M.Sc. (C.S.)- 09

VARDHAMAN MAHAVEER OPEN UNIVERSITY, KOTA

Software Engineering

Course Development Committee

Chairman
Prof. (Dr.) Naresh Dadhich
Vice Chancellor
Vardhaman Mahaveer Open University, Kota

Convener / Coordinator
Prof. (Dr.) D.S. Chauhan
University of Rajasthan Jaipur

Member Secretary/ Internal Coordinator
Sh. Rakesh Sharma
V.M. Open University Kota

Members

1. **Prof. (Dr.) Neeraj Bhargava**
   Department of Mathematics
   M.D.S. University Ajmer
2. **Dr. (Mrs.) Madhavi Sinha**
   Reader& Head (Computer Sc.)
   BIT, Jaipur Campus, Jaipur
3. **Prof. (Ms.) Swati V. Chande**
   Principal (Computer Sc.)
   International School of Informatics &
   Management, Jaipur
4. **Prof. (Dr.) D.P. Sharma**
   Jaipur
5. **Sh. Rajeev Shrivastava**
   LBS College Jaipur

Editing and Course Writing

Editor
Prof. Swati V. Chande
International School of Informatics & Management, Jaipur

Writers

Prof. (Dr.) Neeraj Bhargava
MDS University AJmer

Sh. Rakesh Sharma
VMOU Kota

Sh. Bharat Gupta
MIT Kota

Sh. Rajeev Shrivastava
LBS College Jaipur

Ms. Pratishtha Mathur
AIM & ACT Jaipur

Smt Varsha Gupta
MIT Kota

Academic and Administrative Arrangement

Prof. (Dr.) Naresh Dadhich
Vice Chancellor
Vardhaman Mahaveer Open University

Prof.(Dr.) M.K. Ghadoliya
Director
Academic

Yogendra Goyal
In Charge
Material Production & Distribution

Course Production

Yogendra Goyal
Assistant Production Officer
Vardhaman Mahaveer Open University

Production : May 2010 ISBN No. : 978-81-8496-210-9

Printed and published on behalf of Registrar, VM Open University Kota.
Printed by The Diamond Printing Press, Jaipur, 500 Copies

M.Sc. (C.S.) - 09

VARDHAMAN MAHAVEER OPEN UNIVERSITY, KOTA

Index

SOFTWARE ENGINEERING

UNIT – I

INTRODUCTION TO SOFTWARE ENGINEERING

Structure of the Unit

1.0 OBJECTIVES

After going through this unit students will be able to:

- Understand the meaning of software engineering
- Understand the aims, characteristics and myths of software engineering
- Learn about the various software engineering definitions, paradigms etc.
- Understand the software process, and life cycle
- Understand Computer-Aided Software Engineering

1.1 INTRODUCTION

Computers have been used for commercial purposes for many years. The more powerful a computer is, the more sophisticated programs it can run. Software engineers are required to solve larger and complex problems in cost-effective and efficient ways. The discipline of software engineering provides a systematic, cost effective and efficient approach to develop software.

1.2 SOFTWARE

Software is a logical rather than a physical system element. Software is described by its capabilities. The capabilities include the features, functions and facilities provided by a software. Software is developed by keeping in mind the hardware and operating system i.e. platform to be used by the software.

Software is a collection of computer programs, procedures, associated documentation and data. Software can also be described as a collection of programs that aim at enhancing the capabilities of the hardware to meet out the users objectives.

Software products may be -

- Generic - developed to be sold to a range of different customers, and
- Customised - developed for a single customer to suite the specific needs of the customer.

Software Applications

On the basis of their application, software can be classified into various categories. These include,

- System Software
- Real-Time Software
- Business Software
- Engineering and Scientific Software
- Embedded Software
- Personal computer software
- Artificial Intelligence software
- Web based software etc.

1.3 ENGINEERING

Engineering is the analysis , design, construction, verification, and management of technical or non-technical entities.

For the engineering of an entity, an engineer needs to identify

(i) the problem to be solved, and the characteristies of the entity.

(ii) the structure and process to construct the entity

(iii) errors and updations of enhancements in the entity.

1.4 SOFTWARE ENGINEERING

Software Engineering (SE) is the field of computer science that deals with the construction of large or complex software systems.

Computer science is concerned with theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software usine the theory and fundamentals.

The development of a large software is a group activity or say software engineering activity. A software component or utility written by one can be combined with components written by another to build a system. The components one writes may be modified by others; it may be used by others to construct different versions of system. The difference between programming and software engineering activity is that programming is primarily a personal activity, while software engineering is essentially a team activity.

Various Software Engineering definitions

Software Engineering has been defined by many authors differently. Some of these definitions are:

1. As per the IEEE Software Engineering Standards, Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.
2. Software engineering is a discipline whose aim is to produce error free software that satisfies the user's requirements and can be delivered on time within the budget.
3. Software engineering is an engineering discipline which is concerned with all aspects of software production.
4. Software engineering is a methodological and managerial discipline concerning the systematic production and maintenance of software products that are developed and maintained within anticipated and controlled time and cost limits.

The role of Software Engineer

The evolution of the software engineering field has led to a proper definition of the role of a software engineer. For the development of smaller software products, software engineer must of course be a good programmer, be well-versed in data structures and algorithms, and be fluent in one or more programming languages. These requirements however, are for the small scale software development to be done by a single individual. A software engineer is usually also involved in development of very large scale software, which requires a wider knowledge of the Software Development procedures.

A software engineer must be familiar with several design approaches, be able to translate vague requirements into precise specifications, and be able to communicate with the user of a system in terms of the application terms rather than in computer buzz words. They should also have the flexibility and openness to grasp, and become conversant with the essentials of different application areas. The software engineer needs the ability to move among several levels of abstraction at different stages of the project, from specific application procedures and requirements, to abstractions for the software system, to a specific design for the system, and finally to the detailed design testing and delivery level.

Modeling is another requirement. The software engineer must be able to build and use a model of the application to guide choices of the many trade-offs that he or she will face. The model is used to answer questions about both the behavior of the system and its performance. The model can be used by the engineer, as well as the user.

Software engineer is a member of a team and therefore needs good communication skills and interpersonal skills. The software engineer also needs the ability to schedule work, both of his or her own and that of others.

As discussed above, a software engineer is responsible for many things. In practice, many organizations divide the responsibilities among several specialists with different titles. For example, an analyst is responsible for deriving the requirements and a programmer is responsible for coding the system. A rigid fragmentation of role, however, is often counterproductive.

The qualities that a software engineer shuld posses include:

- Familiarity with several design approaches
- Ability to translate vague requirements into precise specifications
- Ability to communicate with the users of a system in terms of the application terms rather than in computer buzz words.
- Should be well versed in data structures and algorithms
- Must be a good programmer prefereably an expert in one or more programming languages
- Ability to move among several levels of abstraction at different stages of the project
- Ability to build and use a model of the application to guide choices
- Good communication skills and interpersonal skills.

1.5 EMERGENCE OF SOFTWARE ENGINEERING

Software engineering techniques have evolved over many years as a result there are series of innovations in program writing. Let us examine some developments in program writing which have contributed in software engineering discipline.

Early Computer programming

Early computer programs were very small in size and lacked sophistication. These programs were usually written in assembly languages.

High-Level Language Programming

High level languages reduced the effort required to develop software products and helped programmers to write larger programs. The software development style was still exploratory.

Control Flow-based design

As the complexity and size of the programs kept on increasing, it was difficult not only to write cost-effective and correct programs but also difficult to understand the programs written by others. To solve this problem, control flow structure was designed for the programs. A program's control flow structure indicates the sequence in which the program's instructions are executed. (Flow charting technique is used to depict the control flow of a program).

Data Structure-Oriented Design

In this type of development, programmers first pay attention to the design of the important data structures of the program than to the control structure. This technique is called data structure-oriented design.

Data Flow-Oriented Design

This technique advocates that the major data items handled by a system must be first identified and then the processing required on these data items to produce the desired outputs should be determined. The functions (called processes) and the data items that are exchanged between the different functions are represented in a diagram known as Data Flow Diagram (DFD). DFD has proven to be a generic technique which can be used to model all types of systems and not just software systems.

Object-Oriented Design

In this technique the natural objects (such as student, employee etc.) occurring in the problem are identified and then the relationships among the objects are determined. Each object essentially acts as a data hiding or data abstraction entity. It has wide acceptance due to its simplicity, code and design reuse, promise of lower development time, low cost, more robust code and easy maintenance.

1.6 AIMS OF SOFTWARE ENGINEERING

Software engineering has had to evolve its aims based on observation of thousands of projects. The aims of software engineering are:-

1. Make quality the prime objective
2. On time delivery of the software product
3. Determine the problems before writing the requirements
4. Evaluate design alternatives
5. Use an appropriate process model
6. Minimize intellectual distance
7. Ensure focus on the adaptations required as the software's environment evolves.

1.7 SOFTWARE CHARACTERISTICS

The characteristics of software are:-

- Software is developed or engineered; it is not manufactured in the classical sense.
- Software is used for decision making.
- Software doesn't wear out i.e. it works as long as it meets out the objectives.
- Most software is custom built, rather than being assembled from existing components.
- Software is flexible i.e. it can accommodate changes that emerge during its life time.

As an engineering discipline evolves, a collection of standard design reusable components is created. Reusability is an important characteristic of a high quality software component.

1.8 SOFTWARE CRISIS

The problems associated with software development are referred as "Crisis". The term refers to a set of problems that are encountered during the development of computer software. The problems are not only limited to software that does not work properly rather it encompasses problems associated with how we develop software; maintain existing software and how to meet out the pace with the growing demand for more software.

Software engineering appears to be among the few options available to tackle the present software crisis. Organizations are spending a larger portion of their budget on software. Software products are difficult to alter and debug and also often fail to meet the user requirements. They are not reliable, prone to crash and are often delivered late.

It is believed that the only best possible solution to the present software crisis can possibly come from a spread of software engineering practices among the developers.

Reasons of software crisis

- Communication between project leader and customers, software developers, and associated support staff can break down because of the special characteristics of software or the misunderstood requirements.
- If there is delay in any stage of development then scheduling does not match with actual timing.
- Quality of software may not be good because most of the developers use historical data for

development of software.

- Software developers often resist changes or make some patch work to reduce their efforts.

Table 1.1 : Software crisis from programmer's and user's point of view

Programmer's View	User's View
Problem of portability	Software cost is very high
Problem in documentation	Hardware goes down very often.
Problem in co-ordination of work with different people	Problem of different versions of software
Problem of maintenance in proper manner	Problem of bugs
Problem of piracy of software	Lack of specialization in development

1.9 SOFTWARE MYTHS

Software Myths- beliefs about software and the process used to build it - can be traced to the earliest days of computing. Myths have a number of attributes that have made them insidious. For instance, myths appear to be reasonable statements of fact, they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score".

Management Myths

Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if the belief will lessen the pressure.

Myth : We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?

Reality : The book of standards may very well exist, but is it used?

- Are software practitioners aware of its existence?
- Does it reflect modern software engineering practice?
- Is it complete? Is it adaptable?
- Is it streamlined to improve time to delivery while still maintaining a focus on Quality?

In many cases, the answer to these questions is no.

Myth : If we get behind schedule, we can add more programmers and catch up (sometimes called the Mongolian horde concept).

Reality : Software development is not a mechanistic process like manufacturing. In the words of Brooks: "Adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort.

Myth : *If we decide to outsource the software project to a third party, we can just relax and let that firm build it.*

Reality : If an organization does not understand how to manage and control software project internally, it will invariably struggle when it out sources software project.

Customer Myths

A customer who requests computer software may be a person at the next desk, a technical group in the organization, the marketing /sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations and ultimately, dissatisfaction with the developers.

Myth : *A general statement of objectives is sufficient to begin writing programs, we can fill in details later.*

Reality : Although a comprehensive and stable statement of requirements is not always possible, an ambiguous statement of objectives is a recipe for disaster. Unambiguous requirements are developed only through effective and continuous communication between customer and developer.

Myth : *Project requirements continually change, but change can be easily accommodated because software is flexible.*

Reality : It's true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirement changes are requested early, cost impact is relatively small. However, as time passes, cost impact grows rapidly – resources have been committed, a design framework has been established, and then a change can cause upheaval that requires additional resources and major design modification.

Figure 1 depicts the increase in cost ot change with time from the requirements phase to the production phase.

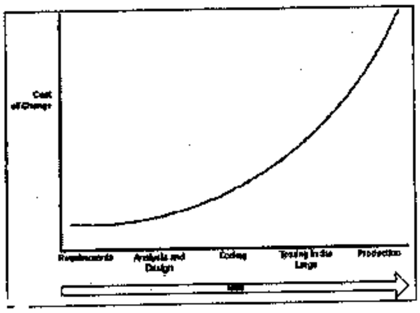


Figure 1.1 : The cost of change increases with time

1.10 SOFTWARE PRODUCT AND QUALITIES OF SOFTWARE PRODUCT

The goal of any engineering activity is to build a product. The product of software engineering is a software system but the difference between software product and other products is that it is modifiable. The user wants the software product to be reliable and user friendly. The designer of the software wants it to be maintainable and portable.

Qualities of Software Product

1. **Correctness** - A program is functionally correct if it behaves according to the specification of the fanctions it should provide (called functional requirements specifications). The definition of correctness assumes that a specification of the system is available and that it is possible to determine unambiguously whether or not a program meets the specifications. Correctness is a mathematical property that establishes the equivalence between the Software and its specification.

2. **Reliability** - Software is reliable if the user can depend on it. The notion of reliability is on the other hand, relative; if the consequence of a software error is not serious, the incorrect software may still be reliable. Software products are commonly released along with a list of known bugs. Users of software take it for granted that Release 1 of a product is buggy. This is one of the most striking symptoms of the immaturity of the software engineering field as an engineering discipline. Software design errors are generally treated as unavoidable. Whereas with all other products the customer receives a guarantee of reliability, with software we get a disclaimer that the software manufacturer is not responsible for any damages due to product errors. Software engineering can truly be called an engineering discipline only when we can achieve software reliability comparable to the reliability of other engineering products.

3. **Robustness** - A program is robust if it behaves reasonably, even in circumstances that were not anticipated in the requirements specification - for example, when it encounters incorrect input data or some hardware malfunction (say, a disk crash). Robustness and correctness are strongly related without a sharp dividing line between them.

4. **User Friendliness** - A software system is user friendly if its human users find it easy to use. This definition reflects the subjective nature of user friendliness. An application used by novice programmers qualifies as user friendly by virtue of different properties than an application that is used by expert programmers. For example a inovice user may appreciate verbose messages, while an experienced user grows to detest and ignore them. Similarly, a nonprogrammer may appreciate the use of menus, while a programmer may be more comfortable with typing commands.

5. **Verifiability** - A software system is verifiable if its properties can be verified easily. Verification can be performed either by formal analysis methods or through testing. Verifiability is usually an internal quality, although it sometimes becomes an external quality also.

6. **Maintainability** - Software maintenance is used to refer to the modifications that are made to software at any time according to the user's requirements or to fix bugs. Usually maintenance costs exceed 60% of the total costs of software. On the basis of the type of modification done, maintenance cost can be divided into three categories – Corrective, adaptive and perfective.

7. **Reusability** - Reusability appears to be more applicable to software components than to whole products but it certainly seems possible to build products that are reusable.

8. **Portability** - Software is portable if it can run in different environments. The term environment can refer to a hardware platform or a software environment such as a particular operating system. Portability refers to the ability to run a system on different hardware platforms.

9. **Data Abstraction** - Abstraction is a process where we identify the important aspects of a phenomenon and ignore its details. Thus, abstraction is a special case of separation of concerns wherein we separate the concern of the important aspects from the concern of the unimportant details. Data abstraction is a concept which encapsulates data structures and well defined procedures/ functions in a single unit. This encapsulation forms a wall which is intended to shield data representation from computer users.

10. **Modularity** - A complex system may be divided into smaller pieces called modules. A system composed of modules is called modular. The main benefit of modularity is that it allows the principle of separation of concerns to be applied in two phases: when dealing with the details of each module in isolation (and ignoring details of other modules); and when dealing with the overall characteristics of all modules and their relationships in order to integrate them into a coherent system.

1.11 SOFTWARE PROCESS

A software process defines the approach that is taken as software is engineered. Software engineering is performed by creative, knowledgeable people who should work within a defined and mature software process.

Software engineering is a layered technology. Figure 1.2 shows the layers in Software Engineering. Any engineering approach must rest on an organizational commitment to quality.

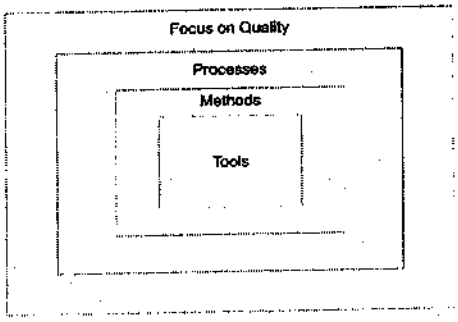


Figure 1.2 : Software Engineering Layers

The bedrock that supports software engineering is a focus on quality.

The foundation for software engineering is the process layer. Process defines a framework for a set of key process areas. They key process areas form the basis for management control of software projects and establish the context in which technical methods are applied, work products are produced, quality is ensured and change is managed.

The software engineering methods rely on a set of basic principles and provide the technical support for building software. Methods complete the tasks that include requirements analysis, design, program construction, testing and maintenance.

Software engineering tools provide automated or semi-automated support for the process and the methods. CASE(Computer-Aided Software Engineering) tools combine software, hardware and a software engineering database(a repository containing information about analysis, design, program construction and testing) to create a software engineering environment.

Key Process Areas (KPAs) - The KPAs of Software Engineering are described by the following characteristics:-

- Goals – the overall objectives to achieve
- Commitments – Requirements that must be met to achieve the goals
- Abilities – Things that must be in place organizationally and technically
- Activities – The specific tasks to achieve the KPA function
- Methods for monitoring implementation
- Methods for verifying implementation

The Software Process

Software process is a framework for the tasks that are needed to construct quality software. It defines the approach that is taken as software in engineered. The activities carried out in a software process are shown in figure 1.3.

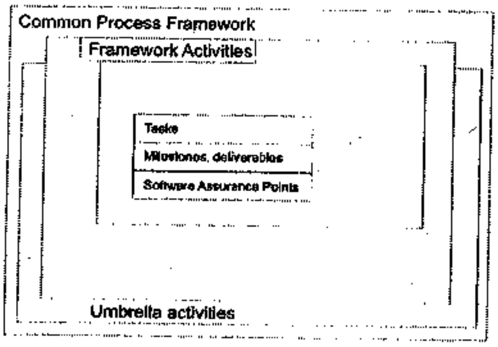


Figure 1.3: Software Process

A common process framework is established by defining a small number of framework activities that are applicable to all software projects.

A number of task sets – collections of S.E. work tasks, project milestones, software work products and software quality assurance points.

Umbrella activities are independent of any one framework activity and occur throughout the process.

1.12 SOFTWARE ENGINEERING PARADIGMS

To solve actual problems in an industry, a software engineer or a team of engineers must incorporate a development strategy that encompasses the process methods and tools layers and the generic phases. This strategy is referred as a process model or a software engineering paradigm. A process model for software engineering is chosen based on the nature of the project. Different software engineering paradigms or process models are covered in unit-4.

1.13 SOFTWARE LIFE CYCLE

Software life cycle is a series of identifiable stages that a software product undergoes during its lifetime. A graphical view of the software development life cycle, that provides a visual explanation of the term waterfall used to denote it is given in figure 1.4. Each phase yields results that flow into the next and the process ideally proceed in an orderly and linear fashion.

Adherence to a life cycle model during software development has become universally accepted by software development organizations. The advantage of adhering to a life cycle model is that it encourages development of software in a systematic and disciplined manner. When a single programmer develops a softwares, he has the freedom to decide the steps of development, but when a software development team works on a project, it is necessary to have a precise understanding among team members about what and when to do specific job. Otherwise it may lead to confusion or failure of the project. It is possible that one member may decide to write the program and other may decide to prepare the test data for the project.

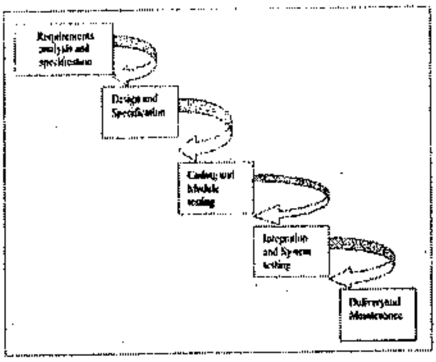

Requirements analysis and specification
Design and Specification
Coding and Module testing
Integration and System testing
Delivery and Maintenance

Figure 1.4 : Waterfall model of the Software Life Cycle

From the inception of an idea for a software system, until it is implemented and delivered to a customer, and even after that, the system undergoes several changes. The software is said to have a life cycle composed of several phases. Each of these phases results in the development of either a part of the system or something associated with the system, such as a test plan or a user manual. In the traditional life cycle model, called the waterfall model, or the Liner Seqnential Model each phase has well-defined starting and ending points, with clearly identifiable inputs to the next phase. In practice, it is rarely so simple.

A sample waterfall life cycle model comprises the following phases:-

- **Requirements analysis and specification**: Requirements analysis is usually the first phase of a large-scale software development project. It is undertaken after a feasibility study has been performed to define the precise costs and benefits of a software system. The purpose of this phase is to identify the exact requirements for the system. Such study may be performed by the customer, the developer, a marketing organization, or any combination of the three. In cases where the requirements are not clear, e.g. for a system that has never been developed before - much interaction is required between the user and the developer. The requirements at this stage are in end- user terms. Various software engineering methodologies advocate that this phase must also produce user manuals and system test plans.

- **Design and specification**: Once the requirements for a system have been documented, software engineers design a software system to meet them. This phase is sometimes split into two sub phases: architectural or high-level design and detailed design. High-level design deals with the overall module structure and organization, rather than the details of the modules. The high- level design is refined by designing each module in detail (detailed design). Separating the requirements analysis phase from the design phase is an instance of a fundamental what/how dichotomy that we encounter quite often in computer science. The general principle involves making a clear distinction between what the problem is and how to solve the problem. In this case, the requirements phase attempts to specify what the problem is. There are usually many ways that the requirements may be met, including some solutions that do not involve the use of computers at all. The purpose of the design phase is to specify a particular software system that will meet the stated requirements. Again, there are usually many ways to build the specified system. In the coding phase, which follows the design phase, a particular system is coded to meet the design specification.

- **Coding and module testing**: This is the phase that produces the actual code that will be delivered to the customer as the running system. The other phases of the life cycle may also develop code, such as prototypes, tests, and test drivers, but these are for use by the developer. Individual modules developed in this phase are also tested before being delivered to the next phase.

- **Integration and system testing**: All the modules that have been developed before and tested individually are put together - integrated - in this phase and tested as a whole system.

- **Delivery and maintenance**: Once the system passes all the tests, it is delivered to the customer and enters the maintenance phase. Any modifications made to the system after initial delivery are usually attributed to this phase.

A commonly used terminology distinguishes between high phases and low phases of the software life cycle: the feasibility study, requirements analysis, and high-level design contribute to the former, and implementation-oriented activities contribute to the latter.

The process may be decomposed into a different set of phases, with different names, different purposes, and different granularity. Entirely different life cycle schemes may even be proposed, not based on a strictly phased waterfall development. For example, it is clear that if any tests uncover defects in the system, we have to go back at least to the coding phase and perhaps to the design phase to correct some mistakes. In general, any phase may uncover problems in previous phases; this will necessitate going back to the previous phase and redoing some earlier work. For example, if the system design phase uncovers inconsistencies or ambiguities in the system requirements, the requirements analysis phase must be revisited to determine what requirements were really intended.

Another simplification in the above presentation is that it assumes that a phase is completed before the next one begins. In practice, it is often expedient to starts a phase before a previous one is finished. This may happen, for example, if some data necessary for the completion of the requirements phase will not be available for some time. Or it might be necessary because the people ready to start the next phase are available and have nothing else to do.

Drawbacks of Linear Sequential Model

- Real projects rarely follow the sequential flow that the model proposes.
- It is difficult for the customer to state all requirements explicitly in the beginning.
- The customer must have patience to get the working version of the software.
- A major mistake left out initially will make the entire software non workable.
- Developers are often delayed unnecessarily.

1.14 COMPUTER-AIDED SOFTWARE ENGINEERING (CASE)

CASE is a tool which aids a software engineer to maintain and develop software. The workshop for software engineering is called an Integrated Project Support Environment (IPSE) and the tool set that fills the workshop is called CASE. It is an automated support tool for software engineers in any software engineering process.

Software engineering mainly includes the following processes:-

- Translation of Customer/user needs into software requirements
- Translate software requirements into design specification
- Implementation of design into code
- Testing of the code
- Documentation

CASE technology provides software process support by automating some process activities and by providing information about the software, which is being developed. Some example activities which can be automated using CASE are:-

- The generation of user interfaces from a graphical interface description, which is created interactively by the user.

- To understand the design by using a data dictionary, which holds information about the entities and relations that exist in a design.
- Program debugging through the provision of information about an executing program.

Levels of CASE

There are three different levels of CASE technology:-

- **Production process support technology** – This includes support for process activities such as specification, design, implementation, testing and so on.
- **Process management and technology** – It includes tools to support process modeling and process management. These tools are used for specific support activities.
- **Meta case technology** – Meta-CASE tools are generators, which are used to create production process management support tools.

Architecture of CASE Environment

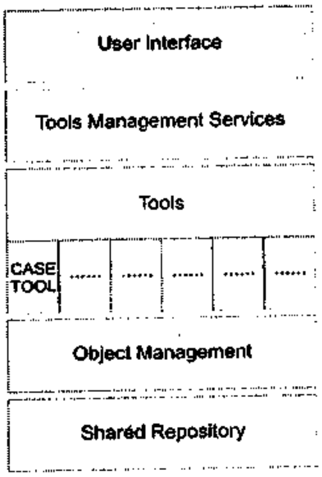


Figure 1.5 : CASE Environment

The important components of a modern CASE environment are:-

- User Interface
- Tools Management System (Tools set)
- Object Management System (OMS)
- Repository

a) **User Interface** – It provides a consistent framework for accessing different tools making easier for the user to interact with different tools.

b) **Tools Management System** – The tools set holds the different types of improved quality tools. Tools management service (TMS) controls the behavior of tools within the environment.

c) **Object Management System** – It maps specification design, text data, project plan etc. into the underlying storage management system i.e. repository. This component provides integration services, a set of standard modules that couple tools with the repository. It also provides Configuration Management services by enabling the identification of all configuration objects.

d) **Repository** – It is the CASE database and the access control functions that enable the Object

Management Layer to interact with the database. CASE repository can be a project database, IPSE database, data dictionary, CASE database and so on.

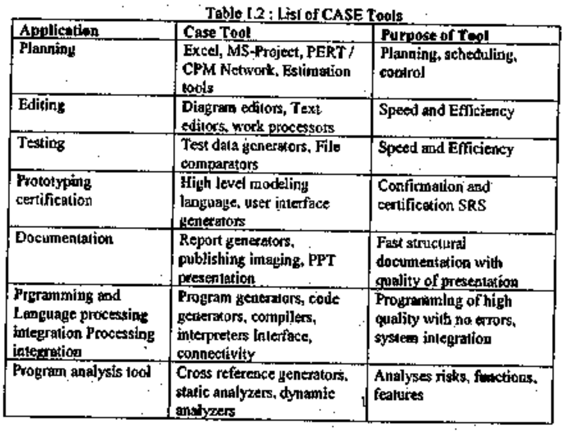
Objectives of CASE

- Improve the objectives of CASE are to, Productivity – Use f CASE increase the speeds with which systems are designed and developed.
- Improve Information System quality
    - Ease and improve the testing process through the use of automated checking
    - Improve the integration of development activities
    - Improve the quality and completeness of documentation
    - Standardize the development process
    - Management of the project improves
    - Time reduces in the overall construction process
    - Improve software portability across environments
- Improve Effectiveness – Effectiveness means doing the right task. CASE tools can suggest procedures to approach a task.

Reasons of rejections of CASE by organizations

- The initial cost of purchasing and using of CASE is high
- Training cost of personnel
- Benefits of using CASE are in the later stages of SDLC
- CASE increases the length of duration in initial stages
- CASE tools cannot easily share information between tools
- Different methodology standards within organization, CASE forces to follow a specific methodology for system development
- Lack of confidence in CASE products

Table 1.2 : List of CASE Tools



Application	Case Tool	Purpose of Tool
Planning	Excel, MS-Project, PERT / CPM Network, Estimation tools	Planning, scheduling, control
Editing	Diagram editors, Text editors, work processors	Speed and Efficiency
Testing	Test data generators, File comparators	Speed and Efficiency
Prototyping certification	High level modeling language, user interface generators	Confirmation and certification SRS
Documentation	Report generators, publishing imaging, PPT presentation	Fast structural documentation with quality of presentation
Prgramming and Language processing integration Processing integration	Program generators, code generators, compilers, interpreters Interface, connectivity	Programming of high quality with no errors, system integration
Program analysis tool	Cross reference generators, static analyzers, dynamic analyzers	Analyses risks, functions, features

Advantages of CASE Tools

The major benefits of using CASF tools include:-

- Improved productivity

- Better documentation
- Improved accuracy
- Intangible benefits
- Improved quality
- Reduced lifetime maintenance
- Opportunity to non-programmers
- Reduced cost of software
- Produce high quality and consistent documents
- Impact on the style of a working of company

1.15 SUMMARY

Software engineering is the field of computer science that deals with the construction of large or complex software system. Software engineering is essentially a team activity. Software engineering is a discipline whose aim is to produce error free software that satisfies the user's requirements and can be delivered on time within the budget. The software engineer must be familiar with several design approaches, be able to translate vague requirements into precise specifications, and be able to communicate with the user of a system in terms of the application terms rather than in computer buzz words. Software engineering has had to evolve its aims based on observation of thousands of projects.

The problems associated with software development are referred as "Crisis". The term refers to a set of problems that are encountered during the development of computer software. Software Myths- beliefs about software and the process used to build it - can be traced to the earliest days of computing.

The product of software engineer is a software system but the difference between software product and other product is that it is modifiable. The designer of the software wants it to be maintainable and portable.

A software process defines the approach that is taken as software is engineered. Software engineering is a layered technology. Any engineering approach must rest on an organizational commitment to quality. A software life cycle is the series of identifiable stages that a software product undergoes during its lifetime. Adherence to a life cycle model during software development has become universally accepted by software development organizations.

CASE is a tool which aids a software engineer to maintain and develop software. It is an automated support tool for the software engineers in any software engineering process. CASE technology provides software process support by automating some process activities and by providing information about the software, which is being developed.

1.16 UNIT-END QUESTIONS

1 What is meant by Software Engineering?

2 What are the characteristics of Software?

3 Define Software Lifecycle.

4 What is CASE? Explain.

5 What is software Crisis? Why it happens?

6 What do you mean by software process?

7 What are the aims of Software Engineering?

8 What are Software Myths?

9 Explain the role of software engineer.

10 Explain different types of software applications.

1.17 FURTHER READINGS

1. Fundamentals of Software Engineering, Rajib Mall, Prentice Hall of India.
2. Software Engineering - A Practitioner's Approach, Roger S. Pressman, McGrawHill

UNIT – II

SOFTWARE REQUIREMENTS ANALYSIS

Structure of the Unit

2.0 OBJECTIVES

After going through this unit students will be able to:

- Understand the meaning of requirement engineering
- Understand the ways for identifying precise requirements of customer
- Understand the Software Requirement Specification document
- Identify the functional and non-functional requirements
- Understand the feasibility studies techniques

2.1 INTRODUCTION

Requirement analysis is a software engineering task that bridges the gap between system-level software allocation and software design. Requirement analysis enables the system engineer to specify software function and performance, indicate software's interface with other system elements and establish constraints that software must meet. Requirement analysis allows the software engineer to refine the software allocation and build models of the data, functional and behavioral domains that will be treated by software.

Requirement analysis provides software designer with models that can be translated in to data, architectural, interface and procedural design.

Requirements analysis in systems engineering and software engineering, encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, Requirements analysis is the first stage in the systems engineering process and software development process.

Systematic requirements analysis is also known as requirements engineering. It is sometimes referred to loosely by names such as requirements gathering, requirements capture, or requirements specification. The term requirements analysis can also be applied specifically to the analysis proper.

Requirements analysis is critical to the success of a development project. Requirements must be actionable, measurable, testable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design. Figure 2.1 shows the components of the Analysis phase.

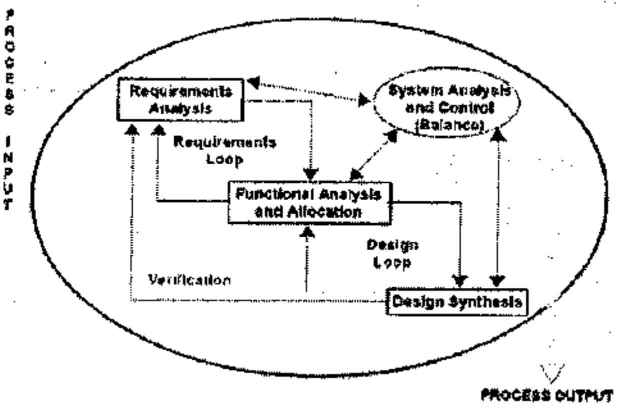


Figure 2.1 : System Analysis

The requirement analysis and specification phase starts once the feasibility study phase is complete and the project is found to be financially sound and technically feasible. The goal of the requirement analysis and specification is to clearly understand the customer requirements and to systematically organize these requirements in a specification document.

2.2 IDENTIFYING SOFTWARE REQUIREMENTS

The requirements can be classified into the following types:-

- Requirements that should be absolutely met.
- Requirements that are highly desirable but not necessary.
- Requirements that are possible but could be eliminiated.

On the basis of the characteristics of requirements, the requirements are classified into the following two types: Functional Requirements and Non Functional Requirements.

2.2.1 Functional Requirements

Functional REquirements are the functionalities required from the system. They define factors like I/O formats, storage structure, computational capabilities, timing etc.

To document the functional requirements of a system, it is essential to learn how to first identify the high-level functional requirements of the system. Through, the execution of a high-level requirement, user can get some useful work done. Each requirement typically involves accepting some data from the user, converting it to the required response and outputting the response to the user. Each high-level functional requirement may involve a series of interactions between the system and one or more users.

Initially some data is given by the user, the system may display some response (called system action). The user may input further data and this process may continue as per the requirement. There can be different interaction scenarios or sequences depending on the user's option selection. The different scenarios are essentially the different paths.

In requirements specification, it is important to define the precise data input to the system and the precise data output by the system.

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem. There can be many types of users of a system and their requirements or expectations from the system may be different. It is often useful to identify the different types of users who might use the system and then try to identify the requirements from each user's perspective.

2.2.2 Non Functional Requirements

They define the characteristics/properties or qualities of a product including usability, efficiency, performance, space, reliability, accuracy etc. Examples of nonfunctional requirements include aspects concerning maintainability, portability and usability. The non-functional requirements may also include reliability issues, accuracy of results, human-computer interface issues and constraints on system implementation.

2.3 USER REQUIREMENTS

User requirements include statements of fact and assumptions that define the expectations of the system in terms of mission objectives, environment, constraints, and measures of effectiveness and suitability. The users are those that perform the eight primary functions of systems engineering, with special emphasis on the operator as the key user. Operational requirements will define the basic need and, at a minimum, answer the questions posed in the following listing:

- Operational distribution or deployment: Where will the system be used?
- Mission profile or scenario: How will the system accomplish its mission objective?
- Performance and related parameters: What are the critical system parameters to accomplish the mission?
- Utilization environments: How are the various system components to be used?
- Effectiveness requirements: How effective or efficient must the system be in performing its mission?
- Operational life cycle: How long will the system be in use by the user?
- Environment: What environments will the system be expected to operate in an effective manner?

2.4 SYSTEM REQUIREMENTS

Systement requirements can be specified under the following heads

- **Performance Requirements:** Performance requirements state the extent to which a mission or function must be executed; generally measured in terms of quantity, quality, coverage, timeliness or readiness. During requirements analysis, performance (how well does it have to be done) requirements will be interactively developed across all identified functions based on system life cycle factors; and characterized in terms of the degree of certainty in their estimate, the degree of criticality to system success, and their relationship to other requirements.
- **Design Requirements:** Design requirements include the "build to," "code to," and "buy to" requirements for products and "how to execute" requirements for processes expressed in technical data packages and technical manuals.
- **Derived Requirements:** Derived Requirements are requiements that are implied or transformed from higher-level requirement. For example, a requirement for long range or high speed may result in a design requirement for low weight.
- **Allocated Requirement:** Allocated requirement is a requirement that is established by dividing or otherwise allocating a high-level requirement into multiple lower-level requirements.

2.5 FEASIBILITY STUDY

The aim of the feasibility study activity is to determine whether the product would be worth or not. The type of study determines if a project can and should be taken. The feasibility study activity involves the analysis of the problem and collection of all relevant information relating to the product i.e. input to the system, processing required on input data, output required to be produced by the system and various constraints on the system. For the conduct of the feasibility study, the developer will usually consider the following types of inter-related feasibility study:-

- **Technical Feasibility** – It emphasises on the configuration of the system than the actual make of hardware. The configuration should give the complete system's requirement.
- **Operational feasibility** – Its emphasises on human, organizational and political aspects. i.e. what changes, skills and structure is required.
- **Economic Feasibility** – It is the technique for evaluating the effectiveness of a proposed system. It is also known as cost/benefit analysis and determines the benefits and savings that are expected from a proposed system.
- **Management Feasibility** – It determines whether the proposed system is acceptable to management. If management does not support the system then the system is considered as non-

feasible.

- **Time Feasibility** – It determines whether the system can be implemented within a stipulated time frame. If project takes longer time then it is likely to be rejected.
- **Legal Feasibility** – It checks whether the proposed system is within the legal aspects of the country.
- **Social Feasibility** – It determines whether a proposed project will be acceptable to the people or not.

2.6 PROCESS OF REQUIREMENT ENGINEERING

Requirement engineering process takes the following steps-

- Requirement Elicitation (gathering)
- Requirement analysis and modeling
- Requirement documentation
- Requirement Review
- Requirement Management

Figure 2.2 shows the steps in the software requiements engineering process:

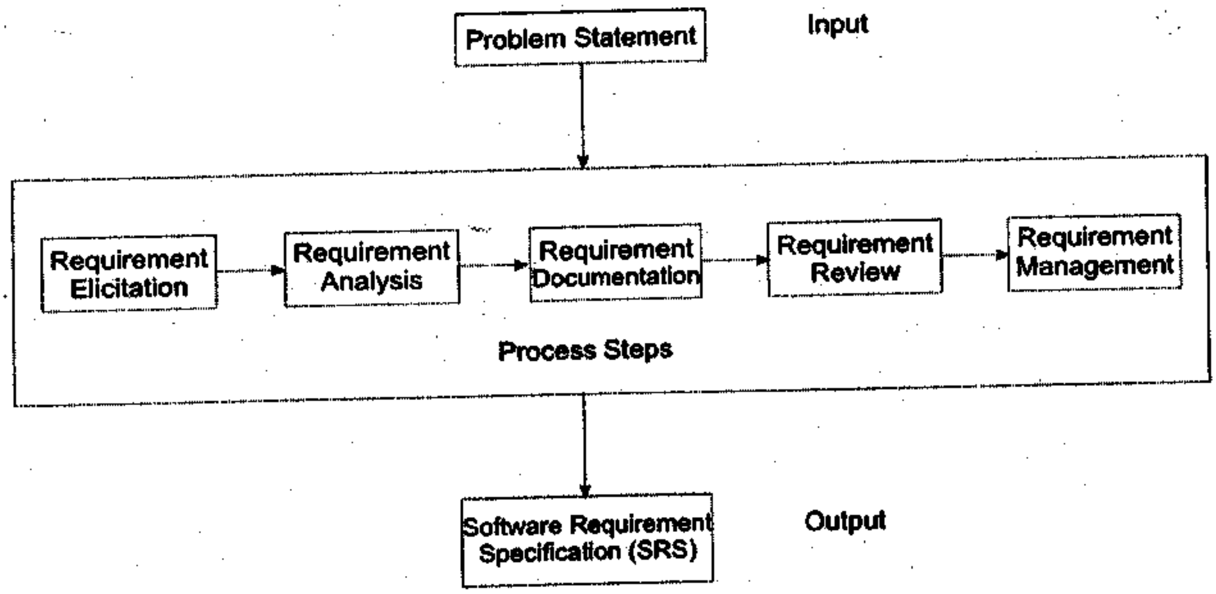


Figure 2.2 : Requirement Engineering Proces

2.6.1 Requirement Elicitation

Requirement Elicitation is the practice of obtaining the requirements of a system from users, customers and other stakeholders. The practice is also sometimes referred as requirements gathering. Requirements elicitation practices include interviews, questionnaires, user observation, workshops, brain storming, and prototyping.

Information is gathered from the multiple sources. The Requirements Analyst draws out from each of these groups what their requirements from the application are and what they expect the application to accomplish. Considering the multiple sources involved, the list of requirements gathered in this manner could run into pages. The level of detail of the requirements list is based on the number and size of user groups, the degree of complexity of business processes and the size of the application.

The process model of requirement elicitation & analysis is shown in figure 2.3.

Problems faced in Requirements Elicitation

- Ambiguous understanding of processes
- Inconsistency within a single process by multiple users
- Insufficient input from sources
- Conflicting interests of multiple sources
- Changes in Requirements after project has begun

2.6.2 Requirement analysis and modeling

Once all requirements have been gathered, a structured analysis of these can be done after modeling the requirements. Some of the Software Requirements Analysis techniques used are requirements animation, automated reasoning, knowledge-based critiquing, consistency checking, analogical and case-based reasoning.

In this phase, each requirement is analyzed from the point of view of consistency, validity and feasibility for organization's consideration. Consistency confirms that it does not conflict with other requirements but supports others where necessary. Validity conforms its relevance to goals and objectives. Feasibility ensures that the necessary inputs are available without error and technology support is possible to execute the requirements.

The technical software development staff works with users to find out -

- what operations the system should provide
- required performance of the system
- hardware/software constraints etc.

This stage depends on the user acceptance i.e. how well the system achieves the user's needs and supports the work to be automated.

Requirements Analysis is a difficult process because,

- Users often don't know what they want from the system.
- Different users have different types of requirements and they express this in different ways.
- Since analysis takes place in an organizational context, political and other factors in organizations may affect the requirements of the system.
- The economic and business environment in which analysis takes place is dynamic i.e. New requirements may emerge from the users who were not originally involved.

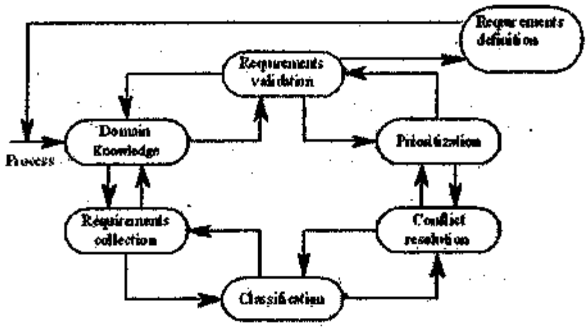


Figure 2.3 : Process model of Requirements elicitation and analysis

Each organization has its own generic process model of elicitation and analysis depending on the factors such as skills of staff, standards used etc.

The process activities shown above are:-

- **Domain Knowledge** – Knowledge of the domain must be clearly understood by the analysts i.e. if a system for any manufacturing industry is to be developed then the analyst must acquire complete knowledge about the manufacturing process.
- **Requirements collection** – It is the process of interacting with the users in the system to find out their requirements. In this step, domain knowledge develops further and is greatly improved.
- **Classification** – This activity receives the unstructured collection of requirements and organizes them into clusters.
- **Conflict resolution** – When many users are involved, requirements may conflict. This step is concerned with finding and resolving these conflicts.
- **Prioritization** – There will be some activities more important than others in the system. In this step, the most important requirements are discovered through interaction with the users of the system.
- **Requirements validation** – The requirements are validated to find whether they are complete, consistent and in accordance with the requirements of the users.

2.6.3 Requirements Documentation

Requirements documentation is also called Software Requirement Specification (SRS). It is a very important activity which is carried out after requirement elicitation and analysis. It is the way to present requirements in a consistent format. It is a specification for a software product.

SRS could be written by a customer or a developer of the system. The SRS written by the user is used to define the needs and expectations of the users and SRS written by the developer serves as a contract document between both the user and the developer.Therefore requirements must be written so that they are meaningful not only to the customers but also to designers of development team.

Requirements Definition

A system requirements definition is description of functions/services which the system should provide and the constraints under which the system must operate. It should only specify the external behaviour of the system it should not be concerned with system design characteristics.

The requirements should not be defined using an implementation model i.e. the definition should be written in such a way that it is understandable to users. The requirements may either be functional or non functional.

The functional requirements should be -

- **Consistent** - The requirements should not have contradictory definitions.
- **Complete** - All functions required by the user should be defined.

2.6.4 Requirements Review

The purpose of the Requirements Review is to review the System Requirements Specification document to ensure that the documented requirements reflect the current knowledge of the user and market requirements, to identify requirements that may not be consistent with product development constraints, and to put the requirements document under version control to serve as a baseline for continuous new product development. The Review checks include -

- Adherence to guidelines
- Individual requirements checks
- Fitness for purpose.

A review is a manual process, requirements reviews can be informal or formal.

Informal Review

It simply involves developers discussing requirements with as many as system users as much possible. Many problems can be detected by talking about the system to users before making a commitment to a formal review.

Formal Review

In this review, the developers explain the system requirements and the implications of each requirement to the user. The review team must check each requirement for consistency and must check the requirements as a whole for completeness. Reviewers must check the following:

- **Verifiability** – Is the requirement as stated, testable?
- **Comprehensibility** – It is the requirement property understood by the users of the system.
- **Tracebility** – It is the process of going back to the source of the requirement to assess the impact of any change. It also checks the impact of change on the rest of the system.
- **Adaptability** – It is requirement that user can accept the system.

2.6.5 Requirements Management

Requirement management is defined as a systematic approach to eliciting, organizing and documenting the requirements of the system and a process that establishes and maintains agreement between user and developers team on the changing requirements of the system. Planning is an essential stage in the requirements management process. Requirements management stage is very expensive for each project. The planning establishes the level of requirements management detail. The steps involved in requirement management are:-

- Requirement identification
- Change management process
- Traceability policies

- CASE tool support

2.7 SOFTWARE REQUIREMENTS SPECIFICATION DOCUMENT (SRS)

SRS stands for Software Requirements Specification. It establishes the basis for agreement between users and developers or suppliers on what the software product is expected to do, as well as what it is not expected to do. Some of the features of SRS are –

- It sets a rigorous assessment of requirements before design can begin.
- It sets the basis for software design, test, deployment, training etc. It also sets pre-requisite for a good design though it is not enough.
- It sets basis for software enhancement and maintenance.
- It sets basis for project plans like Scheduling and Estimation.

This document is generated as output of requirement analysis. SRS should be consistent, correct, unambiguous and complete. The developer can prepare the SRS after detailed discussions with the user. Requirement analysis is a software engineering task that bridges the gap between system-level software allocation and software design. Requirement analysis enables the system engineer to specify software function and performance which indicates software's interface with other system elements and establish constraints that software must meet. Requirement analysis allows the software engineer to refine software allocation and build models of the data, functional and behavioral domains that will be treated by software.

The SRS is produced at the culmination of the analysis task. The function and performance allocated to software as part of system engineering are refined by establishing a complete information description, a detailed functional and behavioral description, an indication of performance requirements and design constraints, appropriate validation criteria and other data pertinent requirements.

The framework for the SRS that can be used for our purposes is as under:-

1. Introduction
    a. Scope
    b. Overview
    c. System Reference
    d. Overall description
    e. Software project constraints
2. Overall Information Description
    a. Information content representation
    b. Information flow representation
        i. Data Flow
        ii. Control Flow
    c. System Interfaces
    d. Hardware and Software interfaces
    e. Operating Environment

f. User Environment

3. Functional Description

a. Functional Partitioning

b. Functional Description

i. Processing Narrative

ii. Constraints

iii. Performance Requirements

iv. Design Limitations

v. Supporting Diagrams

c. Control Description

i. Control Specification

ii. Design Constraints

4. Behavioral Description of the system

a. Various System states

b. Events and actions of systems

5. Other non-functional attributes

a. Security

b. Binary Compatibility

c. Reliability

d. Maintainability

e. Portability

f. Extensibility

g. Reusability

h. Application Affinity/Compatibility

i. Resource Utilization

j. Serviceability

k. ... others as appropriate

6. Validation and Criteria

a. Performance boundaries

b. Classes of tests

c. Expected software response

d. Special considerations

7. Bibliography

8. Appendix

The objectives of the sections are as follows,

1. **Introduction** – states the goals and objectives of the software, describing it in the context of the computer based system.

2. **Overall Information Description** – provides a detailed description of the problem that the software must solve. In this section information content and relationships, flow and structure are documented.

3. **Functional Description** – provides the description of each function required to solve the problem. A processing narrative is provided for each function, design constraints are stated and justified, performance characteristics are stated and one or more diagrams are included to graphically represent the overall structure of the software.

4. **Behavioral Description of the System** – examines the operation of the software as a consequence of external events and internally generated control characteristics.

5. **Validation and Criteria** – provides answers to questions like – how do we recognize a successful implementation? What classes of tests must be conducted to validate function, performance and constraints?

6. **Bibliography and Appendix** – contains references to all documents that relate to the software. It includes other software engineering documentation, technical references, vendor literature and standards. The appendix contains information that supplements the specification like tabular data, charts, graphs etc.

Characteristics of a Software Requirements Specification

A good SRS is

- unambiguous,
- complete,
- verifiable,
- consistent,
- modifiable,
- traceable, and
- Usable during the operation and maintenance phase.

Unambiguous

- Every requirement has only one interpretation.
- Each characteristic of the final product is described using a single unique term.
- A glossary should be used when a term used in a particular context could have multiple meanings.

Complete

A complete SRS must possess the following qualities:

- inclusion of all significant requirements,
- definition of the responses of the software to all realizable classes of input,
- conformity to any standard that applies to it,
- full labeling and referencing of all tables and diagrams and the definition of all terms.

Verifiable

- Every requirement must be verifiable.
- There must exist some finite cost-effective process with which a person or machine can check that the software meets the requirement.

Consistent

- No set of individual requirements described in the SRS can be in conflict.
- Types of likely conflicts:
- Two or more requirements describe the same real world object in different terms.
- The specified characteristics of real world objects might conflict.
- There may be a logical or temporal conflict between two specified actions.

Modifiable

- The structure and style of the SRS are such that any necessary changes to the requirements can be made easily, completely and consistently.
- Requirements:
  - a coherent and easy-to-use organization (including a table of contents, index and cross-referencing),
  - not be redundant - this can lead to errors.

Traceable

- The origin of each requirement must be clear.
- The SRS should facilitate the referencing of each requirement in future development or enhancement documentation.
- Types:
  - Backward traceability
    - Each requirement must explicitly reference its source in previous documents.
  - Forward traceability
    - Each requirement must have an unique name or reference number.

Usable during the operation and maintenance phase

The SRS must address the needs of the operation and maintenance phase, including the eventual replacement of the software.

Benefits of Software Requirements Specification

The following are the benefits of a good Software Requirement Specification (SRS):

- Congruency between the Users/Customers (stakeholders) and the suppliers/developers on what the software product is to do. The complete description of the functions to be performed by the software specified in the Software Requirement Specification help the potential users to determine if the software in question adheres to their needs.
- Reduce the development effort. The preparation of the Software Requirement Specification forces various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the Software Requirement Specification can reveal omissions,

misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.

- Provide a basis for estimating costs and schedules. The description of the product to be developed as given in the Software Requirement Specification is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates.
- Provide a baseline for validation and verification. Organizations can develop their validation and Verification plans much more productively from a good Software Requirement Specification. As a part of the development contract, the SRS provides a baseline against which compliance can be measured.
- Facilitate transfer. The Software Requirement Specification makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.
- Serve as a basis for enhancement. Because the Software Requirement Specification discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued production evaluation.

2.8 SYSTEM MODELS

System models are an important element of the system engineering process. The software engineer creates models that:-

- Define the processes that serve the needs of the vision under consideration.
- Characterize the behavior of the processes and the assumptions on which the behavior is based.
- Explicitly define the inputs to the model.
- Characterize all links that will enable the software engineer to better understand the vision.

To construct a system model, the software engineer should consider a number of limiting factors:-

- Assumptions – to reduce the number of possible permutations and variations.
- Simplifications – to enable the model to be created in a timely manner.
- Constraints – to guide the manner in which the model is created and the approach taken when the model is implemented.
- Limitations - to helps to bind the system.
- Preferences – to indicate the preferred architecture for all data, functions and technology.

The resultant system model may call for a completely automated solution, partly automated solution or non automated solution.

2.9 ANALYTICAL FRAMEWORKS

Analytical frameworks combine reusable solutions with analysis patterns, research, useful organization techniques, and specific examples of successful approaches. An analytical framework is a little like your own personal library, tailored specifically to your own experience and background. Analytical frameworks incorporate patterns and also provide a checklist of skills, tools, and techniques that are necessary for researching a particular area, such as business analysis or system architecture. An analytical framework is useful only if the tools are familiar to the modeler.

Some of the following analysis models cover the analytical framework of different approaches:-

- Structured analysis modeling
- Object oriented modeling
- Data modeling

2.9.1 Structured analysis modeling

Structured Analysis consists of interpreting the system concept (or real world) into data and control terminology i.e. into data flow diagrams. The flow of data and control from bubble to data store and to bubble can be very hard to track and the number of bubbles can get to be extremely large. One approach is to first define events from the outside world that require the system to react, then assign a bubble to that event, bubbles that need to interact are then connected until the system is defined.

Data Flow Diagrams (DFDs) are a graphical representation of the flow of data through a system. They are directed graphs showing processing elements and data stores with the dataflow between them. In a DFD, the nodes are external entitites, processes or data stores, and the edges are data flows.

A process can be further decomposed to a more detailed DFD which shows the sub-processes and data flows within it. The sub-processes can in turn be decomposed further with another set of DFDs until their functions can be easily understood. Processes which do not need to be decomposed further are called functional primitives and are described by a process specification (or mini-spec). The process specification can consist of pseudo-code, flowcharts, or structured English. The DFDs thus model the structure of the system as a network of interconnected processes composed of functional primitives.

A DFD is augmented by a data dictionary that dexcribes all of the data flows, data elements, files and data bases.

DFDs can be used to represent a system in terms of the input data to the system, various processing carried out on these data and the output data generated by the system. The symbols used in DFDs are shown in figure 2.4.

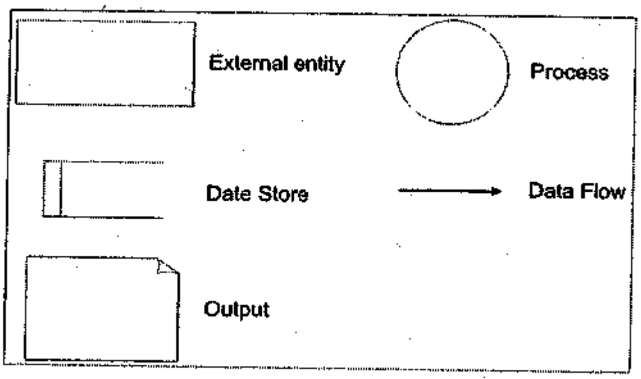


Figure 2.4 : Symbols used in DFDs

Context Diagram

Context diagrams are diagrams that represent all external entities that may interact with a system.

This diagram is the highest level view of a system, similar to Block Diagram, showing a, possibly software-based, system as a whole and its inputs and outputs from/to external factors. It represents the entire system as a single bubble. This bubble is labeled according to the main function of the system. It is also known as Level-0 DFD. Figure 2.5 shows the context diagram of an Accounts Payable system

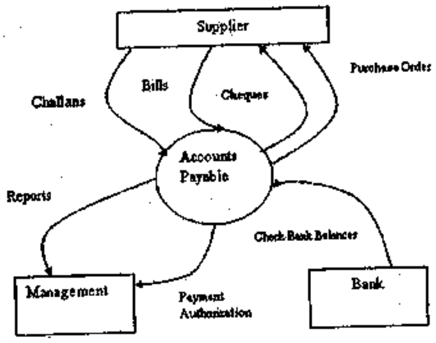


Fig. 2.5 : Context Diagram of an Accounts Payable System

A Context diagram does not show any detail but is an overview of the system. We can elaborate the context diagram into further level-1, level-2 etc. diagrams. The level-1 DFD can be drawn made as in figure 2.6.

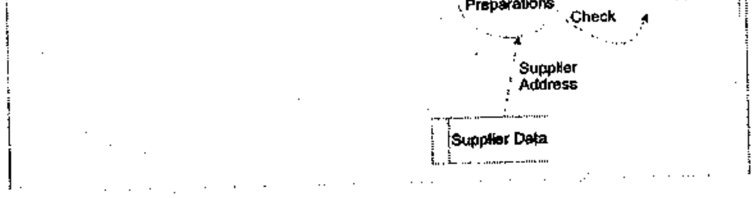


Fig. 2.6 : Level 1 DFD for Accounts Payable System

Common Errors while constructing a DFD model

The following are the most common errors made while constructing a DFD for a system:-

- Many designers/developers commit the mistake of drawing more than one bubble in the context diagram.
- Many designers/developers have external entities at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear at other levels of the DFD.
- It is a common lapse to have either too less or too many bubbles in a DFD. Only 3 to 7 bubbles per diagram should be allowed.
- Another common error is to present control information in DFD. DFD is a data flow

representation of a system and it does not represent control information.

2.9.2 Object oriented modeling

Object modeling was recently introduced into system development. It emphasises on:-

- Combining processes, data and flows into the one modeling paradigm, thus allowing objects to be modeled as independent entities that can be flexibly combined into cooperating systems.
- Easy conversion from analysis to design models, through the use of similar terms.
- Support of multimedia information and not only record structures.

In object analysis it is not necessary to consider in terms of building one large system. We identify objects as independent entities with their own local goals. Such objects can exchange messages between them to achieve a global goal of the large system.

The objective of object oriented analysis is to define all classes and the relationships and behavior associated with them are relevant to the problem. The following concepts must be known to the developer for modeling:-

- Abstraction is about looking only at the information that is relevant at the time.
- When modeling, 2 or 3 orthogonal views of the system are created.
- Diagrams are independent views of the model and may be of different types.
- In object oriented analysis, the object model, or data view, is the primary and most stable view of the system.
- Hardware engineering produces well encapsulated hardware designs.
- Applying encapsulation techniques to software encourages maintainable, re-useable and extensible code.

Object Oriented Analysis builds a class-oriented model that relies on an understanding of OO Concepts.

2.9.3 Data modeling

Data modeling in software engineering is the process of creating a data model by applying formal data model descriptions using data modeling techniques. Data modeling is a method used to define and analyze data requirements needed to support the business processes of an organization. The data requirements are recorded as a conceptual data model with associated data definitions. Actual implementation of the conceptual model is called a logical data model. To implement one conceptual data model may require multiple logical data models. Data modeling defines the relationships between data elements and structures.

Data modeling techniques are used to model data in a standard, consistent, predictable manner in order to manage it as a resource. The use of this standard is strongly recommended for all projects requiring a standard means of defining and analyzing the data resources within an organization.

Data modeling may be performed during various types of projects and in multiple phases of projects. Data models are progressive; there is no such thing as the final data model for a business or application. Instead a data model should be considered a living document that will change in response to a changing business. The data models should ideally be stored in a repository so that they can be retrieved, expanded, and edited over time. Data models represent information areas of interest.

There are several notations for data modeling. The actual model is frequently called "Entity relationship model", because it depicts data in terms of the entities and relationships described in the data. An entity-relationship model (ERM) is an abstract conceptual representation of structured data. *Entity-relationship modeling is a relational schema database modeling method, used in software engineering to produce a type of conceptual data* model (or semantic data model) of a system, often a relational database, and its requirements in a top-down fashion. These models are being used in the first stage of information system design during the requirements analysis to describe information needs or the type of information that is to be stored in a database. Entity relationship diagram is essential for the design of database tables, extracts, and metadata.

2.10 INTRODUCTION TO CASE TOOLS

CASE is the use of computer-based support in the software development process. This includes all kinds of computer-based support for any of the managerial, administrative, or technical aspects of any part of a software project.

Computer-Aided Software Engineering (CASE), in the field of Software Engineering is the scientific application of a set of tools and methods to software development which results in high-quality, defect-free, and maintainable software products. It also refers to methods for the development of information systems together with automated tools that can be used in the software development process.

A CASE tool is a computer-based product aimed at supporting one or more software engineering activities within a software development process.

CASE tools are a class of software that automates many of the activities involved in various life cycle phases. For example, when establishing the functional requirements of a proposed application, prototyping tools can be used to develop graphic models of application forms/screens to assist users to visualize how an application will look after completion. Subsequently, system designers can use automated design tools to transform the prototyped functional requirements into detailed design documents. Developers can then use automated code generators to convert the design documents into code. Automated tools can be used collectively, or individually. For example, prototyping tools could be used to define application requirements that get passed to design technicians who convert the requirements into detailed designs in a traditional manner using flowcharts and narrative documents, without the assistance of automated design software.

The major benefits of using CASE tools includes:-

- Improved productivity
- Better documentation
- Improved accuracy
- Intangible benefits
- Improved quality
- Reduced lifetime maintenance
- Opportunity to non-programmers
- Reduced cost of software
- Produce high quality and consistent documents
- Impact on the style of a working of company

2.11 SUMMARY

Requirement analysis is a software engineering task that bridges the gap between system-level software allocation and software design. Requirement analysis allows the software engineer to refine the software allocation and build models of the data, functional and behavioral domains that will be treated by software. Requirements are classified into Functional Requirements and Non Functional Requirements.

Requirement engineering takes the following processes-

- Requirement elicitation (gathering)
- Requirment analysis and modeling
- Requirment documentation
- Requirements review
- Requirements management

SRS stands for Software Requirement Specification. It establishes the basis for agreement between users and developers or suppliers on what the software product is expected to do, as well as what it is not expected to do. The SRS is produced at the culmination of the analysis task.

Analytical frameworks combine reusable solutions with analysis patterns, research, useful organization techniques, and specific examples of successful approaches. An analytical framework is useful only if the tools are familiar to the modeler.

Computer-Aided Software Engineering (CASE), in the field of Software Engineering is the scientific application of a set of tools and methods to software which results in high-quality, defect-free, and maintainable software products. CASE tools are a class of software that automates many of the activities involved in various life cycle phases.

2.12 UNIT-END QUESTIONS

1. What do you understand by requirements elicitation?
2. Discuss the significance and use of requirement engineering?
3. What is SRS? What are the components of SRS?
4. Discuss the characteristics of good SRS.
5. What do you mean by a DFD? Explain some of the symbols used to draw a DFD.
6. What is data modeling? Explain.
7. What do you mean by requirement management?
8. Describe the idea behind CASE tools.

2.13 FURTHER READINGS

1. Software Engineering - A Practitioner's Approach, Roger S. Pressman, McGrawhill
2. Fundamentals of Software Engineering, Rajib Mall, Prentice Hall of India

UNIT - III

SOFTWARE SPECIFICATIONS

Structure of the Unit

3.0 OBJECTIVES

On reading this chapter the students will be able to understand

- what are critical systems specifications
- how dependability requirements may be identified by analyzing risks faced by critical systems
- how safety requirements are generated from system risk analysis
- the derivation of security requirements
- describe metrics used for reliability specification
- understand issues in user interface specification

3.1 INTRODUCTION

The ability to produce correct computer programs that meet the needs of the user has been a long standing desire on the part of computer professionals. A correct program is one that does what it should do. A program is correct if it meets its specifications. Developing a software system without a specification is a random process. The implementation is doomed to be modified, sometimes forever, because it never precisely matches the client's needs. The goal of a specification is to capture the client's requirements in a concise, clear, unambiguous manner to minimize the risks of failure in the development process. It is much cheaper to change a specification than to change an implementation.

Additionally, the specification must leave as much freedom as possible to the implementer, in order to find the best implementation in terms of development cost, efficiency, usability and maintainability.

In this chapter the critical systems specifications are discussed in detail.

3.2 CRITICAL SYSTEMS SPECIFICATION

Software specifications are of prime importance for critical systems. Because of the high potential costs of system failure it is important to ensure that the specification for critical systems accurately reflects the real needs of users of the system. Only when the specifications are right, the system can be dependable.

Critical systems specification supplements the normal requirements specification process by focusing on the dependability requirements of the system. Its goal is to recognize the risks faced by the system and generate dependability requirements to cope with them.

The user requirements for critical systems are usually specified using natural language and system models.

3.3 RISK-DRIVEN SPECIFICATION

The risk-driven specification process involves understanding the risks faced by the system, discovering their root causes and generating requirements to manage these risks.

The steps in the process are illustrated in figure 3.1 and include,

i) Risk Identification: Potential risks that may arise and are dependent on the environment in which the system is to be used are identified.

   Software related risks are normally concerned with failure to deliver a system service or with the failure of monitoring and protection systems. Risk identification should be done in consultation with experienced engineers, domain experts and professional safety advisors.

ii) Risk Analysis and Classification: Risks that are potentially serious and not improbable are selected for further analysis. Some of the risks may be eliminated at this stage if they seem very unlikely ever to arise.

   Risks can be categorized into three levels, Intolerable, As low as reasonably practical (this risk is tolerated only if risk reduction is impractical or grossly expensive), and Acceptable.

iii) Risk Decomposition: It is the process of discovering the root causes of risks in a particular system. Each risk is analyzed individually to discover potential root causes of that risk.

iv) Risk Reduction Assessment: Proposals for ways in which the identified risks may be reduced or eliminated are made. These then generate system dependability requirements that define the defenses against the risk and how these risks could be managed if it arises. The strategies that can be used are Risk Avoidance, Risk Detection and Removal, and Damage Limitation.

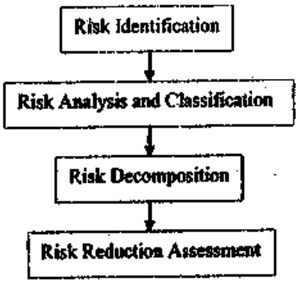


Figure 3.1: The Risk-Driven Specification Process

3.4 SAFETY SPECIFICATION

A safety specification stipulates that nothing bad ever happens. Safety requirements usually apply to the system as a whole rather than to individual sub-systems. In systems engineering terms, the safety of a system is an emergent property.

Safety requirements can be categorized as:

i) Functional safety requirements

These define the safety functions of the protection system i.e. they define how the system should provide protection.

ii) Safety integrity requirements

These define the reliability and availability of the protection system.

The process of safety specification and assurance undergoes various steps, and is part of an overall safety life cycle that is defined in an international standard for safety management, IEC61508. These steps include, defining and identifying the scope of the risk, assessing the probable system hazards and estimate the risk they pose. This is followed by safety requirements specifications and the allocation of these safety requirements to different subsystems. The development activity involves planning and implementation. The safety critical system itself is designed and implemented, as are related external systems that may provide additional protection. In parallel with this, the safety validation, installation, and operation and maintenance of the system are planned. The safety life cycle has been illustrated in figure 3.1.

Safety considerations are taken into account at all phases of the system including delivery, maintenance and decommisioning.

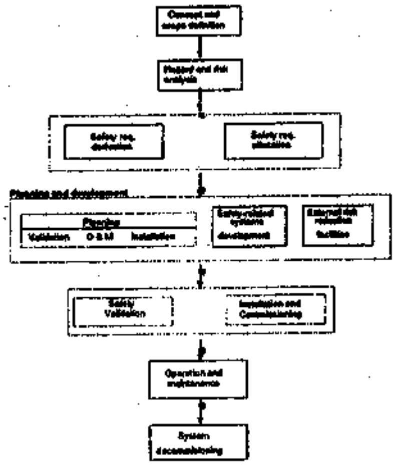


Figure 3.1: The Safety Life-cycle

3.5 SECURITY SPECIFICATION

Security is important in all aspects of life and the increasing pervasiveness and capability of software applications makes software security increasingly so. The problem with most software today is that it contains numerous flaws and errors that are often located and exploited by attackers to compromise the software's security and other required properties.

Secure software is software that is able to resist most attacks, tolerate the majority of attacks it cannot resist and recover quickly with a minimum of damage from the very few attacks it cannot tolerate. Secure software cannot be intentionally subverted or forced to fail. It remains dependable in spite of intentional efforts to compromise that dependability. Software security matters because so many critical functions have come to be completely dependent on software. This makes software a very high-value target for attackers, whose motives may be malicious, criminal, adversarial, or terrorist.

Applications designed with security in mind are safer than those where security is an afterthought.

The stages in the security specification process are:

1. Asset Identification and Evaluation: The assets, data and programs, and their required degree of protection are identified. The evaluation is based on the criticality or the protection required for the assets.

2. Threat Analysis and Risk Assessment: Possible security threats are identified and the risks associated with each ofthese threats are estimated.

3. Threat Assignment: Identified threats are related to the assets so that, for each identified asset, there is a list of associated threats.

4. Technology Analysis: Available security technologies and their applicability against the identified threats are assessed.

5. Security Requirements Specification: It identifies the security technologies that may be used to protect against threats to the system.

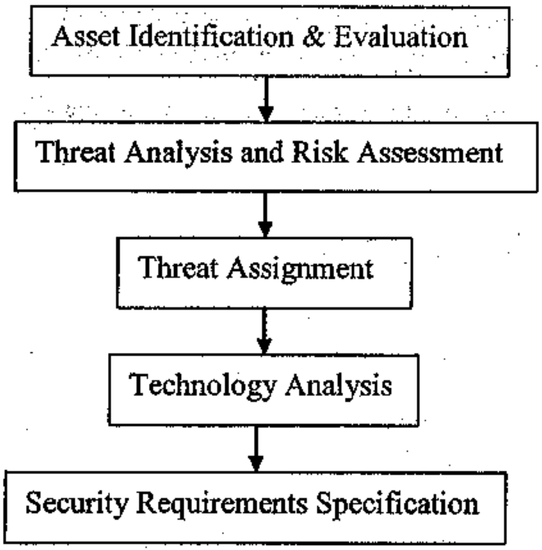

Asset Identification & Evaluation
Threat Analysis and Risk Assessment
Threat Assignment
Technology Analysis
Security Requirements Specification

Figure 3.2: The Security Specification Process

3.6 SOFTWARE RELIABILITY SPECIFICATION

Reliability is the probability that an item will perform a required function without failure under stated conditions for a stated period of time.

In a computer based system, there are three dimensions to be considered when specifying the overall system reliability. These include,

i) Hardware Reliability: It is the probability that a hardware component fails

ii) Software Reliability: It is the ability of a computer program to perform its intended functions and operations in a system's environment, without experiencing failure. It is the probability of failure-free software operation for a specified period of time in a specified environment. It includes the probability that a software

- component will produce an incorrect output
- a software does not wear out
- can continue to operate after a bad result.

iii) Operator Reliability: It is the probability that the operator of a system will make an error.

Reliability is a dynamic system attribute. System reliability should be specified as a non-functional requirement. Reliability metrics are units of measure for system reliability. System reliability is measured by counting the number of operational failures and relating these to demands made on the system at the time of failure. A long-term measurement program is required to assess the reliability of critical systems

3.6.1 Types of System Failure

It is software failures, not software faults, that affect the reliability of a system. The types of failures that can occur are system specific and the consequences of system failure depend on the nature of that failure. Table 3.1 illustrates the classification of system failures.

Type of System Failure	Description
Transient	Only occurs with certain inputs
Permanent	Occurs on all inputs
Recoverable	System can recover without operator help
Unrecoverable	Operator has to help
Non-corrupting	Failure does not corrupt system state or data
Corrupting	System state or data are altered

Table 3.1: Classification of System Failures

Combinations of these types of failures, such as a transient, recoverable and corrupting failure, may occur.

For large systems, composed of several sub-systems, the reliability requirements of the sub-systems may vary. For such systems, the reliability requirements therefore, may be specified for the sub-systems individually rather than for the whole system. This will avoid imposing the same reliability requirements on all sub-systems, thereby reducing time and cost. For smaller systems however, the reliability requirements should be specified on system basis.

3.6.2 Steps in Building Reliability Specification

The steps involved in establishing a reliability specification are,

- For each sub-system identify the types of system failure that may occur and analyze consequences of possible system failures

- From system failure analysis, partition failure into appropriate classes
- For each failure class identified, define the reliability requirements using an appropriate reliability metric.

3.7 USER INTERFACE SPECIFICATION

For the user, the user interface is the system. What users want is for developers to build applications that meet their needs and that are easy to use.

User interface design is important for several reasons including,

i) the more intuitive the user interface, the easier it is to use,

ii) the better the user interface the easier it is to train people to use it, reducing the training costs.

iii) the better the user interface the less help people will need to use it, reducing the support costs.

iv) the better the user interface the more users will like to use it, increasing their satisfaction with the system.

A User Interface (UI) Specification is a document that captures the details of the software user interface into a written document. The specification covers all possible actions that an end user may perform and all visual, auditory and other interaction elements. It is the main source of implementation information for how the software should work. A **UI specification thus** defines the rules of engagement for a user interacting with a specific page on a website or screen within an application.

Before UI specification is created, a lot of work is done already for defining the application and desired functionality. Usually there are requirements for the software which are basis for the use case creation and use case prioritizing.

3.7.1 The User Interface Specification Process

The steps in the UI specification process are,

i) Use case definition: The purpose of writing use cases is to enhance the UI designer's understanding of the features that the product must have and of the actions that take place when the user interacts with the product.

ii) Design draft creation: The UI design draft is done on the basis of the use case analysis. The purpose of the UI design draft is to show the design proposing, and to explain how the user interface enables the user to complete the main use cases, without going into details.

It should be as visual as possible and all the material created must be in such a format that it can be used in the final UI specification, is good time to conduct usability testing or expert evaluations and make changes.

iii) Writing the user interface specification: The Specification should contain all needed user interface details. Good UI specifications take into account the data and context of the user within the application. The UI specification can be seen as an extension on the design draft, i.e. it is a complete description that contains all details, exceptions, error cases, notifications, etc. The document is reviewed by the stakeholders so that all necessary details are in place.

3.8 SUMMARY

- Critical systems specification supplements the normal requirements specification process by focusing on the dependability requirements of the system.

- The goal of critical systems specification is to recognize the risks faced by the system and generate dependability requirements to cope with them.
- The aim of the specification process should be to understand the risks (safety, security, etc.) faced by the system and to define requirements that reduce these risks.
- The stages of risk-based specification are: Risk Identification, Risk Analysis and Classification, Risk Decomposition, Risk Reduction Assessment.
- Safety requirements should be based on an analysis of the possible hazards and risks.
- The Security Requirements Specification identifies the security technologies that may be used to protect against threats to the system.
- Software Reliability is the ability of a computer program to perform its intended functions and operations in a system's environment, without experiencing failure.
- Software Reliability Specification considers, How likely is it that a software component will produce an incorrect output. Software failures are different from hardware failures in that software does not wear out. It can continue in operation even after an incorrect result has been produced.
- A User Interface (UI) Specification is a document that captures the details of the software user interface into a written document.

3.9 UNIT END QUESTIONS

1. Describe the steps in Risk-Driven Specification.
2. What are the characteristics of a Secure Software?
3. Explain the stages in the security specification process.
4. What re the types of failure that affect the reliability of a system?
5. Write a Reliability Specification for Bank Auto-Teller System.
6. What is the need for user interface specification?
7. Write the steps in the User Interface specification process

3.10 FURTHER READINGS

1. Software Engineering, Ian Somerville, Addison Wesley, 6th edition, 2000
2. Software Engineering: A Practitioner's Approach, Roger S. Pressman, McGraw-Hill.

UNIT - IV

SOFTWARE PROCESS MODELS

Structure of the Unit

4.0 OBJECTIVES:

After going through this unit students will be able to,

- Understand the Software Process
- Comprehend the process models
- Appreciate the need for different models.

4.1 INTRODUCTION

A software process is a set of activities and associated results that produce a software product. A software Process provides a framework for managing activities that can easily get out of control. By improving the software development process, the quality of the resulting products can be improved. The strategy is to improve the management of the software process with the assumptionthat improvements in techniques will occur consequently. The resulting improvement in the process as a whole should result in better quality software.

4.2 SOFTWARE PROCESS ACTIVITIES

The production of software involves a number of different activities during which we try to ensure that the required functions and the required level of attributes are delivered.

Production activities can be grouped into four broad categories:

Specification: This is a description of what the software has to do and sets acceptable levels for software attributes. For most software systems going from the user needs to a statement of requirements and then to a precise specification is a difficult and error prone task. The study of Requirements Engineering is an increasingly important part of Software Engineering.

Design: This covers the high-level structural description of the architecture of the system, through the detailed design of the individual components in the system and finally to the implementation of the system on a particular computing platform (usually hardware plus operating software).

Validation and Verification: Validation is the activity of checking the correct system is being constructed (building the right system). Verification is the activity of checking the system is being constructed correctly (building the system right). These activities are essential to ensure that software project is going in the right direction.

Maintenance: This activity ensures that the system keeps track with changes in its operating environment throughout its operational life. This involves correcting errors that are discovered in operation and modifying the system to take account of changes in the system requirements. Repairing Millennium Bug errors in software is an example of Maintenance activity. In one sense we can see the Millennium Bug problem as a change of requirement because when these systems were written their intended lifetime was much shorter than has turned out to be the case.

4.3 PROCESS ITERATION

System Requirements neually evolve int he course of a project os process iteraction where earlier stages are reworked is always part of the process for large systems. Iternations may be applied to any of the generic process models.

4.4 SOFTWARE DEVELOPMENT TECHNIQUES

Some projects have clear objectives and goals while other's end results are uncertain. A'one size fils are project development method cannot therefore be applied. Many factors may affect the chosen project development method of an organization or project team. When evaluating the projct to be taken, it needs to be considered whether the projects is familiar domain with a predictable paht or a new one with uncertain outcomes. The planning therefore could be predictive or adaptive.

4.4.1 Predictive Software Development

Project from a familiar herritory usuall rely on a predictive method of planning. Predictive planning provides a linear, spedific development paln structured around producing a pre-determined end result within a specific time frame. Example of this approach is the 'waterfall' method discussed later.

4.4.2 Adaptive Software Development

Evolving projects that face hanging conditions are best suited for adaptive planning. This approach involves breating a project into small compnents over an undetermined time frame to allow ultimate flexibility n directing the course of the project. Adaptive teams may choose 'agile' techniques.

Agile Software Development

Agile software development is a conceptual framework for undertaking software engineering projects. Agile methods attempt to minimize risk and maximize productivity by developing software in short iterations and de-emphasizing work on secondary or interim work artifacts. Scrum and Extreme Programming (XP) are two of the most popular Agile methods.The following sections discuss the Software Process models. Extreme Programming (XP) is a lighweight, efficient, low-risk, flexible, predictable and scientific way to software development. It follows the incremental planning approach that comes up quickly with an overall plan that is expected to evolve through the life of the project.

4.5 WATERFALL MODEL

Waterfall approach was first Process Model to be introduced and followed widely in Software Engineering to ensure success of the project. In the "Waterfall" approach, the whole process of software development is divided into separate process phases. The phases in Waterfall model are: Requirement Specifications phase, Software Design, Implementation and Testing & Maintenance. All these phases are cascaded so that second phase is started as and when defined set of goals are achieved for first phase and it is signed off, so the name "Waterfall Model".

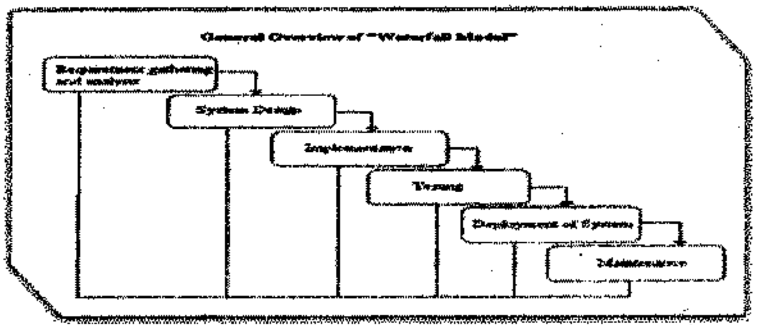

General Overview of "Waterfall Model"
Requirement gathering and analysis
System Design
Implementation
Testing
Deployment of System
Maintenance

Figure 4.1 : The Waterfall Model

4.5.1 The stages of "The Waterfall Model" are:

Requirement Analysis & Definition: All possible requirements of the system to be developed are captured in this phase. Requirements are set of functionalities and constraints that the end-user expects from the system. The requirements are gathered from the end-user by consultation, these requirements are analyzed for their validity and the possibility of incorporating the requirements in the system to be developed is also studied. Finally, a Requirement Specification document is created which acts as a guideline for the next phase of the model.

System & Software Design: Before going in for actual coding, it is very important to understand what is to be created and what it should look like. The requirement specifications from first phase are studied in this phase and system design is prepared. System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture. The system design specifications serve as input for the next phase of the model.

Implementation: On receiving system design documents, the work is divided in modules/units and actual coding is started. The system is first developed in small programs called units, which are then integrated. Each unit is developed and tested for its functionality; this is referred to as Unit Testing.

Testing: This stage uncovers errors introduced during coding and also the errors introduced during the previous phases. This stage thus, uncovers requirement, design, and coding erros int he programs.

Testing is done at different stages. The starting point of testing is **unit testing**: Unit testing verifies if the modules / units meet their specifications.

As modules are integrated into the sytem, **integration testing** is performed integration testing focuses on testing the interconnection between modules. After the system is put together, **system testing** is performed. System testing is done to test the system against system requirements, to check if it works as per the requirements specification.

Deployment of system: Once the system is tested, it is installed at the clients sets.

Maintenance: This phase of the Waterfall Model is virtually a never ending phase. Generally, problems with the system developed come up after its practical use starts, so the issues related to the system are solved after deployment of the system. Not all the problems come in picture directly but they arise from time to time and needs to be solved; hence this process is referred as Maintenance.

4.5.2 Advantages and Disadvantages of Waterfall Model

Advantages: The advantage of waterfall development is that it allows for departmentalization and managerial control. A schedule can be set with deadlines for each stage of development and a product can proceed through the development process and theoretically, be delivered on time. Development moves from concept, through design, implementation, testing, installation, troubleshooting, and ends up at deployment and maintenance. Each phase of development proceeds in strict order, without any overlapping or iterative steps.

Disadvantages: The disadvantage of waterfall development is that it does not allow for much reflection or revision. Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.

4.6 INCREMENTAL MODEL

The incremental model is an intuitive approach to the waterfall model. Multiple development cycles take place here, making the life cycle a "multi-waterfall" cycle. Cycles are divided into smaller, more easily managed iterations. Each iteration passes through the requirements, design, implementation and testing phases.

A working version of software is produced during the first iteration, so that there is a working software early on during the software life cycle. Subsequent iterations build on the initial software produced during the first iteration.

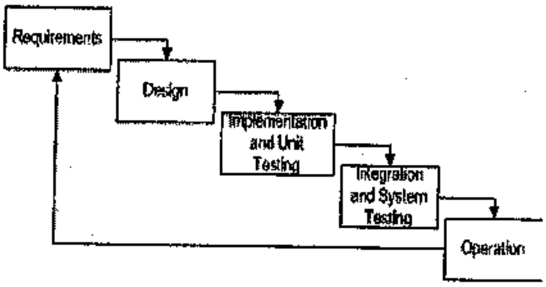

Requirements
Design
Implementation and Unit Testing
Integration and System Testing
Operation

Figure 4.2 : Incremental Life Cycle Model

4.6.1 Advantages & Disadvantages of Incremental Model

Advantages

- Generates working software quickly and early during the software life cycle.
- More flexible – less costly to change scope and requirements.
- Easier to test and debug during a smaller iteration.
- Easier to manage risk because risky pieces are identified and handled during its iteration.
- Each iteration is an easily managed milestone.

Disadvantages

- Each phase of an iteration is rigid and do not overlap each other.
- Problems may arise pertaining to system architecture because not all requirements are gathered up front for the entire software life cycle.

4.7 PROTOTYPING

Software prototyping is the creation of prototypes, i.e., incomplete versions of the software program being developed.

A prototype implements only a small subset of the features of the eventual program, and the implementation may be completely different from that of the eventual product.

The purpose of a prototype is to allow users of the software to evaluate proposals for the design of the eventual product by actually trying them out, rather than having to interpret and evaluate the design based on descriptions.

4.7.1 Prototyping Process

The process of prototyping involves the following steps

- Identify basic requirements
- Determine detailed requirements including the input and output information desired. Details, sueh as security, can however be ignored at this stage.

- Develop initial Prototype
- Develop the initial prototype that includes only user interfaces.
- Review
- The customers, including end-users, examine the prototype and provide feedback on additions or changes.
- Revise and enhance the Prototype

Using the feedback both the specifications and the prototype can be improved. Negotiation about what is within the scope of the product may be required.

4.7.2 Advantages & Disadvantages of Prototyping

There are many advantages to using prototyping in software development these include,

Reduced time and costs: Prototyping can improve the quality of requirements and specifications provided to developers. Because changes cost exponentially more to implement as they are detected later in development, the early determination of what the user really wants can result in faster and less expensive software.

Improved and increased user involvement: Prototyping requires user involvement and allows the user to see and interact with a prototype allowing them to provide better and more complete feedback and specifications. The presence of the prototype being examined by the user prevents many misunderstandings and miscommunications that occur when each side believe the other understands what they said. Since users know the problem domain better than anyone on the development team does, increased interaction can result in final product that has greater tangible and intangible quality.

Disadvantages

Prototyping can also have disadvantages. These are,

Insufficient analysis: The focus on a limited prototype can distract developers from properly analyzing the complete project. This can lead to overlooking better solutions, preparation of incomplete specifications or the conversion of limited prototypes into poorly engineered final projects that are hard to maintain. Further, since a prototype is limited in functionality it may not scale well if the prototype is used as the basis of a final deliverable, which may not be noticed if developers are too focused on building a prototype as a model.

User confusion of prototype and finished system: Users can begin to think that a prototype, intended to be thrown away, is actually a final system that merely needs to be finished or polished. This can lead them to expect the prototype to accurately model the performance of the final system when this is not the intent of the developers. Users can also become attached to features that were included in a prototype for consideration and then removed from the specification for a final system.

Developer attachment to prototype: Developers can also become attached to prototypes they have spent a great deal of effort producing; this can lead to problems like attempting to convert a limited prototype into a final system when it does not have an appropriate underlying architecture.

Excessive development time of the prototype: A key property of prototyping is the fact that it is supposed to be done quickly. If the developers lose sight of this fact, they may try to develop a prototype that is too complex. When the prototype is thrown away the precisely developed requirements that it provides may not yield a sufficient increase in productivity to make up for the time spent developing the prototype. Users can become stuck in debates over details of the prototype, holding up the development team and delaying the final product.

Expense of implementing prototyping: The start up costs for building a development team focused on prototyping may be high. Many companies have development methodologies in place, and changing them can mean retraining, retooling, or both. Many companies tend to just jump into the prototyping without bothering to retrain their workers as much as they should.

A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort behind the learning curve. In addition to training for the use of a prototyping technique, there is an often overlooked need for developing corporate and project specific underlying structure to support the technology. When this underlying structure is omitted, lower productivity can often result.

4.8 CONCURRENT DEVELOPMENT

In the concurrent model of Software Development, multiply features or functions are concurrently developed along with the entire development life cycle. The model difires a series of events that will trigger transitions from state to state for each of the software engineering activities, actions or tasks. It is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities, actions and tasks to a sequence of envets, it defines a network of activities. Each activity, action or task on the network exists simultaneously with other activities, actions or tasks. Events generated at one point int he process network trigger transitions among the states.

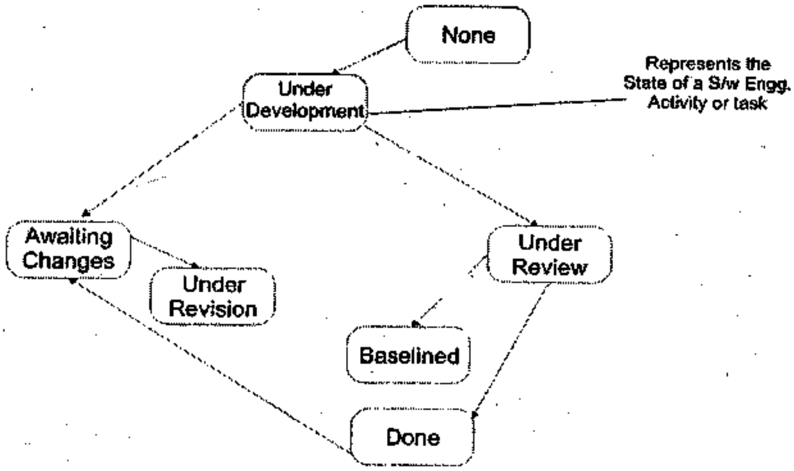


Figure 4.3: An element of the Concurrent Model

4.9 RAPID APPLICATION DEVELOPMENT (RAD)

RAD is a linear sequential software development process model that emphasis is an extremely short development cycle using a component based construction approach. If the requirements are well understood and defined, and the project scope is a constraint, the RAD process enables a development team to create a fully functional system within very short time period.

The RAD concept implies that products can be developed faster and can be of higher quality through:

- Gathering requirements using workshops or focus groups
- Prototyping and early, reiterative user testing of designs
- The re-use of software components
- A rigidly paced schedule that defers design improvements to the next product version

- Less formality in reviews and other team communication

Some companies offer products that provide some or all of the tools for RAD software development. These products include requirements gathering tools, Prototyping tools, Computer-Aided Software Engineering (CASE) tools, language development environments such as those for the Java platform, groupware for communication among development members, and testing tools. RAD usually embraces object-oriented programming methodology, which inherently fosters software re-use. The most popular object-oriented programming languages, C++ and Java, are offered in visual programming packages often described as providing rapid application development environment.

4.9.1 Development Methodology

The traditional software development cycle follows a rigid sequence of steps with a formal sign-off at the completion of each. A complete, detailed requirements analysis is done that attempts to capture the system requirements in a Requirements Specification. Users are forced to "sign-off" on the specification before development proceeds to the next step. This is followed by a complete system design and then development and testing.

But, what if the design phase reveals requirements that are technically unachievable, or extremely expensive to implement? What if errors in the design are found during the build phase? The elapsed time between the initial analysis and testing is usually a period of several months. What if business requirements or priorities change or the users realize they unnoticed critical needs during the analysis phase? These are many of the reasons why software development projects either fail or don't meet the user's expectations when delivered.

RAD is a methodology for compressing the analysis, design, build, and test phases into a series of short, iterative development cycles. This has a number of distinct advantages over the traditional sequential development model.

4.9.2 Phases

RAD model has the following phases:

- Business Modeling: The information flow among business functions is defined by answering questions like what information drives the business process, what information is generated, who generates it, where does the information go, who processes it and so on.

- Data Modeling: The information collected from business modeling is refined into a set of data objects (entities) that are needed to support the business. The attributes (character of each entity) are identified and the relation between these data objects (entities) is defined.

- Process Modeling: The data object defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting or retrieving a data object.

- Application Generation: Automated tools are used to facilitate construction of the software.

- Testing and Turn over: Many of the programming components have already been tested since RAD emphasis reuse. This reduces overall testing time. But new components must be tested and all interfaces must be fully exercised.

4.9.3 Advantages and Disadvantages

RAD reduces the development time and reusability of components help to speed up development. All functions are modularized so it is easy to work with.

For large projects RAD requires highly skilled engineers in the team. Both end customer and developer should be committed to complete the system in a much abbreviated time frame. If commitment is lacking RAD will fail. RAD is based on Object Oriented approach and if it is difficult to modularize the project RAD model may not work well.

4.10 SPIRAL MODEL

The spiral model is similar to the incremental model, with more emphases placed on risk analysis. The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation. A software project repeatedly passes through these phases in iterations (called Spirals in this model). In the baseline spiral, starting in the planning phase, requirements are gathered and risk is assessed. Each subsequent spirals builds on the baseline spiral.

4.10.1 The Spiral Process

Requirements are gathered during the planning phase. In the risk analysis phase, a process is undertaken to identify risks and alternate solutions. A prototype is produced at the end of the risk analysis phase.

Software is produced in the engineering phase, along with testing at the end of the phase. The evaluation phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral.

In the spiral model, the angular component represents progress, and the radius of the spiral represents cost.

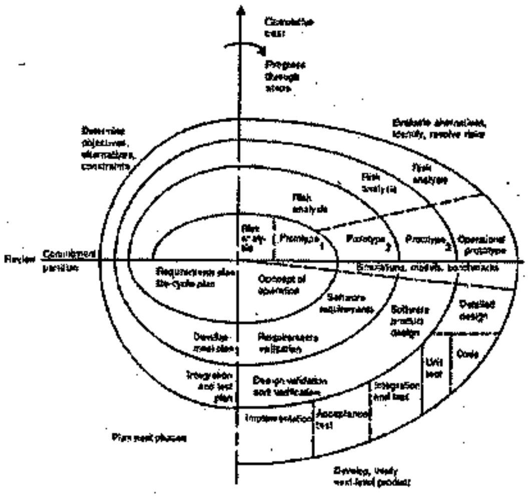


Figure 4.4 : A Typical Spiral Model

4.10.2 Advantages & Disadvantages

Advantages

- High amount of risk analysis
- Good for large and mission-critical projects.
- Software is produced early in the software life cycle.

Disadvantages

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

4.11 UNIFIED DEVELOPMENT PROCESS

The unified development process is an iterative and incremental software development process framework. It is an extensible framework which is customized for specific organizations or projects.

The Unified Process describes how to effectively deploy commercially proven approaches to software development for software development teams. These are called "best practices" not only because one can precisely quantify their value, but also, because they are observed to be commonly used in industry by successful organizations.

4.11.1 Rational Unified Process

The Rational Unified Process (RUP) a unified development process framework created by 'Rational Software Corporation', a division of IBM. The Rational Unified Process provides each team member with the guidelines, templates and tool mentors necessary for the entire team to take full advantage of among others the following best practices:

1. Develop software iteratively
2. Manage requirements
3. Use component-based architectures
4. Visually model software
5. Verify software quality
6. Control changes to software

Develop Software Iteratively - Given today's sophisticated software systems, it is not possible to sequentially first define the entire problem, design the entire solution, build the software and then test the product at the end. An iterative approach that allows an increasing understanding of the problem through successive refinements, and to incrementally grow an effective solution over multiple iterations is required.

The RUP supports an iterative approach to development that addresses the highest risk items at every stage in the lifecycle, significantly reducing a project's risk profile. This iterative approach helps in attacking risk through demonstrable progress frequent, executable releases that enable continuous end user involvement and feedback. Since each iteration ends with an executable release, the development team stays focused on producing results, and frequent status checks help ensure that the project stays on schedule. An iterative approach also makes it easier to accommodate tactical changes in requirements.

Manage Requirements - The Rational Unified Process describes how to elicit, organize, and document required functionality and constraints; track and document tradeoffs and decisions; and easily capture and communicate business requirements. The notions of use case and scenarios proscribed in the process has proven to be an excellent way to capture functional requirements and to ensure that these drive the design, implementation and testing of software, making it more likely that the final system fulfills the end user needs. They provide coherent and traceable threads through both the development and the delivered system.

Use Component-based Architectures - The process focuses on early development and baselining of a robust executable architecture, prior to committing resources for full-scale development. It describes how to design a resilient architecture that is flexible, accommodates change, is intuitively understandable, and promotes more effective software reuse. The Rational Unified Process supports component-based software development and provides a systematic approach to defining an architecture using new and existing components. The components are assembled in a well-defined architecture, such as the Internet, CORBA, and COM, for which an industry of reusable components is emerging.

Visually Modeling Software - The process shows how to visually model software to capture the structure and behavior of architectures and components. This allows hiding the details and writing code using "graphical building blocks." Visual abstractions help in communicating different aspects of the software, see how the elements of the system fit together, make sure that the building blocks are consistent with the code; maintain consistency between a design and its implementation; and promote unambiguous communication.

Verify Software Quality - Poor application performance and poor reliability are common factors which dramatically inhibit the acceptability of today's software applications. Hence, quality should be reviewed with respect to the requirements based on reliability, functionality, application performance and system performance. The Rational Unified Process assists in the planning, design, implementation, execution, and evaluation of these test types. Quality assessment is built into the process, in all activities, involving all participants, using objective measurements and criteria, and not treated as an afterthought or a separate activity performed by a separate group.

Control Changes to Software - The ability to manage change making certain that each change is acceptable, and being able to track changes is essential in an environment in which change is inevitable. The process describes how to control, track and monitor changes to enable successful iterative development. It also guides in how to establish secure workspaces for each developer by providing isolation from changes made in other workspaces and by controlling changes of all software artifacts. And it brings a team together to work as a single unit by describing how to automate integration and build management.

4.12 ASPECT ORIENTED SOFTWARE DEVELOPMENT

Aspect are properties that cross-cut several components in a subsystem i.e. they are concerus that cut across other concrsn. A concern is any area of interest in a software system.

Aspect-Oriented Software Development (AOSD) techniques provide systematic means for the identification, modularisation, representation and composition of crosscutting concerns such as security, mobility and real-time constraints. It addresses modularity problems that are not handed well by other approaches. It complements these approaches such as the structured programming and object oriented programming approaches, and doesn't replace them.

4.13 4GL TECHNIQUES

4GLs or Fourth-Generation Languages specify software at a high level and emplay automatic code generation. They automatically convert a format software specification into a program. The stages in 4GL model are shown in figure 4.5

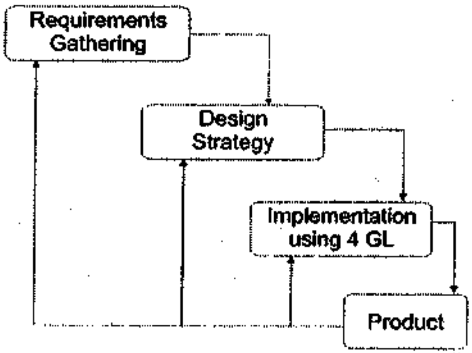


Figure 4.5: Stages in 4GL Model

Requirements are gathered from the customer and further studied and analysed so that these could be directly translated into an operational prototype. After having gathered the requiements, a strategy is laid out for the design of the software. This is necessary for large projects. The code is then generated based on some specification such as input and output forms.

4.14 SUMMARY

- A software process is a set of activities and associated results that produce a software product. A software rocess provides a framework for managing activities that can very easily get out of control.
- The production of software involves a number of different activities during which we try to ensure we deliver the required functions and the required level of attributes.
- Waterfall approach was first Process Model to be introduced and followed widely in Software Engineering to ensure success of the project. In "The Waterfall" approach, the whole process of software development is divided into separate process phases. The phases in Waterfall model are: Requirement Specifications phase, Software Design, Implementation and Testing & Maintenance.
- The incremental model is an intuitive approach to the waterfall model. Multiple development cycles take place here, making the life cycle a "multi-waterfall" cycle. Cycles are divided up into smaller, more easily managed iterations. Each iteration passes through the requirements, design, implementation and testing phases.
- Software prototyping, a possible activity during software development, is the creation of prototypes, i.e., incomplete versions of the software program being developed.
- A prototype typically implements only a small subset of the features of the eventual program, and the implementation may be completely different from that of the eventual product
- Concurrent Development is a software development technique in which multiple modules of the software are concurrently developed along with entire development life cycle.
- The Rapid Application Development (RAD) model is primarily based on the concept of software component reusability & follows the component based construction approach.
- The spiral model is similar to the incremental model, with more emphases placed on risk analysis. The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation.
- The unified Development Process describe how to effectively deploy commercially proven approaches to software development teams.
- Aspect-Oriented Software Development (AOSD) techniques provide systematic means for the identification, modularisation, representation and composition of crosscutting concerns such as security, mobility and real-time constraints.
- 4GL technique specifies software at a high level and also employs automatic code generation.

4.15 UNIT END QUESTIONS

1 What do mean by software development life cycle?

2 What is software process and activity?

3 Write a short note on Agile method.

4 What is RAD process model? When is it used?

5 What is water fall model, write advantages and disadvantages of water fall model.

6 What is prototyping model, why is it important?

7 Write a short note on aspect oriented development.

4.16 FURTHER READINGS

1. Software Engineering: A Practioner's Approach, Roger S.Pressman, McGrawHill

UNIT - V

SOFTWARE DESIGN

Structure of the Unit

5.0 OBJECTIVES

After completing this unit students will be able to

- Understand design concepts
- Be aware of software desgin strategies
- Apply different strategies

- Understant user interface desgin issues & processes
- Differentiate between desgin strategies

5.1 INTRODUCTION

A software design is a meaningful engineering representation of some software product that is to be built. A design can be traced to the customer's requirements and can be assessed for quality against predefined criteria. In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces and components.

5.2 RELATIONSHIP BETWEEN ANALYSIS AND DESIGN

Software Analysis refers to the process which Analysts go through to determine how a system should operate. It is a process for determining what functions the system should perform, whether it's feasible for the system to be developed, what data is going to be collected and stored. Systems Analysis is thus concerned with problem solving - creating a system that will solve an organizational problem.

Systems Design is the process where the analyst designs how the system will operate. The physical components of the system are defined here. It specifies how the problem at hand will be solved. It is at this stage that it is decided how the system will operate, in terms of the hardware, software and network infrastructure, e.g forms, user interface and reports that will be used.

Transition from Analysis to Design

A typical software development lifecycle starts with analysis to guide system design. The challenge however is in transitioning from findings about users, their activities, needs, the feasibility of the project at hand, the functions to be performed by the software etc. into design requirements, constraints and implications that are directly applicable to design. Processes such as contextual design and design objects such as scenarios, or tasks aid in transition.

5.3 DESIGN CONCEPTS

The design process is very important. From a practical point of view, a labourer, would not go for buildinga house without an approved blueprint thereby risking the structural integrity and customer satisfaction. Same is the case with building software products. The emphasis in design is on quality. This phase provides us with representation of software that can be assessed for quality. Also, this is the only phase in which the customer's requirements can be accurately translated into a finished software product or system. Software design thus serves as the foundation for all software engineering steps that follow regardless of which process model is being employed. Without a proper design we risk building an unstable system – one that will fail when small changes are made, one that may be difficult to test, one whose quality cannot be assessed until late in the software process, perhaps when critical deadlines are approaching and much capital has already been invested into the product.

During the design process the software specifications are transformed into design models that describe the details of the data structures, system architecture, interface, and components. Each design product is reviewed for quality before moving to the next phase of software development. At the end of the design process a design specification document is produced. This document is composed of the design models that describe the data, architecture, interfaces and components.

At the data and architectural levels the emphasis is placed on the patterns as they relate to the application to be built. Whereas at the interface level, human ergonomics often determine the design approach employed. Lastly, at the component level the design is concerned with a 'programming approach' which leads to effective data and procedural designs.

Design Specification Models

- **Data design** – created by transforming the analysis information model (data dictionary and ERD) into data structures required to implement the software. Part of the data design may occur in conjunction with the design of software architecture. More detailed data design occurs as each software component is designed.
- **Architectural design** - defines the relationships among the major structural elements of the software, the 'design patterns' than can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which the architectural patterns can be applied. It is derived from the system specification, the analysis model, and the subsystem interactions defined in the analysis model.
- **Interface design** - describes how the software elements communicate with each other, with other systems, and with human users, the data flow and control flow diagrams provide much of the necessary information required.
- **Component-level design** - created by transforming the structural elements defined by the software architecture into procedural descriptions of software components using information obtained from the process specification, control specification, and state transition diagram.

These models collectively form the design model, which is represented diagrammatically as a pyramid structure (Figure 5.1) with data design at the base and component level design at the pinnacle.

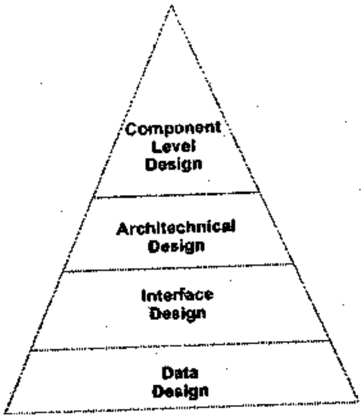


Figure 5.1 : The Design Model

Design Guidelines

In order to evaluate the quality of a design the criteria for a good design should be setup. Such a design should:

- exhibit good architectural structure
- be modular
- contain distinct representations of data, architecture, interfaces, and components
- lead to data structures that are appropriate for the objects to be implemented and be drawn from

recognizable design patterns

- lead to components that exhibit independent functional characteristics
- lead to interfaces that reduce the complexity of connections between modules and with the external environment
- be derived using a reputable method that is driven by information obtained during software requirements analysis

These criteria are not achieved by chance. The software design process encourages good design through the application of fundamental design principles, systematic methodology and through review.

Design Principles

Software design can be viewed as both a process and a model.

The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built.

The design model is equivalent to the architect's plans for a house. It begins by representing the totality of the entity to be built, and slowly refines the entity to provide guidance for constructing each detail. Similarly the design model that is created for software provides a variety of views of the computer software."

The set of principles which has been established to aid the software engineer in navigating the design process are:

1. The design process should not suffer from restricted vision – A good designer should consider alternative approaches. Judging each based on the requirements of the problem, the resources available to do the job and any other constraints.
2. The design should be traceable to the analysis model – because a single element of the design model often traces to multiple requirements, it is necessary to have a means of tracking how the requirements have been satisfied by the model.
3. The design should not create components that aready exist – Systems are constructed using a set of design patterns, many of which may have likely been encountered before. These patterns should always be reused.
4. The design should such that the software should be close to the problem as it exists in the real world - That is, the structure of the software design should mimic the structure of the problem domain.
5. The design should exhibit uniformity and integration – a design is uniform if it appears that one person developed the whole thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.
6. The design should be structured to degrade gently, even with bad data, events, or operating conditions are encountered. It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.
7. The design should be reviewed to minimize conceptual errors – there is sometimes the tendency to focus on minute details when the design is reviewed, missing the forest for the trees. The designer team should ensure that major conceptual elements of the design have been addressed before worrying about the syntax of the design model.
8. Design is not coding, coding is not design – Even when detailed designs are created for program components, the level of abstraction of the design model is higher than source code.
9. The design should be structured to accommodate change.
10. The design should be assessed for quality as it is being created.

When these design principles are properly applied, the design exhibits both external and internal quality factors. External quality factors are those factors that can readily be observed by the user. Internal quality factors relate to the technical quality more so the quality of the design itself. To achieve internal quality factors the designer must understand basic design concepts.

Fundamental Software Design Concepts

A set of fundamental software design concepts has evolved, each providing 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:

1. What criteria can be used to partition software into individual components?
2. How is function or data structure detail separated from a conceptual representation of software?
3. 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. Abstraction can be procedural abstraction - named sequence of events or data abstraction - named collection of data objects.

- **Refinement** - process of elaboration where the designer provides successively more detail for each design component.
- **Modularity** - the degree to which software can be understood by examining its components independently of one another. Modularity is discussed in detail in section 5.5
- **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.
- **Structural partitioning** - Horizontal partitioning defines three partitions namely input, data transformations, and output; vertical partitioning factoring distributes control in a top-down manner with 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** - information (data and procedure) contained within a module is inaccessible to modules that have no need for such information.

5.4 TOOLS AND DIAGRAMMING AIDS USED FOR DESIGN

The input from the analysis phase is used to derive program structures. The software design is framed using the following tools:

- Data Flow Diagram (DFD)
- Data Dictionary
- Decision Tree
- Decision Tables

- Structure Chart
- Structured English

Data Flow Diagram (DFD): A DFD is a graphicalrepresentation of the 'flow' of data through an information system. It shows what data will be input to and output from the system, where the data will come from and go to, and where the data will be stored.

Data Dictionary: Data Dictionary is a soft of metadata which contains the definition & representation of data elements. It can also be described as a life that defines tha basic organizaiton of a database. It contains information such as database design, stored procedure dtails, user permissions and statistics of processes, growth, performance and user.

Decision Tree: It is a graphical represention of decisions and their possible consequence.

Decision Table: It lists causes and effects in a matrix. Each coumn of the matrix represents a unique combination of causes & effects. The causes are the conditions to be checked and the effects are the actions or expected results.

Structure Chart: Structure charts are used to specify the high level design, or architecture, of a software. As a design tool, they aid the programmer in dividing & conquering a large software problems.

Structure English: Structured English is the use of English Language with the syntax of structurned programming. It aims at specifying the software design elements in simple English so that these can be understood easily by everyone.

5.5 MODULARITY

The concept of modularity in computer software has been advocated for about five decades. The software is divided into separately names and addressable components called modules that are integrated to satisfy problem requirements. A reader cannot easily grasp large programs comprising of a single module. The number of variables, control paths and complexity would make understanding virtually impossible. Consequently a modular approach will allow for the software to be intellectually manageable. One cannot however subdivide software indefinitely to make the effort required to understand or develop it negligible. This is because as the number of modules increase, the effort (cost) associated with integrating the modules increases. A good software design is believed to imply clean decomposition of the problem into modules, and the neat arrangement of these modules in a hierarchy. The primary characteristics of neat module decomposition are high cohesion and low coupling.

Cohesion

Cohesion is a measure of functional strength of a module. A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

Classification of cohesion

The different classes of cohesion that a module may possess are depicted in figuge 5.2.

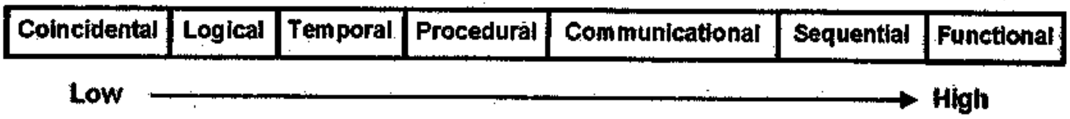


Figure 5.2: Classification of cohesion

Coincidental cohesion: A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely. In this case, the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design.

For example, if in a transaction processing system (TPS), ,he get-input, print-error, and summarize-members functions are grouped into one module. The grouping does not have any relevance to the structure of the problem.

Logical cohesion: A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc. An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.

Temporal cohesion: When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion. The set of functions responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion.

Procedural cohesion: A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message.

Communicational cohesion: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack.

Sequential cohesion: A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next. For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.

Functional cohesion: Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' pay-roll exhibits functional cohesion. Suppose a module exhibits functional cohesion and we are asked to describe what the module does, then we would be able to describe it using a single sentence.

Coupling

Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A module having high cohesion and low coupling is said to be functionally independent of other modules. If two modules interchange large amounts of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity. The interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.

Classification of Coupling

Even if there are no techniques to precisely and quantitatively estimate the coupling between two modules, classification of the different types of coupling will help to quantitatively estimate the degree of coupling between two modules. Five types of coupling can occur between any two modules. This is shown in figure 5.3.

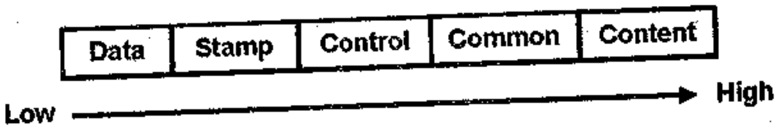


Figure 5.3: Classification of coupling

Data coupling: Two modules are data coupled, if they communicate through a parameter. An example is an elementary data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for the control purpose.

Stamp coupling: Two modules are stamp coupled, if they communicate using a composite data item such as a structure in C.

Control coupling: Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another. An example of control coupling is a flag set in one module and tested in another module.

Common coupling: Two modules are common coupled, if they share data through some global data items.

Content coupling: Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.

Functional independence

A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

Need for functional independence

Functional independence is a key to any good design due to the following reasons:

- **Error isolation:** Functional independence reduces error propagation. The reason behind this is that if a module is functionally independent, its degree of interaction with the other modules is less. Therefore, any error existing in a module would not directly effect the other modules.
- **Scope of reuse:** Reuse of a module becomes possible. Because each module does some well-defined and precise function, and the interaction of the module with the other modules is simple and minimal. Therefore, a cohesive module can be easily taken out and reused in a different program.
- **Understandability:** Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.

5.6 DESIGN STRATEGIES

There exist various strategies to help guide the design process. These strategies include general strategies and also specific strategies or methods. Methods are more specific in that they generally suggest and provide a set of notations to be used with the method, a description of the process to be used when following the method and a set of guidelines in using the method. Such methods are useful as a means of transferring knowledge and as a common framework for teams of software engineers.

General Strategies

Some often mentioned examples of general strategies useful in the design process are divide-and-conquer and stepwise refinement, top-down vs. bottom-up strategies, data abstraction and information hiding, use of heuristics, use of patterns and pattern languages, use of an iterative and incremental approach.

Specific Strategies

The specific strategies include, function-oriented design, object-oriented design, data-oriented design, etc. we discuss some of the specific strategies and the differences between them in the following sections.

5.7 FUNCTION-ORIENTED DESIGN

This is one of the classical methods of software design, where decomposition centers on identifying the major software functions and then elaborating and refining them in a top-down manner. Function oriented or structured design is generally used after structured analysis, thus producing, among other things, data flow diagrams and associated process descriptions. Researchers have proposed various strategies (eg., transformation analysis, transaction analysis) and heuristics (eg., fan-in/fan-out, scope of effect vs. scope of control) to transform a DFD into a software architecture generally represented as a structure chart.

The following are the salient features of a typical function-oriented design approach:

1. A system is viewed as something that performs a set of functions. Starting at this high-level view of

the system, each function is successively refined into more detailed functions. For example, consider a function create-new-library-member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge. Each of these sub-functions may be split into more detailed subfunctions and so on.

2. The system state is centralized and shared among different functions, e.g. data such as member-records is available for reference and updating to several functions.

5.8 OBJECT-ORIENTED DESIGN

In the object-oriented design approach, the system is viewed as a collection of objects. The state is decentralized among the objects and each object manages its own state information. For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data. The functions defined for one object cannot refer or change data of other objects. Objects have their own internal data which define their state. Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from super class.

Function-oriented vs. object-oriented design approach

The following are some of the important differences between function-oriented and object-oriented design.

- Unlike function-oriented design methods, in OOD, the basic abstraction are not real-world functions such as sort, display, track, etc, but real-world entities such as employee, picture, machine, radar system, etc. For example in OOD, an employee pay-roll software is not developed by designing functions such as update-employee-record, get-employee-address, etc. but by designing objects such as employees, departments, etc. Grady Booch sums up this difference as "identify verbs if you are after procedural design and nouns if you are after object-oriented design"

- In OOD, state information is not represented in a centralized shared memory but is distributed among the objects of the system. For example, while developing an employee pay-roll system, the employee data such as the names of the employees, their code numbers, basic salaries, etc. are usually implemented as global data in a traditional programming system; whereas in an object-oriented system these data are distributed among different employee objects of the system. Objects communicate by message passing. The real-world functions however must also be implemented. In OOD, the functions are usually associated with specific real-world entities (objects); they directly access only part of the system state information.

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

5.9 DATA-ORIENTED DESIGN

Data-oriented design shifts the perspective of programming from objects to the data itself. The type of the data, how it is laid out in memory, and how it will be read and processed is the primary concern. The design starts from the data a program manipulates rather than from the function it performs. The software engineer first describes the input and output data structures and then develops the program's control structure based on these data structures.

Programming, by definition, is about transforming data: It's the act of creating a sequence of machine instructions describing how to process the input data and create some specific output data. So it makes sense for us to concentrate primarily on that data instead of on the code that manipulates it. Data-oriented design does not imply that something is data- driven. A data-driven design is usually a design that exposes a large amount of functionality outside of code and lets the data determine the behavior of the design.

5.10 DATA FLOW-ORIENTED DESIGN

Data-flow oriented design involves understanding of how data moves from one module to another in a project. Various modules of a project are identified, and for each of the modules, the inputs provided and the outputs which it provides are found out. The inputs and outputs of each module, with other modules are then associated. For example, module 1 may accept some form of input from the user and result in some output. The output from module 1 may provided as input to module 2, which may perform some process and result in its own output. This output may be passed as input to the next module. This process continues, until the final results are obtained.

Data Flow-oriented Design Steps

- Establish type of information flow
- Determine information flow boundaries
- Map DFD into program structure
- Define control hierarchy by factoring
- Refine model using design measures and heuristics

5.11 REAL TIME DESIGN

Real time system are different from other types of software system. Their correct functioning on the result produced by the system and the time at which these result are produced.

Because of the need to respond to timing demands made by different stimuli/responses, the system architecture must allow for fast switching between stimulus handlers.

Timing demands of different stimuli are different. Real-time systems are therefore usually designed as cooperating processes with a real-time executive controlling these processes.

Design process of real time system

The process of designing a real time system has the following steps:

- Identify the stimuli to be processed and the required responses to these stimuli.
- For each stimulus and response, identify the timing constraints.
- Aggregate the stimulus and response processing into concurrent processes. A process may be associated with each class of stimulus and response. Design algorithms to process each class of stimulus and response. These must meet the given timing requirements.
- Design a scheduling system which will ensure that processes are started in time to meet their deadlines.
- Integrate using a real-time operating system.

5.12 USER INTERFACE DESIGN

Careful user interface design is an essential part of the overall software design process. If a software system is to achieve its full potential, it is essential that its user interface should be designed to match the skills, experience and expectations of its anticipated users. Good user interface design is critical for system dependability. A poorly designed user interface means that users will probably be unable to access some of the system features, will make mistakes and will feel that the system hinders rather than helps them in achieving whatever they are using the system for.

When making user interface design decisions, one should take into account the physical and mental capabilities of the people who use software. Important factors to be considered in user interface design include,

- People have a limited short-term memory. Therefore, if we present users with too much information at the same time, they may not be able to take it all in.
- We all make mistakes, especially when we have to handle too much information or are under stress. When systems go wrong and issue warning messages and alarms, this often puts more stress on users, thus increasing the chances that they will make operational errors.
- We have a diverse range of physical capabilities. Some people see and hear better than others, some people are color-blind, and some are better than others at physical manipulation.

User Interface design principles

User familiarity : The interface should use terms and concepts drawn from the experience of the people who max make most use of the system.

Consistency : The interface should be consistent i.e., wherever possible, comparable operations should be activated in the same way.

Minimal surprise : Users should never be surprised by the behavior of a system.

Recoverability : The interface should include mechanisms to allow users to recover from errors.

User guidance : The interface should provide meaningful feedback when errors occur and provide context-sensitive user help facilities.

User diversity : The interface should provide appropriate interaction facilities for different types of system users. We have different interaction preferences. Some people like to work with pictures, others with text. Direct manipulation is natural for some people, but others prefer a style of interaction that is based on issuing commands to the system.

The general principles are applicable to all user interface designs and should normally be instantiated as more detailed design guidelines for specific organizations or types of system.

The principle of user familiarity suggests that users should not be forced to adapt to an interface because it is convenient to implement. The interface should use terms that are familiar to the user, and the objects manipulated by the system should be directly related to the user's working environment. The underlying implementation of the interface in terms of files and data structures should be hidden from the end user.

The principle of user interface consistency means that system commands and menus should have the same format, parameters should be passed to all commands in the same way, and command punctuation should be similar. Consistent interfaces reduce user learning time. Interface consistency across applications is also important. As far as possibly commands with similar meanings in different applications should be expressed the same way. Errors are often caused when the same keyboard command, such as 'Control-B' means different things in different systems.

The principle of minimal surprise is appropriate because people get very irritated when a system behaves in an unexpected way. As a system is used, users build a mental model of how the system works. If an action in one context causes a particular type of change, it is reasonable to expect that the same action in a different context will cause a comparable change. If something completely different happens, the user is both surprised and confused. Interface designers should therefore try to ensure that comparable actions have comparable effects.

Surprises in user interfaces are often the result of the fact that many interfaces are molded. This means that there are several modes of working (e.g., viewing mode and editing mode), and the effect of a command is different depending on the mode. It is very important that, when designing an interface, one includes a visual indicator showing the user the current mode. The principle of recoverability is important because users inevitably make mistakes when using a system. The interface design can minimize these mistakes (e.g. using menus avoids typing mistakes), but mistakes can never be completely eliminated. Consequently, one should include interface facilities that allow users to recover from their mistakes. These can be of three kinds:

1. Confirmation of destructive actions: If a user specifies an action that is potentially destructive, the system should ask the user to confirm that this is really what is wanted before destroying any information.
2. The provision of an undo facility: Undo restores the system to a state before the action occurred. Multiple levels of undo are useful because users don't always recognise immediately that a mistake has been made.
3. Checkpointing: Checkpointing involves saving the state of a system at periodic inter¬vals and allowing the system to restart from the last checkpoint. Then, when mistakes occur, users can go back to a previous state and start again.

Interfaces should have built-in user assistance or help facilities. These should be integrated with the system and should provide different levels of help and advice. Levels should range from basic information on getting started to a full description of system facilities. Help systems should be structured so that users are not overwhelmed with information when they ask for help.

The principle of user diversity recognizes that, for many interactive systems, there may be different types of users. Some may be casual users who interact occasionally with the system while others may be power users who use the system for several hours each day. Casual users need interfaces that provide guidance, but power users require shortcuts so that they can interact as quickly as possible. Furthermore, users may suffer from disabilities of various types and, if possible, the interface should be adaptable to cope with these. Therefore, one might include facilities to display enlarged text, to replace sound with text, to produce very large buttons and so on.

The principle of recognizing user diversity can conflict with the other interface design principles, since some users may prefer very rapid interaction over, for example, user interface consistency. Similarly, the level of user guidance required can be radically different for different users, and it may be impossible to develop support that is suitable for all types of users. One therefore has to make compromises to reconcile the needs of these users

5.13 USER INTERFACE DESIGN ISSUES

A coherent user interface must integrate user interaction and information presentation. This can be difficult because the designer has to find a compromise between the most appropriate styles of interaction and presentation for the application, the background and experience of the system users, and the equipment that is available.

User interaction means issuing commands and associated data to the computer Command-line interface, and a special-purpose language are early methods to communicate with the machine. Another approach is that of direct manipulation. Here the user interacts directly with objects on the screen. Direct manipulation usually involves a pointing device that indicates the object to be manipulated and the action, which specifies what should be done with that object. For example, to delete a file, one may click on an icon representing that file and drag it to a trash can icon.

Menu selection is a method of interation in which the user selects a command from a list of possibilities called a menu. In this approach, to delete a file, one would select the file icon then select the delete command.

Using the Command language approach The user issues a special command and associated parameters to instruct the system what to do. To delete a file, one would type a delete command with the filename as a parameter of using the command line approachs.

5.14 USER INTERFACE DESIGN PROCESS

User interface design is an iterative process involving close liaisons between users and designers. The 3 core activities in this process are:

- User analysis: Understand what the users will do with the system;
- System prototyping: Develop a series of prototypes for experiment;
- Interface evaluation: Experiment with these prototypes with users.

5.15 USER ANALYSIS

If one doest not understand what the users want to do with a system, one has no realistic prospect of designing an effective interface.User analyses have to be described in terms that users and other designers can understand. Scenarios, where one described typical episodes of use, are one way of describing these analyses.

User preferences on the basis of cognition, cultural background, professional background, expereince etc. must be identified. This analysis of the user should be one of the factors in determining the user interface.

5.16 USER INTERFACE PROTOTYPING

The aim of prototyping is to allow users to gain direct experience with the interface. Without such direct experience, it is impossible to judge the usability of an interface.

Prototyping may be a two-stage process:

(i) Early in the process, paper prototypes may be used;

(ii) The design is then refined and increasingly sophisticated automated prototypes are then developed.

i) **Paper Prototyping**

- Work through scenarios using sketches of the interface.
- Use a storyboard to present a series of interactions with the system.
- Paper prototyping is an effective way of getting user reactions to a design proposal.

ii) **Automated Prototyping**

a) Script-driven prototyping

- Develop a set of scripts and screens using a prototype tool

b) Visual programming

- Use a language designed for rapid development.

c) Internet-based prototyping

- Use a web browser and associated scripts.

5.17 INTERFACE EVALUATION

Some evaluation of a user interface design should be carried out to assess its suitability. The evaluation may be full scale or on a usability specification. Full scale evaluation is very expensive and impractical for most systems. Ideally, an interface should be evaluated against a usability specification. The evaluation can be done using different techniques. These includes:

i Questionnaires for user feedback.

i Video recording of system use and subsequent tape evaluation.

ii Instrumentation of code to collect information about facility use and user errors.

iv The provision of code in the software to collect on-line user feedback.

5.18 SUMMARY

- A software design is a meaningful engineering representation of some software product that is to be built. A design can be traced to the customer's requirements and can be assessed for quality against predefined criteria.
- To analyze the current system and to design the required system software engineer uses some blueprints as a starting point for system design. The blueprints are the actual representation of the analysis done by the software engineers and are called tools of structured analysis.
- The concept of modularity in computer software has been advocated for about five decades. In essence, the software is divided into separately names and addressable components called modules that are integrated to satisfy problem requirements.
- There exist variious strategies for designing a system. Some of thes strategies are, funcitons - oriented design, object oriented design, data oriented design etc.
- Function - oriented design, or structured desgin fouceses first on the functions to be caried out by the sytem, and the data, data flow follow this step.
- In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities). The state is decentralized among the objects and each object manages its own state information.
- In data onented design, the data that the functions one to process, is first identified & structured and then, the functions designed.
- Careful user interface design is an essential part of the overall software design process. If a software system is to achieve its full potential, it is essential that its user interface should be designed to match the skills, experience and expectations of its anticipated users.

5.19 UNIT-END QUESTIONS

1 What are the steps followed in software design phase?

2 Explain Design concepts in software engineering?

3 Write a short note on software design tools?

4 What do mean by modular designing?

5 What is function oriented design?

6 What is the difference between function oriented and object oriented design?

7 What is real time design?

8 What is the role of user interface in software design and why it is important?

9 How do function oriented and data oriented design differ?

10 Discuss the principles of user interface design?

5.20 FURTHER READINGS

1. http://scitec.uwichill.edu.bb/cup/online/cs22/desig_-_concepts_and_principles.htm.

2. Software Engineering: A Practioner's Approach, Roger S. Pressman, McGrawHill.

3. http://cnx.org/context/m14630/latest

UNIT - VI

PROJECT MANAGEMENT

Structure of the Unit

6.0 OBJECTIVES

After studying this unit, students will learn

- Concept and need of Project management
- Management activities
- Project planning

- Project tracking and scheduling
- Risk management

6.1 INTRODUCTION TO PROJECT MANAGEMENT

Project Management is an integral part of software development. It involves planning, monitoring, and control of the people, process, and events that occur as software evolves from the events and from a preliminary concept to an operational implementation. It is an important task in software development because building computer software is a complex task, particularly if it involves many people working over a long period of time. Hence software projects need to be managed. This management is done at various levels. For e.g. Software Engineer manages his/her day to day activities, planning, monitoring, and controlling technical tasks. Project Managers plan, monitor, and control the work of a team of software engineers. Senior managers coordinate the interface between business and software professionals. For a large project a proper management process is essential for success.

6.2 MANAGEMENT ACTIVITIES

The job description for software managers varies tremendously depending on the organisation and on the software product being developed. However project management is composed of several different types of activities such as:

- Proposal writing
- Project planning and scheduling
- Project costing
- Project monitoring and reviews
- Personnel selection and evaluation
- Report writing and presentation

Which can be further decomposed as Analysis and design of objectives and events, Planning the work according to the objectives, Assessing and controlling risk (or Risk Management), Estimating resources, Allocation of resources, Organizing the work, Acquiring human and material resources, Assigning tasks, Directing activities, Controlling project execution, Tracking and reporting progress (Management information system), Analyzing the results based on the facts achieved, Defining the products of the project, Forecasting future trends in the project, Quality Management, Issues management, Issue solving, Defect prevention, Identifying, managing & controlling changes, Project closure (and project debrief), Communicating to stakeholders, Increasing / decreasing a company's workers.

The first task in software project management is writing a proposal to carry out that project. The proposal describes the objectives of the project and how it will be carried out. It usually includes cost and schedule estimates. It may justify why the project contract should be awarded to a particular organization or a team.

Project planning is concerned with identifying the activities, milestones and deliverables produced by a project.

Project cost estimation estimates the cost associated with the esources required to accomplish the project plan. It is based on planning information about the project. Information for initial cost astimation comes from the feasibility study and requirement analysis. It is important for making good management decisions in a software project.

Project monitoring is a continuing project activity. The management must keep track of the progress of the project and compare actual and planned progress and costs. The monitoring can be done using formal mechanism or by the help of informal discussions with project staff.

Project managers usually have to select people to work on their project. Ideally, skilled staff with apppropriate experience will be available to work on the project. Sometimes management has to settle for less than ideal team due to lack of project budget, less availability of experienceal staff etc.

Project managers are usually responsible for reporting on the project to both the client and contract organisation. Project manager must write concise, coherent documents which abstract critical information from detailed project reports. They must be able to present this information during progress reviews.

6.3 MANAGEMENT STRUCTURES

In the scheduling of the project, detailed scheduling is done only after actual assignment of the people has been done, as task assignment needs information about the capabilities of team members. The project team is led by a project manager who does the planning and task assignment. This form of hierarchical management structure is fairly common and manager was earlier called the chief programmer team. The project manager is responsible for all major technical decisions of the project. He does most of the design and assigns coding of the different parts of the design to the programmers. The management structure of team typically consists of programmers, testers, a configuration controller, and possibly a librarian for documentation. There may be other roles like database manager, backup project manager or a back up configuration controller. These roles are logical roles and one person may do multiple such roles. For a small project, a one level hierarchy suffices. For large projects the structure can be extended by partitioning the project into modules, and having module leaders who are responsible for all tasks related to their module and has a team with them for performing these tasks.

6.4 PROJECT PLANNING

Project planning is a part of project management, which relates to the use of schedules such as Gantt charts to plan and subsequently report progress within the project environment. Initially, the project scope is defined and the appropriate methods for completing the project are determined. Following this step, the durations for the various tasks necessary to complete the work are listed and grouped into a work breakdown structure. The logical dependencies between tasks are defined using an activity network diagram that enables identification of the critical path. Float or slack time in the schedule can be calculated using project management software. Then the necessary resources can be estimated and costs for each activity can be allocated to each resource, giving the total project cost. At this stage, the project plan may be optimized to achieve the appropriate balance between resource usage and project duration to comply with the project objectives. Once established and agreed, the plan becomes what is known as the baseline. Progress is measured against the baseline throughout the life of the project.

Software planning involves estimating how much time, effort, money, and resources will be required to build a specific software system. After the project scope is determined and the problem is decomposed into smaller problems, software managers use historical project data (as well as personal experience and intuition) to determine estimates for each. The final estimates are typically adjusted by taking project complexity and risk into account. The resulting work product is called a project management plan.

6.4.1 Project planning Objectives

A plan for a project is devised with the following objectives:

- To provide a framework that enables software manager to make a reasonable estimate of resources, cost, and schedule.
- Project outcomes should be bound by 'best case' and 'worst case' scenarios.

- Estimates should be updated as the project progresses.

6.4.2 Estimation Reliability Factors

The estimations done during the project planning phase rely on the following factors:

- Project complexity
- Project size
- Degree of structural uncertainty (degree to which requirements have solidified, the ease with which functions can be compartmentalized, and the hierarchical nature of the information processed)
- Availability of historical information

6.4.3 Project planning Activities

Following are the activities associated with software project planning:

(i) **Software Scope:** The first activity in software project planning is the determination of software scope. Software scope describes the data and control to be processed, function, performance, constraints, interfaces and reliability. Functions described in the statement of scope are evaluated and in some cases refined to provide more detail prior to the beginning of estimation.

Obtaining necessary information for scope : The most commonly used technique to bridge the communication gap between the customer and developer and to get communicatin process started is to conduct a preliminary meeting or interview. As the communication is initiated, the analyst must ask some context free questions which focus on the customer, the overall goals and benefits. For e.g.

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution?

The next set of questions put to the client, enables the analyst to gain a better understanding of the project:

- Would you chracterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me the environment I in which the solution will be used ? etc.

The final set of questions focuses on effectiveness of the meeting like,

- Are you the right person to answer these questions?
- Are my questions relevant to the problem that you have?
- Can anyone provide additional information ? etc.

Sometimes a number of independent investigators develop a team oriented approach to requirements gathering that can be applied to establish the scope of the project.

(ii) **Feasibility :** Once scope has been identified, it is reasonable to ask:" Can we build software to meet this scope? Is the project feasible?" A software team must work to determine if it can be done within the dimensions just noted.

(iii) Resources: The next Software Planning task is estimation of the resources required to accomplish the software development effort.

Human resources: The planner begins by evaluating scope and selecting the skills required to complete development. Both organizational position e.g. manager, senior software engineer and speciality(e.g. telecommunications, database, client/server) are specified. For a relatively small project a single individual may perform all software engineering tasks, consulting with specialists as required. The number of people required for a software project can be determined only after an estimate of development effort is made.

Reusable software resources: Component based software engineering emphasizes reusability i.e. the creation and reuse of software building blocks. Such building blocks, often called components, must be cataloged for easy reference, standardized for easy application, and validated for easy integration.

Environmental resources :The environmental resources that support the software project, often called the software engineering environment, incorporates Hardware and Software. Hardware provides a platform that supports the tools (Softwrae) required to produce the work products that are an outcome of good software engineering practice. A project planner has to prescribe the time window required for hardware and software and verify that these resources will be available in time.

6.4.4 Project Cost Estimation

In early days of computing, software costs constituted small percentage of the overall computer based system cost. Today, software is the most expensive element of virtually all computer based systems.

Software cost and effort estimation will never be an exact science. Too many variables-human, technical, environmental, political- can affect the ultimate cost of software and effort applied to develop it.

Software Project Estimation Options

To achieve reliable cost and effort estimates, a number of options arise these include,

Delay estimation until late in the project. Base estimates on similar projects already completed. Use simple decomposition techniques to estimate project cost and effort. Use empirical models for software cost and effort estimation. Automated tools may assist with project decomposition and estimation.

The Decomposition Techniques such as

Software sizing (fuzzy logic, function point, standard component, change), Problem-based estimation (using LOC decomposition focuses on software functions, using FP decomposition focuses on information domain characteristics) and Process-based estimation (decomposition based on tasks required to complete the software process framework) aid in estimating the sizes hence cost of the software.

The Empirical Models used for estimation are,

Typically derived from regression analysis of historical software project data with estimated person-months as the dependent variable and KLOC or FP as independent variables, statec estimation models such as Constructive Cost Model (COCOMO) and Dynamic estimation model such as Software Equation

It may be more cost effective to acquire a piece of software rather than develop it. Decision tree analysis provides a systematic way to sort through the make-buy decision. As a rule outsourcing software development requires more skillful management than does in-house development of the same product.

6.5 PROJECT SCHEDULING AND TRACKING

The process of building and monitoring schedules for software development projects is known as project scheduling and tracking. To build complex software systems, many engineering tasks need to occur in parallel with one another to complete the project on time. The output from one task often determines when another may begin. It is difficult to ensure that a team is working on the most appropriate tasks without building a detailed schedule and sticking to it.

6.5.1 Software Project Scheduling

Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. The schedule evolves over time. A number of basic principles guide software project scheduling.

- Compartmentalization - the product and process must be decomposed into a manageable number of activities and tasks

- Interdependency - tasks that can be completed in parallel must be separated from those that must completed serially

- Time allocation - every task has start and completion dates that take the task interdependencies into account

- Effort validation - project manager must ensure that on any given day there are enough staff members assigned to completed the tasks within the time estimated in the project plan

- Defined Responsibilities - every scheduled task needs to be assigned to a specific team member

- Defined outcomes - every task in the schedule needs to have a defined outcome (usually a work product or deliverable)

- Defined milestones - a milestone is accomplished when one or more work products from an engineering task have passed quality review

Each of these principles is applied as the project schedule evolves.

Relationship between People and Effort

In small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases more people must become involved. There is a common myth that is still believed by many managers who are responsible for software development effort: "If we fall behind schedule, we can always add more programmers and catch up later in the project." Unfortunately, adding people to a project after it is behind schedule often causes the schedule to slip further. The relationship between the number of people on a project and overall productivity is not linear (e.g. 3 people do not produce 3 times the work of 1 person, if the people have to work in cooperation with one another) The main reasons for using more than 1 person on a project are to get the job done more rapidly and to improve software quality.

An Empirical relationship

Relation between time to complete the project and human effort applied to project is nonlinear. The number of lines of code L, is related to effort and development time by the equation

$L= P^* E^{1/3} t^{4/3}$

Where E is development effort in person-months, P is a productivity parameter that reflects a variety of factors that lead to high quality software engineering work and, t is the project duration in calendar months.

Rearranging this software equation, we can arrive at an expression for development effort E:

$E = L^3/(P^3t^4)$

Where E is the effort expended over the entire life cycle for software development and maintenance and t is the development time in years.

Project Effort Distribution

A recommended distribution of effort across the definition and development phases is often referred to as the 40-20-40 rule. Forty percent of all effort is allocated to front end analysis and design. A similar percentage is applied to back-end testing. Coding is taking 20 percent efforts only. This effort distribution should be used as guideline only. Generally accepted guidelines are:

- 02-03 % planning
- 10-25 % requirements analysis
- 20-25 % design
- 15-20 % coding
- 30-40 % testing and debugging

Software Project Types

A number of different process models exist for software project. Regardless of whether a software team chooses a linear sequential paradigm, an iterative paradigm, an evolutionary paradigm, a concurrent paradigm or some permutation, the process model is populated by a set of tasks that enable a software team to define, develop, and ultimately support computer software.

No single set of tasks is appropriate for all projects. An effective software process should define a collection of task sets, each designed to meet the needs of different types of projects.

A task set is a collection of software engineering work tasks, milestones, and deliverables that must be accomplished to complete a particular project. Task sets are designed to accommodate different types of projects and different degrees of rigor. Although it is difficult to develop a comprehensive taxonomy of software project types, most software organizations come across the following projects:

- Concept development - initiated to explore new business concept or new application of technology
- New application development - new product requested by customer
- Application enhancement - major modifications to function, performance, or interfaces (observable to user)
- Application maintenance - correcting, adapting, or extending existing software (not immediately obvious to user)
- Reengineering - rebuilding all (or part) of a legacy system

Defining a Task Network

Individual tasks and subtasks have interdependencies based on their sequence. In addition, when more than one person is involved in a software engineering project, it is likely that development activities and tasks will be performed in parallel. A *task network*, also called an activity network, is a graphic representation of the task flow for a project. Following figure shows the task network for a concept development project:

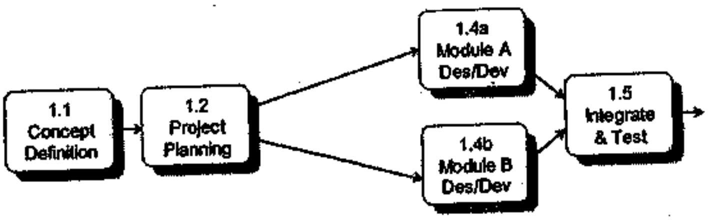


Figure 6.1: Task network for a concept development

Scheduling

Scheduling tools may be used to schedule any non-trivial project some of the tools used include,

- Timeline (Gantt) charts enable software planners to determine what tasks will be need to be conducted at a given point in time based on estimates for effort, start time, and duration for each task.

- The best indicator of progress is the completion and successful review of a defined software work product.

- Time-boxing is the practice of deciding a priori the fixed amount of time that can be spent on each task. When the task's time limit is exceeded, development moves on to the next task.

6.5.2 Tracking the schedule

The project schedule provides a road map for a software project manger. If it has been properly developed, the project schedule defines the tasks and milestones that must be tracked and controlled as the project proceeds. Tracking can be accomplished in a number of different ways:

- Conducting periodic project status meetings in which each team member reports progress and problems.

- Evaluating the results of all reviews conducted throughout the software engineering process.

- Determine whether formal project milestones have been accomplished by the scheduled date.

- Comparing actual start date to planned start date for each project task.

- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.

- Using earned value analysis to access progress quantitatively.

Earned Value Analysis

Earned value is a quantitative measure of percent of project completed so far. The total hours to complete the entire project are estimated and each task is given an earned value based on its estimated percentage contribution to the total.

To determine the earned value, the following steps are performed:

1. The budgeted cost of work scheduled (BCWS) is determined for each work task represented in the schedule.

2. The BCWS values for all work tasks are summed to derive the budget at completion, BAC. Hence,

i. BAC= "(BCWS$_k$) for all tasks k

3. The value for budgeted cost of work performed (BCWP) is computed. The value for BCWP is the sum of the BCWS values for ali tasks that have actually been completed by a point in time on the project schedule.

4. Progress indicator SPI is an indication of the efficiency with which the project is utilizing scheduled resources,

a. SPI =BCWP/BCWS

b. SV is simply an absolute indication of variance from the planned schedule.

i. SV=BCWP-BCWS

5. Percent complete=BCWP/BAC provides a quantitative indication of the percent of completeness of the project at a given point in time , t.

Error Tracking

Allows comparison of current work to past projects and provides a quantitative indication of the quality of the work being conducted. The more quantitative the approach to project tracking and control, the more likely problems can be anticipated and dealt with in a proactive manner.

6.6 RISK MANAGEMENT

Risks are potential problems that might affect the successful completion of a software project. Risks involve uncertainty and potential losses. Risk analysis and management are intended to help a software team understand and manage uncertainty during the development process. The important thing is to remember that things can go wrong and to make plans to minimize their impact when they do. The work product is called a Risk Mitigation, Monitoring, and Management Plan (RMMM).

Risk management is a structured approach to managing uncertainty related to a threat, a sequence of human activities including risk assessment, strategies development to manage risks, and mitigation of risk using managerial resources.

The strategies include transferring the risk to another party, avoiding the risk, reducing the negative effect of the risk, and accepting some or all of the consequences of a particular risk.

Some traditional risk managements strategies are focused on risks stemming from physical or legal causes (e.g. natural disasters or fires, accidents, death and lawsuits). Financial risk management, on the other hand, focuses on risks that can be managed using traded financial instruments.

The objective of risk management is to reduce different risks related to a preselected domain to the level accepted by society. It may refer to numerous types of threats caused by environment, technology, humans, organizations and politics. On the other hand it involves all means available for humans, or in particular, for a risk management entity (person, staff, organization).

Risk Strategies

- Reactive strategies - Also known as fire fighting, reactive strategies are used very commonly. In these the project team sets resources aside to deal with problems and does nothing until a risk becomes a problem.

- Proactive strategies - In proactive strategies, risk management begins long before technical work starts, Risks are identified and prioritized by importance, then team builds a plan to avoid risks if they can, or minimize them if the risks turn into problems.

Software Risks

Risk always involves two characteristics:

- Uncertainty-The risk may or may not happen, that is, there are no 100% probable risks.
- Loss- If the risk becomes a reality, unwanted consequences or losses will occur.

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

- Project risks - threaten the project plan
- Technical risks - threaten product quality and the timeliness of the schedule
- Business risks - threaten the viability of the software to be built (market risks, strategic risks, management risks, budget risks)
- Known risks - predictable from careful evaluation of current project plan and those extrapolated from past project experience
- Unknown risks - some problems simply occur without warning.

6.7 RISK IDENTIFICATION AND PROJECTION

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step towards avoiding them when possible and controlling them when necessary.

There are two different types of risks for each of the categories that have been mentioned above.

- Product-specific risks –can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the project at hand. To identify product specific risks, the project plan and software statement of scope are examined and identify any special characteristics of the product that may threaten the project plan.
- Generic risks - are potential threats to every software product.

One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

- Product size
- Business impact
- Customer characteristics
- Process definition
- Development environment
- Technology to be built
- Staff size and experience

Risk Components and Drivers

The components of risks are:

- Performance, Cost, Support, Schedule
- The impact of a risk could be:

- Negligible, Marginal, critical, catastrophic

The risk drivers affecting each risk component are classified according to their impact category and the potential consequences of each undetected software fault or unachieved project outcome considering are described.

Risk Projection (Estimation)

Project risk estimation contributes to considering risks in a manner that leads to prioritization of risks so that resources could be allocated where they will have the maximum impact. The four primary risk projection steps are:

- Establish a scale that reflects the perceived likelihood of each risk
- Delineate the consequences of the risk
- Estimate the impact of a risk on the project and product
- Note the overall accuracy of the risk projection to avoid misunderstandings

Risk Table Construction

A risk table provides a project manager with a method for risks projection. The steps involved in the construction of a risk table are:

- List all risks in the first column of the table
- Classify each risk and enter the category label in column two
- Determine a probability for each risk and enter it into column three
- Enter the severity of each risk (negligible, marginal, critical, catastrophic) in column four
- Sort the table by probability and impact value
- Determine the criteria for deciding where the sorted table will be divided into the first priority concerns and the second priority concerns
- First priority concerns must be managed (a fifth column can be added to contain a pointer into the RMMM)

Assessing Risk Impact

The project team estimates the effect of risks on the project. The consequences of risks can be determined as follows:

- Factors affecting risk consequences - nature (types of problems arising), scope (combines severity with extent of project affected), timing (when and how long impact is felt)
- If costs are associated with each risk table entry Halstead's risk exposure metric can be computed (RE = Probability * Cost) and added to the risk table.

Risk Assessment

Risk assessment is done to obtain quantitiative or qualitative value of risk related to a tangible situation and predicatable threat. Risk Assessment can be done using the following methods:

- Define referent levels for each project risk that can cause project termination (performance degradation, cost overrun, support difficulty, schedule slippage).
- Attempt to develop a relationship between each risk triple (risk, probability, impact) and each of the reference levels.
- Predict the set of referent points that define a region of termination, bounded by a curve or areas of uncertainty.

- Try to predict how combinations of risks will affect a referent level.

Risk Refinement

Risk refirement is the

- Process of restating the risks as a set of more detailed risks that will be easier to mitigate, monitor, and manage.

6.8 RISK MITIGATION, MONITORING, AND MANAGEMENT (RMMM)

Risks are potential problems that might affect the successful completion of a software project. Risks involve uncertainty and potential losses. Risk analysis and management is intended to help a software team understand and manage uncertainty during the development process. The important thing is to remember that things can go wrong and to make plans to minimize their impact when they do. The work product is called a **Risk Mitigation, Monitoring, and Management Plan (RMMM).**

The implementation of a RMMM Plan is of vital strategic importance. However, in the operational phase, it needs to be decided, whether the approach to risk management should be proactive or reactive.

Reactive Risk Management

The reactive approach is taken usually to obtain information regarding risk and errors in the preliminary as well as monitoring & follow-up phases of the porject.The reactive approach however, does not identify the risks in advance and takes necessary actions only when these risks become problems.

Proactive Risk Management

The proactive approach to Risk Management is a more rational approach as it involves indentifying the risks before they become problems, and planning how to mitigate, monitor and manage them.

In this approach formal risk analysis is performed, the organization corrects the root causes of risk while examining risk sources that lie further the bounds of the software.

The steps involved in proactive RMMM are,

- Risk mitigation (proactive planing for risk avoidance)
- Risk monitoring (assessing whether predicted risks occur or not, ensuring risk aversion steps are being properly applied, collect information for future risk analysis, attempt to determine which risks caused which problems)
- Risk management and contingency planing (actions to be taken in the event that mitigation steps have failed and the risk has become a live problem)

Safety Risks and Hazards

Identifiying safety risks & hazardsis a software quality assurance activity that focuses on identification & assessement of potential hazards that may affect software negatively and cause an entire system to fail. Software safety involves good software engineering practices so as to ascertain risk assessesment & management, implementation of control measures & validation of the efficacy of control measures. It is thus the collection of activities that assure the safe operation of software.

Risk Information Sheets

Risk information sheets contain all the detailed data associated with risk. It is an alternative to RMMM in which each risk is documented individually. Often risk information sheets (RIS) are maintained using a database system. Risk ID, date, probability, impact, description, refinement, mitigation/monitoring, management/contingency/trigger, status, originator, assigned staff member are components of Risk information sheets

6.9 PRODUCTIVITY

Productivity is a manufacturing system that can be measured by counting the number of units which can be produced and dividing this by the number of person hours required to produce them. However, for any software problem, there are many different solutions which have different attributes. One solution may execute more efficiently while another may be more readable and easier to maintain. Productivity hence is a quality of software production process. it measures the efficiency of the process. An efficient process results in faster delivery of the products.

6.10 PROJECT MILESTONES

When planning a project, a series of milestones should be established. A milestone is an endpoint of a software process activity. At each mile stone, there should be a formal output, such as a report, that can be presented to management. Milestone reports need not to be large documents. They may simply be a short report of achievements in a project activity. Milestones should represent the end of a distinct, logical stage of the project.

ACTIVITIES

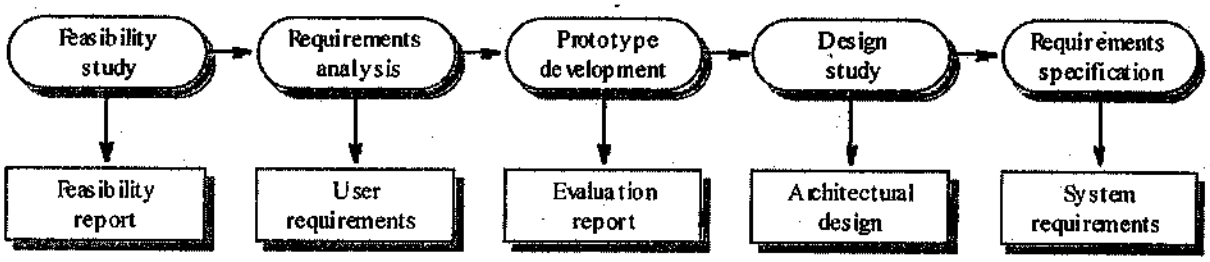


Figure 6.2 : Project Milestones

6.11 WORK DEFINITION, ALLOCATION AND ASSIGNMENTS

At the time of software planning, a plan is been set which draws up at the start of project and is be used as the driver of the project. Most of the plan includes the objectives of the project and constraints which affect the project management. It also describes the way in which the development team is organized, the people involved and their roles in the team. The plan also shows the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity. After identification of activities, its dependency with other activities is judged. It is also estimated that how much time is required to complete this task. Project scheduler coordinates which activities are carried out in parallel and then coordinates these parallel activities and organize the work so that the workforce is used optimally. In estimating schedules, manager must also estimate the resources needed to complete each task. The principal resource is the human effort needed. In estimating schedules, managers should not assume that every stage of the project will be problem free. Individuals working on a project may fall ill or may leave, hardware may breakdown and essential support software or hardware may be delivered late so a good rule of thumb is to estimate as if nothing will go wrong, then increase the estimate to cover anticipated problems. The project schedule is usually represented as a set of charts showing work break down, activities dependencies and staff allocations.

6.12 SUMMARY

Software project management is concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organizations developing and procuring the software.

Project planning is the most time consuming activity. The project plan sets out the resources available to the project, the work breakdown and a schedule for the work.

Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. The schedule evolves over time.

Risks are potential problems that might affect the successful completion of a software project. Risk analysis and management are intended to help a software team understand and manage uncertainty during the development process.

It is important to remember that things can go wrong and to make plans to minimize their impact when they do. Risk Mitigation, Monitoring, and Management Plan (RMMM) must be generated to take care of unplanned events & situations.

6.13 UNIT END QUESTIONS

1 What do you mean by project management? List and discuss the various activities involved in that.

2 What is the importance of project scheduling is Software engineering? What are the factors that affect project scheduling?

3 Define task network. Discuss its use in scheduling.

4 Describe the difference between reactive and proactive risks.

5 What is RMMM? Discuss its role in project management.

6.14 FURTHER READINGS

1. Software Engineering: A Practitioner's Approach, Roger S. Pressman, Sixth edition, McGraw Hill Publication
2. Software Engineering, Ian Sommerville, Sixth Edition, Pearson Education
3. An Integrated Approach to Software Engineering, Pankaj Jalote, Narosa Publishing House
4. http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/Presentations/PPT/4

UNIT - VII

SOFTWARE COST ESTIMATION, METRICS AND MEASURES

Structure of the Unit

7.0 OBJECTIVES

After studying this unit, students will be able to understand,

- The concept of software productivity
- The fundamentals of software costing
- Different techniques for cost estimation
- Software metrics and measurements

7.1 INTRODUCTION

Software cost estimation is important for making good management decisions in a software project. It is also connected to determining how much effort and time a software project requires. Cost estimation has several uses:

- It establishes a firm, reliable budget for an in house project.
- It facilitates competitive contract bids when a software house wants to get the contract for developing a specific software system. It is easier to make a close competitive bid with accurate cost estimates. If they are too high, another competitor will outbid. If they are too low, money will be lost on the deal.
- It determines whether it is cheaper to develop software in-house, to contract for outside development, or to buy a software product off-the-shelf and customize it.

Another aspect is the Productivity estimates which are usually based on measuring some attributes of the software and dividing this by the total effort required for development. It is the responsibility of the project manager to make accurate estimations of effort and cost.

From time to time, different software metrics have been developed to quantify various attributes of a software product. Broadly these may be grouped into two categories. These are

- Product metrics and
- Process metrics.

The following sections discuss the cost estimation methods and metrics & measures.

7.2 SOFTWARE PRODUCTIVITY

Productivity in a manufacturing system can be measured by counting the number of units which can be produced and dividing this by the number of person hours required to produce them. The productivity of engineers in the software development process may have to be estimated by managers. These estimates may be needed for project estimation and to assess whether process or technology improvements are effective. These productivity estimates are usually based on measuring some attributes of the software and dividing this by the total effort required for development. There are two types of measures which have been used:

Size related measures: These are related to the size of output from an activity like number of lines in the code.

Function related measures: These are related to overall functionality of the delivered software. For e.g. Function points, object points.

Lines of source code per programmer-month is a widely-used metric in productivity measurement. This is computed by counting the total number of lines of source code which are delivered. The count is divided by the total time in programmers months required to complete the project. This time therefore includes the time required for analysis and design, coding, testing and documentation.

An alternative to using code size as the estimated product attribute is to use some measure of the functionality of the code. The best known of these measures is the function point count. Function points are language independent so productivity in different programming languages can be compared. Productivity is expressed as function points produced per month. The total number of function points in a program is computed by measuring or estimating the following program features:

- External inputs and outputs
- User interactions

- External interfaces
- Files used by the system.

Each of these is individually accessed for complexity and given a weighting value that varies from 3 to 15 for complex internal files. The unadjusted function point count (UFC) is computed by multiplying each raw count by the estimated weight and summing all values.

UFC= $\sum$ (number of elements of given type) *(weight)

UFC is multiplied by the project complexity factors to produce final function point count.

Object points are an alternative to function points when 4 GLs or comparable languages are used for software development. Object points are not object classes that may be produced when an object oriented approach is taken to software development. Rather, the number of object points in a program is a weighted estimate of:

1. The number of separate screens that are displayed. Simple screen count as 1 object point, moderately complex screens count as 2 and very comlex screens count as 3 object points.
2. The number of reports that are produced. For simple reports, count 2 object points, for moderately complex reports, count 5 and for reports which are likely to be difficult to produce, count 8 object points.
3. The number of 3 GL modules that must be developed to supplement the 4 GL code. Each 3 GL module count as 10 object points.

The productivity of individual engineers working in an organization is affected by a number of factors like

- Application domain experience
- Process quality
- Project size
- Technology support
- Working environment

However, individual differences in ability are more significant than any of these factors.

The problem with measures expressed as volume/time is that they take no account of non functional software characteristics such as reliability, maintainability, etc. They imply that more always means better. These measures also do not take into account the possibility of reusing the software produced. What we really want to estimate is the cost of deriving a particular system with given functionality, quality, performance, maintainability, etc. This is only indirectly related to tangible measures such as system size. So productivity measures must therefore be used only as a guide. They should not be used without careful analysis.

7.3 ESTIMATION TECHNIQUES

It is the responsibility of the project manager to make accurate estimations of effort and cost. This is particularly true for projects subject to competitive bidding where a bid too high compared with competitors would result in loosing the contract or a bid too low could result in a loss to the organization. This does not mean that internal projects are unimportant. From a project leader's estimate the management often decide whether to proceed with the project. Industry has a need for accurate estimates of effort and size at a very early stage in a project. However, when software cost estimates are done early in the software development process the estimate can be based on wrong or incomplete requirements. A software cost estimation (SCE) process is the set of techniques and procedures that an organization uses to arrive at an estimate. An important aspect of software projects is to know the cost and the major contributing factor is effort. To do so, one or more of the techniques described in table 7.1 may be used.

Table 7.1 : Estimation Techniques

Technique	Description
Algoritmic Cost modeling	A model is developed using historical cost information that relates some software metric (usually its size) to the project cost. An estimate is made of that metric and the model predicts the effort required.
Expert Judgement	Several experts on the proposed software development techniques and the application domain are consulted. They each estimate the project cost. These estimates are compared and discussed. The estmation process iterates until an agreed estimate is reached.
Estimation by analogy	This technique is applicable when other projects in the same application domain have been completed. The cost of a new project is estimated by analogy with these completed projects.
Parkinson's law	Parkinson's law states that work expands to fill the time available. The cost is determined by available resources rather than by objective assessment. If the software has to be delivered in 12 months and five people are available, the effort required is estimated to be 60 person months.
Pricing to win	The software cost is estimated to be whatever the customer has available to spend on the project. The estimated effort depends on the the customer's budget and not on the software functionality.

These approaches to cost estimation can be tackled using either a top down approach or a bottom up approach. A top down approach starts at the system level. The estimator starts by examining the overall functionality of the product and how that functionality is provided by incereasing sub functions. The costs of system level activities such as integration, configuration management and documentation are taken into account.

The bottom up approach, by contrast, starts at the component level. The system is decomposed into components and the effort required to develop each of these is computed. These costs are then added to give the effort required for the whole system development.

Each estimation technique has its own strengths and weaknesses. For large projects, you should use several cost estimation techniques and compare their results. If these predict radically different costs, this suggests that enough costing information is not available. More information and repetition of the costing process until the estimate converges, is required.

Why SCE is difficult and error prone ?

- Software cost estimation requires a significant amount of effort to perform it correctly.
- SCE is often done hurriedly, without an appreciation for the effort required.
- Experience at developing estimates, especially for large projects, is necessary

- Human bias i.e. An Estimator is likely to consider how long a certain portion of the system would take, and then to merely extrapolate this estimate to the rest of the system, ignoring the non-linear aspects of software development.

The causes of poor and inaccurate estimation

- Imprecise and drifting requirements
- New software projects are nearly always different form the last.
- Software practitioners don't collect enough information about past projects.
- Estimates are forced to match the resources available.

7.4 ALGORITHMIC COST MODELING

Most work carried out in the software cost estimation field has so far focused on algorithmic cost modeling. In this process costs are analyzed using mathematical formulae linking costs or inputs with metrics to produce an estimated output. The formulae used in a formal model arise from the analysis of historical data. The accuracy of the model can be improved by calibrating the model to the specific development environment, which basically involves adjusting the weightings of the metrics. There are a variety of different models available, the best known are Boehm's COCOMO, Putman's SLIM, and Albrecht's' function points. On an initial instinct one might expect formal models to be advantageous for their 'off-the-shelf' qualities, but after close observation this is regarded as a disadvantage by cost estimators due to the additional overhead of calibrating the system to the local circumstances. However, the more time spent calibrating a formal model, the more accurate the cost estimates should be.

In terms of the estimation process, nearly all algorithmic models deviate from the classical view of the cost estimation process.

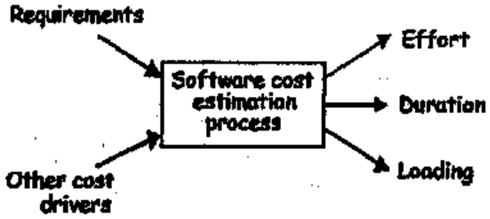


Figure 7.1. Classical view of the algorithmic cost estimation process

An input requirement of an algorithmic model is to provide a metric to measure the size of the finished system. Typically lines of source code are used, this is obviously not known at the start of the project. SLOC is also very dependant on the programming language and programming environment, this is difficult to determine at an early stage in the problem especially as requirements are likely to be sketchy. Despite this SLOC has been the most widely used size metric in the past, but current trends indicate that it is fast becoming less stable. This is probably due to the changes in software development process in recent years highlighted with a tendency to use prototyping, case tools and so forth. An alternative is to use function points which are related to the functionality of the software rather than its size. A more recent approach is to use object points. This is in comparison a new methodology and has not been publicized in the same depth as function points and SLOC. In essence the method is very similar to function points but counts objects instead of functions. Its recent rise has been prompted by the interest in the object orientation revolution.

Algorithmic models generally provide direct estimates of effort or duration. The main input is usually a prediction of software size. Effort prediction models take the general form :

effort = p*S

where p = (1/productivity rate)

where p is a productivity constant and S is the size of the system.

Once the value for p is known. E.g. productivity = 450 source lines of code per month, making p = 0.0022 and the size of the system has been estimated at 8500 KLOC.

effort = 0.0022 * 8500

effort = 18.7 person months

The example above assumes that the relationship between effort and size is linear. Most models allow for non-linear relationships by introducing economies or dis-economies of scale. The general formula being:

7.5 PROJECT DURATION AND STAFFING

In addition to estimating the effort required to develop a software system and the over all cost of that effort, project managers must also estimate how long the software will take to develop and when staff will be needed to work on the project. The development time for the project is called the project schedule. Increasingly, organizations are demanding shorter development schedules so that their products can be brought to market before their competitor's.

The relationship between the number of staff working on a project, the total effort required and the development time is not linear. As the number of staff increases, more effort may be needed. People must spend more time communicating. More time is required to define interfaces between the parts of the system. Doubling the number of staff does not mean that the duration of the project will be halved.

Dividing the effort required on a project by the development schedule does not give a useful indication of the number of people required for the project team. Generally, the number of people employed on a software project builds up from a relatively small number to a peak and then declines.

Only a small number of people are needed at the beginning of a project to carry out planning and specification. As the project progresses and more detailed work is required, the number of staff builds up to a peak. After implementation and unit testing is complete, the number of staff required starts to reduce until it reaches one or two when the product is delivered. A very rapid build up of project staff often correlates with project schedule slippage. Project manager should therefore avoid adding too many staff to a project early in its lifetime.

7.6 EMPIRICAL MODELS

The structure of empirical estimation models is a formula derived from data collected from past software projects that uses software size to estimate effort. Size, itself, is an estimate, described as either lines of code (LOC) or function points (FP). No estimation model is appropriate for all development environments, development processes, or application types. Models must be customised (values in the formula must be altered) so that results from the model agree with the data from the particular environment.

The typical formula of estimation models is:

$\mathbf{E = a + b(S)^c}$

where;

E represents effort, in person months,

S is the size of the software development, in LOC or FP, and,

a, b, and c are values derived from data.

The relationship between development effort and software size is generally proportional, and tends to become relatively stable with larger size.

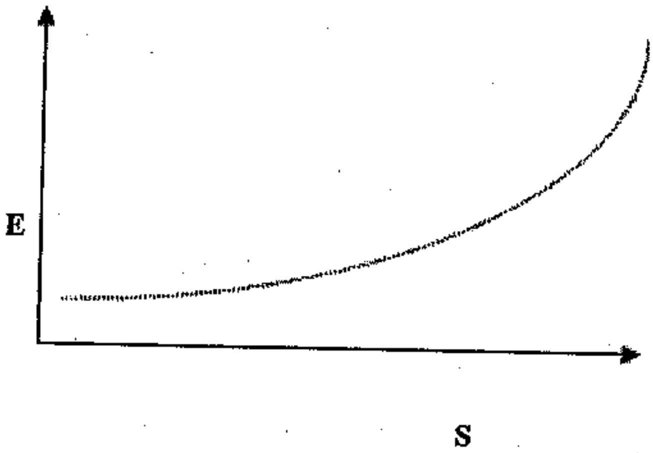


Figure 7.2 : Relationship between development effort & software size

The graph in figure 7.2 demonstrates that the amount of effort accelerates as size increases, i.e., the value c in the typical formula above is greater than 1.

7.7 PUTNAM MODEL

Putman's SLIM (Software LIfe cycle Management) is an automated 'macro estimation model' for software estimation. SLIM uses linear programming, statistical simulation, program evaluation and review techniques to derive a software cost estimate. SLIM enables a software cost estimator to perform the following functions:

- **Calibration**: Fine tuning the model to represent the local software development environment by interpreting a historical database of past projects.
- **Build:** an information model of the software system, collecting software characteristics, personal attributes, and computer attributes etc.
- **Software sizing**: SLIM uses an automated version of the lines of code (LOC) costing technique.

The SLIM model is based on the Putman's own analysis of the software life cycle in terms of the Raleigh distribution of project personnel level versus time. The algorithm used is:

$K = (\text{size} / (C * t^{4/3}))3$

Size is the lines of code, K is the total life-cycle effort (in working years), t is development time (in years).

C is the technology constant, combining the effect of using tools, languages, methodology and QA procedures etc. The values of the technology constant can vary from as little as 610 up to 57314.

Drawbacks of SLIM

- SLIM estimates are extremely sensitive to the technology factor.
- Not suitable for small projects.

Advantages of SLIM

- Uses linear programming to consider development constraints on both cost and effort.
- Requires fewer perameters to generate estimates.

7.8 COCOMO

The best known and most transparent cost model COCOMO (COnstructive COst MOdel) was developed by Boehm, derived from the analysis of 63 software projects. The original COCOMO model was a set of models; 3 development modes (organic, semi-detached, and embedded) and 3 levels (basic, intermediate, and advanced).

COCOMO model levels:

Basic - predicted software size (lines of code) is used to estimate development effort.

Intermediate - predicted software size (lines of code), plus a set of 15 subjectively assessed 'cost drivers' is used to estimate development effort.

Advanced - on top of the intermediate model, the advanced model allows phase-based cost driver adjustments and some adjustments at the module, component, and system level.

COCOMO development modes:

Organic - small relatively small, simple software projects in which small teams with good application experience work to a set of flexible requirements.

Embedded - the software project has tight software, hardware and operational constraints.

Semi-detached - an intermediate (in size and complexity) software project in which teams with mixed experience levels must meet a mix of rigid and less than rigid requirements.

COCOMO model:

The general formula of the basic COCOMO model is:

$$\mathbf{E = a(S)^b}$$

Where:

E represents effort in person-months,

S is the size of the software development in KLOC and,

a and **b** are values dependent on the development mode,

development mode:	organic	**a** = 2.4	**b** = 1.05
	semi-detached	**a** = 3.0	**b** = 1.12
	embedded	**a** = 3.6	**b** = 1.20

The intermediate and advanced COCOMO models incorporate 15 'cost drivers'. These 'drivers' multiply the effort derived for the basic COCOMO model. The importance of each driver is assessed and the corresponding value multiplied into the COCOMO equation, which becomes:

E = a(S)^{b} x product(cost drivers)

As an example of how the intermediate COCOMO model works, the following is a calculation of the estimated effort for a semi-detached project of 56 KLOC. The cost drivers are set as follows:

Product cost drivers (from the table) set	**high**	= 1.15 x 1.08 x 1.15= 1.43
Computer cost drivers (from the table) set	**nominal**	= 1.00
Personnel cost drivers (from the table) set	**low**	= 1.19 x 1.13 x 1.17 x 1.10 x .07 = 1.85
Project cost drivers (from the table) set	**high**	= 0.91 x 0.91 x 1.04 = 0.86

hence, **product(cost drivers)** = 1.43 x 1.00 x 1.85 x 0.86 = 2.28

for a semi-detached project of 56KLOC: **a** = 3.0 **b** = 1.12 **S** = 56

E = a(S)b x product(cost drivers)

E = 3.0 x (56)$^{1.12}$ x 2.28

E = 3.0 x 90.78 x 2.28

E = 620.94 person-months

COCOMO II

COnstructive COst MOdel II (COCOMO II) is a model that allows one to estimate the cost, effort, and schedule when planning a new software development activity. COCOMO II is the latest major extension to the original COCOMO model published in 1981. It consists of three submodels, each one offering increased fidelity the further along one is in the project planning and design process. Listed in increasing fidelity, these submodels are called the Applications Composition, Early Design, and Post-architecture models.

Assessment of COCOMO

Drawbacks

- It is hard to accurately estimate KLOC early on in the project, when most effort estimates are required.
- Extremely vulnerable to mis-classification of the development mode.
- Success depends largely on tuning the model to the needs of the organization, using historical data which is not always available.

Advantages

- COCOMO is transparent, you can see how it works unlike other models such as SLIM.

- Drivers are particularly helpful to the estimator to understand the impact of different factors that affect project costs.

7.9 INTRODUCTION TO SOFTWARE METRICS AND MEASURES

Software process and project metrics are quantitative measures that enable software engineers to gain insight into the efficiency of the software process and the projects conducted using the process framework. In software project management, we are primarily concerned with productivity and quality metrics. The four reasons for measuring software processes, products, and resources are:

- To characterize- an effort to gain an understanding " of processes, products, resources, and environments and to establish baselines for comparisons with future assessments
- To evaluate- determine status with respect to plans
- To predict- - gaining understandings of relationships among processes and products and building models of these relationships, and
- To improve- identifying roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance.

Measures, Metrics, and Indicators

The terms measure, measurement, and metrics are often used interchangeably, it is important to note the subtle difference between them.

- Measure - provides a quantitative indication of the size of some product or process attribute
- Measurement - is the act of obtaining a measure
- Metric - is a quantitative measure of the degree to which a system, component, or process possesses a given attribute

Process and Project Indicators

- Metrics should be collected so that process and product indicators can be ascertained
- Process indicators enable software project managers to: assess project status, track potential risks, detect problem area early, adjust workflow or tasks, and evaluate team ability to control product quality

7.10 PROCESS METRICS

Process metrics are collected across all projects and over long periods of time. Their intent is to provide as a set of process indicators that lead to long term software process improvements.

Process is one of the controllable factors in improving software quality and organizational performance. As shown in figure 7.3 process connects 3 important factors that influence software quality and organizational performance. The skill and motivation of people has been shown to be the single most influential factor in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e. software engineering methods and tools) that populates the process also has an impact.

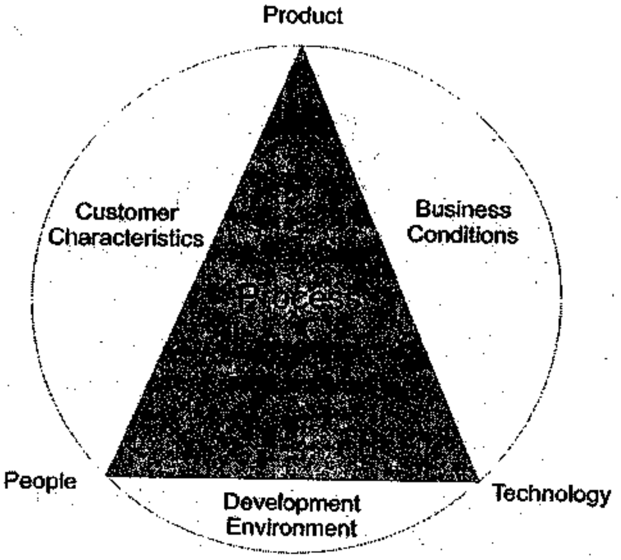


Figure 7.3: Determinants for software qality and organizational effectiveness

In addition, the process triangle exists within a circle of environmental conditions that include the development environment (CASE tools), business conditions (e.g. deadlines, business rules), and customer characteristics (e.g. ease of communication and collaboration).

To improve a process

- Measure specific attributes of process
- Develop meaningful metrics based on these attributes
- These metrics provide indicators that lead to strategy for improvement

Metrics can be derived from the following process outcomes/attributes

- Errors uncovered before software release,
- Defects delivered to & reported by end-user,
- Work products delivered
- Human efforts expended
- Calendar time expended
- Schedule conformance

There are private and public uses for different types of process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on a individual basis hence *Private process metrics* (e.g. defect rates by individual or module) are known only to the individual or team concerned.

Public metrics generally assimilate information that was private to individuals and teams. Project level defect rates, effort, calendar times, and related data are collected. *Public process metrics* enable organizations to make strategic changes to improve the software process. As software metric etiquette metrics should not be used to evaluate the performance of individuals.

As an organization becomes more comfortable with the collection and use of process metrics, the derivation of simple indicators gives way to a more rigorous approach called *statistical software process improvement* (SSPI). Statistical software process improvement helps an organization to discover where it is strong and where weak.

7.11 PROJECT METRICS

Unlike software process metrics that are used for strategic purposes, software project metrics are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project workflow and technical activities. Every project should measure its inputs (resources), outputs (deliverables), and results (effectiveness of deliverables). Project metrics are used to avoid development schedule delays, to mitigate potential risks, and to assess product quality on an on-going basis.

The objectives of project metric are

- To minimize the development schedule by making adjustments that can avoid delays and mitigate potential problems
- To assess product quality on an ongoing basis and when necessary, modify technical approach to improve quality
- To estimate effort

Project metrics enable a software project manager to

- Assess the status of an ongoing project,
- Track potential risks
- Uncover problem areas before they go "critical"
- Adjust work flow or tasks
- Evaluate the project team's ability to control quality of software work products.
- Metrics collected from past projects help to establish time and effort estimates for current software projects, Production rates (in terms of models created, review hours, function points, delivered source lines), Errors uncovered during each software engineering task

Software projects are assessed against the project metrics for improvement

7.12 SOFTWARE MEASUREMENT

Software Measurement is a quantifiable dimension, attribute, or amount of any aspect of a software program, product or process. Software measures can be categorized as direct measures & indirect measures.

- *Direct measures* of software engineering process include cost and effort. Direct measures of the product include lines of code (LOC), execution speed, memory size, defects per reporting time period.
- *Indirect measures* examine the quality of the software product itself (e.g. functionality, complexity, efficiency, reliability, maintainability).

Project metrics can be consolidated to create process metrics that are public to the software organization as a whole. Different individual measures are normalized to create software metrics that enable comparison to broader organizational averages. Both size and function oriented metrics are normalized.

Size-Oriented Metrics

The size oriented metrics are direct measures of a software & the process by which they are developed. They are derived by normalizing (dividing) any direct measure (e.g. defects or human effort) associated with the product or project by LOC. Size oriented metrics are widely used but their validity and applicability is debated.

Example set of Size-oriented metrics include

- Errors per KLOC (thousand lines of code)
- Defects per KLOC
- $ per LOC
- Pages of Documentation per KLOC
- Errors per person-month
- LOC per person-month
- $ per Page of documentation
- Though these metrics widely used they are not universally accepted as the best way to measure a software process
- Controversy lies in using LOC as a key measure
- Proponents of this method claim
    - LOC that is easy to count
    - Many existing estimation models use LOC
    - Large literature & data based on LOC exists
- But opponents argue that
    - LOC measures are programming language dependent
    - When considering productivity, LOC criteria penalizes well designed short programs
    - It Can not accommodate non procedural languages
    - Planner must estimate LOC long before analysis and design

Function-Oriented Metrics

Function oriented metrics are indirect measures of software that focus on frinctionality and utility. These are based oa productivity measurement approach called the function point method. Function points are derived from countable measures and assessments of software complexity. Five characteristies are used to calculate function points. These values are number of user inputs, number of user outputs, number of user inquiries, number of files and number of external interfaces.

Function points are computed from direct measures of the information domain of a business software application and assessment of its complexity. Once computed function points are used like LOC to normalize measures for software productivity, quality, and other attributes. Feature points and 3D function points provide a means of extending the function point concept to allow its use with real-time and other engineering applications. The relationship of LOC and function points depends on the language used to implement the software.

Object-Oriented Metrics

Object oriented metrics are an integral part of object technology and describe aspects of object oriented programming.

They are used for object-oriented projects adn require the use of classes. The set of metrics for OO projects include,

- Number of scenario scripts
  - · Scenario scripts are detailed sequence of steps about user and application interaction
  - · Scenario scripts are directly correlated to application size and no. of test cases
- Number of key classes
  - · Key classes are independent components
  - · They Indicate amount of development effort and potential reuse
- Number of support classes
  - · These indicate amount of development effort and potential reuse
- Average number of support classes per key class
- Number of sub systems (aggregation of classes)
  - · If identified, it is easier to lay out the schedule

Use-Case oriented Metrics

Use cases capture the functional requirements of a system.They describe user visible functions and features and are defined early in software process and can be used as normalization measure before significant activities are initiated use cases independent of programming languageo Number of use cases is directly proportional to size of application in LOC & number of test cases that will be designed. There is no standard size of a use case as they are created at different levels of abstraction. For this reason, it is a suspect as a normalization measure.

Web Engineering Project Metrics

Web engineering promotes systematic, disciplined & quantifiable approach towards uscessful development of quality web based applications.

- Web Engineering project metrics are used for web application projects
- The set of metrics for web projects include
- Number of static web pages
  - · Most common Web Application feature
  - · Static pages represent low relative complexity and require less effort to construct.
  - · Indicator of overall size and effort.
- Number of dynamic web pages
  - · Essential in application like e-commerce, search engines, financial application etc.
  - · Relatively higher complexity and effort in comparison to static pages
  - · They indicate overall size and effort.
- Number of internal page links
  - · As the number of internal links increases, the effort expanded on navigational design and construction also increases.
- Number of persistent data objects

· E.g. data base or data file. As these data objects grow, the complexity of web application also grows.

- Number of external system interfaced

  · Web application must often interface with "backroom" business applications.

  · As the requirement of interfacing grows, system complexity also grows and development effort increases.

- Number of Static content objects

  · Static content objects encompases static text based, graphics, video, animation, audio information incorporated within web application.

- Number of dynamic content objects

  · Dynamic content objects are generated based on end user actions and encompasses internally generated text based, graphics, video, animation and audio information

- Number of executable functions

  · An executable function(e.g. script or sapplet) provides some computational service to the end user. As the number of executable function increases, modeling and construction effort also increase.

7.13 SUMMARY

- Size related and function related measures for Productivity estimates.
- A software cost estimate process is the set of techniques and procedures that an organization uses to arrive at an estimate.
- Algorithmic cost Modelling costs are analyzed using mathematical formulae and linking costs or inputs with metrics to produce an estimated output. The formulae used in a formal model arise from the analysis of historical data.
- The best known cost estimation models are Boehm's COCOMO, Putman's SLIM, and Albrecht's' function points.
- SLIM uses linear programming, statistical simulation, program evaluation and review techniques to derive a software cost estimate.
- In COCOMO basic model predicted software size (lines of code) is used to estimate development effort
- Process metrics are collected across all projects and over long periods of time. Their intent is to provide a set of process indicators that lead to long term software process improvements.
- Project metrics and the indicators derived from them are used by a project manager and a software team to adapt project workflow and technical activities. Every project should measure its inputs (resources), outputs (deliverables), and results (effectiveness of deliverables).

7.14 UNIT END QUESTIONS

1 What are the important factors that effect cost estimation?

2 What is the role of size of the project in cost estimating?

3 Discuss the various method used for cost estimation in brief.

4 Describe software metric. What is the difference between process and product metric?

5 Explain the metric used for web based application. How they are different from other project metrics?

6 What do you mean by software cost estimation? How it is estimated by COCOMO model? Explain.

7.15 FURTHER READINGS

1. Roger S. Pressman, Software Engineering: A Practitioner's Approach, Sixth edition, McGraw Hill Publication
2. Ian Sommerville, Software Engineering, Sixth Edition, Pearson education
3. Pankaj Jalote, An Integrated Approach to Software Engineering, Narosa Publishing House
4. http://en.wikipedia.org/wiki/COCOMO
5. http://en.wikipedia.org/wiki/Cost_estimation_models
6. http://ectc.u-net.com/cost/models.htm

UNIT – VIII

SOFTWARE REUSE

Structure of the Unit

8.0 OBJECTIVES

After completing this unit the students will learn :

- The benefits of software reuse
- Different ways to implement software reuse
- How reusable concepts can be represented as patterns or embedded in program generators
- Development of software product lines.

8.1 INTRODUCTION

The purpose of software reuse is to reduce cost, time, effort, and risk and to increase productivity, quality, performance, and interoperability.

Software reuse is the process of implementing or updating software systems using existing software assets. Software assets, or components, include all software products, from requirements and proposals, to specifications and designs, to user manuals and test suites. Anything that is produced from a software development effort can potentially be reused.

Software reuse does not just indicate the reuse of application code. It is possible to reuse specification and designs. The potential gains from reusing abstract product of development process such as specifications may be greater than those from reusing code components.

In most engineering disciplines the developed process is based on components reuse. Software system design usually consider that all components are to be designed specially for the system being developed. There is no common base apart from libraries such as windows system libraries of reusable software components. By applying widespread and systematic software reuse, demands for lower software design and maintainence costs along with increased quality can be met.

8.2 SOFTWARE COMPONENTS REUSE

Generalized software components are not designed for one system but are tried and tested in a number of different environments. Design and implementation faults are discovered and removed so that reusable component contains few errors. It is impossible to achieve absolute reliability specification but reusable components may have an associated quality explanation. This permits users to integrate them with confidence in their systems.

The reuse of software can be divided at a number of levels:

1) Application system reuse

It is possible to reuse the whole application system. The major problem here is to ensure that the software is portable. It should run on a variety of different platforms.

2) Sub-system reuse

It is possible to reuse major sub-systems of an application.

3) Module or object reuse

It is possible to reuse components of a system representing a collection of function.

4) Function reuse

It is possible to reuse software components, which implement a single function such as a mathematical function.

Application system reuse is widely used in software companies to implement their systems across a range of machines. Function reuse is widely used in standard libraries of reusable function such as graphics and mathematical libraries. Sub-system and module reuse are less usable.

Four aspects of software reuse are

1) Software development with reuse

Software development with reuse is an approach which tries to maximize the reuse of existing software components. Benefit of this approach is that overall development costs of the software are decreased. Cost reduction is only one potential benefit of software reuse.

Systematic reuse in development offers further advantages:

i) System reliability is increased

Reused components in working systems are expected to be more reliable than new components. These components have been tested in variety of operational systems environment and have therefore been exposed to realistic operating conditions.

ii) Overall process risk is reduced

If we use a function which already exists, there is less uncertainty in the cost of reusing that component than in the costs of development. For project management this is an important factor as it decreases uncertainty in project cost elimination, specifically in case of reuse of relatively large components such as sub-systems.

iii) Effective use can made of specialists

Application specialists doing the same work on different project environments can develop reusable components which encapsulate their knowledge.

iv) Organizational standards can be embodied in reusable components

Some standards such as user interface standard , organization standards design standars etc. which can be implemented as a set of standard components can be reused.

v) Software development time can be reduced

It is necessary to bring a system to market as early as possible. The time factor is more important than even the overall development costs. Reusing software components speeds up system production because both development and validation time should be reduced.

2) Software development for reuse

Component reuse may involve making different types of changes.

i) Name generalization: The component name should be modified so that they are neutral rather than a direct reflection of some specific application entity.

ii) Operation generalization: This involves adding operations to a component or removing operations which are very specific to some application domain.

iii)Exception generalization: This involves checking each component to see which exceptions it might throw and including these exceptions in the component interface.

3) Generator based reuse

An alternative way to reuse components is the generator view. In this approach reusable knowledge is confined in a program generator system which can be programmed in a domain oriented language. Program generators involve the reuse of standard patterns and algorithms. These are embedded in the generator and parameterised by user commands. A program is then automatically generated. Progam generators automatically generate programs based on program logic tools such as flowharts.

Generator-based reuse is possible when domain abstractions and their mapping to executable code can be identified.

A domain specific language is used to compose and control these abstractions. High-level language compliers are most widely used 'program generators', where the reusable components are fragments of object code corresponding to high level language construct. The reused elements are abstractions related to programming language statements. When a domain specific notation is used to describe the application, larger domain abstraction can be reused.

Types of program generator

Program generators have been categorized onthe basis of the domain that they develop applications for and tools for software development.Some of the types of program generators are:

- Application generators for business data processing
- Parser and lexical analyser generators for language processing
- Code generators in CASE tools.
- Generator-based reuse is very cost-effective but its applicability is limited to a
- relatively small number of application domains. It is easier for end-users to develop programs using generators compared to other component-based approaches to reuse.

Aspect-oriented development

- Aspect-oriented development addresses a major software engineering problem - the separation of concerns.
- Concerns are often not simply associated with application functionality but are cross-cutting - e.g. all components may monitor their own operation, all components may have to maintain security, etc.
- Cross-cutting concerns are implemented as aspects and are dynamically woven into a program. The concern code is reuse and the new system is generated by the aspect weaver.

8.3 DESIGN PATTERNS

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

Not all software patterns are design patterns. Algorithms are not thought of as design patterns, since they solve computational problems rather than software design problems. Architectural patterns are larger in scope, usually describing an overall pattern followed by an entire program. Programming paradigms describe a style which can be the basis for an entire programming language.

The documentation for a design pattern describes the context in which the pattern is used, the forces within the context that the pattern seeks to resolve, and the suggested solution. There is no single, standard format for documenting design patterns. Rather, a variety of different formats have been used by different pattern authors.

A design pattern usually contains the following sections:

- Pattern Name and Classification: A descriptive and unique name that helps in identifying and referring to the pattern.
- Intent: A description of the goal behind the pattern and the reason for using it.
- Also Known As: Other names for the pattern.
- Motivation (Forces): A scenario consisting of a problem and a context in which this pattern can be used.
- Applicability: Situations in which this pattern is usable; the context for the pattern.
- Structure: A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
- Participants: A listing of the classes and objects used in the pattern and their roles in the design.
- Collaboration: A description of how classes and objects used in the pattern interact with each other.
- Consequences: A description of the results, side effects, and trade offs caused by using the pattern.
- Implementation: A description of an implementation of the pattern; the solution part of the pattern.
- Sample Code: An illustration of how the pattern can be used in a programming language
- Known Uses: Examples of real usages of the pattern.

Related Patterns: Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

Software product line engineering aims to reduce development time, effort, cost, and complexity by taking advantage of the commonality within a portfolio of similar products. The effectiveness of a software product line approach directly depends on how well feature variability within the portfolio is implemented and managed throughout the development lifecycle, from early analysis through maintenance and evolution.

8.4 APPLICATION FRAMEWORKS

Object oriented development suggests that objects are the most appropriate abstraction for reuse. However, experience has shown that objects are often too fine grained and too specialized to a particular application. It has become clear that object oriented reuse is best supported in an object oriented development process through larger-grain abstraction called framework.

A framework is a subsystem design made up of a collection of abstract and concrete classes and the interface between them. Particular details of the application sub system are implemented by adding components to fill in pents of the design and by instantiating the abstract classes in the framework. Applications are normally constructed by integrating a number of framework.

There are three classes of framework.

1. **System infrastructure framework** : These frameworks support the development of system infrastructure such as communications, user interfaces and compilers.
2. **Middleware integration framework** : These consist of a set of standard and associated object classes that support component communication and information exchange.
3. **Enterprise application framework** : These are concerned with specific application domain such as telecommunication or financial systems.

Frameworks are generic and are extended to create a more specific application or sub-system. Extending a framework involves

- Adding concrete classes that inherit operations from abstract classes in the framework
- Adding methods that are called in response to events that are recognized by the framework.

Problem with frameworks is their complexity which means that it takes a long time to use them effectively.

8.5 APPLICATION SYSTEM REUSE

Application system reuse involves reusing entire application systems either by configuring a system for a specific environment (COTS) or by integrating two or more systems to create a new application (Developing application families).

COTS product integration

COTS stands for Commercial Off-The-Shelf Systems. COTS systems are usually complete application systems that offer an Application Programming Interface. Building large systems by integrating COTS systems is a viable development. Strategy for some types of system such as E-commerce systems. The key benefit is faster application development and, usually, lower development costs.

Software Product Line

Software product lines or application families are applications with generic functionality that can be adapted and configured for use in a specific context. Each specific application is specialized in some way. The common core of application family is reused each time a new application is required. The new development may involve specific component configuration, implementing additional components and adapting some of the components to meet new demands.

Various types of specialization of the application product line may be developed.

Platform specialization : Various of the application are developed for different platforms .

Environment Specialization : Versions of the application are created to handle particular operating environment and peripherals devices.

Functional Specialization : Versions of the application are created for specific customer who have different requirements .

Process Specialization : The system is adapted to cope with specific business process.

8.6 SOFTWARE EVOLUTION

Software evolution is the process by which programs change shape, adapt to the marketplace and inherit characteristics from preexisting programs. Software evolution is inevitable since new requirements emerge when the software is used, the application domain modifies, errors must be repaired, new functionality must be accommodated, performance and/or reliability may have to be improved. The updations to the software bring out newer and improved versions of it, and thus it evolves.

Software Evolution can be categorized as Software Maintenance and Software Reengineering. Maintenance and re-engineering may be applied separately or together.

Software Maintenance

Updations to a software may be carried out when there is a known requirement for change, under four categories, these include,

- Corrective maintenance: Reactive modification of a software product performed after delivery to correct discovered problems;
- Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment;
- Perfective maintenance: Modification of a software product after delivery to improve performance or maintainability;
- Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

Software Re-engineering

Software Reengineering is the process of reorganising and modifying existing software systems to make them more maintainable. When a Software is Reengineered, no new functionality is added to the system but it is restructured and reorganised to facilitate future changes. The advantages of re-engineering are reduced risk and reduced cost. There is a high risk in new software development. There may be development problems, staffing problems and specification problems and also the cost of re-engineering is often significantly less than the costs of developing new software.

Prof. Meir M. Lehman, and his colleagues have identified a set of behaviours in the evolution of proprietary software. These behaviours are known as Lehman's Laws. There are eight laws and these are:

Continuing Change, Increasing Complexity, Large Program Evolution, Invariant Work-Rate, Conservation of Familiarity, Continuing Growth, Declining Quality, Feedback System. Table 8.1 describes each of these rules.

Table 8.1 Lehman's Laws for Software Evolution

I	Continuing Change	Systems must be continually adapted else they become progressively less satisfactory in use.
II	Increasing Complexity	As system is evolved its complexity increases unless work is done to maintain or reduce it.
III	Self Regulation	Global system evolution processes are self-regulating. [System attributes such as size, time between releases and the number of reported errors are approximately invariant for each system release.]
IV	Conservation of Organisational Stability	Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving system tends to remain constant over product lifetime.
V	Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.
VI	Continuing growth	The functionality offered by systems must continually increase to maintain user satisfaction.
VII	Declining quality	The quality of systems will appear to be declining unless they are adapted to changes in their operational environment.
VIII	Feedback system	Evolutionary processes incorporate multi-agent, multi-loop feedback systems and must be treated as feedback systems in order to achieve significant product improvement.

These laws however are believed to apply mainly to monolithic, proprietary software and could be challenged for application to other software types.

Giving the increasing dependence on software at all levels of society and economy, the successful evolution of software is becoming increasingly critical.

8.7 SUMMARY

Design with software reuse involves designing software around good design and existing components.

Software reuse advantages are lower costs, faster software development and lower risks.

Design patterns are high-level abstractions that document successful design solutions.

Program generators are an alternative approach to concept reuse where the reusable concepts are embedded in a generator system. The designer specifies the abstractions required using a domain-specific language, and an executable program is generated.

Commercial off the self software (COTS) are ready made & available for sale, lease or license to general public. They may used as alternatives to inhouse developments.

Software product lines are related application that are developed from one or more base applications. A generic system is adapted and specialized to meet specific requirements for functionality, target platform or operational configuration.

8.8 UNIT-END QUESTIONS

1. What do you mean by software reuse? List various level of software reuse.

2. Why are patterns an effective form of design reuse? What are the disadvantages of this approach to reuse?

3. What are the technical and non technical factors in software reuse?

8.9 FURTHER READINGS

1. Software Engineering, Ian Sommerville, Pearson Education, 2004.

UNIT – IX

VERIFICATION AND VALIDATION

Structure of the Unit

9.0 OBJECTIVES

After completing this unit students will be able to understand :

- The difference between software verification and software validation.
- Program inspection methods of discovering defects in program.
- Automated static analysis and its use in verification and validation.
- How static verification is used in the clean room development process.

9.1 INTRODUCTION

Software Verification and Validation (V&V) is the process of ensuring that software being developed or changed will satisfy functional and other requirements (validation) and each step in the process of building the software yields the right products (verification). The differences between verification and validation are unimportant except to the theorists. Practitioners use the term to refer to all of the activities that are aimed at making sure the software will function as required.

Verification & Validation is intended to be a systematic and technical evaluation of software and associated products of the development and maintenance processes. Reviews and tests are done at the end of each phase of the development process to ensure software requirements are complete and testable and that design, code, documentation, and data satisfy those requirements.

9.2 PLANNING VERIFICATION AND VALIDATION ACTIVITIES

The two major Verification and Validation activities are:

reviews, including inspections and walkthroughs, and testing.

1. Reviews, Inspections, and Walkthroughs

Reviews are conducted during and at the end of each phase of the life cycle to determine whether established requirements, design concepts, and specifications have been met. Reviews consist of the presentation of material to a review board or panel. Reviews are most effective when conducted by personnel who have not been directly involved in the development of the software being reviewed.

Informal reviews are conducted on an as-needed basis. The developer chooses a review panel and provides and/or presents the material to be reviewed. The material may be as informal as a computer listing or hand-written documentation.

Formal reviews are conducted at the end of each life cycle phase. The acquirer of the software appoints the formal review panel or board, who may make or affect a go/no-go decision to proceed to the next step of the life cycle.

Formal reviews include the Software Requirements Review, the Software Preliminary Design Review, the Software Critical Design Review, and the Software Test Readiness Review.

An inspection or walkthrough is a detailed examination of a product on a step-by-step or line-of-code by line-of-code basis. The purpose of conducting inspections and walkthroughs is to find errors. The group that does an inspection or walkthrough is composed of peers from development, test, and quality assurance.

2. Testing

Testing is the operation of the software with real or simulated inputs to demonstrate that a product satisfies its requirements and, if it does not, to identify the specific differences between expected and actual results. There are varied levels of software tests, ranging from unit or element testing through integration testing and performance testing, up to software system and acceptance tests.

a. Informal Testing

Informal tests are done by the developer to measure the development progress. 'Informal' in this case does not mean that the tests are done in a casual manner. It just specifies that the acquirer of the software is not formally involved, that witnessing of the testing is not required, and that the prime purpose of the tests is to find errors. Unit, component, and subsystem integration tests are usually informal tests.

Informal testing may be requirements-driven or design- driven. Requirements-driven or black box testing is done by selecting the input data and other parameters based on the software requirements and observing the outputs and reactions of the software. Black box testing can be done at any level of integration. In addition to testing for satisfaction of requirements, some of the objectives of requirements-driven testing are to ascertain:

- Computational correctness.
- Proper handling of boundary conditions,
- Including extreme inputs and conditions that cause extreme outputs.
- State transitioning as expected.
- Proper behavior under stress or high load.
- Adequate error detection, handling, and recovery.

Design-driven or white box testing is the process where the tester examines the internal workings of code. Design- driven testing is done by selecting the input data and other parameters based on the internal logic paths that are to be checked.

The goals of design-driven testing include ascertaining correctness of:

All paths through the code. For most software products, this can be feasibly done only at the unit test level.

- Bit-by-bit functioning of interfaces.
- Size and timing of critical elements of code.

b. Formal Tests

Formal testing demonstrates that the software is ready for its intended use. A formal test should include an acquirer- approved test plan and procedures, quality assurance witnesses, a record of all discrepancies, and a test report. Formal testing is always requirements-driven, and its purpose is to demonstrate that the software meets its requirements.

Each software development project should have at least one formal test, the acceptance test that concludes the development activities and demonstrates that the software is ready for operations.

In addition to the final acceptance test, other formal testing may be done on a project. For example, if the software is to be developed and delivered in increments or builds, there may be incremental acceptance tests. As a practical matter, any contractually required test is usually considered a formal test; others are "informal."

After acceptance of a software product, all changes to the product should be accepted as a result of a formal test. Post acceptance testing should include regression testing. Regression testing involves rerunning previously used acceptance tests to ensure that the change did not disturb functions that have previously been accepted.

c. Verification and Validation

During the Software Acquisition Life Cycle the V&V Plan should cover all V&V activities to be performed during all phases of the life cycle. The verification & Validation activities performed at each of the development cycle are discussed below:

i. Software Concept and Initiation Phase

The major V&V activity during this phase is to develop a concept of how the system is to be reviewed and tested.

Simple projects may compress the life cycle steps, if so, the reviews may have to be compressed. Test concepts may involve simple generation of test cases by a user representative or may require the development of elaborate simulators and test data generators. Without an adequate V&V concept and plan, the cost, schedule, and complexity of the project may be poorly estimated due to the lack of adequate test capabilities and data.

ii. Software Requirements Phase

V&V activities during this phase should include: Analyzing software requirements to determine if they are consistent with, and within the scope of, system requirements, assuring that the requirements are testable and capable of being satisfied, creating a preliminary version of the Acceptance Test Plan, including a verification matrix which relates requirements to the tests used to demonstrate that requirements are satisfied. Beginning development, if needed, of test beds and test data generators. and conducting the phase-ending Software Requirements Review (SRR).

iii. Software Architectural (Preliminary) Design Phase

V&V activities during this phase should include, Updating the preliminary version of the Acceptance Test Plan and the verification matrix, conducting informal reviews and walkthroughs or inspections of the preliminary software and data base designs, the phase-ending Preliminary Design Review (PDR) at which the allocation of requirements to the software architecture is reviewed and approved.

iv. Software Detailed Design Phase

V&V activities during this phase should include: Completing the Acceptance Test Plan and the verification matrix, including test specifications and unit test plans, conducting informal reviews and walkthroughs or inspections of the detailed software and data base designs, the Critical Design Review (CDR) which completes the software detailed design phase.

v. Software Implementation Phase

V&V activities during this phase should include:

Code inspections and/or walkthroughs, Unit testing software and data structures, Locating, correcting, and retesting errors and Development of detailed test procedures for the next two phases.

vi. Software Integration and Test Phase

This phase is a major V&V effort, where the tested units from the previous phase are integrated into subsystems and then the final system. Activities during this phase should include,

Conducting tests per test procedures, Documenting test performance, test completion, and conformance of test results versus expected results, Providing a test report that includes a summary of nonconformance found during testing, Locating, recording, correcting, and retesting nonconformance and The Test Readiness Review (TRR), confirming the product's readiness for acceptance testing.

vii. Software Acceptance and Delivery Phase

V&V activities during this phase should include, Demonstrating that the developed system meets its functional, performance, and interface requirements by conducting test analysis & inspection, Locating, correcting, and retesting nonconformances and The phase-ending Acceptance Review (AR).

viii. Software Sustaining Engineering and Operations Phase

Any V&V activities conducted during the prior seven phases are conducted during this phase as they pertain to the revision or update of the software.

d. Independent Verification and Validation (IV&V)

It is a process whereby the products of the software development life cycle phases are independently reviewed, verified, and validated by an organization that is neither the developer nor the acquirer of the software. The IV&V agent should have no stake in the success or failure of the software. The IV&V agent's only interest should be to make sure that the software is thoroughly tested against its complete set of requirements.

The IV&V activities duplicate the V&V activities step-by- step during the life cycle, with the exception that the IV&V agent does no informal testing. If there is an IV&V agent, the formal acceptance testing may be done only once, by the IV&V agent. In this case, the developer will do a formal demonstration that the software is ready for formal acceptance.

9.3 SOFTWARE INSPECTIONS

Perhaps more tools have been developed to aid the V&V of software (especially testing) than any other software activity. The tools available include code tracers, special purpose memory dumpers and formatters, data generators, simulations, and emulations. Some tools are essential for testing any significant set of software, and, if they have to be developed, may turn out to be a significant cost and schedule driver.

An especially useful technique for finding errors is the formal inspection. Formal inspections were developed by Michael Fagan of IBM. Like walkthroughs, inspections involve the line-by-line evaluation of the product being reviewed. Inspections, however, are significantly different from walkthroughs and are significantly more effective.

Inspections are done by a team, each member of which has a specific role. The team is led by a moderator, who is formally trained in the inspection process. The team includes a reader, who leads the team through the item; one or more reviewers, who look for faults in the item; a recorder, who notes the faults; and the author, who helps explain the item being inspected.

This formal, highly structured inspection process has been extremely effective in finding and eliminating errors. It can be applied to any product of the software development process, including documents, design, and code. One of its important side benefits has been the direct feedback to the developer/author, and the significant improvement in quality that results.

9.4 VERIFICATION AND FORMAL METHODS

Formal method of software development is based on mathematical representation of the software, usually as a formal specification. These formal methods are mainly concerned with a mathematical analysis of the specification with transforming the specification to a more detailed, semantically equivalent representation or with formally verifying that one representation of the system is semantically equivalent to another representation.

Formal method may be used at different stages in the validation and verification process.

1. A formal specification of the system may be developed and mathematically analyzed for inconsistency. This technique is effective in discovering specification errors.

2. A formal verification using mathematical arguments that the code of a software is consistent with its specification. This requires a formal specification and is effective in discovering programming and some design errors. A transformational development process where a formal specification is transformed through a series of more detailed representation or a cleanroom process may be used to support the formal verification process.

9.4.1 Clean Room Software Development

The Cleanroom process is a theory-based, team-oriented process for the development and certification of high-reliability software systems under statistical quality control. Its principal objective is to develop software that exhibits zero failures in use. For this purpose the life cycle is different from conventional software development techniques. The approach combines mathematical-based methods of software specification, design and correctness verification with statistical, usage-based testing to certify software fitness for use. Therefore the goals in this method are to reduce the failures found during testing by enabling good and correct designs that avoid rework. Most designs pass through detailed specifications and modeling which are evaluated and proved for correctness using formal methods.

The clean room approach to software development is based on five key strategies:

Requirement Analysis: To define requirements for the software product (including function, usage, environment, and performance) as well as to obtain agreement with the customer on the requirements as the basis for function and usage specification. Requirements analysis may identify opportunities to simplify the customer's initial product concept and to reveal requirements that the customer has not addressed.

Function Specification: To make sure the requirement behavior of the software in all possible circumstances of use is defined and documented. The function specification is complete, consistent, correct, and traceable to the software requirements. The customer agrees with the function specification as the basis for software development and certification. This process is to express the requirements in a mathematically precise, complete, and consistent form.

Usage Specification: To identify and classify software users, usage scenarios, and environments of use, to establish and analyze the highest level structure and probability distribution for software usage models, and to obtain agreement with the customer on the specified usage as the basis for software certification.

Architecture Specification: The purpose is to define the 3 key dimensions of architecture: Conceptual architecture, module architecture and execution architecture. The Cleanroom aspect of architecture specification is in decomposition of the history-based black box Function Specification into state-based state box and procedure-based clear box descriptions. It is the beginning of a referentially transparent decomposition of the function specification into a box structure hierarchy, and may be used during increment development.

Increment Planning: To allocate customer requirements defined in the Function specification to a series of software increments that satisfy the Software Architecture, and to define schedule and resource allocations for increment development and certification. In the incremental process, a software system grows from initial to final form through a series of increments that implement user function, execute in the system environment, and accumulate into the final system.

Cleanroom Development Processes

There are three terms involved when the cleanroom process is used for large system development, these are:

Software Reengineering: The purpose is to prepare reused software for incorporation into the software product. The functional semantics and interface syntax of the reused software must be understood and documented, and if incomplete, can be recovered through function abstraction and correctness verification. Also, the certification goals for the project must be achieved by determining the fitness for use of the reused software through usage models and statistical testing.

Increment Design: The purpose is to design and code a software increment that conforms to Cleanroom design principles. Increments are designed and implemented as usage hierarchies through box structure decomposition, and are expressed in procedure-based clear box forms that can introduce new black boxes for further decomposition. The design is performed in such a way that it is provably correct using mathematical models. Treating a program as a mathematical function can do this.

Correctness Verification: The purpose is to verify the correctness of a software increment using mathematical based techniques. Black box specifications are verified to be complete, consistent, and correct. State box specifications are verified with respect to black box specifications, and clear box procedures are verified with respect to state box specifications. A set of correctness questions is asked during functional verification. Correctness is established by group consensus and/or by formal proof techniques. Any part of the work changed after verification, must be reverified.

9.5 SUMMARY

- Verification And validation are not the same thing. Verification shows that a program meets its specification, and validation shows that the program does what the user requires.
- Test plan should include details of the items to be tested, the testing schedule, the procedure for managing the testing process, the hardware and software requirements, and any testing problems that may be arise.
- Program inspections are effective in finding program errors. The aim of an inspection is to search faults. A fault check list should derive the inspection process.
- Verification techniques involv examination and analysis of the program to find the errors.
- In a program inspection a team checks the code.
- Cleanroom development relies on static technique for program verification and statistical testing for system reliability certification, it has been successful in producing systems that have a high level of reliability.

9.6 UNIT END QUESTIONS

1. Define validation and verification (V&V)
2. What are the two main objectives of the V&V?
3. Name two types of verification.
4. What is the goal of program testing?
5. What are two types of testing?
6. Explain the difference between testing and debugging.
7. What is the software test plan?
8. What is 'inspection'?
9. What is a static analyser?
10. What are the three stages of static analysis?

9.7 FURTHER READINGS

1. Software Engineering: A Practioner's Approach, Roger S. Pressman, McGrawHill
2. Software Enginering, Ian Someville, Addison Wesley
3. http://khambatti.com/mujtaba/Article & Papers/ Cleanroom Software Development.pdf

UNIT - X

SOFTWARE TESTING

Structure of the Unit

10.0 OBJECTIVES

After reading this unit, students should be able to appreciate the following:

- Testing Fundamentals
- Test Cases and Test Criteria
- Test Case Design
- Functional Testing
- Structural Testing
- Test Plan Activities During Testing
- Strategic Issues in Testing
- Unit testing
- Integration Testing

10.1 INTRODUCTION

Quality of design refers to the characteristics that designers specify for an item. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-grade materials are used, tighter tolerances and greater levels of performance are specified, the design quality of a product increases, if the product is manufactured according to specifications.

Testing is the activity through which this compliance to specifications & quality is ensured. Software testing aims at eveluating the attributes and capbality of the programs and determining that it meets its required resluts.

10.2 TESTING FUNDAMENTALS

Testing is the process of executing the software with the intent of finding errors. The system may be tested in state and dynamic modes.

Static analysis is used to investigate the structural properties of source code. Dynamic test cases are used to investigate the behavior of the source code by executing the program on the test data. As The term "Program Unit" denotes a routine or collection of routines, implemented by an individual programmer. In a well-designed system, a program unit is a stand-alone program or a function unit of a large system.

During the testing process, only failures are observed, by which the presence of faults is deduced. The actual faults are identified by separate activities, commonly referred to as "debugging." In other words, for identifying faults, after testing has revealed the presence of faults, the expensive task of debugging has to be performed. Testing hence, is an expensive method for identification of faults, compared to methods that directly observe faults.

Test Oracles

To test any program, we need to have a description of its expected behavior and a method of determining whether the observed behavior conforms to the expected behavior. For this, we need a test oracle.

A test oracle is a mechanism; different from the program itself, that can be used to check the correctness of the output of the program for the test cases. Conceptually, we can consider testing a process in which the test cases are given to the test oracle and the program under testing. The output of the two is, then, compared to determine if the program behaved correctly for the test cases. This is shown in Figure 10.1.

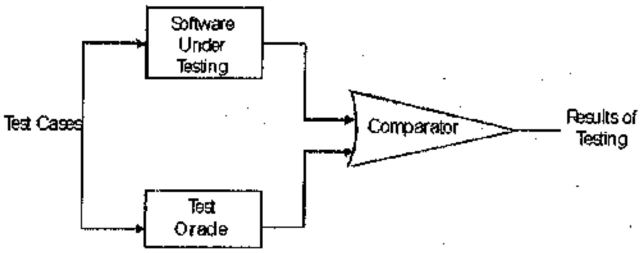


Figure 10.1: Testing and Test Oracles

Test oracles are necessary for testing. Ideally, we would like an automated oracle, which always gives a correct answer. However, often, the oracles are human beings, who mostly compute by hand what the output of the program should be. Often, it is extremely difficult to determine whether the behavior conforms to the expected behavior, our "human oracle" may make mistakes. As a result, when there is a discrepancy between the results of the program and the oracle, we have to verify the results produced by the oracle, before declaring that there is a fault in the program. This is one of the reasons that testing is so cumbersome and expensive.

The human oracles, generally, use the specifications of the program to decide what the "correct" behavior of the program should be. To help the oracle determine the correct behavior, it is important that the behavior of the system or component be unambiguously specified and that the specification itself is error-free. In other words, the specifications should actually specify the true and correct system behavior. These conditions are hard to satisfy. After all, it is the activity of some earlier phase that determines these specifications, and these activities might be error-prone. Hence, the specifications themselves may contain errors, be imprecise, or contain ambiguities. Such shortcomings in the specifications are the major cause of situations where one party claims that a particular condition is not a failure while the other claims it is. However, there is no easy solution to this problem. Testing does require some specifications against which the given system is tested.

There are some systems where oracles are automatically generated from specifications of programs or modules. With such oracles, we are assured that the output of the oracle is consistent with the specifications. However, even this approach does not solve all our problems, because of the possibility of errors in the specifications. Consequently, an oracle generated from the specifications will only produce correct results if the specifications are correct, and it will not be dependable in the case of specification errors. Furthermore, such systems that generate oracles from specifications are likely to require formal specifications, which are frequently not generated during design.

10.3 TESTING TECHNIQUES AND STRATEGIES

Generally, parts of the program are tested before testing the entire program. Besides, partitioning the problem of testing, another reason for testing parts separately is that if a test case detects an error in a large program, it will be extremely difficult to pinpoint the source of the error. That is, if a huge program does not work, determining which module has errors can be a formidable task. Furthermore, it will be extremely difficult to construct test cases so that different modules are executed in a sufficient number of different conditions so that we can feel fairly confident about them. In many cases, it is even difficult to construct test cases so that all the modules will be executed. This increases the chances of a module's errors going undetected. Hence, it is clear that for a large system, one should first test different parts of the system independently, before testing the entire system.

In incremental testing, some parts of the system are first tested independently. Then, these parts are combined to form a (sub) system, which is then tested independently. This combination can be done in two ways: either only the modules that have been tested independently are combined or some new untested modules are combined with tested modules. Both of these approaches require that the order in which modules are to be tested and integrated be planned before commencing testing.

It is assumed that a system is a hierarchy of modules. For such systems, there are two common ways modules can be combined, as they are tested, to form a working program: top-down and bottom-up. In top-down strategy, we start by testing the top of the hierarchy, and we incrementally add modules that it calls and then test the new combined system. This approach of testing requires stubs to be written. A stub is a dummy routine that simulates a module. In the top-down approach, a module (or a collection) cannot be tested in isolation because they invoke some other modules. To allow the modules to be tested before their subordinates have been coded, stubs simulate the behavior of the subordinates.

The bottom-up approach starts from the bottom of the hierarchy. First, the modules at the very bottom, which have no subordinates, are tested. Then these modules are combined with higher-level modules for testing. At any stage of testing, all the subordinate modules exist and have been tested earlier. To perform bottom-up testing, drivers are needed to set up the appropriate environment and invoke the module. It is the job of the driver to invoke the module under testing with the different set of test cases.

Both top-down and bottom-up approaches are incremental, starting with testing single modules and then adding untested modules to those that have been tested, until the entire system is tested. In the first case, stubs must be written to perform testing, and in the other, drivers need to be written. Top-down testing is advantageous, if major flaws occur toward the top of the hierarchy, while bottom-up is advantageous if the major flaws occur toward the bottom. Often, writing stubs can be more difficult than writing drivers, because one may need to know beforehand the set of inputs for the module being simulated by the stub and to determine proper responses for these inputs. In addition, as the stubs often simulate the behavior of a module over a limited domain, the choice of test cases for the super-ordinate module is limited, and deciding test cases is often very difficult.

It is often best to select the testing method to conform with the development method. Thus, if the system is developed in a top-down manner, top-down testing should be used, and if the system is developed in a bottom-up manner, a bottom-up testing strategy should be used. By doing this, as parts of the system are developed, they are tested, and errors are detected as development proceeds. It should be pointed out that we are concerned with actual program development here, not the design method. The development can be bottom-up even if the design was done in a top-down manner.

Test Cases and Test Criteria

Having test cases that are good at revealing the presence of faults is central to successful testing. The reason for this is that if there is a fault in a program, the program can still provide the expected behavior for many inputs. Only for the set of inputs that exercise the fault in the program will the output of the program deviate from the expected behavior. Hence, it is fair to say that testing is as good as its test cases.

Ideally, we would like to determine a set of test cases such that successful execution of all of them implies that there are no errors in the program. This ideal goal cannot, usually, be achieved due to practical and theoretical constraints. Each test case costs money, as effort is needed to generate the test case, machine time is needed to execute the program for that test case, and more effort is needed to evaluate the results. Therefore, we would also like to minimize the number of test cases needed to detect errors. These are the two fundamental goals of a practical testing activity-maximize the number of errors detected and minimize the number of test cases (i.e., minimize the cost). As these two are frequently contradictory, the problem of selecting the set of test cases with which a program should be tested becomes more complex.

While selecting test cases, the primary objective is to ensure that if there is an error or fault in the program, it is exercised by one of the test cases. An ideal test case set is one that succeeds (meaning that its execution reveals no errors) only if there are no errors in the program. One possible ideal set of test cases is one that includes all the possible inputs to the program. This is often called exhaustive testing. However, exhaustive testing is impractical and infeasible, as even for small programs the number of elements in the input domain can be extremely large.

Hence, a realistic goal for testing is to select a set of test cases that is close to ideal. How should we select our test cases? On what basis, should we include some element of the program domain in the set of test cases and not include others? For this, test selection criterion (or simply test criterion) can be used. For a given program P and its specifications, a test selection criterion specifies the conditions that must be satisfied by a set of test cases T. The criterion becomes a basis for test case selection. For example, if the criterion is that all statements in the program be executed at least once during testing, then a set of test cases T satisfies this criterion for a program P if the execution of P with T ensures that each statement in P is executed at least once.

There are two aspects of test case selection specifying a criterion for evaluating a set of test cases, and generating a set of test cases that satisfy a given criterion. As we will see, many test case criteria have been proposed. However, generating test cases for most of these is not easy and cannot, in general, be automated fully. Often, a criterion is spec ified and the tester has to generate test cases that satisfy the criterion. In some cases, guidelines are available for deciding test cases. Overall, the problem of test case selection is very challenging, and current solutions are limited in scope.

There are two fundamental properties for a testing criterion: reliability and validity. A criterion is reliable if all the sets (of test cases) that satisfy the criterion detect the same errors. That is, it is insignificant which of the sets satisfying the criterion is chosen; every set will detect exactly the same errors. A criterion is valid, if for any error in the program, there is some set satisfying the criterion that will reveal the error. A fundamental theorem of testing is that if a testing criterion is valid and reliable, if a set satisfying the criterion succeeds (revealing no faults) then the program contains no errors. However, it has been shown that no algorithm exists that will determine a valid criterion for an arbitrary program.

Psychology of Testing

Devising a set of test cases that will guarantee that all errors will be detected is not feasible. Moreover, there are no formal or precise methods for selecting test cases. Even though, there are a number of heuristics and rules of thumb for deciding the test cases, selecting test cases is still a creative activity that relies on the ingenuity of the tester. Due to this reason, the psychology of the person performing the testing becomes important.

The aim of testing is often to demonstrate that a program works by showing that it has no errors. This is the opposite of what testing should be viewed as. The basic purpose of the testing phase is to detect the errors that may be present in the program. Hence, one should not start testing with the intent of showing that a program works; but the intent should be to show that a program does not work. With this in mind, we define testing as follows: testing is the process of executing a program with the intent of finding errors.

This emphasis on proper intent of testing is a trivial matter because test cases are designed by human beings, and human beings have a tendency to perform actions to achieve the goal they have in mind. So, if the goal is to demonstrate that a program works, we may consciously or subconsciously select test cases that will try to demonstrate that goal and that will beat the basic purpose of testing. On the other hand, if the intent is to show that the program does not work, we will challenge our intellect to find test cases towards that end, and we are likely to detect more errors. Testing is, essentially, a destructive process, where the tester has to treat the program as an adversary that must be beaten by the tester by showing the presence of errors. With this in mind, a test case is "good" if it detects an as-yet-undetected error in the program, and our goal during designing test cases should be to design such "good" test cases.

One of the reasons, many organizations require a product to be tested by people not involved with developing the program before finally delivering it to the customer, is this psychological factor. It is hard to be destructive to something we have created ourselves, and we all like to believe that the program we have written "works." So, it is not easy for someone to test his own program with the proper frame of mind for testing. Another reason for independent testing is that sometimes errors occur because the programmer did not understand the specifications clearly. Testing of a program by its programmer will not detect such errors, whereas independent testing may succeed in finding them.

10.4 TEST CASE DESIGN

There are two basic approaches to test case design: functional and structural. In functional testing, the structure of the program is not considered. Structural testing, on the other hand, is concerned with testing the implementation of the program.

10.4.1 Functional Testing

Equivalence partitioning is a technique for determining which classes of input data have common properties. A program should behave in a comparable way for all members of an equivalence partition.

The equivalence partitions may be identified by using program specification or user documentation and by the tester using experience, to predict which classes of input value are likely to detect errors. For example, if an input specification states that the range of some input values must be a 5-digit integer, that is, between 10000 and 99999, equivalence partitions might be those values less than 10000, values between 10000 and 99999 and values greater than 99999. Similarly, if four to eight values are to be input, equivalence partitions are less than four, between four and eight and more than eight.

In functional testing, the structure of the program is not considered. Test cases are decided solely on the basis of the requirements or specifications of the program or module, and the internals of the module or the program are not considered for selection of test cases. Due to its nature, functional testing is often called "black box testing." In the structural approach, test cases are generated based on the actual code of the program or module to be tested. This structural approach is sometimes called "glass box testing."

The basis for deciding test cases in functional testing is the requirements or specifications of the system or module. For the entire system, the test cases are designed from the requirements specification document for the system. For modules created during design, test cases for functional testing are decided from the module specifications produced during the design.

The most obvious functional testing procedure is exhaustive testing, which as we have stated, is impractical. One criterion for generating test cases is to generate them randomly. This strategy has little chance of resulting in a set of test cases that is close to optimal (i.e., that detects the maximum errors with minimum test cases). Hence, we need some other criterion or rule for selecting test cases. There are no formal rules for designing test cases for functional testing. In fact, there are no precise criteria for selecting test cases. However, there are a number of techniques or heuristics that can be used to select test cases that have been found to be very successful in detecting errors. Here are some of these techniques:

(i) Equivalence Class Partitioning

Functional testing is an approach to testing where the specification of the component being tested is used to derive test cases. The component is a "black box" whose behavior can only be determined by studying its inputs and the related outputs.

The key problem for the tester whose aim is to discover defects is to select inputs, which have a high probability of being members of the set. Effective selection is dependent on the skill and experience of the tester but there are some structured approaches, which can be used to guide the selection of test data.

However, without looking at the internal structure of the program, it is impossible to determine such ideal equivalence classes (even with the internal structure, it usually cannot be done). The equivalence class partitioning method tries to approximate this ideal. Different equivalence classes are formed by putting inputs for which the behavior pattern of the module is specified to be different into similar groups and then regarding these new classes as forming equivalence classes. The rationale of forming equivalence classes like this, is the assumption that if the specifications require exactly the same behavior for each element in a class of values, then the program is likely to be constructed so that it either succeeds or fails for each of the values in that class. For example, the specifications of a module that determine the absolute value for integers specify one behavior for positive integers and another for negative integers. In this case, two equivalence classes-one consisting of positive integers and the other consisting of negative integers will be formed.

For robust software, we must also test for incorrect inputs by generating test cases for inputs that do not satisfy the input conditions. With this in mind, for each equivalence class of valid inputs we define equivalence classes for invalid inputs.

Equivalence classes are usually formed by considering each condition specified on an input as specifying a valid equivalence class and one or more invalid equivalence classes. For example, if an input condition specifies a range of values (say, 0 < count < max), then forms a valid equivalence class with that range and two invalid equivalence classes, one with values less than the lower bound of the range (i.e., count < 0) and the other with values higher than the higher bound (count> max). If the input specifies a set of values and the requirements specify different behavior for different elements in the set, then a valid equivalence class is formed for each of the elements in the set and an invalid class for an entity not belonging to the set.

Essentially, if there is reason to believe that the entire range of an input will not be treated in the same manner, then the range should be split into two or more equivalence classes. Also, for each valid equivalence class, one or more invalid equivalence classes should be identified. For example, an input may be specified as a character. However, we may have reason to believe that the program will perform different actions if a character is an alphabet, a number, or a special character. In that case, we will split the input into three valid equivalence classes.

It is often useful to consider equivalence classes in the output. For an output equivalence class, the goal is to generate test cases such that the output for that test case lies in the output equivalence class. Determining test cases for output classes may be more difficult, but output classes have been found to reveal errors that are not revealed by just considering the input classes.

(ii) Boundary Value Analysis

It has been observed that 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, and selecting such test cases is the aim of the boundary value analysis. In boundary value analysis, we choose an input for a test case from an equivalence class, such that the input lies at the edge of the equivalence classes. Boundary values for each equivalence class, including the equivalence classes of the output, should be covered. Boundary value test cases are also called "extreme cases", Hence, we can say that a boundary value test case is a set of input data that lies on the edge or boundary of a class of input data or that generates output that lies at the boundary of a class of output data.

In case of ranges, for boundary value analysis it is useful to select the boundary elements of the range and an invalid value just beyond the two ends (for the two invalid equivalence classes). So, if the range is $0.0 < x < 1.0$, then the test cases are 0.0, 1.0 (valid inputs), and - 0.1, and 1.1 (for invalid inputs). Similarly, if the input is a list, attention should be focused on the first, and last elements of the list. We should also consider the outputs for boundary value analysis. If an equivalence class can be identified in the output, we should try to generate test cases that will produce the output that lies at the boundaries of the equivalence classes. Furthermore, we should try to form test cases that will produce an output that does not lie in the equivalence class.

(iii) Cause-Effect Graphing

One weakness with the equivalence class partitioning and boundary value methods is that they consider each input separately. That is, both concentrate on the conditions and classes of one input. They do not consider combinations of input circumstances that may form interesting situations that should be tested. One way to exercise combinations of different input conditions is to consider all valid combinations of the equivalence classes of input conditions. This simple approach will result in an unusually large number of test cases, many of which will not be useful for revealing any new errors. For example, if there are n different input conditions, such that any combination of the input conditions is valid, we will have 2 test cases.

Cause-effect graphing is a technique that aids in selecting combinations of input conditions in a systematic way, such that the number of test cases does not become unmanageably large. The technique starts with identifying causes and effects of the system under testing. A cause is a distinct input condition, and an effect is a distinct output condition. Each condition forms a node in the cause-effect graph. The conditions should be stated such that they can be set to either true or false. For example, an input condition can be "file is empty," which can be set to true by having an empty input file, and false by a nonempty file. After identifying the causes and effects, for each effect we identify the causes that can produce that effect and how the conditions have to be combined to make the effect true. Conditions are combined using the Boolean operators "and," "or," and "not," which are represented in the graph by &, I, and ". Then, for each effect, all combinations of the causes that the effect depends on which will make the effect true, are generated. By doing this, we identify the combinations of conditions that make different effects true. A test case is then generated for each combination of conditions, which make some effects true. For example, th elist of Causes and Effects for operating a Bank account could be:

Causes:

c1. Command is credit

c2. Command is debit

c3. Account number is valid

c4. Transaction_amount. is valid

Effects:

el. Print "invalid command"

e2. Print "invalid account-number"

e3. Print "Debit amount not valid"

e4. Debit account

e5. Credit account

Let us illustrate this technique with a small example. Suppose that for a bank database there are two commands allowed:

credit acct-number transaction_amount

debit acct-number transaction_amount

The requirements are that if the command is credit and the acct-number is valid, then the account is credited. If the command is debit, the acct-number is valid, and the transaction_amount is valid (less than the balance), then the account is debited. If the command is not valid, the account number is not valid, or the debit amount is not valid, a suitable message is generated. We can identify the following causes and effects from these requirements, shown in Figure 10.2.

The cause effect of this is shown in Figure 10.2. In the graph, the cause-effect relationship of this example is captured. For all effects, one can easily determine the causes each effect depends on and the exact nature of the dependency. For example, according to this graph, the effect E5 depends on the causes c2, c3, and c4 in a manner such that the effect E5 is enabled when all c2, c3, and c4 are true. Similarly, the effect E2 is enabled if c3 is false.

From this graph, a list of test cases can be generated. The basic strategy is to set an effect to I and then set the causes that enable this condition. The condition of causes forms the test case. A cause may be set to false, true, or don't care (in the case when the effect does not depend at all on the cause). To do this for all the effects, it is convenient to use a decision table. The decision table for this example is shown in Figure 10.3.

This table lists the combinations of conditions to set different effects. Each combination of conditions in the table for an effect is a test case. Together, these condition combinations check for various effects the software should display. For example, to test for the effect E3, both c2 and c4 have to be set. That is, to test the effect "Print debit amount not valid," the test case should be: Command is debit (setting: c2 to True), the account number is valid (setting c3 to False), and the transaction money is not proper (setting c4 to False).

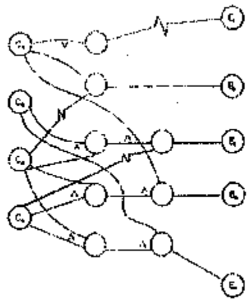


Figure10.2: The Cause Effect Graph

SNo.	1	2	3	4	5
C1 0	1	x	x	1	
C2 0	x	1	1	x	
C3 x	0	1	1	1	
C4 x	x	0	1	1	
E1 1					
E2	1				
E3		1			
E4			1		
E5				1	

Figure 10.3: Decision Table for the Cause-effect Graph

Cause-effect graphing, beyond generating high-yield test cases, also aids the understanding of the functionality of the system, because the tester must identify the distinct causes and effects. There are methods of reducing the number of test cases generated by proper traversing of the graph. Once the causes and effects are listed and their dependencies specified, much of the remaining work can also be automated.

(iv) Special Cases

It has been seen that programs often produce incorrect behavior when inputs form some special cases. The reason is that in programs, some combinations of inputs need special treatment, and providing proper handling for these special cases is easily overlooked. For example, in an arithmetic routine, if there is a division and the divisor is zero, some special action has to be taken, which could easily be forgotten by the programmer. These special cases form particularly good test cases, which can reveal errors that will usually not be detected by other test cases.

Special cases will often depend on the data structures and the function of the module. There are no rules to determine special cases, and the tester has to use his intuition and experience to identify such test cases. Consequently, determining special cases is also called error guessing.

The psychology is particularly important for error guessing. The tester should play the "devil's advocate" and try to guess the incorrect assumptions that the programmer could have made and the situations the programmer could have overlooked or handled incorrectly. Essentially, the tester is trying to identify error prone situations. Then, test cases are written for these situations. For example, in the problem of finding the number of different words in a file (discussed in earlier chapters) some of the special cases can be: file is empty, only one word in the file, only one word in a line, some empty lines in the input file, presence of more than one blank between words, all words are the same, the words are already sorted, and blanks at the start and end of the file.

Incorrect assumptions are usually made because the specifications are not complete or the writer of specifications may not have stated some properties, assuming them to be obvious. Whenever there is reliance on tacit understanding rather than explicit statement of specifications, there is scope for making wrong assumptions. Frequently, wrong assumptions are made about the environments. However, it should be pointed out that special cases depend heavily on the problem, and the tester should really try to "get into the shoes" of the designer and coder to determine these cases.

10.4.2 Structural Testing

A complementary approach to testing is sometimes called structural or White box or Glass box testing. The name contrasts with black box testing because the tester can analyse the code and use knowledge about it and the structure of a component to derive the test data. The advantage of structural testing is that test cases can be derived systematically and test coverage measured. The quality assurance mechanisms, which are setup to control testing, can quantify what level of testing is required and what has be carried out. In the previous section, we discussed functional testing, which is concerned with the function that the tested program is supposed to perform and does not deal with the internal structure of the program responsible for actually implementing that function. Thus, functional testing is concerned with functionality rather than implementation of the program. Various criteria for functional testing were discussed earlier. Structural testing, on the other hand, is concerned with testing the implementation of the program. The intent of structural testing is not to exercise all the different input or output conditions (although that may be a by-product) but to exercise the different programming structures and data structures used in the program.

To test the structure of a program, structural testing aims to achieve test cases that will force the desired coverage of different structures. Various criteria have been proposed for this. Unlike the criteria for functional testing, which are frequently imprecise, the criteria for structural testing are generally quite precise as they are based on program structures, which are formal and precise. Here we will discuss three different approaches to structural testing: control flow-based testing, data flow-based testing, and mutation testing.

Control Flow-Based Criteria

Before we consider the criteria, let us precisely define a control flow graph for a program. Let the control flow graph (or simply flow graph) of a program P be G. A node in this graph represents a block of statements that is always executed together, i.e., whenever the first statement is executed, all other statements are also executed. An edge (i, j) (from node i to node j) represents a possible transfer of control after executing the last statement of the block represented by node i to the first statement of the block represented by node j. A node corresponding to a block, whose first statement is the start statement of P, is called the start node of G, and a node corresponding to a block whose last statement is an exit statement is called an exit node. A path is a finite sequence of nodes (n1, nz, nk), k > I, such that there is an edge (ni, ni+1) for all nodes n; in the sequence (except the last node nk). A complete path is a path whose first node is the start node and the last node is an exit node.

Now, let us consider control flow-based criteria. Perhaps, the simplest coverage criteria is statement coverage, which requires that each statement of the program be executed at least once during testing. In other words, it requires that the paths executed during testing include all the nodes in the graph. This is also called the all-nodes criterion. This coverage criterion is not very strong, and can leave errors undetected. For example, if there is an if statement in the program without having an else clause, the statement coverage criterion for this statement will be satisfied by a test case that evaluates the condition to true. No test case is needed that ensures that the condition in the if statement evaluates to false. This is a serious shortcoming because decisions in programs are potential sources of errors. As an example, consider the following function to compute the absolute value of a number:

```
int xyz (y)
int y;
{
      if (y >= 0) y = 0 -y;
      return (y)
}
```

This program is clearly wrong. Suppose we execute the function with the set of test cases {y-a} (i.e., the set has only one test case). The statement coverage criterion will be satisfied by testing with this set, but the error will not be revealed.

A little more general coverage criterion is branch coverage, which requires that each edge in the control flow graph be traversed at least once during testing. In other words, branch coverage requires that each decision in the program be evaluated to true and false values at least once during testing. Testing based on branch coverage is often called branch testing. The 100% branch coverage criterion is also called the all-edges criterion. Branch coverage implies statement coverage, as each statement is a part of some branch. In the preceding example, a set of test cases satisfying this criterion will detect the error.

The trouble with branch coverage comes if a decision has many conditions in it (consisting of a Boolean expression with Boolean operators and and or). In such situations, a decision can evaluate to true and false without actually exercising all the conditions. For example, consider the following function that checks the validity of a data item. The data item is valid if it lies between 0 and 100.

```
int check(y)
int y;
{
      if "y >=) && (y <= 200))
      check = True;
      else check = False;
}
```

The module is incorrect, as it is checking for y < 200 instead of 100 (perhaps, a typing error made by the programmer). Suppose the module is tested with the following set of test cases: {y = 5, y = -5}. The branch coverage criterion will be satisfied for this module by this set. However, the error will not be revealed, and the behavior of the module is consistent with its specifications for all test cases in this set. Thus, the coverage criterion is satisfied, but the error is not detected. This occurs because the decision is evaluating to true and false because of the condition (y > 0). The condition (y < 200) never evaluates to false during this test, hence the error in this condition is not revealed.

This problem can be resolved by requiring that all conditions evaluate to true and false. However, situations can occur where a decision may not get both true and false values even if each individual condition evaluates to true and false. An obvious solution to this problem is to require decision/condition coverage, where all the decisions and all the conditions in the decisions take both true and false values during the course of testing.

Studies have indicated that there are many errors whose presence is not detected by branch testing because some errors are related to some combinations of branches and their presence is revealed by an execution that follows the path that includes those branches. Hence, a more general coverage criterion is one that requires all possible paths in the control flow graph be executed during testing. This is called the path coverage criterion or the all-paths criterion, and the testing based on this criterion is often called path testing. The difficulty with this criterion is that programs that contain loops can have an infinite number of possible paths. Furthermore, not all paths in a graph may be "feasible" in the sense that there may not be any inputs for which the path can be executed. It should be clear that C path => Cbranch.

As the path coverage criterion leads to a potentially infinite number of paths, some efforts have been made to suggest criteria between the branch coverage and path coverage. The basic aim of these approaches is to select a set of paths that ensure branch coverage criterion and try some other paths that may help reveal errors. One method to limit the number of paths is to consider two paths as same, if they differ only in their sub-paths that are caused due to the loops. Even with this restriction, the number of paths can be extremely large.

Another such approach based on the cyclomatic complexity has been proposed namely, the test criterion. The test criterion is that if the cyclomatic complexity of a module is V, then at least V distinct paths must be executed during testing. We have seen that cyclomatic complexity V of a module is the number of independent paths in the flow graph of a module. As these are independent paths, all other paths can be represented as a combination of these basic paths. These basic paths are finite, whereas the total number of paths in a module having loops may be infinite.

It should be pointed out that none of these criteria is sufficient to detect all kind of errors in programs. For example, if a program is missing out some control flow paths that are needed to check for a special value (like pointer equals nil and divisor equals zero), then even executing all the paths will not necessarily detect the error. Similarly, if the set of paths is such that they satisfy the all-path criterion but exercise only one part of a compound condition, then the set will not reveal any error in the part of the condition that is not exercised. Hence, even the path coverage criterion, which is the strongest of the criteria we have discussed, is not strong enough to guarantee detection of all the errors.

Data Flow-Based Testing

Data Flow based testing implies criteria that select the paths to be executed during testing based on data flow analysis, rather than control flow analysis. In the data flow-based testing approaches, besides the control flow, information about where the variables are defined and where the definitions are used is also used to specify the test cases. The basic idea behind data flow-based testing is to make sure that during testing, the definitions of variables and their subsequent use is tested. Just like the all-nodes and all-edges criteria try to generate confidence in testing by making sure that at least all statements and all branches have been tested, the data flow testing tries to ensure some coverage of the definitions and uses of variables. Approaches for use of data flow information have been proposed in. Our discussion here is based on the family of data flow-based testing criteria that were proposed. Some of these criteria are discussed here.

For data flow-based criteria, a definition-use graph (def-use graph, for short) for the program is first constructed from the control flow graph of the program. A statement in a node in the flow graph representing a block of code has variable occurrences in it. A variable occurrence can be one of the following three types:

- def represents the definition of a variable. The variable on the left-hand side of an assignment statement is the one getting defined.

- c-use represents computational use of a variable. Any statement (e.g., read/write an assignment) that uses the value of variables for computational purposes is said to be making c-use of the variables. In an assignment statement, all variables on the right-hand side have a c-use occurrence. In a read and a write statement, all variable occurrences are of this type.
- p-use represents predicate use. These are all the occurrences of the variables in a predicate (i.e., variables whose values are used for computing the value of the predicate), which is used for transfer of control.

Mutation Testing

Mutation testing is another structural testing technique that differs fundamentally from the approaches discussed earlier. In control flow-based and data flow-based testing, the focus was on which paths to execute during testing. Mutation testing does not take a path-based approach. Instead, it takes the program and creates many mutants of it, by making simple changes to the program. The goal of testing is to make sure that during the course of testing, each mutant produces an output different from the output of the original program. In other words, the mutation-testing criterion does not say that the set of test cases must be such that certain paths are executed; instead, it requires the set of test cases to be such that they can distinguish between the original program and its mutants.

In hardware, testing is based on some fault models that have been developed and that model the actual faults closely. The fault models provide a set of simple faults, combination of which can model any fault in the hardware. In software, however, no such fault model exists. That is why most of the testing techniques try to guess where the faults might lie and then select the test cases that will reveal those faults.

In mutation testing, faults of some pre-decided types are introduced in the program being tested. Testing, then tries to identify those faults in the mutants. The idea is that if all these "faults" can be identified, then the original program should not have these faults; otherwise they would have been identified in that program by the set of test cases.

This technique will be successful only if the changes introduced in the main program capture the most likely faults in some form. This is assumed to hold, due to the competent programmer hypothesis and the coupling effect. The competent programmer hypothesis says that programmers are generally very competent and do not create programs at random, and for a given problem, a programmer will produce a program that is very "close" to a correct program. In other words, a correct program can be constructed from an incorrect program with some minor changes in the program. The coupling effect says that the test cases that distinguish programs with minor differences in each other are so sensitive that they will also distinguish programs with more complex differences. Some experiments are cited, in which it has been shown that the test data that can distinguish mutants created by simple changes can also distinguish up to 99% of the mutants that have been created by applying a series of simple changes.

10.5 TEST CASE GENERATION AND TOOL SUPPORT

Once a coverage criterion is decided, two problems have to be solved to use the chosen criterion for testing. The first is to decide if a set of test cases satisfy the criterion, and the second is to generate a set of test cases for a given criterion. Deciding whether a set of test cases satisfies a criterion without the aid of any tools is a cumbersome task, though it is theoretically possible to do it manually. For almost all the structural testing techniques, tools are used to determine whether the criterion has been satisfied. Generally, these tools will provide feedback regarding what needs to be tested to fully satisfy the criterion.

To generate the test cases, tools are not that easily available, and due to the nature of the problem (i.e., undecidability of "feasibility" of a path), a fully automated tool for selecting test cases to satisfy a criterion is generally not possible. Hence, tools can, at best, aid the tester. One method for generating test cases is to randomly select test data until the desired criterion is satisfied (which is determined by a tool). This can result in a lot of redundant test cases, as many test cases will exercise the same paths.

As test case generation cannot be fully automated, frequently, the test case selection is done manually by the tester, by performing structural testing in an iterative manner, starting with an initial test case set and selecting more test cases based on the feedback provided by the tool for test case evaluation. The test case evaluation tool can tell which paths need to be executed or which mutants need to be killed. This information can be used to select further test cases. For example, to select a test case to execute some path, static data flow analysis tools can be used to decide what input values should be chosen so that when the program is executed, this particular path is executed. Symbolic evaluation tools can also be quite useful here. The paths that need to be executed during testing can be treated as programs in their own right and can be symbolically executed. With symbolic execution, the conditions on input variables that will enable this path to be executed can be determined.

However, even with the aid of tools, selecting test cases is not a simple mechanical process. Ingenuity and creativity of the tester are still important, even with the availability of the tools to determine the coverage. Because of this, and for other freasons, the criteria are often weakened. For example, instead of requiring 100% coverage of statements and branches, the goal might be to achieve some acceptably high percentage (but less than 100%).

Test Plan Activities During Testing

A test plan is a general document for the entire project that defines the scope, approach to be taken, and the schedule of testing as well as identifies the test items for the entire testing process and the personnel responsible for the different activities of testing. The test planning can be done well before the actual testing commences and can be done in parallel with the coding and design phases. The inputs for forming the test plan are: (1) project plan, (2) requirements document, and (3) system design document. The project plan is needed to make sure that the test plan is consistent with the overall plan for the project and the testing schedule matches that of the project plan. The requirements document and the design document are the basic documents used for selecting the test units and deciding the approaches to be used during testing. A test plan should contain the following:

- Test unit specification.
- Features to be tested.
- Approach for testing.
- Test deliverables.
- Schedule.
- Personnel allocation.

One of the most important activities of the test plan is to identify the test units. A test unit is a set of one or more modules, together with associated data, that are from a single computer program and that are the objects of testing. A test unit can occur at any level and can contain from a single module to the entire system. Thus, a test unit may be a module, a few modules, or a complete system.

As seen earlier, different levels of testing have to be used during the testing activity. The levels are specified in the test plan by identifying the test units for the project. Different units are usually specified for unit, integration, and system testing. The identification of test units establishes the different levels of testing that will be performed in the project. Generally, a number of test units are formed during the testing, starting from the lower-level modules, which have to be unit tested. That is, first the modules that have to be tested individually are specified as test units. Then the higher-level units are specified, which may be a combination of already tested units or may combine some already tested units with some untested modules. The basic idea behind forming test units is to make sure that testing is being performed incrementally, with each increment including only a few aspects that need to be tested.

An important factor while forming a unit is the "testability" of a unit. A unit should be such that it can be easily tested. In other words, it should be possible to form meaningful test cases and execute the unit without much effort with these test cases.

For example, a module that manipulates the complex data structure formed from a file input by an input module might not be a suitable unit from the point of view of testability. As forming meaningful test cases for the unit will be hard, and driver routines will have to be written to convert inputs from files or terminals that are given by the tester into data structures suitable for the module. In this case, it might be better to form the unit by including the input module as well. Then the file input expected by the input module can contain the test cases.

Features to be tested include all software features and combinations of features that should be tested. A software feature is a software characteristic specified or implied by the requirements or design documents. These may include functionality, performance, design constraints, and attributes.

The approach for testing specifies the overall approach to be followed in the current project. The techniques that will be used to judge the testing effort should also be specified. This is, sometimes called the testing criterion or the criterion for evaluating the set of test cases used in testing. In the previous sections, we discussed many criteria for evaluating and selecting test cases.

Testing deliverables should be specified in the test plan before the actual testing begins. Deliverables could be a list of test cases that were used, detailed results of testing, test summary report, test log, and data about the code coverage. In general, a test case specification report, test summary report, and a test log should always be specified as deliverables. Test case specification is discussed later. The test summary report summarizes the results of the testing activities and evaluates the results. It defines the items tested, the environment in which testing was done, and any variations from the specifications observed during testing. The test log provides a chronological record of relevant details about the execution of the test cases.

The schedule specifies the amount of time and effort to be spent on different activities of testing, and testing of different units that have been identified. Personnel allocation identifies the persons responsible for performing the different activities.

Specifications for Test Case

The test plan focuses on how the testing for the project will proceed, which units will be tested, and what approaches (and tools) are to be used during the various stages of testing. However, it does not deal with the details of testing a unit, nor does it specify which test cases are to be used.

Test case specification has to be done separately for each unit. Based on the approach specified in the test plan, first the features to be tested for this unit must be determined. The overall approach stated in the plan is refined into specific test techniques that should be followed and into the criteria to be used for evaluation. Based on these, the test cases are specified for testing the unit. Test case specification gives, for each unit to be tested, all test cases, inputs to be used in the test cases, conditions being tested by the test case, and outputs expected for those test cases.

Test case specification is a major activity in the testing process. Careful selection of test cases that satisfy the criterion and, specified approach is essential for proper testing. We have considered many methods of generating test cases and criteria for evaluating test cases. A combination of these can be used to select the test cases. It should be pointed out that test case specifications contain not only the test cases, but also the rationale of selecting each test case (such as what condition it is testing) and the expected output for the test case.

There are two basic reasons test cases are specified before they are used for testing. It is known that testing has severe limitations and the effectiveness of testing depends very heavily on the exact nature of the test cases. Even for a given criterion, the exact nature of the test cases affects the effectiveness of testing. Constructing "good" test cases that will reveal errors in programs is still a very creative activity that depends a great deal on the ingenuity of the tester. Clearly, it is important to ensure that the set of test cases used is of "high quality."

As with many other verification methods, evaluation of quality of test cases is done through "test case review." And for any review, a formal document or work product is needed. This is the primary reason for having the test case specification in the form of a document. The test case specification document is reviewed, using a formal review process, to make sure that the test cases are consistent with the policy specified in the plan, satisfy the chosen criterion, and in general cover the various aspects of the unit to be tested. For this purpose, the reason for selecting the test case and the expected output are also given in the test case specification document. By looking at the conditions being tested by the test cases, the reviewers can check if all the important conditions are being tested. As conditions can also be based on the output, by considering the expected outputs of the test cases, it can also be determined if the production of all the different types of outputs the unit is supposed to produce are being tested. Another reason for specifying the expected outputs is to use it as the "oracle" when the test case is executed.

Besides reviewing, another reason for formally specifying the test cases in a document is that the process of sitting down and specifying all the test cases that will be used for testing helps the tester in selecting a good set of test cases. By doing this, the tester can see the testing of the unit in totality and the effect of the total set of test cases. This type of evaluation is hard to do in on-the-fly testing where test cases are determined as testing proceeds.

Another reason for formal test case specifications is that the specifications can be used as "scripts" during regression testing, particularly if regression testing is to be performed manually. Generally, the test case specification document itself is used to record the results of testing. That is, a column is created when test cases are specified that is left blank. When the test cases are executed, the results of the test cases are recorded in this column. Hence, the specification document eventually also becomes a record of the testing results.

10.6 TEST CASE EXECUTION AND ANALYSIS

With the specification of test cases, the next step in the testing process is to execute them. This step is also not straightforward. The test case specifications only specify the set of test cases for the unit to be tested. However, executing the test cases may require construction of driver modules or stubs. It may also require modules to set up the environment as stated in the test plan and test case specifications. If data is to be collected, then data collection forms need to be set up or data collection software developed. Only after all these are ready can the test cases be executed. Sometimes, the steps to be performed to execute the test cases are specified in a separate document called the test procedure specification. This document specifies any special requirements that exist for setting the test environment and describes the methods and formats for reporting the results of testing. Measurements, if needed, are also specified, along with how to obtain them.

Various outputs are produced as a result of test case execution for the unit under test. These outputs are needed to evaluate if the testing has been satisfactory. The most common outputs are the test log, the test summary report, and the error report. The test log describes the details of testing. As mentioned earlier, the test case specification document itself can act as the document for logging the details of testing. The test summary report is meant for project management, where the summary of the entire test case execution is provided. The summary gives the total number of test cases executed, the number and nature of errors found, and a summary of any metrics data (e.g., effort) collected. The error report gives the summary of all the errors found. The errors might also be categorized into different levels, if such a categorization is available and its use has been planned in the test plan. This information can also be obtained from the test log, but it is usually given as a separate document. This report is frequently used to track the status of defects found during testing.

After testing is complete, the efficiency of the various defect removal techniques can be studied. The efficiency of a defect removal process can be defined if the total number of errors in the software is known. This data is not known but can be approximated more accurately after all the defects found in testing are known. The defect removal efficiency of a defect removing process is defined as the percentage reduction of the defects that are present before the initiation of the process. The cumulative defect removal efficiency of a series of defect removal processes is the percentage of defects that have been removed by this series, based on the number of defects present at the beginning of the series.

For example, suppose a total of 10 defects are detected during development and I field operation. We can estimate the total number of errors in the software before the defect removal operations began as 10. Suppose that during reviews, four defects were removed. The defect removal efficiency of reviews in this example is 40%. Suppose that during testing, another four defects are removed. The defect removal efficiency of testing then is 66% (as it removed four out of the six remaining defects). The cumulative defect removal efficiency of reviews followed by testing is 80%. Defect removal efficiencies of the different methods are useful for evaluating the quality assurance process being used. It can also be used to evaluate how well the activities are performed in a given project, if process data from previous projects is available.

Testing and coding are the two phases that require careful monitoring, as these phases involve the maximum number of people. A few parameters can be observed for monitoring the testing process. Testing effort is the total effort spent by the project team in testing activities; and is an excellent indicator of whether or not testing is sufficient. In particular, if inadequate testing is done, it will be reflected in a reduced testing effort. From past experience, we know that the total testing effort should be about 40% of the total effort for developing the software (the exact percentage will depend on the process and will have to be determined for the process). From this, the estimate of the effort required for testing, compared to coding or design, can be computed and used for monitoring. Such monitoring can catch the "miracle finish" cases, where the project "finishes" suddenly, soon after the coding is done. Such "finishes" occur for reasons such as unreasonable schedules, personnel shortages, and slippage of schedule. Such a finish, usually, implies that to finish the project, the testing phase has been compressed too much, which is likely to mean that the software has not been evaluated properly.

Computer time consumed during testing is another measure that can give valuable information to project management. In general, in a software development project, the computer time consumption is low at the start, increases as time progresses, and reaches a peak. Thereafter, it is reduced, as the project reaches its completion. Maximum computer time is consumed during the latter part of coding and testing. By monitoring the computer time consumed, one can get an idea about how thorough the testing has been. Again, by comparing the previous buildups in computer time consumption, computer time consumption of the current project can provide valuable information about whether or not the testing is adequate.

Error tracing is an activity that does not directly affect the testing of the current project, but it has many long-term quality control benefits. By error tracing, we mean that when a fault is detected after testing, it should be studied and traced back in the development cycle to determine where it was introduced. This exercise has many benefits. First, it gives quantitative data about how many errors slip by the earlier quality control measures and which phases are more error-prone. If some particular phase is found to be more error-prone, the verification activity of that phase should be strengthened in the future and proper standards and procedures need to be developed to reduce the occurrence of errors in the future. The volume and nature of faults slipping by the earlier quality assurance measures provide valuable input for evaluation of the quality control strategies. This evaluation can be used to determine which quality control measures should be strengthened and what sort of techniques should be added. Another benefit of error tracing is productivity improvement in the future. Error tracing is a feedback mechanism that is invaluable for learning. A designer or programmer, by seeing the mistakes that occurred during his activities, will learn from the information and is less likely to make similar mistakes in the future, thereby increasing his productivity. If this feedback is not provided, such learning will not take place.

10.7 STRATEGIC ISSUES IN TESTING

Testing is a very important phase in software development life cycle. But the testing may not be very effective if proper strategy is not used. For the implementation of successful software testing strategy, the following issues must be taken care of: -

- Before the start of the testing process, all the requirements must be specified in a quantifiable manner.
- Testing objectives must be clarified and stated explicitly.
- A proper testing plan must be developed.
- Build "robust" software that is designed to test itself.
- Use effective formal technical reviews as a filter prior to testing. Formal technical reviews can be as effective as testing in uncovering errors. For this reason, reviews can reduce the amount of testing effort that is required to produce high-quality software.
- Conduct formal technical reviews to assess the test strategy and the cases themselves. Formal technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.
- Develop a continuous improvement approach for the testing process. The test strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for software testing.

10.8 LEVELS OF TESTING

Faults can occur during any phase in the software development cycle. Verification is performed on the output of each phase, but some faults are likely to remain undetected by these methods. These faults will be eventually reflected in the code. Testing is usually relied on to detect these faults, in addition to the faults introduced during the coding phase itself. Due to this, different levels of testing are used in the testing process; each level of testing aims to test different aspects of the system.

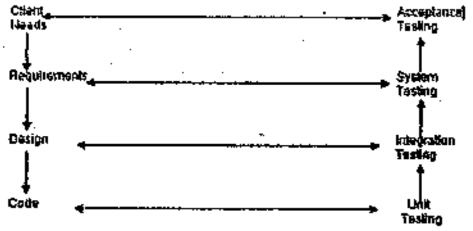


Figure 10.4: Levels of Testing

The basic levels are unit testing, integration testing, testing system and acceptance testing. These different levels of testing attempt to detect different types of faults. The relation of the faults introduced in different phases, and the different levels of testing as shown in figure 10.4.

The first level of testing is called unit testing. In this, different modules are tested against the specifications produced during design for the modules. Unit testing is, essentially, for verification of the code produced during the coding phase, hence the goal is to test the internal logic of the modules. It is typically done by the programmer of the module. A module is considered for integration and use by others only after it has been unit tested satisfactorily. Due to its close association with coding, the coding phase is frequently called "coding and unit testing". As the focus of this testing level is on testing the code, structural testing is best suited for this level. In fact, as structural testing is not very suitable for large programs, it is used mostly at the unit testing level.

The next level of testing is often called integration testing. In this, many unit-tested modules are combined into subsystems, which are then tested. The goal here is to see if the modules can be integrated properly. Hence, the emphasis is on testing interfaces between modules. This testing activity can be considered testing the design.

The next levels are system testing and acceptance testing. Here the entire software system is tested. The reference document for this process is the requirements document, and the goal is to see if the software meets its requirements. This is essentially a validation exercise, and in many situations, it is the only validation activity. Acceptance testing is sometimes performed with realistic data of the client to demonstrate that the software is working satisfactorily. Testing here focuses on the external behavior of the system; the internal logic of the program is not emphasized. Consequently, mostly functional testing is performed at these levels.

These levels of testing are performed when a system is being built from the components that have been coded. There is another level of testing called regression testing that is performed when some changes are made to an existing system. Changes are fundamental to software; any software must undergo changes. Frequently, a change is made to "upgrade" the software by adding new features and functionality. Clearly, the modified software needs to be tested to make sure that the new features to be added do, indeed, work. However, as modifications have been made to an existing system, testing also has to be done to make sure that the modification has not had any undesired side effect of making some of the earlier services faulty. That is, besides ensuring the desired behavior of the new services, testing has to ensure that the desired behavior of the old services is maintained. This is the task of regression testing.

For regression testing, some test cases that have been executed on the old system are maintained, along with the output produced by the old system. These test cases are executed again on the modified system and its output compared with the earlier output to make sure that the system is working as before on these test cases. This, frequently, is a major task when modifications are to be made to existing systems.

A consequence of this is that the test cases for systems should be properly documented for future use. Often, when we test our programs, the test cases are treated as "throw way" cases; after testing is complete, test cases and their outcomes are thrown away. With this practice, every time regression testing has to be done, the set of test cases will have to be re-created, resulting in increased cost. In fact, for many systems that are frequently changed, regression testing "scripts" are used to automatically perform the regression testing after some changes. A regression testing script contains all the inputs given by the test cases and the output produced by the system for these test cases. These scripts are typically produced during the system testing, as regression testing is generally done only for complete systems or subsystem. When the system is modified, the scripts and comparing the outputs with the outputs given in the scripts. Given the scripts, though the use of tools, regression testing can be largely automated.

Unit Testing

Unit testing compromises the set of tests performed by an individual programmer prior to integration of the unit into a larger system. The situation is illustrated as follows:

Coding and debugging Unit Testing Integration Testing

A program unit is usually small enough programmer who developed it can test it in great detail, and certainly in greater detail the will be possible when the unit is integrated into an evolving software product. There are four categories of tests that a programmer will typically perform on a program unit:

- Function Tests
- Performance Test
- Stress Tests

- Structure Tests

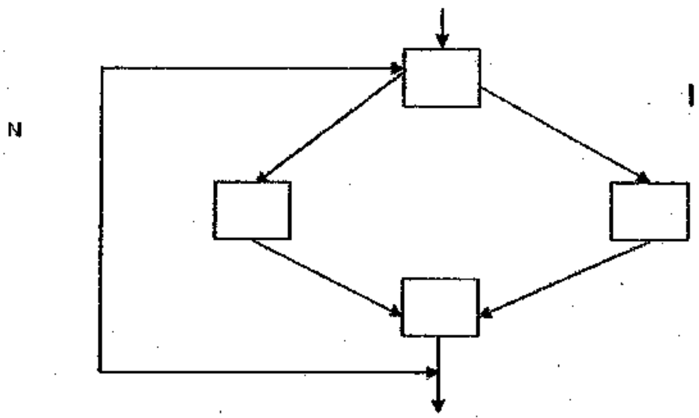
Functional test cases involve exercising the code with nominal input values for which the expected results are known, as well as boundary values (minimum values, maximum values, and values on and just outside the functional boundaries) and special values such as logically related inputs, 1x1 matrices, the identity matrix, files of identical elements, and empty files.

Performances testing determines the amount of execution time spend in various parts of the unit, program throughout, response time, and device utilization by the program unit. A certain amount of performance tuning may be done during unit testing. However, caution must be exercised to avoid expending too much effort on fine-tuning of a program unit that contributes little to the overall performance of the entire system. Performance testing is most productive at the subsystem and system levels.

Stress tests are those tests designed to intentionally break the unit. A great deal can be learned about the strengths and limitations of a program by examining the manner in which a program unit breaks.

Structure tests are concerned with exercising the internal logic of a program and traversing particular execution paths. Some authors refer collectively to functional, performance, and stress testing as "black box" testing, while structure testing is referred to as "white box" or "glass box". The major activities in structural attesting are deciding which path to exercise, deriving test data to exercise those and measuring the test coverage achieved when the test case are exercised.

A test coverage (or test completion) criterion must be established for unit testing, because program units usually contain too many paths to permit exhaustive testing. This can be seen by the examining the program segment in Figure 10.5. As illustrated in Figure 10.5, loops introduce combinatorial numbers of execution paths and make exhaustive testing impossible.



N	P
0	2
1	4
2	8
10	2048

$P=2^{N+1}$

Figure 10.5: Unit Testing

Even if it were possible to successfully test all paths through a program, correctness would not be guaranteed by path testing because the program might have missing paths and computational errors that were not discovered by the particular test cases chosen. A missing path error occurs when a branching statement and the associated computations are accidentally omitted. Missing path errors can only be detected by functional test cases derived from the requirements specifications. Thus, tests based solely on the program structure cannot detect all the potential errors in a source program. Coincidental correctness occurs when a test case fails to detect a computation error. For instance, the expressions (A+A) and (A*A) have identical values when A has the value 2.

Integration Testing

Bottom-up integration is the traditional strategy to integrate the components of a software system into a functioning whole. Bottom-up integration consists of unit testing, followed by subsystem testing, followed by testing of the entire system. Unit testing has the goal of discovering errors in the individual modules of the system. Modules are tested in isolation from one another in an artificial environment known as a "test harness," which consists of the driver programs and data necessary to exercise the modules. Unit testing should be as exhaustive as possible to ensure that each representative handled by each module has been tested. Unit testing is eased by a system structure that is composed of small, loosely coupled modules.

A subsystem consists of several modules that communicate with each other through well-defined interfaces. Normally, a subsystem implements a major segment operation of the interfaces between modules in the subsystem. Both control and of subsystem testing: lower level subsystems are successively combined to form higher-level subsystems. In most software systems, exhaustive testing of subsystem capabilities is not feasible due to the combinational complexity of the module interfaces; therefore, test cases must be carefully chosen to exercise the interfaces in the desired manner.

System testing is concerned with subtleties in the interfaces, decision logic, control flow, recovery procedures, throughput, capacity, and timing characteristics of the entire system. Careful test planning is required to determine the extent and nature of system testing to be performed and to establish criteria by which the results will be evaluated.

Disadvantages of bottom-up testing include the necessity to write and debug test harness for the modules and subsystems, and the level of complexity that results from combining modules and subsystems into larger and larger units. The extreme case of complexity results when each module is unit tested in isolation and "big bang" approach to integration testing. The main problem with big-bang integration is the difficulty of isolating the sources of error.

Test harnesses provide data environments and calling sequences for the routines and subsystems that are being tested in isolation. Test harness preparation can amount to 50 per cent or more of the coding and debugging effort for a software product.

Top-down integration starts with the main routine and one or two immediately subordinate routines in the system structure. After this top-level, when "skeleton" has been thoroughly tested, it becomes the test harness for its immediately subordinate routines. Top-down integration requires the use of program stubs to simulate the effect of lower-level routines that are called by those being tested.

Regression Testing

When some errors occur in a program then these are rectified. For rectification of these errors, changes are made to the program. Due to these changes some other errors may be incorporated in the program. Therefore, all the previous test cases are tested again. This type of testing is called regression testing.

In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that supports it) is changed. Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools. Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

A representative sample of tests that will exercise all software functions.

Additional tests that focus on software functions that are likely to be affected by the change.

Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large.

Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

10.9 BLACK BOX AND WHITE BOX TESTING

Black box testing takes an external perspective of the test object to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid inputs and determines the correct output. There is no knowledge of the test object's internal structure.

This method of test design is applicable to all levels of software testing: unit, integration, functional testing, system and acceptance. The higher the level, and hence the bigger and more complex the box, the more one is forced to use black box testing to simplify. While this method can uncover unimplemented parts of the specification, one cannot be sure that all existent paths are tested.

White box testing (a.k.a. clear box testing, glass box testing, transparent box testing, or structural testing) uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software. The tester chooses test case inputs to exercise paths through the code and determines the appropriate outputs. In electrical hardware testing, every node in a circuit may be probed and measured; an example is in-circuit testing (ICT).

Since the tests are based on the actual implementation, if the implementation changes, the tests probably will need to change, too. For example ICT needs updates if component values change, and needs modified/new fixture if the circuit changes. This adds financial resistance to the change process, thus buggy products may stay buggy. Automated optical inspection (AOI) offers similar component level correctness checking without the cost of ICT fixtures, however changes still require test updates.

While white box testing is applicable at the unit, integration and system levels of the software testing process, it is typically applied to the unit. While it normally tests paths within a unit, it can also test paths between units during integration, and between subsystems during a system level test. Though this method of test design can uncover an overwhelming number of test cases, it might not detect unimplemented parts of the specification or missing requirements, but one can be sure that all paths through the test object are executed.

10.10 SUMMARY

- Testing is the activity through which compliance to specifications & quality is ensured. It is the process of executing a software with the intent of finding errors.
- The two basic approaches to test case design are, structural & functional.
- Test case generation cannot be fully automated
- Levels of testing include unit testing, integration testing, Repression testing
- Black Box testing tahes an external view of the test object to device test case.
- White box testing uses an internal purspective of the syste.

10.11 UNIT-END QUESTIONS

1. What are the important testing fundamentals?

2. What are top-down and bottom-up approaches of testing.
3. Describe the procedure of unit testing.
4. Describe the procedure of integration testing.
5. What do you understand by regression testing and where do we use it?
6. Define testing. List and describe various testing techniques briefly.
7. What are test cases? How these are generated in black box and white box testing? Describe through suitable examples.
8. Does simply presence of fault mean software failure? If no, justify your answer with proper example.
9. What are test oracles?
10. What is the objective of testing?
11. What do you understand by functional testing?
12. Describe different types of functional testing techniques.
13. How we can design a test case?
14. Name all types of testing techniques.
15. Differentiate between unit and integration testing.
16 What is the effect of psychology of testing?
17. What do you understand by structural testing?
18. Discuss test case generation.
19. Differentiate between test case and test plan.
20. What do you understand by test case execution and analysis?

10.12 FURTHER READINGS

1. Software Engineering and Testing: An Introduction, B. B. Agarwal, S. P. Tayal, M. Gupta, Laxmi Publications Pvt. Ltd., 2008.
2. www.wikipedia.org

UNIT - XI

QUALITY MANAGEMENT

Structure of the Unit

11.0 OBJECTIVES

After reading this chapter, the students will be able to understand,

- Software quality
- Quality assurance
- Quality control
- Software reliability
- Software quality factors
- Software validation
- Software maintenance
- Software configuration management

11.1 INTRODUCTION

The word "Quality" has various meanings.

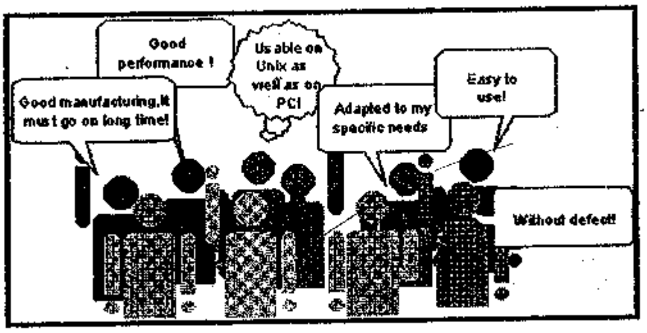

Good performance !
Us able on Unix as well as on PC!
Easy to use!
Good manufacturing,it must go on long time!
Adapted to my specific needs
Without defect!

Figure 11.1 : Different Aspects of Quality

The definition given by the ISO/IEC 8402 standard is:

"The totality of features and characteristics of a product or a service that bear on its ability to satisfy stated or implied needs". Software quality can not be specified only as software without error. The software quality specification must be more accurate and detailed. The formalisation of the software quality can be done using a quality model.

In 1977, McCall and his colleagues proposed a quality model to specify software quality. This model is based on three aspects of a software product. These are product operation, product revision, and product transition.

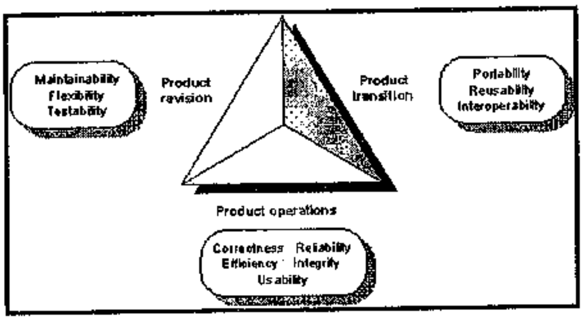


Figure 11.2: McCall's Quality Factors

The McCall quality model is organized around three types of Quality Characteristics:

- Factors (To specify): Describe the external view of the software, as viewed by the users.
- Criteria (To build): Describe the internal view of the software, as seen by the developer.
- Metrics (To control): Are defined and used to provide a scale and method for measurement.

Since then, various quality models have been defined, adopted and enhanced over the years for example those proposed by Boehm or Forse.

The need for one recognized standard quality model became more and more urgent. The ISO/IEC 9126 standard is the result of a consensus for a software quality model. As with McCall's this is also based on three levels:

- Characteristics (Functionality, Reliability, Usability, Efficiency, Maintainability, Portability);
- Sub-characteristics;
- . Metrics.

Each characteristic is refined to a set of sub-characteristics and each sub-characteristic is evaluated by a set of metrics. Some metrics are common to several sub-characteristics.

There is not a single formal method or technique to specify the metrics. The standardization process is ongoing by the ISO project "Evaluation and Metrics". The specification of metrics is not an easy task. Two problems have to be resolved. The first is to choose the right metric from amongst the large amount of existing ones. The second is to implement the data collection. The efficiency of a metric, i.e. the relevance of the results to cost, is an important parameter in the choice.

The quantification of metrics reduces the subjectivity in the software evaluation, even if the analysis of results is still dependant on the skill and the experience of experts. The results of measurement are values on the scales of metrics (a range 1 to10, a binary response Yes/No, a rate...). For a measurement value, a rating level is required. The ISO/IEC 9126 proposes four rating levels: Excellent, Good, Fair and Poor. The first three are considered as satisfactory, the last is considered as unsatisfactory.

Metrics are also used for maintainability evaluation. These metrics are of various types & measure software textual complexity, data flow complexity & also inheritance complexity for object oriented languages.

The metrics used to evaluate maintainability could be the following:

Analyzability:

- cyclomatic number
- number of statements
- comments rate
- calling proof

Changeability:

- number of jump
- number of nested levels
- average size of statement
- number of variables

Stability:

- number of parameters referenced
- number of global variables
- number of parameters changed
- number of called relationships

Testability:

- number of non-cyclic path
- number of nested levels
- cyclomatic number
- number of call-paths

The actual implementation of each metric depends on the programming language.

11.2 QUALITY ASSURANCE

Quality assurance, or QA for short, refers to a program for the systematic monitoring and evaluation of the various aspects of a project, service, or facility to ensu.e that standards of quality are being met.

It is important to realize also that quality is determined by the program sponsor. QA cannot absolutely guarantee the production of quality products, unfortunately, but makes this more likely.

Two key principles characterise QA: "fit for purpose" (the product should be suitable for the intended purpose) and "right first time" (mistakes should be eliminated). QA includes regulation of the quality of raw materials, assemblies, products and components; services related to production; and management, production and inspection processes.

It is important to realize also that quality is determined by the intended users, clients or customers, not by society in general: it is not the same as 'expensive' or 'high quality'. Even goods with low prices can be considered quality items if they meet a market need.

Quality assurance versus quality control

Quality control emphasizes testing of products to uncover defects, and reporting to management who make the decision to allow or deny the release, whereas quality assurance attempts to improve and stabilize production, and associated processes, to avoid, or at least minimize, issues that led to the defects in the first place.

To prevent mistakes from arising, several QA methodologies are used. However, QA does not eliminate the need for QC: some product parameters are so critical that testing is still essential. QC activities are treated as an one of the overall QA processes.

Failure testing

A valuable process to perform on a whole consumer product is failure testing or stress testing. In mechanical terms this is the operation of a product until it fails, often under stresses such as increasing vibration, temperature, and humidity. This exposes many unanticipated weaknesses in a product, and the data are used to drive engineering and manufacturing process improvements. Often quite simple changes can dramatically improve product service.

Statistical control

Many organizations use statistical process control to bring the organization to Six Sigma levels of quality so that the likelihood of an unexpected failure is confined to six standard deviations on the normal distribution. This probability is less than four one-millionths. Items controlled often include clerical tasks such as order-entry as well as conventional manufacturing tasks.

Traditional statistical process controls in manufacturing operations usually proceed by randomly sampling and testing a fraction of the output. Variances in critical tolerances are continuously tracked and where necessary corrected before poor quality components are produced.

Total quality management

The Quality of output is directly dependent upon that of the participating constituents, some of which are sustainably and effectively controlled while others are not. The fluid state spells lack of Quality control, and the process(es) which are properly managed for Quality such that Quality is assured, pertain to Total Quality Management.

The major characteristics to needed to ensume effective quality management are, Reliability, Maintainability, Safety, and Strength.

11.3 QA IN SOFTWARE DEVELOPMENT

The following are examples of QA models relating to the software development process.

Models and Standards

ISO 17025 is an international standard that specifies the general requirements for the competence to carry out tests and or calibrations. There are 15 management requirements and 10 technical requirements. These requirements outline what a laboratory must do to become accredited. Management system refers to the organization's structure for managing its processes or activities that transform inputs of resources into a product or service which meets the organization's objectives, such as satisfying the customer's quality requirements, complying with regulations, or meeting environmental objectives.

The CMMI (Capability Maturity Model Integration) model is widely used to implement Quality Assurance (PPQA) in an organization. The CMMI maturity levels can be divided in to 5 steps, which a company can achieve by performing specific activities within the organization.

The company-wide quality approach places an emphasis on three aspects :

1. Elements such as controls, job management, defined and well managed processes, performance and integrity criteria, and identification of records
2. Competence, such as knowledge, skills, experience, and qualifications
3. Soft elements, such as personnel integrity, confidence, organizational culture, motivation, team spirit, and quality relationships.

The quality of the outputs is at risk if any of these three aspects is deficient in any way.

In the context of software engineering, software quality measures how well software is designed (quality of design), and how well the software conforms to that design (quality of conformance), although there are several different definitions including.

i) One of the challenges of Software Quality is that "everyone feels they understand it".

ii) A definition in Steve McConnell's Code Complete divides software into two pieces: internal and external quality characteristics. External quality characteristics are those parts of a product that face its users, where internal quality characteristics are those that do not.

iii) Another definition by Dr. Tom DeMarco says "a product's quality is a function of how much it changes the world for the better". This can be interpreted as meaning that user satisfaction is more important than anything in determining software quality.

Source code quality

A computer has no concept of "well-written" source code. However, from a human point of view source code can be written in a way that has an effect on the effort needed to comprehend its behavior. Many source code programming style guides, which often stress readability and usually language-specific conventions are aimed at reducing the cost of source code maintenance. Some of the issues that affect code quality include:

- Readability
- Ease of maintenance, testing, debugging, fixing, modification and portability
- Low complexity
- Low resource consumption: memory, CPU
- Number of compilation or lint warnings
- Robust input validation and error handling, established by software fault injection

Software Reliability

Software reliability is an important facet of software quality. It is defined as "the probability of failure-free operation of a computer program in a specified environment for a specified time".

One of reliability's distinguishing characteristics is that it is objective, measurable, and can be estimated, whereas much of software quality is subjective criteria. This distinction is especially important in the discipline of Software Quality Assurance. These measured criteria are typically called software metrics.

The Goal of Reliability

The need for a means to objectively determine software quality comes from the desire to apply the techniques of contemporary engineering fields to the development of software. That desire is a result of the common observation, by both lay-persons and specialists, that computer software does not work the way it ought to. In other words, software is seen to exhibit undesirable behaviour, up to and including outright failure, with consequences for the data which is processed, the machinery on which the software runs, and by extension the people and materials which those machines might negatively affect. The more critical the application of the software to economic and production processes, or to life-sustaining systems, the more important is the need to assess the software's reliability.

Regardless of the criticality of any single software application, it is also more and more frequently observed that software has penetrated deeply into most every aspect of modern life through the technology we use. It is only expected that this infiltration will continue, along with an accompanying dependency on the software by the systems which maintain our society. As software becomes more and more crucial to the operation of the systems on which we depend, the argument goes, it only follows that the software should offer a concomitant level of dependability. In other words, the software should behave in the way it is intended, or even better, in the way it should.

The Challenge of Reliability

The circular logic of the preceding sentence is not accidental-it is meant to illustrate a fundamental problem in the issue of measuring software reliability, which is the difficulty of determining, in advance, exactly how the software is intended to operate. The problem seems to stem from a common conceptual error in the consideration of software, which is that software in some sense takes on a role which would otherwise be filled by a human being. This is a problem on two levels. Firstly, most modern software performs work which a human could never perform, especially at the high level of reliability that is often expected from software in comparison to humans. Secondly, software is fundamentally incapable of most of the mental capabilities of humans which separate them from mere mechanisms: qualities such as adaptability, general-purpose knowledge, a sense of conceptual and functional context, and common sense.

Nevertheless, most software programs could safely be considered to have a particular, even singular purpose. If the possibility can be allowed that said purpose can be well or even completely defined, it should present a means for at least considering objectively whether the software is, in fact, reliable, by comparing the expected outcome to the actual outcome of running the software in a given environment, with given data. Unfortunately, it is still not known whether it is possible to exhaustively determine either the expected outcome or the actual outcome of the entire set of possible environment and input data to a given program, without which it is probably impossible to determine the program's reliability with any certainty.

However, various attempts are in the works to attempt to rein in the vastness of the space of software's environmental and input variables, both for actual programs and theoretical descriptions of programs. Such attempts to improve software reliability can be applied at different stages of a program's development, in the case of real software. These stages principally include: requirements, design, programming, testing, and runtime evaluation. The study of theoretical software reliability is predominantly concerned with the concept of correctness, a mathematical field of computer science which is an outgrowth of language and automata theory.

Reliability in program Development

The reliability factors be considered in the Software Development process are,

a) Requirements

A program cannot be expected to work as desired if the developers of the program do not, in fact, know the program's desired behaviour in advance, or if they cannot at least determine its desired behaviour in parallel with development, in sufficient detail. What level of detail is considered sufficient is hotly debated. The idea of perfect detail is attractive, but may be impractical, if not actually impossible, in practice. This is because the desired behaviour tends to change as the possible range of the behaviour is determined through actual attempts, or more accurately, failed attempts, to achieve it.

Whether a program's desired behaviour can be successfully specified in advance is a moot point if the behaviour cannot be specified at all, and this is the focus of attempts to formalize the process of creating requirements for new software projects. In situ with the formalization effort is an attempt to help inform non-specialists, particularly non-programmers, who commission software projects without sufficient knowledge of what computer software is in fact capable. Communicating this knowledge is made more difficult by the fact that, as hinted above, even programmers cannot always know in advance what is actually possible for software in advance of trying.

b) Design

While requirements are meant to specify what a program should do, design is meant, at least at a high level, to specify how the program should do it. The usefulness of design is also questioned by some, but those who look to formalize the process of ensuring reliability often offer good software design processes as the most significant means to accomplish it. Software design usually involves the use of more abstract and general means of specifying the parts of the software and what they do. As such, it can be seen as a way to break a large program down into many smaller programs, such that those smaller pieces together do the work of the whole program.

The purposes of high-level design are as follows. It separates what are considered to be problems of architecture, or overall program concept and structure, from problems of actual coding, which solve problems of actual data processing. It applies additional constraints to the development process by narrowing the scope of the smaller software components, and thereby-it is hoped-removing variables which could increase the likelihood of programming errors. It provides a program template, including the specification of interfaces, which can be shared by different teams of developers working on disparate parts, such that they can know in advance how each of their contributions will interface with those of the other teams. Finally, and perhaps most controversially, it specifies the program independently of the implementation language or languages, thereby removing language-specific biases and limitations which would otherwise creep into the design, perhaps unwittingly on the part of programmer-designers.

c) Programming

The history of computer programming language development can often be best understood in the light of attempts to master the complexity of computer programs, which otherwise becomes more difficult to understand in proportion (perhaps exponentially) to the size of the programs. (Another way of looking at the evolution of programming languages is simply as a way of getting the computer to do more and more of the work, but this may be a different way of saying the same thing.) Lack of understanding of a program's overall structure and functionality is a sure way to fail to detect errors in the program, and thus the use of better languages should, conversely, reduce the number of errors by enabling a better understanding.

Improvements in languages tend to provide incrementally what software design has attempted to do in one fell swoop: consider the software at ever greater levels of abstraction. Such inventions as statement, sub-routine, file, class, template, library, component and more have allowed the arrangement of a program's parts to be specified using abstractions such as layers, hierarchies and modules, which provide structure at different granularities, so that from any point of view the program's code can be imagined to be orderly and comprehensible.

In addition, improvements in languages have enabled more exact control over the shape and use of data elements, culminating in the abstract data type. These data types can be specified to a very fine degree, including how and when they are accessed, and even the state of the data before and after it is accessed..

d) **Software Build and Deployment**

Many programming languages such as C and Java require the program "source code" to be translated in to a form that can be executed by a computer. This translation is done by a program called a compiler. Additional operations may be involved to associate, bind, link or package files together in order to create a usable runtime configuration of the software application. The totality of the compiling and assembly process is generically called "building" the software.

The software build is critical to software quality because if any of the generated files are incorrect the software build is likely to fail. And, if the incorrect version of a program is inadvertently used, then testing can lead to false results.

Software builds are typically done in work area unrelated to the runtime area, such as the application server. For this reason, a deployment step is needed to physically transfer the software build products to the runtime area. The deployment procedure may also involve technical parameters, which, if set incorrectly, can also prevent software testing from beginning. For example, a Java application server may have options for parent-first or parent-last class loading. Using the incorrect parameter can cause the application to fail to execute on the application server.

The technical activities supporting software quality including build, deployment, change control and reporting are collectively known as Software configuration management. A number of software tools have arisen to help meet the challenges of configuration management including file control tools and build control tools.

e) **Testing**

Software testing, when done correctly, can increase overall software quality of conformance by testing that the product conforms to its requirements. Testing includes, but is not limited to:

1. Unit Testing
2. Functional Testing
3. Regression Testing
4. Performance Testing
5. Failover Testing
6. Usability Testing

A number of agile methodologies use testing early in the development cycle to ensure quality in their products. For example, the test-driven development practice, where tests are written before the code they will test, is used in Extreme Programming to ensure quality.

11.4 SOFTWARE QUALITY FACTORS

A software quality factor is a non-functional requirement for a software program which is not called up by the customer's contract, but nevertheless is a desirable requirement which enhances the quality of the software program. Note that none of these factors are binary; that is, they are not"either you have it or you don't" traits. Rather, they are characteristics that one seeks to maximize in one's software to optimize its quality. So rather than asking whether a software product "has" factor s, ask instead the degree to which it does (or does not).

Some software quality factors are listed here:

i **Understandability-clarity of purpose.** This goes further than just a statement of purpose; all of the design and user documentation must be clearly written so that it is easily understandable. This is obviously subjective in that the user context must be taken into account: for instance, if the software product is to be used by software engineers it is not required to be understandable to the layman.

ii **Completeness-presence** of all constituent parts, with each part fully developed. This means that if the code calls a subroutine from an external library, the software package must provide reference to that library and all required parameters must be passed. All required input data must also be available.

iii **Conciseness**-minimization of excessive or redundant information or processing. This is important where memory capacity is limited, and it is generally considered good practice to keep lines of code to a minimum. It can be improved by replacing repeated functionality by one subroutine or function which achieves that functionality. It also applies to documents.

iv **Portability**-ability to be run well and easily on multiple computer configurations. Portability can mean both between different hardware-such as running on a PC as well as a smartphone-and between different operating systems-such as running on both Mac OS X and GNU/Linux.

v **Consistency-uniformity in notation**, symbology, appearance, and terminology within itself.

vi **Maintainability**-propensity to facilitate updates to satisfy new requirements. Thus the software product that is maintainable should be well-documented, should not be complex, and should have spare capacity for memory, storage and processor utilization and other resources.

vii **Testability**-disposition to support acceptance criteria and evaluation of performance. Such a characteristic must be built-in during the design phase if the product is to be easily testable; a complex design leads to poor testability.

viii **Usability**-convenience and practicality of use. This is affected by such things as the human-computer interface. The component of the software that has most impact on this is the user interface (UI), which for best usability is usually graphical (i.e. a GUI).

ix **Reliability**-ability to be expected to perform its intended functions satisfactorily. This implies a time factor in that a reliable product is expected to perform correctly over a period of time. It also encompasses environmental considerations in that the product is required to perform correctly in whatever conditions it finds itself

x **Structuredness**-organisation of constituent parts in a definite pattern. A software product written in a block-structured language such as Pascal will satisfy this characteristic.

xi **Efficiency**-fulfillment of purpose without waste of resources, such as memory, space and processor utilization, network bandwidth, time, etc.

xii **Security**-ability to protect data against unauthorized access and to withstand malicious or inadvertent interference with its operations. Besides the presence of appropriate security mechanisms such as authentication, access control and encryption, security also implies resilience in the face of malicious, intelligent and adaptive attackers.

11.5 SOFTWARE VERIFICATION AND VALIDATION

ISVV stands for Independent Software Verification and Validation. ISVV is targeted at safety-critical software systems and aims to increase the quality of software products, thereby reducing risks and costs through the operational life of the software. ISVV provides assurance that software performs to the specified level of confidence and within its designed parameters and defined requirements.

ISVV activities are performed by independent engineering teams, not involved in the software development process, to assess the processes and the resulting products. The ISVV team independency is performed at three different levels: financial, managerial and technical.

ISVV goes far beyond "traditional" verification and validation techniques, applied by development teams. While the latter aim to ensure that the software performs well against the nominal requirements, ISVV is focused on non-functional requirements such as robustness and reliability, and on conditions that can lead the software to fail. ISVV results and findings are fed back to the development teams for correction and improvement.

11.6 SOFTWARE MAINTENANCE

Software maintenance in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment .

The key software maintenance issues are both managerial and technical. Key management issues are: alignment with customer priorities, staffing, which organization does maintenance, estimating costs. Key technical issues are: limited understanding, impact analysis, testing, maintainability measurement.

11.7 SOFTWARE CONFIGURATION MANAGEMENT (SCM)

In software engineering, software configuration management (SCM) is the task of tracking and controlling changes in the software. Configuration management practices include revision control and the establishment of baselines.

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 thus becomes a matter of comparing different results and of analysing their differences. Traditional configuration management typically focused on controlled creation of relatively simple products. Now, implementers of SCM face the challenge of dealing with relatively minor increments under their own control, in the context of the complex system being developed.

Purposes

The goals of SCM generally are:

Configuration identification - Identifying configurations, configuration items and baselines.

- Configuration control - Implementing a controlled change process. This is usually achieved by setting up a change control board whose primary function is to approve or reject all change requests that are sent against any baseline.
- Configuration status accounting - Recording and reporting all the necessary information on the status of the development process.
- Configuration auditing - Ensuring that configurations contain all their intended parts and are sound with respect to their specifying documents, including requirements, architectural specifications and user manuals.
- Build management - Managing the process and tools used for builds.
- Process management - Ensuring adherence to the organization's 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 source.

11.8 SUMMARY

- Quality Management (QM) focuses not only of the product quality but also on the means to achieve it.
- QM has three major components, namely, quality control, quality assurance and quality improvement.
- Quality control is the process of revision of the quality of all factors involved in production.
- Quality assurance (QA) refers to a set of steps for systematic monitoring & evaluation of various aspects of a Software project ot ensure that standards of quality are met.
- Quality improvement includes product improvement & process improvement
- Software quality Assurance consists of ameans of monitoring the software engineering process & methods used to ensure quality.
- Software Configuration Management (SCM) is the task of tracking & controlling changes in the software.

11.9 UNIT-END QUESTIONS

1 Is it possible to assess the quality of software if the customer keeps changing what it is supposed to do?

2 Quality and reliability are related concepts but are fundamentally different in a number of ways. Discuss them.

3 Can a program be correct and still not be reliable? Explain.

4 Can a program be correct and still not exhibit good quality? Explain.

5 Why is the First Law of System Engineering true? How does it affect our perception of software engineering paradigms.

6 Assume that you're the manager of a small project. What baselines would you define for the project and how would you control them?

7 Do some research on object-oriented databases and write a paper that describes how they can be used in the context of SCM.

11.10 FURTHER READINGS

1. www.wikipedia.org
2. www.cse.dcu.ie/essiscope.sm2/charact.html

UNIT - XII

PROCESS IMPROVEMENT AND MEASUREMENT

Structure of the Unit

12.0 OBJECTIVE

After completing this unit students will be able to explain

- how software processes can be improved to produce better software
- understand the principles of software process improvement and why process improvement is worthwhile
- understand how software process factors influence software quality and the productivity of software developers;

- be able to develop simple models of software process;
- understand the notions of process capability and process maturity and the general form of the CMMI model for process improvement.

12.1 INTRODUCTION

Software engineering organizations are offered several different methods for process improvement. These include the Capability Maturity Model – Integration (CMMI), Six Sigma, Lean, and ISO Standard 15939 (SPICE). However, there is very less guidance about the relative advantages of the different methods or criteria for selecting among them. This unit will provide an overview and comparison of these approaches, then focus on the CMMI approach in more detail as a general framework for integrating the other approaches.

Most organizations have limited resources for investment in process improvement. It is therefore necessary for them to appreciate the importance of process imporvment & then to invest wisely and improve their return on investment.

12.2 PROCESS AND PRODUCT QUALITY IMPROVEMENT

Software processes are complex and involve a very large number of activities. Like products, processes also have attributes or characteristics as shown in figure. It is not possible to make process improvement that optimizes all process attributes simultaneously. For example, if the aim is to have a RAD (rapid application development process) then one may have to reduce the process visibility. Making a process visible means producing documents at regular intervals. This, slows down the process.

Process improvement does not simply mean adopting particular methods or tools or using some model of a process that has been used elsewhere. Although organizations that develop the same type of software clearly have much in common, there are always local organizational factors, procedures and standards that influence the process. One will rarely be successful in introducing process improvements if one simply attempts to change the process to one that is used somewhere else. Process improvement should always be seen as specific to an organization or a part of a larger organization.

Process improvement is a cyclical activity, as shown in figure 12.1. It involves three principal stages. These stages are:

1 **Process measurement:** Attributes of the current project or the product are measured. The aim is to improve the measure according to the goals of the organization involved in process improvement.

2 **Process analysis:** The current process is assessed, and process weaknesses and bottlenecks are indentified. Process models that describe the process are usually developed during this stage.

3 **Process change:** Changes to the process that have been indentified during analysis are introduced.

12.2.1 Process attributes

The attributes of a process that goven its effectiveness & results are described here.

Process characteristic	**Description**
Understandability	To what extent is the process explicitly defined and how easy is it to understand the process definition.

Visibility	Do the process activities culminate in clear results so that the progress is externally visible?
Supportability	To what extent can CASE tools be used to support the process activities?
Acceptability	Is the defined process acceptable to and usable by the engineers responsible for producing the software product?
Reliability	Is the process designed in such a way that process errors are avoided or trapped before they result in product error?
Robustness	Can the process evolve to reflect changing organizational requirements or identified process improvements?
Rapidity	How fast can the process of delivering a system from a given specification be completed?

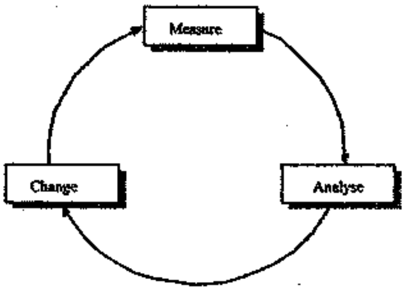


Figure 12.1: The Process Change cycle

12.2.2 Process and product quality

Process improvement is based on the assumption that the quality of the product development process is critical to product quality. The notion of process improvement was given by W.E. Deming.

Deming and others introduced the idea of statistical quality control. This is based on measuring the number of product defects and relating these defects to the process. The aim is to reduce the number of product defects by improving the process until it is repeatable. That is , the results of the process are predictable and the number of defects reduced. The process is then standardized and a further improvement cycle begins.

For small projects however where there are only a few team members, the quality of the development team is more important than the development process used. If the team has a high level of ability and experience the quality of the product is likely to be high. If the team is inexperienced and unskilled a good process may limit the damage but will not lead to high-quality software.

Where teams are small, good development technology is particularly important. A small team cannot devote a log of time to tedious administrative procedures. The team members spend much of their productivity. For large projects, a basic level of development technology is essential for information management. Paradoxically, sophisticated CASE tools are often less important large projects. Team members spend a smaller proportion of their time in development activities and more time communicating and understanding other parts of the system. This is the dominant factor affecting their productivity. Development tools make no difference to this.

12.2.3 Benefits of Process Improvement

The following are some of the benefits and business reasons for implementing process improvement:

- The quality of a system is highly influenced by the quality of the process used to acquire, develop, and maintain it.
- Process improvement increases product and service quality as organizations apply it to achieve their business objectives.
- Process improvement objectives are aligned with business objectives.

12.3 PROCESS CLASSIFICATION

The process can be observed in all organizations from small industry to large multinational company. These processes are of different types depending on the degree of the formality of the process, the size of product and the size of the company and so on. There are four classes of processes. Figure 12.2 illustrates examples of these:

1. **Informal Processes**: When there is no strictly defined model, the development team selects the process that they will use. Informal processes may use formal procedures such as configuration management , but the procedure and the relationships between procedures are defined as the required by development team.
2. **Managed Processes**: A defined process model is used to drive the development process. The process model defines the procedures, their scheduling and the relationships between the procedures.
3. **Methodical process** : When some defined development method or methods are used, these processes benefit from CASE tool support for design and analysis processes.
4. **Improving Processes**: Processes that have inharit improvement objectives have a specific budget for improvements and procedures for introducing such improvements. As part of this, quantitative process measurement may be introduced.

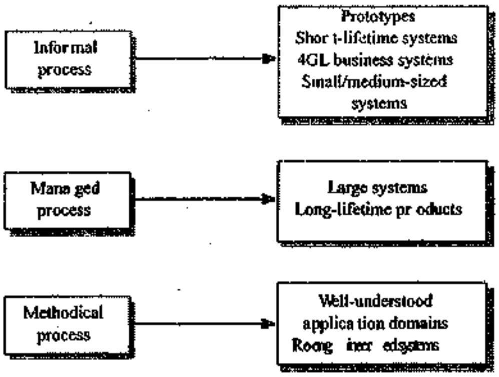


Figure 12.2 :Process Applicability

12.4 PROCESS MEASUREMENTS

Process measurements are quantitative data about the software process, measurements of process and product attributes is essential for process improvement. Process measurement can be used to access whether the efficiency of a process has been improved. For example the effort and time devoted to testing can be monitored.Effective improvements to the testing process should reduce the effort, testing time or both.

Process measurement can be used to improve the quality of a product. The product quality data should also be collected and related to the process activity.

Three classes of process metrics can be collected:

1. **The time taken for a particular process to be completed :** this can be a total time devoted to the process ,calendar time , the time spend on the process by particular engineers and so on.
2. **The resources required for a particular process:** The resources might include total effort in person-days, travel cost and computer change.
3. **The member of occurrences of a particular event :** The event might be monitored include the number of defects discovered code inspection, the number of lines of code modified in response to a requirement change.

12.5 PROCESS ANALYSIS AND MODELING

Process analysis and modeling involves studying existing processes and developing an abstract model of these processes that captures their key characters. These models help in understanding the processes and communicating that understanding to others.

Process analysis is concerned with studying existing process in order to know the relationships between part and process.

Following are the techniques of process analysis.

1. **Questionnaires and Interviews:** The engineers working on the project are questioned about what actually goes on. The answers to formal questionnaires are refined during personal interviews with those involved in the process. The discussion can be structured around a version of the process model that is refined as new information becomes available.
2. **Ethnographic studies :** This kind of study can used to know the nature of software development as a human activity. Such analysis reveals subtleties and complexities that may be discovered using other techniques.

12.6 PROCESS CHANGE

Process change involves making modification to the existing process. It may include introducing new practices, methods or tools and changes in the process activities.

There are five key stages in the process change process.

1. **Improvement Identification:** This stage is concerned with using the result of the process analysis to identify quality, schedule or cost bottlenecks where process factor might adversely influence the product quality. Process improvement should focus on loosening these bottlenecks by proposing new procedures, methods and tools to address the problems.
2. **Improvement Prioritisation:** This stage is concerned with assessing the possible changes and prioritizing them into implementation. When many possible changes have been identified, it is usually impossible to introduced them all at once.
3. **Process Change Introduction:** Process change introduction means putting new procedure, methods and tools into place, and integrating them with other process activities.
4. **Process Change Training:** Without training, it is not possible to gain the full benefits from process changes. Process managers and software engineers may simply refuse to accept the new process.

5. **Change Tuning:** Proposed changes will never be completely effective as soon as they are introduced. Tuning is need at the phase where minor problems are discovered and modifications to the process are proposed and are introduced. This tuning phase should last for several months until the development engineers are happy with the new process.

Once a change has been introduced, the improvement process can iterate with further analysis used to identify process problems, propose improvements, and so on. It is however, impractical to introduc too many changes. Introducing too many changes makes it impossible to access the effect of each change on the process.

12.7 CAPABILITY MATURITY MODEL INTEGRATION (CMMI)

Capability Maturity Model Integration (CMMI) is a process improvement approach that provides organizations with the essential elements of effective processes. It can be used to guide process improvement across a project, a division, or an entire organization. CMMI helps integrate traditionally separate organizational functions, set process improvement goals and priorities, provide guidance for quality processes, and provide a point of reference for appraising current processes.

12.7.1 Capability Maturity Model

The Capability Maturity Model (CMM) is a process capability maturity model which aids in the definition and understanding of an organisation's processes.

The CMM was originally described in the book Managing the Software Process (Addison Wesley Professional, Massachusetts, 1989) by Watts Humphrey. The CMM was conceived by Watts Humphrey, who based it on the earlier work of Phil Crosby. Active development of the model by the SEI (US Dept. of Defence Software Engineering Institute) began in 1986.

The CMM was originally intended as a tool for objectively assessing the ability of government contractors' processes to perform a contracted software project. Though it comes from the area of software development, it can be applied as a generally applicable model to assist in understanding the process capability maturity of organisations in diverse areas. For example, software engineering, system engineering, project management, risk management, system acquisition, information technology (IT), personnel management. It has been used extensively for avionics software and government projects around the world.

Though still thus widely used as a general tool, for software development purposes the CMM has been superseded by CMMI (Capability Maturity Model Integration).

- Maturity Model
- Structure of CMM
- Levels of the CMM

Maturity Model

A maturity model is a structured collection of elements that describe certain aspects of maturity in an organization. A maturity model may provide, for example:

- a place to start
- the benefit of a community's prior experiences
- a common language and a shared vision
- a framework for prioritizing actions
- a way to define what improvement means for your organization.

A maturity model can be used as a benchmark for assessing different organizations for equivalent comparison. The model describes the maturity of the company based upon the project the company is handling and the related clients.

12.7.2 Structure of CMM

The **CMM** involves the following aspects:

- *Maturity Levels:* It is a layered framework providing a progression to the discipline needed to engage in continuous improvement an organization develops the ability to assess the impact of a new practice, technology, or tool on their activity. Determining how innovative efforts influence existing practices is important when improvement is made. This really empowers projects, teams, and organizations by giving them the foundation to support reasoned choice.

- *Key Process Areas:* A Key Process Area (KPA) identifies a cluster of related activities that, when performed collectively, achieve a set of goals considered important.

- *Goals:* The goals of a key process area summarize the states that must exist for that key process area to have been implemented in an effective and lasting way. The extent to which the goals have been accomplished is an indicator of how much capability the organization has established at that maturity level. The goals signify the scope, boundaries, and intent of each key process area.

- *Common Features:* Common features include practices that implement and institutionalize a key process area. The five types of common features include: Commitment to Perform, Ability to Perform, Activities Performed, Measurement and Analysis, and Verifying Implementation.

- *Key Practices:* The key practices describe the elements of infrastructure and practice that contribute most effectively to the implementation and institutionalization of the key process areas.

12.7.3 Levels of the CMM

There are five levels of the CMM. According to the SEI, *"Predictability, effectiveness, and control of an organization's software processes are believed to improve as the organization moves up these five levels. While not rigorous, the empirical evidence to date supports this belief."*

Level 1 - Initial

At *maturity level 1*, processes are usually not documented and change based on the user or event. The organization does not have a stable environment and may not know or understand all of the components that make up the environment. As a result, success in these organizations depends on the institutional knowledge, the competence and heroics of the people in the organization, and the level of effort expended by the team. In spite of this chaotic environment, maturity level 1 organizations often produce products and services however, they frequently exceed the budget and schedule of their projects. Due to the lack of formality, level 1 organizations, often over-commit, abandon processes during a crisis, and are unable to repeat past successes. There is very little planning and executive buy-in for projects and process acceptance is limited. IT organizations at level 1 are often seen as a service instead of a partner.

Level 2 - Repeatable

At maturity level 2, some software development processes are repeatable, possibly with consistent results. The processes may not repeat for all the projects in the organization. The organization may use some basic project management to track cost and schedule.

Process discipline is unlikely to be rigorous, but where it exists it may help to ensure that existing practices are retained during times of stress. When these practices are in place, projects are performed and managed according to their documented plans.

Project status and the delivery of services are visible to management at defined points, for example, at major milestones and at the completion of major tasks.

Basic project management processes are established to track cost, schedule, and functionality. The minimum process discipline is in place to repeat earlier successes on projects with similar applications and scope. There is still a significant risk of exceeding cost and time estimates.

Level 3 - Defined

The organization's set of standard processes, which are the basis for level 3, are established and subject to some degree of improvement over time. These standard processes are used to establish consistency across the organization. Projects establish their defined processes by applying the organization's set of standard processes, tailored, if necessary, within similarly standardized guidelines.

The organization's management establishes process objectives for the organization's set of standard processes, and ensures that these objectives are appropriately addressed.

A critical distinction between level 2 and level 3 is the scope of standards, process descriptions, and procedures. At level 2, the standards, process descriptions, and procedures may be quite different in each specific instance of the process, for example, on each particular project. At level 3, the standards, process descriptions, and procedures for a project are tailored from the organization's set of standard processes to suit a particular project or organizational unit.

Level 4 - Managed

Using process metrics, management can effectively control the process e.g., for software development. In particular, management can identify ways to adjust and adapt the process to particular projects without measurable losses of quality or deviations from specifications. Organizations at this level set quantitative quality goals for both software process and software maintenance. Subprocesses are selected that significantly contribute to overall process performance. These selected subprocesses are controlled using statistical and other quantitative techniques. A critical distinction between maturity level 3 and maturity level 4 is the predictability of process performance. At maturity level 4, the performance of processes is controlled using statistical and other quantitative techniques, and may be quantitatively predictable. At maturity level 3, processes are only qualitatively predictable.

Level 5 - Optimizing

Maturity level 5 focuses on continually improving process performance through both incremental and innovative technological improvements. Quantitative process-improvement objectives for the organization are established, continually revised to reflect changing business objectives, and used as criteria in managing process improvement. The effects of deployed process improvements are measured and evaluated against the quantitative process-improvement objectives. Both the defined processes and the organization's set of standard processes are targets of measurable improvement activities.

Process improvements to address common causes of process variation and measurably improve the organization's processes are identified, evaluated, and deployed.

Optimizing processes that are nimble, adaptable and innovative depends on the participation of an empowered workforce aligned with the business values and objectives of the organization. The organization's ability to rapidly respond to changes and opportunities is enhanced by finding ways to accelerate and share learning.

A critical distinction between maturity level 4 and maturity level 5 is the type of process variation addressed. At maturity level 4, processes are concerned with addressing special causes of process variation and providing statistical predictability of the results. Though processes may produce predictable results, the results may be insufficient to achieve the established objectives. At maturity level 5, processes are concerned with addressing common causes of process variation and changing the process, that is, shifting the mean of the process performance to improve process performance, while maintaining statistical probability to achieve the established quantitative process-improvement objectives.

12.7.4 CMMI Benefits

The following are some of the benefits and business reasons for implementing process improvement:

The CMMI Product Suite is at the forefront of process improvement because it provides the latest best practices for product and service development and maintenance. The CMMI models improve the best practices of previous models in many important ways. CMMI best practices enable organizations to do the following:

- more explicitly link management and engineering activities to their business objectives
- expand the scope of and visibility into the product lifecycle and engineering activities to ensure that the product or service meets customer expectations
- incorporate lessons learned from additional areas of best practice (e.g., measurement, risk management, and supplier management)
- implement more robust high-maturity practices
- address additional organizational functions critical to their products and services
- more fully comply with relevant ISO standards.

12.8 GLOSSARY OF PRODUCT AND PROCESS QUALITY IMPROVEMENT TERMS

Common-Cause Variation: Any normal variation inherent in a work process.

Complexity: Unnecessary work; any activity that makes a work process more complicated without adding value to the resulting product or service.

Continuous Improvement Process: The ongoing enhancement of work processes for the benefit of the customer and the organization; activities devoted to maintaining and improving work process performance through small and gradual improvements as well as radical innovations.

Control Chart: A line graph that identifies the variation occurring in a work process over time; helps distinguish between common-cause variation and special-cause variation.

Cost of Quality: A term used by many organizations to quantify the costs associated with producing quality products. Typical factors taken into account are prevention costs (training, work process analyses, design reviews, customer surveys), appraisal costs (inspection and testing), and failure costs (rework, scrap, customer complaints, returns).

Cross Functional: Involving the cooperation of two or more departments within the organization (e.g., Marketing and Product Development).

Customer: Any person or group inside or outside the organization who receives a product or service.

Customer Expectations: The "needs" and "wants" of a customer that define "quality" in a specified product or service.

Deming Cycle (also known as Shewart's Wheel): A model that describes the cyclical interaction of research, sales, design, and production as a continuous work flow, so that all functions are involved constantly in the effort to provide products and services that satisfy customers and contribute to improved quality.

Department Improvement Team: Made up of all members of a department and usually chaired by the manager or supervisor, department improvement teams function as a vehicle for all employees to continuously participate in ongoing quality improvement activities.

Executive Steering Committee (or Executive Improvement Team): Includes top executives and is chaired by the CEO; encourages and participates in a quality initiative by reviewing, approving, and implementing improvement activities.

Fitness-For-Use: Juran's definition of quality suggesting that products and services need to serve customers' needs, instead of meeting internal requirements only.

Improving Steering Council (also known as Quality Steering Committee): A group of people with representation from all functions in the organization, usually drawn from management levels, chartered to develop and monitor a quality improvement process in their own functions. This group is often responsible for deciding which improvement projects or work processes will be addressed and in what priority.

Internal Customer: Anyone in the organization who relies on you for a product or service.

Internal Supplier: Anyone in the organization you rely on for a product or service.

Juran Trilogy: The interrelationship of three basic managerial processes with which to manage quality, quality control, and quality improvement.

Just-In-Time (JIT): A method of production and inventory cost control based on delivery of parts and supplies at the precise time they are needed in a production process.

Kaizen: Japanese term meaning continuous improvement involving everyone-managers and employees alike.

Key Expectations: The requirements concerning a specified product or service that a customer holds to be most important.

Quality Circle: A small group of employees organized to solve work-related problems often voluntarily usually not chaired by a department manager.

Quality Initiative: A formal effort by an organization to improve the quality of its products and services; usually involves top management development of a mission statement and long-term strategy.

Special-Cause Variation: Any violation arising from circumstances that are not a normal part of the work process.

Supplier: Any person or group inside or outside the organization that produces a product or service. Suppliers improve quality by identifying customer expectations and adjusting work processes so that products and services meet or exceed those expectations.

Task Force: An ad hoc, cross-functional team formed to resolve a major problem as quickly as possible usually includes subject matter experts temporarily relieved of their regular duties.

Total Quality Control (TQM): A management approach advocating the involvement of all employees in the continuous improvement process-not-just quality control specialists.

Work Partnership: A mutually beneficial work relationship between internal and external customers and suppliers.

Work Process: A series of work steps that produce a particular product or service for the