MODULE-2

Software Requirements Engineering

Requirements Analysis

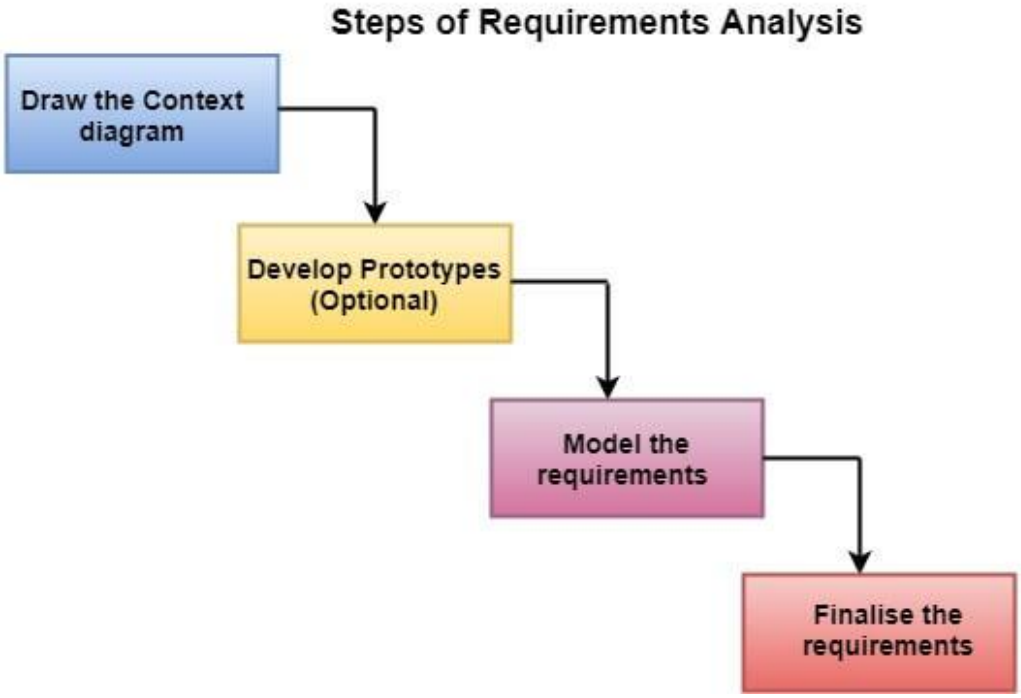
Requirement analysis is significant and essential activity after elicitation. We analyze, refine, and scrutinize the gathered requirements to make consistent and unambiguous requirements.

This activity reviews all requirements and may provide a graphical view of the entire system.

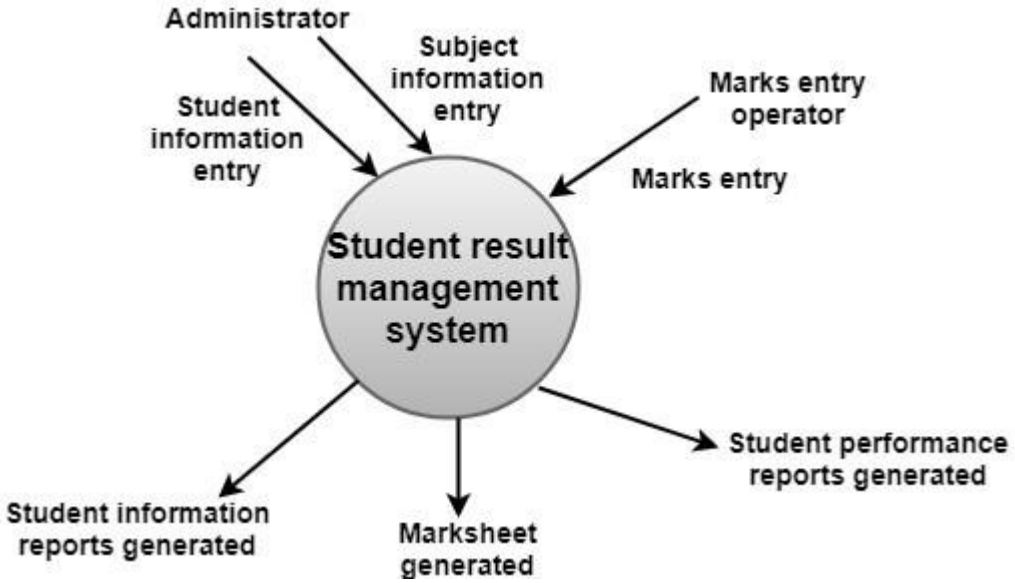
After the completion of the analysis, it is expected that the understandability of the project may improve significantly.

Here, we may also use the interaction with the customer to clarify points of confusion and to understand which requirements are more important than others.

The various steps of requirement analysis are shown in fig:



(i) Draw the context diagram: The context diagram is a simple model that defines the boundaries and interfaces of the proposed systems with the external world. It identifies the entities outside the proposed system that interact with the system. The context diagram of student result management system is given below:



(**ii) Development of a Prototype (optional):** One effective way to find out what the customer wants is to

construct aprototype, something that looks and preferably acts as part of the system they say they want.

We can use their feedback to modify the prototype until the customer is satisfied continuously. Hence, the prototype helps the client to visualize the proposed system and increase the understanding of the requirements. When developers and users are not sure about some of the elements, a prototype may help both the parties to take a final decision.

Some projects are developed for the general market. In such cases, the prototype should be shown to some representative sample of the population of potential purchasers. Even though a person who tries out a prototype may not buy the final system, but their feedback may allow us to make the product more attractive to others.

The prototype should be built quickly and at a relatively low cost. Hence it will always have limitations and would not be acceptable in the final system. This is an optional activity.

(iii) Model the requirements: This process usually consists of various graphical representations of the functions, data entities, external entities, and the relationships between them. The graphical view may help to find incorrect, inconsistent, missing, and superfluous requirements. Such models include the Data Flow diagram, Entity-Relationship diagram, Data Dictionaries, State-transition diagrams, etc.

(iv) Finalise the requirements: After modeling the requirements, we will have a better understanding of the system behavior. The inconsistencies and ambiguities have been identified and corrected. The flow of data amongst various modules has been analyzed. Elicitation and analyze activities have provided better insight into the system. Now we finalize the analyzed requirements, and the next step is to document these requirements in a prescribed format.

Functional vs Non Functional Requirements

Requirements analysis is very critical process that enables the success of a system or software project to be assessed. Requirements are generally split into two types: *Functional* and *Non-functional requirements*.

Functional Requirements: These are the requirements that the end user specifically demands as basic facilities that the system should offer. All these functionalities need to be necessarily incorporated into the system as a part of the contract. These are represented or stated in the form of input to be given to the system, the operation performed and the output expected. They are basically the requirements stated by the user which one can see directly in the final product, unlike the non-functional requirements.

Non-functional requirements: These are basically the quality constraints that the system must satisfy according to the project contract. The priority or extent to which these factors are implemented varies from one project to

other. They are also called non-behavioral requirements. They basically deal with issues like:

- Portability
- Security
- Maintainability
- Reliability
- Scalability
- Performance
- Reusability
- Flexibility

Following are the differences between Functional and Non Functional Requirements

Functional Requirements	**Non Functional Requirements**
A functional requirement defines a system or its component.	A non-functional requirement defines the quality attribute of a software system.
It specifies “What should the software system do?”	It places constraints on “How should the software system fulfill the functional requirements?”

Functional Requirements	Non Functional Requirements
Functional requirement is specified by User.	Non-functional requirement is specified by technical peoples e.g. Architect, Technical leaders and software developers.
It is mandatory.	It is not mandatory.
It is captured in use case.	It is captured as a quality attribute.
Defined at a component level.	Applied to a system as a whole.
Helps you verify the functionality of the software.	Helps you to verify the performance of the software.
Functional Testing like System, Integration, End to End, API testing, etc are done.	Non-Functional Testing like Performance, Stress, Usability, Security testing, etc are done.

Functional Requirements	Non Functional Requirements
Usually easy to define.	Usually more difficult to define.
Example **1)** Authentication of user whenever he/she logs into the system. **2)** System shutdown in case of a cyber attack. **3)** A Verification email is sent to user whenever he/she registers for the first time on some software system.	**Example** **1)** Emails should be sent with a latency of no greater than 12 hours from such an activity. **2)** The processing of each request should be done within 10 seconds **3)** The site should load in 3 seconds when the number of simultaneous users are > 10000

Software Requirement Specification (SRS)

n order to form a good SRS, here you will see some points which can be used and should be considered to form a structure of good SRS. These are as follows : 1. Introduction

- **(i)** Purpose of this document
- **(ii)** Scope of this document
- **(iii)** Overview

2. General description 3. Functional Requirements 4. Interface Requirements 5. Performance Requirements 6. Design Constraints 7. Non-Functional Attributes 8. Preliminary Schedule and Budget 9. Appendices **Software Requirement Specification (SRS) Format** as name suggests, is complete specification and description of requirements of software that needs to be fulfilled for successful development of software system. These requirements can be functional as well as non-functional depending upon type of requirement. The interaction between different customers and contractor is done because its necessary to fully understand needs of customers.

Depending upon information gathered after interaction, SRS is developed which describes requirements of software that may include changes and modifications that is needed to be done to increase quality of product and to satisfy customer's demand.

1. **Introduction :**
    - **(i) Purpose of this Document –** At first, main aim of why this document is necessary and what's purpose of document is explained and described.
    - **(ii) Scope of this document –** In this, overall working and main objective of document and what value it will provide to customer is described and explained. It also includes a description of development cost and time required.

- **(iii) Overview –** In this, description of product is explained. It's simply summary or overall review of product.

2. **General description :** In this, general functions of product which includes objective of user, a user characteristic, features, benefits, about why its importance is mentioned. It also describes features of user community.
3. **Functional Requirements :** In this, possible outcome of software system which includes effects due to operation of program is fully explained. All functional requirements which may include calculations, data processing, etc. are placed in a ranked order.
4. **Interface Requirements :** In this, software interfaces which mean how software program communicates with each other or users either in form of any language, code, or message are fully described and explained. Examples can be shared memory, data streams, etc.
5. **Performance Requirements :** In this, how a software system performs desired functions under specific condition is explained. It also explains required time, required memory, maximum error rate, etc.
6. **Design Constraints :** In this, constraints which simply means limitation or restriction are specified and explained for design team. Examples may include

use of a particular algorithm, hardware and software limitations, etc.

7. **Non-Functional Attributes :** In this, non-functional attributes are explained that are required by software system for better performance. An example may include Security, Portability, Reliability, Reusability, Application compatibility, Data integrity, Scalability capacity, etc.
8. **Preliminary Schedule and Budget :** In this, initial version and budget of project plan are explained which include overall time duration required and overall cost required for development of project.
9. **Appendices :** In this, additional information like references from where information is gathered, definitions of some specific terms, acronyms, abbreviations, etc. are given and explained.

Uses of SRS document:

1. Devlopment team require it for developing product according to the need.
2. Test plans are generated by testing group based on the describe external behavior.
3. Maintenance and support staff need it to understand what the software product is supposed to do.
4. Project manager base their plans and estimates of schedule, effort and resources on it.
5. customer rely on it to know that product they can expect.

6. As a contract between developer and customer.
7. in documentation purpose.

IEEE 830 Guidelines

It describes the content and qualities of a good software requirements speciÞcation (SRS) and presents several sample SRS outlines.

Like many IEEE standards for software engineering, Standard 830 includes guidance and recommended approaches for specifying software requirements. It's not a complete tutorial on requirements development, but it does contain some useful information. The bulk of the text is a detailed suggested template for organizing the different kinds of requirements information for a software product -- an SRS.

The heart of the SRS consists of descriptions of both functional and nonfunctional requirements. The IEEE standard provides several suggestions of how to organize functional requirements: by mode, user class, object, feature, stimulus, functional hierarchy or combinations of these criteria. There is no single organizational approach that's best; use whatever makes sense for your project.

Decision Tree and Decision Table

Decision Tree

A Decision Tree is a graph that uses a branching method to display all the possible outcomes of any decision.

It helps in processing logic involved in decision-making, and corresponding actions are taken.

It is a diagram that shows conditions and their alternative actions within a horizontal tree framework.

It helps the analyst consider the sequence of decisions and identifies the accurate decision that must be made.

Links are used for decisions, while Nodes represent goals.

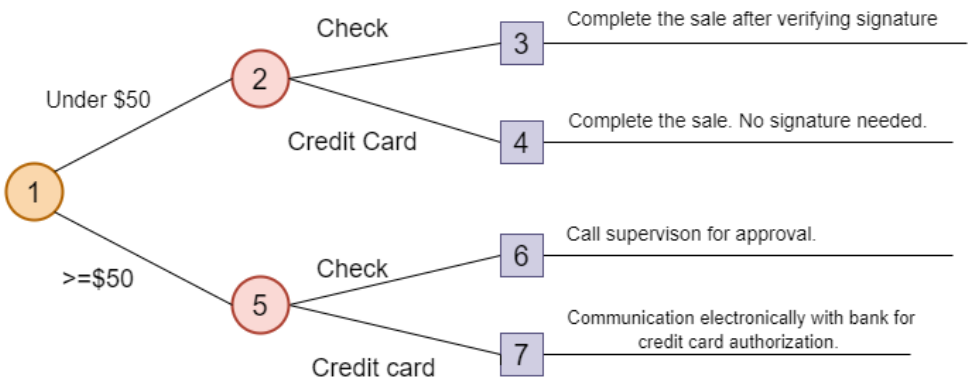
Decision trees simplify the knowledge acquisition process and are more natural than frames and rule knowledge representation techniques.

Let's understand this with an example:

Conditions included the sale amount (under $50) and whether the customer paid by cheque or credit card. The four steps possible were to:

- Complete the sale after verifying the signature.
- Complete the sale with no signature needed.
- Communicate electronically with the bank for credit card authorization.
- Call the supervisor for approval.

The below figure illustrates how this example can be drawn as a decision tree. In drawing the tree.



Drawing a decision tree to show the noncash purchase approval actions for a department store.

1. Identify all conditions and actions and their order and timing (if they are critical).
2. Begin building the tree from left to right, making sure you list all possible alternatives before moving to the right.

Advantages of decision trees

- Decision trees represent the logic of If-Else in a pictorial form.
- Decision trees help the analyst to identify the actual decision to be made.
- Decision trees are useful for expressing the logic when the value is variable or action depending on a nested decision.
- It is used to verify the problems that involve a limited number of actions.

Decision Tables

Data is stored in the tabular form inside decision tables using rows and columns. A decision table contains condition entries, condition stubs, action entries, and action stubs. The upper left quadrant contains conditions. The upper right quadrant contains condition alternatives or rules. The lower right quadrant contains action rules,

and the lower-left quadrant contains actions to be taken. Verification and validation of the decision table are much easy to check, such as Inconsistencies, Contradictions, Incompleteness, and Redundancy.

Example of Decision Table

Let's consider the decision table given in table 1.

In the table, there are multiple rules for a single Decision. The rules from a decision table can be made by just putting AND between conditions.

The major rules which can be extracted (taken out) from the table are:

- R1 = If (working-day = Y) ^ (holiday = N) ^ (Rainy-day = Y) Then, Go to office.
- R2 = If (working-day = N) ^ (holiday = N) ^ (Rainy-day = N) Then, Go to office.
- R3 = If (working-day = N) ^ (holiday = Y) ^ (Rainy-day = Y) Then, Watch TV.
- R4 = If (working-day = N) ^ (holiday = Y) ^ (Rainy-day = N) Then, Go to picnic.

The above rules can be optimized by:

Optimized R1= If (working-day = Y) then Go to office

Or

Optimized R1= If (holiday = N) then Go to office

Optimized R3= If (working-day = N) ^ (Rainy-day = Y) Then Watch TV

Or

Optimized R3= If (holiday = Y) ^ (Rainy-day = Y) Then Watch TV

Optimized R4= If (working-day = N) ^ (Rainy-day = N)

Then go to the picnic.

Or

Optimized R4= If (holiday = Y) ^ (Rainy-day = N)

Then go to the picnic.

The tree given below is the resultant tree of Table 1.

The following rules are constructed from the decision tree as shown below.

R1= If (Day = Working) ^ (Outlook = Rainy)

Then Go To Office

R2= If (Day = Working) ^ (Outlook = Sunny)

Then Go To Office

R3= If (Day = Holiday) ^ (Outlook = Rainy)

Then Watch TV

R4=If (Day = Holiday) ^ (Outlook = Sunny)

Then Go To Picnic

In R1 and R2, there is no need to check the condition Outlook = Rainy and Outlook = Sunny if day = working because if the day is working, whether it is a sunny or

rainy day, the decision is to Go to the office. The following rules are the optimized version of R1 and R2 above rules.

R1 optimized: If (Day = Working) Then Go To Office

R2 optimized: If (Day = Working) Then Go To Office

The refinement/optimization step result is effective, efficient, and accurate rules.

Conversion of decision table into decision tree

Data can be transformed from a decision table into a tree structure. The decision table can be converted into a decision tree by using the conversion method discussed or some other technique. The resultant tree has two categories: balanced trees and unbalanced trees. The figure shows the input and output of the conversion process.

Conversion of decision table into decision tree

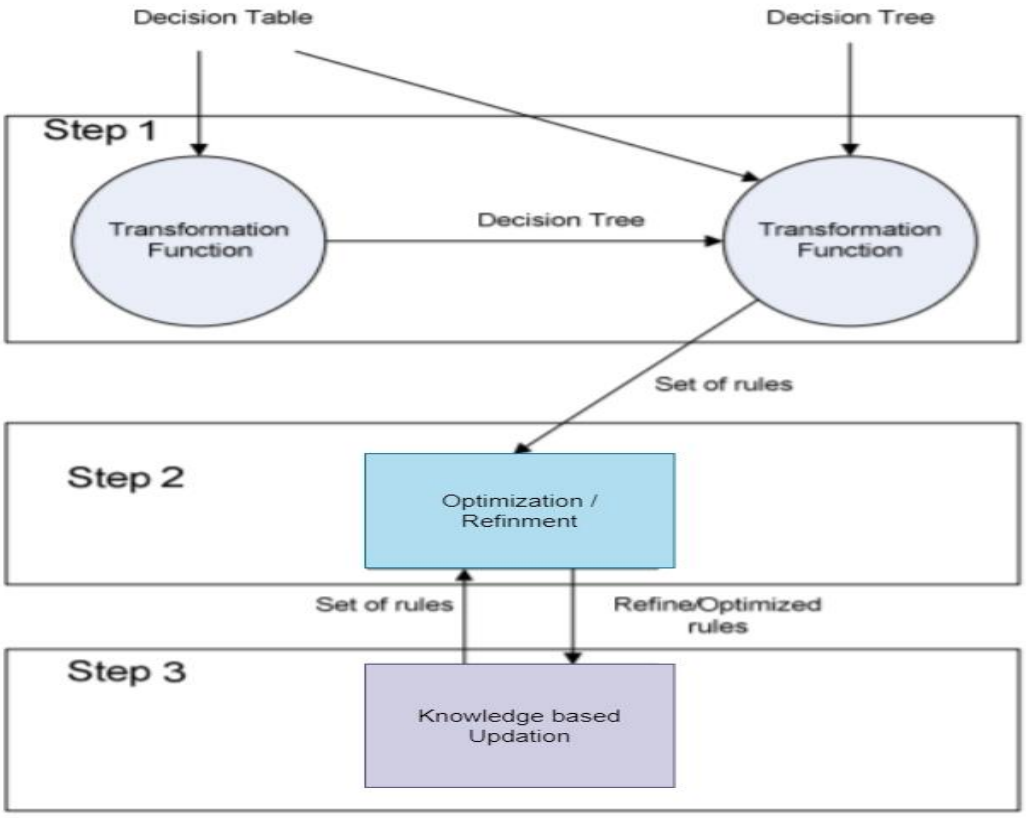


Figure 2. Proposed Framework

Advantages of a Decision tree over Decision table

- The decision tree takes advantage of the sequential structure of decision tree branches to notice the order of checking conditions and executing actions immediately.
- Decision tree is used to verify the problems that involve a limited number of actions.
- All those actions and conditions that are critical are connected directly to other conditions and actions, whereas the conditions that do not matter are absent. In other words, the trees do not have to be symmetrical.

- Decision tree is helpful to express the logic when the value is variable, or action is dependent on the nested decision.

Difference between Decision Table and Decision Tree:

S. No.	Decision Table	Decision Tree
1.	Decision Tables are a tabular representation of conditions and actions.	Decision Trees are a graphical representation of every possible outcome of a decision.
2.	We can derive a decision table from the decision tree.	We can not derive a decision tree from the decision table.
3.	It helps to clarify the criteria.	It helps to take into account the possible relevant outcomes of the decision.
4.	In Decision Tables, we can include more than one ‘or’ condition.	In Decision Trees, we can not include more than one ‘or’ condition.
5.	It is used when there are small number of properties.	It is used when there are more number of properties.
6.	It is used for simple	It can be used for complex

S. No.	Decision Table	Decision Tree
	logic only.	logic as well.
7.	It is constructed of rows and tables.	It is constructed of branches and nodes.
8.	The goal of using a decision table is the generation of rules for structuring logic on the basis of data entered in the table.	A decision tree's objective is to provide an effective means to visualize and understand a decision's available possibilities and range of possible outcomes.

Structured Analysis and Design

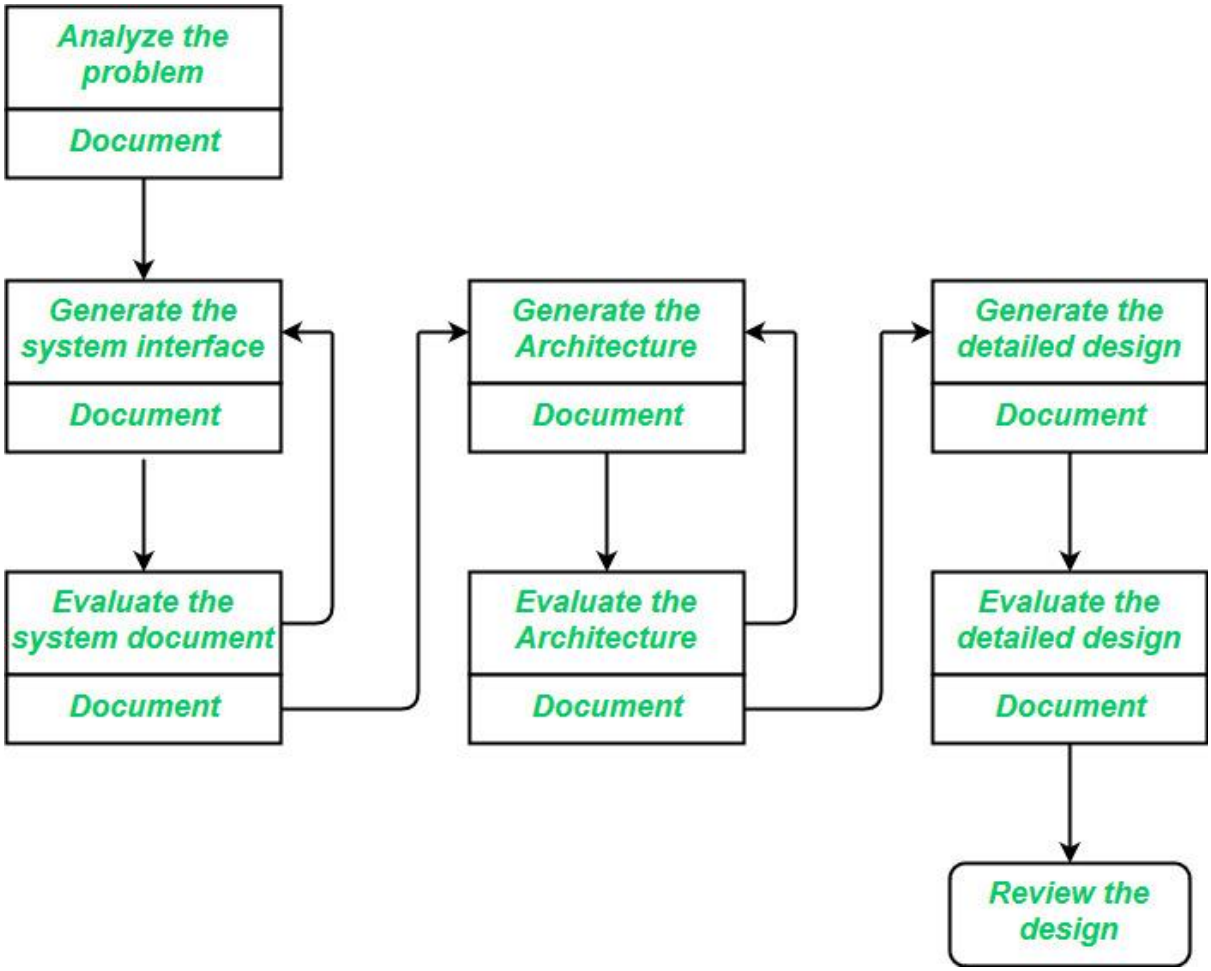
Software Design Process

The design phase of software development deals with transforming the customer requirements as described in the SRS documents into a form implementable using a programming language. The software design process can be divided into the following three levels of phases of design:

1. Interface Design
2. Architectural Design
3. Detailed Design

Elements of a System:

1. **Architecture –** This is the conceptual model that defines the structure, behavior, and views of a system. We can use flowcharts to represent and illustrate the architecture.
2. **Modules –** These are components that handle one specific task in a system. A combination of the modules makes up the system.
3. **Components –** This provides a particular function or group of related functions. They are made up of modules.
4. **Interfaces –** This is the shared boundary across which the components of a system exchange information and relate.
5. **Data** – This is the management of the information and data flow.



Interface Design: ***Interface design*** **is the specification of the** interaction between a system and its environment. this phase proceeds at a high level of abstraction with respect to the inner workings of the system i.e, during interface design, the internal of the systems are completely ignored and the system is treated as a black box. Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts. The design problem statement produced during the problem analysis step should identify the people, other systems, and devices which are

collectively called *agents*. Interface design should include the following details:

- Precise description of events in the environment, or messages from agents to which the system must respond.
- Precise description of the events or messages that the system must produce.
- Specification of the data, and the formats of the data coming into and going out of the system.
- Specification of the ordering and timing relationships between incoming events or messages, and outgoing events or outputs**.**

Architectural Design: *Architectural design* is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them. In architectural design, the overall structure of the system is chosen, but the internal details of major components are ignored. Issues in architectural design includes:

- Gross decomposition of the systems into major components.
- Allocation of functional responsibilities to components.
- Component Interfaces
- Component scaling and performance properties, resource consumption properties, reliability properties, and so forth.

- Communication and interaction between components.

The architectural design adds important details ignored during the interface design. Design of the internals of the major components is ignored until the last phase of the design. **Detailed Design:** *Design* is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures. The detailed design may include:

- Decomposition of major system components into program units.
- Allocation of functional responsibilities to units.
- User interfaces
- Unit states and state changes
- Data and control interaction between units
- Data packaging and implementation, including issues of scope and visibility of program elements
- Algorithms and data structures

High level and detailed design

1. High Level Design :

High Level Design in short HLD is the general system design means it refers to the overall system design. It describes the overall description/architecture of the application. It includes the description of system architecture, data base design, brief description on systems, services, platforms and relationship among

modules. It is also known as macro level/system design. It is created by solution architect. It converts the Business/client requirement into High Level Solution. It is created first means before Low Level Design.

2. Low Level Design :

Low Level Design in short LLD is like detailing HLD means it refers to component-level design process. It describes detailed description of each and every module means it includes actual logic for every system component and it goes deep into each modules specification. It is also known as micro level/detailed design. It is created by designers and developers. It converts the High Level Solution into Detailed solution. It is created second means after High Level Design.

Difference between High Level Design and Low Level Design :

S.No.	HIGH LEVEL DESIGN	LOW LEVEL DESIGN
01.	High Level Design is the general system design means it refers to the overall system design.	Low Level Design is like detailing HLD means it refers to component-level design process.
02.	High Level Design in short called as HLD.	Low Level Design in short called as LLD.

S.No.	HIGH LEVEL DESIGN	LOW LEVEL DESIGN
03.	It is also known as macro level/system design.	It is also known as micro level/detailed design.
04.	It describes the overall description/architecture of the application.	It describes detailed description of each and every module.
05.	High Level Design expresses the brief functionality of each module.	Low Level Design expresses details functional logic of the module.
06.	It is created by solution architect.	It is created by designers and developers.
07.	Here in High Level Design the participants are design team, review team, and client team.	Here in Low Level Design participants are design team, Operation Teams, and Implementers.

S.No.	HIGH LEVEL DESIGN	LOW LEVEL DESIGN
08.	It is created first means before Low Level Design.	It is created second means after High Level Design.
09.	In HLD the input criteria is Software Requirement Specification (SRS).	In LLD the input criteria is reviewed High Level Design (HLD).
10.	High Level Solution converts the Business/client requirement into High Level Solution.	Low Level Design converts the High Level Solution into Detailed solution.
11.	In HLD the output criteria is data base design, functional design and review record.	In LLD the output criteria is program specification and unit test plan.

Coupling and Cohesion

Coupling and Cohesion are two key concepts in software engineering that are used to measure the quality of a software system's design.

Coupling refers to the degree of interdependence between software modules.

High coupling means that modules are closely connected and changes in one module may affect other modules.

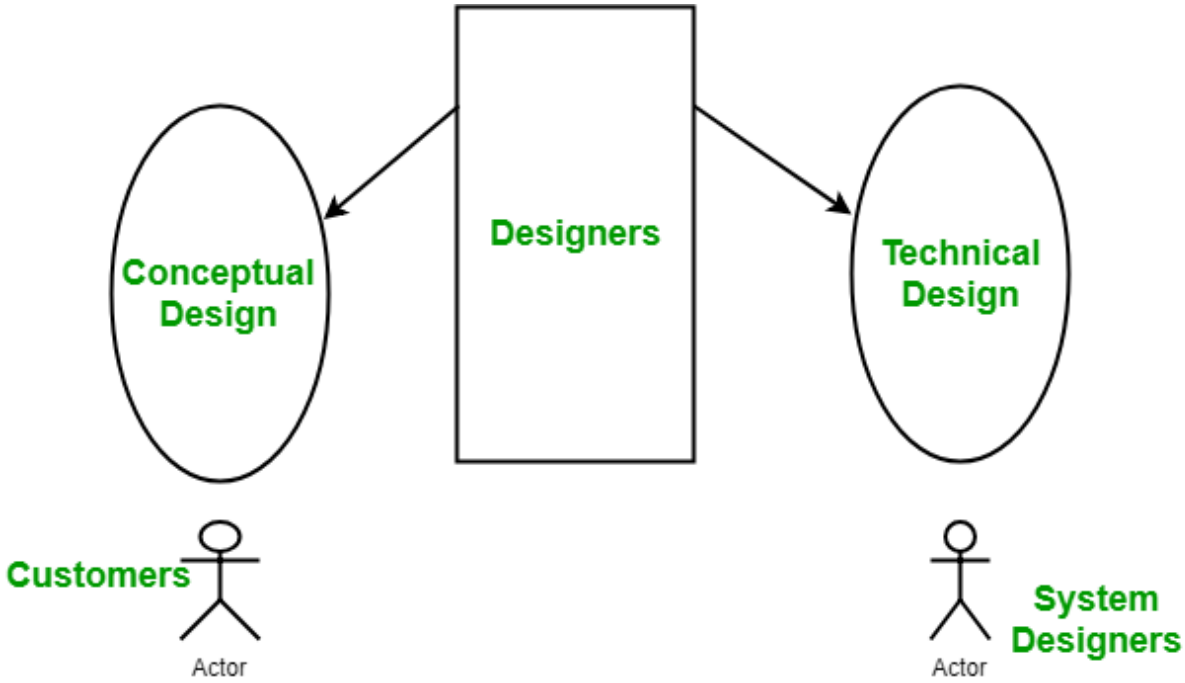
Low coupling means that modules are independent and changes in one module have little impact on other modules.

Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose.

High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes.

Both coupling and cohesion are important factors in determining the maintainability, scalability, and reliability of a software system. High coupling and low cohesion can make a system difficult to change and test, while low coupling and high cohesion make a system easier to maintain and improve.

Basically, design is a two-part iterative process. First part is Conceptual Design that tells the customer what the system will do. Second is Technical Design that allows the system builders to understand the actual hardware and software needed to solve customer's problem.



Conceptual design of the system:

- Written in simple language i.e. customer understandable language.
- Detailed explanation about system characteristics.
- Describes the functionality of the system.
- It is independent of implementation.
- Linked with requirement document.

Technical Design of the system:

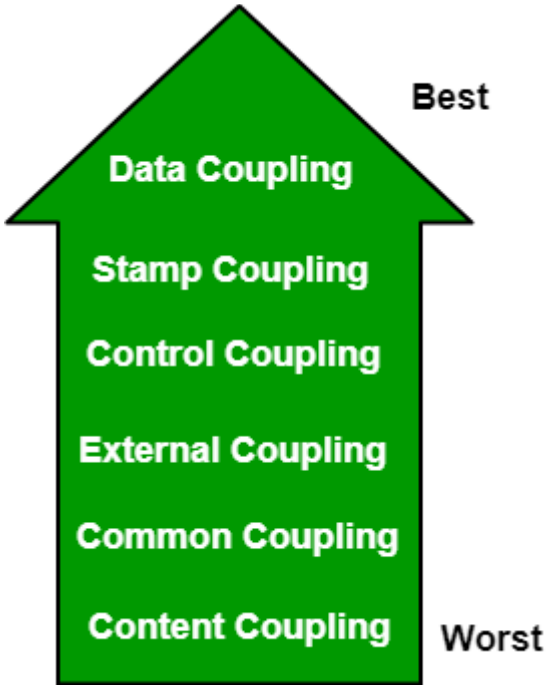
- Hardware component and design.
- Functionality and hierarchy of software components.
- Software architecture
- Network architecture
- Data structure and flow of data.
- I/O component of the system.

- Shows interface.

Modularization: Modularization is the process of dividing a software system into multiple independent modules where each module works independently. There are many advantages of Modularization in software engineering. Some of these are given below:

- Easy to understand the system.
- System maintenance is easy.
- A module can be used many times as their requirements. No need to write it again and again.

Coupling: Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.



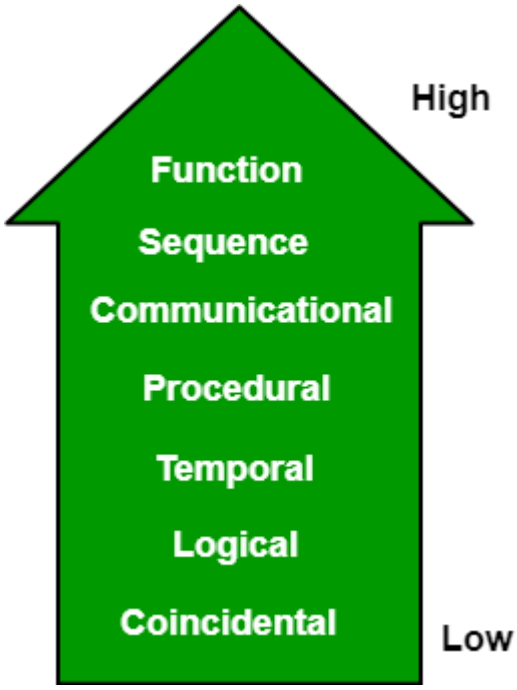
Types of Coupling:

- **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent of each other and communicate through data. Module communications don't contain tramp data. Example-customer billing system.
- **Stamp Coupling** In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice was made by the insightful designer, not a lazy programmer.
- **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example-sort function that takes comparison function as an argument.
- **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.
- **Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules,

reduced ability to control data accesses, and reduced maintainability.

- **Content Coupling:** In a content coupling, one module can modify the data of another module, or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

Cohesion: Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.



Types of Cohesion:

- **Functional Cohesion:** Every essential element for a single computation is contained in the component. A

functional cohesion performs the task and functions. It is an ideal situation.

- **Sequential Cohesion:** An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
- **Communicational Cohesion:** Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer.
- **Procedural Cohesion:** Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.
- **Temporal Cohesion:** The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time.
- **Logical Cohesion:** The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.
- **Coincidental Cohesion:** The elements are not related(unrelated). The elements have no conceptual

relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component.

ADVANTAGES OR DISADVANTAGES:

Advantages of low coupling:

- Improved maintainability: Low coupling reduces the impact of changes in one module on other modules, making it easier to modify or replace individual components without affecting the entire system.
- Enhanced modularity: Low coupling allows modules to be developed and tested in isolation, improving the modularity and reusability of code.
- Better scalability: Low coupling facilitates the addition of new modules and the removal of existing ones, making it easier to scale the system as needed.

Advantages of high cohesion:

- Improved readability and understandability: High cohesion results in clear, focused modules with a single, well-defined purpose, making it easier for developers to understand the code and make changes.
- Better error isolation: High cohesion reduces the likelihood that a change in one part of a module will affect other parts, making it easier to
- isolate and fix errors.Improved reliability: High cohesion leads to modules that are less prone to errors and that function more consistently,

- leading to an overall improvement in the reliability of the system.

Disadvantages of high coupling:

- Increased complexity: High coupling increases the interdependence between modules, making the system more complex and difficult to understand.
- Reduced flexibility: High coupling makes it more difficult to modify or replace individual components without affecting the entire system.
- Decreased modularity: High coupling makes it more difficult to develop and test modules in isolation, reducing the modularity and reusability of code.

Disadvantages of low cohesion:

- Increased code duplication: Low cohesion can lead to the duplication of code, as elements that belong together are split into separate modules.
- Reduced functionality: Low cohesion can result in modules that lack a clear purpose and contain elements that don't belong together, reducing their functionality and making them harder to maintain.
- Difficulty in understanding the module: Low cohesion can make it harder for developers to understand the purpose and behavior of a module, leading to errors and a lack of clarity.

Modularity ad Layering

Modularity

The module simply means the software components that are been created by dividing the software. The software is divided into various components that work together to form a single functioning item but sometimes they can perform as a complete function if not connected with each other. This process of creating software modules is known as **Modularity** in software engineering.

It simply measures the degree to which these components are made up than can be combined.

Some of the projects or software designs are very complex that it's not easy to understand its working and functioning. In such cases, modularity is a key weapon that helps in reducing the complexity of such software or projects.

The basic principle of Modularity is that "Systems should be built from cohesive, loosely coupled components (modules)" which means s system should be made up of different components that are united and work together in an efficient way and such components have a well-defined function.

To define a modular system, several properties or criteria are there under which we can evaluate a design method while considering its abilities.

These criteria are defined by Meyer. Some of them are given below:

1. **Modular Decomposability –**
   Decomposability simply means to break down something into smaller pieces. Modular

decomposability means to break down the problem into different sub-problems in a systematic manner. Solving a large problem is difficult sometimes, so the decomposition helps in reducing the complexity of the problem, and sub-problems created can be solved independently. This helps in achieving the basic principle of modularity.

2. **Modular Composability –**
   Composability simply means the ability to combine modules that are created. It's actually the principle of system design that deals with the way in which two or more components are related or connected to each other. Modular composability means to assemble the modules into a new system that means to connect the combine the components into a new system.
3. **Modular Understandability –**
   Understandability simply means the capability of being understood, quality of comprehensible. Modular understandability means to make it easier for the user to understand each module so that it is very easy to develop software and change it as per requirement. Sometimes it's not easy to understand the process models because of its complexity and its large size in structure. Using modularity understandability, it becomes easier to understand the problem in an efficient way without any issue.
4. **Modular Continuity –**
   Continuity simply means unbroken or consistent or uninterrupted connection for a long period of time

without any change or being stopped. Modular continuity means making changes to the system requirements that will cause changes in the modules individually without causing any effect or change in the overall system or software.

5. **Modular Protection –**
   Protection simply means to keep something safe from any harms, to protect against any unpleasant means or damage. Modular protection means to keep safe the other modules from the abnormal condition occurring in a particular module at run time. The abnormal condition can be an error or failure also known as run-time errors. The side effects of these errors are constrained within the module.

Layering

Software engineering is a fully layered technology, to develop software we need to go from one layer to another. All the layers are connected and each layer demands the fulfillment of the previous layer.

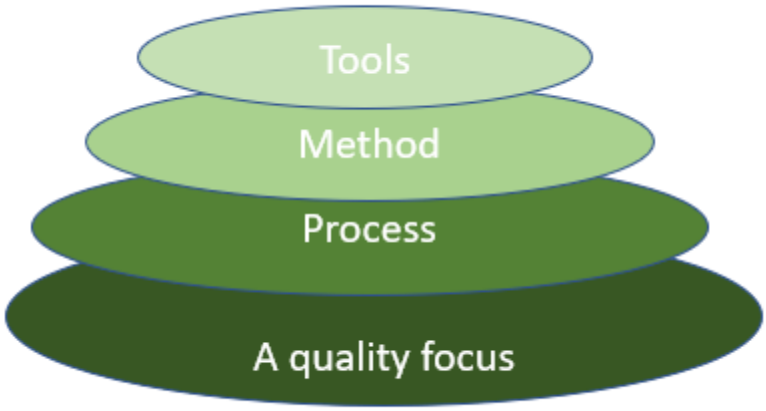
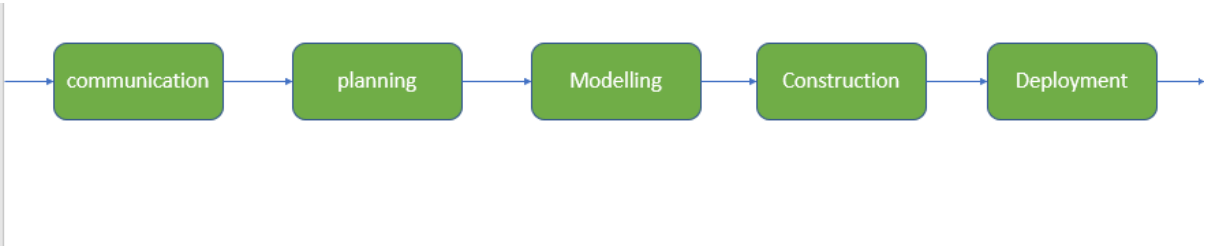


Fig: The diagram shows the layers of software development

Layered technology is divided into four parts:

1. A quality focus: It defines the continuous process improvement principles of software. It provides integrity that means providing security to the software so that data can be accessed by only an authorized person, no outsider can access the data. It also focuses on maintainability and usability.

2. Process: It is the foundation or base layer of software engineering. It is key that binds all the layers together which enables the development of software before the deadline or on time. Process defines a framework that must be established for the effective delivery of software engineering technology. The software process covers all the activities, actions, and tasks required to be carried out for software development.



Process activities are listed below:-

- **Communication:** It is the first and foremost thing for the development of software. Communication is necessary to know the actual demand of the client.
- **Planning:** It basically means drawing a map for reduced the complication of development.

- **Modeling:** In this process, a model is created according to the client for better understanding.
- **Construction:** It includes the coding and testing of the problem.
- **Deployment:-** It includes the delivery of software to the client for evaluation and feedback.

3. Method: During the process of software development the answers to all "how-to-do" questions are given by method. It has the information of all the tasks which includes communication, requirement analysis, design modeling, program construction, testing, and support.

4. Tools: Software engineering tools provide a self-operating system for processes and methods. Tools are integrated which means information created by one tool can be used by another.

Function oriented software design

Structured Analysis

Structured Analysis and Structured Design (SA/SD) is a diagrammatic notation that is designed to help people understand the system. The basic goal of SA/SD is to improve quality and reduce the risk of system failure. It establishes concrete management specifications and documentation. It focuses on the solidity, pliability, and maintainability of the system.

Structured Analysis and Structured Design (SA/SD) is a software development method that was popular in the

1970s and 1980s. The method is based on the principle of structured programming, which emphasizes the importance of breaking down a software system into smaller, more manageable components.

In SA/SD, the software development process is divided into two phases: Structured Analysis and Structured Design. During the Structured Analysis phase, the problem to be solved is analyzed and the requirements are gathered. The Structured Design phase involves designing the system to meet the requirements that were gathered in the Structured Analysis phase.

The following are the steps involved in the SA/SD process:

1. **Requirements gathering:** The first step in the SA/SD process is to gather requirements from stakeholders, including users, customers, and business partners.
2. **Structured Analysis:** During the Structured Analysis phase, the requirements are analyzed to identify the major components of the system, the relationships between those components, and the data flows within the system.
3. **Data Modeling:** During this phase, a data model is created to represent the data used in the system and the relationships between data elements.
4. **Process Modeling:** During this phase, the processes within the system are modeled using flowcharts and data flow diagrams.

5. **Input/Output Design:** During this phase, the inputs and outputs of the system are designed, including the user interface and reports.
6. **Structured Design**: During the Structured Design phase, the system is designed to meet the requirements gathered in the Structured Analysis phase. This may include selecting appropriate hardware and software platforms, designing databases, and defining data structures.
7. **Implementation and Testing:** Once the design is complete, the system is implemented and tested.

SA/SD has been largely replaced by more modern software development methodologies, but its principles of structured analysis and design continue to influence current software development practices. The method is known for its focus on breaking down complex systems into smaller components, which makes it easier to understand and manage the system as a whole.

Basically, the approach of SA/SD is based on the **Data Flow Diagram**. It is easy to understand SA/SD but it focuses on well-defined system boundary whereas the JSD approach is too complex and does not have any graphical representation.

SA/SD is combined known as SAD and it mainly focuses on the following 3 points:

1. System
2. Process

3. Technology

SA/SD involves 2 phases:

1. **Analysis Phase:** It uses Data Flow Diagram, Data Dictionary, State Transition diagram and ER diagram.
2. **Design Phase:** It uses Structure Chart and Pseudo Code.

1. Analysis Phase:
Analysis Phase involves data flow diagram, data dictionary, state transition diagram, and entity-relationship diagram.

1. **Data Flow Diagram:**
   In the data flow diagram, the model describes how the data flows through the system. We can incorporate the Boolean operators and & or link data flow when more than one data flow may be input or output from a process.

For example, if we have to choose between two paths of a process we can add an operator or and if two data flows are necessary for a process we can add an operator. The input of the process “check-order” needs the credit information and order information whereas the output of the process would be a cash-order or a good-credit-order.

2. **Data Dictionary:**
   The content that is not described in the DFD is

described in the data dictionary. It defines the data store and relevant meaning. A physical data dictionary for data elements that flow between processes, between entities, and between processes and entities may be included. This would also include descriptions of data elements that flow external to the data stores.

A logical data dictionary may also be included for each such data element. All system names, whether they are names of entities, types, relations, attributes, or services, should be entered in the dictionary.

3. **State Transition Diagram:**
   State transition diagram is similar to the dynamic model. It specifies how much time the function will take to execute and data access triggered by events. It also describes all of the states that an object can have, the events under which an object changes state, the conditions that must be fulfilled before the transition will occur and the activities were undertaken during the life of an object.

4. **ER Diagram:**
   ER diagram specifies the relationship between data store. It is basically used in database design. It basically describes the relationship between different entities.

2. Design Phase:

Design Phase involves structure chart and pseudocode.

1. **Structure Chart:**
   It is created by the data flow diagram. Structure Chart specifies how DFS's processes are grouped into tasks and allocate to the CPU. The structured chart does not show the working and internal structure of the processes or modules and does not show the relationship between data or data-flows. Similar to other SASD tools, it is time and cost-independent and there is no error-checking technique associated with this tool.

The modules of a structured chart are arranged arbitrarily and any process from a DFD can be chosen as the central transform depending on the analysts' own perception. The structured chart is difficult to amend, verify, maintain, and check for completeness and consistency.

2. **Pseudo Code:**
   It is the actual implementation of the system. It is an informal way of programming that doesn't require any specific programming language or technology.

Advantages of Structured Analysis and Structured Design (SA/SD):

1. Clarity and Simplicity: The SA/SD method emphasizes breaking down complex systems into smaller, more

manageable components, which makes the system easier to understand and manage.

2. Better Communication: The SA/SD method provides a common language and framework for communicating the design of a system, which can improve communication between stakeholders and help ensure that the system meets their needs and expectations.
3. Improved maintainability: The SA/SD method provides a clear, organized structure for a system, which can make it easier to maintain and update the system over time.
4. Better Testability: The SA/SD method provides a clear definition of the inputs and outputs of a system, which makes it easier to test the system and ensure that it meets its requirements.

Disadvantages of Structured Analysis and Structured Design (SA/SD):

1. Time-Consuming: The SA/SD method can be time-consuming, especially for large and complex systems, as it requires a significant amount of documentation and analysis.
2. Inflexibility: Once a system has been designed using the SA/SD method, it can be difficult to make changes to the design, as the process is highly structured and documentation-intensive.

3. Limited Iteration: The SA/SD method is not well-suited for iterative development, as it is designed to be completed in a single pass.

Data Flow Diagrams

System analysts use process models (i.e. data flow diagrams, DFDs) to show information flow and processing in a system. The model usually starts with a context diagram showing the system bubble surrounded by the external environment identified by external entities. Data flows bring information to and from the system process.

A process can explode to a child diagram that presents its details using data stores, data flows and sub processes. The diagram leveling process allows complex systems to be easily partitioned into a stack of simple diagrams with rigorous balancing of information between levels. Information structures are defined in an associated data dictionary.

A process can explode to a child diagram in the same or a different DFD document. Each developer in a team can work independently on a DFD document that contains diagrams for a portion of the entire system.

Structure Charts

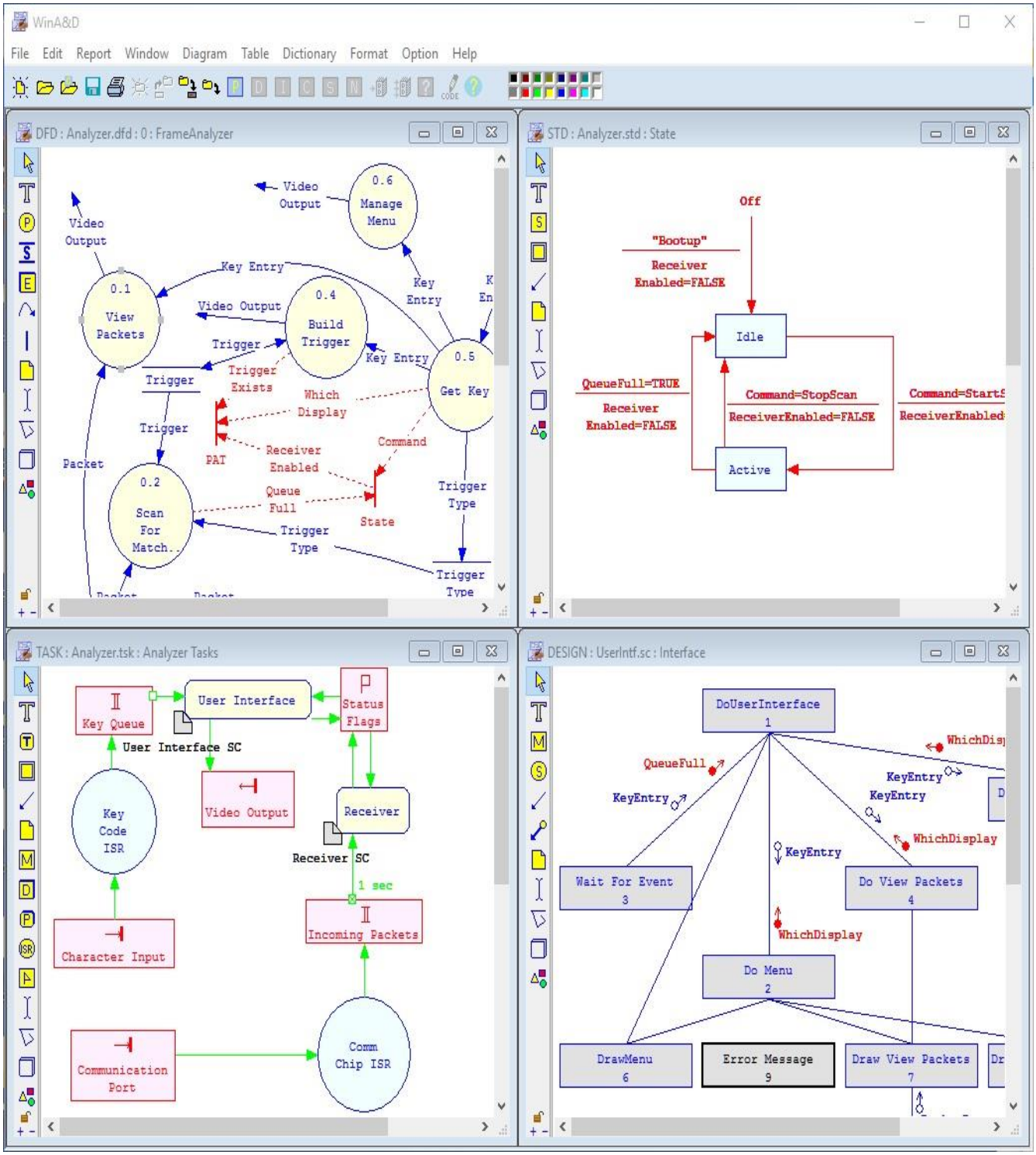
Structure charts show module structure and calling relationships. In a multi-threaded system, each task (thread of execution) is represented as a structure chart. Large structure charts are leveled into a stack of connected diagrams.

State Models

State models include diagrams and tables that show the significant states in a system, events that cause transitions between states and the actions that result.

Task Diagrams

Task diagrams show threads of execution and the real-time operating system services like queues, event flags and semaphores that connect them in a multi-tasking environment. Each task can be associated with its structure chart representation.



Object Oriented Analysis and Design

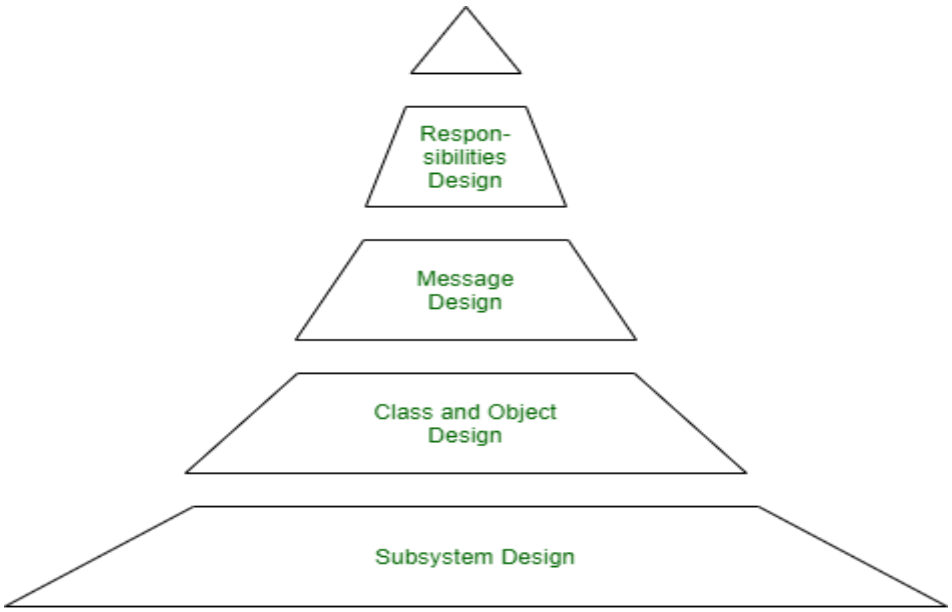
Object-Oriented Analysis (OOA) is the first technical activity performed as part of object-oriented software engineering. OOA introduces new concepts to investigate a

problem. It is based on a set of basic principles, which are as follows-

1. The information domain is modeled.
2. Behavior is represented.
3. The function is described.
4. Data, functional, and behavioral models are divided to uncover greater detail.
5. Early models represent the essence of the problem, while later ones provide implementation details.

The above notes principles form the foundation for the OOA approach.

Object-Oriented Design (OOD): An analysis model created using object-oriented analysis is transformed by object-oriented design into a design model that works as a plan for software creation. OOD results in a design having several different levels of modularity i.e., The major system components are partitioned into subsystems (a system-level “modular”), and data manipulation operations are encapsulated into objects (a modular form that is the building block of an OO system.). In addition, OOD must specify some data organization of attributes and a procedural description of each operation. Shows a design pyramid for object-oriented systems. It is having the following four layers.



The Object Oriented Design Pyramid

1. The Subsystem Layer : It represents the subsystem that enables software to achieve user requirements and implement technical frameworks that meet user needs.
2. The Class and Object Layer : It represents the class hierarchies that enable the system to develop using generalization and specialization. This layer also represents each object.
3. The Message Layer : It represents the design details that enable each object to communicate with its partners. It establishes internal and external interfaces for the system.
4. The Responsibilities Layer : It represents the data structure and algorithmic design for all the attributes and operations for each object.

The Object-Oriented design pyramid specifically emphasizes specific product or system design. Note, however, that another design layer exists, which forms the base on which the pyramid rests. It focuses on the core layer the design of the domain object, which plays an important role in building the infrastructure for the Object-Oriented system by providing support for human/computer interface activities, task management.

Some of the *terminologies* that are often encountered while studying Object-Oriented Concepts include:

1. Attributes: a collection of data values that describe a class.

2. Class: encapsulates the data and procedural abstractions required to describe the content and behavior of some real-world entity. In other words, A class is a generalized description that describes the collection of similar objects.

3. Objects: instances of a specific class. Objects inherit a class's attributes and operations.

4. Operations: also called methods and services, provide a representation of one of the behaviors of the class.

5. Subclass: specialization of the super class. A subclass can inherit both attributes and operations from a super class.

6. Superclass: also called a base class, is a generalization of a set of classes that are related to it.

Advantages of OOAD:

1. Improved modularity: OOAD encourages the creation of small, reusable objects that can be combined to create more complex systems, improving the modularity and maintainability of the software.
2. Better abstraction: OOAD provides a high-level, abstract representation of a software system, making it easier to understand and maintain.
3. Improved reuse: OOAD encourages the reuse of objects and object-oriented design patterns, reducing the amount of code that needs to be written and improving the quality and consistency of the software.
4. Improved communication: OOAD provides a common vocabulary and methodology for software developers, improving communication and collaboration within teams.

Disadvantages of OOAD:

1. Complexity: OOAD can add complexity to a software system, as objects and their relationships must be carefully modeled and managed.
2. Overhead: OOAD can result in additional overhead, as objects must be instantiated, managed, and interacted with, which can slow down the performance of the software.
3. Steep learning curve: OOAD can have a steep learning curve for new software developers, as it requires a strong understanding of OOP concepts and techniques.

User Interface Design

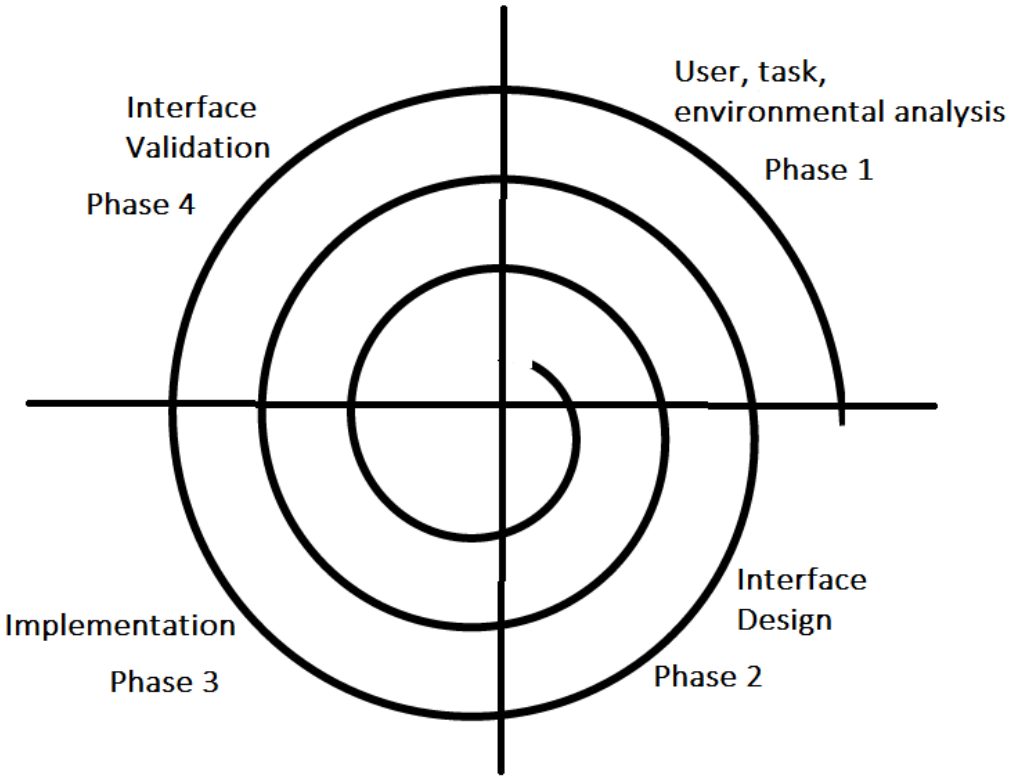
User interface is the front-end application view to which user interacts in order to use the software. The software becomes more popular if its user interface is:

- Attractive
- Simple to use
- Responsive in short time
- Clear to understand
- Consistent on all interface screens

There are two types of User Interface:

1. Command Line Interface: Command Line Interface provides a command prompt, where the user types the command and feeds to the system. The user needs to remember the syntax of the command and its use.
2. Graphical User Interface: Graphical User Interface provides the simple interactive interface to interact with the system. GUI can be a combination of both hardware and software. Using GUI, user interprets the software.

User Interface Design Process:



The analysis and design process of a user interface is iterative and can be represented by a spiral model. The analysis and design process of user interface consists of four framework activities.

1. User, task, environmental analysis, and modeling: Initially, the focus is based on the profile of users who will interact with the system, i.e. understanding, skill and knowledge, type of user, etc, based on the user's profile users are made into categories. From each category requirements are gathered. Based on the requirements developer understand how to develop the interface. Once all the requirements are gathered a detailed analysis is conducted. In the analysis part, the tasks that the user performs to establish the goals of the system are identified, described and elaborated. The analysis of

the user environment focuses on the physical work environment. Among the questions to be asked are:

- Where will the interface be located physically?
- Will the user be sitting, standing, or performing other tasks unrelated to the interface?
- Does the interface hardware accommodate space, light, or noise constraints?
- Are there special human factors considerations driven by environmental factors?

2. Interface Design: The goal of this phase is to define the set of interface objects and actions i.e. Control mechanisms that enable the user to perform desired tasks. Indicate how these control mechanisms affect the system. Specify the action sequence of tasks and subtasks, also called a user scenario. Indicate the state of the system when the user performs a particular task. Always follow the three golden rules stated by Theo Mandel. Design issues such as response time, command and action structure, error handling, and help facilities are considered as the design model is refined. This phase serves as the foundation for the implementation phase.
3. Interface construction and implementation: The implementation activity begins with the creation of prototype (model) that enables usage scenarios to be evaluated. As iterative design process continues a User Interface toolkit that allows the creation of windows, menus, device interaction, error messages,

commands, and many other elements of an interactive environment can be used for completing the construction of an interface.

4. Interface Validation: This phase focuses on testing the interface. The interface should be in such a way that it should be able to perform tasks correctly and it should be able to handle a variety of tasks. It should achieve all the user's requirements. It should be easy to use and easy to learn. Users should accept the interface as a useful one in their work.

Golden Rules:

The following are the golden rules stated by Theo Mandel that must be followed during the design of the interface. Place the user in control:

- Define the interaction modes in such a way that does not force the user into unnecessary or undesired actions: The user should be able to easily enter and exit the mode with little or no effort.
- Provide for flexible interaction: Different people will use different interaction mechanisms, some might use keyboard commands, some might use mouse, some might use touch screen, etc, Hence all interaction mechanisms should be provided.
- Allow user interaction to be interruptible and undoable: When a user is doing a sequence of actions the user must be able to interrupt the sequence to do some other work without losing the work that had

been done. The user should also be able to do undo operation.

- Streamline interaction as skill level advances and allow the interaction to be customized: Advanced or highly skilled user should be provided a chance to customize the interface as user wants which allows different interaction mechanisms so that user doesn't feel bored while using the same interaction mechanism.
- Hide technical internals from casual users: The user should not be aware of the internal technical details of the system. He should interact with the interface just to do his work.
- Design for direct interaction with objects that appear on screen: The user should be able to use the objects and manipulate the objects that are present on the screen to perform a necessary task. By this, the user feels easy to control over the screen.

Reduce the user's memory load:

- Reduce demand on short-term memory: When users are involved in some complex tasks the demand on short-term memory is significant. So the interface should be designed in such a way to reduce the remembering of previously done actions, given inputs and results.
- Establish meaningful defaults: Always initial set of defaults should be provided to the average user, if a

user needs to add some new features then he should be able to add the required features.

- Define shortcuts that are intuitive: Mnemonics should be used by the user. Mnemonics means the keyboard shortcuts to do some action on the screen.
- The visual layout of the interface should be based on a real-world metaphor: Anything you represent on a screen if it is a metaphor for real-world entity then users would easily understand.
- Disclose information in a progressive fashion: The interface should be organized hierarchically i.e. on the main screen the information about the task, an object or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

Make the interface consistent:

- Allow the user to put the current task into a meaningful context: Many interfaces have dozens of screens. So it is important to provide indicators consistently so that the user know about the doing work. The user should also know from which page has navigated to the current page and from the current page where can navigate.
- Maintain consistency across a family of applications: The development of some set of applications all should follow and implement the same design, rules so that consistency is maintained among applications.

- If past interactive models have created user expectations do not make changes unless there is a compelling reason.

Command Language

A command language is a type of interpreted language using a command line structure. Command languages are typically not compiled but are interpreted on the fly. A prominent example is the MS-DOS computer system that controlled earlier personal computers where a command line structure was used to generate user-driven processes.

Command languages have many uses in computer science and the administration of operating systems. They often serve to provide immediate responses to end-user events. For example, a command language for batch processing has specific commands that help to organize and manipulate files. Command languages can be clear-cut ways to implement a set of instructions that might not need the power of a fully compiled, object-oriented language for them to function well.

Menu System

This is a menu system designed as the top-level interface to a general purpose operating system and an extensive variety of layered software. The system tested was a base system without customization. This product was released in the early t980's. Menus were used to access options, and function keys were used in the applications. On-line help was available.

Iconic Systems 1 and 2

These systems are both commercially available, iconic, top-level interfaces that provide access to a variety of functions including file manipulation, text editing, graphics, and other applications. Both make extensive use of icons to represent files, functions, and states. Both also make extensive use of a mouse as a pointing and selection device. Both systems were released in the early 1980's. They differ in many details of presentation, input syntax, file structure, treatment of help, and hardware