NARAYANA ENGINEERING COLLEGE
NELLORE, AP (AUTONOMOUS)

SOFTWARE ENGINEERING

II B.Tech, CSE – C

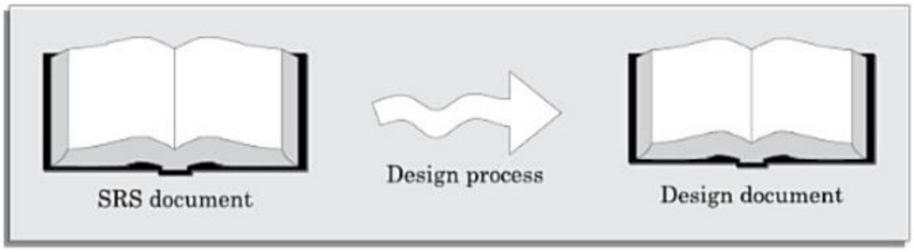
Dr. Sunil Kumar Battarusetty

MODULE-3

Software Design: Overview of the design process, How to characterize a good software design? Layered arrangement of modules, Cohesion and Coupling, approaches to software design.
Agility: Agility and the Cost of Change, Agile Process, Extreme Programming (XP), Other Agile Process Models, Tool Set for the Agile Process (Text Book 2)
Function-Oriented Software Design: Overview of SA/SD methodology, Structured analysis, Developing the DFD model of a system, Structured design, Detailed design, and Design Review.
User Interface Design: Characteristics of a good user interface, Basic concepts, Types of user interfaces, Fundamentals of component-based GUI development, and user interface design methodology.

Software Design

● The activities carried out during the design phase (called as design process) transform the SRS document into the design document.
● The design document produced at the end of the design phase should be implementable using a programming language in the subsequent (coding) phase.



The design process

3.1 Overview of the Design Process

3.1.1 Outcome of Design Process:

The following items are designed and documented during the design phase.
● **Different modules required:** The different modules in the solution should be clearly identified. Each module is a collection of functions and the data shared by the functions of the module. Each module should accomplish some well-defined task out of the overall responsibility of the software. Each module should be named according to the task it performs.

- **Control relationships among modules**: A control relationship between two modules essentially arises due to function calls across the two modules.
- **Interfaces among different modules**: The interfaces between two modules identifies the exact data items that are exchanged between the two modules.
- **Data structures of the individual modules:** Suitable data structures for storing and managing the data of a module need to be properly designed and documented.
- **Algorithms required to implement the individual modules:** The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due considerations given to the accuracy of the results, space and time complexities.

3.1.2 Classification of Design Activities:

- A good software design requires iterating over a series of steps called the design activities.
- We can broadly classify them into two important stages.
    - **Preliminary (or high-level) design:**
        - Through high-level design, a problem is decomposed into a set of modules.
        - The control relationships among the modules are identified, and also the interfaces among various modules are identified.
        - The outcome of high-level design is called the program structure or the software architecture.
        - High-level design is a crucial step in the overall design of a software.
    - **Detailed design:**
        - Once the high-level design is complete, detailed design is undertaken.
        - During detailed design each module is examined carefully to design its data structures and the algorithms.
        - The outcome of the detailed design stage is usually documented in the form of a module specification (MSPEC) document.

3.2 How to characterize a good Design?

- There is no general agreement among software engineers and researchers on the exact criteria to use for judging a design even for a specific category of application.
- However, most researchers and software engineers agree on a few desirable characteristics.
    - **Correctness:** A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.
    - **Understandability:** A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.
    - **Efficiency:** A good design solution should adequately address resource, time, and cost optimization issues.
    - **Maintainability:** A good design should be easy to change. This is an important requirement, since change requests usually keep coming from the customer even after product release.

3.2.1 Understandability of a Design: A Major Concern

- Understandability of a design solution is possibly the most important issue to be considered while judging the goodness of a design.

● Unless a design solution is easily understandable, it could lead to an implementation having a large number of defects and at the same time tremendously pushing up the development costs.
● Therefore, a good design solution should be simple and easily understandable.
● Understandability of a design solution can be enhanced through clever applications of the principles of abstraction and decomposition.
● A design solution should have the following characteristics to be easily understandable:
  ○ It should assign consistent and meaningful names to various design components.
  ○ It should make use of the principles of decomposition and abstraction in good measures to simplify the design.
● A design solution is understandable, if it is modular and the modules are arranged in distinct layers.

Modularity
● A modular design is an effective decomposition of a problem.
● A modular design, in simple words, implies that the problem has been decomposed into a set of modules that have only limited interactions with each other.
● Decomposition of a problem into modules facilitates taking advantage of the divide and conquer principle.
● There are no quantitative metrics available yet to directly measure the modularity of a design.
● However, we can quantitatively characterize the modularity of a design solution based on the cohesion and coupling.
● A design solution is said to be highly modular, if the different modules in the solution have high cohesion and their inter-module couplings are low.

Layered design
● A layered design is one in which when the call relations among different modules are represented graphically, it would result in a tree-like diagram with clear layering.
● In a layered design solution, the modules are arranged in a hierarchy of layers.
● A module can only invoke functions of the modules in the layer immediately below it.
● A layered design can make the design solution easily understandable, since to understand the working of a module, one would at best have to understand how the immediately lower layer modules work without having to worry about the functioning of the upper layer modules.

3.3 COHESION AND COUPLING
● Effective problem decomposition is an important characteristic of a good design.
● Good module decomposition is indicated through high cohesion of the individual modules and low coupling of the modules with each other.
● Cohesion is a measure of the functional strength of a module, whereas the coupling between two modules is a measure of the degree of interaction (or interdependence) between the two modules.

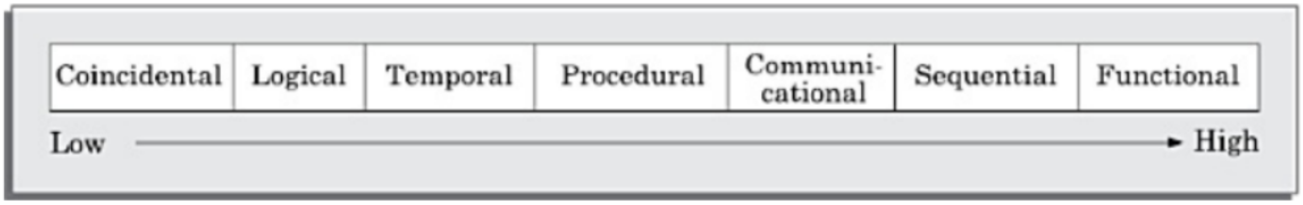
Cohesion
● When the functions of the module co-operate with each other for performing a single objective, then the module has good cohesion.

● If the functions of the module do very different things and do not cooperate with each other to perform a single piece of work, then the module has very poor cohesion.
● A module that is highly cohesive and also has low coupling with other modules is said to be functionally independent of the other modules.
● Functional independence is a key to any good design primarily due to the following advantages:

- ○ Error isolation
- ○ Scope of reuse
- ○ Understandability

3.3.1 Classifications of cohesiveness:

● Cohesiveness of a module is the degree to which the different functions of the module co-operate to work towards a single objective.
● The different modules of a design can possess different degrees of freedom.
● Cohesiveness increases from coincidental to functional cohesion



● **Coincidental cohesion:** A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all.
● **Logical cohesion**: A module is said to be logically cohesive, if all elements of the module perform similar operations, such as error handling, data input, data output, etc.
● **Temporal cohesion**: When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion.
● **Procedural cohesion**: A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data.
● **Communicational cohesion**: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure.
● **Sequential cohesion**: A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence.
● **Functional cohesion**: A module is said to possess functional cohesion, if different functions of the module co-operate to complete a single task.

Coupling:

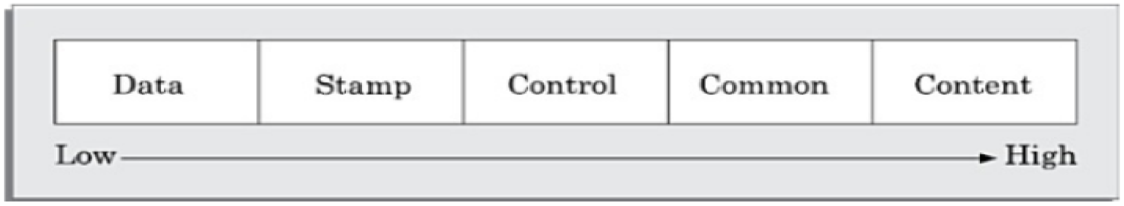
● Intuitively, we can think of coupling as follows. Two modules are said to be highly coupled, if either of the following two situations arise:

- ○ If the function calls between two modules involve passing large chunks of shared data, the modules are tightly coupled.
- ○ If the interactions occur through some shared data, then also we say that they are highly coupled.

● If two modules either do not interact or at best interact by passing no data or only a few primitive data items, they are said to have low coupling.

3.3.2 Classification of Coupling:

- The coupling between two modules indicates the degree of interdependence between them.
- If two modules interchange large amounts of data, then they are highly interdependent or coupled.
- The degree of coupling between two modules depends on their interface complexity.
- The interface complexity is determined based on the number of parameters and the complexity of the parameters that are interchanged while one module invokes the functions of the other module.
- The degree of coupling increases from data coupling to content coupling.



- **Data coupling**: Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc.
- **Stamp coupling**: Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.
- **Control coupling**: Control coupling exists between two modules, if data from one module is used to direct the order of instruction execution in another.
- **Common coupling**: Two modules are common coupled, if they share some global data items.
- **Content coupling**: Content coupling exists between two modules, if they share code.

3.4 APPROACHES TO SOFTWARE DESIGN

- There are two fundamentally different approaches to software design that are in use today— function-oriented design, and object-oriented design.
- Though these two design approaches are radically different, they are complementary rather than competing techniques.
- The object oriented approach is a relatively newer technology and is still evolving.
- On the other hand, function-oriented designing is a mature technology and has a large following.

3.4.1 Function-oriented Design:

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

- Top-down decomposition:
    - A system, to start with, is viewed as a black box that provides certain services (also known as high-level functions) to the users of the system.
    - In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed functions.
    - For example, consider a function create-new-library member.
    - This high-level function may be refined into the following sub functions:
        - Assign-membership-number
        - Create-member-record
        - Print-bill

Centralized system state:

- The system state can be defined as the values of certain data items that determine the response of the system to a user action or external event.
- The system state is centralized and shared among different functions.
- For example, in the library management system, several functions such as the following share data such as member-records for reference and updation
    - Create-new-member
    - Delete-member
    - update-member-record

3.4.2 Object-oriented Design:

- In the object-oriented design (OOD) approach, a system is viewed as being made up of a collection of objects (i.e. entities).
- Each object is associated with a set of functions that are called its methods.
- Each object contains its own data and is responsible for managing it.
- The data internal to an object cannot be accessed directly by other objects and only through invocation of the methods of the object.
- The object-oriented design paradigm makes extensive use of the principles of abstraction and decomposition.
- Objects decompose a system into functionally independent modules. Objects can also be considered as instances of abstract data types (ADTs).
- There are three important concepts associated with an ADT.
    - **Data abstraction**: The principle of data abstraction implies that how data is exactly stored is abstracted away.
    - **Data structure**: A data structure is constructed from a collection of primitive data items.
    - **Data type**: A type is a programming language terminology that refers to anything that can be instantiated.
- In object-orientation, classes are ADTs.
- There are three main advantages of using ADTs in programs:
    - The data of objects are encapsulated within the methods. The encapsulation principle is also known as data hiding.
    - An ADT-based design displays high cohesion and low coupling.
    - Since the principle of abstraction is used, it makes the design solution easily understandable and helps to manage complexity.

Agility: Agility and the Cost of Change, Agile Process, Extreme Programming (XP), Other Agile Process Models, Tool Set for the Agile Process.

AGILITY

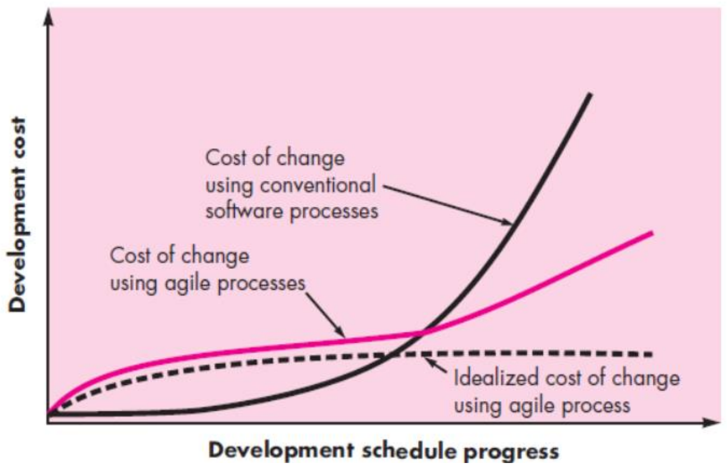
- Agile software engineering has a set of development guidelines. The mechanism encourages customer satisfaction and early incremental delivery of software.
- Software engineers and other project stakeholders (managers, customers, end users) work together on an agile team
- Agile software engineering represents a reasonable alternative to conventional software engineering.

Agility:

- Agility means effective (rapid and adaptive) response to change, effective communication among all stakeholder.
- Drawing the customer onto team and organizing a team so that it is in control of work performed. The Agile process is light-weight methods and people-based rather than plan-based methods.
- The agile process forces the development team to focus on software itself rather than design and documentation.
- The agile process believes in iterative method.
- The aim of agile process is to deliver the working model of software quickly to the customer For example: Extreme programming is the best known of agile process.

AGILITY AND THE COST OF CHANGE

The cost of change increases as the project progresses.(Figure, solid black curve). It is relatively easy to accommodate a change when a software team is gathering requirements (early in a project).



The team is in the middle of validation testing (something that occurs relatively late in the project), and stakeholder is requesting a major functional change. The change requires a modification to the architectural design of the software, the design and construction of three new components, modifications to another five components, the design of new tests, and so on. Costs and time increases quickly.

A well-designed agile process "flattens" the cost of change curve (Figure, shaded, solid curve), allowing a software team to accommodate changes late in a software project without cost and time impact. the agile process encompasses incremental delivery.

AGILE PROCESS

Any agile software process is characterized in a manner that addresses a number of key assumptions about the majority of software projects:

1. It is difficult to predict(estimate for future change) in advance which software requirements will persist and which will change.
2. For many types of software, design and construction are performed simultaneously. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not expected. (from a planning point of view)

Agility Principles

12 agility principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development.
3. Deliver working software frequently, from a couple of weeks to a couple of Months.
4. stake holders and software engineers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development.
9. Continuous attention to technical excellence and good design enhances (increases) agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self– organizing teams.
12. At regular intervals, the team reflects on how to become more effective and then adjusts its behavior accordingly.

Not every agile process model applies these 12 principles with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more of the principles.

The Politics of Agile Development

- There is debate about the benefits and applicability of agile software development as opposed to more conventional software engineering processes(produces documents rather than working product).
- Even within the agile, there are many proposed process models each with a different approach to the agility.

Human Factors

- Agile software development take the importance of "people factors". Number of different talents must exist among the people on an agile team and the team itself:
- **Competence**:"competence" encompasses talent, specific software-related skills, and overall knowledge of the process .
- **Common focus:** Members of the agile team may perform different tasks and all should be focused on one goal—to deliver a working software increment to the customer within the time promised.
- **Collaboration:** Team members must collaborate with one another and all other Stakeholders to complete the their task.
- **Decision-making ability**: Any good software team (including agile teams) must be allowed the freedom to control its own destiny.
- **Fuzzy problem-solving ability:** The agile team will continually have to deal with ambiguities(confusions or doubts).
- **Mutual trust and respect:** The agile team exhibits the trust and respect.
- **Self-organization:**

  (1) the agile team organizes itself for the work to be done

  (2) the team organizes the process to best accommodate its local environment

  (3) the team organizes the work schedule to best achieve delivery of the software increment.

EXTREME PROGRAMMING

Extreme Programming (XP), the most widely used approach to agile software development. XP proposed by ***kent beck*** during the late 1980's.

XP Values

Beck defines a set of five values —communication, simplicity, feedback, courage, and respect. Each of these values is used in XP activities, actions, and tasks.

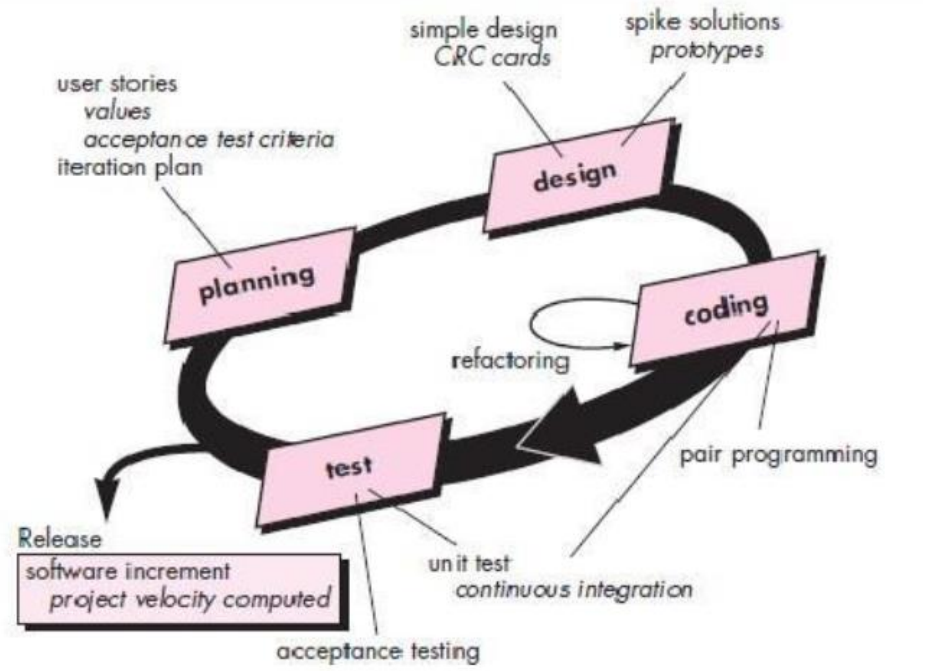
- Effective **communication** between software engineers and other stakeholders.
- To achieve **simplicity,** XP restricts developers to design only for immediate needs, rather than future needs.
- **Feedback** is derived from three sources: the implemented software itself, the customer, and other software team members.
- **courage(discipline)** An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically.
- the agile team inculcates **respect** among its members, between other stakeholders and team members.

The XP(Extreme Programming) Process

XP Process have four framework activities: planning, design, coding, and testing.

Planning. The planning activity begins with *listening*—a requirements gathering activity.

- ➢ Listening leads to the creation of a set of "stories" (also called user stories) that describe required output, features, and functionality for software to be built.
- ➢ Each story is written by the customer and is placed on an index card. The customer assigns a value (i.e., a priority) to the story based on the overall business value of the feature or function.
- ➢ Members of the XP team then assess each story and assign a cost measured in development weeks—to it.
- ➢ If the story is estimated to require more than three development weeks, the story into smaller stories and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time.
- ➢ The stories with highest value will be moved up in the schedule and implemented first.



Design: XP design follows the KIS (keep it simple) principle.

- ➢ If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a *spike* solution.
- ➢ XP encourages *refactoring*—a construction technique that is also a method for design optimization.
- ➢ Refactoring is the process of changing a software system in a way that it does not change the external behavior of the code and improves the internal structure.

Coding: design work is done, the team does not move to code, develops a series of
unit tests for each of the stories that is to be included in the current release (software increment).

- ➢ Once the unit test has been created, the developer is better able to focus on what must be implemented to pass the test.
- ➢ Once the code is complete, it can be unit-tested immediately, and providing feedback to the developers.

- A key concept during the coding activity is pair programming. i.e.., two people work together at one computer workstation to create code for a story.
- As pair programmers complete their work, the code they develop is integrated with the work of others.

Testing:

- As the individual unit tests are organized into a "universal testing suite" integration and validation testing of the system can occur on a daily basis.
- XP acceptance tests, also called customer tests, are specified by the customer and focus on overall system features and functionality.
- Acceptance tests are derived from user stories that have been implemented as part of a software release.

Industrial XP

Joshua Kerievsky describes *Industrial Extreme Programming* (IXP)
IXP has six new practices that are designed to help XP process works successfully for projects within a large organization.

Readiness assessment. the organization should conduct a *readiness* assessment.
(1) Development environment exists to support IXP(Industrial Extreme Programming).
(2) the team will be populated by the proper set of stakeholders.
(3) the organization has a distinct quality program and supports continuous improvement.
(4) the organizational culture will support the new values of an agile Team.

Project community.

- Classic XP suggests that the right people be used to populate the agile team to ensure success.
- The people on the team must be well-trained, adaptable and skilled.
- A community may have a team members and customers who are central to the success of the project.

Project chartering.:

- Chartering examines the context of the project to determine how it complements, extends, or replaces existing systems or processes.

Test-driven management.

- Test-driven management establishes a series of measurable "destinations" and then defines mechanisms for determining whether or not these destinations have been reached.

Retrospectives.

- An IXP team conducts a specialized technical reviews after a software increment is delivered. Called a retrospective.
- The review examines "issues, events, and lessons-learned" across a software increment and/or the entire software release.

Continuous learning.

- Learning is a vital part of continuous process improvement, members of the XP team are encouraged to learn new methods and techniques that can lead to a higher quality product.

The XP Debate

- **Requirements volatility**. The customer is an active member of the XP team, changes to requirements are requested informally. As a consequence, the scope of the project can change and earlier work may have to be modified to accommodate current needs.
    - **Conflicting customer needs:** Many projects have multiple customers, each with his own set of needs.
    - **Requirements are expressed informally:** User stories and acceptance tests are the only explicit manifestation of requirements in XP. Specification is often needed to remove inconsistencies, and errors before the system is built.
    - **Lack of formal design**: when complex systems are built, design must have the overall structure of the software then it will exhibit quality.

Other Agile Process Models

- The most widely used of all agile process model is Extreme Programming (XP). But many other agile process models have been proposed and are in use across the industry.

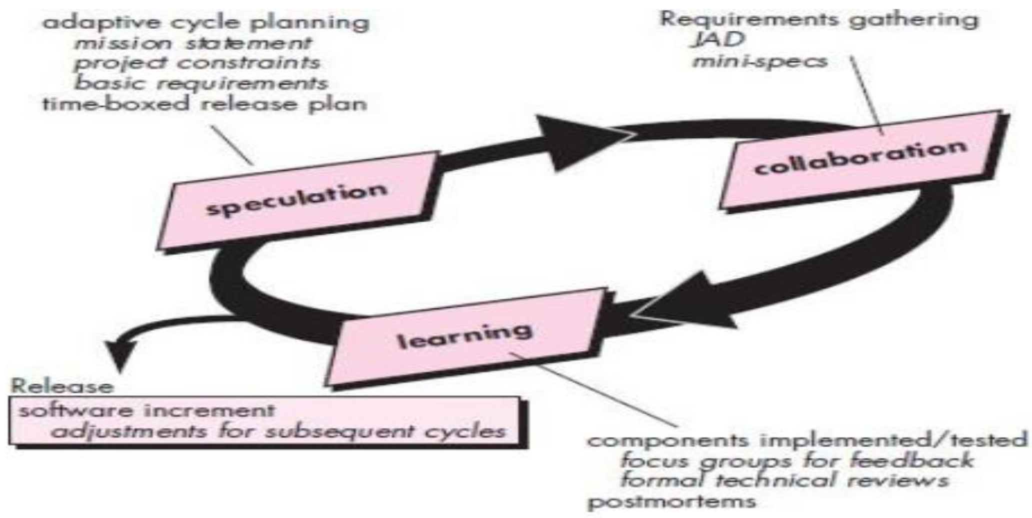
Among the most common are:

- Adaptive Software Development (ASD)
- Scrum
- Dynamic Systems Development Method (DSDM)
- Crystal
- Feature Drive Development (FDD)
- Lean Software Development (LSD)
- Agile Modeling (AM)
- Agile Unified Process (AUP)

Adaptive Software Development (ASD): Adaptive Software Development (ASD) has been proposed by Jim High smith as a technique for building complex systems. ASD focus on human collaboration and team self- organization.

ASD "life cycle" (Figure) has three phases:- speculation, collaboration, and learning.

Speculation, the project is initiated and adaptive cycle planning is conducted. Adaptive cycle planning uses project initiation information—the customer's statement, project constraints (e.g., delivery dates or user descriptions), and basic requirements—to define the set of release cycles (software increments) that will be required for the project.

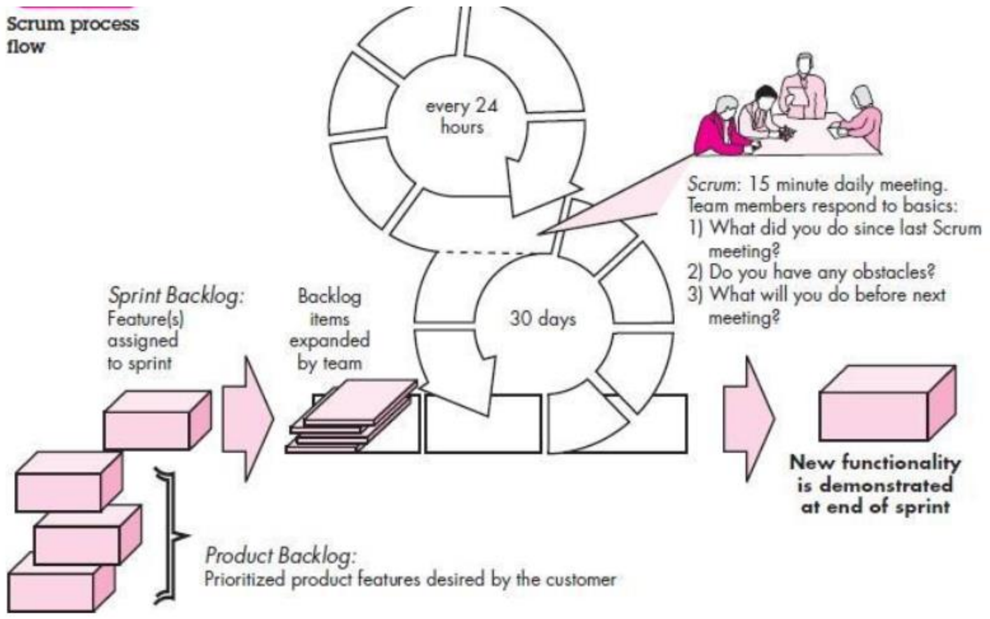


- Motivated people use **collaboration**
- People working together must trust one another to (1) without criticize, (2) work as hard as or harder than they do, (3) have the skill set to contribute to the work and (4) communicate problems in a way that leads to effective action.
- **learning** will help them to improve their level of real understanding.

Scrum

- Scrum is an agile software development method that was coined by Jeff Sutherland and his development team in the early 1990's.
- Scrum has the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, actions and work tasks occur within a process called a *sprint*.
- scrum defines a set of development actions:

Backlog—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time (this is how changes are introduced). The product manager assesses the backlog and updates priorities as required.
Sprints—consist of work units that are required to achieve a requirement defined in the backlog .


Scrum process flow
every 24 hours
Scrum: 15 minute daily meeting.
Team members respond to basics:
1) What did you do since last Scrum meeting?
2) Do you have any obstacles?
3) What will you do before next meeting?
Sprint Backlog: Feature(s) assigned to sprint
Backlog items expanded by team
30 days
New functionality is demonstrated at end of sprint
Product Backlog: Prioritized product features desired by the customer

Scrum meetings—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members.

- What did you do since the last team meeting?
- What obstacles are you encountering?
- What do you plan to accomplish by the next team meeting?

A team leader, called a *Scrum master*, leads the meeting and assesses the responses from each person.
Demos—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer.

Dynamic Systems Development Method (DSDM)

- The Dynamic Systems Development Method (DSDM) is an agile software development approach.

- The DSDM—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete (100 percent) application.
- DSDM is an iterative software process in which each iteration follows the 80 percent rule. That is, only enough work is required for each increment to facilitate movement to the next increment.
- The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

The activities of DSDM are:

Feasibility study—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.
Business study—establishes the functional and information requirements that will allow the application to provide business value.
Functional model iteration—produces a set of incremental prototypes that demonstrate functionality for the customer.

- The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.

Design and build iteration—prototypes built during functional model iteration to ensure that each has been engineered in a manner that it will provide operational business value for end users.
Implementation—places the latest software increment (an "operationalized" prototype) into the operational environment. It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place.

Crystal

- Alistair Cockburn and Jim High smith created the Crystal family of agile methods .
- Cockburn characterizes as "a resource limited, cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game".
- Cockburn and High smith have defined a set of methodologies, each with core elements that are common to all, and roles, process patterns, work products, and practice that are unique to each.
- The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects.

Feature Driven Development (FDD)
Feature Driven Development (FDD) was originally coined by Peter Coad and his colleagues as a process model for object-oriented software engineering.

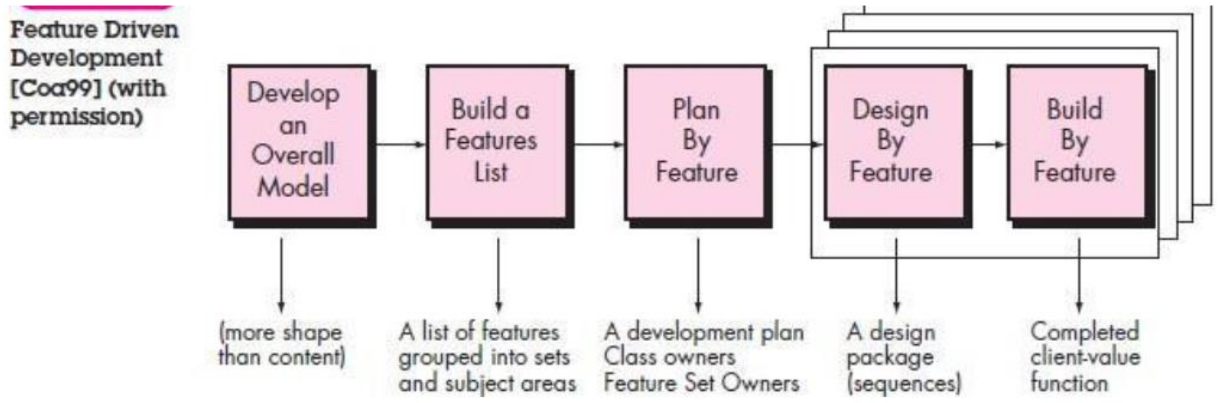
A feature "is a client-valued function that can be implemented in two weeks or less".

The definition of features provides the following benefits:

- features are small blocks of deliverable functionality, users can describe them more easily.
- Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks.
- Because features are small, their design and code representations are easier.

The following template for defining a feature:
<action> the <result> <by for of to> a(n) <object>

Where
<object> is “a person, place, or thing.”
Examples of features for an **e-commerce application** might be:
1) Add the product to shopping cart
2) Store the shipping-information for the customer


Feature Driven Development [Coa99] (with permission)
Develop an Overall Model
Build a Features List
Plan By Feature
Design By Feature
Build By Feature
(more shape than content)
A list of features grouped into sets and subject areas
A development plan Class owners Feature Set Owners
A design package (sequences)
Completed client-value function

- A feature set groups related features into business-related categories and is defined as:

<action><-ing> a(n) <object>

For example: Making a product sale is a feature set that would encompass the features noted earlier and others.

- The FDD approach defines five “collaborating” framework activities as shown in Figure.
- It is essential for developers, their managers, and other stakeholders to understand project status.
- For that FDD defines six milestones during the design and implementation of a feature: “design walkthrough, design, design inspection, code, code inspection, promote to build”.
- For that FDD defines six milestones during the design and implementation of a feature: “design walkthrough, design, design inspection, code, code inspection, promote to build”.

Lean Software Development (LSD)

- Lean Software Development (LSD) has adapted the principles of lean manufacturing to the world of software engineering.
- LSD process can be summarized as eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people, and optimize the whole.

  For example,

  eliminate waste within the context of an agile software project as
  (1)adding no extraneous features or functions
  (2) assessing the cost and schedule impact of any newly requested requirement,
  (3) removing any superfluous process steps,
  (4) establishing mechanisms to improve the way team members find information,
  (5) ensuring the testing finds as many errors as possible

Agile Modeling (AM)
Agile Modeling(AM) suggests a wide array of “core” and “supplementary” modeling principles.

Model with a purpose. A developer who uses AM should have a specific goal (e.g., to communicate information to the customer or to help better understand aspect of the software) in mind before creating the model.
Use multiple models. There are many different models and notations that can be used to describe software.

- Each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.

Travel light. As software engineering work proceeds, keep only those models that will provide long-term value.
Content is more important than representation. Modeling should impart information to its intended audience.
Know the models and the tools you use to create them. Understand the strengths and weaknesses of each model and the tools that are used to create it.
Adapt locally. The modeling approach should be adapted to the needs of the agile team.

Agile Unified Process (AUP)

The *Agile Unified Process* (AUP) adopts a “serial in the large” and “iterative in the small” philosophy for building computer-based systems.

Each AUP iteration addresses the following activities:
• **Modeling**. UML representations of the business and problem domains are created.
• **Implementation**. Models are translated into source code.
• **Testing**. Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.
• **Deployment.** focuses on the delivery of a software increment and the acquisition of feedback from end users.
• **Configuration and project management**. Configuration management addresses change management, risk management, and the control of any persistent work products that are produced by the team.
Project management tracks and controls the progress of the team and coordinates team activities.
• **Environment management**. Environment management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

A tool set for agile development

- Automated software tools (e.g., design tools) should be viewed as a minor supplement to the team’s activities, and not to the success of the team.
- Collaborative and communication “tools” are generally low technology and incorporate any mechanism(whiteboards, poster sheets) that provides information and coordination among agile developers.
- other agile tools are used to optimize the environment in which the agile team works, improve the team culture by social interactions, physical devices (e.g., electronic whiteboards), and process enhancement (e.g., pair programming or time-boxing)”.

Function-Oriented Software Design: Overview of SA/SD Methodology, Structured Analysis, Structured Design, Detailed Design, Design Review.

Introduction to Function-Oriented Software Design

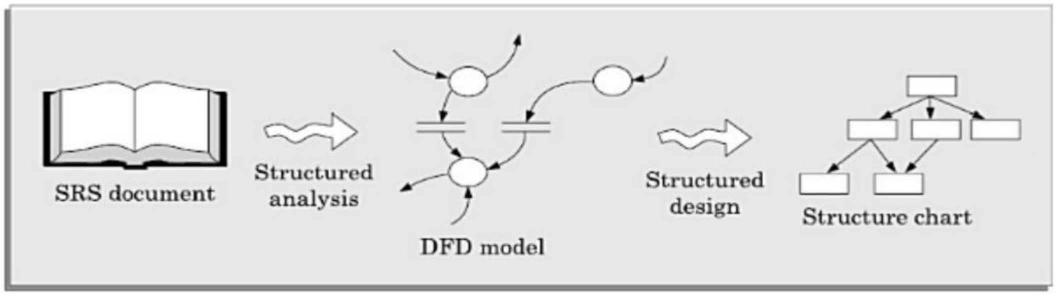
- Function-oriented design techniques were proposed nearly four decades ago.
- still very popular and are currently being used in many software development organizations. These techniques, to start with, view a system as a black-box that provides a set of services to the users of the software.
- These services are also known as the high-level functions supported by the software. During the design process, these high-level functions are successively decomposed into more detailed functions
- The term top-down decomposition i s often used to denote the successive decomposition of a set of high-level functions into more detailed functions.
- different identified functions are mapped to modules and a module structure is created.
- We shall discuss a methodology that has the essential features of several important function-oriented design methodologies.
- The design technique discussed here is called structured analysis/structured design (SA/SD) methodology.
- The SA/SD technique can b e used to perform the high-level design of a software.

Overview of SA/SD methodology.

As the name itself implies, SA/SD methodology involves carrying out two distinct activities:

- Structured analysis (SA)
- Structured design (SD).

The roles of structured analysis (SA) and structured design (SD) have been shown schematically in below figure.



- The ***structured analysis*** activity transforms the SRS document into a graphic model called the DFD model. During structured analysis, functional decomposition of the system is achieved. The purpose of structured analysis is to capture the detailed structure of the system as perceived by the user.
- During ***structured design***, all functions identified during structured analysis are mapped to a module structure. This module structure is also called the high level design or the software architecture for the given problem. This is represented using a structure chart. The purpose of structured design is to define the structure of the solution that is suitable for implementation.
- The high-level design stage is normally followed by a detailed design stage.

Structured Analysis

During structured analysis, the major processing tasks (high-level functions) of the system are analyzed, and t h e data flow among these processing tasks are represented graphically. The structured analysis technique is based on the following underlying principles:

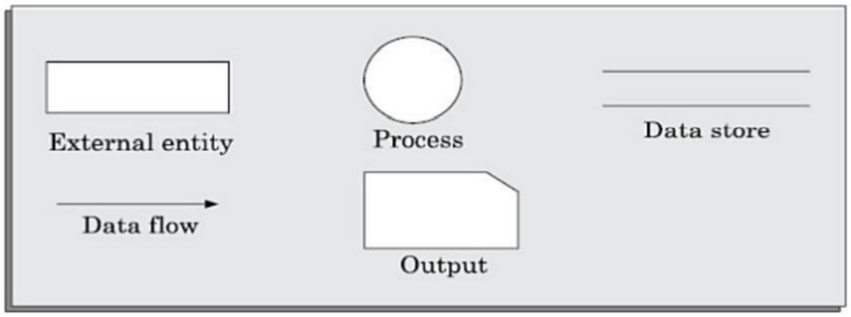
- Top-down decomposition approach.
- Application of divide and conquer principle.
- Through this each high level function is independently decomposed into detailed functions. Graphical representation of the analysis results using data flow diagrams (DFDs).

What is DFD?

- A DFD is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among those functions.
- Though extremely simple, it is a very powerful tool to tackle the complexity of industry standard problems.
- A DFD model only represents the data flow aspects and does not show the sequence of execution of the different functions and the conditions based on which a function may or may not be executed.
- In the DFD terminology, each function is called a process or a bubble. each function as a processing station (or process) that consumes some input data and produces some output data.
- DFD is an elegant modeling technique not only to represent the results of structured analysis but also useful for several other applications.
- Starting with a set of high-level functions that a system performs, a DFD model represents the subfunctions performed by the functions using a hierarchy of diagrams.

Primitive symbols used for constructing DFDs

There are essentially five different types of symbols used for constructing DFDs.



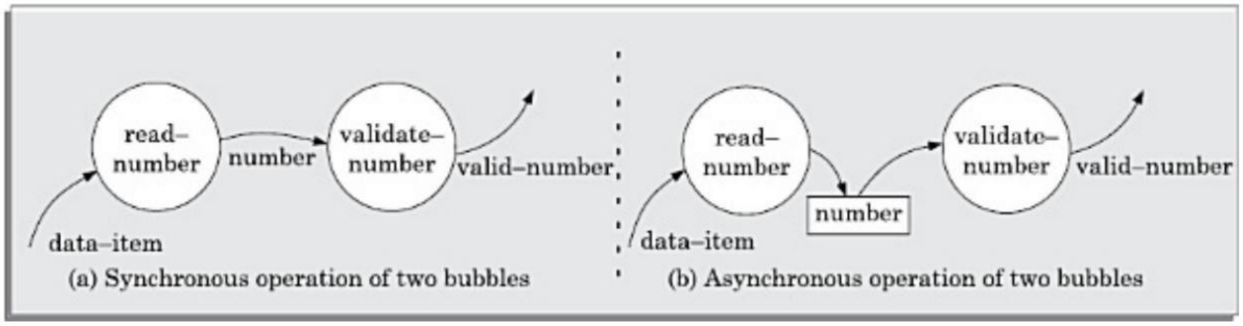
- **Function symbol:** A function is represented using a circle. This symbol is called a process or a bubble. Bubbles are annotated with the names of the corresponding functions
- **External entity symbol:** represented by a rectangle. The external entities are essentially those physical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system.
- **Data flow symbol:** A directed arc (or an arrow) is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow.
- **Data store symbol:** A data store is represented using two parallel lines. It represents a logical file. That is, a data store symbol can represent either a data structure or a

physical file on disk. Connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store.

- **Output symbol:** The output symbol i s as shown in Figure 6.2. The output symbol is used when a hard copy is produced.

Important concepts associated with constructing DFD models Synchronous and asynchronous operations

- If two bubbles are directly connected by a data flow arrow, then they are synchronous. This means that they operate at the same speed.
- If two bubbles are connected through a data store, as in Figure (b) then the speed of operation of the bubbles are independent.



(a) Synchronous operation of two bubbles (b) Asynchronous operation of two bubbles

Data dictionary

- Every DFD model of a system must be accompanied by a data dictionary. A data dictionary lists all data items that appear in a DFD model.
- A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items.
- It includes all data flows and the contents of all data stores appearing on all the DFDs in a DFD model.
- For the smallest units of data items, the data dictionary simply lists their name and their type.
- Composite data items are expressed in terms of the component data items using certain operators.

- The dictionary plays a very important role in any software development process, especially for the following reasons:
    - A data dictionary provides a standard terminology for all relevant data for use by the developers working in a project.
    - The data dictionary helps the developers to determine the definition of different data structures in terms of their component elements while implementing the design.
    - The data dictionary helps to perform impact analysis. That is, it is possible to determine the effect of some data on various processing activities and vice versa.
- For large systems, the data dictionary can become extremely complex and voluminous.
- Computer-aided software engineering (CASE) tools come handy to overcome this problem.
- Most CASE tools usually capture the data items appearing in a DFD as the DFD is drawn, and automatically generate the data dictionary.

Developing the DFD model of a system

- The DFD model of a problem consists of many DFDs and a single data dictionary. The DFD model of a system i s constructed by using a hierarchy of DFDs.
- The top level DFD is called the level 0 DFD or the context diagram.
    - This is the most abstract (simplest) representation of the system (highest level). It is the easiest to draw and understand.
- At each successive lower level DFDs, more and more details are gradually introduced.
- To develop a higher-level DFD model, processes are decomposed into their subprocesses and the data flow among these subprocesses are identified.
- Level 0 and Level 1 consist of only one DFD each. Level 2 may contain up to 7 separate DFDs, and level 3 up to 49 DFDs, and so on.
- However, there is only a single data dictionary for the entire DFD model.

Context Diagram/Level 0 DFD:

- The context diagram is the most abstract (highest level) data flow representation of a system. It represents the entire system as a single bubble.
- The bubble in the context diagram is annotated with the name of the software system being developed.
- The context diagram establishes the context in which the system operates; that is, who are the users, what data do they input to the system, and what data they received by the system.
- The data input to the system and the data output from the system are represented as incoming and outgoing arrows.

Level 1 DFD:

- The level 1 DFD usually contains three to seven bubbles.
- The system is represented as performing three to seven important functions.
- To develop the level 1 DFD, examine the high-level functional requirements in the SRS document.
- If there are three to seven high level functional requirements, then each of these can be directly represented as a bubble in the level 1 DFD.
- Next, examine the input data to these functions and the data output by these functions as documented in the SRS document and represent them appropriately in the diagram.

Decomposition:

- Each bubble in the DFD represents a function performed by the system.
- The bubbles are decomposed into subfunctions at the successive levels of the DFD model. Decomposition of a bubble is also known as factoring o r exploding a bubble.
- Each bubble at any level of DFD is usually decomposed to anything from three to seven bubbles. Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm.

STRUCTURED DESIGN

- The aim of structured design is to transform the results of the structured analysis into a structure chart.
- A structure chart represents the software architecture.
- The various modules making up the system, the module dependency (i.e. which module calls which other modules), and the parameters that are passed among the different modules.
- The structure chart representation can be easily implemented using some programming language.
- The basic building blocks using which structure charts are designed are as following:
    - ***Rectangular boxes:*** A rectangular box represents a module.
    - ***Module invocation arrows:*** An arrow connecting two modules implies that during program execution control is passed from one module to the other in the direction of the connecting arrow.
    - ***Data flow arrows:*** These are small arrows appearing alongside the module invocation arrows. represent the fact that the named data passes from one module to the other in the direction of the arrow.
    - ***Library modules:*** A library module is usually represented by a rectangle with double edges. Libraries comprise the frequently called modules.
    - ***Selection:*** The diamond symbol represents the fact that one module of several modules connected with the diamond symbol i s invoked depending on the outcome of the condition attached with the diamond symbol.
    - ***Repetition:*** A loop around the control flow arrows denotes that the respective modules are invoked repeatedly.
- In any structure chart, there should be one and only one module at the top, called the root.
- There should be at most one control relationship between any two modules in the structure chart. This means that if module A invokes module B, module B cannot invoke module A.

<u>Flow Chart vs Structure chart:</u>

- Flow chart is a convenient technique to represent the flow of control in a program.
- A structure chart differs from a flow chart in three principal ways:
    1. It is usually difficult to identify the different modules of a program from its flow chart representation.
    2. Data interchange among different modules is not represented in a flow chart.
    3. Sequential ordering of tasks that i s inherent to a flow chart is suppressed in a structure chart.

<u>Transformation of a DFD Model into Structure Chart:</u>

- Systematic techniques are available to transform the DFD representation of a problem into a module structure represented as a structure chart.
- Structured design provides two strategies to guide transformation of a DFD into a structure chart:
    - Transform analysis
    - Transaction analysis

- Normally, one would start with the level 1 DFD, transform it into module representation using either the transform or transaction analysis and then proceed toward the lower level DFDs
- At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD.

Whether to apply transform or transaction processing?

Given a specific DFD of a model, one would have to examine the data input to the diagram. If all the data flow into the diagram are processed in similar ways (i.e. if all the input data flow arrows are incident on the same bubble in the DFD) then transform analysis is applicable. Otherwise, transaction analysis is applicable.

Transform Analysis:

- Transform analysis identifies the primary functional components (modules) and the input and output data for these components.
- The first step in transform analysis is to divide the DFD into three types of parts:
    - Input (afferent branch)
    - Processing (central transform)
    - Output (efferent branch)

- In the next step of transform analysis, the structure chart is derived by drawing one functional component each for the central transform, the afferent and efferent branches.
- These are drawn below a root module, which would invoke these modules.
- In the third step of transform analysis, the structure chart is refined by adding sub functions required by each of the high-level functional components.

Transaction Analysis:

- Transaction analysis is an alternative to transform analysis and is useful while designing transaction processing programs.
- A transaction allows the user to perform some specific type of work by using the software.

- For example, ‘issue book’, ‘return book’, ‘query book’, etc., are transactions.

- As in transform analysis, first all data entering into the DFD need to be identified.
- In a transaction-driven system, different data items may pass through different computation paths through the DFD.
- This is in contrast to a transform centered system where each data item entering the DFD goes through the same processing steps.
- Each different way in which input data is processed is a transaction. For each identified transaction, trace the input data to the output.
- All the traversed bubbles belong to the transaction.
- These bubbles should be mapped to the same module on the structure chart.
- In the structure chart, draw a root module and below this module draw each identified transaction as a module.

Detailed Design:

- During detailed design the pseudo code description of the processing and the different data structures are designed for the different modules of the structure chart.

- These are usually described in the form of module specifications (MSPEC). MSPEC is usually written using structured English.
- The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower level modules.
- The MSPEC for the leaf-level modules should describe in algorithmic form how the primitive processing steps are carried out.
- To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.

Design Review:

- After a design is complete, the design is required to be reviewed.
- The review team usually consists of members with design, implementation, testing, and maintenance perspectives.
- The review team checks the design documents especially for the following aspects:
    - ***Traceability***: Whether each bubble of the DFD can be traced to some module in the structure chart and vice versa.
    - ***Correctness***: Whether all the algorithms and data structures of the detailed design are correct.
    - ***Maintainability***: Whether the design can be easily maintained in future.
    - ***Implementation***: Whether the design can be easily and efficiently implemented.
- After the points raised by the reviewers are addressed by the designers, the design document becomes ready for implementation.

User Interface Design: Characteristics of a good user interface, Basic concepts, Types of user interfaces, Fundamentals of component-based GUI development, and user interface design methodology.

USER INTERFACE DESIGN

- The user interface portion of a software product is responsible for all interactions with the user. Almost every software product has a user interface.
- User interface part of a software product is responsible for all interactions.
- The user interface part of any software product is of direct concern to the end-users.
- No wonder then that many users often judge a software product based on its user interface
- an interface that is difficult to use leads to higher levels of user errors and ultimately leads to user dissatisfaction.
- Sufficient care and attention should be paid to the design of the user interface of any software product.
- Systematic development of the user interface is also important.
- Development of a good user interface usually takes a significant portion of the total system development effort.
- For many interactive applications, as much as 50 per cent of the total development effort is spent on developing the user interface part.
- Unless the user interface is designed and developed in a systematic manner, the total effort required to develop the interface will increase tremendously.

CHARACTERISTICS OF A GOOD USER INTERFACE

The different characteristics that are usually desired of a good user interface. Unless we know what exactly is expected of a good user interface, we cannot possibly design one.

- ***Speed of learning:***
    - A good user interface should be easy to learn.
    - A good user interface should not require its users to memorize commands.
    - Neither should the user be asked to remember information from one screen to another
        - **Use of metaphors and intuitive command names:** Speed of learning an interface is greatly facilitated if these are based on some dayto-day real-life examples or some physical objects with which the users are familiar with. The abstractions of real-life objects or concepts used in user interface design are called metaphors.
        - **Consistency**: Once a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions.
        - **Component-based interface:** Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar with.
    - The speed of learning characteristic of a user interface can be determined by measuring the training time and practice that users require before they can effectively use the software.

- ***<u>Speed of use:</u>***
  - Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands.
  - It indicates how fast the users can perform their intended tasks.
  - The time and user effort necessary to initiate and execute different commands should be minimal.
  - This can be achieved through careful design of the interface.
  - The most frequently used commands should have the smallest length or be available at the top of a menu.
- ***<u>Speed of recall:</u>***
  - Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximized.
  - This characteristic is very important for intermittent users.
  - Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures, and intuitive command names.
- ***<u>Error prevention:</u>***
  - A good user interface should minimize the scope of committing errors while initiating different commands.
  - The error rate of an interface can be easily determined by monitoring the errors committed by an average user while using the interface.
  - The interface should prevent the user from entering wrong values.
- ***<u>Aesthetic and attractive:</u>***
  - A good user interface should be attractive to use.
  - An attractive user interface catches user attention and fancy.
  - In this respect, graphics-based user interfaces have a definite advantage over text-based interfaces.
- ***<u>Consistency:</u>***
  - The commands supported by a user interface should be consistent.
  - The basic purpose of consistency is to allow users to generalize the knowledge about aspects of the interface from one part to another.
- ***<u>Feedback:</u>***
  - A good user interface must provide feedback to various user actions.
  - Especially, if any user request takes more than a few seconds to process, the user should be informed about the state of the processing of his request.
  - In the absence of any response from the computer for a long time, a novice user might even start recovery/shutdown procedures in panic.
- ***<u>Support for multiple skill levels:</u>***
  - A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users.
  - This is necessary because users with different levels of experience in using an application prefer different types of user interfaces.

    - Experienced users are more concerned about the efficiency of the command issue procedure, whereas novice users pay importance to usability aspects.
    - The skill level of users improves as they keep using a software product and they look for commands to suit their skill levels.
- ***<u>Error recovery (undo facility):</u>***
    - While issuing commands, even the expert users can commit errors.
    - Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface.
    - Users are inconvenienced if they cannot recover from the errors they commit while using a software.
    - If the users cannot recover even from very simple types of errors, they feel irritated, helpless, and out of control.
- ***<u>User guidance and on-line help:</u>***
    - Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software.
    - Whenever users need guidance or seek help from the system, they should be provided with appropriate guidance and help.

BASIC CONCEPTS:

User Guidance and Online help:

Users may seek help about the operation of the software any time while using the software. This is provided by the on-line help system.

1. ***Online help system:***
    - Users expect the on-line help messages to be tailored to the context in which they invoke the "help system".
    - Therefore, a good online help system should keep track of what a user is doing while invoking the help system and provide the output message in a context-dependent way.
2. ***Guidance messages:***
    - The guidance messages should be carefully designed to prompt the user about the next actions he might pursue, the current status of the system, the progress so far made in processing his last command, etc.
    - A good guidance system should have different levels of sophistication.
3. ***Error Messages:***
    - Error messages are generated by a system either when the user commits some error or when some errors encountered by the system during processing due to some exceptional conditions, such as out of memory, communication link broken, etc.
    - Users do not like error messages that are either ambiguous or too general such as "invalid input or system error". Error messages should be polite.

Mode-based and modeless interfaces:

- A mode is a state or collection of states in which only a subset of all user interaction tasks can be performed.

- In a modeless interface, the same set of commands can be invoked at any time during the running of the software.
- Thus, a modeless interface has only a single mode and all the commands are available all the time during the operation of the software.
- On the other hand, in a mode-based interface, different sets of commands can be invoked depending on the mode in which the system is.
- A mode-based interface can be represented using a state transition diagram.

Graphical User Interface versus Text-based User Interface:

- In a GUI multiple windows with different information can simultaneously be displayed on the user screen. user has the flexibility to simultaneously interact with several related items at any time
- Iconic information representation and symbolic information manipulation is possible in a GUI.
- A GUI usually supports command selection using an attractive and user-friendly menu selection system.
- In a GUI, a pointing device such as a mouse or a light pen can be used for issuing commands.
- A GUI requires special terminals with graphics capabilities for running and also requires special input devices such a mouse.
- A text-based user interface can be implemented even on a cheap alphanumeric display terminal.
- Graphics terminals are usually much more expensive than alphanumeric terminals, They have become affordable.

Types of User Interfaces

- Broadly speaking, user interfaces can be classified into the following three categories:
  - Command language-based interfaces.
  - Menu-based interfaces.
  - Direct manipulation interfaces.
- Each of these categories of interfaces has its own characteristic advantages and disadvantages.

Command Language-based Interface

- A command language-based interface—as the name itself suggests, is based on designing a command language which the user can use to issue the commands.
- The user is expected to frame the appropriate commands in the language and type them appropriately whenever required.
- A simple command language-based interface might simply assign unique names to the different commands.
- However, a more sophisticated command language-based interface may allow users to compose complex commands by using a set of primitive commands.
- The command language interface allows for the most efficient command issue procedure requiring minimal typing.

- a command language-based interface can be implemented even on cheap alphanumeric terminals.
- a command language-based interface is easier to develop compared to a menu-based or a direct-manipulation interface because compiler writing techniques are well developed
- command language-based interfaces suffer from several drawbacks.
- command language-based interfaces are difficult to learn and require the user to memorize the set of primitive commands.
- Most users make errors while formulating commands.
- All interactions with the system are through a key-board and cannot take advantage of effective interaction devices such as a mouse.

- The designer has to decide what mnemonics (command names) to use for the different commands.
- The designer has to decide whether the users will be allowed to redefine the command names to suit their own preferences.
- The designer has to decide whether it should be possible to compose primitive commands to form more complex commands.

Menu-Based Interfaces:

- An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands.
- A menu-based interface is based on recognition of the command names, rather than recollection.
- In a menu-based interface the typing effort is minimal.
- A major challenge in the design of a menu-based interface is to structure a large number of menu choices into manageable forms.
- Techniques available to structure a large number of menu items:
    - ***Scrolling menu:***
        - Sometimes the full choice list is large and cannot be displayed within the menu area, scrolling of the menu items is required.
        - In a scrolling menu all the commands should be highly correlated, so that the user can easily locate a command that he needs.
        - This is important since the user cannot see all the commands at any one time.
    - ***Walking menu:***
        - Walking menu is very commonly used to structure a large collection of menu items.
        - When a menu item is selected, it causes further menu items to be displayed adjacent to it in a sub-menu.
        - A walking menu can successfully be used to structure commands only if there are tens rather than hundreds of choices.

- ***Hierarchical menu:***
    - This type of menu is suitable for small screens with limited display area such as that in mobile phones.
    - The menu items are organized in a hierarchy or tree structure.
    - Selecting a menu item causes the current menu display to be replaced by an appropriate sub-menu.

Direct Manipulation Interfaces:

- Direct manipulation interfaces present the interface to the user in the form of visual models.
- Direct manipulation interfaces are sometimes called iconic interfaces.
- In this type of interface, the user issues commands by performing actions on the visual representations of the objects, e.g., pull an icon representing a file into an icon representing a trash box, for deleting the file.
- Important advantages of iconic interfaces include the fact that the icons can be recognised by the users very easily, and that icons are language independent.
- However, experienced users find direct manipulation interfaces very useful too.
- Also, it is difficult to give complex commands using a direct manipulation interface.

USER INTERFACE DESIGN METHODOLOGY

- At present, no step-by-step methodology is available which can be followed by rote to come up with a good user interface.
- What we present in this section is a set of recommendations which you can use to complement your ingenuity.

A GUI Design Methodology:

- The GUI design methodology we present here is based on the seminal work of Frank Ludolph.
- Our user interface design methodology consists of the following important steps:
    - Examine the use case model of the software.
    - Interview, discuss, and review the GUI issues with the end-users.
    - Task and object modeling.
    - Metaphor selection.
    - Interaction design and rough layout.
    - Detailed presentation and graphics design.
    - GUI construction.
    - Usability evaluation.

Examining the use case model

- The starting point for GUI design is the use case model.
- This captures the important tasks the users need to perform using the software.
- Metaphors help in interface development at lower effort and reduced costs for training the users.
- Metaphors can also be based on physical objects such as a visitor's book, a catalog, a pen, a brush, a scissor, etc.

- A solution based on metaphors is easily understood by the users, reducing learning time and training costs.

Task and object Modeling:

- A task is a human activity intended to achieve some goals.
- Examples of task goals can be as follows:
    - Reserve an airline seat
    - Buy an item Transfer money from one account to another
    - Book a cargo for transmission to an address.
- A task model is an abstract model of the structure of a task.
- A task model should show the structure of the subtasks that the user needs to perform to achieve the overall task goal.
- Each task can be modeled as a hierarchy of subtasks.
- A task model can be drawn using a graphical notation similar to the activity network model.
- A user object model is a model of business objects which the end-users believe that they are interacting with.
- The objects in a library software may be books, journals, members, etc.

Metaphor selection:

- The first place one should look for while trying to identify the candidate metaphors is the set of parallels to objects, tasks, and terminologies of the use cases.
- If no obvious metaphors can be found, then the designer can fall back on the metaphors of the physical world of concrete objects.
- The appropriateness of each candidate metaphor should be tested by restating the objects and tasks of the user interface model in terms of the metaphor.

Interaction design and rough layout

- The interaction design involves mapping the subtasks into appropriate controls, and other widgets such as forms, text box, etc.
- This involves making a choice from a set of available components that would best suit the subtask.
- Rough layout concerns how the controls, and other widgets to be organized in windows.

Detailed presentation and graphics design

- Each window should represent either an object or many objects that have a clear relationship to each other.
- At one extreme, each object view could be in its own window. But, this is likely to lead to too much window opening, closing, moving, and resizing.
- At the other extreme, all the views could be placed in one window side-by-side, resulting in a very large window.
- This would force the user to move the cursor around the window to look for different objects.

GUI construction

- Some of the windows have to be defined as modal dialogs.
- When a window is a modal dialog, no other windows in the application are accessible until the current window is closed.
- When a modal dialog is closed, the user is returned to the window from which the modal dialog was invoked.
- Modal dialogs are commonly used when an explicit confirmation or authorisation step is required for an action.

User interface inspection

- Nielson studied common usability problems and built a check list of points which can be easily checked for an interface. The following checklist is based on the work of Nielson:
  - Visibility of the system status
  - Match between the system and the real world
  - Undoing mistakes
  - Consistency
  - Recognition rather than recall
  - Support for multiple skill levels
  - Aesthetic and minimalist design
  - Help and error messages
  - Error prevention.