

**LECTURE NOTES ON
SOFTWARE ENGINEERING & OOAD
CODE: MCA -201**

By
Asst. Prof. Mrs. Mrs Etuari Oram
Asst. Prof. Mr Sanjib Kumar Nayak
Asst. Prof. Mr Bighnaraj Naik

SYLLABUS

Module I (10 Lectures)

Introductory concepts: Introduction, definition, objectives, Life cycle – Requirements analysis and specification.

Design and Analysis: Cohesion and coupling, Data flow oriented Design: Transform centered design, Transaction centered design. Analysis of specific systems likes Inventory control, Reservation system.

Module II (10 Lectures)

Object-oriented Design: Object modeling using UML, use case diagram, class diagram, interaction diagrams: activity diagram, unified development process.

Module III (10 Lectures)

Implementing and Testing: Programming language characteristics, fundamentals, languages, classes, coding style efficiency. Testing: Objectives, black box and white box testing, various testing strategies, Art of debugging. Maintenance, Reliability and Availability: Maintenance: Characteristics, controlling factors, maintenance tasks, side effects, preventive maintenance - Re Engineering - Reverse Engineering - configuration management - Maintenance tools and techniques. Reliability: Concepts, Errors, Faults, Repair and availability, reliability and availability models, Recent trends and developments.

Module IV (10 Lectures)

Software quality: SEI CMM and ISO-9001. Software reliability and fault-tolerance, software project planning, monitoring, and control. Computer-aided software engineering (CASE), Component model of software development, Software reuse.

Book:

1. Rajib Mall, Fundamentals of Software Engineering, PHI.
2. R.S. Pressman, Software Engineering Practitioner's Approach, TMH.
3. S.L. Pfleeger, Software Engineering - Theory and Practice, 2nd Edition, Pearson Education.
4. M.L. Shooman, Software Engineering - Design, Reliability and Management, McGraw Hill.

Contents

Module: I

Lecture 1: Introduction to Software Engineering

Lecture 2: Definition & Principles of Software Engg., Software Characteristics

Lecture 3: Causes & Solution of Software Crisis, Software Application and

processes.

Lecture 4: Software Life Cycles Methods and Description of Classical Water fall Model.

Lecture 5: Iterative water fall life cycle Model, Prototyping/ Rapid Prototyping Model and Spiral Model

Lecture 6: Software requirement & Specification

Lecture 7: Complex Logic: Decision tree & Decision Table, Specification of Complex Logic.

Lecture 8: Cohesion and Coupling

Lecture 9: Data Flow Oriented Design

Lecture 10: Transaction Analysis, Inventory Control System

Module II

Module III

Module IV

MODULE-I

Lecture Note: 1

Software:

Software is defined as a collection of programs, procedures, rules, data and associated documentation. The s/w is developed keeping in mind certain h/w and operating system consideration commonly known as platform. And engineering means systematic procedure to develop software. Some of the software characteristics are, it

can be engineer or developed and second thing is software is complex in nature. Important of software are due to much reason as it is used in:

i)Business decision making

Ex- accounting s/w, billing s/w

ii)For scientific research & engineering problem solving.

Ex-weather forecasting system, space research s/w

iii)It is embedded in multifunctional systems such as medical, telecom entertainment etc.

Ex-s/w for medical patient automation, s/w of GSM/CDMA service provides.

Software Quality

Several quality factors associated with software quality are as following:

- **Portability:** A software product is said to be portable, if it can be easily made to work in different operating system environments, in different machines, with other software products, etc.
- **Usability:** A software product has good usability, if different categories of users (i.e. both expert and novice users) can easily invoke the functions of the product.
- **Reusability:** A software product has good reusability, if different modules of the product can easily be reused to develop new products.
- **Correctness:** A software product is correct, if different requirements as specified in the SRS document have been correctly implemented.
- **Maintainability:** A software product is maintainable, if errors can be easily corrected, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

Types of software:-

Computer s/w is mainly divided into two types.

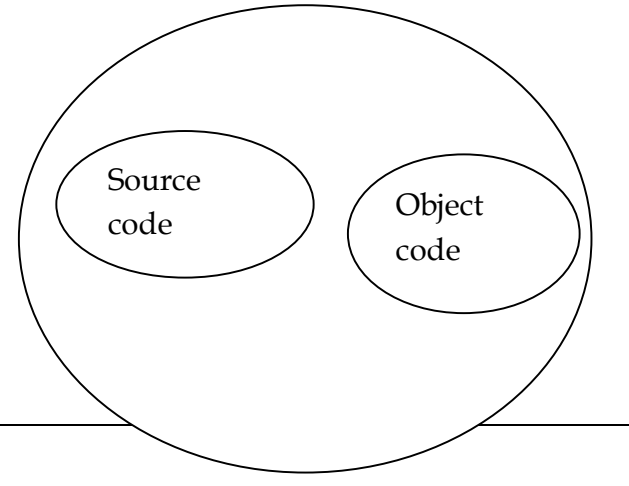
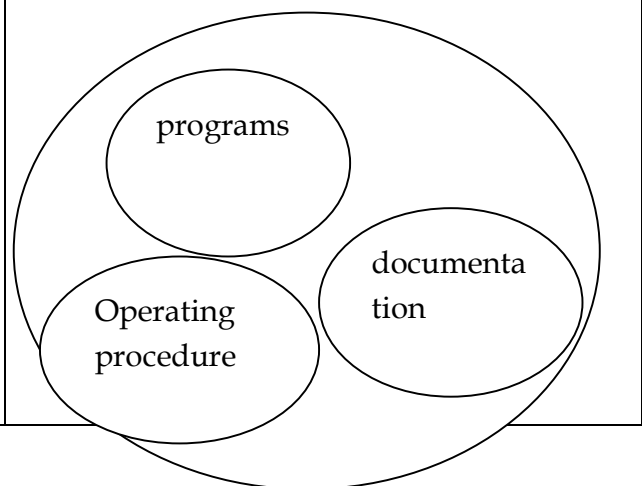
a)system s/w

Application s/w consists of programs to perform user oriented tasks. System s/w includes the operating system & all the utilities to enable the computer to run. Ex-window operating system

b)application s/w

Application s/w consists of programs to perform user oriented tasks. Ex-word processor, database management. Application s/w sits about the system s/w because it needs help of the system s/w to run.

Program vs Software product

PROGRAMS	PRODUCTS
Set of instruction related each other	Collection of program designed for specific task.
Programs are defined by individuals for their personal use.	A sum product is usually developed by a group of engineers working as a team.
Usually small size.	Usually large size.
Single user.	Large no of users.
Single developer.	Team of developer.
Lack proper documentation.	Good documentation support.
ADHOC development.	Systematic development.
Lack of UI.	Good UI.
Have limited functionality.	Exhibit more functionality.
	

Types of software products:

-*Generic products:* [This type of software product are developed by a organization and sold on open market to any customer], (System software,, application software)

-Customized (or bespoke) products: This type of software products are developed by a software contractor and especially for a customer.

-Embedded Product: Combination of both hardware and software

Software Engineering

Application of engineering for development of software is known as software engineering. It is the systematic, innovative technique and cost effective approach to develop software. And person involved in developing product is called software engineer. S/w engineer is a licensed professional engineer who is skilled in engineering discipline.

Qualities / Skills possessed by a good software engineer:

1. **General Skill** (Analytical skill, Problem solving skill, Group work skill)
2. **Programming Skill** (Programming language , Data structure , Algorithm , Tools(Compiler, Debugger))
3. **Communication skill** (Verbal , Written, Presentation)
4. **Design Skill** (s/w engineer must be familiar with several application domain)

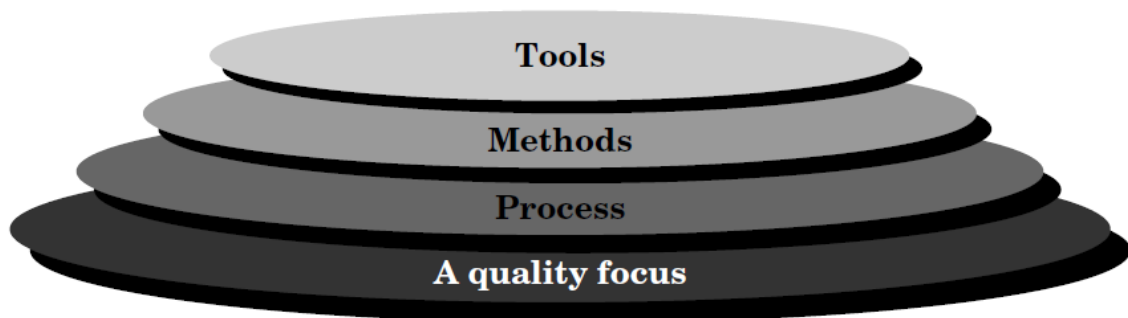
Lecture Note : 2

IEEE definition of Software engineering: A systematic, disciplined and quantifiable approach to the development, operation, maintenance and refinement of software.

Factor in emergence of software engineering:

1. People who are developing software were consistently wrong in their estimation of time, effort and cost.
2. Reliability and maintainability was difficult of achieved
3. Fixing bug in a delivered software was difficult
4. Delivered software frequently didn't work
5. Changes in ration of hw to s/w cost
6. Increased cost of software maintenance
7. Increased demand of software
8. Increased demand for large and more complex software system
9. Increasing size of software

S/W ENGINEERING PRINCIPLES:-



Software engineering is a layered technology. The bedrock that supports software engineering is a quality focus. The foundation for software engineering is the *process* layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of *key process areas* that must be established for effective delivery of software engineering technology. The key process areas form the basis for management control of software projects and establish the context in which technical methods are applied, work product (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

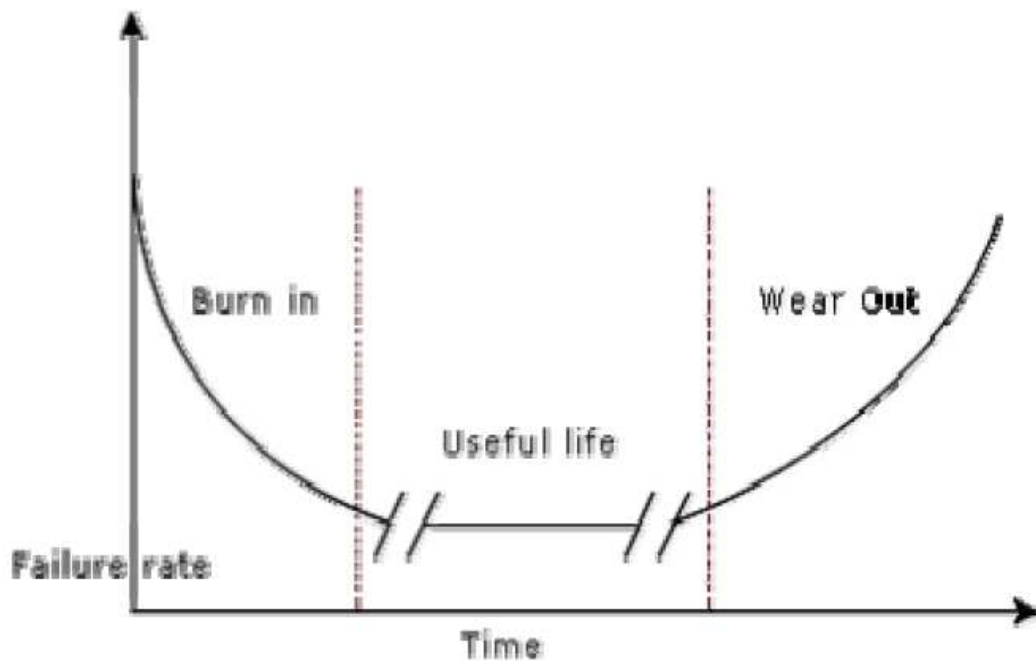
Software engineering *methods* provide the technical how-to's for building software. That encompass requirements analysis, design, program construction, testing, and support.

Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

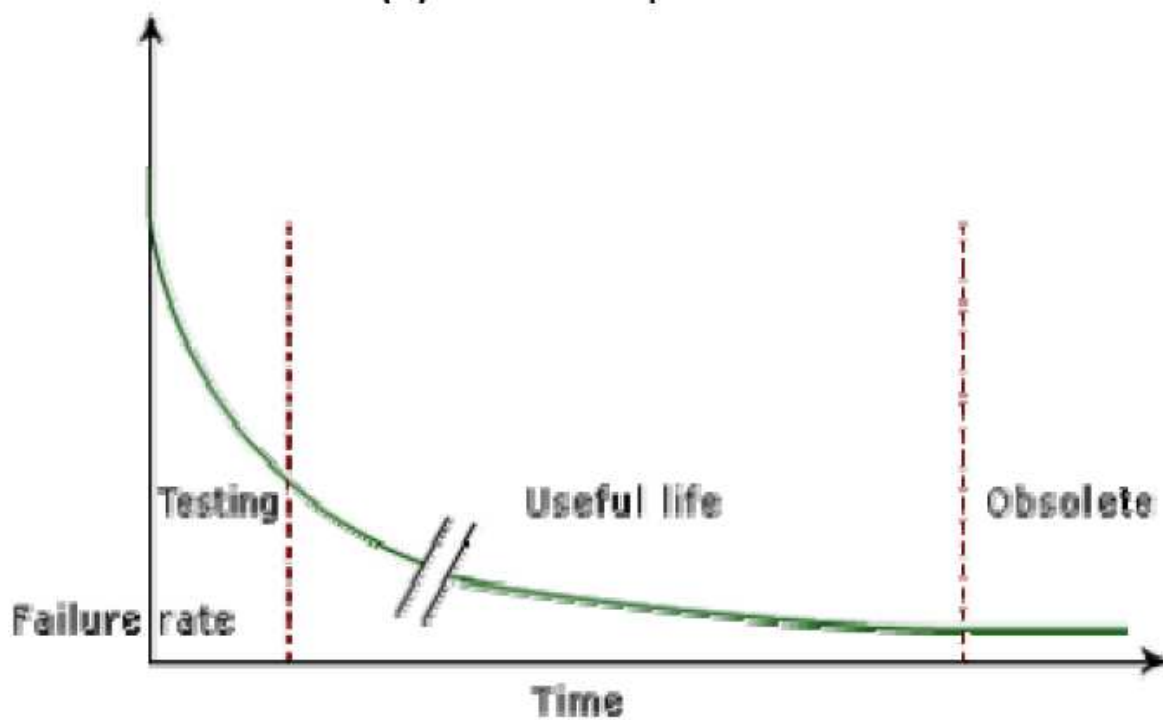
Software engineering *tools* provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering(CASE)*, is established. CASE combines software, hardware, and a software engineering database (a repository containing important information about analysis, design, program construction, and testing) to create a software engineering environment analogous to CAD/CAE (computer-aided design/engineering) for hardware.

S/W CHARACTERISTICS:-

Characteristics of a s/w can be easily distinguished as of from the h/w.



(a) Hardware product



(b) Software product

The change of failure rate over the product lifetime for a typical hardware and a software product are sketched in above fig.

For hardware products, it can be observed that failure rate is initially high but decreases as the faulty components are identified and removed. The system then enters its useful life.

After some time (called product life time) the components wear out, and the failure rate increases. This gives the plot of hardware reliability over time its characteristics is like "bath tub" shape.

On the other hand, for software the failure rate is at its highest during integration and test. As the system is tested, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no error corrections occurs and the failure rate remains unchanged.

Therefore after analyzing the facts we can write the key characteristics as follows:-

- a) Most s/w s are custom built rather than assembled from existing components.
- b) s/w is developed or engineered not manufactured.
- c) s/w is flexible.
- d) s/w does not wear out.

CAUSES AND SOLUTION FOR S/W CRISIS:-

Software engineering appears to be among the few options available to tackle the present software crisis.

Let us explain the present software crisis in simple words, by considering the following.

The expenses that organizations all around the world are incurring on software purchases compared to those on hardware purchases have been showing a worrying trend over the years (as shown in fig. 1.6)

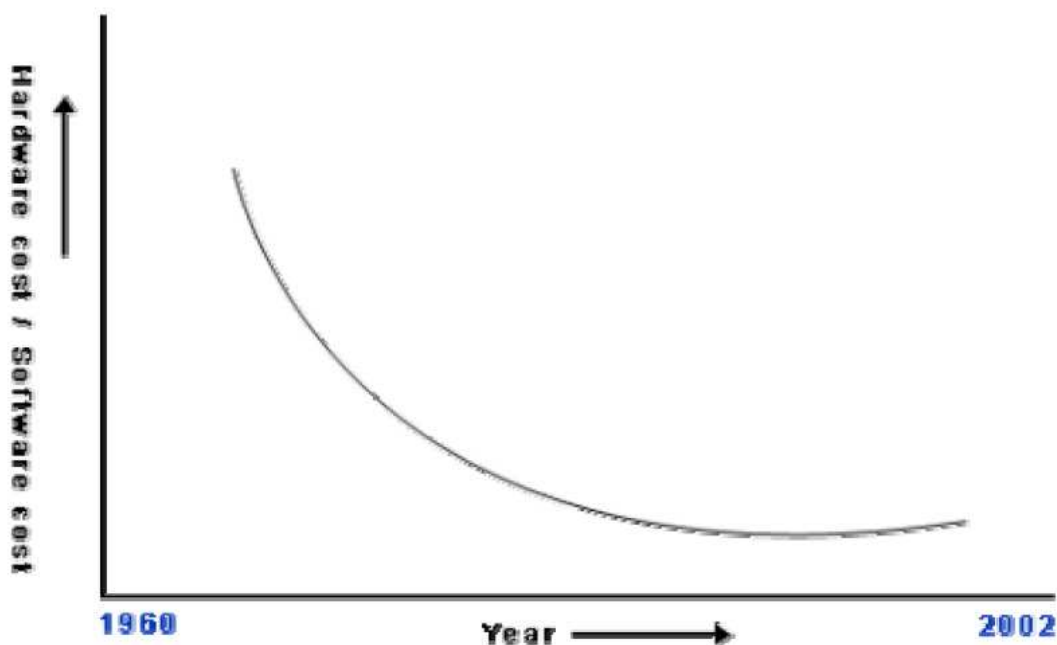


Fig. 1.6: Change in the relative cost of hardware and software over time

Organizations are spending larger and larger portions of their budget on software not only are the software products turning out to be more expensive than hardware, but also presented lots of other problems to the customers such as: software products are difficult to alter, debug, and enhance; use resources non optimally; often fail to meet the user requirements; are far from being reliable; frequently crash; and are often delivered late.

Due to ineffective development of the product characterized by inefficient resource usage and time and cost over-runs.

Other factors are larger problem sizes, lack of adequate training in software engineering, increasing skill shortage, and low productivity Improvements.

S/W crisis from programmer point of view:-

- i) Problem of compatibility.
- ii) Problem of Portability.
- iii) Proclaiming documentation.
- iv) Problem of pirated s/w.
- v) Problem in co-ordination of work of different people.
- vi) Problem of proper maintenance.

S/W crisis from user point of view:-

- i) s/w cost is very high.
- ii) Price of h/w grows down.
- iii) Lack of development specification.
- iv) Problem of different s/w versions.
- v) Problem of bugs or errors.

S/W Application:-

S/W applications can be grouped into 8 different areas as given below.

- i) System s/w
- ii) Real-time s/w
- iii) Embedded s/w
- iv) Business s/w
- v) PC s/w
- vi) AI s/w
- vii) Web-based s/w
- viii) Engineering & scientific s/w

S/W engineering processes:-

Process: The process is a series of states that involves activities, constraints, resources that produce an intended output of some kind. Or it is a state that takes some input and produces output.

Software process: A s/w process is a related set of activities & sub processes that are involved in developing & involving a s/w system. There are four fundamental process activities carried out by s/w engineering while executing the s/w process are .

- i) s/w specification
The functionality of s/w & constraints on its operation must be defined.
- ii) s/w development

The s/w that mixes the specification must be produced.

iii)s/w validation

The s/w must be validated to ensure that it performs desired customer activities.

iv)s/w evolution:-

The s/w must evolve to meet changing customer needs with time.

The scenario of s/w process & its sub process are shown below:-

The s/w industry considers the entire s/w development task as a process according to Booch & Rumbaugh. According to them a process defines who is doing what, when & how to reach a certain goal.

Generic attributes in a software process:

1. Understandability
2. Visibility
3. Reliability
4. Robustness
5. Adaptability
6. Rapidity
7. Maintainability
8. Supportability

Characteristic of a software process:

1. Understandability
2. Visibility
3. Reliability
4. Robustness
5. Adaptability
6. Rapidity
7. Maintainability
8. Supportability

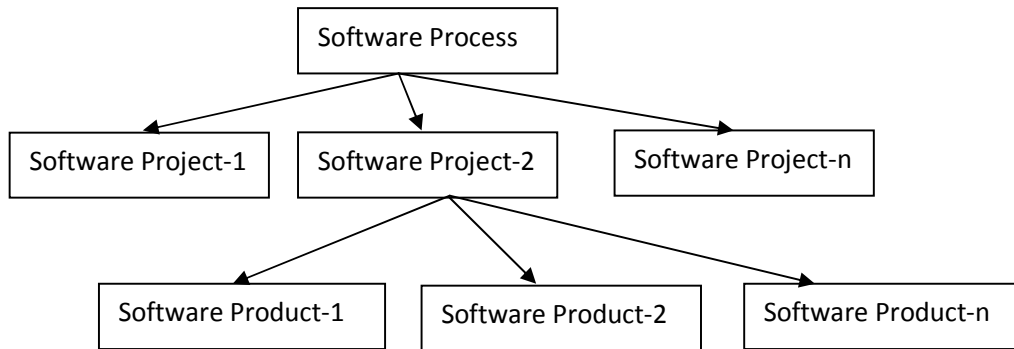
Software Project:

1. A project is a temporary endeavor undertaken to create a unique product.
2. Temporary means every project has a definite beginning and a definite end.
3. Unique means the product must be different in some ways from all similar product

Software Product:

Outcome of software project is known as software product.

Software process vs software project vs software product



S/W LIFE CYCLE METHOD:-

Introduction:-

A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle. A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken.

In other words, a life cycle model maps the different activities performed to develop software product from its inception to retirement. Different life cycle models may map the basic development activities to phases in different ways.

The need for s/w life cycle model:-

-The development team must identify a suitable life cycle model for the particular project and then adhere to it.

-Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner. So when a software product is being developed by a team there must be a clear understanding among team members about when and what to do, Otherwise it would lead to chaos and project failure.

-A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied. So without software life cycle model the entry and exit criteria for a phase cannot be recognized. Without software life cycle models (such as classical waterfall model, iterative waterfall model, prototyping model, evolutionary model, spiral model etc.) it becomes difficult for software project managers to monitor the progress of the project.

Different s/w life cycle model:-

Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

- a. Classical Waterfall Model
- b. Iterative Waterfall Model
- c. Prototyping Model
- d. Evolutionary Model
- e. Spiral Model

a. Classical Waterfall Model:

Different phases of the classic waterfall model:-

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it cannot be used in actual software development projects. Thus, this model can be considered to be a *theoretical way of developing software*.

Classical waterfall model divides the life cycle into the following phases as shown in fig.2.1:

- Feasibility Study
- Requirements Analysis and Specification
- Design
- Coding and Unit Testing
- Integration and System Testing
- Maintenance

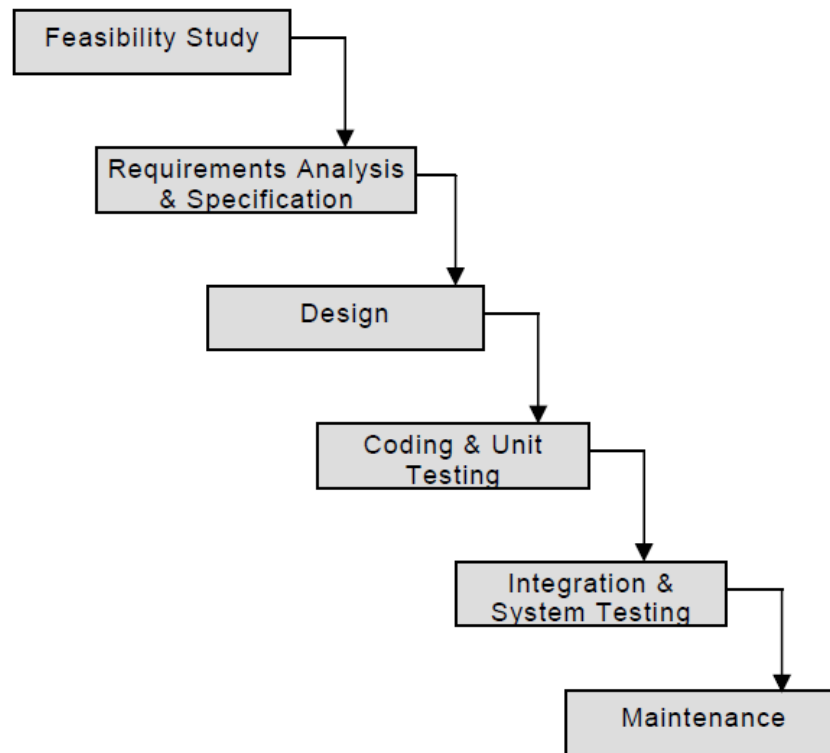


Fig 2.1: Classical Waterfall Model

Activities in each phase of the life cycle:

• Activities undertaken during feasibility study: -

The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product. Rough understanding between the team members about estimate based on the client side requirement. After over all discussion they search for variety of solution on the basis of kind of resources and time requirement etc.

Activities undertaken during requirements analysis and specification: -

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

- Requirements gathering and analysis, and
- Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.

-After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start.

-During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document.

Activities undertaken during design: -

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. During the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available:

- the traditional design approach and
- the object-oriented design approach.

Activities undertaken during coding and unit testing:-

The purpose of the coding and unit testing phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested then proceeds for next stage. During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation way as this is the most efficient way to debug the errors identified at this stage.

Activities undertaken during integration and system testing: -

Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:

α - testing: It is the system testing performed by the development team.

β - testing: It is the system testing performed by a friendly set of customers.

acceptance testing: It is the system testing performed by the customer himself after the product delivery to decide whether to accept or reject the delivered product.

Activities undertaken during maintenance: -

Maintenance of a typical software product requires much more effort than the effort necessary to develop the product itself. Maintenance involves performing any one or more of the following three kinds of activities:

- Correcting errors that were not discovered during the product development phase. This is called corrective maintenance.
- Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. It is called perfective maintenance.
- Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. It is called adaptive maintenance.

Disadvantages of waterfall model:-

- i)It can not handle satisfactorily different types of risks associated with real time s/w project, because, the requirements have to be pre decided by the client.
- ii)Most real life project can't follow the exact frame sequence of the waterfall model.

b. ITERATIVE WATERFALL LIFE CYCLE MODEL:-

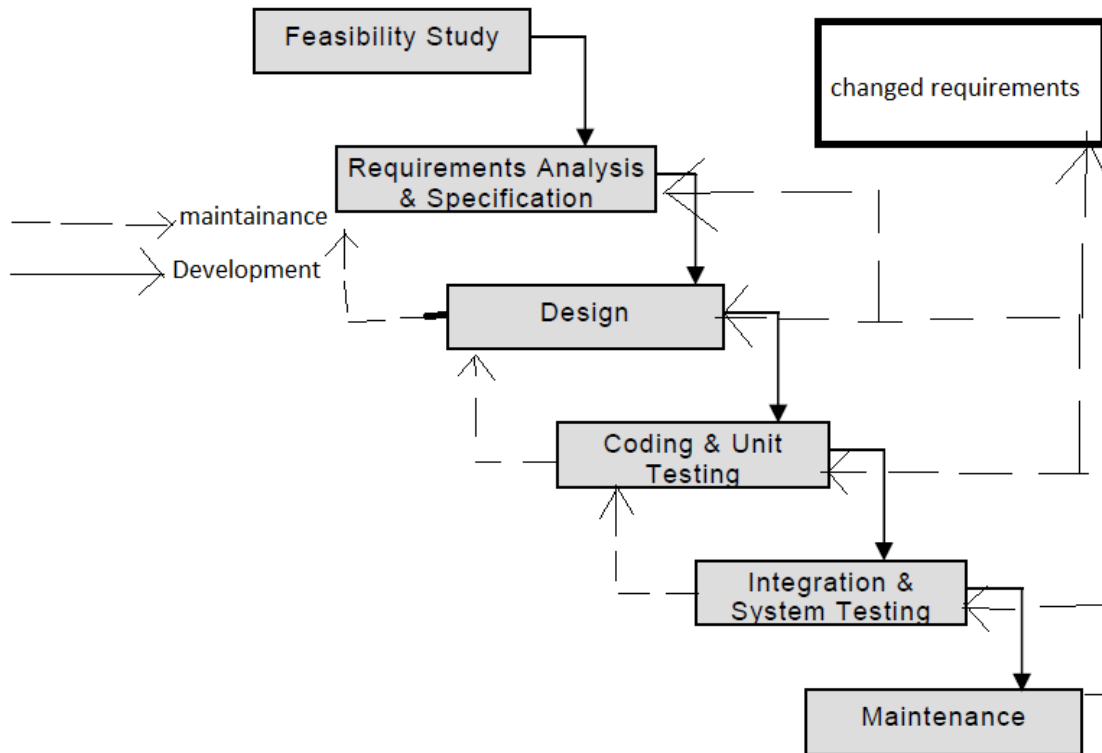
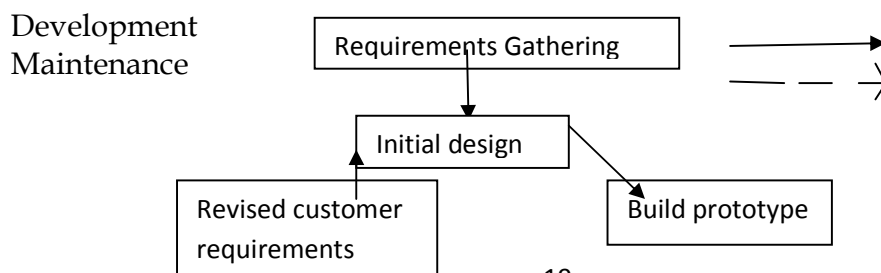


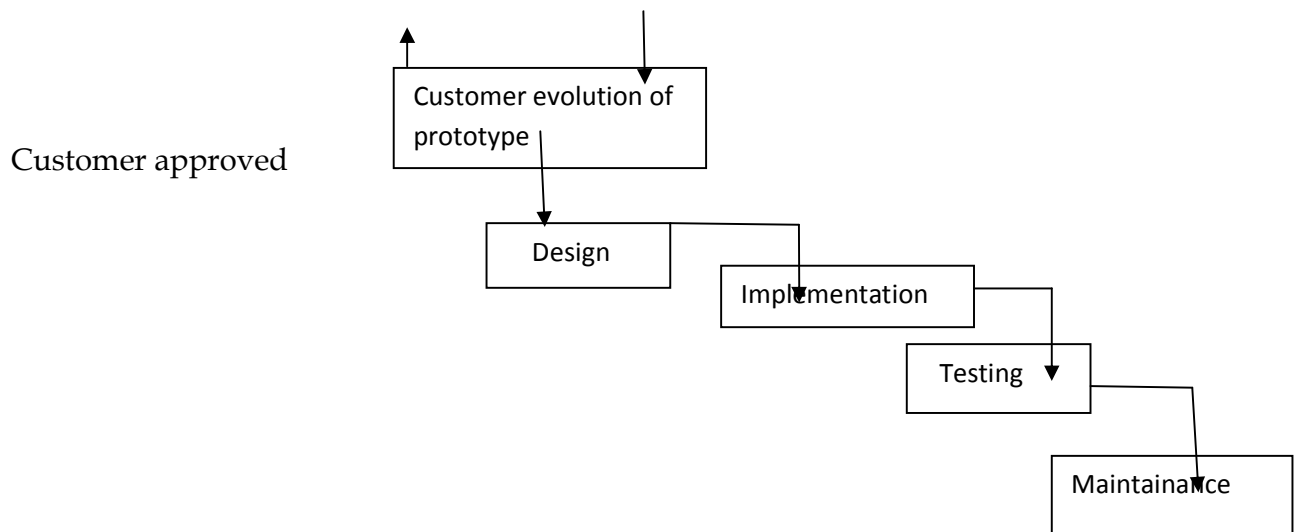
Fig: Basic iterative waterfall life cycle model

This waterfall model allows us to correct the error committed during the developmental phase.

The principle of detecting errors as close to their point of introduction as possible is known as phase containment of errors. These errors are not possible to be always detected at their point of occurrence. But, they must be detected as they are possible, because finding the errors at higher stage causes entire system testing hence cost is increased/high.

c. PROTOTYPING/RAPID PROTOTYPING MODEL:-





[Fig: Prototyping life cycle model]

This model specifies that before the actual system is being built the working model or the prototype system should be built in. Prototype means dummy, it is a modern and advanced implementation system with limited functional capabilities & is comparatively inefficient with the actual system.

Need for a prototype in software development

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs. The possibilities may be:

- how the screens might look like
- how the user interface would behave
- how the system would produce outputs

This is something similar to what the architectural designers of a building do; they show a prototype of the building to their customer. The customer can evaluate whether he likes it or not and the changes that he would need in the actual product. A similar thing happens in the case of a software product and its prototyping model.

Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.

A prototyping model can be used when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the

technical issues associated with the product development. Often, major design decisions depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.

Examples for prototype model

A prototype of the actual product is preferred in situations such as:

- user requirements are not complete
- technical issues are not clear

ADVANTAGES:-

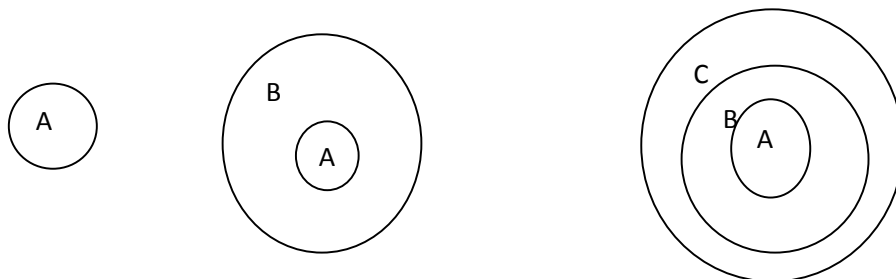
i)The prototype which is developed is generally discarded after the system is being built but the experience gain during the development of prototype helps the design issue as expected from the customer.

ii)Developing a prototype in place of actual system & putting it for customer reference resolves different technical & design issue as expected from the customer.

d. EVOLUTIONARY/INCREMENTAL LIFE CYCLE MODEL:-

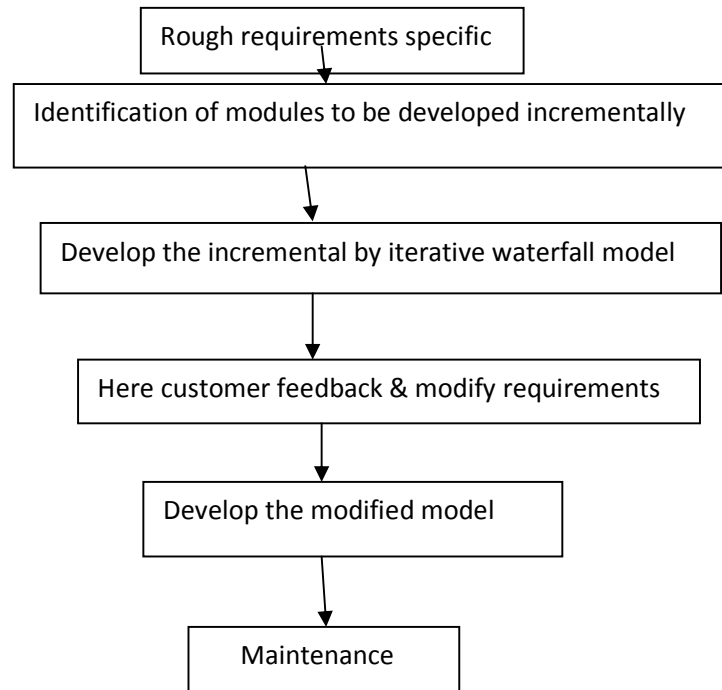
Evolutionary model is based upon principle of incremented developments of the functional unit & then integrating it for visible. Each incremental version of the product is a fully functional s/w product providing more features than its previous version. Each incremental version is tested individually before being integrated into the system. Development of incremental module do not need large amount of resources for project development as in the later case. This incremental model, they are developed in different phases can be useful to the customers as they can be used as standalone units when they are treated independently.

This evolutionary model can be described as follows:-



[A, B, C: Incremental modules developed & developed together]

This life cycle model can be defined as follows:



ADVANTAGES:-

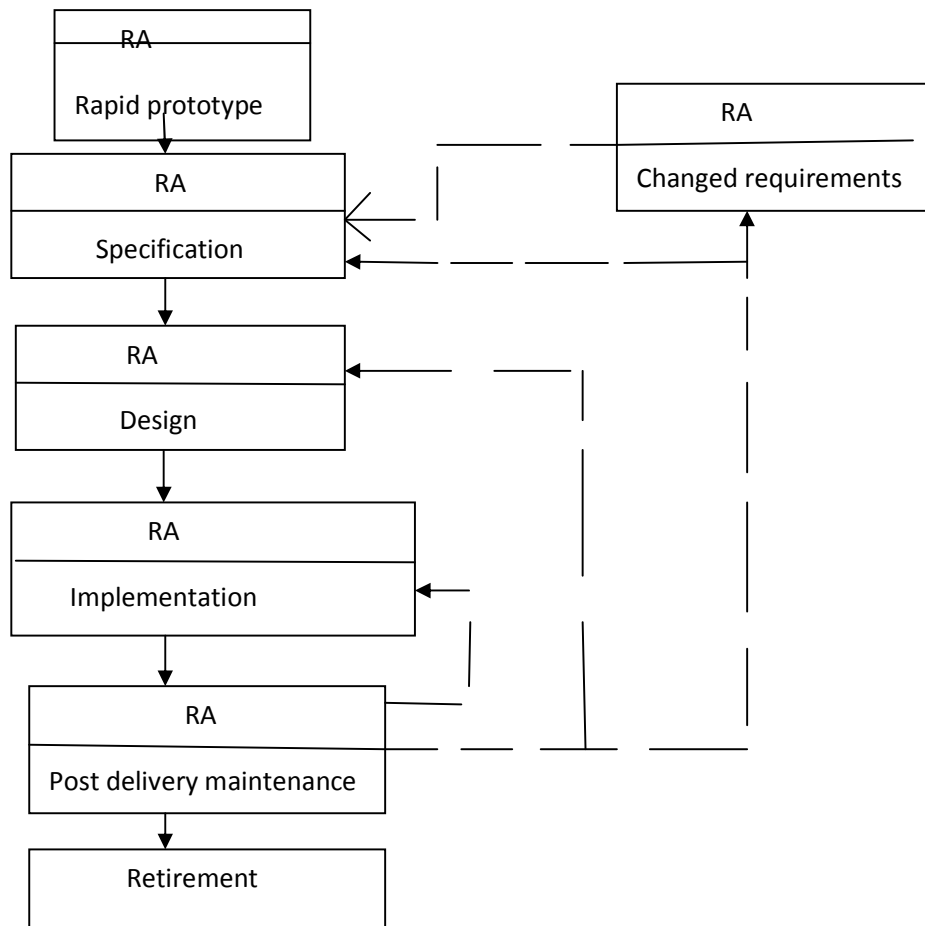
- i) Exact destination of customer requirements can be known.
- ii) It also reduces the maintenance tasks duration because before delivery of a module, the customers are consulted.
- iii) As the individual module takes place, so final product will contain limited amount of errors.
- iii) It can be built upon object oriented platform which promotes modular s/w development.

DISADVANTAGES:-

- i) It is only helpful for large s/w products because we can find individual modules for incremental implementation.
- ii) It is also used when the customer is ready to receive the product.

e. SPIRAL MODEL/RISK MANAGEMENT MODEL:-

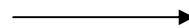
This model of s/w development can be described as follows:



[RA-Risk analysis]

----->maintenance

Development



This model contains the elements from all the other life cycle model along with the necessary risks involves in each of the developmental phase. So it is called as meta model.

The Spiral model of software development is shown in below fig. 2.2. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study. The next loop with be requirements specification, the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants)

as shown in fig. The activities are carried out during each phase of a spiral model are described below.

The radial dimension of the full spiral model provided by Bohem in 1988 to represent the cumulating cost along with the triangular dimension.

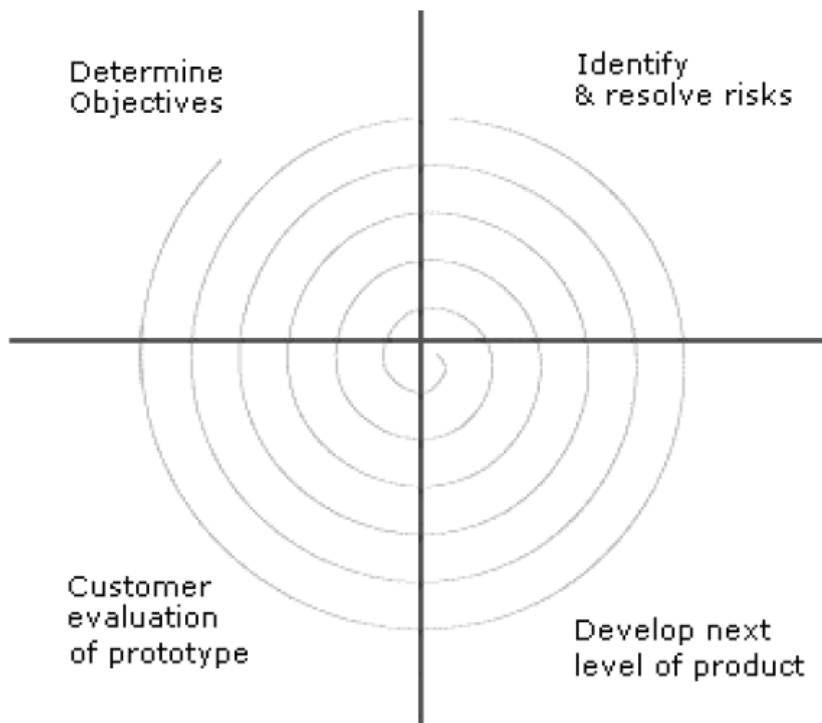


Fig. 2.2: Spiral Model

-First quadrant (Objective Setting)

- During the first quadrant, it is needed to identify the objectives of the phase.
- Examine the risks associated with these objectives.

- Second Quadrant (Risk Assessment and Reduction)

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

-Third Quadrant (Development and Validation)

- Develop and validate the next level of the product after resolving the identified risks.

- Fourth Quadrant (Review and Planning)

- Review the results achieved so far with the customer and plan the next iteration

around the spiral.

- Progressively more complete version of the software gets built with each iteration around the spiral.

Comparison of different life-cycle models:

The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model cannot be used in practical development projects, since this model supports no mechanism to handle the errors committed during any of the phases. This problem may be overcome in the iterative waterfall model. The iterative waterfall model is probably the most widely used software development model evolved so far. This model is simple to understand and use. However, this model is suitable only for well-understood problems; it is not suitable for very large projects and for projects that are subject to many risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood. It is especially popular for development of the user-interface part of the projects.

The evolutionary approach is suitable for large problems which can be decomposed into a set of different modules for incremental development and delivery of the product. This model is also widely used for object-oriented development projects. Of course, this model can only be used if the incremental delivery of the system is acceptable to the customer.

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models

Another important advantage of the incremental model is that it reduces the customer's trauma of getting used to an entirely new system. The gradual introduction of the product via incremental phases provides time to the customer to adjust to the new product. Also, from the customer's financial viewpoint, incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

S/W REQUIREMENT AND SPECIFICATION:-

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

- Requirements gathering and analysis, and
- Requirements specification

a) Requirements gathering and analysis:-

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed and analyzed.

This phase is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed. This activity begins by collecting all relevant data regarding the product to be developed from the user and from the customer through interviews and discussions.

For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore it is necessary to identify all ambiguities and contradictions in the requirements phase and resolve them through further discussions with the customer.

b) Requirements specification:-

After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document.

The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document.

The important components of this document are:

- functional requirements of the system
- nonfunctional requirements of the system
- goal of implementation.

Functional requirements:-

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high level functions $\{f_i\}$. The functional view of the system is shown in fig.3.1. Each function f_i of the system can be considered as a transformation of a set of input data (i_i) to the corresponding set of output data (o_i). The user can get some meaningful piece of work done using a high-level function.

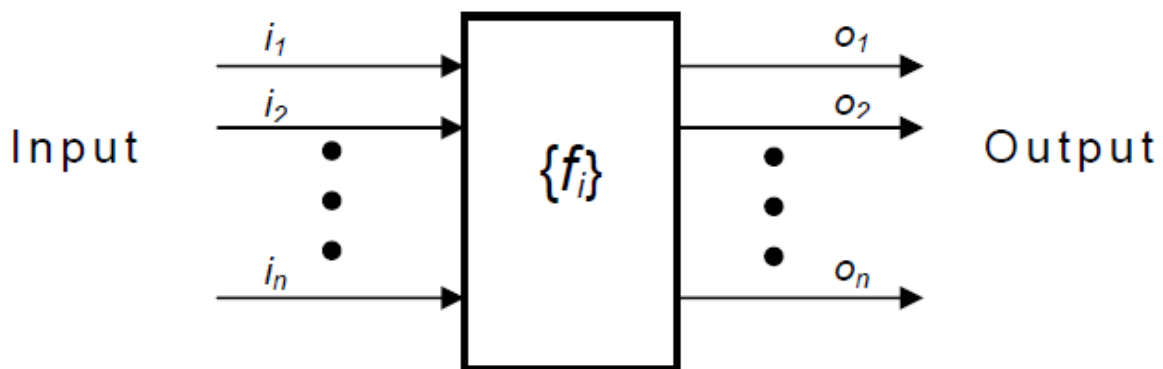


Fig. 3.1: View of a system performing a set of functions

Nonfunctional requirements:-

Nonfunctional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc. It may include:

- # reliability issues,
- # accuracy of results,
- # human - computer interface issues,
- # constraints on the system implementation, etc.

Goals of implementation:-

The goals of implementation part documents, some general suggestions regarding development such as guide trade-off among design goals, document issues as revision to system functionalities that may required in future, new devices to be supported in the future, reusability etc.

These are the issues which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

Properties of a good SRS document

- Some of the important properties of a good SRS document are the following:

a) Concise:

The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.

b) Structured:

It should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the customer requirements. Often, the customer requirements evolve over a period of time. Therefore, in order to make the modifications to the SRS document easy, it is important to make the document well-structured.

c) Black-box view:

It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS document should view the system to be developed as black box, and should specify the externally visible behavior of the system. For this reason, the SRS document is also called the black-box specification of a system.

d) Conceptual integrity:

It should show conceptual integrity so that the reader can easily understand it.

e) Response to undesired events:

It should characterize acceptable responses to undesired events. These are called system response to exceptional conditions.

f) Verifiable:

All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation.

Problems without a SRS document

- The important problems that an organization would face if it does not develop an SRS document are as follows:
 - Without developing the SRS document, the system would not be implemented according to customer needs.
 - Software developers would not know whether what they are

developing is what exactly required by the customer.

-Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system.

-It would be very much difficult for user document writers to write the users' manuals properly without understanding the SRS document.

Problems with an unstructured specification

- It would be very much difficult to understand that document.
- It would be very much difficult to modify that document.
- Conceptual integrity in that document would not be shown.
- The SRS document might be unambiguous and inconsistent.

TECHNIQUES FOR REPRESENTING COMPLEX LOGIC:-

Good SRS documents sometimes may have the conditions which are complex & which may have overlapping interactions & processing sequences. There are two main techniques available to analyze & represent complex processes logic are.

- A) Decision tree
- B) Decision table

After these two techniques are finalizing the decision making logic is captured in form of tree or table. We can automatically find out the test cases to validate the decision from the decision making logic. These two techniques are widely used in applications of information theory & switching theory.

Decision tree:

A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken. The edges of a decision tree represent conditions and the leaf nodes represent the actions to be performed depending on the outcome of testing the condition.

Example: -

Consider Library Membership Automation Software (LMS) where it should support the following three options:

- New member to join
- Renewal books
- Cancel membership
- New member option-

Decision:

When the 'new member' option is selected, the software asks details about the member like the member's name, address, phone number, id etc.

Action: If proper information is entered then a membership record for the member is created and a bill is printed for the annual membership charge plus the security deposit payable.

Renewal option-

Decision: If the 'renewal' option is chosen, the LMS asks for the member's name and membership number to check whether he is a valid member or not.

Action: If the membership is valid then membership expiry date is updated and the annual membership bill is printed, otherwise an error message is displayed.

Cancel membership option-

Decision: If the 'cancel membership' option is selected, then the software asks for member's name and his membership number.

Action: The membership is cancelled, a cheque for the balance amount due to the member is printed and finally the membership record is deleted from the database.

Decision tree representation of the above example -

The following tree (fig. 3.4) shows the graphical representation of the above example. After getting information from the user, the system makes a decision and then performs the corresponding actions. Ask for member’s name, address, etc.
Create membership details Print cheque.

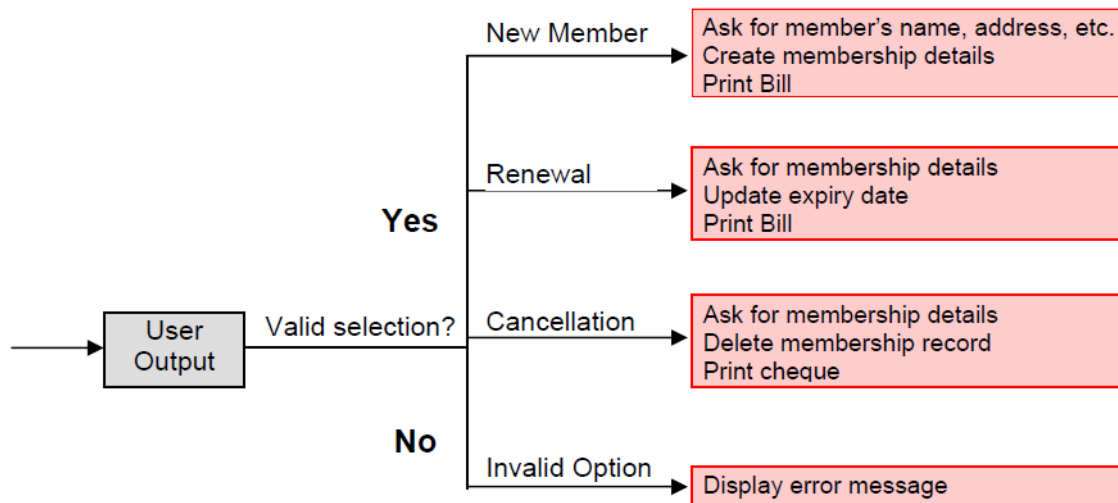


Fig. 3.4: Decision tree for LMS

Decision table

A decision table is used to represent the complex processing logic in a tabular or a matrix form. The upper rows of the table specify the variables or conditions to be evaluated. The lower rows specify the actions to be taken when the corresponding conditions are satisfied. A column in a table is called a *rule*. A rule implies that if a condition is true, then the corresponding action is to be executed.

Example: -

Consider the previously discussed LMS example. The following decision table (fig. 3.5) shows how to represent the LMS problem in a tabular form. Here the table is divided into two parts, the upper part shows the conditions and the lower part shows what actions are taken. Each column of the table is a rule.

Conditions				
Valid selection	No	Yes	Yes	Yes
New member	-	Yes	No	No
Renewal	-	No	Yes	No
Cancellation	-	No	No	Yes
Actions				
Display error message	x	-	-	-
Ask member's details	-	x	-	-
Build customer record	-	x	-	-
Generate bill	-	x	x	-
Ask member's name & membership number	-	-	x	x
Update expiry date	-	-	x	-
Print cheque	-	-	-	x
Delete record	-	-	-	x

Fig. 3.5: Decision table for LMS

From the above table we can easily understand that, if the valid selection condition is false then the action taken for this condition is 'display error message'. Similarly, the actions taken for other conditions can be inferred from the table.

CHARACTERISTICS OF BAD SRS:-

If SRS document is written by novice then it is said to be bad SRS. It might be the problem of contradiction, incompleteness and ambiguity, other problem that is chances to occur are:

(a)Over specification:-

Over specification occurs when the SRS document includes how to issue. Ex:- In case of library automation system we must not record the fact whether the membership records are sorted in descending order by considering members first name.

b) Forward Reference:-

This aspect specifies that, we must not include details of the requirements in the initial stage which will be discussed later in the SRS documents. For example in case of faster processing, the books may be kept separately by indexing with respect to their ISBN(International standard book number). These details are irrelevant in the initial stages 1 & 2 etc.

(c)Whish-full Thinking:-

This aspect specifies that requirements those will be difficult to implement need not be describe/mention in the SRS.

(d)Noise:-

Noise means irrelevant materials associated to software development which might rarely or not used at all.

Different categories of user of SRS documents & their expectations:-

1. user, customer & marketing personnel
2. s/w developers
3. test engineers
4. user documentation writers
- 5.project managers
6. maintenance engineer

SPECIFICATION OF COMPLEX LOGIC:-

There are two type complex logics such as follows:

- a)Axiomatic specification
- b)Algebraic specification

Axiomatic specification

In axiomatic specification of a system, first-order logic is used to write the pre and post-conditions to specify the operations of the system in the form of axioms.

The pre-conditions basically capture the conditions that must be satisfied before an operation. In essence, the pre-conditions capture the requirements on the input parameters of a function. The post-conditions are the conditions that must be satisfied when a function completes execution. Thus, the post conditions are essentially constraints on the results produced for the function execution to be considered successful.

The following are the sequence of steps that can be followed to systematically develop the axiomatic specifications of a function:

- It establishes the range of input values over which the function should behave correctly. And also it finds out other constraints on the input parameters and writes it in the form of a predicate.
- It specifies a predicate defining the conditions which must hold on the output of the function.

- It establishes the changes made to the function's input parameters after execution of the function. Pure mathematical functions do not change their input and therefore this type of assertion is not necessary for pure functions.

- Now on combining all of the above into pre and post conditions of the function as in below example

Example1: -

Specify the pre- and post-conditions of a function that takes a real number as argument and returns half the input value if the input is less than or equal to 100, or else returns double the value.

$f(x : \text{real}) : \text{real}$

pre: $x \in \mathbb{R}$

post: $\{(x \leq 100) \wedge (f(x) = x/2)\} \vee \{(x > 100) \wedge (f(x) = 2x)\}$

Algebraic specification

In the algebraic specification technique an object class or type is specified in terms of relationships existing between the operations defined on that type.

Representation of algebraic specification

Algebraic specifications define a system as a heterogeneous algebra. A heterogeneous algebra is a collection of different sets on which several operations are defined. homogeneous algebras are traditional.

A homogeneous algebra consists of a single set and several operations; $\{I, +, -, *, /\}$. In contrast, alphabetic strings together with operations of concatenation and length $\{A, I, \text{con}, \text{len}\}$, is not a homogeneous algebra, since the range of the length operation is the set of integers.

Each set of symbols in the algebra, is called a *sort* of the algebra. To define a heterogeneous algebra, we first need to specify its signature, the involved operations, and their domains and ranges. Using algebraic specification, we define the meaning of a set of interface procedure by using equations. An algebraic specification is usually presented in four sections.

Types section:-

In this section, the sorts (or the data types) being used is specified.

Exceptions section:-

This section gives the names of the exceptional conditions that might occur when different operations are carried out. These exception conditions are used in the later sections of an algebraic specification.

For example, in a queue, possible exceptions are no value (empty queue), underflow (removal from an empty queue), overflow etc.

Syntax section:-

This section defines the signatures of the interface procedures. The collection of sets that form input domain of an operator and the sort where the output is produced are called the signature of the operator.

For example, the append operation takes a queue and an element and returns a new queue. It is represented as:

append: queue x element → queue

Equations section:-

This section gives a set of rewrite rules (or equations) defining the meaning of the interface procedures in terms of each other. In general, this section is allowed to contain conditional expressions.

For example, a rewrite rule to identify an empty queue may be written as:

isempty(create()) = true

Example:-

Let us specify a FIFO queue supporting the operations *create*, *append*, *remove*, *first*, and *isempty* where the operations have their usual meaning.

Types:

defines queue uses boolean, integer

Exceptions:

underflow, no value

Syntax:

1. *create* : $\varnothing \rightarrow \text{queue}$
2. *append* : *queue x element* → *queue*
3. *remove* : *queue* → *queue* + {underflow}
4. *first* : *queue* → *element* + {no value}
5. *isempty* : *queue* → Boolean

Equations:

1. *isempty(create()) = true*
2. *isempty(append(q,e)) = false*
3. *first(create()) = no value*
4. *first(append(q,e)) = is isempty(q) then e else first(q)*
5. *remove(create()) = underflow*
6. *remove(append(q,e)) = if isempty(q) then create() else append(remove(q),e)*

In this example, there are two basic constructors (*create* and *append*), one extra construction operator (*remove*) and two basic inspectors (*first* and *empty*). Therefore, there are $2 \times (1+2) + 0 = 6$ equations.

Example:-

Let us specify a data type point supporting the operations *create*, *xcoord*, *ycoord*, *isequal*; where the operations have their usual meaning.

Types:

defines point, uses boolean, integer

Syntax:

1. *create* : integer \times integer \rightarrow point
2. *xcoord* : point \rightarrow integer
3. *ycoord* : point \rightarrow integer
4. *isequal* : point \times point \rightarrow Boolean

Equations:

1. *xcoord*(*create*(*x*, *y*)) = *x*
2. *ycoord*(*create*(*x*, *y*)) = *y*
3. *isequal*(*create*(*x1*, *y1*), *create*(*x2*, *y2*)) = ((*x1* = *x2*) and (*y1* = *y2*))

In this example, there is only one basic constructor (*create*), and three basic inspectors (*xcoord*, *ycoord*, and *isequal*). Therefore, there are only 3 equations.

ORGANISATION OF AN SRS DOCUMENTS:-

1. INTRODUCTION

- 1.1 purposes
- 1.2 scopes
- 1.3 definition acronyms & abbreviation
- 1.4 references
- 1.5 overview

2. THE OVERALL DESCRIPTION

- 2.1 Product Perspective
 - 2.1.1 system interfaces
 - 2.1.2 interfaces
 - 2.1.3 h/w interfaces
 - 2.1.4 s/w interfaces
 - 2.1.5 communications requirements
 - 2.1.6 memory constraints
 - 2.1.7 operation
 - 2.1.8 site adaptation requirements
- 2.2 product function
- 2.3 user characteristics
- 2.4 constraints
- 2.5 operating environments

- 2.6 user environments
- 2.7 assumptions & dependencies
- 2.8 apportioning of requirements

3. SPECIFIC REQUIREMENTS

- 3.1 external interfaces
 - i) user interfaces
 - ii) h/w interfaces
 - iii) s/w interfaces
 - iv) communication interfaces
- 3.2 functions
- 3.3 performance requirements
- 3.4 logical database requirements
- 3.5 design constraints
 - 3.5.1 standard compliance
- 3.6 s/w system attribute
 - 3.6.1 reliability
 - 3.6.2 availability
 - 3.6.3 security
 - 3.6.4 maintainability
 - 3.6.5 portability
- 3.7 organizing the specific requirements
 - 3.7.1 system mode
 - 3.7.2 user class
 - 3.7.3 objects
 - 3.7.4 feature
 - 3.7.5 stimulus
 - 3.7.6 response
 - 3.7.7 functional hierarchy
- 3.8 additional comments

4. SUPPORTING INFORMATION

- 4.1 table of content & index
- 4.2 appendixes

Cohesion

When function of one module cooperates with another to perform single objective then module is said to be good cohesion. The primary characteristics of neat module decomposition are high cohesion and low coupling. 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. High cohesion has features of reliability, robustness and reusability where as low cohesion has difficult to test, reuse even difficult to understand and maintenance.

Classification of cohesion

The different classes of cohesion that a module may possess are described as follows from lower cohesion to higher where coincidental cohesion worst type and functional is best type.

Coincidental	Logical	Temporal	Procedural	Communicational	Sequential	Functional
--------------	---------	----------	------------	-----------------	------------	------------

Low  High

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 or parts of the module grouped arbitrarily. Different functions in module carry out different activities. For example, in a transaction processing system (TPS), the 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, such as module contain set of print functions to generate various type of output report such as salary, grade sheet, manual report etc. Here parts of the modules are logically categorized to do same thing

even if they are different by nature (e.g. grouping all mouse and keyboards input handling routines).

Temporal cohesion: When a module contains functions that are related with each other and all the functions must be executed in the same time span, then module is said to exhibit temporal cohesion. For example when computer booted several functions need to be performed at the same time one by one such as initialization of memory, devices, loading OS etc. and The set of functions in a single module responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion.

Another example we can consider as a functions which called after catching an exception which closes opened files, give notification to user, create error log etc.

Procedural cohesion: A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message. As in the above, the parts/functions of the modules always follow certain sequence of execution. Another example is function check file permission and then opened. Plus we can consider the activities included in the trading house can contain functions login (), place-order (), print-bill (),...log-out() etc which are operate on different data but flow sequentially.

Communicational cohesion: Communicational cohesion is also called informational cohesion. Parts/functions of the modules grouped together to operate on same data. 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.

Another example of informational cohesion is let us consider a module name student which contain various functions like admit student, enter marks, print grade sheet etc and all the data stored in an array name student record that is defined with in a module.

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

Another example in an online store system when customer request for some item then functions involved are follow as sequence of: create-order(), check-avail() and then order ()).

Functional cohesion: Functional cohesion is said to exist, if different elements/parts/functions of a module cooperate to achieve a single task. For example,

a module containing all the functions required to manage employees' pay-roll exhibits functional cohesion such as compute-overtime(), compute-workhr(), compute-deduction() etc . It is considered one of the most superior cohesion in the module but in certain cases not achievable and instead communicational cohesion overcome the solution.

Coupling

Coupling between two modules/classes/components is a measure of the degree of interdependence or interaction between the two modules or mutual interdependence between modules. A module having high cohesion and low coupling is said to be functionally independent of other modules.

Two modules are said to be highly coupled when:

- function calls between two modules involve & shared large chunk of data.
- interaction occurs through shared data.

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 determined based on the parameter, no. of parameter etc. Module with low coupling is better.

Classification of Coupling

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. These are shown below according to flow of lower coupling to higher, indicated by an arrow.

Data	Stamp	Control	Common	Content
------	-------	---------	--------	---------

Low  High

Data coupling: Two modules are data coupled, if they communicate by using an elementary data item e.g. an integer, a float, a character, etc such as passing any float value through function that computes square root. 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 record in PASCAL or a structure in C and used part of these data.

Control coupling: Control coupling exists between two modules, if data from one module is used in another module to direct the order of instructions execution. 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. It is also known as global coupling. Example, sharing global variable. Changing global variable leads changes in all modules.

Content coupling: Also known as pathological coupling. Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module or if one module changes on internal working of another module (e.g accessing local data of another module).

No coupling exists when modules do not communicate with each other. From the above degree of coupling increases from the data coupling (that has less interdependency, coordination and data flow) to content coupling(that has high interdependency, coordination and information flow). Other coupling may be there but not given in the book.

High coupling among modules makes design solution difficult to understand and maintenance, increase development effort, harder to reuse and test particular module, assembling of module may required more effort, change in one module usually leads a ripple effect and difficult to develop modules in independent way, these are the disadvantages. High cohesion is contrasted with high coupling and somehow it correlates with loose coupling and vice versa.

Data flow oriented design:

Data flow-oriented design technique identifies:-

- Different processing stations (functions) in a system
- The data items that flows between processing stations

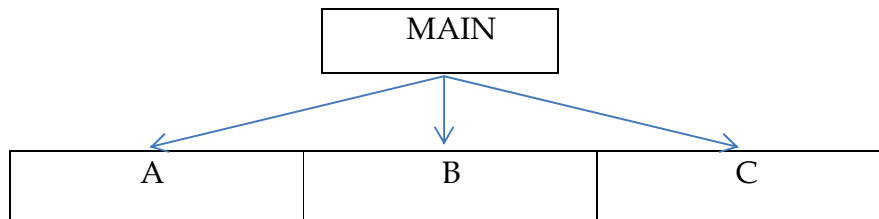
In the late 1970s Data flow oriented design was developed. Experienced programmers stated that to have a good program structure, one has to study how the data flows from input to the output of the program. Every program reads data and then processes that data to produce some output. Once the data flow structure is identified, then from there one can derive the program structure.

Object-oriented design (1980s) is the latest and very widely used technique. Design approach in which natural objects (such as employees, pay-roll register, etc.) occurring in a problem are first identified. Relationships among objects (such as composition, reference and inheritance) are determined. Each object essentially acts as a data hiding entity.

Transform centered design:

Transform analysis consists of five broad steps:

1. To start with, a level-2 or a level-3 data flow diagram of the problem is considered so that the processes represent elementary functions.
2. The data flow diagram is divided into three parts:
 - (a) The *input part (the afferent branch)* that includes processes that transform input data from physical (e.g., character from terminal) to logical form (e.g., internal table).
 - (b) The *logical (internal) processing part (central transform)* that converts input data in the logical form to output data in the logical form.
 - (c) The *output part (the efferent branch)* that transforms output data in logical form (e.g., internal error code) to physical form (e.g., error report)
3. A high-level structure chart is developed for the complete system with the *main module* calling the *inflow controller (the afferent) module*, the *transform flow controller module*, and the *outflow controller (the efferent) module*. This is called the *first-level factoring*. Figure13.15 shows the high-level structure chart for this scheme.



(Fig. 13.15. First-level factoring)

Here on activated, the main module carries out the entire task of the system by calling upon its subordinate modules. A is the input controller module when it activated, will enable the subordinate afferent modules to send the input data streams to flow towards the main module. C is the output controller module which, when activated, will likewise enable its subordinate modules to receive output data streams from the main module and output them as desired. B is the transform flow controller which, when activated, will receive the input streams from the main module, pass them down to its subordinate modules, receive their output data streams, and pass them up to the main module for subsequent processing and outputting by the efferent modules.

4. The high-level structure chart is now factored again (*the second level factoring*) to obtain the *first-cut design*. The second-level factoring is done by mapping individual transforms (bubbles) in the data flow diagram into appropriate modules within the program structure. A rule that is helpful during the second-level factoring process is to ensure that, the processes appearing in the afferent flow in the data flow diagram form themselves into modules that form the lowest-level in the structure chart and sending data upwards to the main module. The processes appearing in the efferent flow in the data flow diagram form themselves into modules that also appear at the lowest-level of the structure chart and receive data from the main module downwards. Figure 13.16 shows the first-cut design.

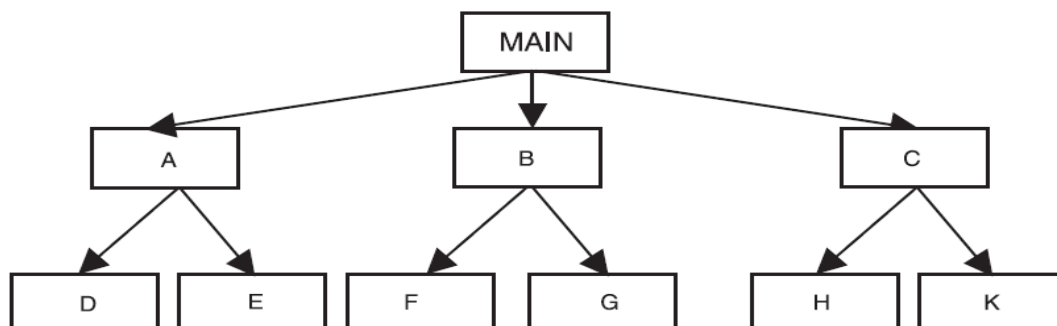


Fig. 13.16. First-cut design

The first-cut design is important as it helps the designer to write a brief processing narrative that forms the *first-generation design specification*. The specification should include

- (a) The data in and out of every module (the interface design),
- (b) The data stored in the module (the local data structure),
- (c) A procedural narrative (major tasks and decisions), and
- (d) Special restrictions and features.

5. The first-cut design is now refined by using design heuristics for improved software quality. The design heuristics are the following:

- (a) Apply the concepts of module independence that is, the modules should be designed so as to be highly cohesive and loosely coupled.
- (b) Minimize high fan-out, and strive for fan-in as depth increases, so that the overall shape of the structure chart is dome-like.
- (c) Avoid pathological connections by avoiding flow of control and by having only single-entry, single-exit modules.
- (d) Keep scope of effect of a module within the scope of control of that module.

We take a hypothetical data flow diagram (Figure 13.17) to illustrate the transform analysis strategy for program design. It is a data flow diagram with elementary functions. It contains 11 processes, two data stores, and 21 data flows. The two vertical lines divide the data flow diagram into three parts, the afferent part, the central transform, and the efferent part.

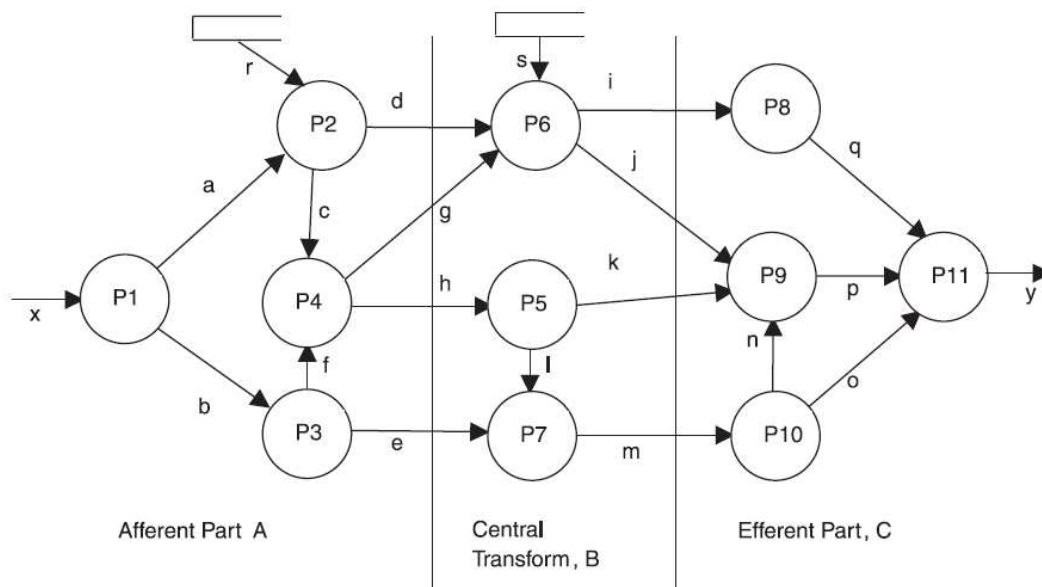


Fig. 13.17. DFD with elementary functions

Figure 13.18 is the structure chart showing the first-level structuring of the data flow diagram. Here module A represents the functions to be done by processes P1 through

P4. Module B does the functions P5 through P7, and module C does the functions P8 through P9.

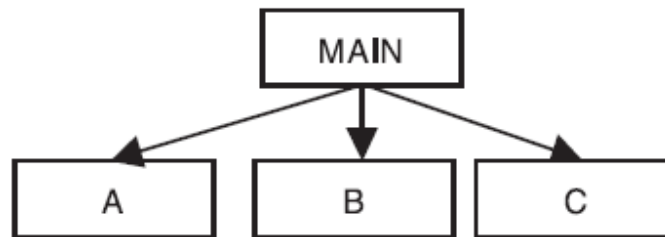


Fig. 13.18. First-level factoring

We now carry out a second-order factoring and define subordinate modules for A, B, and C. To do this, we look at the functions of various processes of the data flow diagram which each of these modules is supposed to carry out.

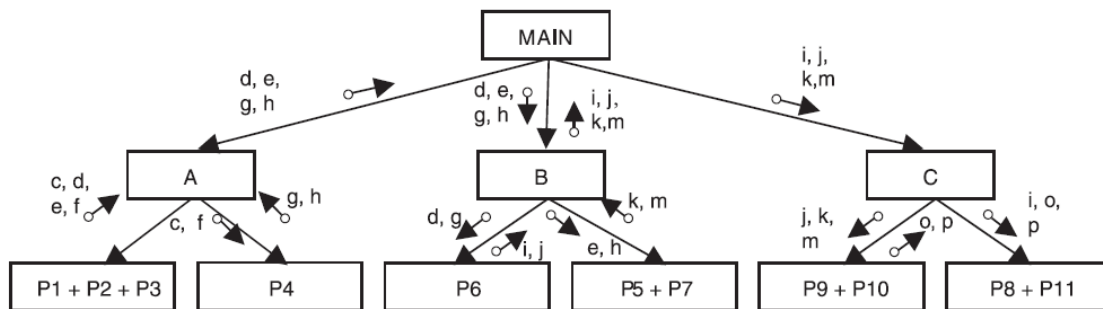


Fig. 13.19. First-cut design

Notice: in the above figure flow of data from and to the modules, the data flows are consistent with the data flow diagrams also that we have chosen the bottom-level modules in such a way that they have either functional or sequential or communicational cohesion. The module P1+ P2 + P3 contains too many functional components and perhaps can be broken down into its subordinate modules. A modification of the first-cut design is given in Fig. 13.20 which may be accepted as the final design of architecture of the problem depicted in the data flow diagram (Figure 13.17).

Transaction Analysis:

Transform analysis is the dominant approach in structured design; often special structures of the data flow diagram can be utilized to adopt alternative approaches. Transaction analysis is recommended in situations where a transform splits the input data stream into several discrete output sub streams.

For example, a transaction may be a receipt of goods from a vendor or shipment of goods to a customer. Thus once the type of transaction is identified, the series of actions is fixed. The process in the data flow diagram that splits the input data into different transactions is called the *transaction center*. In the below figure 13.20 gives a data flow diagram in which the process P1 splits the input data streams into three different transactions, each following its own series of actions. P1 is the transaction center here.

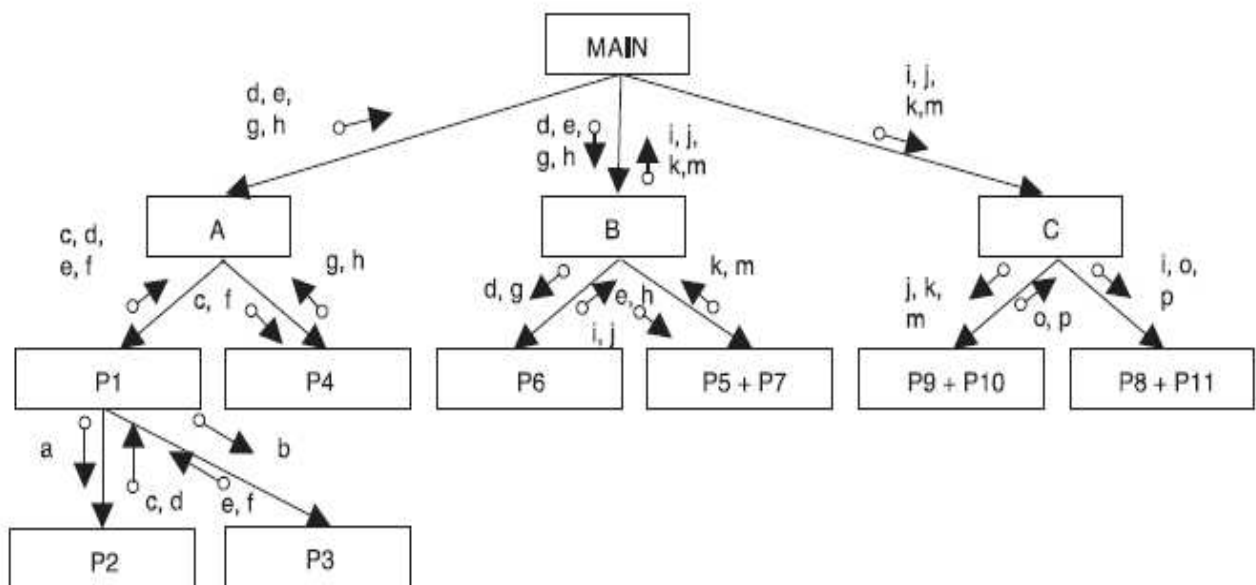


Fig. 13.20. Final design of architecture

And an appropriate structure chart for a situation depicted in Fig. 13.21 is the one that first identifies the type of transaction read and then invokes the appropriate subordinate module to process the actions required for this type of transaction. Figure 13.22 is one such high-level structure chart.

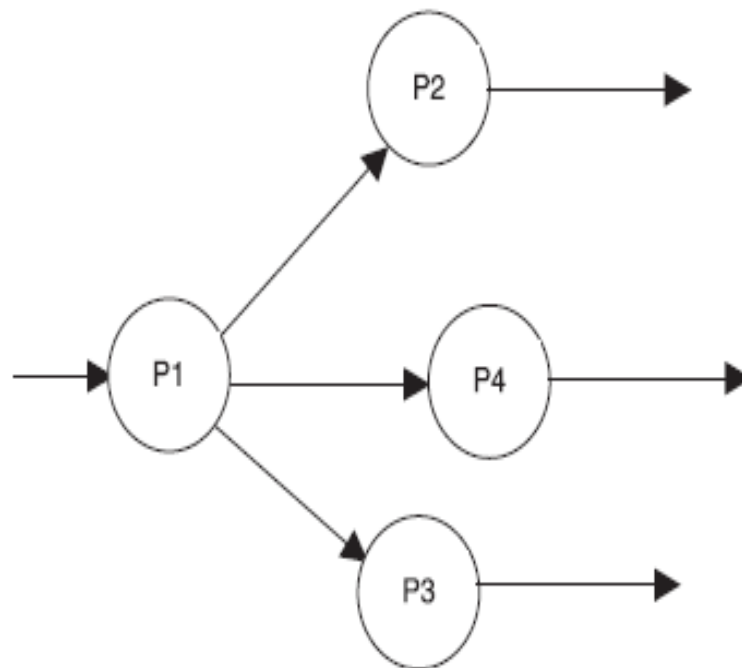


Fig. 13.21. Transaction center in a DFD

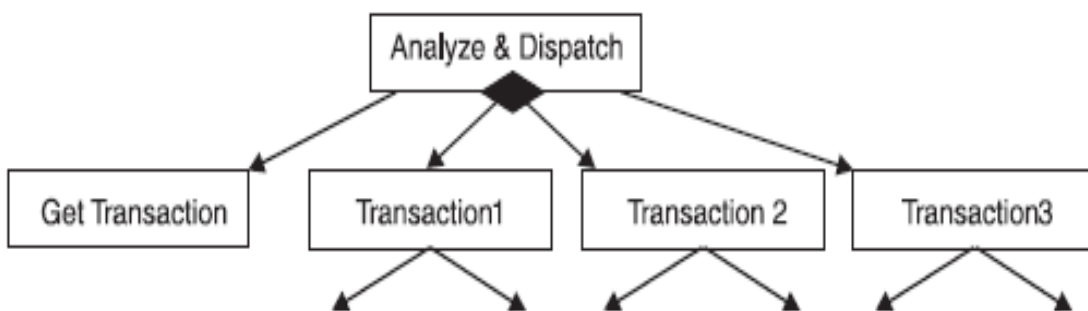


Fig. 13.22. High-level structure chart for transaction analysis

Transaction analysis consists of five steps:

1. The problem specifications are examined and transaction sources are identified.
2. The data flow diagram (level 2 or level 3) is examined to locate the transaction center that produces different types of transactions and locate and group various functions for each type of transaction.
3. A high-level structure chart is created, where the top level is occupied by the transaction center module that calls various transaction modules, each module for a specific type of transaction.
4. The transaction modules are factored to build the complete structure chart.
5. Then 'first-cut' program structure refined using the design heuristics for improved software quality. In practice, often a combination strategy is used. This strategy combines the features of transform analysis and transaction analysis. For example, when transform analysis alone cannot identify a reasonable central transform then, transaction analysis is used to break the system (or program) into subsystems. Similarly, during a transaction analysis, if defining the root module as at transaction center makes it too complex, several transaction centers can be identified.

Inventory control system:

An inventory control system is required for a major supplier of auto parts. The analyst finds that problems with the current manual system include

- (1) Inability to obtain the status of a component rapidly
- (2) two- or three-day turnaround to update a card file
- (3) Multiple reorders to the same vendor

Because there is no way to associate vendors with components, and so forth. Once problems have been identified, the analyst determines what information is to be produced by the new system and what data will be provided to the system. For instance, the customer desires a daily report that indicates what parts have been taken from inventory and how many similar parts remain. The customer indicates that inventory clerks will log the identification number of each part as it leaves the inventory area.

Upon evaluating current problems and desired information (input and output), the analyst begins to synthesize one or more solutions. To begin, the data objects, processing functions, and behavior of the system are defined in detail. Once this information has been established, basic architectures for implementation are considered. A client/server approach would seem to be appropriate, but does the software to support this architecture fall within the scope outlined in the *Software Plan*? A database management system would seem to be required, but is the user/customer's need for associatively justified? The process of evaluation and synthesis continues until both analyst and customer feel confident that software can be adequately specified for subsequent development steps.

Reservation system:

Reservation system used the concept of *Reuse domain*, consider the airline reservation system, the reusable objects can be seats, flights, airports, crew, meal orders, etc. The reusable operations can be scheduling a flight, reserving a seat, assigning crew to flights, etc. The domain analysis generalizes the application domain. A domain model transcends specific applications. The common characteristics or the similarities between systems are generalized.

Similarly the reservation system also used the concept of *host-slave*, consider the Railway-reservation system here the master at any time directs the slaves what to do. A slave can only make requests and master takes over and tells what to do.

MODULE-II

Lecture Note: 11

Object Modeling using UML

Basic Ideas on UML

Model:

A model is the graphical, textual, mathematical, or program code-based representation. A model captures important aspect for some application while omitting (or abstracting) the rest. Models are very useful in documenting the design and analysis results.

Need for a model

An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following:

- Analysis
- Specification
- Code generation
- Design
- Visualize and understand the problem and the working of a system
- Testing, etc

Unified Modeling Language (UML)

UML, as the name implies, is a modeling language. It may be used to visualize, specify, construct, and document the artifacts of a software system. It provides a set of notations (e.g. rectangles, lines, ellipses, arrow etc.) to create a visual model of the system. UML was developed to standardize the large number of object-oriented modeling notations that existed and were used extensively in the early 1990s. Some of the principles that ones in use were:

- Object Management Technology [Rumbaugh 1991]
- Booch's methodology [Booch 1991]
- Object-Oriented Software Engineering [Jacobson 1992]
- Odell's methodology [Odell 1992]

- Shaler and Mellor methodology [Shaler 1992]

UML was adopted by Object Management Group (OMG) as a de facto standard in 1997. OMG is an association of industries which tries to facilitate early formation of standards.

UML diagrams

UML can be used to construct several different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.

The UML diagrams can capture the following five views of a system:

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

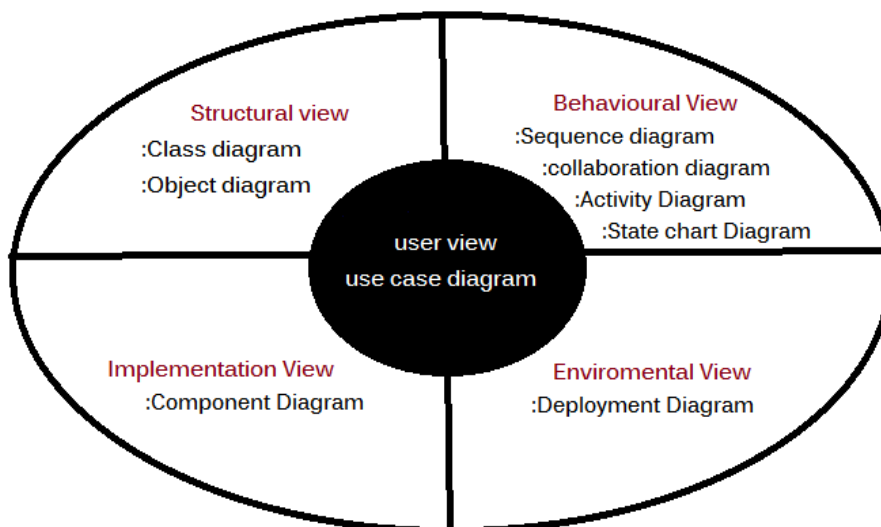


Fig. 1: Different types of diagrams and views supported in UML

User's view:

This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system

where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. It is very different from all other that it is a functional model compared to the object model of all other views. The users' view can be considered as the central view and all others are expected to conform to this view.

Structural view:

The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.

Behavioral view:

The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.

Implementation view:

This view captures the important components of the system and their dependencies.

Environmental view:

This view models how the different components are implemented on different pieces of hardware platform

Use Case Model

The use case model for any system consists of a set of "use cases", users and relationship among them within the system. Use case describes the action that user takes on a system. Use case diagram model the process flow of a system. It is the way to communicate with the user. Intuitively, use cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: "What the users can do using the system?". Four basic components to construct use case diagram are:

- a) **System:** Something that performs a function or system is a piece/ multiple piece of software that perform some function for its user.
- b) **actor:** actor is something that uses our system. It is named based on their job title. An actor can be a person or another external system like developer, student, clerk, teacher, officer etc.
- c) **use cases:** are the actions that a user takes on a system. According to above example

developer creates s/w, teacher record grade, officer create command etc.

d)**relationship:** connection between the actor to the use cases. Actor can relates to multiple use cases and a use cases relate to multiple actors.

Thus for the Library Information System (LIS), the use cases could be:

- issue-book
- query-book
- return-book
- create-member
- add-book, etc

Similarly the use case model for the Supermarket Prize Scheme is shown in fig. 2. As discussed earlier, the use cases correspond to the high level functional requirements. From the problem description and the context diagram. In fig.2 we can identify three use cases: “register-customer”, “register-sales”, and “select-winners”. As a sample, the text description for the use case “register-customer” is shown below.

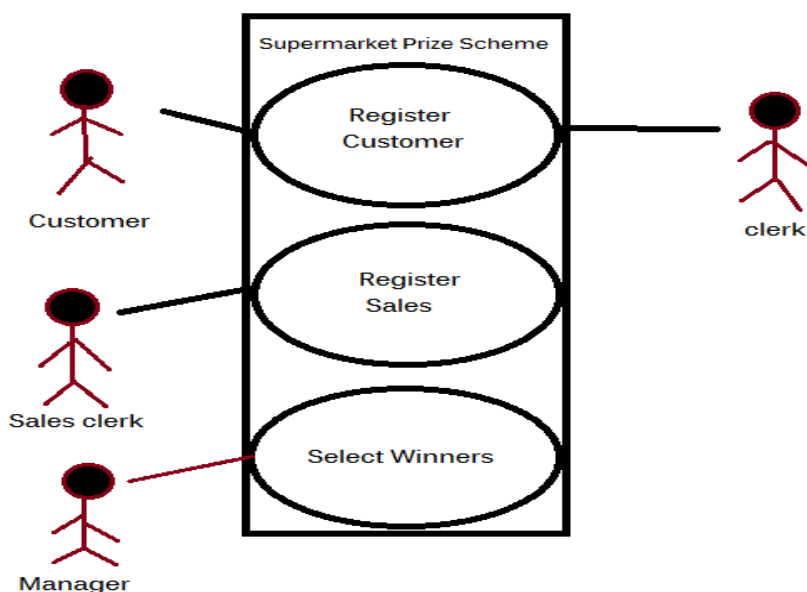


Fig. 2 Use case model for Supermarket Prize Scheme

Text description

U1: register-customer: Using this use case, the customer can register himself by providing the necessary details.

Scenario 1: Mainline sequence

1. **Customer:** select register customer option.
2. **System:** display prompt to enter name, address, and telephone number.
3. **Customer:** enter the necessary values.
4. **System:** display the generated id and the message that the customer has been successfully registered.

Scenario 2: at step 4 of mainline sequence

1. **System:** displays the message that the customer has already registered.

Scenario 2: at step 4 of mainline sequence

1. **System:** displays the message that some input information has not been entered. The system displays a prompt to enter the missing value.

Utility of use case diagrams

Use cases are represented by ellipses. They along with the accompanying text description serve as a type of requirements specification of the system and form the core model to which all other models must conform. UML offers three mechanisms for factoring of use cases as follows:

Generalization

Generalization is a technique that used to inherit an item in UML. It can be applied to both actor and use cases to indicate that their child item inherit functionality from parent. It can be used when one use case is similar to another. The child use case inherits the behavior and meaning of the parent use case. But the base and the derived use cases are separate use cases and should have separate text descriptions.



Fig. 3: Representation of use case generalization

Includes

The includes relationship in the older versions of UML (prior to UML 1.1) was known as the use cases relationship. Include relationship indicate that use case will include functionality from an additional use cases. The relationship involves one use case including the behavior of another use case in its sequence of events and actions. The includes relationship occurs when a chunk of behavior that is similar across a number of use cases. Its advantage is to make helpful to decompose large and complex use cases into more manageable part. It is pointed to represented as line and open arrow and inside pre defines sterio type such as <<include>> as in below.

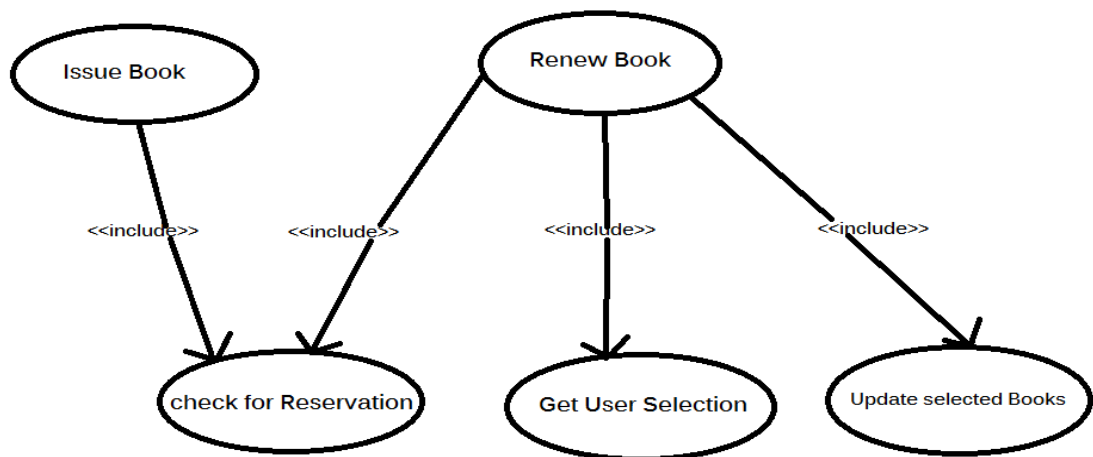
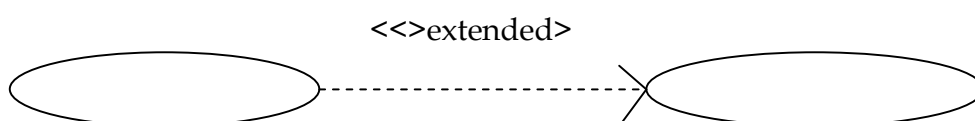


Fig. 4: Example use case inclusion

Extends

Extended relationship indicates that use case may be extended by another use case. The main idea behind the extends relationship among the use cases is that it allows us to show optional system behavior. And an optional system behavior is extended only under certain conditions. This relationship among use cases is also predefined as a stereotype as shown below and as in the include relationship. The extend relationship is similar to generalization. But unlike generalization, the extending use case can add additional behavior only at an extension point only when certain conditions are satisfied.



Organization of use cases

Use cases can be organized hierarchically. The high level use cases are refined into a set of smaller and more refined use cases as shown below. Only the complex use cases should be decomposed and organized in a hierarchy. It is not necessary to decompose simple use cases. The functionality of the super-ordinate use cases is traceable to their sub-ordinate use cases. Thus, the functionality provided by the super-ordinate use cases is composite of the functionality of the sub-ordinate use cases.

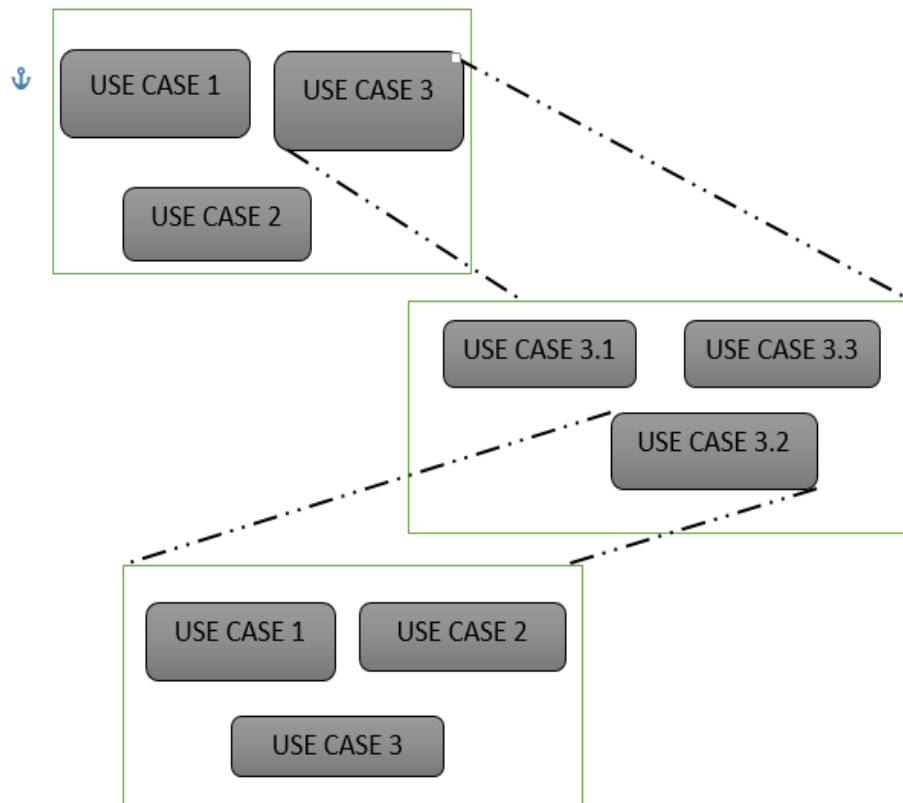


Fig. 5: Hierarchical organization of use cases

Class and Interaction Diagrams

Class diagrams

A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system comprises of a number of class diagrams and their dependencies. The main constituents of a class diagram are classes and their relationships: generalization, aggregation, association, and various kinds of dependencies.

Classes

The classes represent entities with common features, i.e. attributes and operations. Classes are represented as solid outline rectangles with compartments. It has a mandatory name compartment where the name is written centered in boldface. The class name is usually written using mixed case convention and begins with an uppercase. The class names are usually chosen to be singular nouns.

Attributes

An attribute is a property of a class written in left justified. It represents the kind of data that an object might contain. Attributes are listed with their names, and may be contain specification of their type, an initial value, and constraints. The type of the attribute is written by appending a colon and the type name after the attribute name as

book Name : String

Operation

Operation is the implementation of a service/activity that supported by class and invoked by other class object. It can be requested from any object of the class to affect behavior. An object's data or state can be changed by invoking an operation of the object. A class may have any number of operations or no operation at all. Generally the first letter of an operation name is a small letter. Its return type is Boolean as E.g

issueBook(in bookName):Boolean

Association

When two classes associated with each other, their exists association relation. It is represented by straight line between concerned classes. Associations enable objects to communicate with each other. An association describes a connection between classes. The association relation between two objects is called object connection or link. Links are instances of associations. Mathematically, a link can be considered to be a tuple, i.e. an ordered list of object instances. An association describes a group of links with a common structure and common semantics. Each side of association relation multiplicity is noted as an individual number or value range. Multiplicity (*) indicate how many instance of one class are associated with the other. Range of multiplicity are noted by specifying range of minimum to maximum by giving two dots i.e 1..5. An asterisk is used as a wild card (last entry), zero or more.

Let us consider the statement that below illustrates the graphical representation of the association relation. The name of the association is written alongside the association line. An arrowhead is placed on the association line to indicate the reading direction of the association. The arrowhead should not be misunderstood to be indicating the

direction of a pointer implementing an association. The multiplicity indicates how many instances of one class are associated with each other.

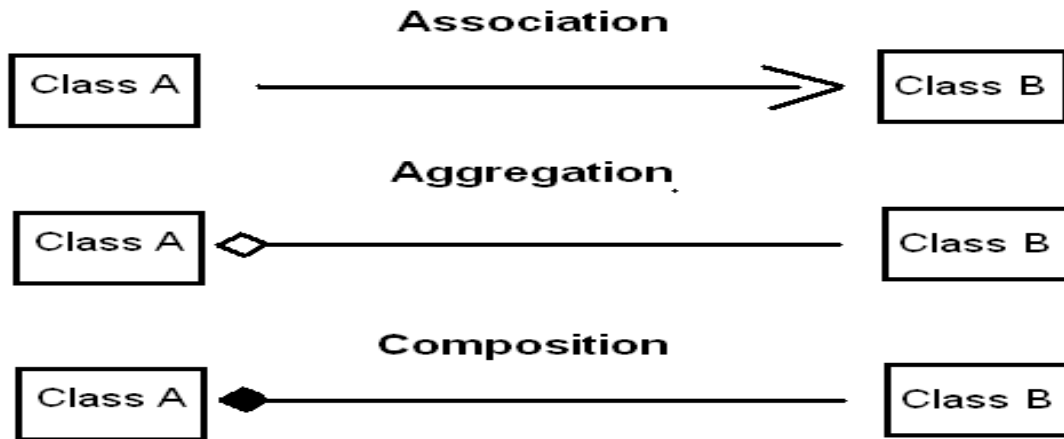


Fig. 6: Association between two classes

Aggregation

Aggregation is a special type of association relation, where classes are not only associated but also whole part relation exist between them. Here involved classes that takes the responsibility of forwarding messages to the appropriate parts. Thus, it takes the responsibility of delegation and leadership. When an instance of one object contains instances of some other objects, then aggregation relationship exists between the composite object and the component object. Aggregation is represented by the diamond symbol at the composite end of a relationship. For example boo register is an aggregation of book objects similarly document can be consider as an aggregation of paragraph and aggregation of line. Here 1 and * means one document can have many paragraph and many lines constitute a paragraph. This relation cannot be reflexive means an object cannot have the object of same class as itself nor symmetric (i.e two classes cannot contain instance of each other). However aggregation relationship is transitive and it can consist of arbitrary number of levels.

Composition

Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole. Means that the life of the parts closely ties to the life of the whole. The composition relationship is represented as a filled diamond drawn at the composite-end. An example of the composition relationship is

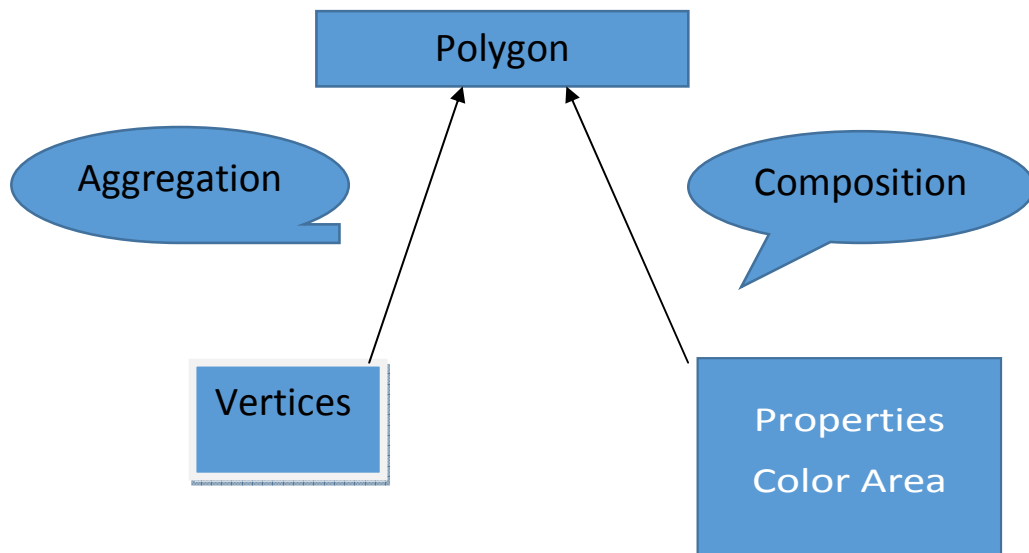


Fig. 7: Diagram representing aggregation and composition.

Dependency

It is represented by a dotted arrow from the dependent class to the independent class. It provides pre and post conditions for operations to define the ordering of an item.

Association vs. Aggregation vs. Composition

- Association is the most general (m:n) relationship. Aggregation is a stronger relationship where one is a part of the other. Composition is even stronger than aggregation.
- Association relationships can be reflexive (objects can have relationships with themselves), but aggregation cannot be reflexive. Aggregation is anti-symmetric (If B is a part of A, A cannot be a part of B).
- Composition has the property of exclusive aggregation, i.e., an object can be a part of only one composite at a time. For example, a **Frame** belongs to exactly one **Window**, whereas in simple aggregation, a part may be shared by several objects. For example, a **Wall** may be a part of one or more **Room** objects.
- In composition, the whole has the responsibility for the decomposition of all its parts, i.e., for their creation and destruction. For example, when a **Frame** is created, it has to be attached to an enclosing **Window**. Similarly, when the **Window** is destroyed, it must in turn destroy its **Frame** parts.

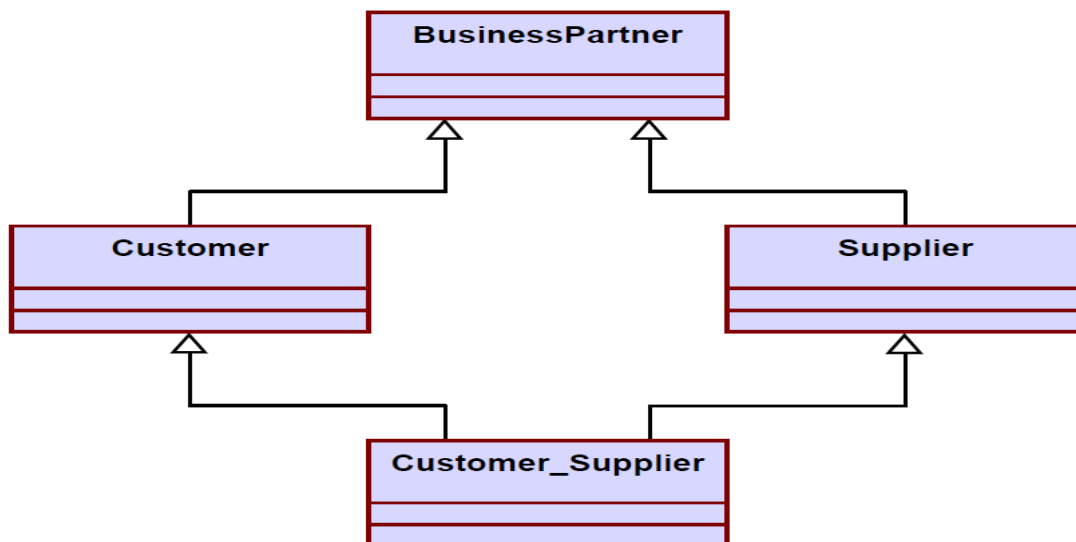
Inheritance vs. Aggregation/Composition

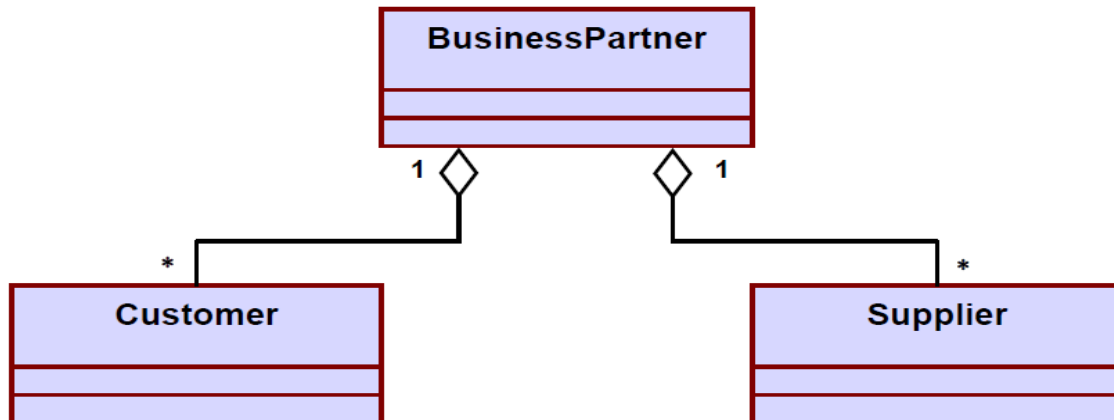
- Inheritance describes 'is a' / 'is a kind of' relationship between classes (base class - derived class) whereas aggregation describes 'has a' relationship between classes. Inheritance means that the object of the derived class inherits the properties of the base class; aggregation means that the object of the whole has objects of the part.

- Inheritance means the objects of the subclass can be used anywhere the super class may appear, but not the reverse; i.e. wherever we could use instances of 'payment' in the system, we could substitute it with instances of 'cash payment', but the reverse cannot be done.

- Inheritance is defined statically. It cannot be changed at run-time. Aggregation is defined dynamically and can be changed at run-time. Aggregation is used when the type of the object can change over time.

For example, consider this situation in a business system. A **Business Partner** might be a **Customer** or a **Supplier** or both. During its lifetime, a business partner might become a customer as well as a supplier, or it might change from one to the other. In such cases, we prefer aggregation instead. Here, a business partner is a **Customer** if it has an aggregated **Customer** object, a **Supplier** if it has an aggregated **Supplier** object and a "**Customer Supplier**" if it has both. Here, we can have only two types. Hence, we are able to model it as inheritance.





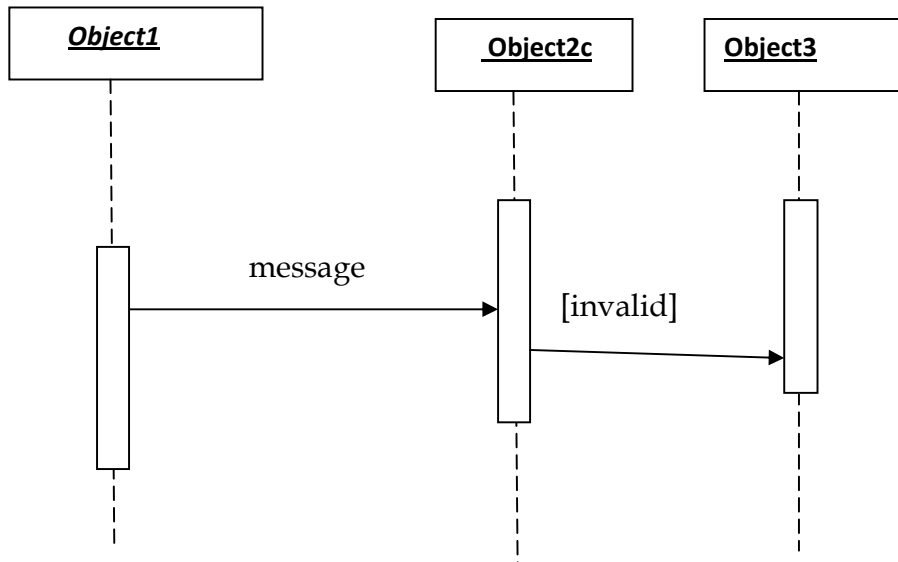
- The advantage of aggregation is the integrity of encapsulation. The operations of an object are the interfaces of other objects which imply low implementation dependencies. The disadvantage of aggregation is the increase in the number of objects and their relationships. On the other hand, inheritance allows for an easy way to modify implementation for reusability.

Interaction Diagrams

Interaction diagram describe the interaction among group of objects through message passing. Interaction diagrams are models that describe how group of objects collaborate to realize some behavior. Each interaction diagram realizes the behavior of a single use case. An interaction diagram shows a number of example objects and the messages that are passed between the objects. There are two kinds of interaction diagrams: sequence diagrams and collaboration diagrams.

Sequence Diagram:

A sequence diagram shows interaction among objects as a two dimensional chart. The chart is read from top to bottom. The objects participating in the interaction are shown at the top of the chart as boxes attached to a vertical dashed line. Vertical dashed line is called object life line. Any point on life line implies that object exist at that point. If object is destroyed, lifeline of object is crossed at that point and lifeline of that object is not drowned at that point. Control rectangle is used as activation symbol drown from lifeline of an object that the point at which object is active. Object is active as long as control rectangle is exist in lifeline of object. Each message is labeled with message name. Inside the box the name of the object is written with a colon separating it from the name of the class and both the name of the object and the class is underlined. The objects appearing at the top signify that the object already existed when the use case execution was initiated. **Note: single use case represent single sequence diagram.**



- A condition (e.g. [invalid]) indicates that a message is sent, only if the condition is true.
- An iteration marker shows the message is sent many times to multiple receiver objects. As it happened elements of an array are being iterated. The basis of the iteration can also be indicated e.g. [for every book object].

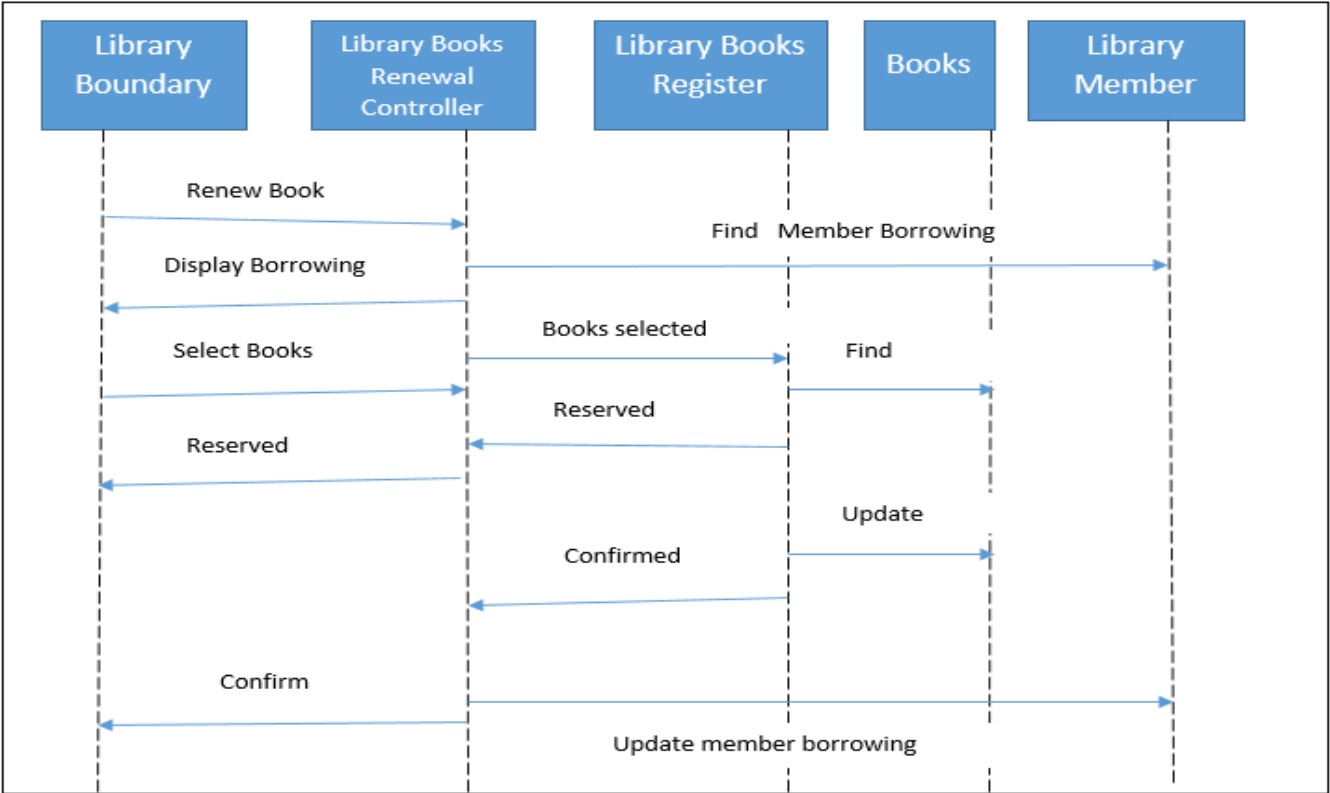


Fig. 8: Sequence diagram for the renew book use case

Collaboration Diagram

Collaboration diagram is a cross between the class diagram and sequence diagram. It model the objects and sequenced communication between each other. Collaboration diagram shows both structural and behavioral aspects explicitly. Structural aspect consists of objects and links among the objects which indicate the association. Object in collaboration diagram is called collaborator. But in sequence diagram it shows only the behavioral aspects. The link between objects is shown as a solid line and can be used to send messages between two objects. The message is shown as a labeled arrow placed near the link. The use of the collaboration diagrams in our development process is to help us to determine which classes are associated with which other classes. This model used to represent interaction between objects and role. Link between object is shown in solid line and can be used to send messages between two objects. Collaboration diagram derived from sequence diagram which realize for single use case. It helps us to, which class associated with the other class. For example collaboration diagram for renew book use case

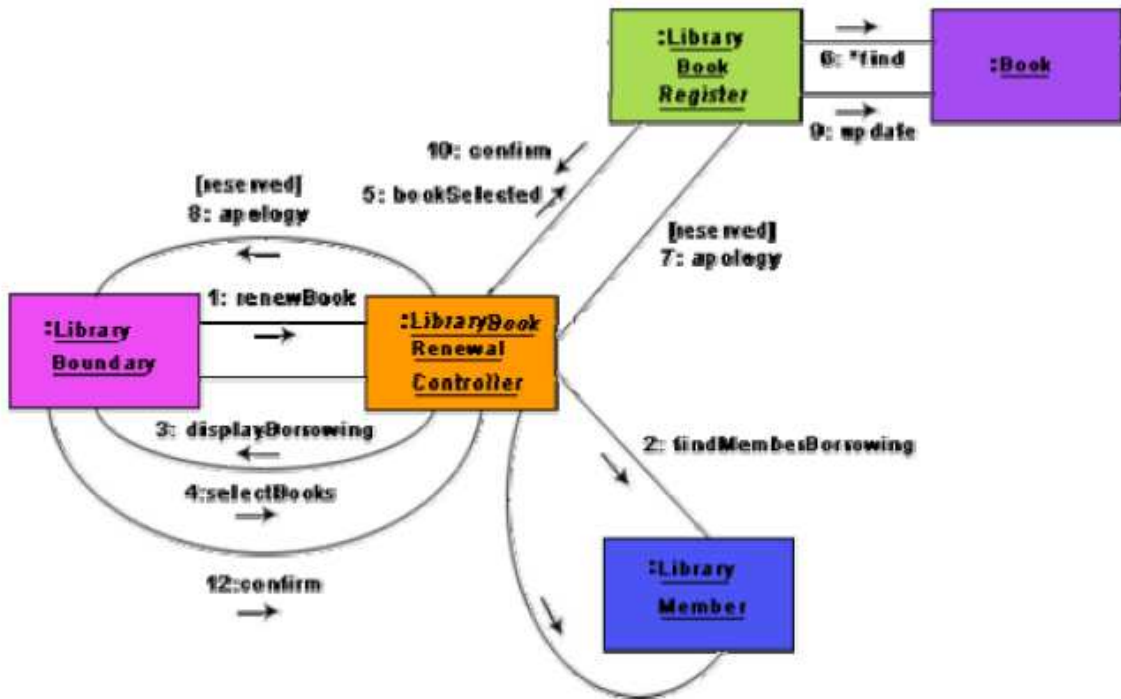


Fig. 9: Collaboration diagram for the renew book use case

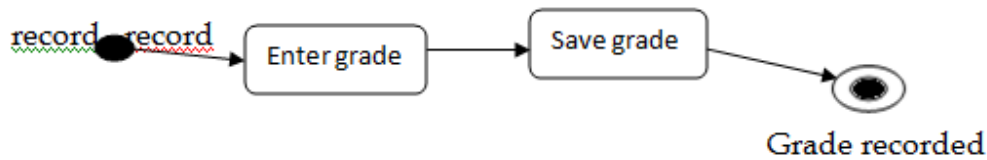
Activity diagrams

Activity diagram represent the various activity or chunk of process and their sequence of activation. It shows the system execution and changes of direction based on different condition. It model workflow for use case. It is possibly based on the event diagram of Odell [1992] through the notation is very different from that used by Odell. An interesting feature of the activity diagrams is the swim lanes. Swim lanes enable us to group activities based on who is performing them, e.g. academic department vs. hostel office. This is carried out during the initial stages of requirements analysis and specification. Activity diagram gives start and end state and can show the path within the use case as well as between the use cases. It explains what condition needs to meet for use cases for use case to be valid condition state etc.

Components of activity diagram are:

- a) Activities or action state: notation used for activity/action is rectangle with rounded corner and it indicate action.
- b) State: activity diagram can have only one start state but can have several end/stop state. UML may describe two special state called start state and stop state represented as: black dot and black dot with round circle.

c) Transition: used to show control flow from one state to another state. It shows flow from state to an activity, between activity or between states.



Guard is noted on transition between two activity or states. And diamond is used as decision point for condition. An activity is a state with an internal action and one or more outgoing transition which automatically follow the internal activity termination.

Activity diagram is similar to procedural flow chart, difference is that activity diagram support description of parallel activity and synchronization exist between activity and parallel activities are represented by swim lane.

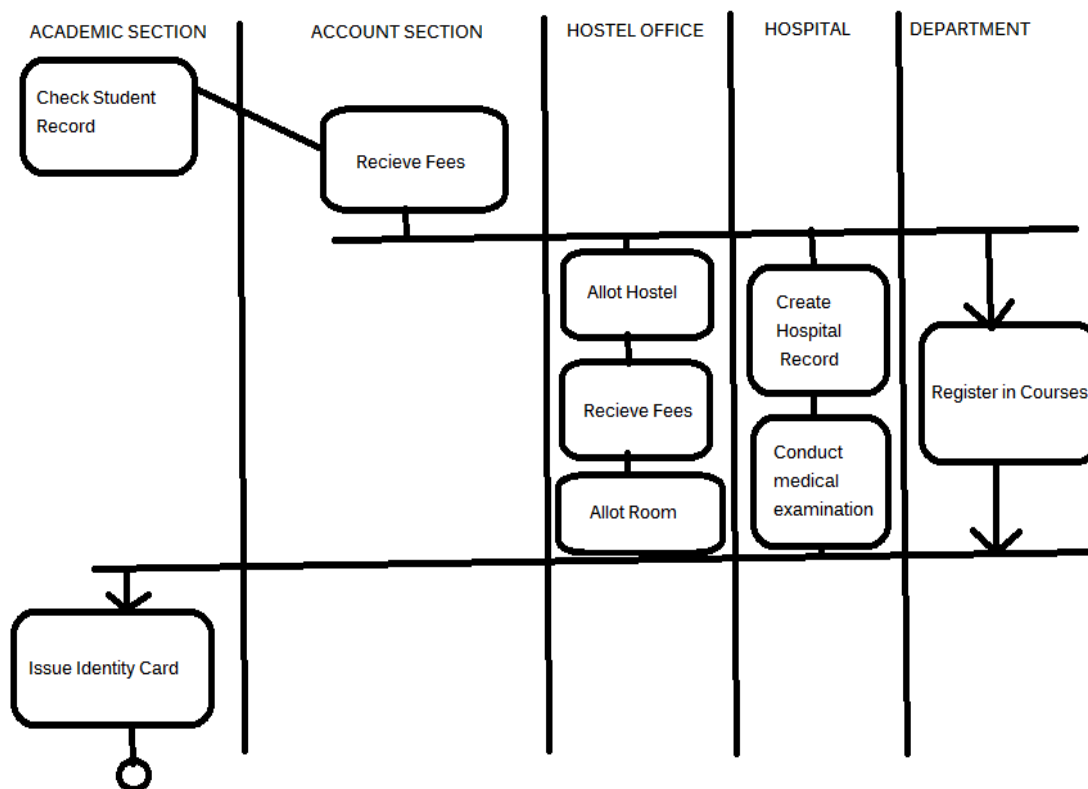


Fig. 9: Activity diagram for student admission procedure at IIT

Unified Development Process

Qualified process for every project is most important which is to be followed thoroughly. Many parts of the process are iterative, especially during the design and planning phases. This flexibility in procedure allows for constant review and improvement, even during the course of rapid development. Some of the approaches are

- A Modular Process
- A Strategic Focus
- A User Centered Approach

A Modular Process

Key competitive advantages of any project is the modularity of overall procedure. All of products and services can be delivered as part of a full development schedule, or individually when needed. In some cases, it leads the strategic planning section of a project where others complete the graphic design or programming tasks.

In some cases, it is provided usability and information architecture services upon which others build functional systems.

In some cases, it is designed the presentation for a campaign that others execute, or we create a graphical interface for software that others programmed.

A Strategic Focus

Every aspect of a project must be driven by a clearly defined, relevant digital strategy. We strongly focus in the strategy in setting clear, measurable goals, and actively analyzing the achievement of those goals to identify a project's rate of success.

A User Centered Approach

This approach focuses on the product's users and potential users and uses from the very beginning. The design process developer generally follows is based on the principles of User Centered Design (UCD). UCD has a broader scope than purely "usability"; and continually throughout the planning, conceptualization, design and development process.

Programming Languages

A programming language is a formal constructed language designed to communicate instructions to a machine, particularly a computer. Programming

languages can be used to create programs to control the behaviour of a machine or to express algorithms.

Three levels of computer programming languages may be distinguished:

- machine languages;
- assembly languages;
- high-level languages.

A program written in a high-level language is called a source program or source text. Rules that prescribe the structure and “grammar” of the source text are called syntactic rules. Rules of content, interpretation and meaning are called semantic rules.

A high-level programming language is determined by its syntactic and semantic rules, i.e. its syntax and semantics.

Every processor has its own language and can execute only those programs that are written in that language. For the processor to understand the program written in the high-level language (i.e. the source text), some method of translation must be in place. There are two techniques to achieve this aim:

- (1) the compiler, and
- (2) the interpreter.

The compiler is a special program which creates an object program in machine code from the source program written in high-level language. The compiler treats the source program as a single unit, and executes the following steps:

- Lexical analysis;
- Syntactic analysis;
- Semantic analysis;
- Code generation.

Classification

I. Imperative (algorithmic) languages

When the programmer writes a program text in these languages, he or she codes an algorithm, and this algorithm makes the processor work. The program is a sequence of statements and instruction. The most important programming feature is the variable, which provides direct access to the memory, and makes it possible to directly manipulate the values stored within. The algorithm changes the values of variables, so

the program takes effect on the memory. Imperative languages are closely connected to the von Neumann architecture.

Imperative languages fall into one of the following sub-groups:

- Procedural languages
- Object-oriented languages

II. Declarative (non-algorithmic) languages

These languages are not connected as closely to the von Neumann architecture as imperative languages. The programmer has to present only the problem, as the mode of the solution is included in language implementations. The programmer cannot perform memory operations, or just in a limited way.

Declarative languages fall into one of the following sub-groups:

- Functional (applicative) languages
- Logic languages
- Other languages

Characteristics

The following are the characteristics of a programming language

1. Readability: A good high-level language will allow programs to be written in some ways that like quite-English description of the algorithms. The coding may be done in a way that is essentially self-documenting.

2. Portability: High-level languages, being essentially machine independent, should be able to develop portable software.

3. Generality: Most high-level languages allow the writing of a wide variety of programs, language should have the ability to implement the algorithm with less amount of code. Programs expressed in high-level languages are often considerably shorter than their low-level equivalents.

4. Error checking: Being human, a programmer is likely to make many mistakes in the development of a computer program. Many high-level languages enforce a great deal of error checking both at compile-time and at run-time.

5. Efficiency: It should permit the generation of efficient object code.

Classes

The classes represent entities with common features, i.e. attributes and operations. Classes have optional attributes and operations compartments.

Attributes

An attribute is a named property of a class. It represents the kind of data that an object might contain. An example for an attribute is given.

Book-Name : String

Operation

Operation is the implementation of a service that can be requested from any object of the class to affect behaviour. An example for an operation is given.

Issue-Book (in book-Name): Boolean

Association

Associations are needed to enable objects to communicate with each other. An association describes a connection between classes.

Coding

Any organizations normally require their programmers to adhere to some well-defined and standard style of coding called coding standards. Most software development organizations formulate their own coding standards that suit them most, and require their engineers to follow these standards rigorously. The purpose of requiring all engineers of an organization to adhere to a standard style of coding is the following:

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It enhances code understanding.
- It encourages good programming practices.

A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error return conventions, etc.

Coding standards and guidelines

Good software development organizations usually develop their own coding standards and guidelines depending on what best suits their organization and the type of products they are going to develop. The following are some representative coding standards.

Rules for limiting the use of global:

These rules list what types of data can be declared global and what local and what not.

Contents of the headers preceding codes for different modules:

The information contained in the headers of different modules should be standard for an organization. The exact format in which the header information is organized in the header can also be specified. The following are some standard header data:

- Name of the module.
- Date on which the module was created.
- Author's name.
- Modification history.
- Synopsis of the module.
- Different functions supported, along with their input/output parameters.
- Global variables accessed/modified by the module.

Naming conventions for global variables, local variables, and constant identifiers: A possible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.

Error return conventions and exception handling mechanisms:

The way error conditions are reported by different functions in a program are handled should be standard within an organization. For example, different functions while encountering an error condition should either return a 0 or 1 consistently. The following are some representative coding guidelines recommended by many software development organizations.

Do not use a coding style that is too clever or too difficult to understand:

Code should be easy to understand. Clever coding can obscure meaning of the code and hamper understanding. It also makes maintenance difficult.

Avoid obscure side effects:

The side effects of a function call include modification of parameters passed by reference, modification of global variables, and I/O operations. Obscure side effects make it difficult to understand a piece of code. For example, if a global variable is changed obscurely in a called module or some file I/O is performed which is difficult to infer from the function's name and header information, it becomes difficult for anybody trying to understand the code.

Do not use an identifier for multiple purposes:

Programmers often use the same identifier to denote several temporary entities. For example, some programmers use a temporary loop variable for computing and a storing the final result. For such multiple uses of variables is memory efficiency, e.g. three variables use up three memory locations, whereas the same variable used in three different ways uses just one memory location. However, there are several things wrong

with this approach and hence should be avoided. Some of the problems caused by use of variables for multiple purposes as follows:

- Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable for multiple purposes can lead to confusion and make it difficult for somebody trying to read and understand the code.
- Use of variables for multiple purposes usually makes future enhancements more difficult.

The code should be well-documented:

As a rule of thumb, there must be at least one comment line on the average for every three-source line.

The length of any function should not exceed 10 source lines:

A function that is very lengthy is usually very difficult to understand as it probably carries out many different functions. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.

Do not use goto statements:

Use of goto statements makes a program unstructured and makes it very difficult to understand.

Code review

Code review for a model is carried out after the module is successfully compiled and the all the syntax errors have been eliminated. Code reviews are extremely cost-effective strategies for reduction in coding errors and to produce high quality code. Normally, two types of reviews are carried out on the code of a module. These two types code review techniques are **code inspection and code walk through**.

Code Walk Throughs

1. Code walk through is an informal code analysis technique.
2. In this technique, after a module has been coded, successfully compiled and all syntax errors eliminated. A few members of the development team are given the code few days before the walk through meeting to read and understand code.
3. Each member selects some test cases and simulates execution of the code by hand (i.e. trace execution through each statement and function execution).
4. The main objectives of the walk through are to discover the algorithmic and logical errors in the code.
5. The members note down their findings to discuss these in a walk through meeting where the coder of the module is present. Even though a code walks

through is an informal analysis technique, several guidelines have evolved over the years for making this naïve but useful analysis technique more effective.

6. Therefore, these guidelines should be considered as examples rather than accepted as rules to be applied dogmatically. Some of these guidelines are the following.

- The team performing code walk through should not be either too big or too small. Ideally, it should consist of between three to seven members.
- Discussion should focus on discovery of errors and not on how to fix the discovered errors.
- In order to foster cooperation and to avoid the feeling among engineers that they are being evaluated in the code walk through meeting. Managers should not attend the walk through meetings.

Code Inspection

1. In contrast to code walk through, the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming.
2. During code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk through.
3. In addition to the commonly made errors, adherence to coding standards is also checked during code inspection. Good software development companies collect statistics regarding different types of errors commonly committed by their engineers and identify the type of errors most frequently committed. Such a list of commonly committed errors can be used during code inspection to look out for possible errors.

Following is a list of some classical programming errors which can be checked during code inspection:

- Use of uninitialized variables.
- Jumps into loops.
- Non-terminating loops.
- Incompatible assignments.
- Array indices out of bounds.
- Improper storage allocation and de allocation.
- Mismatches between actual and formal parameter in procedure calls.
- Use of incorrect logical operators or incorrect precedence among operators.
- Improper modification of loop variables.
- Comparison of equally of floating point variables, etc.

Software documentation

When various kinds of software products are developed then not only the executable files and the source code are developed but also various kinds of documents such as

users' manual, software requirements specification (SRS) documents, design documents, test documents, installation manual, etc are also developed as part of any software engineering process. All these documents are a vital part of good software development practice. Good documents are very useful and server the following purposes:

- Good documents enhance understand ability and maintainability of a software product. They reduce the effort and time required for maintenance.
- Use documents help the users in effectively using the system.
- Good documents help in effectively handling the manpower turnover problem. Even when an engineer leaves the organization, and a new engineer comes in, he can build up the required knowledge easily.
- Production of good documents helps the manager in effectively tracking the progress of the project. The project manager knows that measurable progress is achieved if a piece of work is done and the required documents have been produced and reviewed. Different types of software documents can broadly be classified into the following:
 - Internal documentation
 - External documentation

Internal documentation:

It is the code comprehension features provided as part of the source code itself. Internal documentation is provided through appropriate module headers and comments embedded in the source code. Internal documentation is also provided through the useful variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant identifiers, use of user-defined data types, etc. Careful experiments suggest that out of all types of internal documentation meaningful variable names is most useful in understanding the code.

External documentation:

It is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test documents etc. A systematic software development style ensures that all these documents are produced in an orderly fashion.

Program Testing

Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected. If the program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction. Some commonly used terms associated with testing are:

- **Failure:** This is a manifestation of an error (or defect or bug). But, the mere presence of an error may not necessarily lead to a failure.

- **Test case:** This is the triplet [I,S,O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.
- **Test suite:** This is the set of all test cases with which a given software product is to be tested.

Aim of testing

The aim of the testing process is to identify all defects existing in a software product. However for most practical systems, even after thoroughly carrying out the testing phase, it is not possible to guarantee that the software is error free. Because of the fact that the input data domain of most software products is very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume. Testing provides a practical way of reducing defects in a system and increasing the users' confidence in a developed system.

Testing

After design phase is over software products are normally tested first at the individual component (or unit) level. This is referred to as testing in the small. After testing all the components individually, the components are slowly integrated and tested at each level of integration (integration testing). Finally, the fully integrated system is tested (called system testing). Integration and system testing are known as testing in the large.

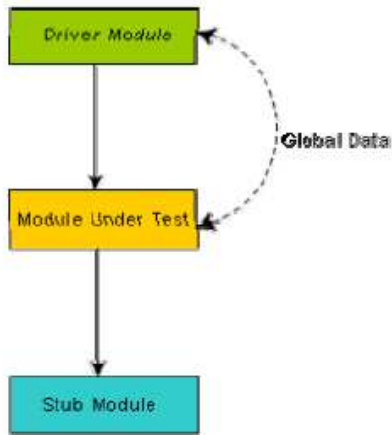
Unit testing

Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation. In order to test a single module, a complete environment is needed to provide all that is necessary for execution of the module. That is, besides the module under test itself, the following steps are needed in order to be able to test the module:

- The procedures belonging to other modules that the module under test calls.
- Non-local data structures that the module accesses.
- A procedure to call the functions of the module under test with appropriate parameters.

Modules required to provide the necessary environment (which either call or are called by the module under test) is usually not available until they too have been unit tested, stubs and drivers are designed to provide the complete environment for a module. The role of stub and driver modules is pictorially shown below. A stub procedure is a dummy procedure that has the same I/O parameters as the given procedure but has a highly simplified behaviour. For example, a stub procedure may produce the expected behaviour using a simple table lookup mechanism and it may use global data. A driver

module contains the nonlocal data structures accessed by the module under test, and would also have the code to call the different functions of the module with appropriate parameter values.



Unit testing with the help of driver and stub modules

Black box testing

In black box testing test cases are designed using only functional specification of s/w. Test cases designed are based on the input/output behaviour(i.e functional behaviour) and does not required any knowledge of internal structure. For this reason black box testing is also known as functional testing.

The following are the two main approaches to designing black box test cases.

- Equivalence class partitioning
- Boundary value analysis

Equivalence Class Partitioning

In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behaviour of the program is similar for every input data belonging to the same equivalence class. The main idea behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class. Equivalence classes for a software can be designed by examining the input data and output data. The following are some general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

Example#1: For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000, there are three equivalence classes: The set of negative integers, the set of integers in the range of 0 and 5000, and the integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes and a possible test set can be: {-5,500,6000}.

Example#2: Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form $y=mx + c$.

The equivalence classes are the following:

- Parallel lines ($m1=m2, c1 \neq c2$)
- Intersecting lines ($m1 \neq m2$)
- Coincident lines ($m1=m2, c1=c2$)

Now, selecting one representative value from each equivalence class, the test suit (2, 2) (2, 5), (5, 5) (7, 7), (10, 10) (10, 10) are obtained.

Boundary Value Analysis

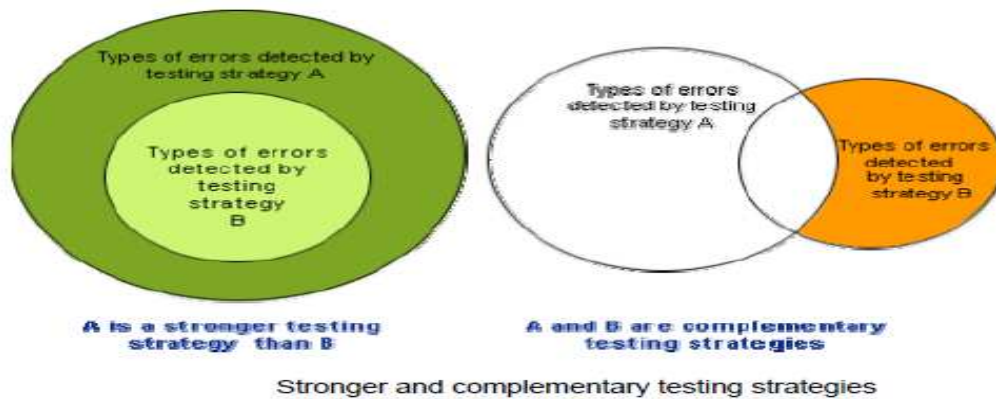
Boundary value analysis based test suit design involves designing test cases using the values at the boundaries of the different equivalent classes. A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. So in order to design boundary value test cases, it is required to examine the equivalent class to check, if any of equivalence class contain a range of values. Programmers often fail to see the special processing required by the input values that lie at the boundary of the different equivalence classes.

For example, programmers may improperly use $<$ instead of \leq , or conversely \leq for $<$. Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes.

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

White box testing

White box testing is also known as structure testing or glass testing. One white-box testing strategy is said to be *stronger than* another strategy, if all types of errors detected by the first testing strategy is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. When two testing strategies detect errors that are different at least with respect to some types of errors, then they are called *complementary*. The concepts of stronger and complementary testing are schematically illustrated .



Basic concepts associated with the white box testing are testing strategy that can be either:

-coverage based testing

- statement coverage
- branch coverage
- condition coverage
- path coverage

-fault based testing

Coverage based testing:

In this strategy certain program are executed to discover the error. And the specific elements required by the strategy are called testing criteria of the strategy.

Statement coverage

The statement coverage strategy aims to design test cases so that every statement in a program is executed at least once. The principal idea governing the statement coverage strategy is that unless a statement is executed, it is very hard to determine if an error exists in that statement. Unless a statement is executed, it is very difficult to observe whether it causes failure due to some illegal memory access, wrong result computation, etc. However, executing some statement once and observing that it behaves properly for that, input value is no guarantee that it will behave correctly for all input values. In the following, designing of test cases using the statement coverage strategy have been shown.

Example: Consider the Euclid's GCD computation algorithm:

```
int compute_gcd(x, y)
int x, y;
{
1 while (x! = y){
2 if (x>y) then
```

```

3 x= x - y;
4 else y= y - x;
5 }
6 return x;
}

```

By choosing the test set $\{(x=3, y=3), (x=4, y=3), (x=3, y=4)\}$, we can exercise the program such that all statements are executed at least once.

Branch coverage

In the branch coverage-based testing strategy, test cases are designed to make each branch condition to assume true and false values in turn. Branch testing is also known as **edge testing** as in this testing scheme, each edge of a program's control flow graph is traversed at least once.

Branch testing guarantees statement coverage and thus is a stronger testing strategy compared to the statement coverage-based testing. For Euclid's GCD computation algorithm, the test cases for branch coverage can be $\{(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)\}$.

Condition coverage

In this structural testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, in the conditional expression $((c1.and.c2).or.c3)$, the components $c1$, $c2$ and $c3$ are each made to assume both true and false values. Branch testing is probably the simplest condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. Thus, condition testing is a stronger testing strategy than branch testing and branch testing is stronger testing strategy than the statement coverage-based testing. For a composite conditional expression of n components, for condition coverage, 2^n test cases are required. Thus, for condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, a condition coverage-based testing technique is practical only if n (the number of conditions) is small.

Path coverage

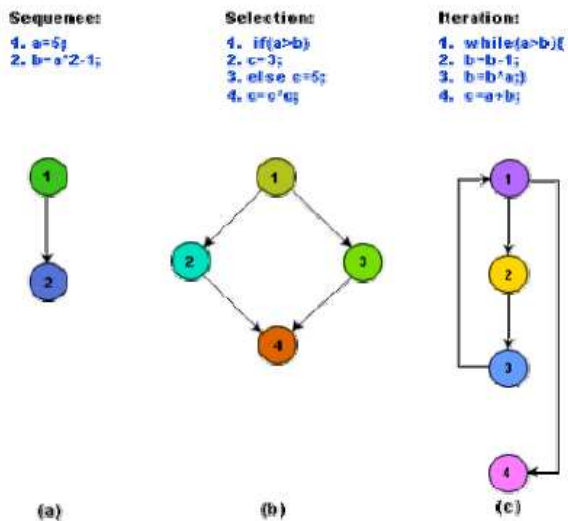
The path coverage-based testing strategy requires to design test cases such that all linearly independent paths in the program are executed at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

Control Flow Graph (CFG)

A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program. In order to draw the control flow graph of a program, all the statements of a program must be numbered first. The different

numbered statements serve as nodes of the control flow graph as shown below. An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node.

The CFG for any program can be easily drawn by knowing how to represent the sequence, selection, and iteration type of statements in the CFG. After all, a program is made up from these types of statements, how the CFG for these three types of statements can be drawn. We have to note that for the iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore the control flow from the last statement of the loop is always to the top of the loop. Using these basic ideas, the CFG of Euclid's GCD computation algorithm can be drawn as shown below.

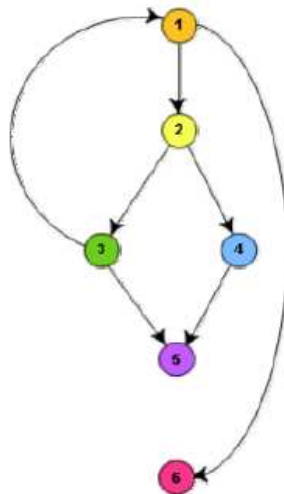


CFG for (a) sequence, (b) selection, and (c) iteration type constructs

```

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

```



(a) Example program

(b) Control Flow Graph

Control flow diagram

Path

A path through a program is a node and edge sequence from the starting node to a terminal node of the control flow graph of a program. There can be more than one terminal node in a program. Writing test cases to cover all the paths of a typical program is impractical. For this reason, the path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths.

Linearly independent path

A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent. This is because; any path having a new node automatically implies that it has a new edge. But, a path that is sub path of another path is not considered to be a linearly independent path.

Control flow graph

In order to understand the path coverage-based testing strategy, it is very necessary to understand the control flow graph (CFG) of a program. Control flow graph (CFG) of a program has been discussed earlier.

Linearly independent path

The path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths. Linearly independent paths have been discussed earlier.

Cyclomatic complexity

For more complicated programs it is not easy to determine the number of independent paths of the program. McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, it is very simple to compute. Thus, the McCabe's cyclomatic complexity metric provides a practical way of determining the maximum number of linearly independent paths in a program. Though the McCabe's metric does not directly identify the linearly independent paths, but it informs approximately how many paths to look for. There are three different ways to compute the cyclomatic complexity.

Method 1:

Given a control flow graph G of a program, the cyclomatic complexity $V(G)$ can be computed as:

$$V(G) = E - N + 2$$

where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of example shown in fig. 10.4, $E=7$ and $N=6$. Therefore, the cyclomatic complexity = $7-6+2 = 3$.

Method 2:

An alternative way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows: **$V(G) = \text{Total number of bounded areas} + 1$**

In the program's control flow graph G , any region enclosed by nodes and edges can be called as a bounded area. It is an easy way to determine the McCabe's cyclomatic complexity, But, what if the graph G is not planar, i.e. however you draw the graph, two or more edges intersect? Actually, it can be shown that structured programs always yield planar graphs. But, presence of GOTO's statement can easily add intersecting edges.

Therefore, for non-structured programs, this way of computing the McCabe's cyclomatic complexity cannot be used. The number of bounded areas increases with the number of decision paths and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability. For the CFG example shown above, from a visual examination of the CFG the number of bounded areas is 2. Therefore the cyclomatic complexity, computing with this method is also $2+1 = 3$. This method provides a very easy way of computing the cyclomatic complexity of CFGs, just from a visual examination of the CFG.

On the other hand, the other method of computing CFGs is more amenable to automation, i.e. it can be easily coded into a program which can be used to determine the cyclomatic complexities of arbitrary CFGs.

Method 3:

The cyclomatic complexity of a program can also be easily computed by computing the number of decision statements of the program. If N is the number of decision statement of a program, then the McCabe's metric is equal to $N+1$. From above example 2 decision statements ie if a and while so $2+1=3$

Data flow-based testing

Data flow-based testing method selects test paths of a program according to the locations of the definitions and uses of different variables in a program. For a statement numbered S , let

**DEF(S) = {X/statement S contains a definition of X}, and
USES(S) = {X/statement S contains a use of X}**

For the statement **S: a=b+c;** DEF(S) = {a}. USES(S) = {b,c}. The definition of variable X at statement S is said to be live at statement $S1$, if there exists a path from statement S to statement $S1$ which does not contain any definition of X . The definition-use chain (or DU chain) of a variable X is of form $[X, S, S1]$, where S and $S1$ are statement numbers, such that $X \in \text{DEF}(S)$ and $X \in \text{USES}(S1)$, and the definition of X in the statement S is live at statement $S1$. One simple data flow testing strategy is to require that every DU chain be covered at least once. Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements.

Fault based testing

Example of fault base testing the mutaion testing.

Mutation testing

In mutation testing, test cases are designed to detect specific type of fault in a program. In mutation testing s/w first tested by initial test suit that is designed by other white box testing strategies. After the initial testing is complete, mutation testing is taken up. The main idea behind mutation testing is to make few arbitrary changes to a program at a time. Each time the program is changed, it is called as a mutated program and the change effected is called as a mutant. A mutated program is tested against the full test suite of the program. If there exists at least one test case in the test suite for which a mutant gives an incorrect result, then the mutant is said to be dead. If a mutant remains alive even after all the test cases have been exhausted, the test data is enhanced to kill the mutant.

The process of generation and killing of mutants can be automated by predefining a set of primitive changes that can be applied to the program. These primitive changes can be

alterations such as changing an arithmetic operator, changing the value of a constant, changing a data type, etc. A major disadvantage of the mutation-based testing approach is that:

-it is computationally very expensive, since a large number of possible mutants can be generated.

-Since mutation testing generates a large number of mutants and requires us to check each mutant with the full test suite, it is not suitable for manual testing.

Mutation testing should be used in conjunction of some testing tool which would run all the test cases automatically.

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.

Unit Testing

Unit testing comprises 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 part that can test it in great detail, and certainly in greater detail there 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

Function test:

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.

Performance testing:

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:

Stress tests are those tests designed to intentionally break the unit. Through this testing we can learn about the strengths and limitations of a program by examining the manner in which a program unit breaks.

Structure Tests:

Structure tests are concerned with exercising the internal logic part 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.

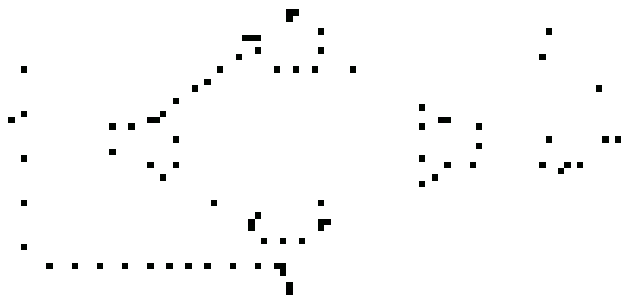


Figure 6.7

Even if it were possible to successfully test all paths through a program, correctness would not be guaranteed by path testing or by this testing. Because there would be the possible of program of missing the 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.

Program errors can be again classified as

- missing path errors previously discussed,
- computational errors and
- domain errors.

Tai has observed that $N + m1$ linearly independent test cases are required to establish computational correctness of a program that performs only linear calculations on N input variables. By linear calculations, we can conclude that all computations are linear functions of the input variables when symbolic execution techniques)

A domain error occurs when a program traverses the wrong path because of an incorrect predicate in a branching statement. White and Cohen have shown that, for very simple programs, domain errors can be detected by the test cases that are either on or near the borders of the path domains.

The borders of a path domain are determined by the inequalities in the test cases.

1. Coincidental correctness does not occur for any test case. If a test traverses an incorrect path, the output values are different from the values that would be computed on the correct path.
2. There are no missing paths associated with the path being tested.
3. Each border is produced by a predicate having only one relational operator.
4. Each adjacent path domain computes a function different from the function computed by the path being tested.
5. The border being tested is a linear function of the input variables. If the border is in correct border is linear.
6. The input space is continuous rather than discrete.
7. Each test case of the border (each off point) is a small distance, epsilon, from the border.

Integration Testing

After tested each module individually, integrated testing phase is proceeded. Testing approach can be either top-up or bottom-up. 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, which is 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 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 interfaces. 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, sometimes testing subsystem is not possible due to the complexity of module interface. Therefore test cases must be chosen carefully to exercise the 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 occurs 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.

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.

Levels of Testing

When we go through the testing process 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 further 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.

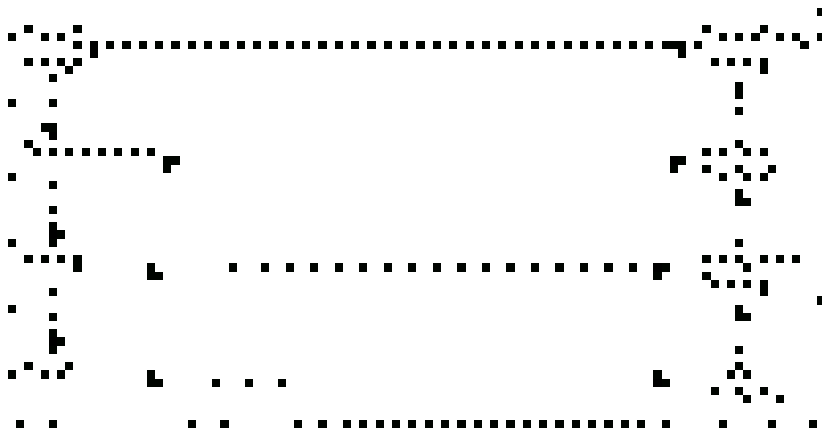


Figure 6.8

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 6.8.

ART OF DEBUGGING

Debugging means fixing the error. It is carried by the development team with intentionally removed of the error. Debugging is a necessary process in almost any new software or hardware development process, whether a commercial product or an enterprise or personal application program. For complex products, debugging is done

as the result of the unit test for the smallest unit of a system, again at component test when parts are brought together, again at system test when the product is used with other existing products, and again during customer beta test, when users try the product out in a real world situation. Program that has lots of bugs is referred to as "buggy."

Debugging tools (called *debuggers*) help identify coding errors at various development stages. Some programming language packages include a facility for checking the code for errors as it is being written.

Need for debugging

Once errors are identified in a program code, it is necessary to first identify the precise program statements responsible for the errors and then to fix them. Identifying errors in a program code and then fix those up are known as debugging.

Debugging approaches

The following are some of the approaches popularly adopted by programmers for Debugging.

Brute Force Method:

This is the most common method of debugging but is the least efficient method. In this approach, the program is loaded with print statements to print the intermediate values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly.

Backtracking:

This is also a fairly common approach. In this approach, beginning from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large thus limiting the use of this approach.

Cause Elimination Method:

In this approach, a list of causes which could possibly have contributed to the error symptom is developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

Program Slicing:

This technique is similar to back tracking. Here the search space is reduced by defining slices. A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

Debugging guidelines

Debugging is often carried out by programmers based on their ingenuity. The following are some general guidelines for effective debugging:

- Many times debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the system design and implementation may require an inordinate amount of effort to be put into debugging even simple problems.
- Debugging may sometimes even require full redesign of the system. In such cases, a common mistake that novice programmers often make is attempting not to fix the error but its symptoms.
- One must be beware of the possibility that an error correction may introduce new errors. Therefore after every round of error-fixing, regression testing must be carried out.

Software Maintenance

Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes .A common perception of maintenance is that it merely involves fixing defects. However, one study indicated that the majority, over 80%, of the maintenance effort is used for non-corrective actions. This perception is perpetuated by users submitting problem reports that in reality are functionality enhancements to the system. Software maintenance is needed to correct error enhance feature, portability to new platform.

Necessity of software maintenance

- Software maintenance is becoming an important activity of a large number of software organizations.
- This is no surprise, given the rate of hardware obsolescence, the immortality of a software product per se, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features.
- When the hardware platform is changed, and a software product performs some low-level functions, maintenance is necessary.
- Whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface.
- For instance, a software product may need to be maintained when the operating system changes.

Thus, every software product continues to evolve after its development through maintenance efforts. Therefore it can be stated that software maintenance is needed to correct errors, enhance features, port the software to new platforms, etc.

Software maintenance task:

There are basically three types of software maintenance. These are:

- **Corrective:**

Corrective maintenance of a software product is necessary to rectify the bugs observed while the system is in use.

- **Adaptive:**

A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.

- **Perfective:**

A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

Problems associated with software maintenance

1. Software maintenance work typically is much more expensive than what it should be and takes more time than required.
2. In software organizations, maintenance work is mostly carried out using adhoc techniques.
3. There are no systematic and planned activities for maintaining software.
4. Software maintenance has a very poor image in industry. Therefore, an organization often cannot employ bright engineers to carry out maintenance work.
5. Even though maintenance suffers from a poor image, the work involved is often more challenging than development work.
6. During maintenance it is necessary to thoroughly understand someone else's work and then carry out the required modifications and extensions.
7. Another problem associated with maintenance work is that the majority of software products needing maintenance are legacy products.

Legacy software products

A legacy system is hard to maintain. The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product. Many of the legacy systems were developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

Factors on which software maintenance activities depend

The activities involved in a software maintenance project are depends on several factors such as:

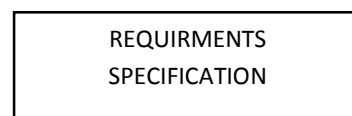
1. The extent of modification to the product required
2. The resources available to the maintenance team
3. The conditions of the existing product (e.g., how structured it is, how well documented it is, etc.)
4. The expected project risks, etc.

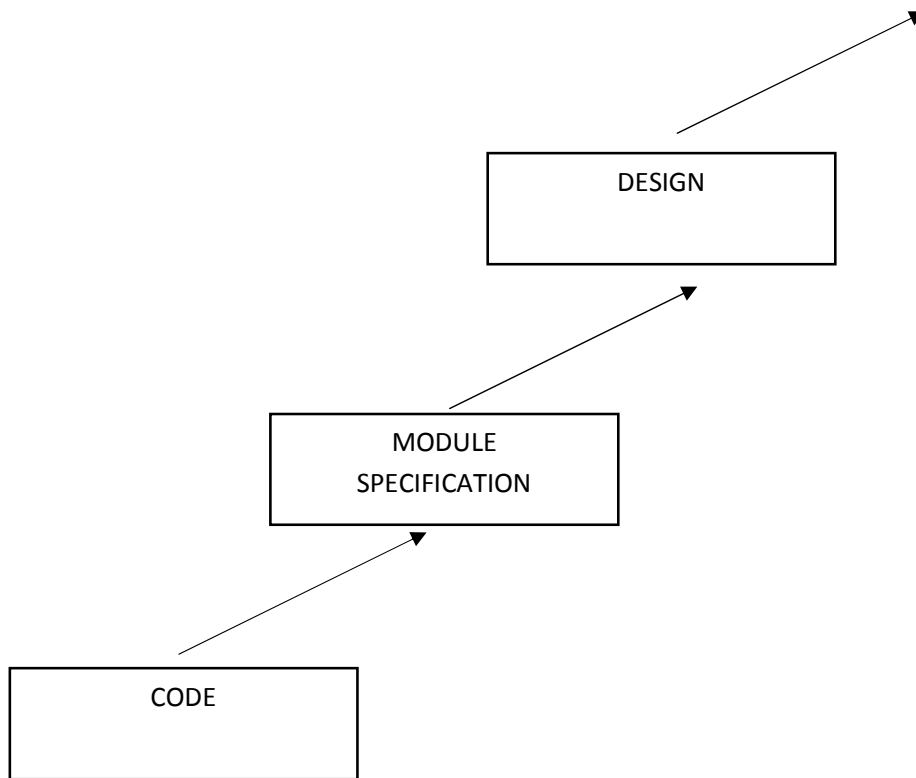
When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents. But more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

Software reverse engineering:

Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understand ability of a system and to produce the necessary documents for a legacy system. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

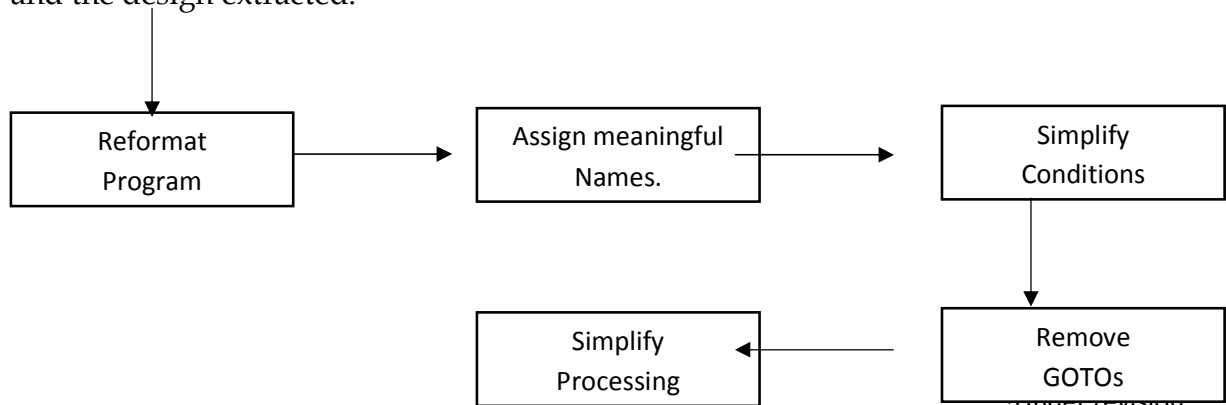
The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understand ability, without changing of its functionalities. A process model for reverse engineering has been shown in below fig. A program can be reformatted using any of the several available pretty printer programs which layout the program neatly. Many legacy software products with complex control structure and unthoughtful variable names are difficult to comprehend. Assigning meaningful variable names is important because meaningful variable names are the most helpful thing in code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible. Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.





[A process model for reverse engineering]

After the cosmetic changes have been carried out on a legacy software, the process of extracting the code, design, and the requirements specification can begin. These activities are schematically shown in fig. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code. The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.



[Cosmetic changes carried out before reverse engineering]

Legacy software products:

It is prudent to define a legacy system as any software system that is hard to maintain. The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product. Many of the legacy systems were developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

Factors on which software maintenance activities depend:

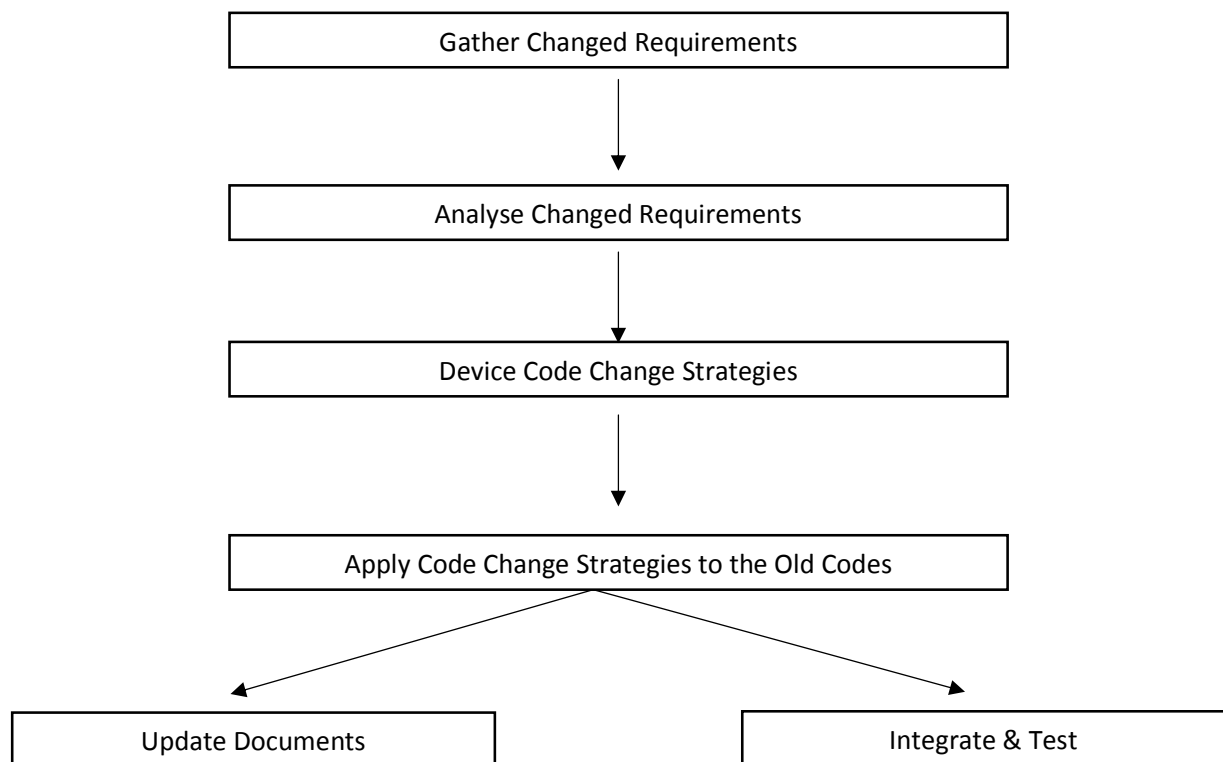
The activities involved in a software maintenance project are not unique and depend on several factors such as:

- The extent of modification to the product required
- The resources available to the maintenance team
- The conditions of the existing product (e.g., how structured it is, how well documented it is, etc.)
- The expected project risks, etc.

When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents. But more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

Software maintenance process models:

Two broad categories of process models for software maintenance can be proposed. The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later. This maintenance process is graphically presented in fig. In this approach, the project starts by gathering the requirements for changes. The requirements are next analysed to formulate the strategies to be adopted for code change. At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle time, especially for projects involving unstructured and inadequately documented code.

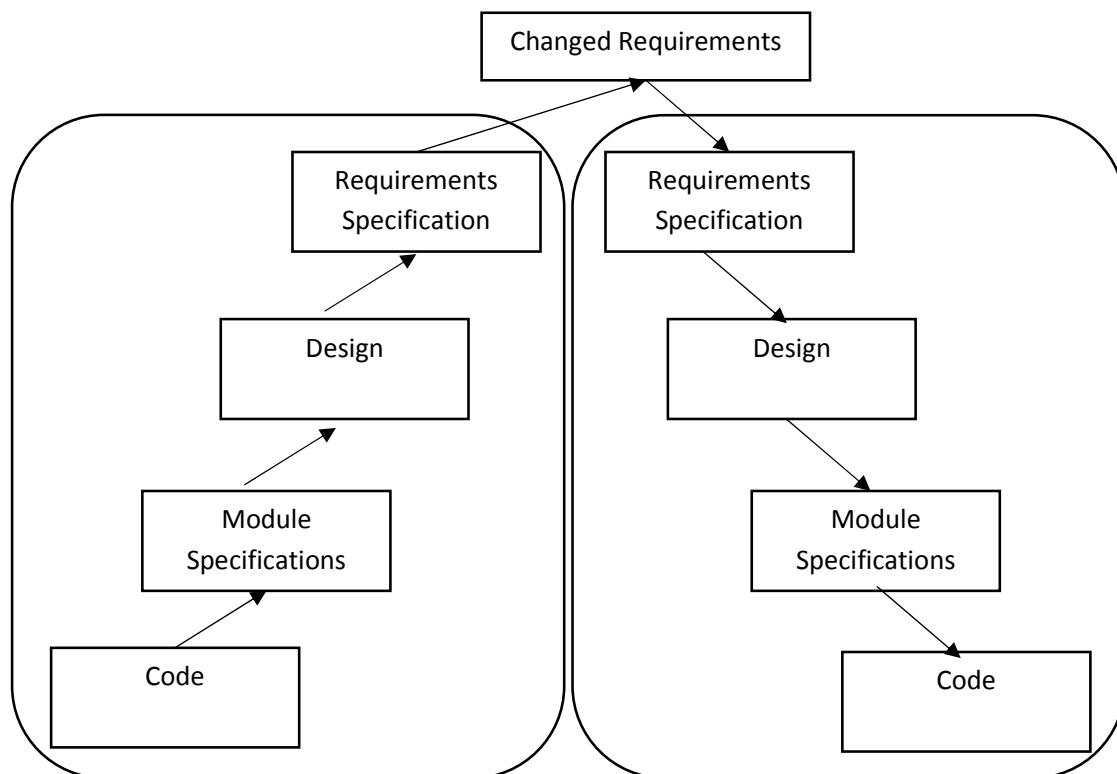


[Maintenance process model 1]

The second process model for software maintenance is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an

approach is also known as software reengineering. This process model is depicted in the below fig. The reverse engineering cycle is required for legacy products. During the reverse engineering, the old code is analysed (abstracted) to extract the module specifications. The module specifications are then analysed to produce the design. The design is analysed to produce the original requirements specification. The change requests are then applied to this requirements specification to arrive at the new requirements specification. At the design, module specification, and coding a substantial reuse is made from the reverse engineered products. An important advantage of this approach is that it produces a more structured design compared to what the original product had, produces good documentation, and very often results in increased efficiency.

- Reengineering might be preferable for products which exhibit a high failure rate.
- Reengineering might also be preferable for legacy products having poor design and code structure.



[Maintenance process model 2]

Software reengineering:

Software reengineering is a combination of two consecutive processes i.e. software reverse engineering and software forward engineering as shown in the fig.

Estimation of approximate maintenance cost:

It is well known that maintenance efforts require about 60% of the total life cycle cost for a typical software product. However, maintenance costs vary widely from one application domain to another. For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost. Boehm [1981] proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model. Boehm's maintenance cost estimation is made in terms of a quantity called the Annual Change Traffic (ACT). Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion.

$$\text{ACT} = (\text{KLOC added} + \text{KLOC deleted}) / \text{KLOC total}$$

Where, KLOC added is the total kilo lines of source code added during maintenance. KLOC deleted is the total KLOC deleted during maintenance. Thus, the code that is changed, should be counted in both the code added and the code deleted. The annual change traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost:

$$\text{Maintenance cost} = \text{ACT} \times \text{development cost}$$

Most maintenance cost estimation models, however, yield only approximate results because they do not take into account several factors such as experience level of the engineers, and familiarity of the engineers with the product, hardware requirements, software complexity, etc.

Software configuration management

The results (also called as the deliverables) of a large software development effort typically consist of a large number of objects, e.g. source code, design document, SRS document, test document, user's manual, etc. These objects are usually referred to and modified by a number of software engineers throughout the life cycle of the software. The state of all these objects at any point of time is called the configuration of the

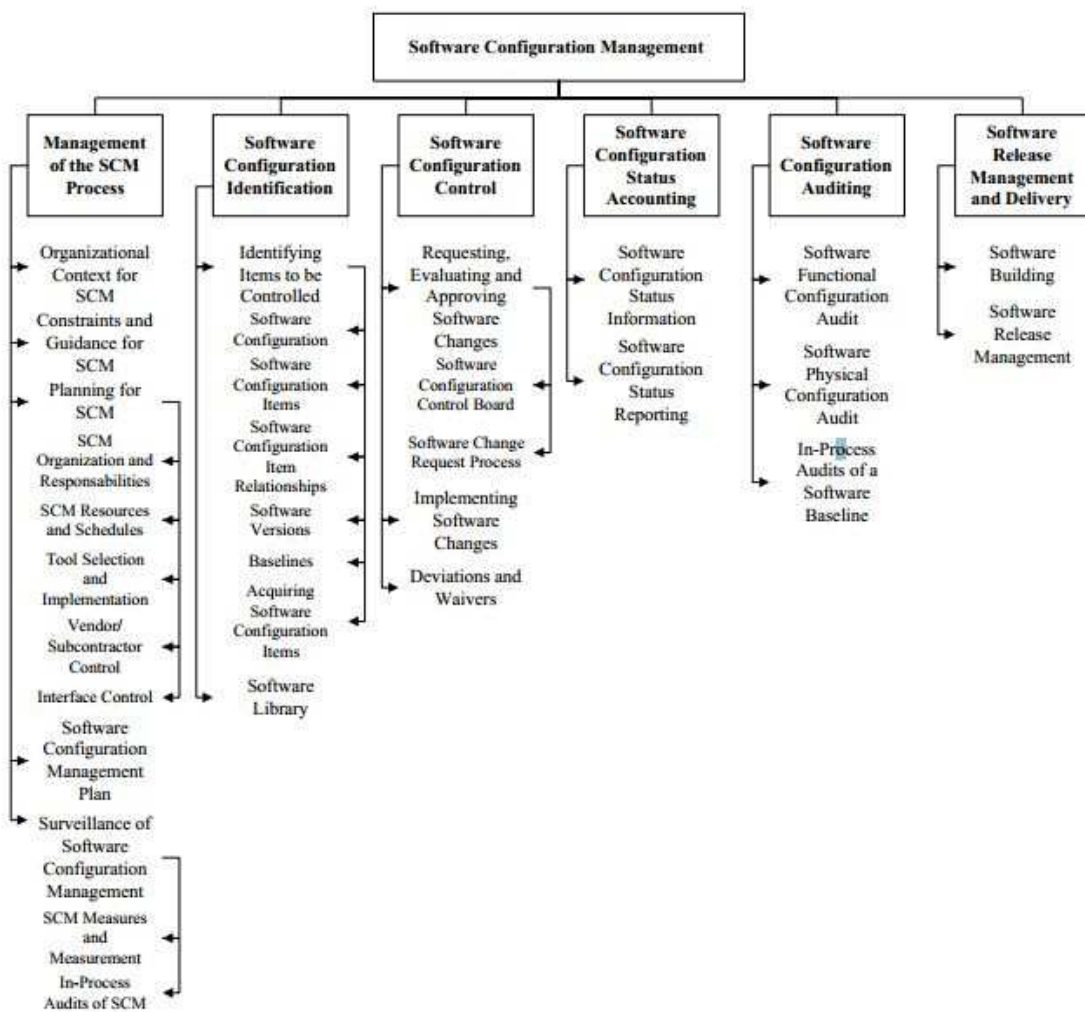
software product. The state of each deliverable object changes as development progresses and also as bugs is detected and fixed.

- Configuration management determines clearly about the items that make up the software or system. These items include source code, test scripts, third-party software, hardware, data and both development and test documentation.
- Configuration management is also about making sure that these items are managed carefully, thoroughly and attentively during the entire project and product life cycle.
- Configuration management has a number of important implications for testing. Like configuration management allows the testers to manage their testware and test results using the same configuration management mechanisms.
- Configuration management also supports the build process, which is important for delivery of a test release into the test environment. Simply sending Zip archives by e-mail will not be sufficient, because there are too many opportunities for such archives to become polluted with undesirable contents or to harbor left-over previous versions of items. Especially in later phases of testing, it is critical to have a solid, reliable way of delivering test items that work and are the proper version.

Configuration management is a topic that is very complex. So, advanced planning is very important to make this work. During the project planning stage – and perhaps as part of your own test plan – make sure that configuration management procedures and tools are selected. As the project proceeds, the configuration process and mechanisms must be implemented, and the key interfaces to the rest of the development process should be documented.

Techniques of SCM Process

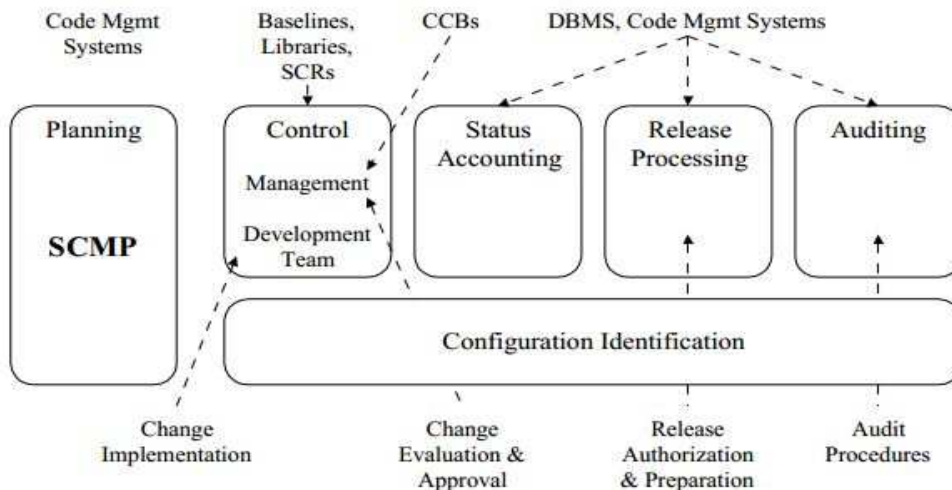
SCM is closely related to the software quality assurance (SQA) activity. The goals of SQA can be characterized as monitoring the software and its development process, ensuring compliance with standards and procedures, and ensuring that product, process, and standards defects are visible to management. SCM activities help in accomplishing these SQA goals.



Tool Selection and Implementation

Different types of tool capabilities, and procedures for their use, support the SCM activities. Depending on the situation, these tool capabilities can be made available with some combination of manual tools, automated tools providing a single SCM capability, automated tools integrating a range of SCM capabilities, or integrated tool environments that serve the needs of multiple participants in the software development process.

Automated tool support becomes increasingly important, and increasingly difficult to establish, as projects grow in size and as project environments get more complex. These tool capabilities provide support for: the SCM Library, the software change request (SCR) and approval procedures, code (and related work products) and change management tasks, reporting software configuration status and collecting SCM measurements, software auditing, managing and tracking software documentation, performing software builds, and managing and tracking software releases and their distribution. The use of tools in these areas increases the potential for obtaining product and process measurements to be used for project management and process improvement purposes.



In this example, code management systems support the operation of software libraries by controlling access to library elements, coordinating the activities of multiple users, and helping to enforce operating procedures. Other tools support the process of building software and release documentation from the software elements contained in

the libraries. Tools for managing software change requests support the change control procedures applied to controlled software items. Other tools can provide database management and reporting capabilities for management, development, and quality assurance activities. As mentioned above, the capabilities of several tool types might be integrated into SCM systems, which, in turn, are closely coupled to various other software activities. The planning activity assesses the SCM tool needs for a given project within the context of the software engineering environment to be used and selects the tools to be used for SCM. The planning considers issues that might arise in the implementation of these tools, particularly if some form of culture change is necessary.

Concept of Software Reliability

Most important and dynamic characteristic of software is its reliability. Informally, the reliability of a software system is a measure of how well it provides the services expected of it by its users but a useful formal definition of reliability is much harder to express. Software reliability metrics such as 'mean time between failures' may be used but they do not take into account the subjective nature appropriate and useful.

Users do not consider all services to be of equal importance and a system might be viewed as unreliable if it ever fails to provide some critical service. For example, say a system was used to control braking on an aircraft but failed to work a single set of very rare conditions. If the aircraft crashed because these failure conditions occurred, pilots of similar aircraft would (reasonably) regard the software as unreliable.

On the other hand, say a comparable software system provided some visual indication of its actions to the pilot. Assume this failed once per month in a predictable way without the main system function being affected and other indicators showed that the control system was working normally. In spite of frequent failure, pilots would not consider that software as unreliable as the system, which caused the catastrophic failure.

Reliability is a dynamic system characteristic, which is a function of the number of software failures. A software failure is an execution event where the software behaves in an unexpected way. This is not the same as a software fault, which is a static program characteristic. Software faults cause software failures when the faulty code is executed with a particular set of inputs. Faults do not always manifest themselves as failures so the reliability depends on how the software is used. It is not possible to produce a single, universal statement of the software reliability.

Software faults are not just program defects. Unexpected behaviour can occur in circumstances where the software conforms to its requirements themselves are complete. Omissions in software documentations can also lead to unexpected behaviour, although the software may not contain defects.

Reliability depends on how the software is used, so it cannot be specified absolutely. Each user uses a program in different ways so faults, which affect the reliability of the system for one user, may never manifest themselves under a different mode of working. Reliability can only be accurately specified if the normal software operational profile is also specified.

As reliability is related to the probability of an error occurring in operational use, a program may contain known faults but may still never be seen select an erroneous input; the program always appears to be reliable. Furthermore, experienced users may 'work around' known software faults and deliberately avoid using features, which they know to be faulty. Repairing the faults in these features may make no practical difference to the reliability as perceived by these users.

For example, the word processor used to write this book has an automatic hyphenation capability. This facility is used when text columns are short and any faults might manifest themselves when users produce multi-column documents. I never use hyphenation so, from my viewpoint, faults in the hyphenation code do not affect the reliability of the word

Error, Fault and Failure

So far, we have used the intuitive meaning of the term *error* to refer to problems in requirements, design, or code. Sometimes error, fault, and failure are used interchangeably, and sometimes they refer to different concepts. Let us start by defining these concepts clearly. We follow the IEEE definitions for these terms.

The term *error* is used in two different ways. It refers to the discrepancy between a computed, observed, or measured value and the true, specified, or theoretically correct value. That is, error refers to the difference between the actual output of a software and the correct output. In this interpretation, error is essentially a measure of the difference between the actual and the ideal. Error is also used to refer to human action those results in software containing a defect or fault. This definition is quite general and encompasses all the phases.

Fault is a condition that causes a system to fail in performing its required function. A fault is the basic reason for software malfunction and is synonymous with the commonly used term *bug*. The term error is also often used to refer to defects (taking a variation of the second definition of error). In this book, we will continue to use the terms in the manner commonly used, and no explicit distinction will be made between errors and faults, unless necessary. It should be noted that the only " faults that a software has are "design faults"; there is no wear and tear in software.

Failure is the inability of a system or component to perform a required function, according to its specifications. A software failure occurs if the behaviour of the software is different from the specified behaviour. Failures may be caused due to functional or performance reasons. A failure is produced only when there is a fault in the system. However, presence of a fault does not guarantee a failure. In other words, faults have the potential to cause failures and their presence is a necessary but not a sufficient condition for failure to occur. Definition does not imply that a failure must be *observed*. It is possible that a failure may occur but not be detected.

There are some implications of these definitions. Presence of an error (in the state) implies that a failure must have occurred, and the observance of a failure implies that a fault must be present in the system. However, the presence of a fault does not imply that a failure must occur. The presence of a fault in a system only implies that the fault has a *potential* to cause a failure to occur. Whether a fault actually manifests itself in certain time duration depends on many factors. This means that if we observe the behaviour of a system for some time and we do not observe any errors, we cannot say anything about the presence or absence of faults in the system. If, on the other hand, we observe some failure in this duration, we can say that there are some faults in the system.

Hardware reliability vs. software reliability

Reliability behaviour for hardware and software are very different. For example, hardware failures are inherently different from software failures. Most hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part. On the other hand, a software product would continue to fail until the error is tracked down and either the design or the code is changed. For this reason, when a hardware is repaired its reliability is maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug introduces new errors). To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, inter-failure times remain constant). On the other hand, software reliability study aims at reliability growth (i.e. inter-failure times increase). The change of failure rate over the product lifetime for a typical hardware and a software product are sketched below. For hardware products, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed.

Reliability metrics

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the SRS (software requirements specification) document. In order to be able to do this, some metrics are needed to quantitatively express the reliability of a software product. A good reliability measure should be

observer-dependent, so that different people can agree on the degree of reliability a system has. For example, there are precise techniques for measuring performance, which would result in obtaining the same performance value irrespective of who is carrying out the performance measurement. However, in practice, it is very difficult to formulate a precise reliability measurement technique. The next base case is to have measures that correlate with reliability. There are six reliability metrics which can be used to quantify the reliability of software products.

Rate of occurrence of failure (ROCOF).

- ROCOF measures the frequency of occurrence of unexpected behavior (i.e. failures).
- ROCOF measure of a software product can be obtained by observing the behavior of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval.
- It

Mean Time To Failure (MTTF).

- MTTF is the average time between two successive failures, observed over a large number of failures.
- To measure MTTF, we can record the failure data for n failures. An MTTF of 500 means that one failure can be expected in every 500 time unit.
- MTTF is relevant for system with long transaction i.e when processing takes long time

Mean Time To Repair (MTTR).

- Once failure occurs, some time is required to fix the error.
- MTTR measures the average time it takes to track the errors causing the failure and to fix them.

Mean Time Between Failure (MTBF).

- MTTF and MTTR can be combined to get the MTBF metric: $MTBF = MTTF + MTTR$.
- MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, time measurements are real time and not the execution time as in MTTF.

Probability of Failure on Demand (POFOD).

- This metric does not explicitly involve time measurements.
- POFOD measures the likelihood of the system failing when a service request is made.
- For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure.

Availability.

- Availability of a system is a measure of how likely shall the system be available for use over a given period of time.

- This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs.
- This metric is important for systems such as telecommunication systems, and operating systems, which are supposed to be never down and where repair and restart time are significant and loss of service during that time is important.

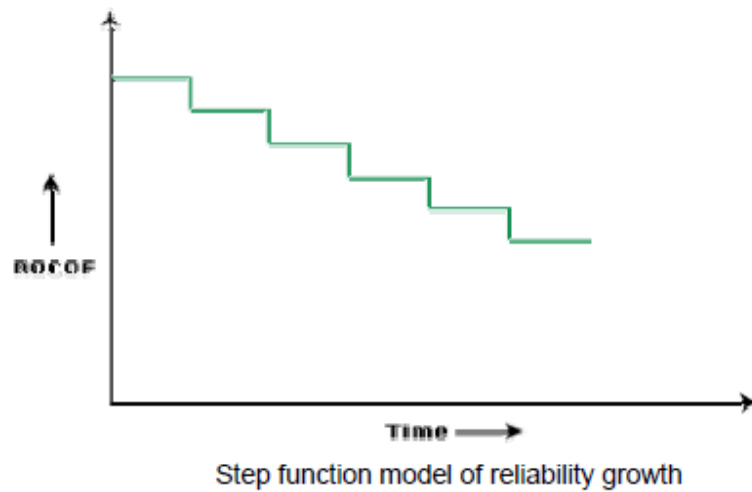
$$\text{AVAILABILITY} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})} \times 100$$

Reliability growth models

A reliability growth model is a mathematical model of how software reliability improves as errors are detected and repaired. A reliability growth model can be used to predict when (or if at all) a particular level of reliability is likely to be attained. Thus, reliability growth modelling can be used to determine when to stop testing to attain a given reliability level. Although several different reliability growth models have been proposed, in this text we will discuss only two very simple reliability growth models.

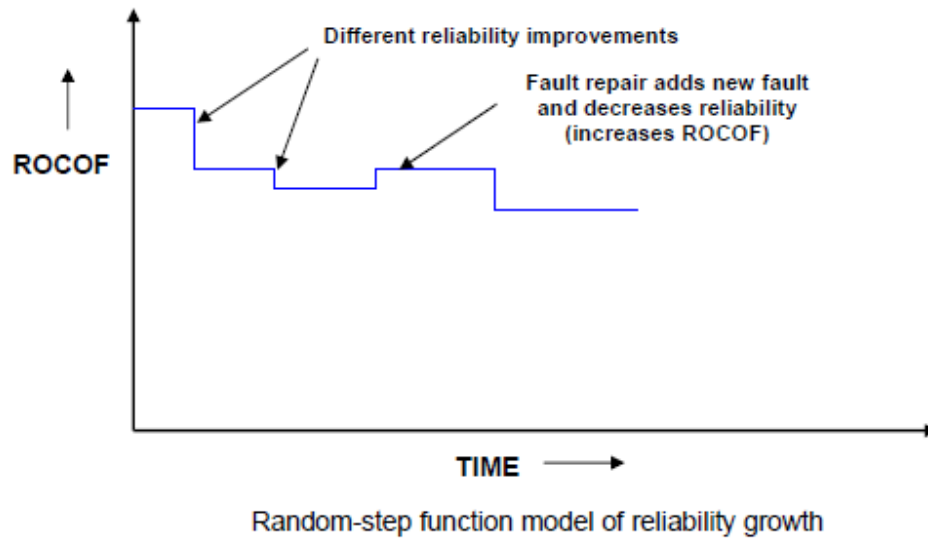
Jelinski and Moranda Model

The simplest reliability growth model is a step function model where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired. However, this simple model of reliability which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic since it is already known that correction of different types of errors contribute differently to reliability growth.



Littlewood and Verall's Model

This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It also models the fact that as errors are repaired, the average improvement in reliability per repair decreases. It treats an error's contribution to reliability improvement to be an independent random variable having Gamma distribution. This distribution models the fact that error corrections with large contributions to reliability growth are removed first. This represents diminishing return as test continues.



Software quality, SEI CMM and ISO-9001

Software quality:

A quality product is defined in terms of its fitness of purpose robustness. That is, a quality product does exactly what the users want it to do. For software products, fitness of purpose is usually interpreted in terms of satisfaction of the requirements laid down in the SRS document.

The modern view of a quality associates with a software product several quality factors such as the following:

- ✓ **Portability:** A software product is said to be portable, if it can be easily made to work in different operating system environments, in different machines, with other software products, etc. It should platform independent.
- ✓ **Usability:** A software product has good usability, if different categories of users (i.e. both expert and novice users) can easily invoke the functions of the product easy to handle.
- ✓ **Reusability:** A software product has good reusability, if different modules of the product can easily be reused to develop new products.
- ✓ **Correctness:** A software product is correct, if different requirements as specified in the SRS document have been correctly implemented.
- ✓ **Maintainability:** A software product is maintainable, if errors can be easily corrected as and when they show up, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

SEI Capability Maturity Model:

- SEI Capability Maturity Model (SEI CMM) helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits.
- SEI CMM can be used two ways: capability evaluation and software process assessment.

Capability evaluation:

Capability evaluation provides a way to assess the software process capability of an organization.

Software process assessment:

Software process assessment is used by an organization with the objective to improve its process capability.

- SEI CMM classifies software development industries into the following five maturity levels.

Level 1: Initial.

A software development organization at this level is characterized by ad hoc activities. Very few or no processes are defined and followed.

Level 2: Repeatable.

At this level, the basic project management practices such as tracking cost and schedule are established.

Level 3: Defined.

At this level the processes for both management and development activities are defined and documented.

Level 4: Managed.

At this level, the focus is on software metrics.

Level 5: Optimizing.

At this stage, process and product metrics are collected.

Personal software process:

- Personal Software Process (PSP) is a scaled down version of the industrial software process.
- PSP is suitable for individual use. SEI CMM does not tell software developers how to analyze, design, code, test, or document software products, but assumes that engineers use effective personal practices.
- PSP recognizes that the process for individual use is different from that for a team.

Six sigma:

- The purpose of Six Sigma is to improve processes to do things better, faster, and at lower cost.
- It can be used to improve every facet of business, from production, to human resources, to order entry, to technical support.
- Six Sigma can be used for any activity that is concerned with cost, timeliness, and quality of results.

Sub-methodologies: DMAIC and DMADV.

The Six Sigma DMAIC process (defines, measure, analyze, improve, control) is an improvement system for existing processes failing below specification and looking for incremental improvement.

The Six Sigma DMADV process (define, measure, analyze, design, verify) is an improvement system used to develop new processes or products at Six Sigma quality levels.

ISO 9000 certification:

- ISO certification serves as a reference for contract between independent parties.
- The ISO 9000 standard specifies the guidelines for maintaining a quality system.
- ISO 9000 specifies a set of guidelines for repeatable and high quality product development.

Types of ISO 9000 quality standards:

ISO 9000 is a series of three standards: ISO 9001, ISO 9002, and ISO 9003.

ISO 9001:

ISO 9001 applies to the organizations that engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most software development organizations.

ISO 9002:

ISO 9002 applies to those organizations which do not design products but are only involved in production.

ISO 9003:

ISO 9003 applies to organizations that are involved only in installation and testing of the products.

Salient features of ISO 9001 certification:

The salient features of ISO 9001 are as follows:

- ✓ All documents concerned with the development of a software product should be properly managed, authorized, and controlled.
- ✓ Important documents should be independently checked and reviewed for effectiveness and correctness.
- ✓ The product should be tested against specification.
- ✓ Several organizational aspects should be addressed e.g., management reporting of the quality team

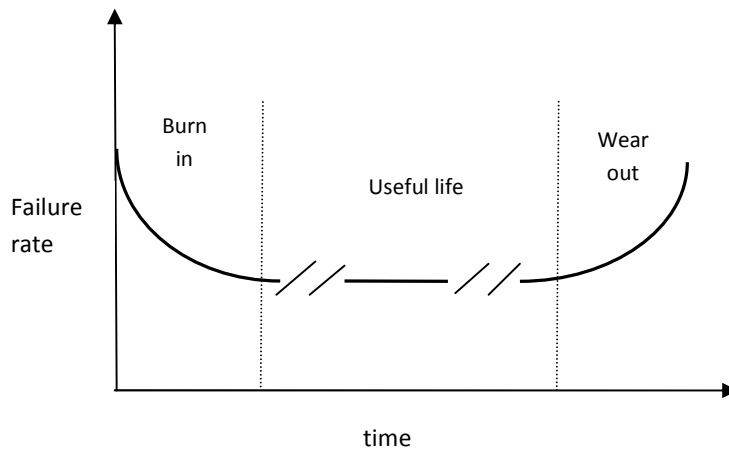
Software Reliability

- Reliability of software product can be defined as the probability of the product working correctly over a given period of time.
- Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time"
- It is obvious that a software product having a large number of defects is unreliable. It is also clear that the reliability of a system improves, if the number of defects in it is reduced.
- Reliability of a product depends not only on the number of latent errors but also on the exact location of the errors.
- The reliability figure of a software product is observer-dependent, and it is very difficult to absolutely quantify the reliability of the product.

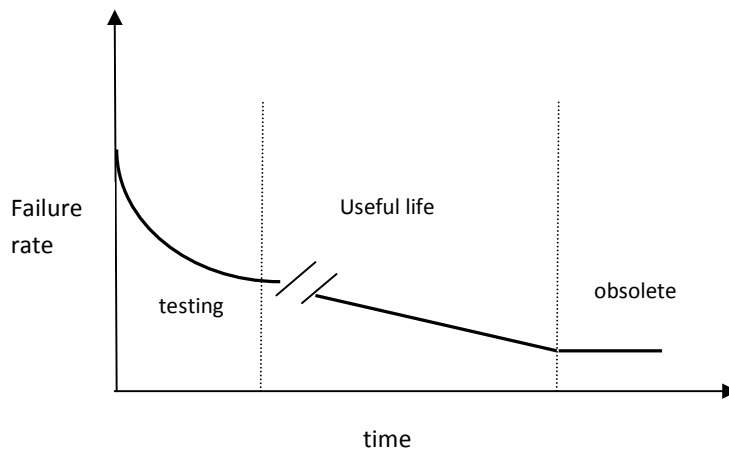
Hardware vs Software Reliability:

Reliability behaviour for hardware and software are very different. For example, hardware failures are inherently different from software failures. Most hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part.

On the other hand, a software product would continue to fail until the error is tracked down and either the design or the code is changed. For this reason, when a hardware is repaired its reliability is maintained at the level that existed before the failure occurred.



(a) Hardware product



(b) Software product

Reliability Metrics:

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the SRS (software requirements specification) document. In order to be able to do this, some metrics are needed to quantitatively express the reliability of a software product.

There are six reliability metrics which can be used to quantify the reliability of software products.

1. Rate of occurrence of failure (ROCOF) :

ROCOF measures the frequency of occurrence of unexpected behaviour (i.e. failures).

ROCOF measure of a software product can be obtained by observing the behaviour of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval.

2. Mean Time To Failure (MTTF)

MTTF is the average time between two successive failures, observed over a large number of failures. To measure MTTF, it is recorded the failure data for n failures. Let the failures occur at the time instants t_1, t_2, \dots, t_n . Then, MTTF can be calculated as calculated as:

$$\sum_{i=1}^n \frac{t_{i+1}-t_i}{(n-1)}$$

3. Mean Time To Repair (MTTR)

Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.

4. Mean Time Between Failure (MTBF)

MTTF and MTTR can be combined to get the MTBF metric: $MTBF = MTTF + MTTR$. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, time measurements are real time and not the execution time as in MTTF.

5. Probability of Failure on Demand (POFOD)

POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure.

6. Availability

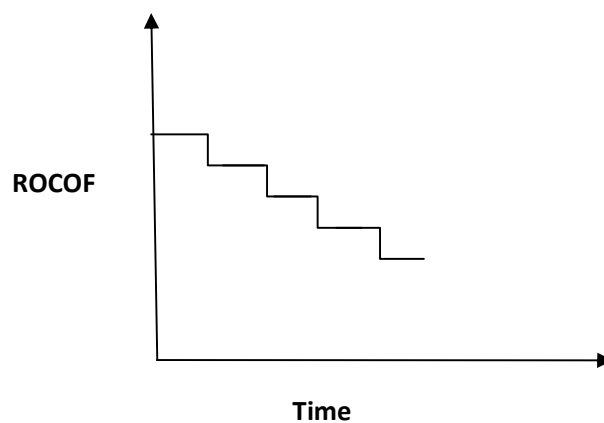
It considers the time interval up to which the product remains unaffected. Availability of a system is a measure of how likely shall the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs.

Reliability Growth Modelling:

A reliability growth model is a mathematical model of how software reliability improves as errors are detected and repaired. A reliability growth model can be used to predict when (or if at all) a particular level of reliability is likely to be attained.

Jelinski and Moranda Model:

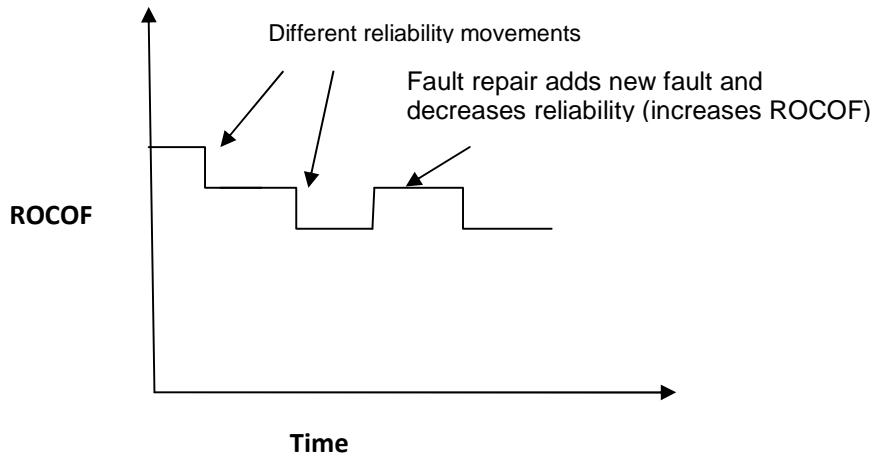
- The simplest reliability growth model is a step function model where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired.
- This simple model of reliability which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic since it is already known that correction of different types of errors contribute differently to reliability growth.



[Step function model of reliability growth]

Littlewood and Verall's Model:

- This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors.
- It also models the fact that as errors are repaired, the average improvement in reliability per repair decreases



[Random-step function model of reliability growth]

Fault Tolerance

- Fault tolerance is defined as how to provide reliability, by redundancy, service complying with the specification in spite of faults having occurred or occurring.
- Other important concepts according to Laprie is that fault tolerance is accomplished using redundancy. This argument is good for errors which are not caused by design faults, however, replicating a design fault in multiple places will not aide in complying with a specification.

Software Fault Tolerance:

- Software fault tolerance is the ability for software to detect and recover from a fault that is happening or has already happened in either the software or hardware in the system in which the software is running in order to provide service in accordance with the specification.
- Software fault tolerance is a necessary component in order to construct the next generation of highly available and reliable computing systems from embedded systems to data warehouse systems.

Tools, Techniques, and Metrics

Metrics:

- ✓ Metrics in the area of software fault tolerance, (or software faults,) are generally pretty poor.

- ✓ The data sets that have been analyzed in the past are surely not indicative of today's large and complex software systems.

Tools:

- ✓ Software fault tolerance has an extreme lack of tools in order to aid the programmer in making reliable system.
- ✓ This lack of adequate tools is not very different from the general lack of functional tools in software development that go beyond an editor and a compiler.

Techniques:

Recovery Blocks:

The recovery block method is a simple method developed by Randell from what was observed as somewhat current practice at the time. The recovery block operates with an adjudicator which confirms the results of various implementations of the same algorithm.

N-version Software:

The N-version software concept attempts to parallel the traditional hardware fault tolerance concept of N-way redundant hardware. In an N-version software system, each module is made with up to N different implementations.

Self-Checking Software:

Self checking software is not a rigorously described method in the literature, but rather a more ad hoc method used in some important systems. Self-checking software has been implemented in some extremely reliable and safety-critical systems already deployed in our society, including the Lucent ESS-5 phone switch and the Airbus A-340 airplane

Hardware Fault Tolerance:

- Current software fault tolerance is based on traditional hardware fault tolerance, (for better or worse.) Both hardware and software fault tolerance are beginning to face the new class of problems of dealing with design faults.
- Hardware designers will soon face how to create a microprocessor that effectively uses one billion transistors; as part of that daunting task, making the microprocessor correct becomes more challenging. In the future, hardware and

software may cooperate more in achieving fault tolerance for the system as a whole.

Software Project Planning, Monitoring, and Control

Project planning:

Once a project is found to be feasible, software project managers undertake project planning. Project planning is undertaken and completed even before any development activity starts. Project planning consists of the following essential activities:

- Estimating the following attributes of the project:
 - Project size:** What will be size of the project and problem complexity in terms of the effort and time required to develop the product?
 - Cost:** How much is it going to cost to develop the project?
 - Duration:** How long is it going to take to complete development?
 - Effort:** How much effort would be required?
- Scheduling manpower and other resources
- Staff organization and staffing plans
- Risk identification, analysis, and abatement planning
- Miscellaneous plans such as quality assurance plan, configuration management plan, etc

Project Monitoring and Control:

The project monitoring and control activities are undertaken once the development activities start with the aim of ensuring that the development proceeds as per plan and changing the plan whenever required to cope up with the situation.

Computer Aided Software Engineering(CASE)

- CASE is a tool that helps a software engineer to maintain the level of the software.
- This CASE tool is an integrated part of workshop of software engineering, which is called integrated project support environment (IPSE).
- CASE provides automated support to following software engineering processes.

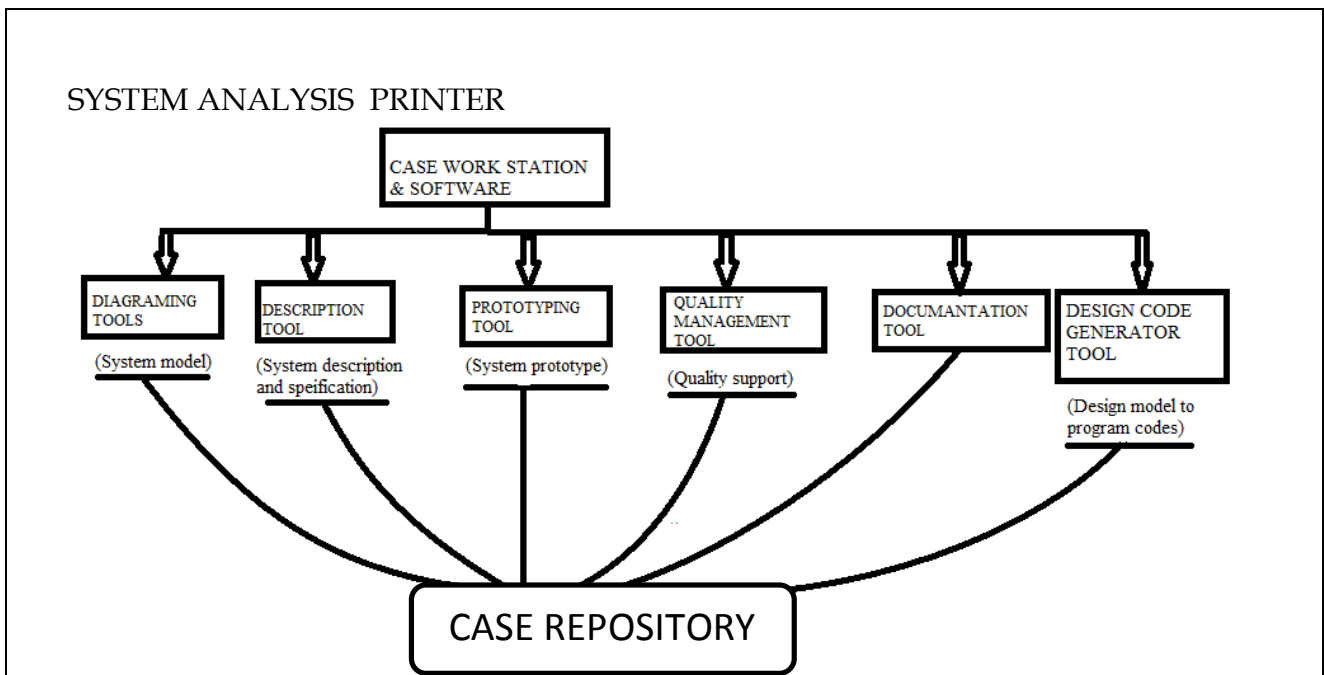
Examples:

- Translation of user need into software requirements.

- Translation of software requirements into design specifications.
- Implementation of design into code.
- Testing of the code written by the programmers.
- Documentation of the project.

It also provides automation for various process activities such as:

- Development of graphical system model as a part of the requirement specifications for software engineering.
- Understanding a design using data dictionary.
- Generation of user interface (GUI) from graphical interface designed by users.
- Automated translation of program from old to most recent language.
i.e. COBOL to Java



- **Case repository-** case repository usually stores into that they may be shared by multiple projects and participants..

Level of Case:

There are three different level of the CASE technology which are as follows.

- **Production process support technology:**

This includes the support for process activity such as specification, design, implementation, testing, etc.

- **Process management and technology:**

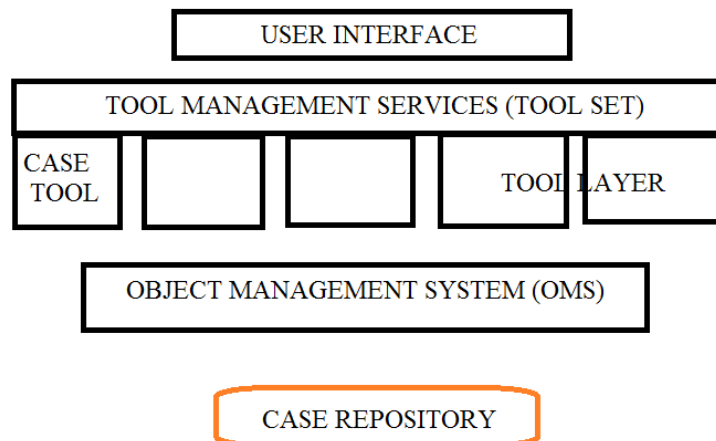
This includes tools to support process modelling and management.

- **Meta case technology:**

These tools are generator to create production process management support tools.

Case Environment Architecture:

Diagram:



Important component of a modern CASE environment are

1. **User interface**
2. **Tool set**
3. **Object management system**
4. **CASE repository**

1. **User interface:**

User interface provides a common framework for accessing different tools by simplifying and reducing the time of learning of the tools.

2. **Tool set:**

It control the behaviour of the tool within the environment by performing multitask synchronisation and composition. Co-ordinating the flow of information repository to OMS into the tools.

It provides security function, collect feedbacks on tool usage.

3. Object management system:

It performs the mapping of logical entities such as specification data, design data, project planning data, etc into the storage management system. i.e. repository.

It also has object management language (OML) module to provide support for version control, change control, status control and security control.

4. CASE repository:

This is the case data base containing **access control function** to enable the OMS to interact with the data base.

*** Provided by case environment externally supported.**

Objectives of Case:

The various objectives are

- **Improvement in productivity:** by reducing the time of design and development.
- **Improvement in information system quality:** by providing automation.
- **Improvement of effectiveness:** performing the right task with minimal effort.

Disadvantages Of Case:

1. High cost of purchase and use.
2. High cost of training the user.
3. It s not easy to share information between tools.
4. Software people often see CASE as a theft to their job security.

CHARACTERISTICS OF THE CASE TOOLS:

1. Graphical interface to draw diagram, models, etc.
2. An information repository, data dictionary for official information management, selection, usage, application, storage.
3. Common GUI for integrated multiple tools.
4. Automatic code generator.
5. Automatic testing tools.

Sl. No.	Application	CASE tools	Purpose of the tools
1	Planning	Excel, Spreadsheet, MS-project, PERT/CPM, Network, Estimation tools	Functional application planning in scheduling control
2	Editing	Diagram editors, Text editors, word processors	Speed and effective
3	Testing	Test data generators, FILE comparators	Speed and efficiency
4	Prototyping	High level modelling language,UI- generators	Confirmation and certification of RDD and SRS
5	Documentation	Report generator, Publishing, Imaging, ppt presentation	Faster structural documentation with quality of presentation.
6	Programming language processing and integration	Program generator Code generator Compiler Interpreter Interface connectivity	Programming of high quality with no errors system integration
7	Templates	-----	Guided systemic development
8	Re engineering tools	Cross reference system Restructuring system	Reverse engineering to fixed structure design and destination information
9	Program analysis tools	Cross interface generator Static analyzers Dynamic analyzers	Analyses risk functions and features.

Component Model of Software Development

Components:

- Components provide a service without regard to where the component is executing or its programming language.
- A component is an independent executable entity that can be made up of one or more executable objects.
- The component interface is published and all interactions are through the published interface.

Definitions:

A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.

Component model:

- A component model is a definition of standards for component implementation, documentation and deployment.
- Examples of component models
 - ✓ EJB model (Enterprise Java Beans)
 - ✓ COM+ model (.NET model)
 - ✓ Corba Component Model
- The component model specifies how interfaces should be defined and the elements that should be included in an interface definition.

Component-based development:

It is the enabler of the client-server technology. Component-based development is radically different from traditional software development. In component-based development, a developer essentially integrates pre-built components purchased off-the-shelf. This is akin to the way hardware developers integrate ICs on a Printed Circuit Board (PCB). Components might reside on different computers which act as servers and clients.

Definition of Component Based Software Engineering:

Component-based software engineering (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable

Component-based software engineering (CBSE) is a branch of software engineering which emphasizes the separation of concerns in respect of the wide-ranging functionality available throughout a given software system. This practice aims to bring about an equally wide-ranging degree of benefits in both the short-term and the long-term for the software itself and for organizations that sponsor such software.

Objectives of Component Based Software Engineering:

The main objectives of component based software engineering are given below.

a) Reduction of cost and time for building large and complicated systems: main objective of Component based approach is to build complicated software systems using off the shelf component so that the time to build the software diminish drastically.

b) Improving the quality of the software: The quality of the software can be improved by improving the quality of the component. Sometimes quality of the assembled systems may not be directly related to quality of the component in sense that improving the quality of the component does not necessarily imply the improvement of the systems.

c) Detection of defect within the systems: Component approach helps the system to find its defect readily by testing the components. But the source of defects is difficult to find in case of component development approach.

few basic definition regarding software components:-

Component—It is a nontrivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture.

Run-time software component—It is a dynamic bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered in run time.

Software component—It is a unit of composition with contractually specified and explicit context dependencies only.

Business component—It is the software implementation of an “autonomous” business concept or business process.

In addition to these descriptions, software components can also be characterized based on their use in the CBSE process

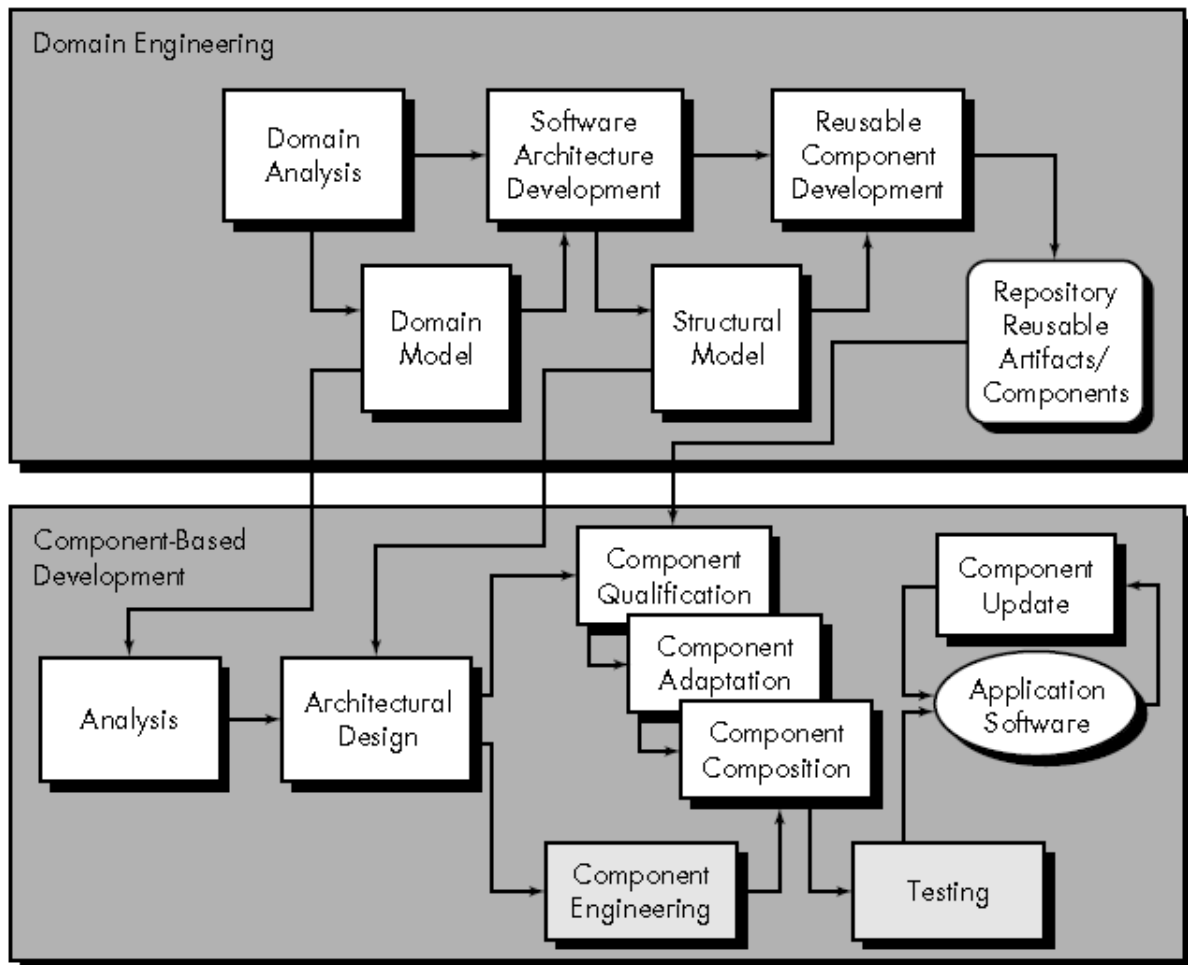
Qualified components—assessed by software engineers to ensure that not only functionality, but performance, reliability, usability, and other quality factors conform to the requirements of the system or product to be built.

Adapted components—adapted to modify (also called mask or wrap) unwanted or undesirable characteristics.

Assembled components—integrated into an architectural style and interconnected with an appropriate infrastructure that allows the components to be coordinated and managed effectively.

Updated components—replacing existing software as new versions of components become available.

Component Based Software Engineering Processes:



Disadvantages of Component Based Development:

The main disadvantages of CBD are given below:

1. it is very difficult to build the environment that is fitted to component
2. Concept of reuse: Standards are needed regarding middleware in which the component is supposed to work. (Middleware: A communication layer which enables components to interact with higher level component in a network). But among the three standards available, only CORBA is language independent. Hence the comparison is not viable among the standards.

Advantages of Component Based Development:

1. Flexibility: runtime components can work independently if properly designed and they are less dependent on the environment
2. Reuse: Once developed, it can be used everywhere regarding the programming language and OS. But domain engineering should be kept in mind

3. Easy to maintain because ideally the functionality is implemented once
4. Development cost is much lower
5. Lesser time required to build the software

Software Reuse

Software reuse makes repeated use of component from the previously developed software in order to reduce the software cost by providing high quality and reliability.

The components which can be re-used are:-

- a. Requirement specification
- b. Design
- c. Coding
- d. Test cases
- e. Knowledge and Experience

The entire component can be re effectively re use by providing with proper documentation.

Reason for limited reuse:

1. It is very difficult to credit the component they can be reused across different application.
2. Developers are not consulting about providing necessary attention toward the component which can be reuse from the previously developed system.

Basic issues in program reuse:

The issues are:-

- a. Component Creation.
- b. Component Index and Testing.
- c. Component Search.
- d. Component Understanding.
- e. Component Adoption
- f. Repository Maintenance

Reuse Approach:

The reuse approach is critically end toward identifying reusable component by domain analysis i.e. analyzing a problem domain.

Domain Analysis:

- It provides a reuse domain which is technically related set of application area.
For example: Account Software Domain, Banking Software Domain, Business Software Domain, Telecommunication Software Domain etc.
- A part of problem domain is considered for reuse. If it is shared by understanding some community, characterized by concept/ techniques and coherence in the terminologies.
- During domain analysis a community of software developer analysis application domain to identify the reusable component. The detailed list of reusable component for a particular component is called domain engineering.

As a domain developer we can classify it in different stages:-

Stage-1: No reusable components are available.

No clear and consistency methodology is available.

Stage-2: Knowledge re-use is made in the new development method.

Stage-3: Domain is made suitable for reuse by providing standard solution to the problem. We can make knowledge in component reuse.

Stage-4: Complete domain development can be done by reuse and the program are written using application generator.

Component Classification:

Component can be classified by their level of ambiguity for effective storage structure. For this a classification is used called as Prito-Diaz's Classification scheme.

Each component is best described using number of characteristics or facts:

Ex: Object can be classified using following.

- Action they embody.
- Object they Manipulate
- Data Structure Used
- System they are part of end so forth.

This classification scheme makes us search for required component which is best suited for a system this is very difficult.

Component Search:

Component searching for reuse can be easily written through a web based domain repository contains thousands of items. However inside the repository there must be classification between component / design / models / requirement / knowledge to make searching easier.

Repository Maintenance:

It involves adding of new reused artifacts, deleting those items which are no use, modifying the search criteria to improve effectiveness of search.

Reuse without Modification:

Reuse without modification is reused once standard solution for program models are available.

Reuse of Organization level:

Three cases involved in the reorder.

1. Assessment of a product for reuse.
2. Replacement of a product for improvement for reusability.
3. Handling portability problem of reuse problems.

Current Stage of reuse Methodology:

Factor effective reuses of software product are mostly non-technically these are as follows:

- Limited commitment from the top management.
- No ad acute documentation for support reuse.
- No ad acute record reward to those who reuse.
- Providing information and access to reuse Component.
-

