Software Design

4.1 Definition: Software design is a problem-solving activity which bridge the gap between specifications and coding. It is more creative than analysis as it produces a solution to a problem given in SRS document. Good design is always system dependent and what is good for one system may be bad for another.

A framework of design is given below. It starts with initial requirements and ends up with the final design. Here data is gathered on user requirements and analysed accordingly. A high level design is prepared after answering questions of requirements. Moreover, design is validated against requirements on regular basis. Design is refined in every cycle and finally it is documented to produce software design document.

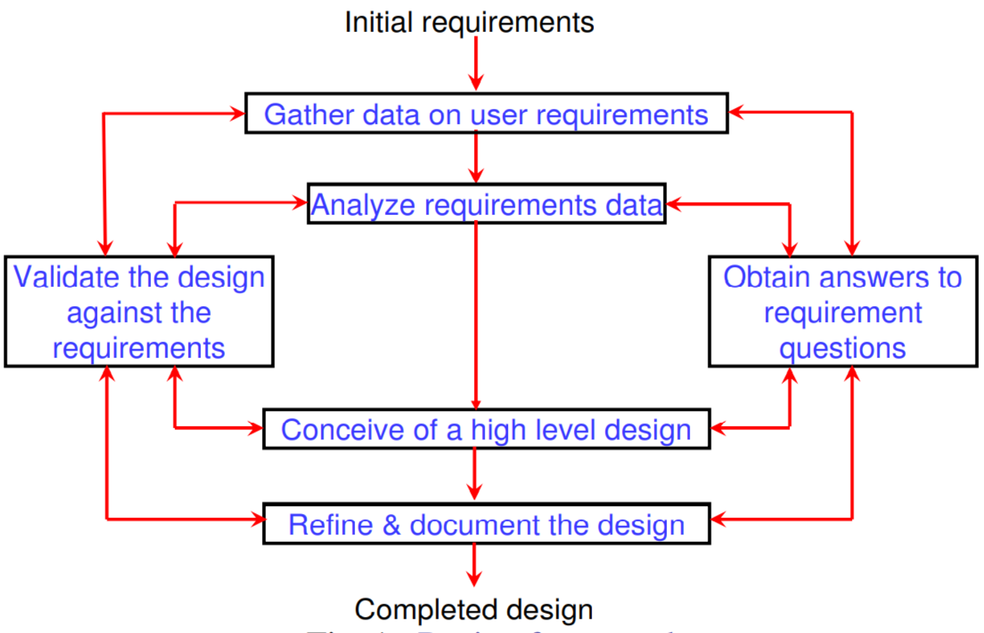

Initial requirements
Gather data on user requirements
Analyze requirements data
Validate the design against the requirements
Obtain answers to requirement questions
Conceive of a high level design
Refine & document the design
Completed design

Fig. 1 : Design framework

4.1.1 Types of design

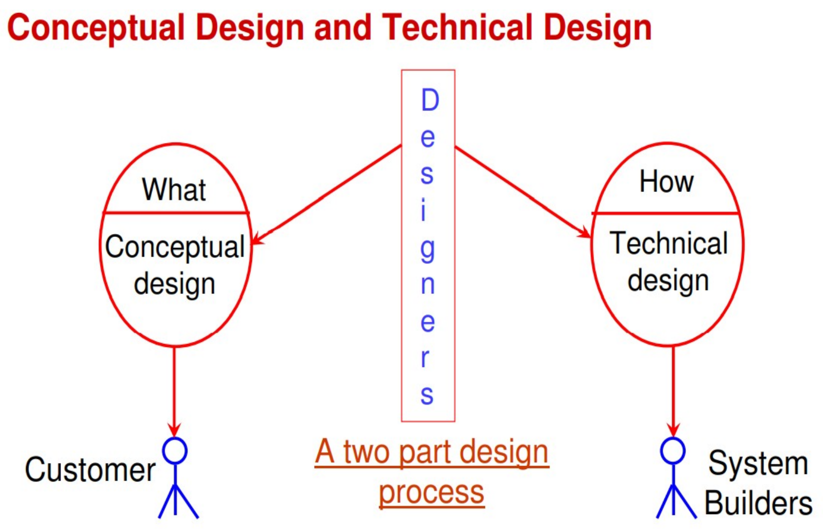


Fig. 2 : A two part design process

Design is really a two-part, iterative process. First, we produce conceptual design that tells the customer what the system will do in a simple language avoiding technical terms and implementation details. It contains information about

- source of input data
- data transformations in the system
- appearance of system to users and user choices
- timing of various events
- screen and report format

Once the customer approves the conceptual design, we translate it into technical design that allows system builders to understand the actual hardware and software needed to solve customer's problem. It describes

- Hardware configuration
- Software needs
- Communication interfaces
- I/O of the system
- Software architecture
- Network architecture
- Any other thing that translates the requirements in to a solution to the customer's problem.

4.1.2 Objectives and importance of design

Design bridges the gap between specifications and coding. The design needs to be

- Correct & complete
- Understandable
- At the right level
- Maintainable

The design process involves adding details as the design is developed with constant backtracking to correct earlier, less formal, designs as shown below.

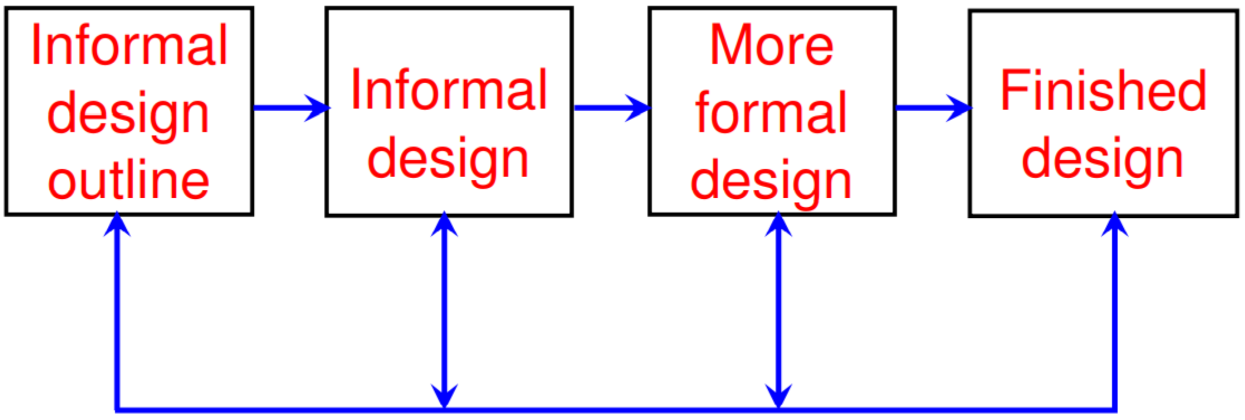


Fig. 3 : The transformation of an informal design to a detailed design.

A good design is the key to successful product. It must

a) implement all explicit and implicit requirements

b) be a readable guide for programmers and testers

c) provide a complete picture of software covering data, functional and behavioural domains.

Good design has attributes: durability, utility and charm. Without a good design:

(a) system will fail even with minor changes.

(b) system's maintenance will be difficult.

(c) quality assessment is not possible until late.

4.2 Modularity

There are many definitions of the term module. It can be
Procedures & functions of PASCAL & C
C++ / Java classes and Java packages
Work assignment for an individual programmer
A modular system consist of well defined manageable units with well defined interfaces among the units. Desirable properties of a modular system include

- each module is a well defined subsystem
- each module has well defined purpose
- modules can be separately compiled and stored in a library.
- modules can use other modules
- modules should be easier to use than to build
- modules should be simpler from outside than from the inside.

Modularity is the single attribute of software that allows a program to be intellectually manageable. It enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of software product. Under modularity and over modularity should be avoided. The relationship between cost and number of modules is shown below. As shown, the integration effort grows with number of modules.

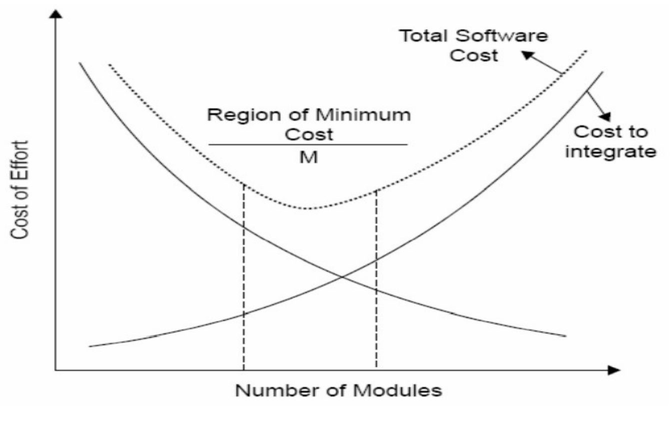


Fig. 4 : Modularity and software cost

4.2.1 Module Coupling

Coupling is the measure of the degree of interdependence between modules.

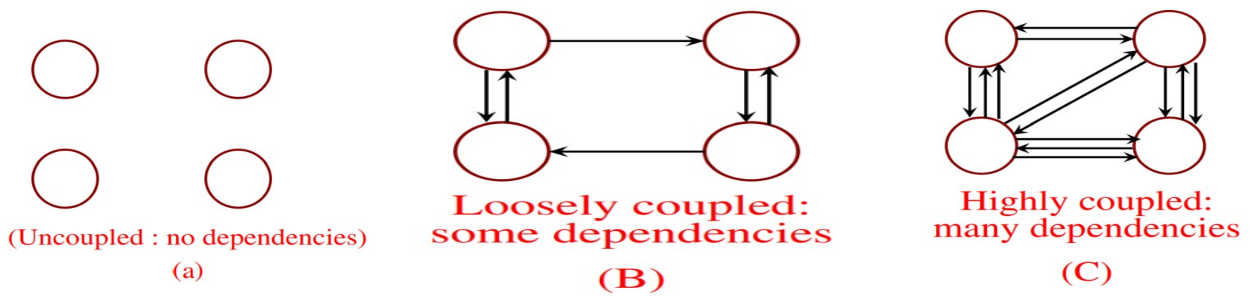

(Uncoupled : no dependencies)
(a)
Loosely coupled: some dependencies
(B)
Highly coupled: many dependencies
(C)

Fig. 5: Module coupling

A good design will have low coupling. This can be achieved in the following ways:

- Controlling the number of parameters passed amongst modules.
- Avoid passing undesired data to calling module.
- Maintain parent / child relationship between calling & called modules.
- Pass data, not the control information.

Consider the example of editing a student record in a 'student information system'.

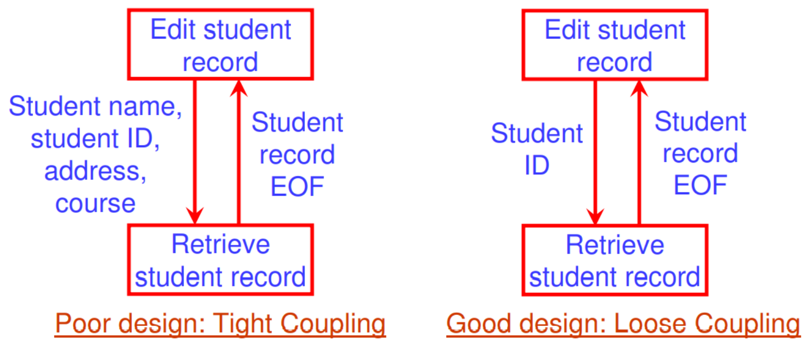


Fig. 6 : Example of coupling

Types of coupling

Data coupling	Best
Stamp coupling	↑
Control coupling	
External coupling	
Common coupling	
Content coupling	Worst

Fig. 7 : The types of module coupling

Given two procedures A & B, we can identify number of ways in which they can be coupled.

Data coupling: The dependency between module A and B is said to be data coupled if their dependency is based on the fact they communicate by only passing of data. Other than communicating through data, the two modules are independent.

Stamp coupling: Stamp coupling occurs between module A and B when complete data structure is passed from one module to another.

Control coupling: Module A and B are said to be control coupled if they communicate by passing of control information. This is usually accomplished by means of flags that are set by one module and reacted upon by the dependent module.

External coupling: A module has a dependency to other module, external to software being developed or to a particular type of hardware.

Common coupling: With common coupling, module A and module B have shared data. Global data areas are commonly found in programming languages. Making a change to the common data means tracing back to all the modules which access that data to evaluate the effect of changes.

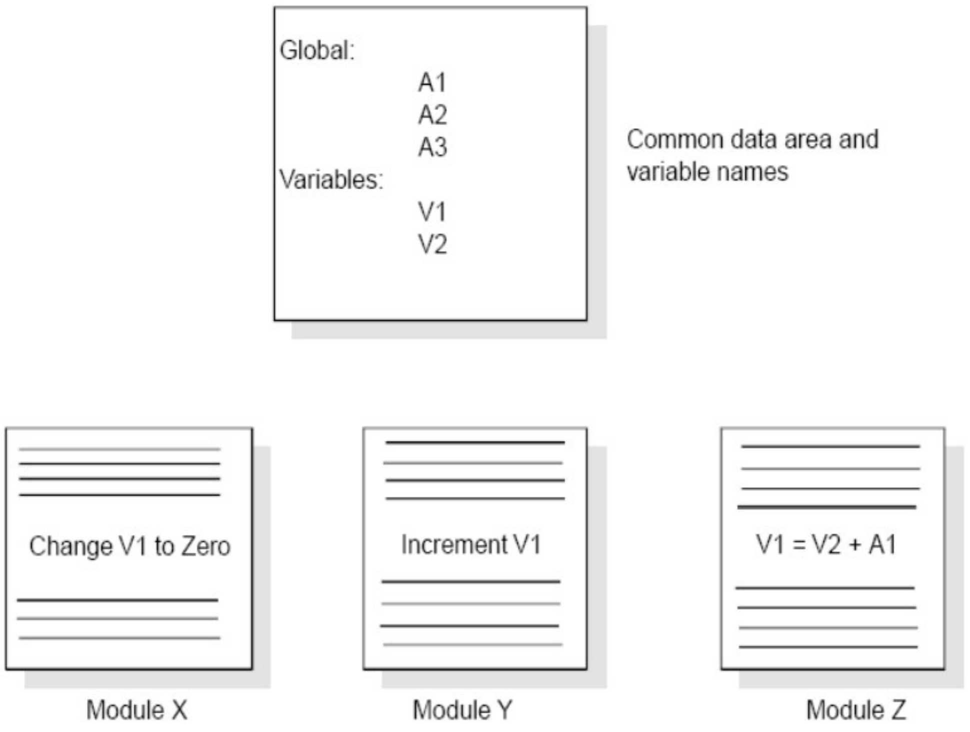


Fig. 8 : Example of common coupling

Content coupling: Content coupling occurs when module A changes data of module B or when control is passed from one module to the middle of another. In Fig. 9, module B branches into D, even though D is supposed to be under the control of C.

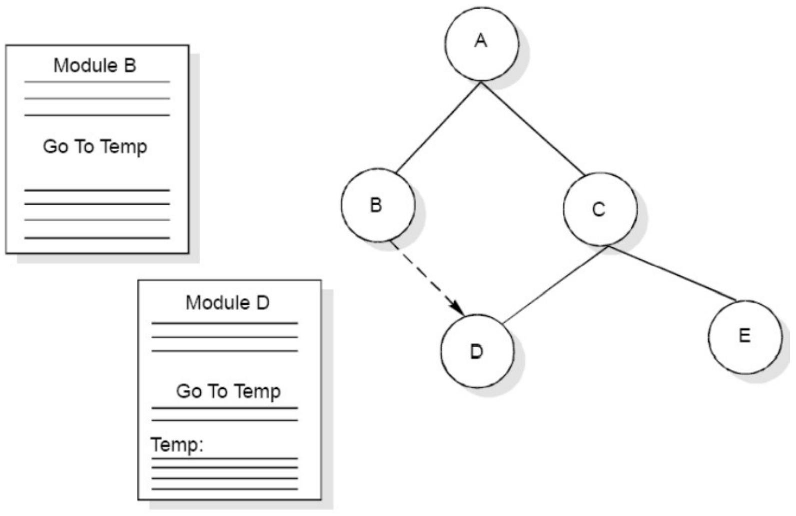


Fig. 9 : Example of content coupling

4.2.2 Module Cohesion

Cohesion is a measure that defines the degree of interdependence between the elements of a module. The greater the cohesion, the better is the program design.

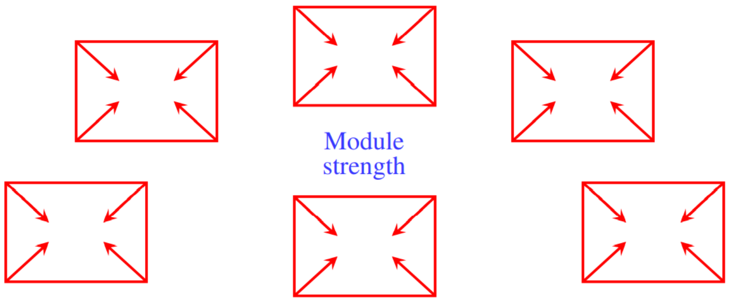


Fig. 10 : Cohesion=Strength of relations within modules

Types of Cohesion

There are seven types or levels of cohesion and are shown in Fig. 11. Given a procedure that carries out operations X and Y, we can describe various forms of cohesion between X and Y.

Functional Cohesion	Best (high)
Sequential Cohesion	
Communicational Cohesion	
Procedural Cohesion	
Temporal Cohesion	
Logical Cohesion	
Coincidental Cohesion	Worst (low)

Fig. 11 : Types of module cohesion

Functional Cohesion

Elements of a functionally cohesive module contribute to a single well defined function.

Sequential Cohesion

When elements of a module are grouped because the output of one element serves as input to another and so on. It is called sequential cohesion.

Communicational Cohesion

When elements of a module are grouped together, which are executed sequentially and work on same data, it is called communicational cohesion.

Procedural Cohesion

Procedural Cohesion occurs in modules whose instructions although accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed.

Temporal Cohesion

Module exhibits temporal cohesion when it contains tasks that are related by the fact that all tasks must be executed in the same time-span.

Logical Cohesion

When logically categorized elements are put together into a module, it is called logical cohesion.

Coincidental Cohesion

Coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another.

Relationship between Cohesion & Coupling

If the software is not properly modularized, a host of seemingly trivial enhancement or changes will result into death of the project. Therefore, a software engineer must design the modules with goal of high cohesion and low coupling.

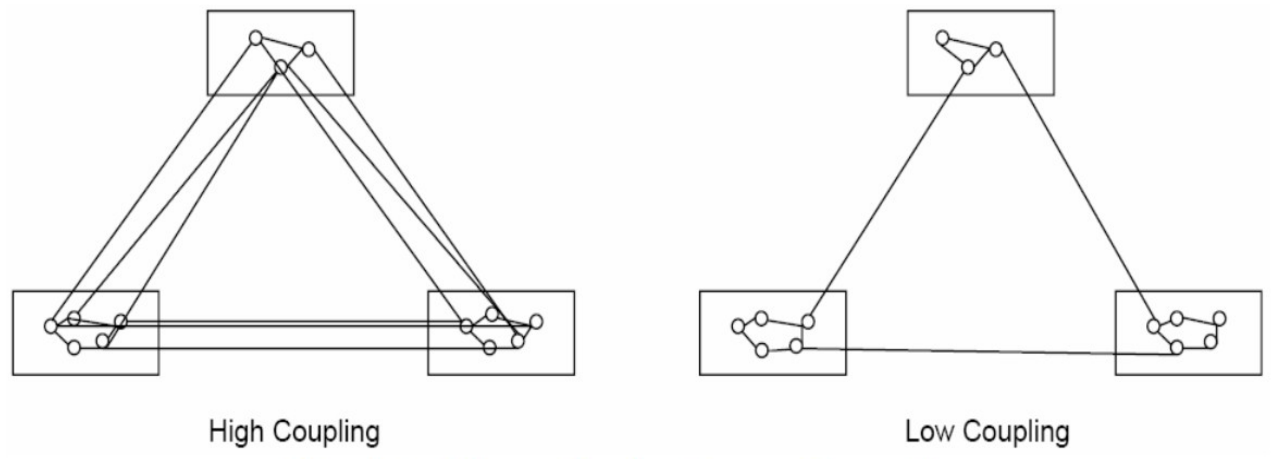


Fig. 12 : View of cohesion and coupling

4.3 Strategy Of Design

A good system design is to organise the program modules in such a way that are easy to develop and change. Structured design techniques help developers to deal with the size and complexity of programs. Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program. It is important for two reasons:

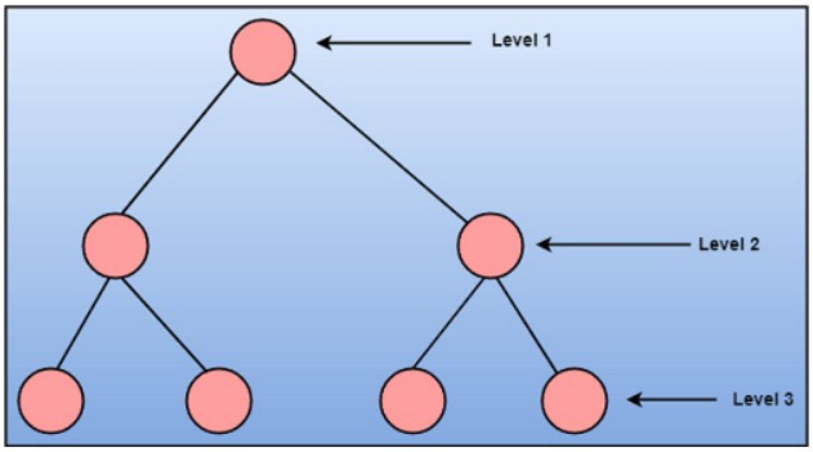
- If any pre-existing code needs to be understood, organised and pieced together.
- It is common for the project team to have to write some code and produce original programs that support the application logic of the system.

Major strategies for performing system design are top-down, bottom-up and hybrid design.

4.3.1 Top-down design

A top down design approach starts by identifying the major modules of the system, decomposing them into their lower level modules and iterating until the desired level of detail is achieved. This is stepwise refinement; starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly.

Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.



4.3.2 Bottom-Up Design

The bottom up design model starts with most specific and basic modules. It proceeds with composing higher level of modules by using basic or lower level modules. It keeps creating higher level modules until the desired system is not evolved as one single module. The set of these modules form a hierarchy as shown below.

Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.

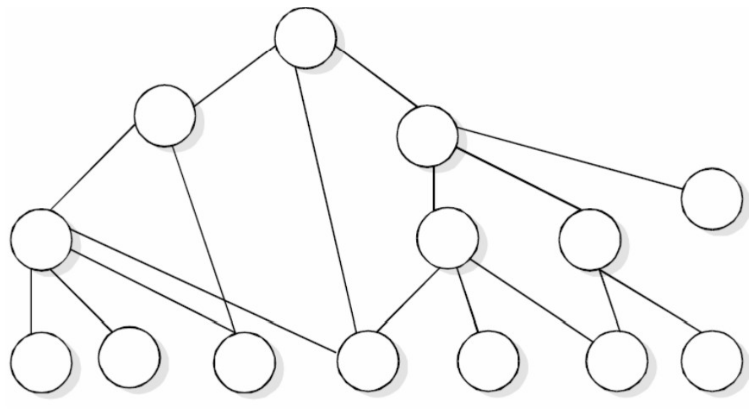


Fig. 13 : Bottom-up tree structure

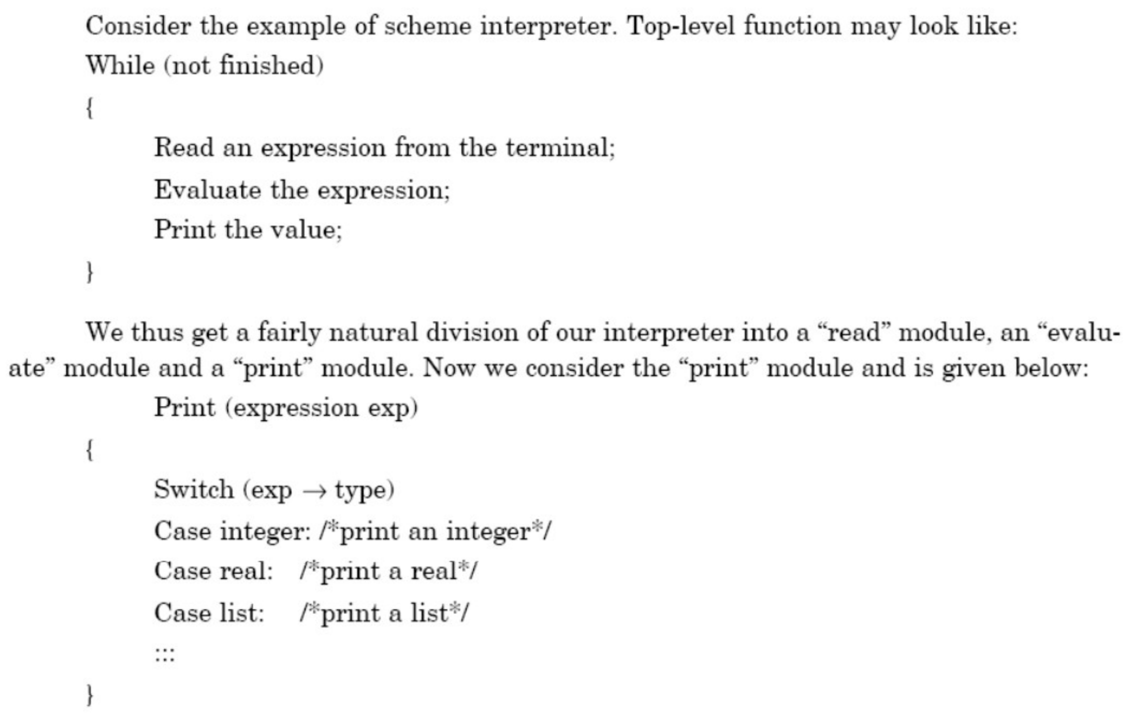
4.3.3 Hybrid Design

Hybrid design is a combination of both the top–down and bottom–up design strategies. In this we can reuse the modules. For a bottom-up approach to be successful, we must have a good idea about the top modules. For top-down approach to be effective, some bottom-up approach is essential for the following reasons:

- To permit common sub modules.
- Near the bottom of the hierarchy, where the intuition is simpler, and the need for bottom-up testing is greater, because there are more number of modules at low levels than high levels.
- In the use of pre-written library modules, in particular, reuse of modules.

4.4 Function Oriented Design

Function Oriented design is a method to software design where the model is decomposed into a set of interacting units or modules where each unit or module has a clearly defined function. Thus, the system is designed from a functional viewpoint.



Consider the example of scheme interpreter. Top-level function may look like:

```
While (not finished)
{
        Read an expression from the terminal;
        Evaluate the expression;
        Print the value;
}
```

We thus get a fairly natural division of our interpreter into a “read” module, an “evaluate” module and a “print” module. Now we consider the “print” module and is given below:

```
        Print (expression exp)
{
        Switch (exp → type)
        Case integer: /*print an integer*/
        Case real:    /*print a real*/
        Case list:    /*print a list*/
        :::
}
```

We continue the refinement of each module until we reach the statement level of our programming language. At that point, we can describe the structure of our program as a tree of refinement as in design top-down structure as shown in fig. 14.

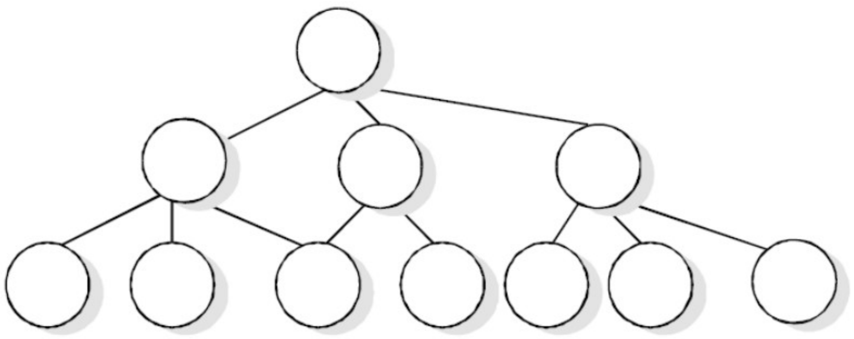
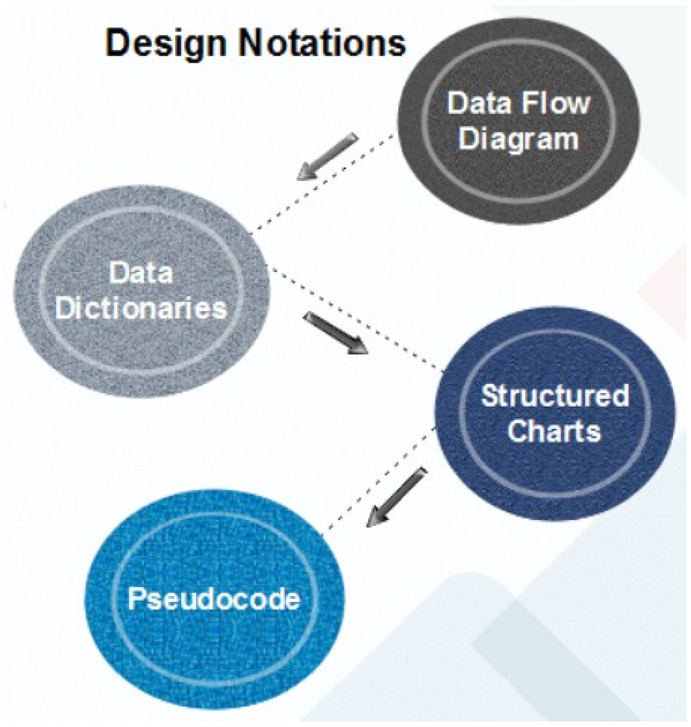


Fig. 14 : Top-down structure

4.4.1 Design Notations

Design Notations are primarily meant to be used during the process of design and are used to represent design or design decisions. For a function-oriented design, the design can be represented graphically or mathematically by the following:

- Data flow diagrams
- Data Dictionaries
- Structure Charts
- Pseudocode



Structure Chart

The structure chart is one of the most commonly used method for system design. Structure Charts are design tools for describing a system according to its functions. It partition a system into black boxes. A black box means that functionality is known to the user without the knowledge of internal design. Inputs are given to black box and appropriate outputs are generated by the black box. This concept reduces the complexity because details are hidden from those who have no need to know.

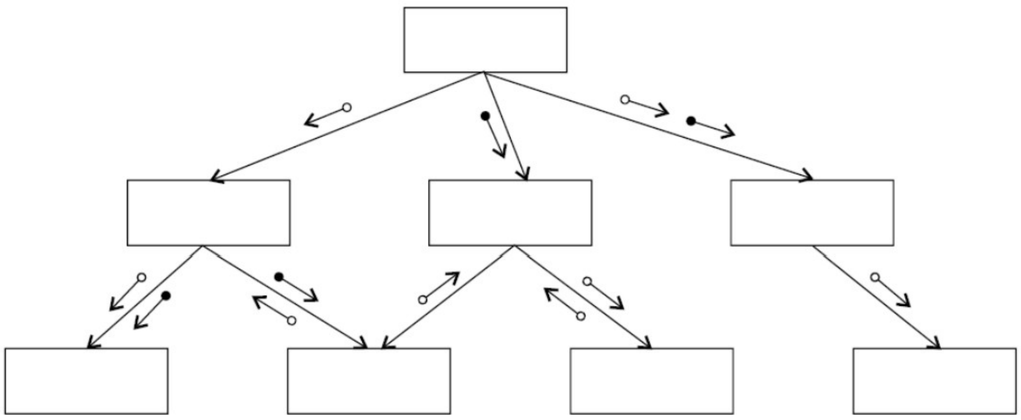


Fig. 16 : Hierarchical format of a structure chart

Structured Chart is a graphical representation which shows:

- System partitions into modules
- Hierarchy of component modules
- The relation between processing modules
- Interaction between modules
- Information passed between modules

Structure Chart Notations

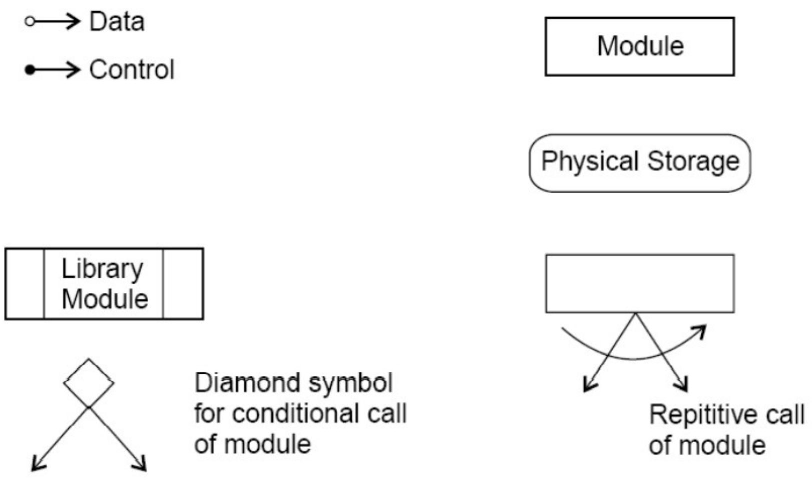

Data
Control
Module
Physical Storage
Library Module
Diamond symbol for conditional call of module
Repititive call of module

Fig. 17 : Structure chart notations

A structure chart for "update file" is given in fig. 18.

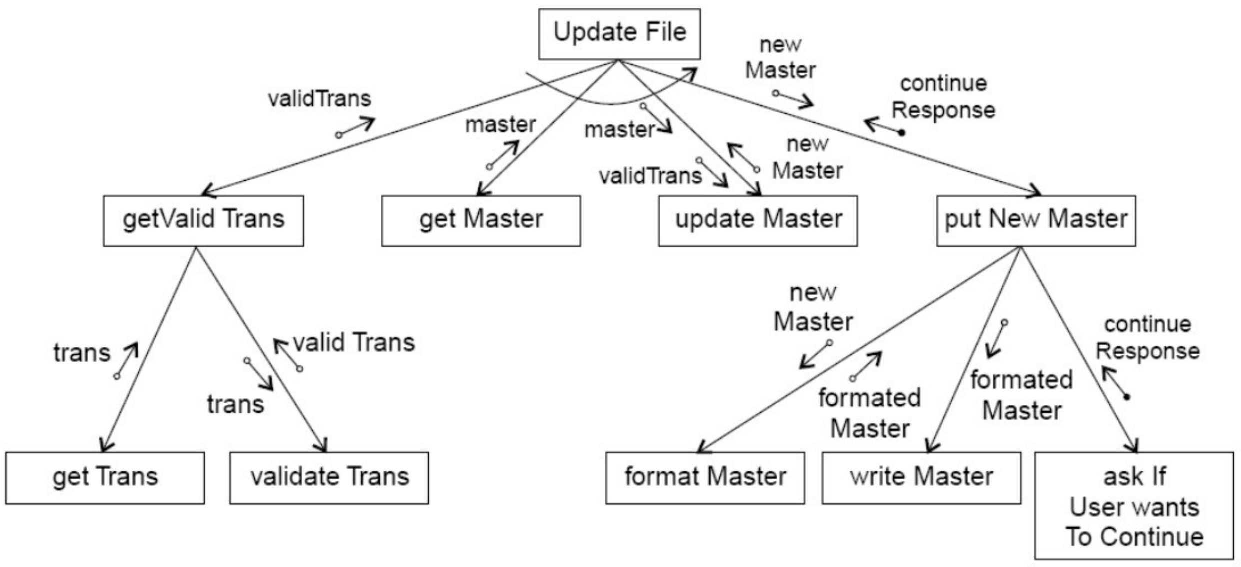


Fig. 18 : Update file

This type of structure chart is often called as transform - centered structures. Transform – centered structure receive an input which is transformed by a sequence of operations, with each operation being carried out by one module.

A transaction centered structure describes a system that processes a number of different types of transactions. It is illustrated in Fig.19

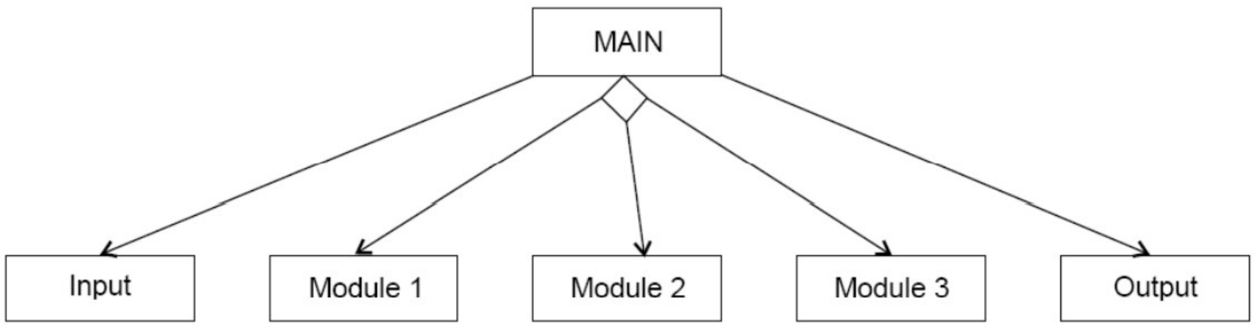


Fig. 19 : Transaction-centered structure

In the above figure the MAIN module controls the system operation its functions is to:

- invoke the INPUT module to read a transaction;
- determine the kind of transaction and select one of a number of transaction modules to process that transaction, and
- output the results of the processing by calling OUTPUT module.

Pseudocode

Pseudocode notation can be used in both the preliminary and detailed design phases. Using pseudocode, the designer describes system characteristics using short, concise, English language phrases that are structured by key words such as If-Then-Else, While-Do, and End.

Pseudocode can replace flowcharts and reduce the amount of external documentation required to describe a system.

4.5 IEEE Recommended practice for software design descriptions (IEEE STD 1016-1998)

4.5.1 Scope

An SDD is a representation of a software system that is used as a medium for communicating software design information.

4.5.2 References

1) IEEE std 830-1998, IEEE recommended practice for software requirements specifications.
2) IEEE std 610.12-1990, IEEE glossary of software engineering terminology.

4.5.3 Definitions

Few important definitions are given below:

Design entity: An element (Component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.

Design View: A subset of design entity attribute information that is specifically suited to the needs of a software project activity.

Entity attributes: A named property or characteristics of a design entity. It provides a statement of fact about the entity.

Software design description (SDD): A representation of a software system created to facilitate analysis, planning, implementation and decision making.

4.5.4 Purpose of an SDD

The SDD shows how the software system will be structured to satisfy the requirements identified in the SRS. It is basically the translation of requirements into a description of the software structure, software components, interfaces, and data necessary for the implementation phase. Hence, SDD becomes the blue print for the implementation activity.

4.5.5 Design Description Information Content

i) Introduction: SDD is a model of the system.

ii) Design entities

iii) Design entity attributes

a) **Identification:** The unique name of the entity.
b) **Type:** A description of type of entity such as module, process or data store.
c) **Purpose:** A description of why entity exists.
d) **Function:** A statement of what the entity does.
e) **Subordinates:** The composition of all entities composing a particular entity.
f) **Dependencies:** Description of relationship between entities.

g) **Interface:** A description of methods of interaction and the rules governing those interactions.

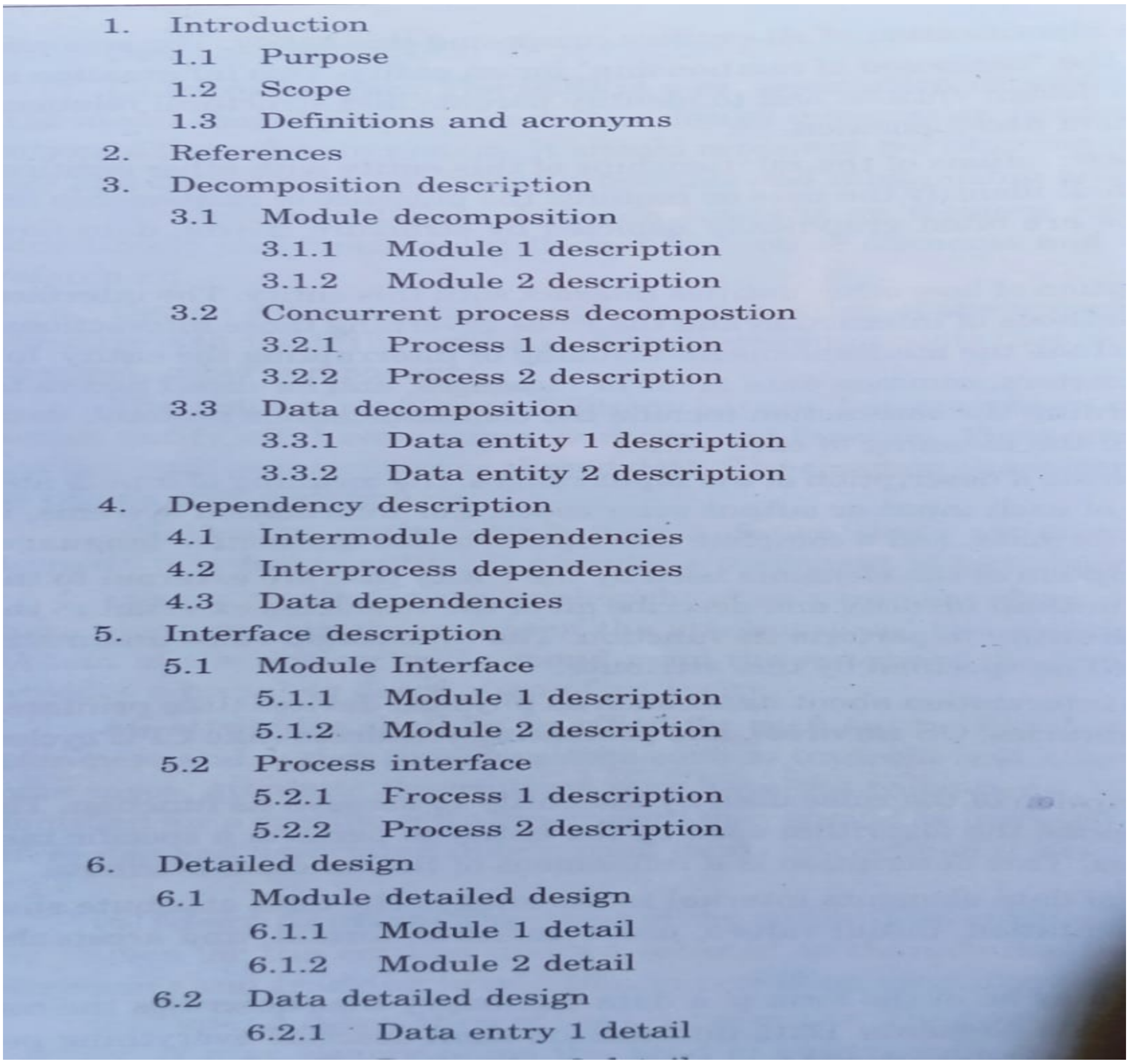
h) **Resources:** A description of external resources used by entity such as memory space, CPU cycles etc.

i) **Processing:** A description of algorithm used by the entity to perform a specific task.

j) **Data:** A description of data elements internal to the entity such as initial values, format etc.

4.5.6 Design Description Organization

The organization of SDD is given below. This is one of the possible ways to organize and format the SDD.



1. Introduction
    - 1.1 Purpose
    - 1.2 Scope
    - 1.3 Definitions and acronyms
2. References
3. Decomposition description
    - 3.1 Module decomposition
        - 3.1.1 Module 1 description
        - 3.1.2 Module 2 description
    - 3.2 Concurrent process decompostion
        - 3.2.1 Process 1 description
        - 3.2.2 Process 2 description
    - 3.3 Data decomposition
        - 3.3.1 Data entity 1 description
        - 3.3.2 Data entity 2 description
4. Dependency description
    - 4.1 Intermodule dependencies
    - 4.2 Interprocess dependencies
    - 4.3 Data dependencies
5. Interface description
    - 5.1 Module Interface
        - 5.1.1 Module 1 description
        - 5.1.2 Module 2 description
    - 5.2 Process interface
        - 5.2.1 Frocess 1 description
        - 5.2.2 Process 2 description
6. Detailed design
    - 6.1 Module detailed design
        - 6.1.1 Module 1 detail
        - 6.1.2 Module 2 detail
    - 6.2 Data detailed design
        - 6.2.1 Data entry 1 detail

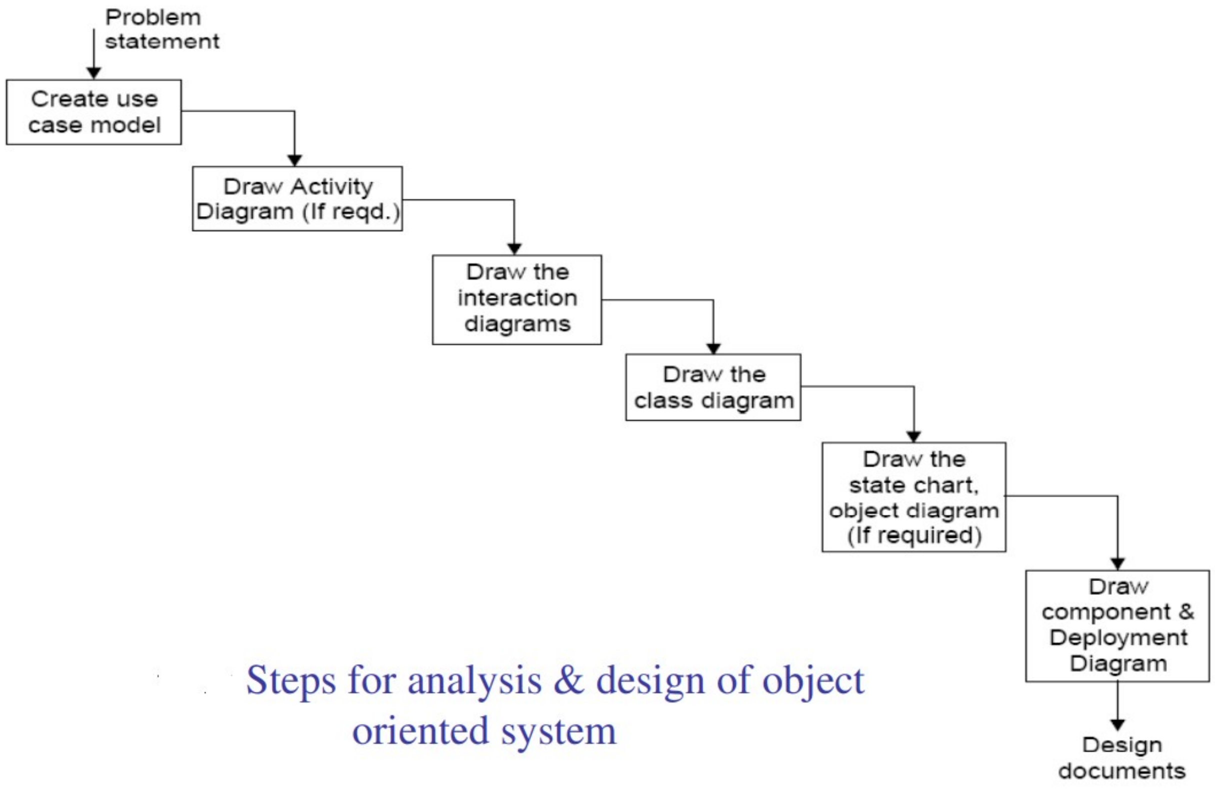
Organisation of SDD

Design View	Scope	Entity attribute	Example representation
Decomposition description	Partition of the system into design entities	Identification, type purpose, function, subordinate	Hierarchical decomposition diagram, natural language
Dependency description	Description of relationships among entities of system resources	Identification, type, purpose, dependencies, resources	Structure chart, data flow diagrams, transaction diagrams
Interface description	List of everything a designer, developer, tester needs to know to use design entities that make up the system	Identification, function, interfaces	Interface files, parameter tables
Detail description	Description of the internal design details of an entity	Identification, processing, data	Flow charts, PDL etc.

Table 2: Design views

4.6 Steps to Analyze and Design Object Oriented System

There are various steps in the analysis and design of an object oriented system and are given in fig. Below.


Problem statement
Create use case model
Draw Activity Diagram (If reqd.)
Draw the interaction diagrams
Draw the class diagram
Draw the state chart, object diagram (If required)
Draw component & Deployment Diagram
Design documents

Steps for analysis & design of object oriented system

(i) Create use case model: First step is to identify the actors interacting with the system. We should then write the use case and draw the use case diagram.

(ii) Draw activity diagram (if required): Activity Diagram illustrate the dynamic nature of a system by modeling the flow of control form activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Following figure shows the activity diagram processing an order to deliver some goods.

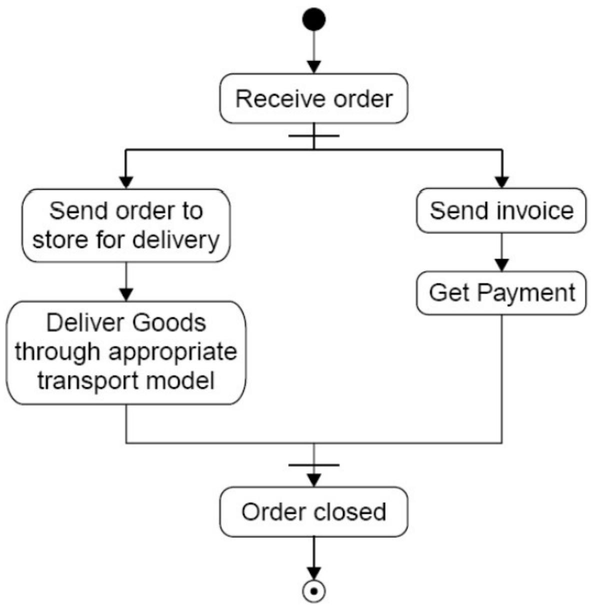

Receive order
Send order to store for delivery
Send invoice
Get Payment
Deliver Goods through appropriate transport model
Order closed

Fig. 26: Activity diagram

(iii) Draw the interaction diagram: An interaction diagram shows an interaction, consisting of a set of objects and their relationship, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system. Steps to draw interaction diagrams are as under:

a) Firstly, we should identify that the objects with respects to every use case.

b) We draw the sequence diagrams for every use case.

c) We draw the collaboration diagrams for every use case.

A sequence diagram (**Fig. 29)** is an interaction diagram that emphasizes the time ordering of messages; a collaboration diagram (**Fig. 30**) is an interaction diagram that emphasizes the structural organisation of the objects that send and receive objects.

The object used in this analysis model are entity objects; interface objects and control objects as given below. The entity object represents information held for a long time by the system. Eg; login-detail. Interface object represents information which depend on the interface to the system. Eg; login screen. Control object represents any control information. Eg; login-checker

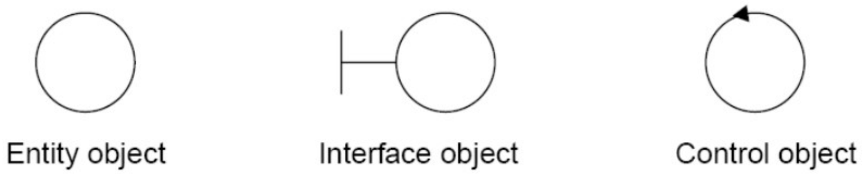
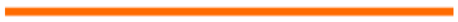


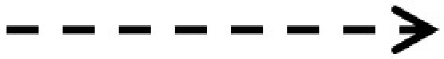
Fig. 27: Object types

(iv) Draw the class diagram: The class diagram shows the relationship amongst classes. There are four types of relationships in class diagrams.

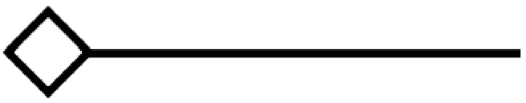
(a) **Association** are semantic connection between classes. When an association connects two classes, each class can send messages to the other in a sequence. Associations can be bi-directional or unidirectional.



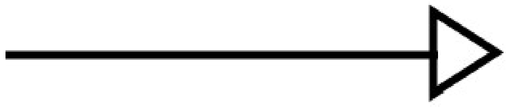
(b) **Dependencies** connect two classes. Dependencies are always unidirectional and show that one class, depends on the definitions in another class.



(c) **Aggregations** are stronger form of association. An aggregation is a relationship between a whole and its parts.



(d) **Generalizations** are used to show an inheritance relationship between two classes



One class diagram can be drawn for entire system or for each use case.

(v) Design of state chart diagrams: A state chart diagram is used to show the state space of a given class, the event that cause a transition from one state to another, and the action that result from a state change. A state transition diagram for a "book" in the library system is given below.

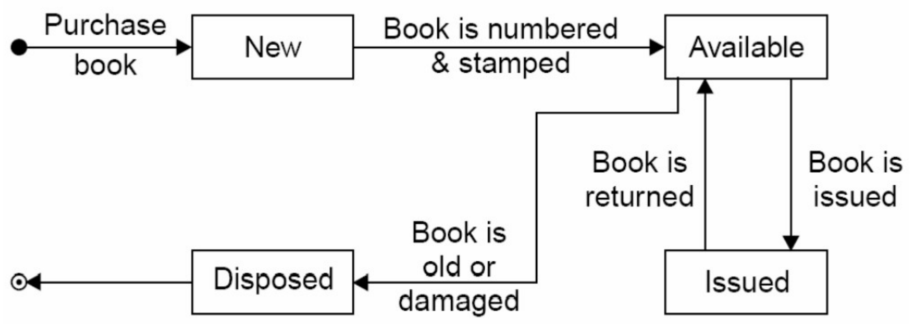


Fig. 28: Transition chart for "book" in a library system.

(vi) Draw component and deployment diagram: Component diagrams are used to visualize the organization and relationships among the physical components (classes, packages, files etc.) in a system.

A deployment diagram focuses on the structure of a software system and is useful for showing the physical distribution of a software system among hardware platforms and execution environments.

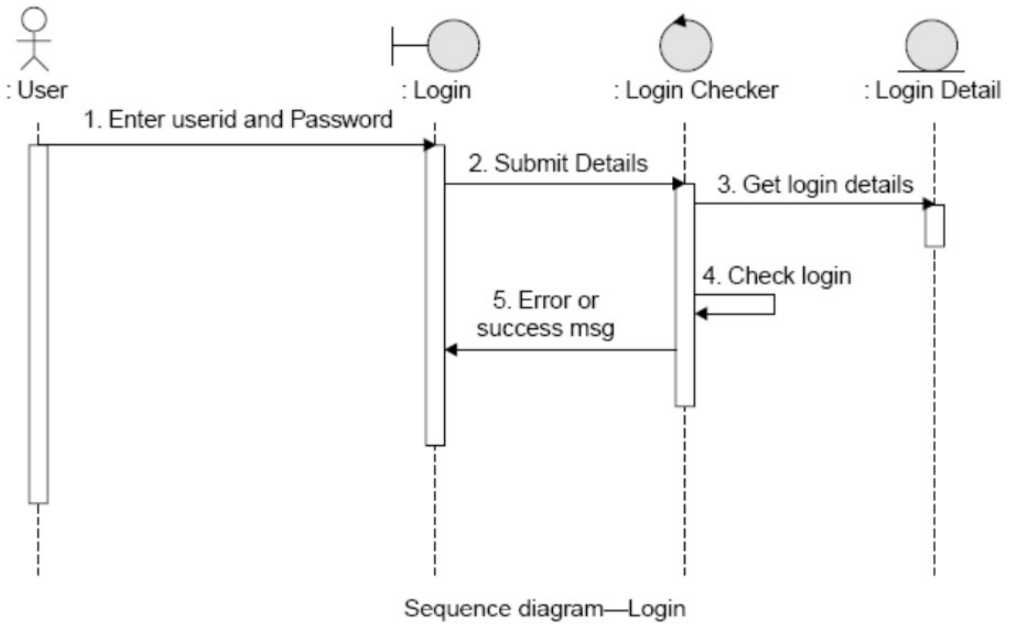


Fig. 29 ***Sequence Diagram***

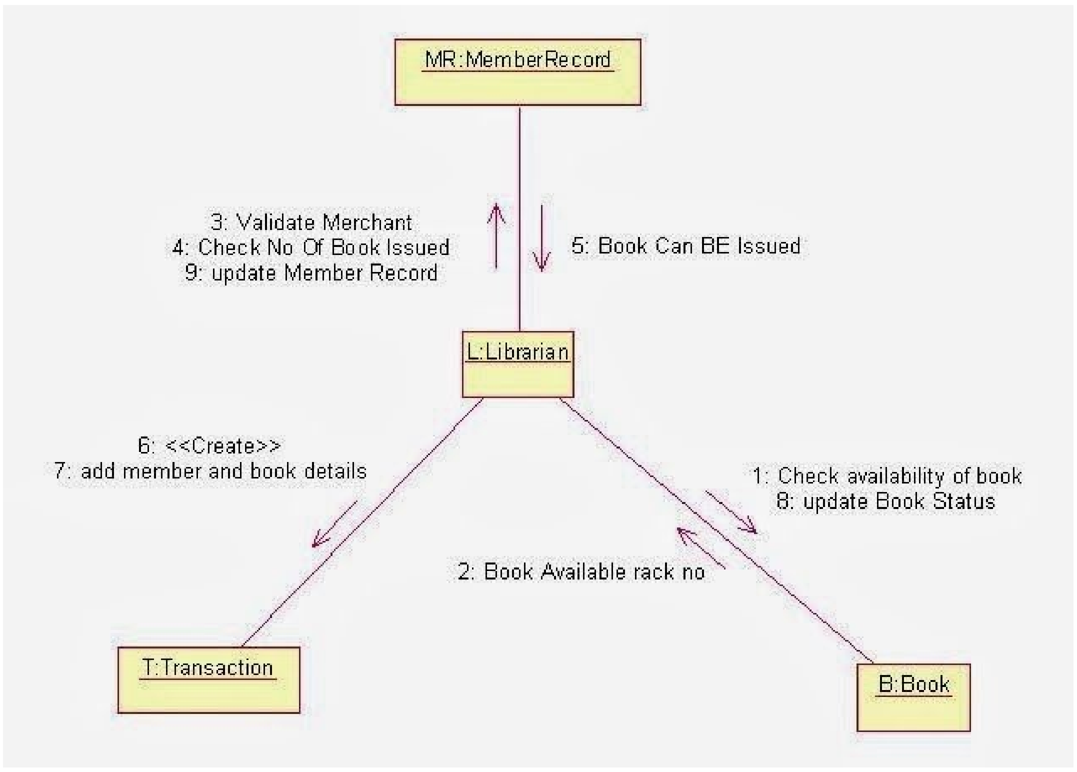


Fig. 30 ***Collaboration Diagram***

4.9 Capability Maturity Model (CMM)

The CMM was developed by SEI (Software Engineering Institute) of Carnegie-Mellon University. It is a strategy for improving the software process, irrespective of the actual life cycle model used.

CMM is used to judge the maturity of the software processes of an organization and to identify the key practices that are required to increase the maturity of these processes. The CMM is organized into five maturity levels as shown below.

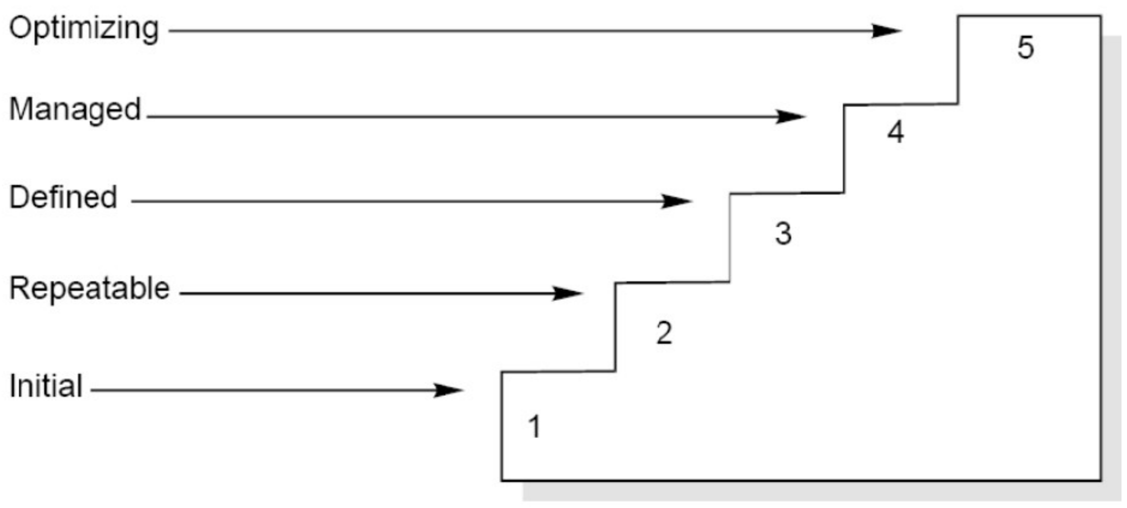


Fig.7.23: Maturity levels of CMM

Maturity Levels:

✓ Initial (Maturity Level 1)

✓ Repeatable (Maturity Level 2)

✓ Defined (Maturity Level 3)

✓ Managed (Maturity Level 4)

✓ Optimizing (Maturity Level 5)

Level-1: Initial –

- No KPA's defined.
- Processes followed are adhoc and immature and are not well defined.
- Unstable environment for software development.
- No basis for predicting product quality, time for completion, etc.

Level-2: Repeatable –

- At this level, the fundamental project management practices like tracking cost and schedule are established. Size and cost estimation methods, like function point analysis, COCOMO, etc. are used.
- Experience with earlier projects is used for managing new similar natured projects.

The key process areas are:

Project Planning- Develop a realistic plan for management and development.

Configuration Management- Ensure consistency of all work products.

Requirements Management-Common agreement on requirements by both customers and developers.

Subcontract Management- Select subcontractors if needed and manage them.

Software Quality Assurance- Periodically check quality factors of both process and product.

Level-3: Defined –

- At this level, documentation of the standard guidelines and procedures takes place.
- It is a well defined integrated set of project specific software engineering and management processes.

The key process areas are:

Peer Reviews- Earlier detection and removal of defects with reviews.

Intergroup Coordination- It consists of planned interactions between different development teams to ensure efficient and proper fulfilment of customer needs.

Organization Process Definition- It's key focus is on the development and maintenance of the standard development processes.

Organization Process Focus- It includes activities and practices that should be followed to improve the process capabilities of an organization.

Training Programs- It focuses on the enhancement of knowledge and skills of the team members including the developers and ensuring an increase in work efficiency.

Level-4: Managed –

At this level, the focus is on software metrics. Two kinds of metrics are composed. Product metrics measure the features of the product being developed, such as its size, reliability, time complexity, understandability, etc. Process metrics follow the effectiveness of the process being used, such as average defect correction time, productivity, the average number of defects found per hour inspection, the average number of failures detected during testing per LOC, etc. The software process and product quality are measured, and quantitative quality requirements for the product are met. Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality. The process metrics are used to analyze if a project performed satisfactorily. Thus, the outcome of process measurements is used to calculate project performance rather than improve the process. The key process areas are:

Software Quality Management- Quantitative control of products quality.

Quantitative Process Management- Quantitative control of process performance.

Level-5: Optimizing –

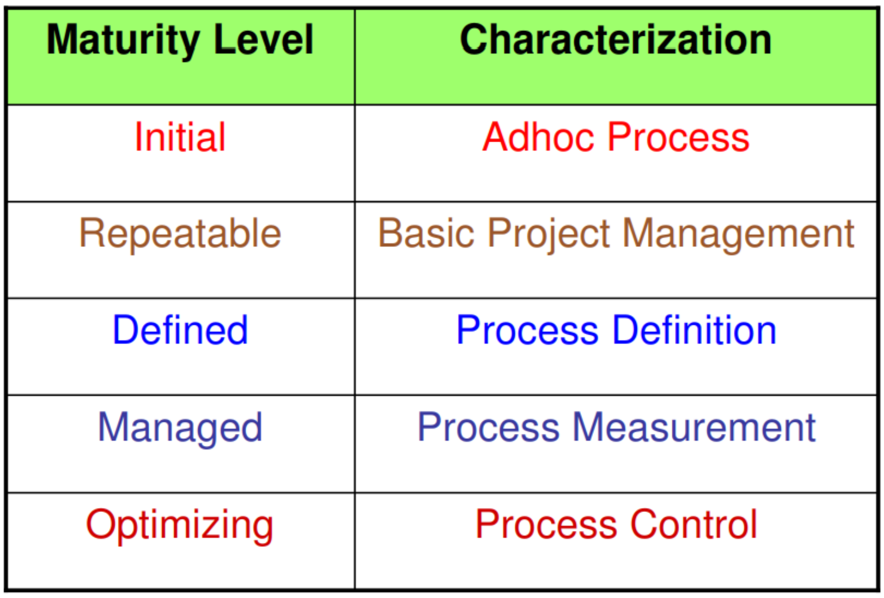
- At this phase, process and product metrics are collected. Process and product measurement data are evaluated for continuous process improvement.
- Use of new tools, techniques and evaluation of software processes is done to prevent recurrence of known defects.

The key areas are:

Process Change Management- Its focus is on the continuous improvement of organization’s software processes to improve productivity, quality and cycle time for the software product.

Technology Change Management- It consists of identification and use of new technologies to improve product quality and decrease the product development time.

Defect Prevention- It focuses on identification of causes of defects and to prevent them from recurring in future projects by improving project defined process.



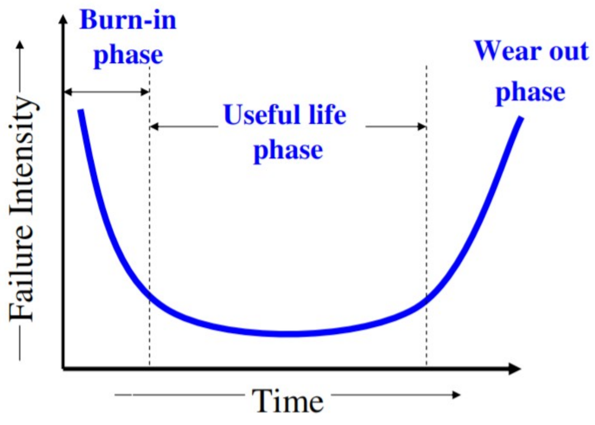
Maturity Level	Characterization
Initial	Adhoc Process
Repeatable	Basic Project Management
Defined	Process Definition
Managed	Process Measurement
Optimizing	Process Control

Fig.7.24: The five levels of CMM

4.7 Software Reliability

4.7.1 Hardware and Software Reliability

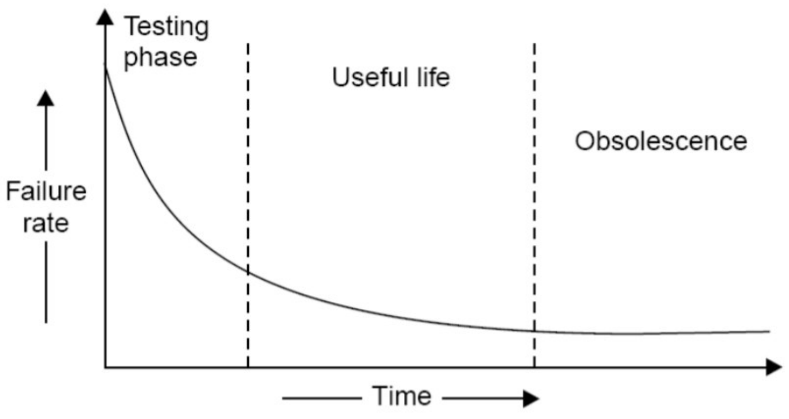
There are three phases in the life of any hardware component i.e., burn-in, useful life & wear-out. In burn-in phase, failure rate is quite high initially, and it starts decreasing gradually as the time progresses. During useful life period, failure rate is approximately constant. Failure rate increase in wear-out phase due to wearing out/aging of components. The best period is useful life period. The shape of this curve is like a "bath tub" and that is why it is known as bath tub curve. The "bath tub curve" is given below.



We do not have wear out phase in software. Software becomes more reliable over time, instead of wearing out. It becomes obsolete due to following reasons.

- change in environment
- change in infrastructure/technology
- major change in requirements
- increase in complexity
- extremely difficult to maintain
- deterioration in structure of the code
- slow execution speed
- poor graphical user interfaces

The expected curve for software is given below.



4.7.2 Definition of Software Reliability:-As per IEEE standard: "Software reliability is defined as the ability of a system or component to perform its required functions under stated conditions for a specified period of time".

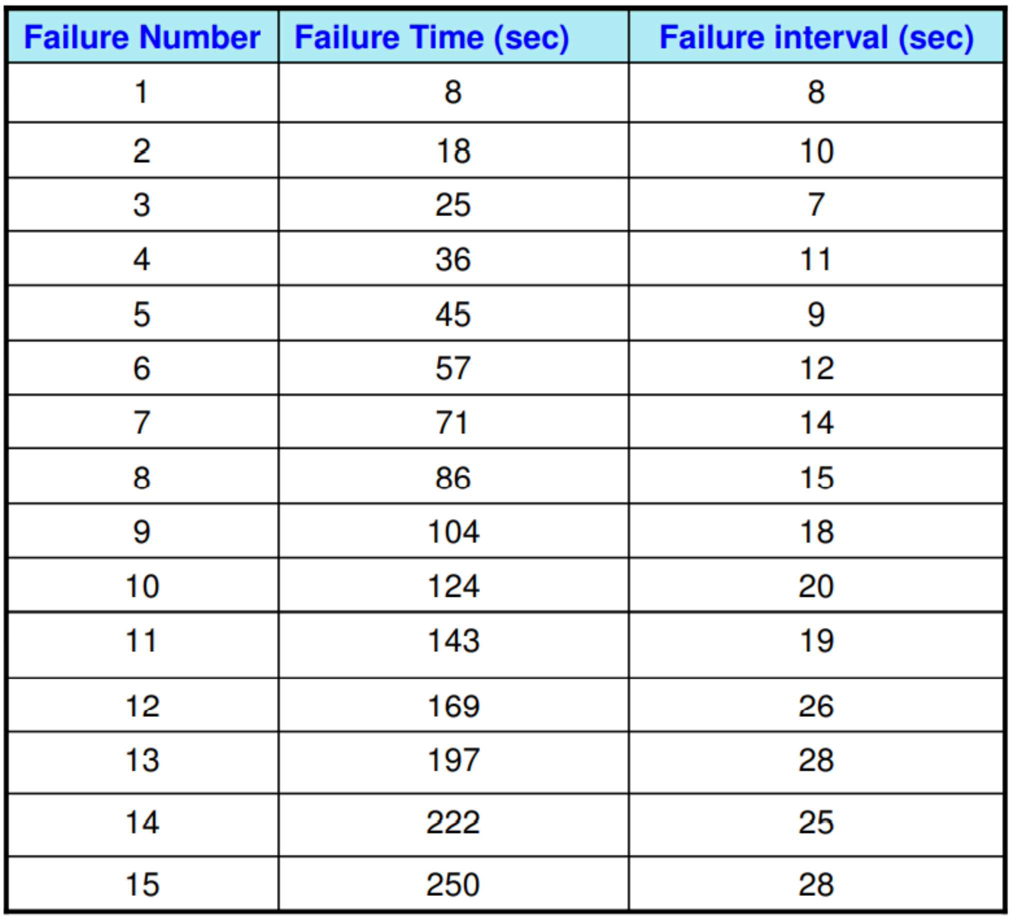
The most acceptable definition of software reliability is: "It is the probability of a failure free operation of a program for a specified time in a specified environment".

4.7.3 Failures and Faults

A fault is the defect in the program that, when executed under particular conditions, causes a failure. The execution time for a program is the time that is actually spent by a processor in executing the instructions of that program. The second kind of time is calendar time. It is the familiar time that we normally experience.

There are four general ways of characterising failure occurrences in time:

1. time of failure,
2. time interval between failures,
3. cumulative failure experienced up to a given time,
4. failures experienced in a time interval.



Failure Number	Failure Time (sec)	Failure interval (sec)
1	8	8
2	18	10
3	25	7
4	36	11
5	45	9
6	57	12
7	71	14
8	86	15
9	104	18
10	124	20
11	143	19
12	169	26
13	197	28
14	222	25
15	250	28

Table 7.1: Time based failure specification

Time (sec)	Cumulative Failures	Failure in interval (30 sec)
30	3	3
60	6	3
90	8	2
120	9	1
150	11	2
180	12	1
210	13	1
240	14	1

Table 7.2: Failure based failure specification

4.8 Software Quality

Our objective of software engineering is to produce good quality maintainable software in time and within budget. Here quality is very important.

Different people understand different meanings of quality like:

- conformance to requirements
- fitness for the purpose
- level of satisfaction

4.8.1 McCall Software Quality Model

Quality can be defined as "conformance to requirements". This model distinguishes between two levels of quality attributes. Higher level quality attributes are known as quality factors. The second level of quality attributes is named as quality criteria. The software quality factors are organised in three product quality factors are shown below.

Product Operation:- Factors which are related to the operation of a product are combined and are given below.

- Correctness: The extent to which a software meets its specifications.
- Efficiency: The amount of computing resources and code required by software to perform a function.
- Integrity: The extent to which access to software or data by the unauthorized persons can be controlled.

- Reliability: The extent to which a software performs its intended functions without failure.
- Usability: The extent of effort required to learn, operate and understand the functions of the software.

Product Revision:- The factors which are required for testing & maintenance are combined and are given below:

Maintainability: The effort required to locate and fix an error during maintenance phase.

Flexibility: The effort required to modify an operational program.

Testability: The effort required to test a software to ensure that it performs its intended functions

Product Transition:- We may have to transfer a product from one platform to an other platform or from one technology to another technology. The factors related to such a transfer are combined and given below:

- Portability: The effort required to transfer a program from one platform to another platform.
- Reusability: The extent to which a program can be reused in other applications.
- Interoperability: The effort required to couple one system with another.

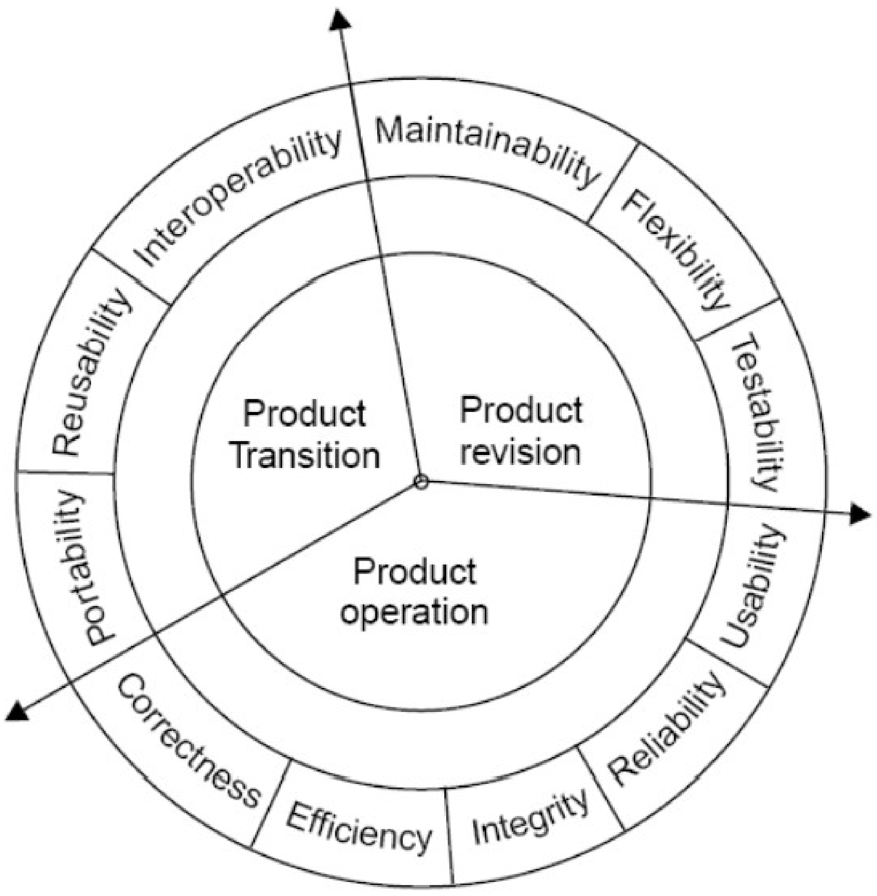


Fig 7.9: Software quality factors

We have 11 quality factors and each quality factor has many second level quality attributes as shown below.

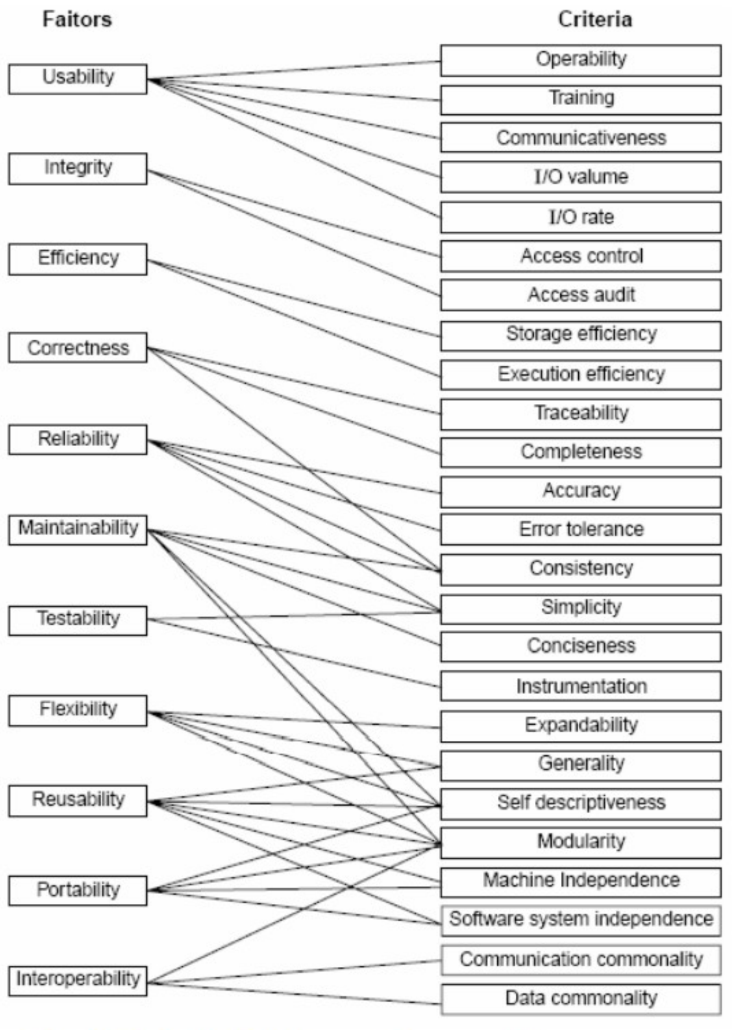


Fig 7.10: McCall's quality model

1	Operability	The ease of operation of the software.
2	Training	The ease with which new users can use the system.
3	Communicativeness	The ease with which inputs and outputs can be assimilated.
4	I/O volume	It is related to the I/O volume.
5	I/O rate	It is the indication of I/O rate.
6	Access control	The provisions for control and protection of the software and data.
7	Access audit	The ease with which software and data can be checked for compliance with standards or other requirements.
8	Storage efficiency	The run time storage requirements of the software.
9	Execution efficiency	The run-time efficiency of the software.

10	Traceability	The ability to link software components to requirements.
11	Completeness	The degree to which a full implementation of the required functionality has been achieved.
12	Accuracy	The precision of computations and output.
13	Error tolerance	The degree to which continuity of operation is ensured under adverse conditions.
14	Consistency	The use of uniform design and implementation techniques and notations throughout a project.
15	Simplicity	The ease with which the software can be understood.
16	Conciseness	The compactness of the source code, in terms of lines of code.
17	Instrumentation	The degree to which the software provides for measurements of its use or identification of errors.
18	Expandability	The degree to which storage requirements or software functions can be expanded.
19	Generability	The breadth of the potential application of software components.
20	Self-descriptiveness	The degree to which the documents are self explanatory.
21	Modularity	The provision of highly independent modules.
22	Machine independence	The degree to which software is dependent on its associated hardware.
23	Software system independence	The degree to which software is independent of its environment.
24	Communication commonality	The degree to which standard protocols and interfaces are used.
25	Data commonality	The use of standard data representations.

Table 7.5 (b): Software quality criteria