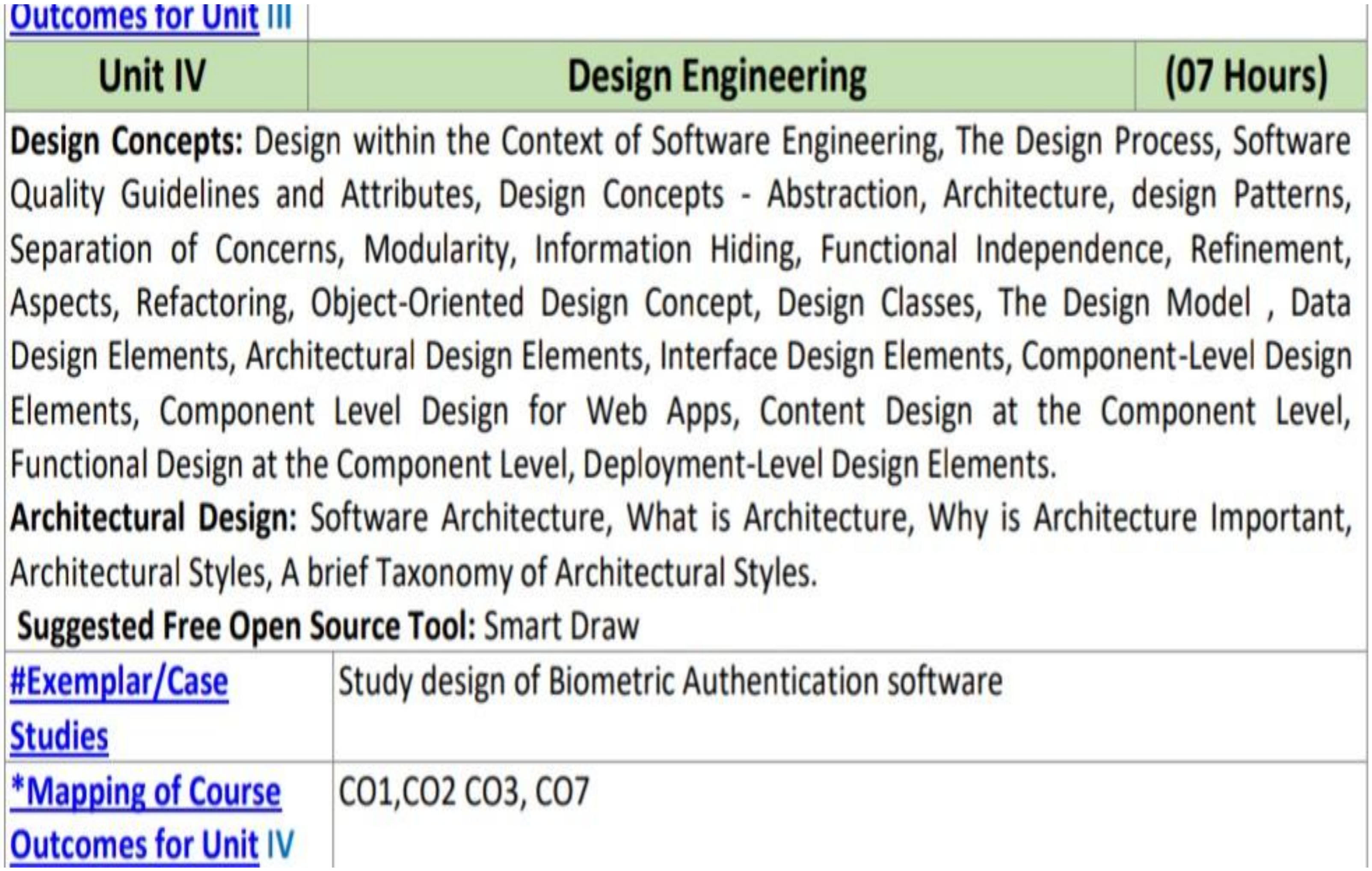
Unit-IV
Design Engineering

Presented by,
Prof. Barkha M Shahaji
Department of Computer Engineering,
Trinity College of Engineering and research, Pune-48.

Syllabus



Outcomes for Unit III

Unit IV	Design Engineering	(07 Hours)
Design Concepts: Design within the Context of Software Engineering, The Design Process, Software Quality Guidelines and Attributes, Design Concepts - Abstraction, Architecture, design Patterns, Separation of Concerns, Modularity, Information Hiding, Functional Independence, Refinement, Aspects, Refactoring, Object-Oriented Design Concept, Design Classes, The Design Model , Data Design Elements, Architectural Design Elements, Interface Design Elements, Component-Level Design Elements, Component Level Design for Web Apps, Content Design at the Component Level, Functional Design at the Component Level, Deployment-Level Design Elements. **Architectural Design:** Software Architecture, What is Architecture, Why is Architecture Important, Architectural Styles, A brief Taxonomy of Architectural Styles. **Suggested Free Open Source Tool:** Smart Draw		
#Exemplar/Case Studies	Study design of Biometric Authentication software	
*Mapping of Course Outcomes for Unit IV	CO1,CO2 CO3, CO7	

Basic of Software Design

- Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

- 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 the design phase all the relevant entities such as the customer, business requirements and technical considerations collaborate to formulate a product or a system.

Software Quality Attributes

Qualities of good Software design:

1. Innovative
2. Functional
3. Honest
4. User-oriented
5. Correctness

1. Innovative

- Innovative design can be either completely new design or redesign of existing product.
- New design gives unseen value to market where as redesign improves the quality of an existing product.

2. functional:

- Good design fulfils all its intended functions to solve user’s problem.
- It focuses on usefulness of a product by optimizing its functionality.

3.Honest:

- A good design is honest. An honest design expresses the functions and values it offers. It never attempts to modify Organizer’s and User’s view with promises it can’t keep.

4.User oriented:

- Good design is developed based on its use and intended to improve solutions to problem for is user. User oriented design gives intellectual as well as material value to system which in turn achieves user’s satisfaction.

5. Correctness:

- Correctness is an important quality of good design. A good design should correctly achieve all required functionalities as per SRS document.

Software Design Principles

1. Should not suffer from “Tunnel Vision” –
While designing the process, it should not suffer from “tunnel vision” which means that is should not only focus on completing or achieving the aim but on other effects also.

2. Traceable to analysis model –
The design process should be traceable to the analysis model which means it should satisfy all the requirements that software requires to develop a high-quality product.

3. Should not “Reinvent The Wheel” –
The design process should not reinvent the wheel that means it should not waste time or effort in creating things that already exist. Due to this, the overall development will get increased.

4. Minimize Intellectual distance –
The design process should reduce the gap between real-world problems and software solutions for that problem meaning it should simply minimize intellectual distance.

5. Exhibit uniformity and integration –
The design should display uniformity which means it should be uniform throughout the process without any change. Integration means it should mix or combine all parts of software i.e. subsystems into one system.

6. Accommodate change –
The software should be designed in such a way that it accommodates the change implying that the software should adjust to the change that is required to be done as per the user's need.

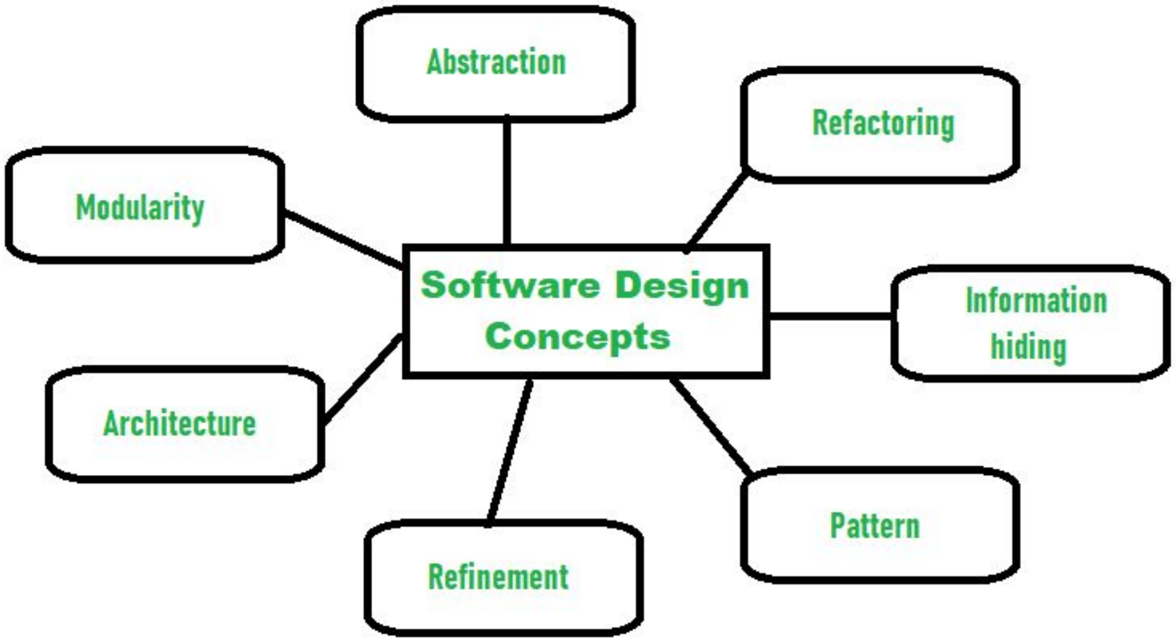
7. Degrade gently –
The software should be designed in such a way that it degrades gracefully which means it should work properly even if an error occurs during the execution.

8. Assessed or quality –
The design should be assessed or evaluated for the quality meaning that during the evaluation, the quality of the design needs to be checked and focused on.

9. Review to discover errors –
The design should be reviewed which means that the overall evaluation should be done to check if there is any error present or if it can be minimized.

10. Design is not coding and coding is not design –
Design means describing the logic of the program to solve any problem and coding is a type of language that is used for the implementation of a design.

Design Concepts



1. Abstraction

An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation. Here, there are two common abstraction mechanisms

1. Functional Abstraction
2. Data Abstraction

1. Functional Abstraction

i. It includes the use of parameterized subprograms.
ii. It can be created as a set of subprograms known as groups

Functional abstraction forms the basis for **Function oriented design approaches**.

2. Data Abstraction

Details of the data elements are not visible to the users of data. Data Abstraction forms the basis for **Object Oriented design approaches**.

2. Modularity

- Modularity specifies to the division of software into separate modules which are differently named and addressed and are integrated later on in to obtain the completely functional software.
- Single large programs are difficult to understand and read due to a large number of reference variables, control paths, global variables, etc.

The desirable properties of a modular system are:

1. Each module is a well-defined system that can be used with other applications.
2. Each module has single specified objectives.
3. Modules can be separately compiled and saved in the library.
4. Modules should be easier to use than to build.
5. Modules are simpler from outside than inside.

3. Information hiding

- The fundamental of Information hiding suggests that modules can be characterized by the design decisions that protect from the others,
- i.e., In other words, modules should be specified that data include within a module is inaccessible to other modules that do not need for such information.
- The use of information hiding as design criteria for modular system provides the most significant benefits when modifications are required during testing's and later during software maintenance.

4. Architecture

Software architecture refers to the structure of the system, which is composed of various components of a program/ system, the attributes (properties) of those components and the relationship amongst them. The software architecture enables the software engineers to analyze the software design efficiently. In addition, it also helps them in decision-making and handling risks. The software architecture does the following.

- Provides an insight to all the interested stakeholders that enable them to communicate with each other
- Highlights early design decisions, which have great impact on the software engineering activities (like coding and testing) that follow the design phase
- Creates intellectual models of how the system is organized into components and how these components interact with each other.

5. Patterns

A pattern provides a description of the solution to a recurring design problem of some specific domain in such a way that the solution can be used again and again. The objective of each pattern is to provide an insight to a designer who can determine the following.

- Whether the pattern can be reused
- Whether the pattern is applicable to the current project
- Whether the pattern can be used to develop a similar but functionally or structurally different design pattern.

6. Stepwise Refinement

- Stepwise refinement is a top-down design strategy used for decomposing a system from a high level of abstraction into a more detailed level (lower level) of abstraction.
- At the highest level of abstraction, function or information is defined conceptually without providing any information about the internal workings of the function or internal structure of the data.
- As we proceed towards the lower levels of abstraction, more and more details are available.

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 impacting the behavior or its functions.
- Fowler has defined refactoring as “the process of changing a software system in a way that it won’t impact the behavior of the design and improves the internal structure”.

Functional independence

- Functional independence is achieved by developing functions that perform only one kind of task and do not excessively interact with other modules.
- Independence is important because it makes implementation more accessible and faster.
- The independent modules are easier to maintain, test, and reduce error propagation and can be reused in other programs as well.
- Thus, functional independence is a good design feature which ensures software quality.

It is measured using two criteria:

- Cohesion: It measures the relative function strength of a module.
- Coupling: It measures the relative interdependence among modules.

Cohesion

- In computer programming, cohesion defines to the degree to which the elements of a module belong together.
- Thus, cohesion measures the strength of relationships between pieces of functionality within a given module.
- For example, in highly cohesive systems, functionality is strongly related. Cohesion is an ordinal type of measurement and is generally described as "high cohesion" or "low cohesion."
- Good system design must have high cohesion between the components of system

Different types of cohesion:

1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.
2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.
3. **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.
4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.

5. Temporal Cohesion: When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.
6. Logical Cohesion: A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.
7. Coincidental Cohesion: Unplanned cohesion which results into decomposition of a program into smaller components for the modularization of the system is called Coincidental Cohesion. Typically these type of cohesions are never accepted.

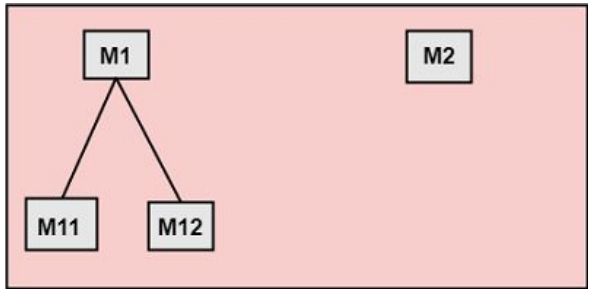
Coupling

- In software engineering, the coupling is the degree of interdependence between software modules.
- Two modules that are tightly coupled are strongly dependent on each other.
- However, two modules that are loosely coupled are not dependent on each other.
- Uncoupled modules have no interdependence at all within them.

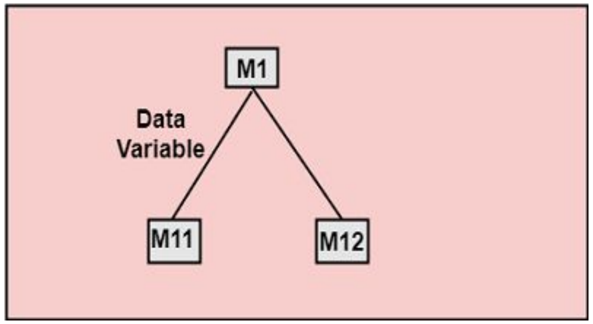
Types of Module Coupling

1. No Direct Coupling
2. Data Coupling
3. Stamp Coupling
4. Control Coupling
5. External Coupling
6. Common Coupling
7. Content Coupling:

1. **No Direct Coupling:** There is no direct coupling between M1 and M2. In this case, modules are subordinates to different modules. Therefore, no direct coupling.



2. **Data Coupling:** When data of one module is passed to another module, this is called data coupling.

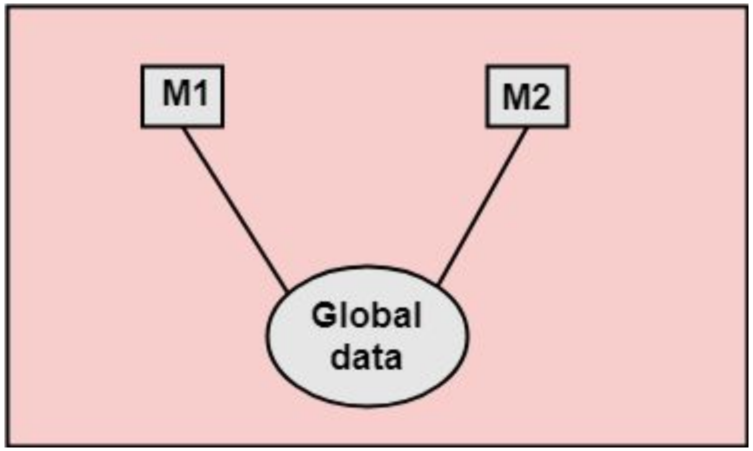


3. Stamp Coupling: Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.

4. Control Coupling: Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

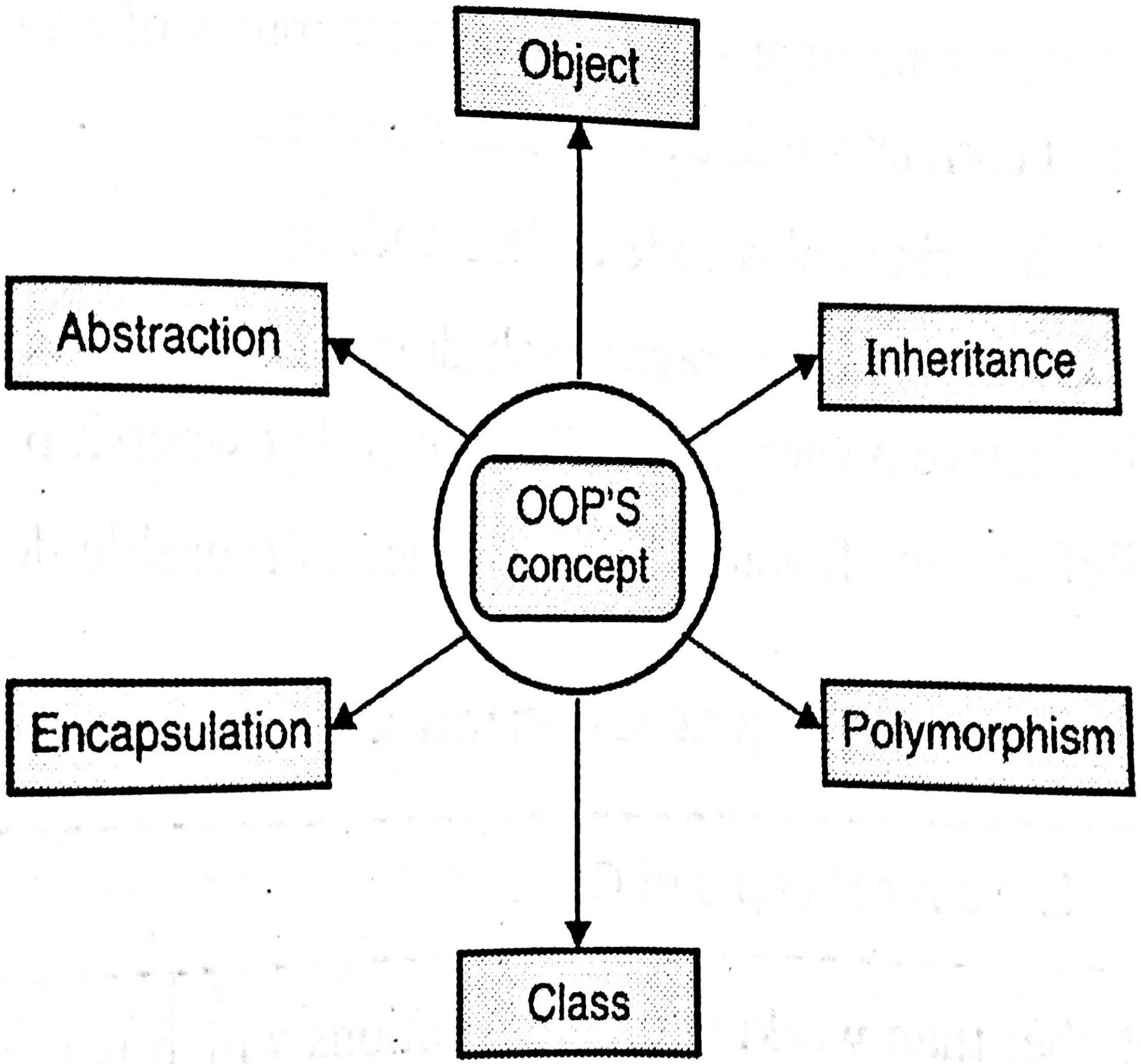
5. External Coupling: External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

6. Common Coupling: Two modules are common coupled if they share information through some global data items.



7. Content Coupling: Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

Object oriented Design



The different terms related to object design are:

1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.

2. **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.

3. **Encapsulation:** Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.

4. Inheritance: OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses.This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.

5. Polymorphism: OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types. Depending upon how the service is invoked, the respective portion of the code gets executed.

6. Abstraction In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.

The Design Model

Software design model consists of 4 designs:

1. Data/Class design
2. Architectural design
3. Interface design
4. Component design

Data Design Elements

- The data design element produced a model of data that represent a high level of abstraction.

- This model is then more refined into more implementation specific representation which is processed by the computer based system.

- The structure of data is the most important part of the software design.

Architectural Design Elements

- The architecture design elements provides us overall view of the system.

- The architectural design element is generally represented as a set of interconnected subsystem.

- Requirements of a software should be transformed into an architecture that describes the software’s top level structure and recognizes its components.

- Dynamic Model- This model shows how the structure or system configuration changes as the function changes because of the change in external environment.

- Process Model- This model concentrates on the design of the business or technical process, which must be implemented in the system.

- Functional Model- This model signifies the functional hierarchy of a system

Interface Design Elements

- The interface design elements for software represents the information flow within it and out of the system.

- They communicate between the components defined as part of architecture.

- **Following are the important elements of the interface design:**
  1. The user interface
  2. The external interface to the other systems, networks etc.
  3. The internal interface between various components.

Types of User Interface

Command Line Interface: The Command Line Interface provides a command prompt, where the user types the command and feeds it to the system. The user needs to remember the syntax of the command and its use.

Graphical User Interface: Graphical User Interface provides a simple interactive interface to interact with the system. GUI can be a combination of both hardware and software. Using GUI, the user interprets the software.

Principles of user interface design:

1. **Structure:** Design should organize the user interface purposefully, in the meaningful and usual based on precise, consistent models that are apparent and recognizable to users, putting related things together and separating unrelated things, differentiating dissimilar things and making similar things resemble one another. The structure principle is concerned with overall user interface architecture.
2. **Simplicity:** The design should make the simple, common task easy, communicating clearly and directly in the user's language, and providing good shortcuts that are meaningfully related to longer procedures.

3. Visibility: The design should make all required options and materials for a given function visible without distracting the user with extraneous or redundant data.
4. Feedback: The design should keep users informed of actions or interpretation, changes of state or condition, and bugs or exceptions that are relevant and of interest to the user through clear, concise, and unambiguous language familiar to users.
5. Tolerance: The design should be flexible and tolerant, decreasing the cost of errors and misuse by allowing undoing and redoing while also preventing bugs wherever possible by tolerating varied inputs and sequences and by interpreting all reasonable actions.

The golden rules

1. Strive for consistency :
by utilizing familiar icons, colors, menu hierarchy, call-to-actions, and user flows when designing similar situations and sequence of actions. Standardizing the way information is conveyed ensures users are able to apply knowledge from one click to another; without the need to learn new representations for the same actions. Consistency plays an important role by helping users become familiar with the digital landscape of your product so they can achieve their goals more easily.

2. Enable frequent users to use shortcuts.

With increased use comes the demand for quicker methods of completing tasks. For example, both Windows and Mac provide users with keyboard shortcuts for copying and pasting, so as the user becomes more experienced, they can navigate and operate the user interface more quickly and effortlessly.

3. Offer informative feedback.

The user should know where they are at and what is going on at all times. For every action there should be appropriate, human-readable feedback within a reasonable amount of time. A good example of applying this would be to indicate to the user where they are at in the process when working through a multi-page questionnaire. A bad example we often see is when an error message shows an error-code instead of a human-readable and meaningful message.

4. Design dialogue to yield closure.

Don’t keep your users guessing. Tell them what their action has led them to. For example, users would appreciate a “Thank You” message and a proof of purchase receipt when they’ve completed an online purchase.

5. Offer simple error handling.

No one likes to be told they’re wrong, especially your users. Systems should be designed to be as fool-proof as possible, but when unavoidable errors occur, ensure users are provided with simple, intuitive step-by-step instructions to solve the problem as quickly and painlessly as possible. For example, flag the text fields where the users forgot to provide input in an online form.

6. Permit easy reversal of actions.

Designers should aim to offer users obvious ways to reverse their actions. These reversals should be permitted at various points whether it occurs after a single action, a data entry or a whole sequence of actions.

7. Support internal locus of control

Allow your users to be the initiators of actions. Give users the sense that they are in full control of events occurring in the digital space. Earn their trust as you design the system to behave as they expect.

8. Reduce short-term memory load.

Human attention is limited and we are only capable of maintaining around five items in our short-term memory at one time. Therefore, interfaces should be as simple as possible with proper information hierarchy, and choosing recognition over recall. Recognizing something is always easier than recall because recognition involves perceiving cues that help us reach into our vast memory and allowing relevant information to surface. For example, we often find the format of multiple choice questions easier than short answer questions on a test because it only requires us to recognize the answer rather than recall it from our memory.

UI design issues:

1. Prioritizing library organization over design
2. Not testing the website design enough
3. Inconsistent design
4. Focusing too strongly on standing out rather than on usability
5. Confusing navigation
6. Too Many Words
7. Putting style over substance
8. Failing to address your target users' needs
9. Forgetting to be inclusive
10. Following design trends blindly
11. Insufficient feedback
12. Response time
13. Error handling
14. Menus and command labeling

Component Design Elements

- The main use function of component level design is to define data structures,algorithms,interface characteristics as well as a communication mechanisms for all the present software components which are identified in the phase of architectural design.

- The component level design for the software completely describes the internal details of the each software component.

- The processing of data structure occurs in a component and an interface which allows all the component operations.

- In a context of object-oriented software engineering, a component shown in a UML diagram.

- The UML diagram is used to represent the processing logic.

Principles for designing class based components

- **Open Closed Principle(OCP):**

Extensions should be allowed in the class but not modification.

- **Liskov Substition Principle(LSP):**

It should be possible to substitute subclasses for their base classes.

- **Dependency Inversion Principle(DIP):**

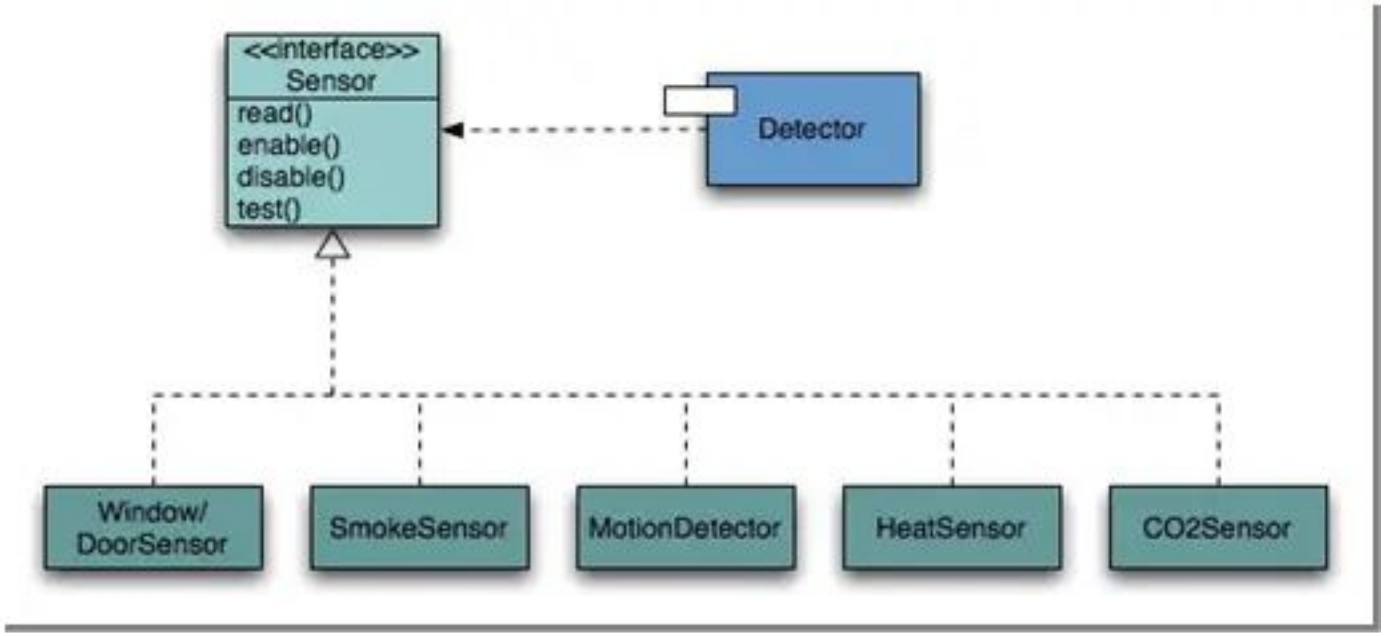
Depend on abstractions, do not depend on concretions.

- **Interface Segregation Principle(ISP):**

Multiple client oriented interfaces are considered as better compared to one general purpose interface.

Open Closed Principle(OCP):

- A module or component should be open for extension but closed for modification
- The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications to the existing parts of the component

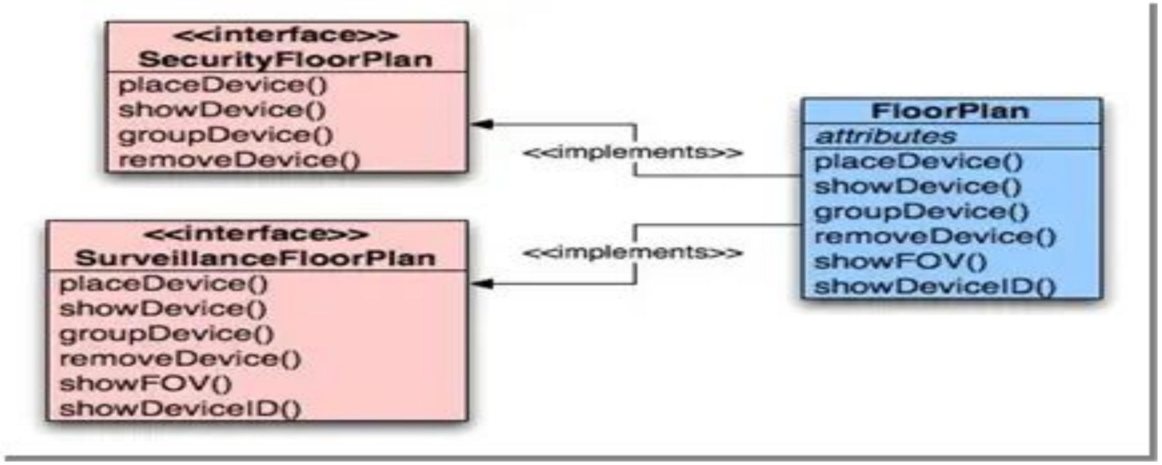


Liskov substitution principle (LSP)

–Subclasses should be <u>substitutable</u> for their base classes
–A component that uses a base class should continue to <u>function properly</u> if a subclass of the base class is passed to the component instead

Interface segregation principle

–<u>Many</u> client-specific <u>interfaces</u> are better than one general purpose interface
–For a server class, <u>specialized interfaces</u> should be created to serve major categories of clients
–Only those operations that are <u>relevant</u> to a particular category of clients should be <u>specified</u> in the interface



- **Release Reuse Equivalency Principle(REP):**

The granule of reuse is the granule of release.

- **Common Closure Principle(CCP):**

Classes that change together belong together.

- **Common Reuse Principle(CRP):**
- Classes that can’t be used together should not be grouped together.

Guidelines for Component Design

Components should have a loose coupling, meaning little interdependence. Loosely coupled components can be changed or replaced independently without affecting other components. Some ways to achieve loose coupling include:

- Defining clear interfaces that encapsulate implementation details.
- Using asynchronous messaging instead of direct function calls.
- Avoiding global variables and sharing data only through interfaces.

Components should have a high cohesion, focusing on a single, well-defined purpose. High cohesion makes components easier to understand, maintain, test, and reuse. Some tips for high cohesion include:

- Assigning components logical and closely related responsibilities.
- Not combining unrelated responsibilities in the same component.
- Naming components clearly and consistently based on their purpose.

Components should provide abstraction, hiding their internal implementation details behind interfaces. This allows components to be used without understanding their inner workings. Some examples of abstraction include:

- Defining interfaces that specify the services a component provides without specifying how they work.
- Using access modifiers like “private” to hide implementation details.
- Choosing descriptive names that convey a component’s purpose without revealing implementation.

Components should be designed with reusability in mind. Reusable components reduce duplication and improve maintainability. To design reusable components:

- Make them self-contained, independent, and focused on a single purpose.
- Define a clear interface and keep implementation details private.
- Avoid assumptions about the context the component will be used in.
- Make the component flexible and customizable through configuration rather than modification.

WebApp design Principles

1. Anticipation: A Webapp should be designed in such as manner that it should be able to anticipates the use its next move
2. Communication: the interface should be designed in such as manner that it should be able to communicate the status of any activity which has been initiated by the user
3. Consistency: The color,shape,layout of navigation controls, menus,icons must be consistent throughout the application.
4. Controlled autonomy: The interface should be designed in such a manner that it should be able to facilitate user movement throughout the WebApp,but it must be implemented in such a manner which enforces navigation conventions that have been set for the application

5. Efficiency: The design of the WebApp and its interface should optimize the user's work efficiency of the web engineer who designs and builds it or the client-server environment that executes it.

6. Focus: The WebApp interface along with its content has responsibility o stay focused on the user.

7. Latency reduction: The WebApp should be able to implement multitasking in a manner that allows the user proceed with work as if the operation has been completed.

8. Learnability:A WebApp should be designed in such manner that it should be able to minimize learning time and once learned to lessoen the amount of relearning needed when the WebApp is revisited.

9. Maintain work product integrity: A work product such as online form filled by user should be automatically saved so that it will not be lost in case of any error.

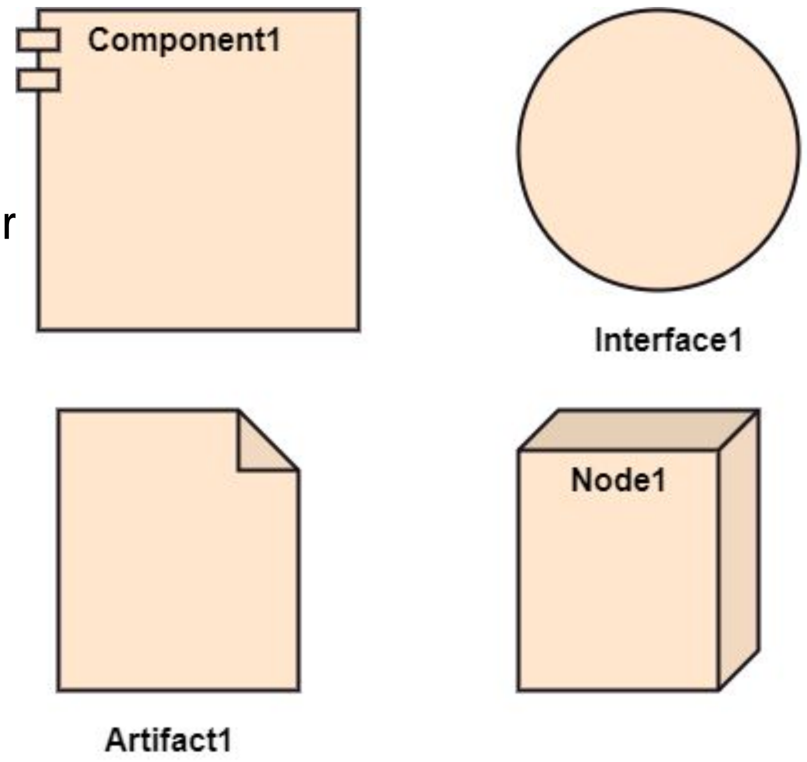
10. Readability: Persons of all ages should be able to read the information presented through the interface.

11. Track state: When suitable the user state regarding her interaction must be tracked as well as stored so that a user can logoff and come back later to continue from where she left off.

Deployment level Design Elements

- A deployment diagram is a diagram which displays the configuration of run time processing nodes and the elements that live on them.
- The deployment diagram visualizes the physical hardware on which the software will be deployed. It portrays the static deployment view of a system. It involves the nodes and their relationships.
- The main purpose of the deployment diagram is to represent how software is installed on the hardware component. It depicts in what manner a software interacts with hardware to perform its execution

The deployment diagr



1. Artifact: A product developed by the software, symbolized by a rectangle with the name and the word “artifact” enclosed by double arrows.
You can use artifacts to represent.
- an executable file
- a framework used during software development
- a source file
- an output file
- Documentation

2. Association: A line that indicates a message or other type of communication between nodes.
3. Component: This notation represents other software elements present in the system. A rectangle with two tabs that indicates a software element. For instance, a bank application running on an Android device is a component of the node (Android device).

4. Dependency: A dashed line that ends in an arrow, which indicates that one node or component is dependent on another.
5. Interface: A circle that indicates a contractual relationship. Those objects that realize the interface must complete some sort of obligation.
6. Node: A hardware or software object, shown by a three-dimensional box.
7. Node as container: A node that contains another node inside of it—such as in the example below, where the nodes contain components.
8. Stereotype: A device contained within the node, presented at the top of the node, with the name bracketed by double arrows. For example, the stereotype for Java Machine is <<Java Machine>>.

Software Architecture

- Software architecture refers to the structure of the system, which consists of several components regarding a program / system, the attributes (properties) of those components and the relationship among them.

- The software architecture helps the software engineers in the process of analyzing the software design proficiently.

Architectural style

- Architectural styles describe a set of systems which are internally linked with each other and share structural and semantic properties.
- In short, the purpose of using architectural styles is to set up a structure for all the components present in a system.

Every architectural style defines a system type that includes the following:

a. Components

b. Connectors

c. Constraints

d. semantic model

The design categories of architectural styles includes:

1. A set of components such as database, computational modules which perform the function required by the system.
2. A set of connectors that allows the communication, coordination and cooperation between the components.
3. The constraints which define the integration of components to form the system.
4. Semantic model allows a designer to understand the overall properties of a system by using analysis of elements.

Architectural styles

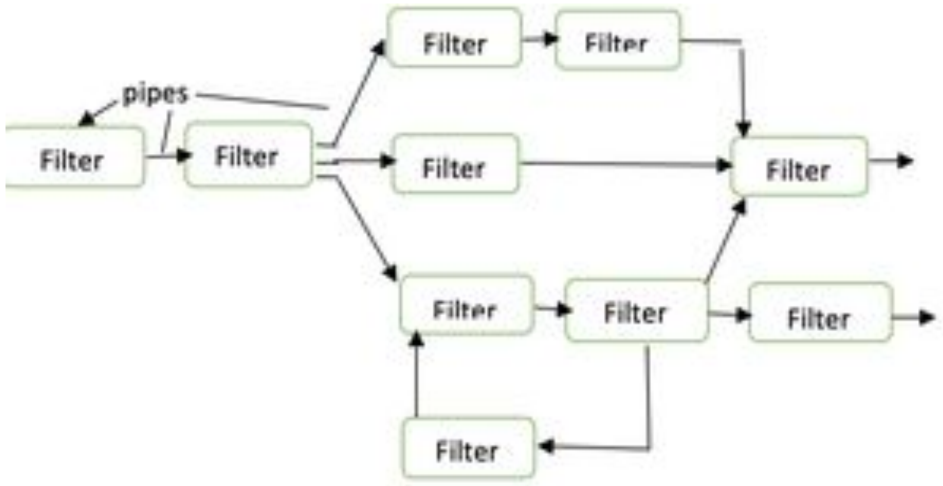
- The architectural styles that are used while designing the software as follows:

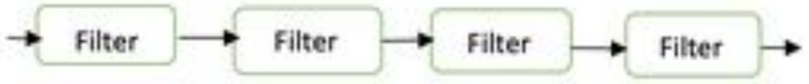
1. Data-flow architecture
2. Object-oriented architectures
3. Layered architectures
4. Data-centered architecture
5. Call and return architectures

1. Data-flow architecture

- This kind of architecture is used when input data is transformed into output data through a series of computational manipulative components.
- The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by lines.
- Pipes are used to transmitting data from one component to the next.
- Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don’t require any knowledge of the working of neighboring filters.
- If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.

Data Flow architecture





2. Object-oriented architectures

The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.

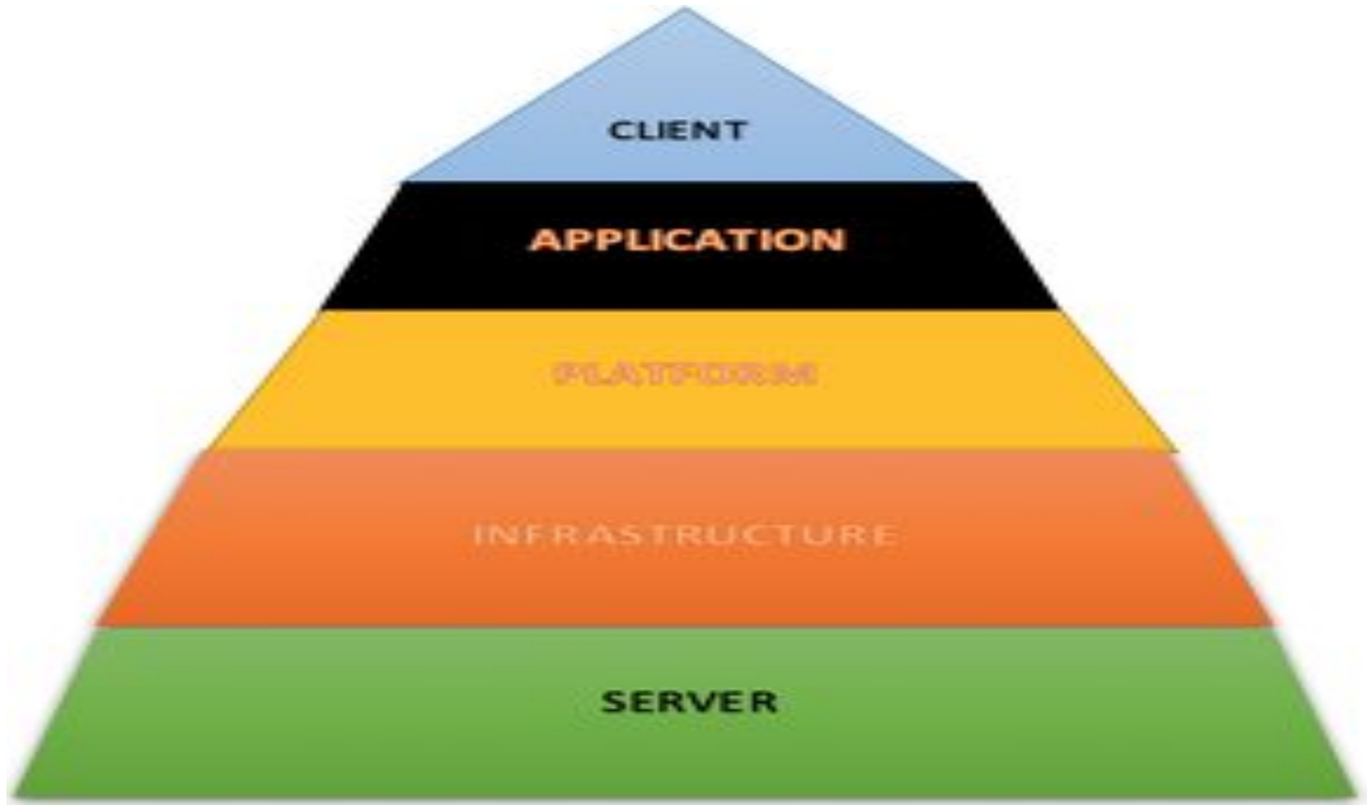
Characteristics of Object Oriented architecture

- Object protect the system's integrity.
- An object is unaware of the depiction of other items.

3. Layered architectures

- A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively.
- At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing(communication and coordination with OS)
- Intermediate layers to utility services and application software functions.
- One common example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organisation for Standardisation) communication system.

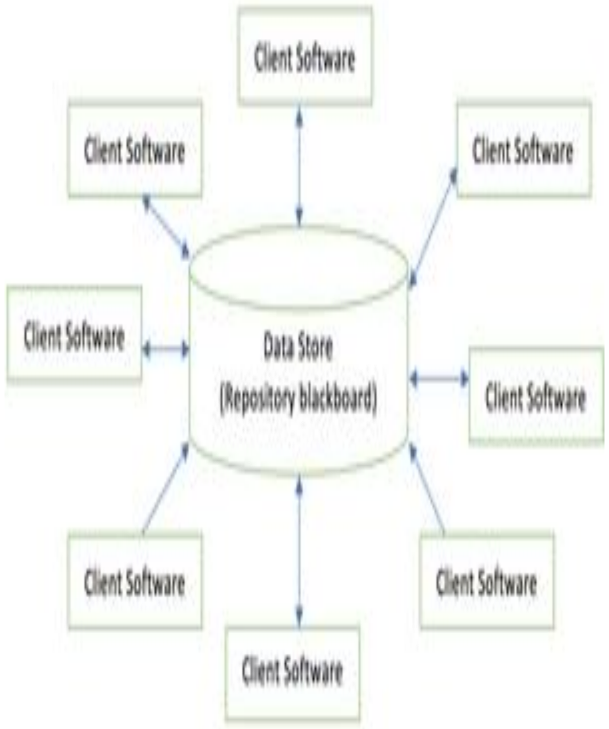
Layered architecture:



4.Data centered architectures:

- A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete or modify the data present within the store.
- The figure illustrates a typical data centered style. The client software access a central repository. Variation of this approach are used to transform the repository into a blackboard when data related to client or data of interest for the client change the notifications to client software.
- This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.

Data centered architectures

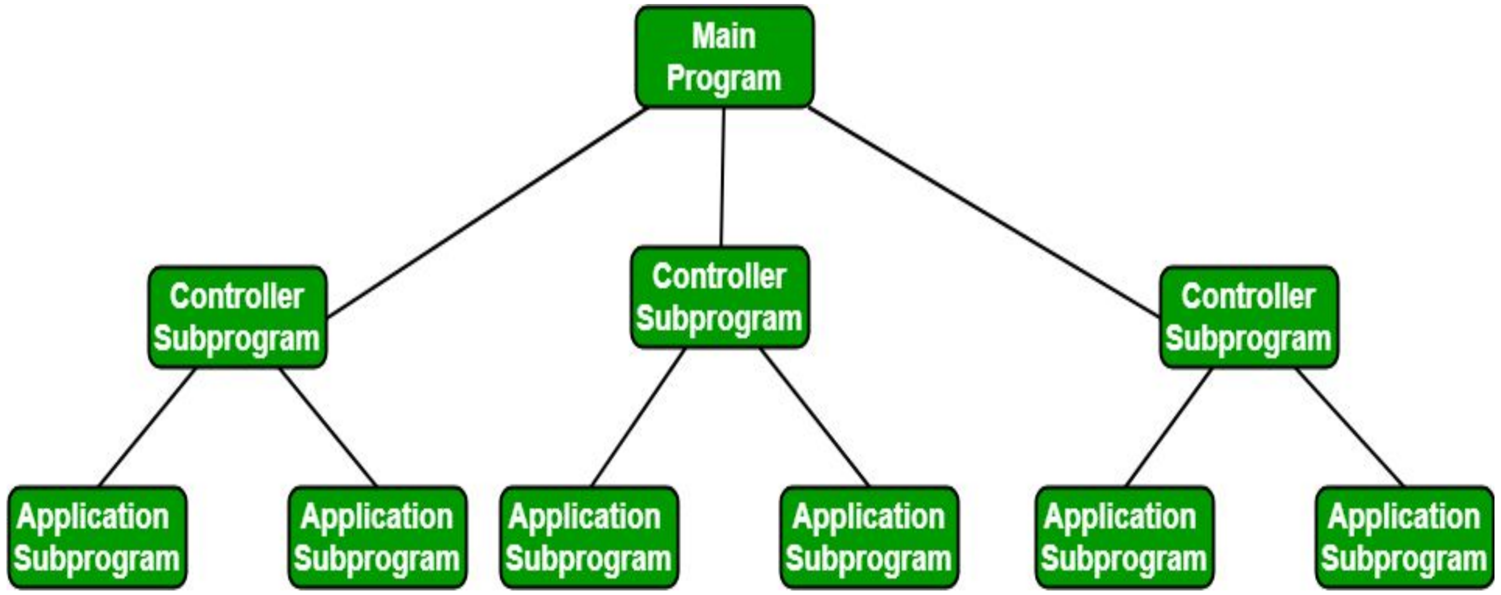


5. Call and return architectures

It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.

- **Remote procedure call architecture:** This components is used to present in a main program or sub program architecture distributed among multiple computers on a network.
- **Main program or Subprogram architectures:** The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.

Call and return architectures


Main Program
Controller Subprogram
Controller Subprogram
Controller Subprogram
Application Subprogram
Application Subprogram
Application Subprogram
Application Subprogram
Application Subprogram
Application Subprogram