. Finding the problems and get them fixed, is the purpose of debugging in programming phase.

Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another.

Although each debugging experience is unique, certain general principles can be applied in debugging. This section particularly addresses debugging software, although many of these principles can also be applied to debugging hardware.

The basic steps in debugging are:

- Recognize that a bug exists
- Isolate the source of the bug
- Identify the cause of the bug
- Determine a fix for the bug
- Apply the fix and test it

8.10.1. The Debugging Process

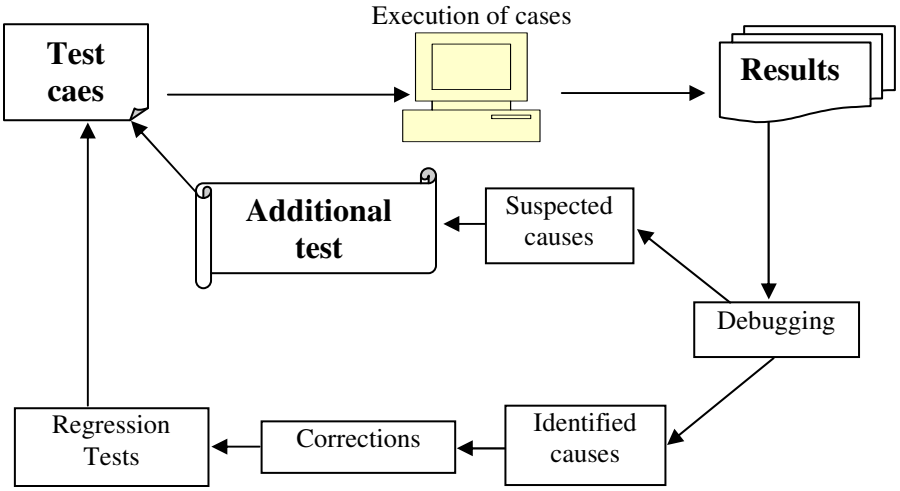


Fig. 8.5 Debugging Process

The debugging process begins with the execution of the test case. Results are assessed and a lack of correspondence between expected and actual result is encountered. In many cases, the non-corresponding data is found to be a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

Characteristics of bugs ([Cheung 1990])

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled program structures exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.

3. The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

The debugging process will always have one of two outcomes:

(1) The cause will be found, corrected, or removed, or

(2) The cause will not be found.

In general, three categories for debugging approaches are proposed:

- Brute Force Approach
- Backtracking Approach
- Cause Elimination Approach

Each of the debugging approaches can be supplemented with debugging tools such as debugging compliers, dynamic debugging aids, automatic test case generators, memory dumps, and cross-reference maps.

a. Brute Force Approach

The ***brute force*** category of debugging is the most common method for isolating the cause of an error. This method is generally the least efficient and is used when everything else fails. A philosophy such as "let the computer find the error" is used in this approach. Memory dumps are taken, run-time traces are invoked, and the program is loaded with "write" statements, with the hope that from the mass of information produced, we will find a clue that can lead us to the cause of an error.

b. Back Tracking Approach

Debugging by ***backtracking*** involves working backward in the source code from the point where the error was observed in an attempt to identify the exact point where the error occurred. It may be necessary to run additional test cases in order to collect more information. This approach can be used successfully in small programs. The disadvantage with this approach is that if the program is too large, then the potential backward paths may become unmanageably large.

c. Cause Elimination Approach

Cause elimination approach is manifested by ***induction* or *deduction.*** This approach proceeds as follows:

1. List possible causes for the observed failure by organizing the data related to the error occurrence.
2. Devise a "cause hypothesis".
3. Prove or disprove the hypothesis using the data.
4. Implement the appropriate corrections.
5. Verify the correction. Rerun the failure case to be sure that the fix corrects the observed symptom.

Check your Progress

i. _____is the process used to help identify the correctness, completeness, security and quality of developed computer software.
ii. In other words, testing is nothing but _____ or _____ that is comparing the actual value with expected one.
iii. _____ is done to validate the code written and is usually done by the author of the code.
iv. _____ and _____ are sub-categories of System testing.
v. White box testing is also called as _____ and _____

Solutions

I

.....

Ii

.....

Iii

.....

Iv

.....

V

.....

8.11. REVIEW QUESTIONS

1. What are the Testing objectives, rules in S/W testing fundamentals?
2. Explain the testing information flow.
3. What are the stages available in testing process?
4. Describe art of debugging in software testing strategies.
5. Write short notes on: a. Data flow testing b. Integration testing.

6. What is the objective of unit testing?
7. Explain Control structure testing, Dataflow testing Loop testing and Block box testing.
8. What is Equivalent partitioning?
9. Explain the boundary value analysis method?
10. Write a short notes on,
    a. Comparison testing
    b. Testing for real-time testing
11. Differentiate Verification and Validation?
12. What are the different structural testing methods? Explain.

8.12. LET US SUM UP

The purpose of testing is not only to prove that the code performs in accordance with the design specifications, but to prove that it does not fail when subjected to undefined inputs.

There are several types of testing. **Unit testing** tests one module of code for correct inputs, outputs, and functionality. **System testing** tests all the modules of a software system together. **User acceptance testing** is a form of systems testing to see if not only the system works, but it meets the requirements of the business user. **Regression testing** is testing performed after a system change to make sure that all system features are still present after the change.

Debugging is the process of isolating and correcting the causes of known errors. Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. During debugging, we might encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g., causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to find the cause also increases. Success at debugging requires highly developed problem solving skills.

8.13. LESSON END ACTIVITIES

i. State few salient characteristics of modern testing tools.

ii. If you could only select three test case design methods to apply during unit testing, what could they be and why?

8.14. POINTS FOR DISCUSSION

i. Differentiate Static Testing and Dynamic Testing.

8.15. REFERENCES

1. Hetzel W., *The Complete Guide to Software Testing*, QED Information Sciences, 1984.
2. Myers G., *The Art of Software Testing,* Wiley, 1979
3. A.J. Albrecht and J.E. Gaffney, Jr, Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation, IEEE Trans. Software Engineering. Vol. 9. 6:639-648, (1983)

Lesson 9: Software Maintenance

Contents

9.0. AIMS AND OBJECTIVE

- To identify those software development tasks, products, and activities that contribute directly to a maintainable software product
- To describe the software maintenance process
- To identify factors that influence software maintenance resource estimation
- To justify some of the design guidelines introduced earlier in terms of their contributions to building a maintainable software product.

9.1. INTRODUCTION

Maintenance is the challenge of system development. It holds the software industry captivity, tying up programming resources; Analysts and programmers spend far more time maintaining programs than they do writing them. Maintenance accounts for 50-80% of total system development and the percentage continues to rise as more software is produced. Maintenance is defined by describing four activities that are undertaken after a program is released for use:

- Corrective Maintenance
- Adaptive Maintenance
- Perfective Maintenance or Enhancement

- Preventive Maintenance or Reengineering

Only about 20% of all maintenance work is spent "fixing mistakes". The remaining 80% is spent adapting existing systems to changes in their external environment, making enhancements requested by users, and reengineering an application for future use.

1. *Perfective maintenance* means those changes demanded by the user or the system programmer which improve the system in some way without changing its functionality.
2. *Adaptive maintenance* is maintenance due to changes in the environment of the program.
3. *Corrective maintenance* is the correction of undiscovered system errors.
4. The techniques involved in maintaining a software system are essentially those used in building the system in the first place.

- New requirements must be formulated and validated,
- Components of the system must be redesigned and implemented
- Part or all of the system must be tested.

The techniques used in these activities are the same as those used during development.

5. There are no special technical tricks which should be applied to software maintenance.

Important

6. Large organizations devoted about 50% of their total programming effort to maintaining existing systems.

- The program maintainer pays attention to the *principles of information hiding*. For a long-lived system, it is quite possible that a set of changes to a system may themselves have to be maintained. It is a characteristic of any change that the original program structure is corrupted. The greater the corruption, the less understandable the program becomes and the more difficult it is to change. The program modifier should try, as far as possible, to minimize effects on the program structure.
- One of the problems of managing maintenance is that maintenance has a poor image among software engineers. It is seen as a less skilled process than program development and, in many organizations, maintenance is allocated to inexperienced staff. The end result of this negative image is that maintenance costs are probably increased because staffs allocated to the task are less skilled and experienced than those involved in system design.

Software Maintenance Characteristics

In order to understand the characteristics of S/W maintenance, the following three points are considered.

1. Structured and unstructured maintenance- The actual flow of events that occur as a result of a maintenance request is illustrated in the figure below. If the only available element of a S/W configuration is source code. Maintenance activity begins with the evaluation of the code, often complicated by poor internal documentation. The delicate characteristics such as program structure, global data structure, system interfaces, and performance and design constraints are difficult to handle and are often misinterpreted. The amounts of changes that are made to the code are difficult to assess. The Regression tests are impossible to conduct since no record of testing exists.

Suppose, a complete configuration exists, the maintenance task begins with an evaluation of the design documentation. Important structural, performance and interface characteristics of the S/W are determined. The impact of required modifications or the corrections are assessed, and an approach is planned. The design is modified and reviewed. Now source code is developed, the regression tests are conducted using information contained in the test specification and the S/W is released again.

2. Maintenance Cost – The cost of S/W maintenance has increased steadily during the past several years. One intangible cost of S/W maintenance is the development opportunity that is postponed or lost since the available resources must be channeled to maintenance tasks. Other intangible costs include:

1. Customer dissatisfaction when legitimate request for repair or modification cannot be addressed in a timely manner.
2. Reduction in overall S/W quality as a result of changes that introduce latent errors in the maintained S/W.
3. The upheaval caused during development efforts when the staff must be pulled to work on a maintenance task.

The effort expended on maintenance may be divided into productive activities and wheel spinning activity. The following expression provides a model of maintenance effort.

$$M = p + Ke^{(c-d)}$$

Where M = total effort expended on maintenance

P = Productive effort

K = An empirical constant

C= The measure of complexity that can be attributed to a lack of good design and documentation.

d= The measure of the degree of familiarity with S/W

3. Problems Most of the problems associated with S/W maintenance can be traced to deficiencies in the way was planned and developed. A lack of control and discipline in S/W engineering development activities nearly always translates into problems during S/W maintenance. The following are the problems that can be associated with S/W maintenance.

- It is often difficult or impossible to trace the evolution of the S/W through many versions or releases. Changes are not adequately documented.
- It is often difficult or impossible to trace the process through which S/W was created.
- It is difficult to understand someone else program.
- Someone else is often not around to explain. Mobility among S/W personnel is high. We cannot rely upon a personal explanation of S/W by the developer when maintenance is required.
- The documentation does not exist. The recognition that S/W must be documented is a first step, but documentation must be understandable and consistent with source code to be of any value.
- Most of the S/W is not designed for change. Unless, a design method accommodates change through concepts such as functional independence or object classes, modifications to S/W are difficult and error prone.
- The Maintenance has not been viewed as a glamorous work. Much of this perception comes from the high frustration level associated with maintenance work.

9.2. ENHANCING MAINTAINABILITY DURING DEVELOPMENT

Maintainability can be defined as the ease with which S/W can be understood, corrected, adapted and with which S/W can be understood, corrected, adapted and enhanced. Many activities performed during software development enhance the maintainability of a software product. Some of these Activities are:

- Analysis activities
    - Develop standards and guidelines
    - Set milestones for the supporting documents
    - Specify quality assurance procedures
    - Identify likely product enhancements
    - Determine resources required for maintenance
    - Estimate maintenance cost
- Architectural Design Activities

    - Emphasize clarity and modularity as design criteria
    - Design to ease likely enhancements
    - Use standardized notations to document data flow, functions, structure, and interconnections
    - Observe the principles of information hiding, data abstraction, and top-down hierarchical decomposition.
- Detailed Design Activities
    - Use standardized notations to specify algorithms, data structures, and procedure interface specifications.
    - Specify side effects and exception handling for each routine
    - Provide cross-reference directories
- Implementation Activities
    - Use single entry, single exist constructs
    - Use standard indentation of constructs
    - Use simple, clear coding style
    - Use symbolic constants to parameterize routines
    - Provide margins on resources
    - Provide standard documentation prologues for each routine
    - Follow standard internal commenting guidelines.
- Other Activities
    - Develop a maintenance guide
    - Develop a test suite
    - Provide test suite documentation

9.3. MANAGERIAL ASPECTS OF SOFTWARE MAINTENANCE

Successful software maintenance, like all software engineering activities, requires a combination of managerial skills and technical expertise. In this section, we discuss some of the managerial concerns of software maintenance. Technical issues in software maintenance are also discussed in the following sections.

One of the most important aspects of software maintenance involves tracking and control of maintenance activities. Maintenance activity for a software product usually occurs in response to a change request filed by a user of the product. The biggest challenge in Software Maintenance is managing the maintenance staff.

Five steps for improving the motivation of maintenance staff

- Couple software objectives to organizational goals.
- Couple software maintenance rewards to organizational performance.
- Integrate software maintenance personnel into operational teams.
- Create a flexible perfective maintenance budget.
- Involve maintenance staff early in the software process during standards preparation reviews and test preparation.

9.4. SOFTWARE CONFIGURATION MANAGEMENT

9.4.1. Need for Software Configuration Management

When you build computer software, change happens. And because it happens, you need to control it effectively. Software configuration management (SCM) is a set of activities that are designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling changes that are imposed, and auditing and reporting on the changes that are made.

The Software configuration management (SCM) is an activity that is applied throughout the S/W engineering process since a change can occur at any time, SCM activities are developed for the following,

- Identify change,
- Control change,
- Ensure that change is properly implemented,
- Report change to others who may have interest.

SCM is a set of tracking and control activities that begin when a software development project begins and ends only when the software is retired.

The process of identifying and defining the configuration items in a software system, controlling the release, versioning and change of these items though out the software system life cycle, recording and reporting the status of configuration items and change requests, and verifying the completeness and correctness of configuration items.

Software configuration management (SCM) is a "set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made." In other words, SCM is a methodology to control and manage a software development project.

SCM concerns itself with answering the question: *somebody did something, how can one reproduce it?* Often the problem involves not reproducing "it"

identically, but with controlled, incremental changes. Answering the question will thus become a matter of comparing different results and of analysing their differences. Traditional *CM* typically focused on controlled *creation* of relatively simple products. Nowadays, implementators of SCM face the challenge of dealing with relatively minor increments under their own control, in the context of the complex system being developed.

The goals of SCM are generally:

- Configuration Identification- What code are we working with?
- Configuration Control- Controlling the release of a product and its changes.
- Status Accounting- Recording and reporting the status of components.
- Review- Ensuring completeness and consistency among components.
- Build Management- Managing the process and tools used for builds.
- Process Management- Ensuring adherence to the organizations development process.
- Environment Management- Managing the software and hardware that host our system.
- Teamwork- Facilitate team interactions related to the process.
- Defect Tracking- making sure every defect has traceability back to the src

9.4.2. Configuration management techniques

An important aspect of configuration management. A configuration item is a unit of configuration that can be individually managed and versioned. Typically, a configuration management system will control files, requirements, or another definable unit. These units are managed with a combination of process and tools to avoid the introduction of errors and to maintain high quality results. The units themselves can be considered configuration items, or they may be combined into an overall collection that is managed under the same set of processes and tools.

From the perspective of the implementor of a change, the configuration item is the "what" of the change. Altering a specific baseline version of a configuration item creates a new version of the same configuration item, itself a baseline. In examining the effect of a change, we first ask "what configuration items are affected" and then proceed to "how have the configuration items been affected". A release (itself a versioned entity) may consist of several configuration items. The set of changes to each configuration item will appear in the release notes, and the notes may contain specific headings for each configuration item.

As well as participating in the implementation of a change and in the management of a change, the listing and definition of each configuration item may act as a common vocabulary across all groups connected to the product. It should be defined at a level such that an individual involved with product

marketing and an individual at the coal face of implementation can agree to a common definition when they use the name of the configuration item. Selection and identification of configuration items for a particular project can be seen as the first step in developing an overall architecture from the top down.

Configuration items, their versions and their changes form the basis of any configuration audit.

9.5. SOURCE CODE METRICS

A great deal of effort has been expended on developing metrics to measure the complexity of source code. Most of the metrics incorporate easily compute properties of the source code, such as the no. of operators and operands, the complexity of the control flow graph, the no. of parameters and global variable in routines and the no. of levels and manner of interconnects of the call graph.

Source-code complexity measures can be used to determine the complexity of a program before and after modification, and used to identify candidate routines for further refinement and rework. Two source-code metrics are discussed in this section: a. Halstead's effort equation, and McCabe's cyclomatic complexity measure.

Measuring program maintainability

- Maintainability metrics are based on the assumption that the maintainability of a program is related to its complexity.
- The metrics measure some aspects of the program complexity.
- It is suggested that high complexity values correlate with difficulties in maintaining a system component.
- The complexity of a program can be measured by considering (Halstead's effort)
    - the number of unique operators,
    - the number of unique operands,
    - the total frequency of operators, and
    - the total frequency of operands in a program.
- The program complexity is not dependent on size but on the decision structure of the program. Measurement of the complexity of a program depends on transforming the program so that it may be represented as a graph and counting the number of nodes, edges and connected components in that graph ([McCabe cyhclomatic complexity measure]).

Program evolution dynamics by Lehman

Lehman's laws

1. **The law of continuing change:** *A program that is used in a real-world environment necessarily must change or become less and less useful in that environment.*
2. **The law of increasing complexity:** *As an evolving program changes, its structure becomes more complex unless active efforts are made to avoid this phenomenon.*
3. **The law of large program evolution:** *Program evolution is a self-regulating process and measurement of system attributes such as size, time between releases, number of reported errors, etc., reveals statistically significant trends and invariances.*
4. **The law of organizational stability:** *Over the lifetime of a program, the rate of development of that program is approximately constant and independent of the resources devoted to system development.*
5. **The law of conservation of familiarity:** *Over the lifetime of a system, the incremental system change in each release is approximately constant.*

- The first law tells us that system maintenance is an inevitable process. Fault repair is only part of the maintenance activity and that changing system requirements will always mean that a system must be changed if it is to remain useful. Thus, the constant theme of this text is that software engineering should be concerned with producing systems whose structure is such that the costs of change are minimized.
- The second law states that, as a system is changed, its structure is degraded and additional costs, over and above those of simply implementing the change, must be accepted if the structural degradation is to be reversed. The maintenance process should perhaps include explicit restructuring activities which are simply aimed at improving the adaptability of the system. It suggests that program restructuring is an appropriate process to apply.
- The third law suggests that large systems have a dynamic all of their own and that is established at an early stage in the development process. This dynamic determines the gross trends of the system maintenance process and the particular decisions made by maintenance management are overwhelmed by it. This law is a result of fundamental structural and organizational effects.
- The fourth law suggests that most large programming projects work in what he terms a 'saturated' state. That is, a change of resources or staffing has imperceptible effects on the long-term evolution of the system.

- The fifth law is concerned with the change increments in each system release.
- Lehman's laws are really hypotheses and it is unfortunate that more work has not been carried out to validate them. Nevertheless, they do seem to be sensible and maintenance management should not attempt to circumvent them but should use them as a basis for planning the maintenance process. It may be that business considerations require them to be ignored at any one time (say it is necessary to make several major system changes). In itself, this is not impossible but management should realize the likely consequences for future system change.

9.6. OTHER MAINTENANCE TOOLS AND TECHNIQUES

Any craftsperson, a mechanic or a carpenter, needs a good workshop with tools. The workshop for software engineering is called an integrated project support environment and the tool set that fills the workshop is called ***Computer-Aided Software Engineering (CASE)***. CASE provides the software engineer with the ability to automate manual activities and to improve engineering insight. Like computer-aided engineering and design tools that are used by engineers in other disciplines, CASE tools help to ensure that quality is designed in before the product is built.

Building Blocks for Case

Computer aided software engineering can be as simple as a single tool or as complex as a complete environment. The building blocks for CASE are:

- environment architecture
- hardware platform
- operating system
- portability services
- integration framework, and
- CASE tools

Each building block forms a foundation for the next, with tools sitting at the top of the heap. Successful environments for software engineering are built on an ***environment architecture*** that encompasses appropriate hardware and systems software.

Taxonomy of Case Tools

It is necessary to create taxonomy of CASE tools - to better understand the breadth of CASE and to better appreciate where such tools can be applied in the software engineering process. CASE tools can be classified by function, by their role as instruments for managers or technical people, by their use in the various steps of the software engineering process, by the environment

architecture (hardware and software) that supports them, or even by their origin or cost. The taxonomy presented here uses function as a primary criterion.

Business process engineering tools:

By modeling the strategic information requirements of an organization, business process engineering tools provide a 'meta-model' from which specific information systems are derived. The primary objective for tools in this category is to represent business data objects, their relationships, and how these data objects flow between different business areas within a company.

Process modeling and management tools:

Process modeling tools (also called process technology tools) are used to represent the key elements of a process so that it can be better understood. Process management tools provide links to other tools that provide support to define process activities.

Project planning tools:

Tools in this category include software project effort and cost estimation tools and project scheduling tools. Project scheduling tools enable the manger to define all project tasks, create a task network, represent task interdependencies, and model the amount of parallelism possible for the project.

Risk analysis tools:

Risk analysis tools enable a project manager to build a risk table by providing detailed guidance in the identification and analysis of risks.

Project management tools:

Tools in the category are often extensions to project planning tools and are used to collect metrics that will ultimately provide an indication of software product quality.

Requirements tracing tools:

The objective of requirements tracing tools is to provide a systematic approach to the isolation of requirements, beginning with the customer request for proposal or specification. The typical requirements tracing tool combines human-interactive text evaluation with a database management system that stores and categories each system requirement that is “parsed" from the original specification.

Metrics and managements tools:

Metrics or measurement tools focus on process and product characteristics. Management-oriented tools capture project specific metrics (e.g., LOC/person-month, defects per function point) that provide an overall indication of productivity or quality. Technically oriented tools determine technical metrics that provide greater insight into the quality of design or code.

Documentation tools:

Documentation production and desktop publishing tools support nearly every aspect of software engineering and represent a substantial "leverage" opportunity for all software developers. It is not unusual for a software development organization to spend as much as 20 or 30 percent of all software development effort on documentation.

System software tools:

CASE is a workstation technology. Therefore, the CASE environment must accommodate high quality network system software, object management services, distributed component support, electronic mail, bulletin boards, and other communication capability.

Quality assurance tools:

The majority of CASE tools that claim to focus and quality assurance are actually metrics tools that audit source code to determine compliance with language standards. Other tools extract technical metrics in an effort to project the quality of the software that is been built.

Database management tools:

Database management software serves as a foundation for the establishment of a CASE database (repository) that is called the project database.

Software configuration management tools:

Software Configuration Management (SCM) lies at the kernel of every CASE environment. Tools can assist in all five major SCM tasks - identification, version control, change control, auditing, and status accounting.

Analysis and design tools:

Analysis and design tools enable a software engineer to create models of the system to be built. The models contain a representation of data, function, and behavior and characterizations of data, architecture, component-level, and interface design.

Pro/Sim tools:

PRO/SIM (Prototyping and Simulation) tools provide the software engineer with the ability to predict the behavior of a real-time system prior to the time that it is built. In addition, these tools enable the software engineer to develop mock-ups of the real-time system, allowing the customer to gain insight into the function, operation, and response prior to actual implementation.

Interface design and development tools:

Interface design and development tools are actually a tool kit of software components (classes) such as menus, buttons, window structures, icons, scrolling mechanisms, device drivers, and so forth.

Prototyping tools:

Prototyping tools such as *screen printers* enable a software engineer to define screen layout rapidly for interactive applications. More sophisticated CASE prototyping tools enable the creation of a data design, coupled with both screen and report layouts.

Programming tools:

The programming tools category encompasses the compilers, editors, and debuggers that are available to support most conventional and object–oriented programming languages, graphical programming environments, application generators, and database query languages reside within this category.

Web development tools:

The activities associated with web engineering are supported by a variety of tools for Web Application development that assist in the generation of text, graphics, forms, scripts, applets, and other elements of a Web page.

Integration and testing tools:

The testing tools categories are:

- *Data acquisition* - tools that acquire data to be used during testing.
- *Static measurement* - tools that analyze source code without executing test cases.
- *Dynamic measurement* - tools that analyze source code during execution.
- *Simulation* - tools that simulate functions of hardware or other externals.
- *Testing management* - tools that assist in the planning, development, and control of testing.
- *Cross-functional tools* - tools that cross the bounds of the preceding categories.

Static analysis tools:

Static testing tools assist the software engineer in deriving test cases. Three different types of static testing tools are used in the industry: code-based testing tools, specialized testing languages, and requirements-based testing tools. Code-based testing tools accept source code as input and perform a number of analyses that result in the generation of test cases. Specialized testing languages enable a software engineer to write detailed test specifications that describe each test case and the logistics for its execution. Requirements-

based testing tools isolate specific user requirements and suggest test cases that will exercise the requirements.

Dynamic analysis tools:

Dynamic testing tools interact with an executing program, checking path coverage, testing assertions about the value of specific variables, and otherwise incrementing the execution flow of the program. Dynamic tools can be either intrusive or nonintrusive. An intrusive tool changes the software to be tested by inserting probes. Nonintrusive tools use a separate hardware processor that runs in parallel with the processor containing the program that is being tested.

Test management tools:

Test management tools are used to control and coordinates software testing for each of the major testing steps. Tools in this category manage and coordinate regression testing, perform comparisons that ascertain differences between actual and expected output, and conduct batch testing of programs with interactive human/computer interfaces. In addition to the functions noted, many test management tools also serve as genetic test drivers. A test driver reads one or more test cases from a testing file, formats the test data to conform to the needs of the software under test, and then invokes the software to be tested.

Client/server testing tools:

The C/S environment demands specialized testing tools that exercise the graphical user interface and the network communications requirements for client and server.

Reengineering tools:

The reengineering tools category can be subdivided into the following functions:

1) Reverse engineering to specification tools - take source code as input and generate graphical structural analysis and design models, where -user lists, and other design information.
2) Code restructuring and analysis tools - analyze program syntax, generate a control flow graph, and automatically generate a structured program.
3) On-line system reengineering tools - are used to modify on-line database systems.

These tools are limited to specific programming languages and require some degree of interaction with the software engineer.

Check your progress

i. The _____ is an activity that is applied throughout the S/W engineering process

ii. A great deal of effort has been expended on developing metrics to measure the _____ of source code.

iii. The workshop for software engineering is called an integrated project support environment and the tool set that fills the workshop is called _____.

Solutions

i

.....

ii

.....

iii

.....

9.7. REVIEW QUESTIONS

1) What are the different project management tools? Explain them.
2) Explain different Analysis and design tools.
3) What are the different programming tools? Explain them.
4) What are the different Integration and testing tools?
5) Explain the three static analysis tools.
6) Write short notes on,
    a. Dynamic analysis tools.
    b. Test management tools.
    c. Prototyping tools.
7) Write a short note on Tools Integration

9.8. LET US SUM UP

Software Maintenance Activities

- Maintenance can be defined as four activities:
    - **Corrective Maintenance**
        - A process that includes diagnosis and correction of errors.
    - **Adaptive Maintenance**

- Activity that modifies software to properly interface with a changing environment (hardware and software).

- **Perfective Maintenance**
  - Activity for adding new capabilities, modifying existing functions and making general enhancements.
  - This accounts for the majority of all effort expended on maintenance.

- **Preventive Maintenance**
  - Activity which changes software to improve future maintainability or reliability or to provide a better basis for future enhancements.
  - Still relatively rare.

- Distribution of maintenance activities (based on a study of 487 software development organizations):
  - Perfective: 50%
  - Adaptive: 25%
  - Corrective: 21%
  - Others: 4%

9.9. LESSON END ACTIVITIES

Pick up a program and its documentation from a classmate. Without contact with the classmate, exercise the program by designing and running several test cases. Could you maintain this program? If not, what additional information would you need to maintain it?

9.10. POINTS FOR DISCUSSION

1) If a software product or part of it spends 65% of its operational life cycle in maintenance, why do you suppose so little attention is paid to maintainability during the design phase?

9.11. REFERENCES

1) Ian Sommerville, Software engineering, Pearson education Asia, 6th edition, 2000.

2) James F Peters and Witold Pedryez, "Software Engineering – An Engineering Approach", John Wiley and Sons, New Delhi, 2000.

3) Pankaj Jalote- An Integrated Approach to Software Engineering, Springer Verlag, 1997.

4) Richard Fairley, "Software Engineering Concepts", Tata McGraw-Hill, 1997.

5) Roger S.Pressman, Software engineering- A practitioner's Approach, McGraw-Hill International Edition, 5th edition, 2001.

6) Shooman, "Software Reliability", McGrawHill Edition.

7) http://www.sei.cmu.edu

8) http://www.rai.com/soft_eng/sme.htm