UNIT 3

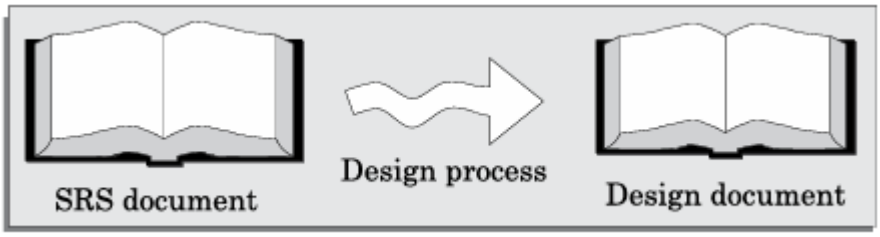
Contents:

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
Function-Oriented Software Design: Overview of SA/SD Methodology, Structured Analysis, Structured Design, Detailed Design, Design Review.
User Interface Design: Characteristics of Good User Interface, Basic Concepts,

Software Design: The activities carried out during the design phase (called as design process) transform the SRS document into the design document.



The design process

OVERVIEW OF THE DESIGN PROCESS:

The design process is a structured approach used by designers to solve problems and create solutions that meet specific needs. It's a combination of creativity, research, planning, and testing. Here's an overview of the typical steps involved in the design process.

Outcome of the Design Process:

Different modules required: The different modules in the solution should be identified. Each module is a collection of functions and the data shared by these functions. Each module should accomplish some well-defined task out of the overall responsibility of the software**.**

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: Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module.

Algorithms required to implement the individual modules: Each function in a module usually performs some processing activity. 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.

Classification of Design Activities:

Design activities can be classified into various categories based on the type of design process, goals, and the specific stage of development. These classifications help organize and structure the design process, making it more efficient and manageable.

- Preliminary (or high-level) design, and
- Detailed design.
- 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. When the high-level design is complete, the problem should have been decomposed into many small functionally independent modules that are cohesive, have low coupling among themselves, and are arranged in a hierarchy. Many different types of notations have been used to represent a high-level design. A notation that is widely being used for procedural development is a tree-like diagram called the structure chart.
- The outcome of the detailed design stage is usually documented in the form of a module specification (MSPEC) document. After the high-level design is complete, the problem would have been decomposed into small modules, and the data structures and algorithms to be used described using MSPEC and can be easily grasped by programmers for initiating coding.

Classification of Design Methodologies:

The design activities vary considerably based on the specific design methodology being used. A large number of software design methodologies are available. We can roughly classify these methodologies into procedural and object-oriented approaches. These two approaches are two fundamentally different design paradigms.

Do design techniques result in unique solutions?

Even while using the same design methodology, different designers usually arrive at very different design solutions. The reason is that a design technique often requires the designer to make many subjective decisions and work out compromises to contradictory objectives. As a result, it is possible that even the same designer can work out many different solutions to the same problem. Therefore, obtaining a good design would involve trying out several alternatives (or candidate solutions) and picking out the best one.

Analysis versus design:

Analysis and design activities differ in goal and scope. The analysis results are generic and does not consider implementation or the issues associated with specific platforms. The analysis model is usually documented using some graphical formalism. In case of the function-oriented approach that we are going to discuss, the analysis model would be documented using data flow diagrams (DFDs), whereas the design would be documented using structure chart. On the other hand, for object-oriented approach, both the design model and the analysis model will be documented using unified modelling language (UML).

HOW TO CHARACTERISE A GOOD SOFTWARE DESIGN?

A good software design can be characterized by several key attributes that ensure the system is efficient, maintainable, scalable, and user-friendly. Here are some important aspects:

1. Simplicity

- The design should be as simple as possible while still meeting all requirements. Avoid over-engineering, unnecessary complexity, or adding features that aren't essential. A simple design is easier to understand, implement, and maintain.

2. Modularity

- A good design breaks the system into smaller, independent modules that are responsible for distinct functionalities. Each module should have a well-defined interface and should interact with other modules in a predictable and minimal way.
- This promotes reusability, testing, and ease of modification.

3. Cohesion

- Cohesion refers to how closely related the responsibilities of a module are. A highly cohesive module performs tasks that are closely related, making it easier to understand, maintain, and extend.

4. Coupling

- Coupling refers to the degree of dependency between different modules. A good software design minimizes coupling, meaning that changes in one module should ideally not require changes in others. Loose coupling allows for better flexibility and easier maintenance.

5. Scalability

- The design should support growth, whether it's handling more data, more users, or more features. A scalable design allows the software to expand in size or capacity with minimal changes to the underlying structure.

6. Maintainability

- The design should facilitate easy updates, bug fixes, and improvements. Well-written and documented code, as well as good architectural choices, make the software easier to maintain over time.

7. Testability

- A good design ensures that components and modules are easy to test in isolation. This can be achieved by adhering to principles like modularity, separation of concerns, and using dependency injection to make testing easier.

8. Performance

- The design should meet performance requirements, including speed, memory usage, and responsiveness. However, it's essential to strike a balance; optimizing for performance too early might introduce unnecessary complexity. Instead, design with performance in mind but also focus on other principles like maintainability and scalability.

9. Flexibility

- Good software design accommodates future changes without major overhauls. The design should be extensible and adaptable, allowing new features to be added easily and without disrupting existing functionality.

10. Reusability

- Components of the design should be reusable across different parts of the system or even in other projects. This reduces redundancy and development time, and makes the system more efficient in the long run.

11. Separation of Concerns

- A good design separates different concerns (e.g., UI, business logic, data access) so that each component addresses a distinct aspect of the system. This makes it easier to manage, maintain, and understand the system.

12. Clear Documentation

- A well-documented design (including code comments, architectural diagrams, and flowcharts) allows developers to understand the reasoning behind design choices and how different parts of the system interact.

13. Security

- Security considerations should be embedded in the design from the outset. This includes ensuring data protection, authentication, authorization, and the ability to handle vulnerabilities in a structured way.

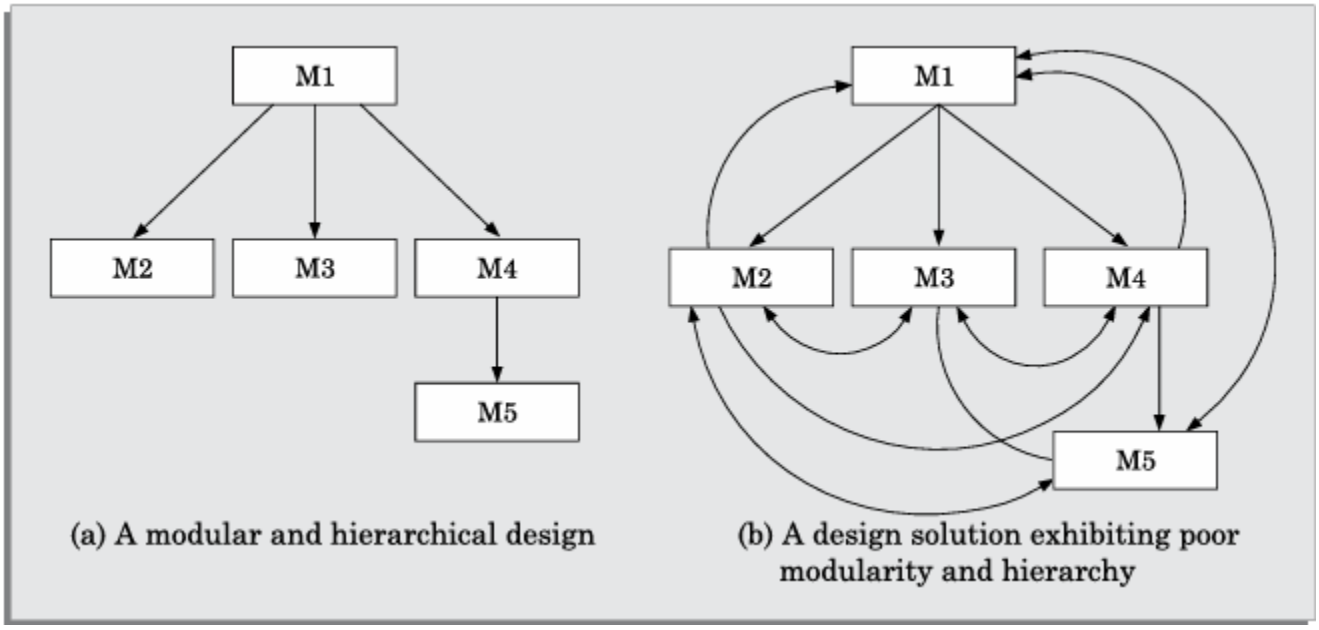
14. User-Centric Design

- A good software design also keeps the end user in mind. This means ensuring the software is intuitive, responsive, and solves the user's problems efficiently. It also involves considering user feedback and iterating the design accordingly.

15. Compliance and Standards

- Following industry best practices, coding standards, and relevant compliance guidelines (such as GDPR, HIPAA, etc.) ensures that the software design meets necessary legal, regulatory, and security requirements.

In summary, a good software design is one that balances competing factors—performance, simplicity, flexibility, maintainability, and usability—while being structured in a way that supports future growth and changes.



Two design solutions to the same problem.

LAYERED ARRANGEMENT OF MODULES

In software engineering, a layered arrangement of modules, often referred to as layered architecture, is a widely used design pattern that structures an application into distinct layers, each with a specific responsibility.

An important characteristic feature of a good design solution is layering of the modules. A layered design achieves control abstraction and is easier to understand and debug.

In a layered design solution, the modules are arranged into several layers based on their call relationships. A module is allowed to call only the modules that are at a lower layer. That is, a module should not call a module that is either at a higher layer or even in the same layer. Figure 5.6(a) shows a layered design, whereas Figure 5.6(b) shows a design that is not layered. Observe that the design solution shown in Figure 5.6(b), is actually not layered since all the modules can be considered to be in the same layer. In the following, we state the significance of a layered design and subsequently we explain it.

In a layered design, the top-most module in the hierarchy can be considered as a manager that only invokes the services of the lower level module to discharge its responsibility. The modules at the intermediate layers offer services to their higher layer by invoking the services of the lower layer modules and also by doing some work themselves to a limited extent. The modules at the lowest layer are the worker modules. These do not invoke services of any module and entirely carry out their

responsibilities by themselves.

Superordinate and subordinate modules: In a control hierarchy, a module that controls another module is said to be superordinate to it. Conversely, a module controlled by another module is said to be subordinate to the controller.

Visibility: A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visible to a module.

Control abstraction: In a layered design, a module should only invoke the functions of the modules that are in the layer immediately below it. In other words, the modules at the higher layers, should not be visible (that is, abstracted out) to the modules at the lower layers. This is referred to as control abstraction.

Depth and width: Depth and width of a control hierarchy provide an indication of the number of layers and the overall span of control respectively. For the design of Figure 5.6(a), the depth is 3 and width is also 3.

Fan-out: Fan-out is a measure of the number of modules that are directly controlled by a given module. In Figure 5.6(a), the fan-out of the module M1 is 3. A design in which the modules have very high fan-out numbers is not a good design. The reason for this is that a very high fan-out is an indication that the module lacks cohesion. A module having a large fan-out (greater than 7) is likely to implement several different functions and not just a single cohesive function.

Fan-in: Fan-in indicates the number of modules that directly invoke a given module. High fan-in represents code reuse and is in general, desirable in a good design. In Figure 5.6(a), the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2

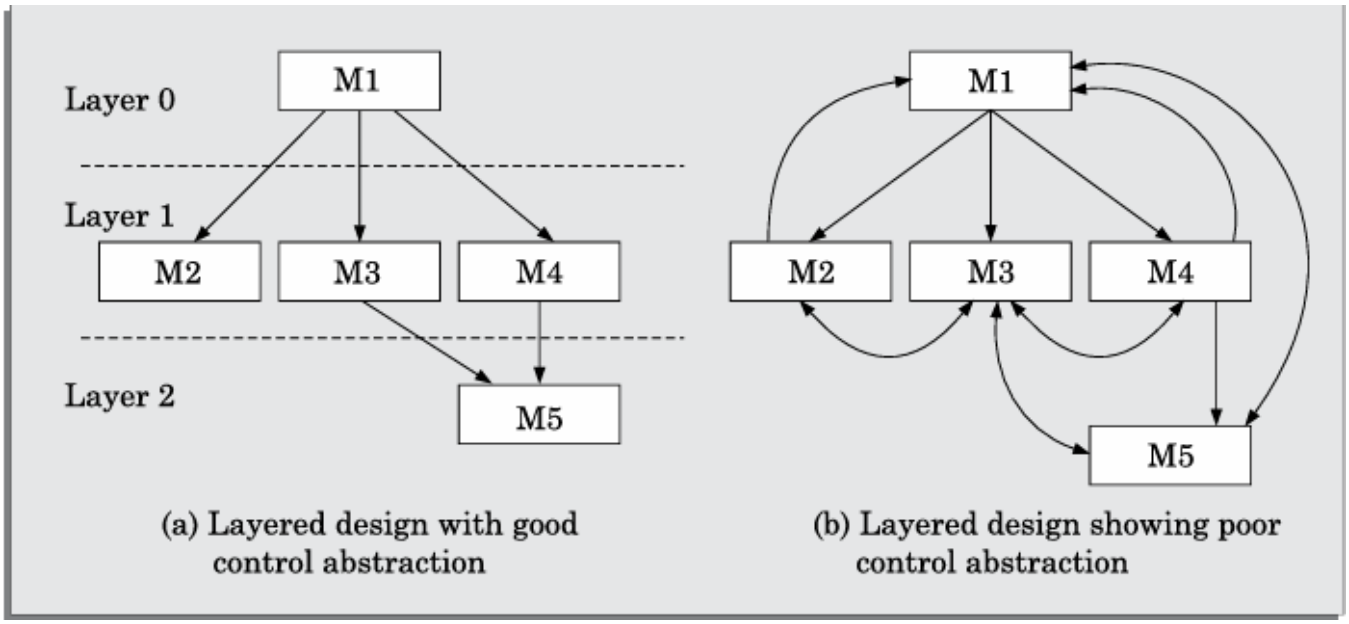


FIGURE 5.6 Examples of good and poor control abstraction.

COHESION AND COUPLING

We have so far discussed that 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. Let us now define what is meant by cohesion and coupling. In this section, we first elaborate the concepts of cohesion and coupling. Subsequently, we discuss the classification of cohesion and coupling

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.

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 with each other at all or at best interact by passing no data or only a few primitive data items, they are said to have low coupling.

Cohesion: To understand cohesion, let us first understand an analogy. Suppose you listened to a talk by some speaker. You would call the speech to be cohesive, if all the sentences of the speech played some role in giving the talk a single and focused theme. Now, we can extend this to a module in a design solution. 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 co-operate with each other to perform a single piece of work, then the module has very poor cohesion

Classification 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.

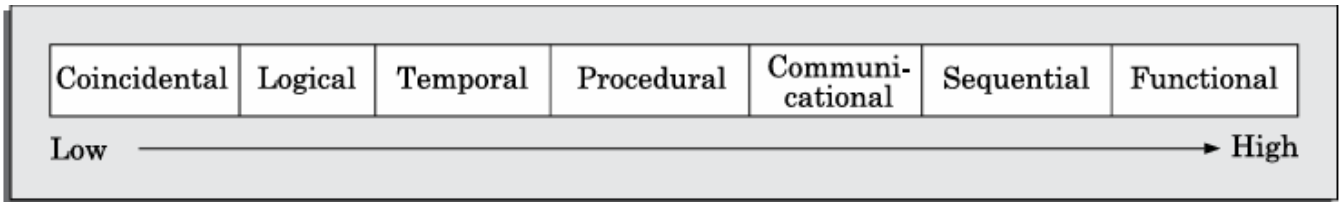


FIGURE 5.3 Classification of cohesion.

Coincidental cohesion: A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, we can say that the module contains a random collection of functions. It is likely that the functions have been placed in the module out of pure coincidence rather than through some thought or design.

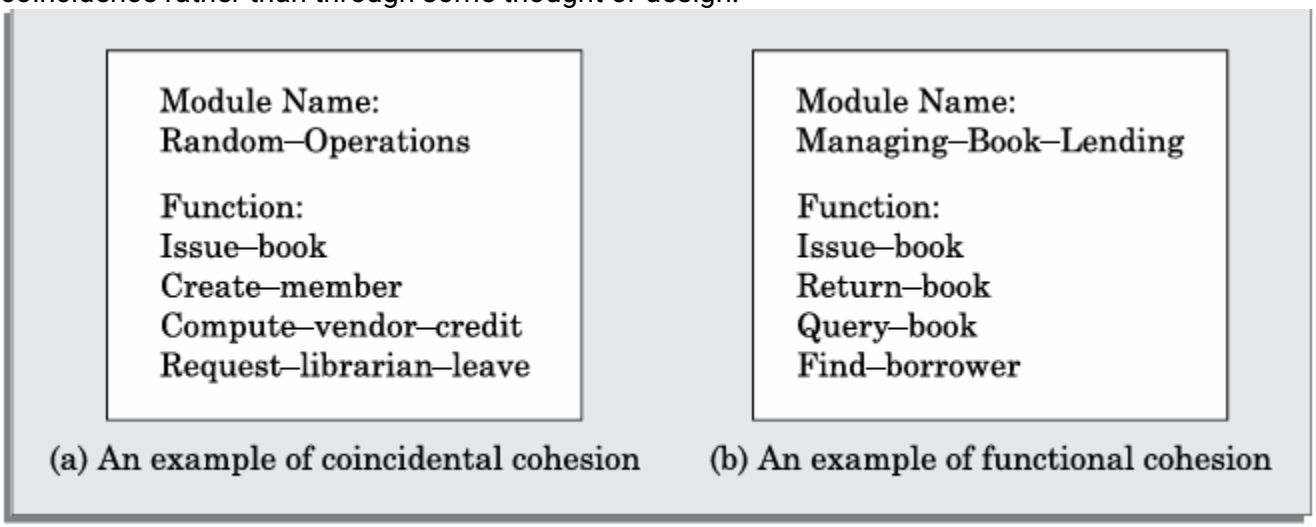


FIGURE 5.4 Examples of cohesion.

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. For example, a module containing all the functions required to manage employees' pay-roll displays functional cohesion. In this case, all the functions of the module (e.g., compute Overtime (), compute Work Hours (), compute Deductions (), etc.) work together to generate the pay slips of the employees.

Classification of Coupling:
The coupling between two modules indicates the degree of interdependence between them. Intuitively, if two modules interchange large amounts of data, then they are highly interdependent or coupled. We can alternately state this concept as follows.
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 between two modules depends on their interface complexity.

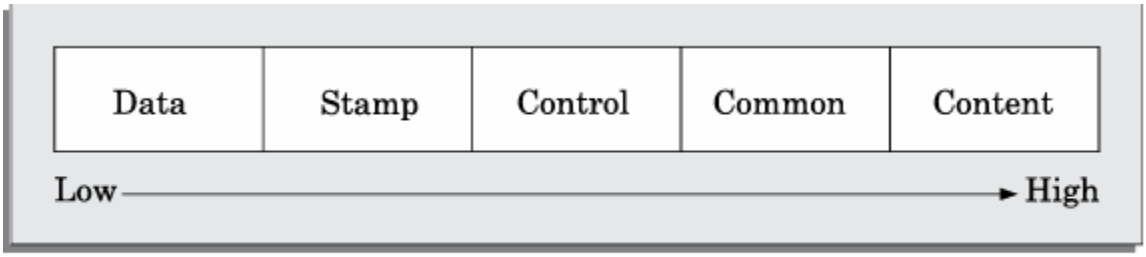


FIGURE 5.5 Classification of coupling.

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. This data item should be problem related and not used for control purposes.
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. An example of control coupling is a flag set in one module and tested in another module.
Common coupling: Two modules are common coupled, if they share some global data items.
Content coupling: Content coupling exists between two modules, if they share code. That is, a jump from one module into the code of another module can occur. Modern high-level programming languages such as C do not support such jumps across modules.

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. For development of large programs, the object-oriented approach is becoming increasingly popular due to certain advantages that it offers. On the other hand, function oriented designing is a mature technology and has a large following.

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 which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge. This high-level function may be refined into the following subfunctions:

- assign-membership-number
- create-member-record
- print-bill

Each of these subfunctions may be split into more detailed subfunctions and so on.

Centralised 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. For example, the set of books (i.e. whether borrowed by different users or available for issue) determines the state of a library automation system. Such data in procedural programs usually have global scope and are shared by many modules. The system state is centralised 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

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 system state is decentralised since there is no globally shared data in the system and data is stored in each object. For example, in a library automation software, each library member may be a separate object with its own data and functions to operate on the stored data. The methods defined for one object cannot directly refer to or change the data of other objects.

The object-oriented design paradigm makes extensive use of the principles of abstraction and decomposition as explained below. Objects decompose a system into functionally independent modules. Objects can also be considered as instances of abstract data types (ADTs). The ADT concept did not originate from the object-oriented approach. In fact, ADT concept was extensively used in the ADA programming language introduced in the 1970s. ADT is an important concept that forms an important pillar of object-orientation.

Data abstraction: The principle of data abstraction implies that how data is exactly stored is abstracted away. This means that any entity external to the object (that is, an instance of an ADT) would have no knowledge about how data is exactly stored, organised, and manipulated inside the object. The entities external to the object can access the data internal to an object only by calling certain well-defined methods supported by the object. Consider an ADT such as a stack. The data of a stack object may internally be stored in an array, a linearly linked list, or a bidirectional linked list. The external entities have no knowledge of this and can access data of a stack object only through the supported operations such as push and pop

Data structure: A data structure is constructed from a collection of primitive data items. Just as a civil

engineer builds a large civil engineering structure using primitive building materials such as bricks, iron rods, and cement; a programmer can construct a data structure as an organised collection of primitive data items such as integer, floating point numbers, characters, etc.

Data type: A type is a programming language terminology that refers to anything that can be instantiated. For example, int, float, char, etc., are the basic data types supported by C programming language. Thus, we can say that ADTs are user defined data types.

In object-orientation, classes are ADTs. But, what is the advantage of developing an application using ADTs? Let us examine the 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. The encapsulation principle requires that data can be accessed and manipulated only through the methods supported by the object and not directly. This localises the errors. The reason for this is as follows. No program element is allowed to change a data, except through invocation of one of the methods. So, any error can easily be traced to the code segment changing the value. That is, the method that changes a data item, making it erroneous can be easily identified.
- An ADT-based design displays high cohesion and low coupling. Therefore, object- oriented designs are highly modular.
- Since the principle of abstraction is used, it makes the design solution easily understandable and helps to manage complexity.

Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from a super class. Conceptually, objects communicate by message passing. Objects have their own internal data. Thus an object may exist in different states depending the values of the internal data. In different states, an object may behave differently.

Agility: Agility, in a broad sense, refers to the ability to move quickly and easily, or to think and understand quickly. However, in the context of business and especially software development, it has a more specific meaning.

Definition of Agility (in a business/software context):

Agility is the capacity of an organization or team to:

Respond quickly and effectively to changes: This includes changes in market conditions, customer needs, technology, and other factors.

Adapt to evolving requirements: Instead of rigidly sticking to a plan, agile entities embrace flexibility and are willing to adjust their approach as needed.

Deliver value iteratively and incrementally: This involves breaking down work into smaller, manageable chunks and delivering results frequently, allowing for continuous feedback and improvement.

Foster collaboration and communication: Agility emphasizes teamwork, open communication, and close collaboration among stakeholders.

Embrace continuous improvement: Agile entities are constantly seeking ways to improve their processes and become more efficient.

Agility and the Cost of Change

In software engineering, "agility" and "the cost of change" are closely related concepts, particularly within the context of agile development methodologies. Here's a breakdown of their relationship:

Agility:

- Agility in software development refers to the ability to respond effectively and rapidly to changes. These changes can include:
    - Evolving customer requirements.

    - Shifting market conditions.
    - Technological advancements.
- Agile methodologies, such as Scrum and Kanban, emphasize:
    - Iterative development.
    - Frequent feedback.
    - Close collaboration.
    - Flexibility.

The Cost of Change:

- Traditionally, in software development (especially with waterfall models), the cost of change increases significantly as a project progresses.
    - Changes made early in the development lifecycle are relatively inexpensive.
    - Changes made later, during testing or deployment, can be very costly and disruptive.
- Agile methodologies aim to "flatten" the cost of change curve.
    - By embracing iterative development and frequent feedback, agile teams can incorporate changes throughout the development process with less disruption.
    - This is achieved through:
        - Short development cycles (sprints).
        - Continuous integration and testing.
        - Regular communication with stakeholders.

The Relationship:

- Agility is, in essence, a strategy to manage and reduce the cost of change.
- Agile practices are designed to make it easier and less expensive to adapt to evolving requirements.
- By prioritizing flexibility and responsiveness, agile teams can:
    - Deliver valuable software more quickly.
    - Reduce the risk of building software that no longer meets the customer's needs.
    - Improve customer satisfaction.

Key Takeaways:

- Agile development recognizes that change is inevitable in software development.
- Agile practices are implemented to minimize the negative impact of change on project cost and schedule.
- The goal is to create a process that can adapt to change efficiently and effectively.

Key Agile Practices That Minimize Change Costs:

- **Short Iterations:**
    - Shorter development cycles mean smaller, more manageable changes.
    - Changes are contained within a single sprint, limiting their impact.
- **Continuous Integration and Continuous Delivery (CI/CD):**
    - Automated testing and deployment ensure that changes can be integrated and released quickly and reliably.
    - This reduces the risk and cost of late-stage changes.
- **Regular Communication and Feedback:**
    - Close collaboration with stakeholders ensures that requirements are constantly validated and adjusted.
    - This minimizes the risk of building software that doesn't meet evolving needs.
- **Flexible Design:**
    - Agile encourages flexible architectures that can adapt to change.
    - This reduces the amount of rework required when changes occur.

Agile Process

An Agile process in software engineering is a collection of methodologies and practices that prioritize

iterative development, flexibility, and collaboration to deliver software efficiently and effectively. It contrasts with traditional, linear approaches like the Waterfall model. Here's a detailed look:

Fig. Agile Model

Core Principles and Values (Agile Manifesto):

The foundation of Agile lies in the Agile Manifesto, which emphasizes:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.[1]
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

These values are supported by 12 principles that guide Agile practices.

Key Components of an Agile Process:

1. **Iterative and Incremental Development:**
    - Software is developed in short cycles called iterations or sprints.
    - Each iteration produces a working, potentially shippable increment of the product.
    - This allows for early and frequent feedback, enabling adjustments along the way.
2. **Sprints/Iterations:**

- Sprints are time-boxed periods (typically 2-4 weeks) during which a specific set of features is developed.
- Each sprint includes planning, development, testing, and review.
- At the end of each sprint, a working increment of the software is delivered.

3. **User Stories:**
   - Requirements are expressed as user stories, which are short, simple descriptions of a feature from the user's perspective.
   - Example: "As a user, I want to be able to search for products, so that I can find what I need quickly."
4. **Product Backlog:**
   - A prioritized list of all the features, user stories, and tasks that need to be completed.
   - The product backlog is constantly evolving and updated based on feedback and changing requirements.
5. **Sprint Planning:**
   - At the beginning of each sprint, the team selects user stories from the product backlog to be completed during that sprint.
   - Tasks are broken down, and estimates are provided.
6. **Daily Stand-up Meetings (Daily Scrum):**
   - Short, daily meetings (typically 15 minutes) where team members share their progress, challenges, and plans for the day.
   - This promotes communication and helps identify and resolve issues quickly.
7. **Sprint Review:**
   - At the end of each sprint, the team demonstrates the completed work to stakeholders and gathers feedback.
8. **Sprint Retrospective:**
   - A meeting where the team reflects on the sprint and identifies areas for improvement.
   - This promotes continuous improvement and helps the team become more efficient.
9. **Continuous Integration and Continuous Delivery (CI/CD):**
   - Automated processes for building, testing, and deploying software.
   - CI/CD enables frequent and reliable releases, reducing the risk of errors and improving efficiency.
10. **Collaboration and Communication:**

- Agile emphasizes close collaboration between developers, testers, product owners, and other stakeholders.
- Open communication and feedback are essential for success.

Common Agile Methodologies:

- **Scrum:** A popular framework that uses sprints, daily stand-ups, and other practices to manage development.
- **Kanban:** A visual workflow management system that focuses on limiting work in progress and improving flow.
- **Extreme Programming (XP):** A methodology that emphasizes technical practices such as pair programming, test-driven development, and continuous integration.

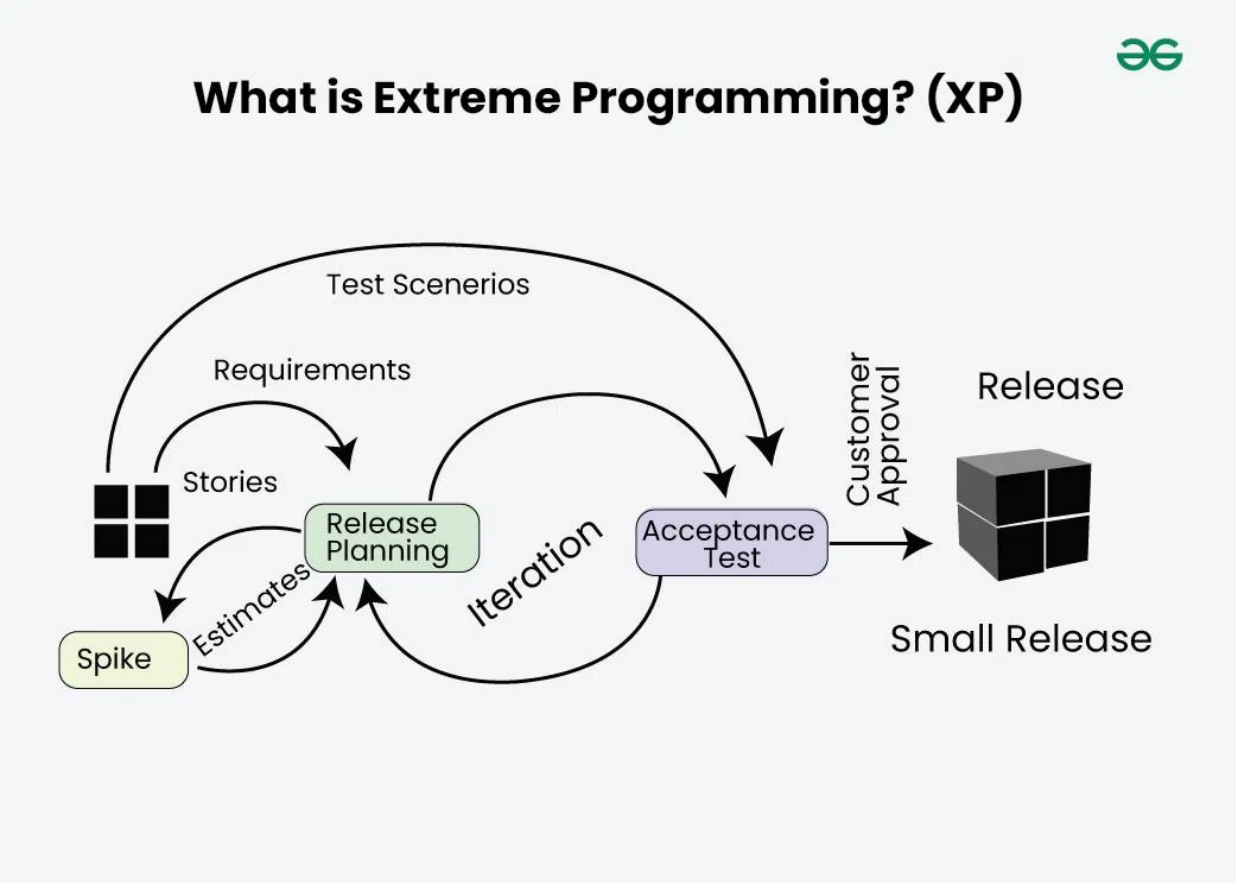
Benefits of Agile:

- Increased flexibility and adaptability.
- Improved customer satisfaction.
- Faster time to market.
- Reduced risk.
- Enhanced team collaboration.
- Higher quality software.

Agile is not a one-size-fits-all solution, and the specific practices used can vary depending on the project and the team. However, the core principles of iterative development, flexibility, and collaboration remain constant.

Extreme Programming (XP)

Extreme Programming (XP) is an agile software development framework that emphasizes high quality, responsiveness to changing customer requirements, and close collaboration. It's known for its strong focus on technical practices. Here's a detailed overview:



Core Values:

XP is built upon five core values that guide its practices:

- **Communication:**
  - XP emphasizes clear and frequent communication among team members and with the customer.
  - Face-to-face communication is preferred.
- **Simplicity:**
  - XP advocates for designing and implementing the simplest solution that meets the current requirements.
  - Avoid unnecessary complexity.
- **Feedback:**
  - XP relies on frequent feedback loops to ensure that the software is meeting the customer's needs and that the team is improving its processes.
  - This includes feedback from tests, customers, and team members.
- **Courage:**
  - XP encourages team members to make bold decisions, such as refactoring code or

discarding code that is no longer needed.

- It also includes the courage to communicate problems.

- **Respect:**
  - XP emphasizes respect for all team members and their contributions.

Key Practices:

XP implements these values through a set of key practices:

- **Planning Game:**
  - The customer and developers collaborate to plan the project.
  - User stories are used to define requirements.
  - Releases and iterations are planned.
- **Small Releases:**
  - Software is released in small, frequent increments.
  - This allows for early feedback and reduces the risk of major problems.
- **Metaphor:**
  - A common metaphor is used to describe the system, which helps to improve communication and understanding among team members.
- **Simple Design:**
  - The design is kept as simple as possible, focusing on meeting the current requirements.
- **Testing:**
  - Test-Driven Development (TDD): Tests are written before the code is written. This helps to ensure that the code meets the requirements and that it is of high quality.
  - Unit tests and acceptance tests are used.
- **Refactoring:**
  - The code is continuously refactored to improve its design and maintainability.
- **Pair Programming:**
  - Two programmers work together at one workstation. This helps to improve code quality and knowledge sharing.
- **Collective Ownership:**
  - All team members are responsible for all of the code.

- **Continuous Integration:**
    - Code changes are integrated and tested frequently.
- **40-Hour Week:**
    - XP emphasizes sustainable development, which includes avoiding overtime.
- **On-Site Customer:**
    - A customer representative is available to answer questions and provide feedback.
- **Coding Standards:**
    - The team follows a common set of coding standards.

Key Characteristics:

- Strong emphasis on technical excellence.
- High level of customer involvement.
- Frequent feedback and adaptation.
- Focus on simplicity and maintainability.

Benefits:

- Improved code quality.
- Increased responsiveness to changing requirements.
- Enhanced team collaboration.
- Reduced risk.

Extreme Programming is a very disciplined Agile methodology that places a very strong emphasis on the engineering practices of software development.

Other Agile Process Models

Besides Scrum and Extreme Programming (XP), several other Agile process models exist in software engineering. Each has its own focus and set of practices. Here's a detailed look at some of them:

1. Kanban:

- **Focus:** Visualizing workflow, limiting work in progress (WIP), and continuous delivery.
- **Key Principles:**
    - **Visualize the workflow:** Use a Kanban board to represent the flow of work.
    - **Limit WIP:** Restrict the number of tasks in progress at any given time to improve flow.
    - **Manage flow:** Focus on optimizing the flow of work through the system.

- **Make process policies explicit:** Define clear rules for how work is done.
- **Implement feedback loops:** Regularly review and improve the process.
- **Improve collaboratively, evolve experimentally:** Use data to drive process improvements.

- **Characteristics:**
  - Emphasis on continuous flow rather than iterations.
  - Pull system: Work is pulled into the workflow when capacity is available.
  - Focus on measuring and improving cycle time.
  - Suitable for projects with continuous delivery requirements.
- **Benefits:**
  - Improved workflow visualization.
  - Reduced bottlenecks.
  - Increased throughput.
  - Enhanced flexibility.

2. Lean Software Development:

- **Focus:** Eliminating waste, amplifying learning, and delivering fast.
- **Key Principles:**
  - **Eliminate waste:** Identify and remove anything that doesn't add value.
  - **Amplify learning:** Use short feedback loops and continuous improvement.
  - **Decide as late as possible:** Defer decisions until the last responsible moment.
  - **Deliver as fast as possible:** Focus on rapid delivery of value.
  - **Empower the team:** the team autonomy and responsibility.
  - **Build integrity in:** Ensure that the software is of high quality.
  - **See the whole**: Optimize the entire value stream.
- **Characteristics:**
  - Emphasis on efficiency and value delivery.
  - Focus on continuous improvement and learning.
  - Strong customer focus.
  - Often used in conjunction with other Agile methods.

- **Benefits:**
    - Reduced waste.
    - Faster delivery.
    - Improved quality.
    - Increased customer satisfaction.

3. Dynamic Systems Development Method (DSDM):

- **Focus:** Rapid application development with a strong focus on business needs.
- **Key Principles:**
    - Active user involvement is imperative.
    - DSDM teams must be empowered to make decisions.
    - The focus is on frequent delivery of products.[1]
    - The primary criterion for acceptance of deliverables is that they meet the business need.
    - Iterative and incremental development is necessary to converge on the correct business solution.
    - All changes during development are reversible.
    - The high-level scope and plan should be established up front.
- **Characteristics:**
    - Time-boxed iterations.
    - Strong emphasis on business requirements.
    - Formalized roles and responsibilities.
    - Focus on fitness for business purpose.
- **Benefits:**
    - Rapid application development.
    - High level of user involvement.
    - Strong business focus.

4. **Feature-Driven Development (FDD):**

- **Focus:** Designing and building software based on features.
- **Key Principles:**

- Develop an overall model.
- Build a features list.
- Plan by feature.
- Design by feature.
- Build by feature.

- **Characteristics:**
  - Short iterations (typically two weeks).
  - Emphasis on detailed design.
  - Use of domain modelling.
  - Regular build and integration.
- **Benefits:**
  - Clear and well-defined process.
  - Emphasis on high-quality design.
  - Good for large or complex projects.

Each of these agile methodologies brings different strengths to software projects, and the best choice depends on the specific project requirements, team dynamics, and organizational culture.

Tool Set for the Agile Process

To effectively implement an Agile process, a team needs a robust set of tools that support collaboration, communication, and efficient workflow. Here's a breakdown of essential tool categories and examples:

1. Project Management & Collaboration:

- **Purpose:** To track progress, manage tasks, and facilitate collaboration.

- **Examples:**
    - **Jira:** A powerful tool for Agile project management, issue tracking, and sprint planning. It's highly customizable and integrates with many other tools.
    - **Trello:** A visual Kanban-style tool that's easy to use and great for managing simple workflows.
    - **Asana:** A flexible work management platform that offers various views (list, board, calendar) and is suitable for cross-functional teams.
    - **Azure DevOps:** A comprehensive suite that includes project management, version control, and CI/CD pipelines.
    - **Monday.com:** A customizable work OS that allows teams to build workflows, dashboards, and integrations.
    - **Confluence:** A collaborative workspace for documentation, knowledge sharing, and team communication.

2. Communication:

- **Purpose:** To enable real-time communication and keep everyone aligned.
- **Examples:**
    - **Slack/Microsoft Teams:** Instant messaging, file sharing, and video conferencing platforms that integrate with other tools.
    - **Zoom/Google Meet:** Video conferencing tools for meetings, stand-ups, and demos.

3. **Version Control & Code Collaboration:**

- **Purpose:** To manage code changes, collaborate on code, and ensure code quality.
- **Examples:**
    - **Git (GitHub, GitLab, Bitbucket):** Distributed version control systems that allow for branching, merging, and code reviews.

4. Continuous Integration/Continuous Delivery (CI/CD):

- **Purpose:** To automate the build, test, and deployment process.
- **Examples:**
    - **Jenkins:** A highly customizable open-source automation server.
    - **Circle CI/Travis CI:** Cloud-based CI/CD platforms that integrate with Git repositories.
    - **GitLab CI/CD:** Integrated CI/CD pipelines within GitLab.
    - **Azure DevOps Pipelines:** Part of the Azure DevOps suite.

5. Testing:

- Purpose: To automate testing and ensure code quality.
- Examples:
    - JUnit/NUnit/pytest: Unit testing frameworks for various programming languages.
    - Selenium/Cypress: Automated web browser testing tools.
    - Postman/SoapUI: API testing tools.
    - SonarQube: A code quality and security analysis tool.

6. Design & Prototyping:

- Purpose: To create UI/UX designs and prototypes.
- Examples:
    - Figma/Sketch/Adobe XD: UI/UX design and prototyping tools with collaboration features.
    - In Vision: A prototyping and collaboration platform.

7. Agile Planning & Estimation:

- Purpose: To assist in story mapping, backlog refinement, and sprint planning.
- Examples:
    - Many of the project management tools mentioned above have built-in Agile planning features.
    - Online whiteboarding tools like Miro or Mural can be used for collaborative planning and estimation sessions.
    - Planning Poker apps.

Key Considerations for Tool Selection:

- **Integration:** Ensure that the tools integrate well with each other.
- **Scalability:** Choose tools that can scale as your team and project grow.
- **Ease of Use:** Prioritize tools that are easy to learn and use.
- **Cost:** Consider the cost of tools and choose options that fit your budget.
- **Team Preferences:** Involve the team in the tool selection process.

Introduction to Function-Oriented Software Design:

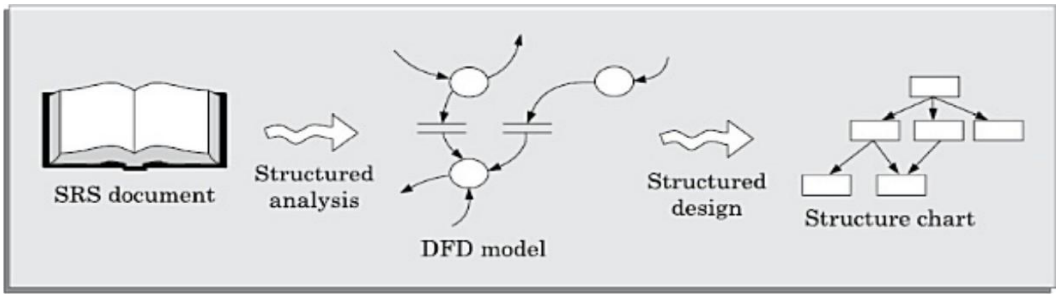
- Function-oriented design techniques were proposed nearly four decades ago.
- still very popular and are currently being used in many software development organizations. These techniques, to start with, view a system as a black-box that provides a set of services to the users of the software.
- These 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:

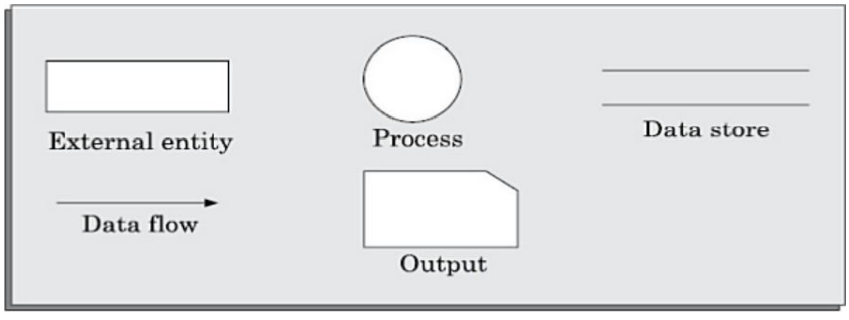
- Top-down decomposition approach.
- Application of divide and conquer principle.
- Through this each high level function is independently decomposed into detailed functions. Graphical representation of the analysis results using data flow diagrams (DFDs).

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.

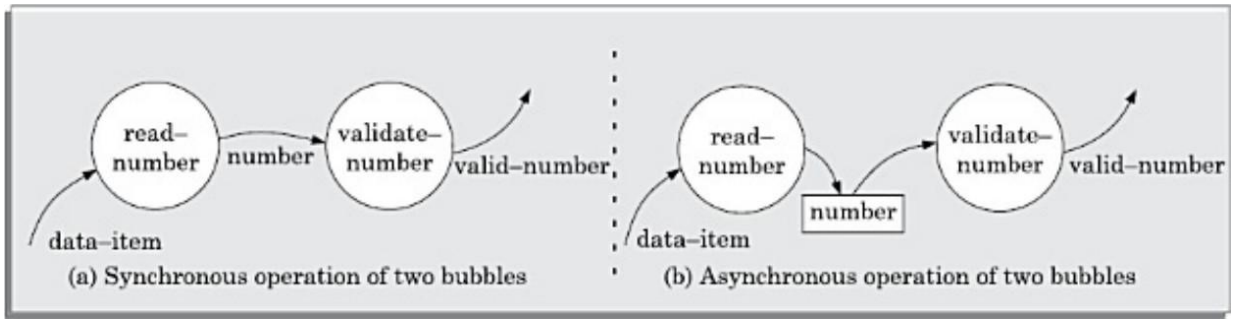


- **Function symbol:** A function is represented using a circle. This symbol is called a process or a bubble. Bubbles are annotated with the names of the corresponding functions
- **External entity symbol:** represented by a rectangle. The external entities are essentially those physical entities external 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.



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.

Self Study:

☐ ***Data Definition***

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.

Self Study: Examples of DFD Models

- ☐ ***RMS Calculator***
- ☐ ***Trading House Automation System***

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.

Flow Chart vs Structure chart:

- 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.

Transformation of a DFD Model into Structure Chart:

- 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

- 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.

- ***Speed of use:***
  - 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.
- ***Speed of recall:***
  - 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.
- ***Error prevention:***
  - 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.
- ***Aesthetic and attractive:***
  - 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.
- ***Consistency:***
  - 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.
- ***Feedback:***
  - 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.
- ***Support for multiple skill levels:***
  - 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.

- ***Error recovery (undo facility):***
  - 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.
- ***User guidance and on-line help:***
  - 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.

Issues in designing a command language based interface:

- 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