UNIT 2

REQURIMENTS ANALYSIS AND SOFTWARE DESIGN

Requirements gathering and analysis

The software requirements are description of features and functionalities of the target system. Requirements convey the expectations of users from the software product. The requirements can be obvious or hidden, known or unknown, expected or unexpected from client's point of view.

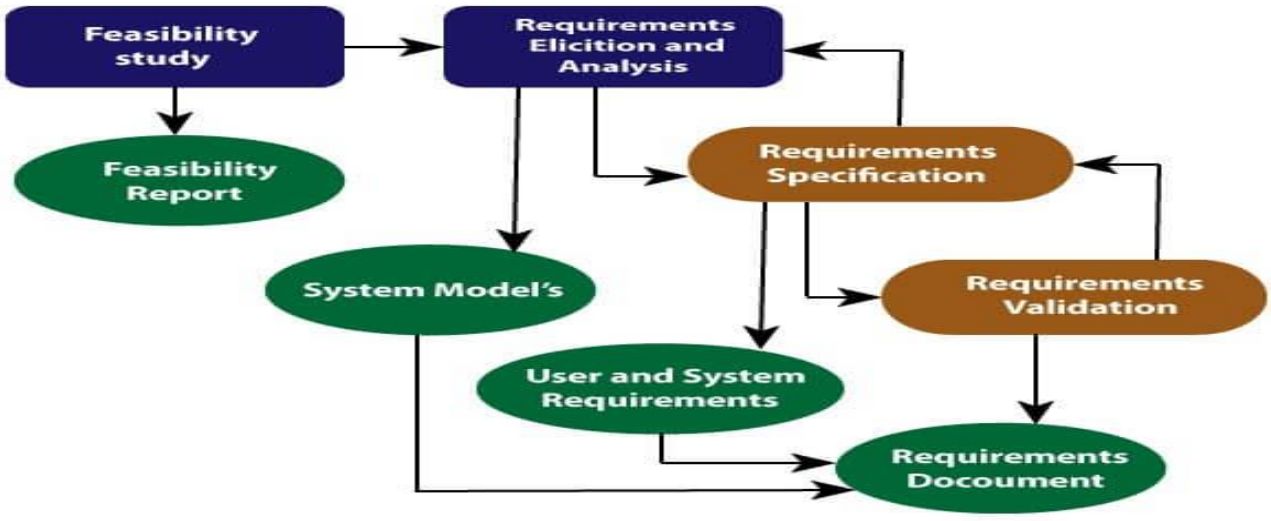
Requirement Engineering

The process to gather the software requirements from client, analyze and document them is known as requirement engineering. The goal of requirement engineering is to develop and maintain sophisticated and descriptive 'System Requirements Specification' document.

Requirement Engineering Process

It is a four step process, which includes –

- Feasibility Study
- Requirement Gathering/ **Elicitation and Analysis**
- Software Requirement Specification
- Software Requirement Validation



Requirement Engineering Process

Feasibility study

When the client approaches the organization for getting the desired product developed, it comes up with rough idea about what all functions the software must perform and which all features are expected from the software.

Referencing to this information, the analysts does a detailed study about whether the desired system and its functionality are feasible to develop.

This feasibility study is focused towards goal of the organization. This study analyzes whether the software product can be practically materialized in terms of implementation, contribution of project to organization, cost constraints and as per values and objectives of the organization. It explores technical aspects of the project and product such as usability, maintainability, productivity and integration ability.

The output of this phase should be a feasibility study report that should contain adequate comments and recommendations for management about whether or not the project should be undertaken.

Types of Feasibility:

1. **Technical Feasibility** - Technical feasibility evaluates the current technologies, which are needed to accomplish customer requirements within the time and budget.
2. **Operational Feasibility** - Operational feasibility assesses the range in which the required software performs a series of levels to solve business problems and customer requirements.
3. **Economic Feasibility** - Economic feasibility decides whether the necessary software can generate financial profits for an organization.

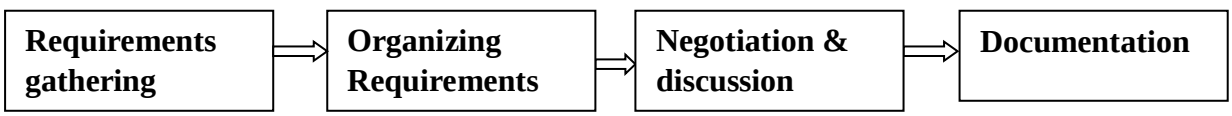
Requirement Elicitation and Analysis:

This is also known as the **gathering of requirements**. Here, requirements are identified with the help of customers and existing systems processes, if available.

If the feasibility report is positive towards undertaking the project, next phase starts with gathering requirements from the user. Analysts and engineers communicate with the client and end-users to know their ideas on what the software should provide and which features they want the software to include.

Requirement Elicitation Process

Requirement elicitation process can be depicted using the folloiwng diagram:



- **Requirements gathering -** The developers discuss with the client and end users and know their expectations from the software.
- **Organizing Requirements -** The developers prioritize and arrange the requirements in order of importance, urgency and convenience.
- **Negotiation & discussion -** If requirements are ambiguous or there are some conflicts in requirements of various stakeholders, if they are, it is then negotiated and discussed with stakeholders. Requirements may then be prioritized and reasonably compromised.

  The requirements come from various stakeholders. To remove the ambiguity and conflicts, they are discussed for clarity and correctness. Unrealistic requirements are compromised reasonably.

- **Documentation -** All formal & informal, functional and non-functional requirements are documented and made available for next phase processing

Software Requirement Specification

SRS is a document created by system analyst after the requirements are collected from various stakeholders.

SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.

The requirements received from client are written in natural language. It is the responsibility of system analyst to document the requirements in technical language so that they can be comprehended and useful by the software development team.

SRS should come up with following features:

- User Requirements are expressed in natural language.
- Technical requirements are expressed in structured language, which is used inside the organization.
- Design description should be written in Pseudo code.
- Format of Forms and GUI screen prints.
- Conditional and mathematical notations for DFDs etc.

Software Requirement Validation

After requirement specifications are developed, the requirements mentioned in this document are validated. User might ask for illegal, impractical solution or experts may interpret the requirements incorrectly. This results in huge increase in cost if not nipped in the bud. Requirements can be checked against following conditions -

- If they can be practically implemented
- If they are valid and as per functionality and domain of software
- If there are any ambiguities
- If they are complete
- If they can be demonstrated

Software Requirement Specification

A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.

Qualities of SRS:

- Correct
- Unambiguous
- Complete
- Consistent
- Ranked for importance and/or stability
- Verifiable
- Modifiable
- Traceable

Types of Requirements:

The below diagram depicts the various types of requirements that are captured during SRS.

A software requirement can be of 3 types:

- Functional requirements
- Non-functional requirements
- Domain requirements

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

For example, in a hospital management system, a doctor should be able to retrieve the information of his patients. Each high-level functional requirement may involve several interactions or dialogues between the system and the outside world. In order to accurately describe the functional requirements, all scenarios must be enumerated.

There are many ways of expressing functional requirements e.g., natural language, a structured or formatted language with no rigorous syntax and formal specification language with proper syntax.

Non-functional requirements: These are basically the quality constraints that the system must satisfy according to the project contract. The priority or extent to which these factors are implemented varies from one project to other. They are also called non-behavioral requirements.
They basically deal with issues like:

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

NFR's are classified into following types:

- Interface constraints
- Performance constraints: response time, security, storage space, etc.
- Operating constraints
- Life cycle constraints: maintainability, portability, etc.
- Economic constraints

The process of specifying non-functional requirements requires the knowledge of the functionality of the system, as well as the knowledge of the context within which the system will operate.

Domain requirements: Domain requirements are the requirements which are characteristic of a particular category or domain of projects. The basic functions that a system of a specific domain must necessarily exhibit come under this category. For instance, in an academic software that maintains records of a school or college, the functionality of being able to access the list of faculty and list of students of each grade is a domain requirement. These requirements are therefore identified from that domain model and are not user specific.

1. Introduction
- 1.1 Purpose
- 1.2 Document conventions
- 1.3 Project scope
- 1.4 References

2. Overall description
- 2.1 Product perspective
- 2.2 User classes and characteristics
- 2.3 Operating environment
- 2.4 Design and implementation constraints
- 2.5 Assumptions and dependencies

3. System features
- 3.x System feature X
  - 3.x.1 Description
  - 3.x.2 Functional requirements

4. Data requirements
- 4.1 Logical data model
- 4.2 Data dictionary
- 4.3 Reports
- 4.4 Data acquisition, integrity, retention, and disposal

5. External interface requirements
- 5.1 User interfaces
- 5.2 Software interfaces
- 5.3 Hardware interfaces
- 5.4 Communications interfaces

6. Quality attributes
- 6.1 Usability
- 6.2 Performance
- 6.3 Security
- 6.4 Safety
- 6.x [others]

7. Internationalization and localization requirements

8. Other requirements

Appendix A: Glossary

Appendix B: Analysis models

Characteristics of an SRS

This section discusses, in brief, some characteristics that describe a good SRS:

Correct – Requirements should be correct and should reflect exactly what the client wants. Every requirement should be such that it is required in the final product.

User review is used to ensure the correctness of requirements stated in the SRS. SRS is said to be correct if it covers all the requirements that are actually expected from the system.

Complete – SRS is said to be complete if everything the software is supposed to do is covered in the document. It should include all the functional and non-functional requirements, the correct numbering of the pages, any diagrams if required.

Unambiguous – All the requirements should have the same interpretation.

Verifiable – SRS is said to be verifiable if and only if each requirement is verifiable; there must be a way to determine whether every requirement is met in the final product.

Consistent – SRS is said to be consistent if there aren't any conflicts between the requirements.

Ranking for importance and stability – Each requirement should be ranked for its importance or stability.

Modifiability – An SRS is said to have modifiability quality if it is capable of adapting changes in the future as much as possible.

Traceability – Tracing of requirements to a design document, particular source code module or test cases should be possible.

Design Independence – SRS should include options of design alternatives for the final system; it should not have any implementation details.

Testability – An SRS is said to be testable if it is easier for the testing team to design test plans, test scenarios, and test cases using an SRS.

Understandable by the customer – An SRS should be written with easy and clear language so that the customer can understand the document

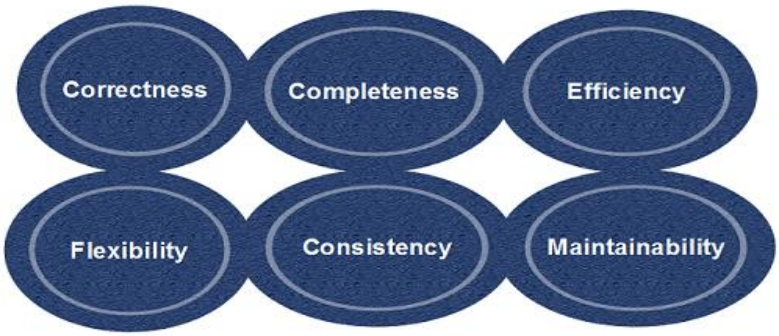
Chapter 2

Software Design / Design process

- Software design is an iterative process through which requirements are translated into the blueprint for building the software.
- Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language.
- The software design phase is the first step in **SDLC (Software Design Life Cycle)**, which moves the concentration from the problem domain to the solution domain. In software design, we consider the system to be a set of components or modules with clearly defined behaviors & boundaries.

Objectives of Software Design

Following are the purposes of Software design:



Objectives of Software Design

1. **Correctness:** Software design should be correct as per requirement.
2. **Completeness:** The design should have all components like data structures, modules, and external interfaces, etc.
3. **Efficiency:** Resources should be used efficiently by the program.
4. **Flexibility:** Able to modify on changing needs.
5. **Consistency:** There should not be any inconsistency in the design.
6. **Maintainability:** The design should be so simple so that it can be easily maintainable by other designers.

Different levels of Software Design:

There are three different levels of software design. They are:

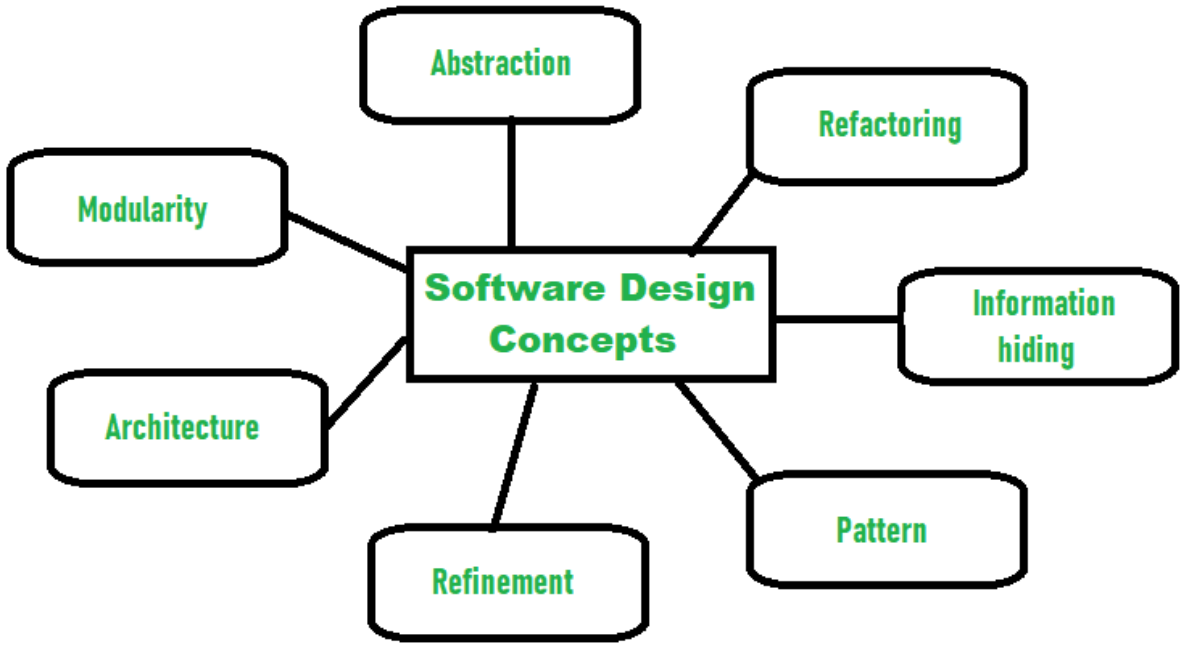
1. **Architectural Design:**
   The architecture of a system can be viewed as the overall structure of the system & the way in which structure provides conceptual integrity of the system. The architectural design identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of the proposed solution domain.

2. **Preliminary or high-level design:**
   Here the problem is decomposed into a set of modules, the control relationship among various modules identified, and also the interfaces among various modules are identified. The outcome of this stage is called the program architecture. Design representation techniques used in this stage are structure chart and UML.

3. **Detailed design:**
   Once the high-level design is complete, a detailed design is undertaken. In detailed design, each module is examined carefully to design the data structure and algorithms. The stage outcome is documented in the form of a module specification document.

Design concepts

Software Design Concepts:
Concepts are defined as a principal idea or invention that comes into our mind or in thought to understand something. The **software design concept** simply means the idea or principle behind the design. It describes how you plan to solve the problem of designing software, the logic, or thinking behind how you will design software. It allows the software engineer to create the model of the system or software or product that is to be developed or built. The software design concept provides a supporting and essential structure or model for developing the right software. There are many concepts of software design and some of them are given below:



The following **points should be considered while designing Software:**

1. **Abstraction- hide Irrelevant data**
   Abstraction simply means to hide the details to reduce complexity and increases efficiency or quality. Different levels of Abstraction are necessary and must be applied at each stage of the design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution. The solution should be described in broad ways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.
2. **Modularity- subdivide the system**
   Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays there are different software available like Monolithic software that is hard to grasp for software engineers. So, modularity in design has now become a trend and is also important. If the system contains fewer components then it would mean the system is complex which requires a lot of effort (cost) but if we are able to divide the system into components then the cost would be small.
3. **Architecture- design a structure of something**
   Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.
4. **Refinement- removes impurities**
   Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is

actually a process of developing or presenting the software or system in a detailed manner that means to elaborate a system or software. Refinement is very necessary to find out any error if present and then to reduce it.

5. **Pattern- a repeated form**
   The pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.
6. **Information Hiding- hide the information**
   Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can't be accessed by any other modules.
7. **Refactoring- reconstruct something**
   Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without affecting the behavior or its functions. Fowler has defined refactoring as "the process of changing a software system in a way that it won't affect the behavior of the design and improves the internal structure".
8. **Functional independence**
   The functional independence is the concept of separation and related to the concept of modularity, abstraction and information hiding.

The functional independence is accessed using two criteria i.e Cohesion and coupling.

Cohesion

- Cohesion is an extension of the information hiding concept.
- A cohesive module performs a single task and it requires a small interaction with the other components in other parts of the program.

Coupling

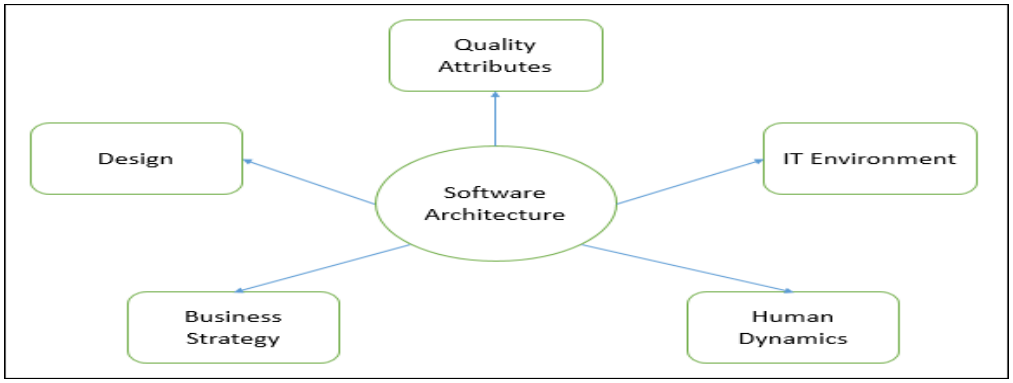
Coupling is an indication of interconnection between modules in a structure of software.

9. Design classes

- The model of software is defined as a set of design classes.
- Every class describes the elements of problem domain and that focus on features of the problem which are user viqaazssible.

Architectural Concepts

The architecture of a system describes its major components, their relationships (structures), and how they interact with each other. Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.



We can segregate Software Architecture and Design into two distinct phases: Software Architecture and Software Design. In **Architecture**, non-functional decisions are cast and separated by the functional requirements. In Design, functional requirements are accomplished.

- ✓ The design of software architecture considers two levels of the design

  1. data design

  2. architectural design.

- ✓ **Data design** enables us to represent the data component of the architecture.
- ✓ **Architectural design** focuses on the representation of the structure of software components, their properties, and interactions.

Why Is Architecture Important?

1. Representations of software architecture are an enabler for **communication between all stakeholders** interested in the development of a computer-based system.
2. The architecture **highlights early design decisions** that will have a profound impact on all software engineering work
3. Architecture "**constitutes a relatively small**, intellectually graspable model of how the system is structured and how its components work together"

ARCHITECTURAL STYLES

- The software that is built for **computer-based systems** also exhibits one of many architectural styles.
- **Each style describes a system category that encompasses**

(1) A set of **components** that perform a function required by a system;

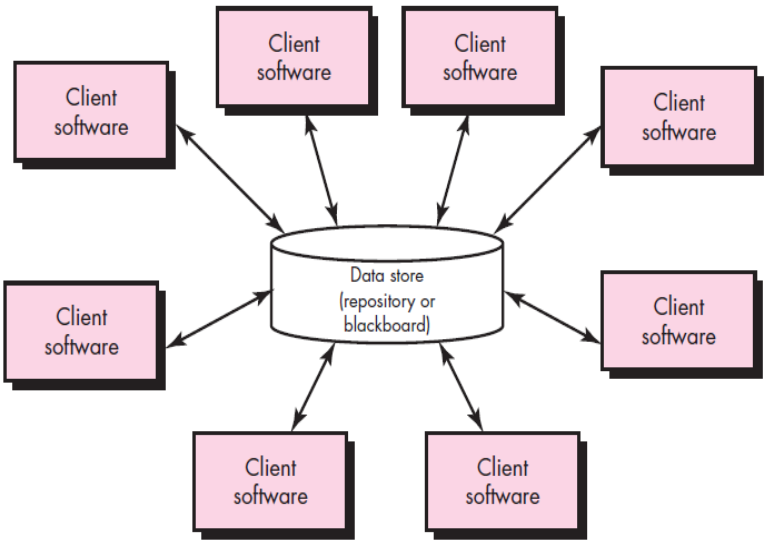
(2) A set of **connectors** that enable "communication, coordination's and cooperation" among components;

(3) **Constraints** that define how components can be integrated to form the system;

(4) **Semantic models** that enable a designer to understand the overall properties of a system.

Data-centered architectures:

- A **data store** (e.g., a file or database) resides at the **center of this architecture** and is accessed frequently by other components that **update, add, delete,** or otherwise **modify data within the store.**
- **Fig: data centered architecture**

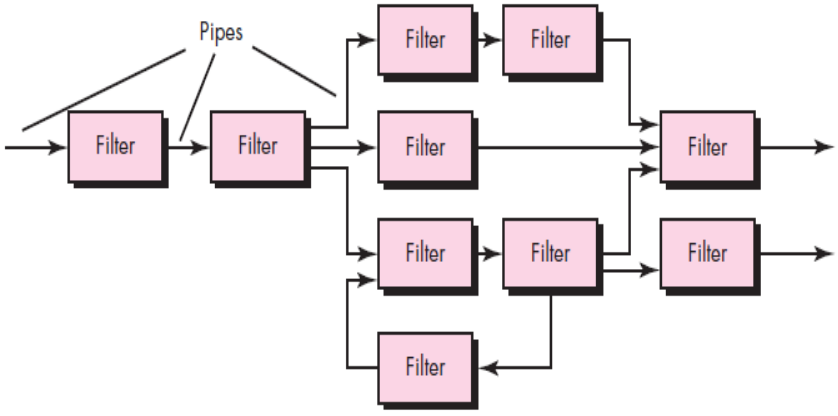


- A variation on this approach transforms the repository into a "**blackboard**" that sends notification to client software when data of interest to the client changes.
- Data-centered architectures promote ***integrability***.

  (i.e. existing components can be changed and new client components can be added to the architecture without concern about other clients)
- In addition, Data can be passed among clients using the **blackboard mechanism**.

Data-flow architectures:

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.

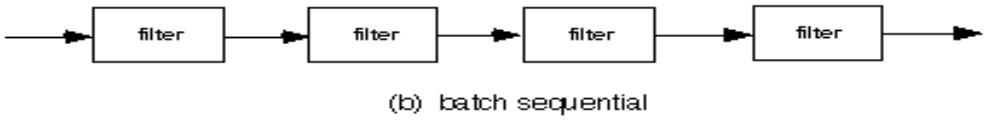


Pipes and filters

Fig(a) : Data Flow Architecture

- A *pipe-and-filter* pattern has a set of components, called ***filters***, connected by ***pipes*** that transmit data from one component to the next.
- Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- The filter does not require knowledge of the workings of its neighboring filters.

If the data flow degenerates into a single line of transforms, it is termed ***batch sequential***.



(b) batch sequential

Call and return architectures:

- This architectural style enables you to achieve a program structure that is relatively easy to modify and scale.
- A number of substyles exist within this category:

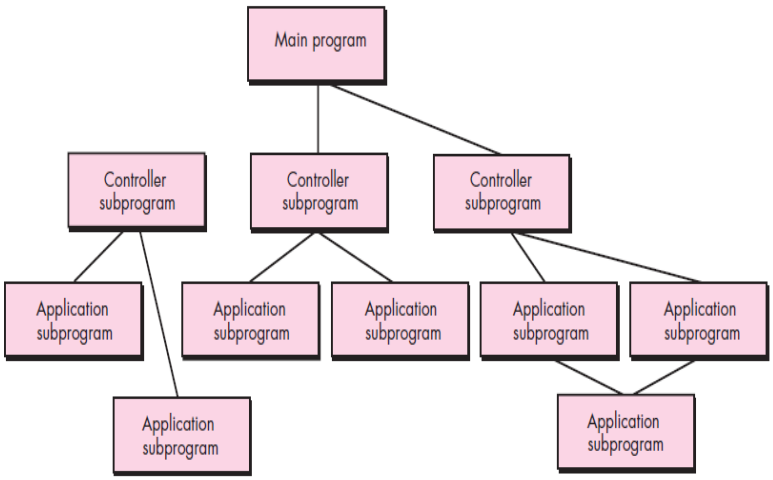


Fig: Main program/subprogram Architecture

i) Main program/subprogram architectures.

This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components.

ii) Remote procedure call architectures.

The components of a main program/ subprogram architecture are distributed across multiple computers on a network

Object-oriented architectures:

- ✓ The components of a system encapsulate data and the operations that must be applied to manipulate the data.
- ✓ Communication and coordination between components is accomplished via message passing.

Layered architectures:

- ✓ The basic structure of a layered architecture is illustrated in Figure:

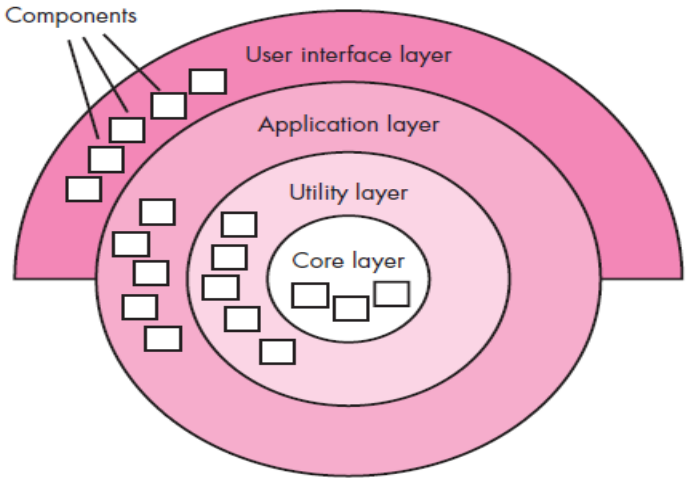


Fig: layered Architecture

- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the **outer layer**, components service user interface operations.
- At the **inner layer**, components perform operating system interfacing.
- **Intermediate layers** provide utility services and application software functions.

Architectural Patterns

- An ***architectural pattern,*** like an architectural style, imposes a transformation the design of architecture.
- A *architectural* pattern differs from a *architectural* style in a number of fundamental ways:
  - i. The **scope of a pattern is less broad**, focusing on one aspect of the architecture rather than the architecture in its entirety.
  - ii. **A pattern imposes a rule on the architecture**, describing how the software will handle some aspect of its functionality at the infrastructure level.

iii. Architectural patterns tend to **address specific behavioral issues** within the context of the architectural.

The architectural pattern has some characteristics:

i. **Concurrency—**applications must handle multiple tasks in a manner that simulates parallelism

- *operating system process management* pattern
- *task scheduler* pattern

ii. **Persistence—**Data persists if it survives past the execution of the process that created it. Two patterns are common:

- a ***database management system*** pattern that applies the storage and retrieval capability of a DBMS to the application architecture
- an ***application level persistence*** pattern that builds persistence features into the application architecture

iii. **Distribution—** the manner in which systems or components within systems communicate with one another in a distributed environment

- A ***broker*** acts as a 'middle-man' between the client component and a server component**.**