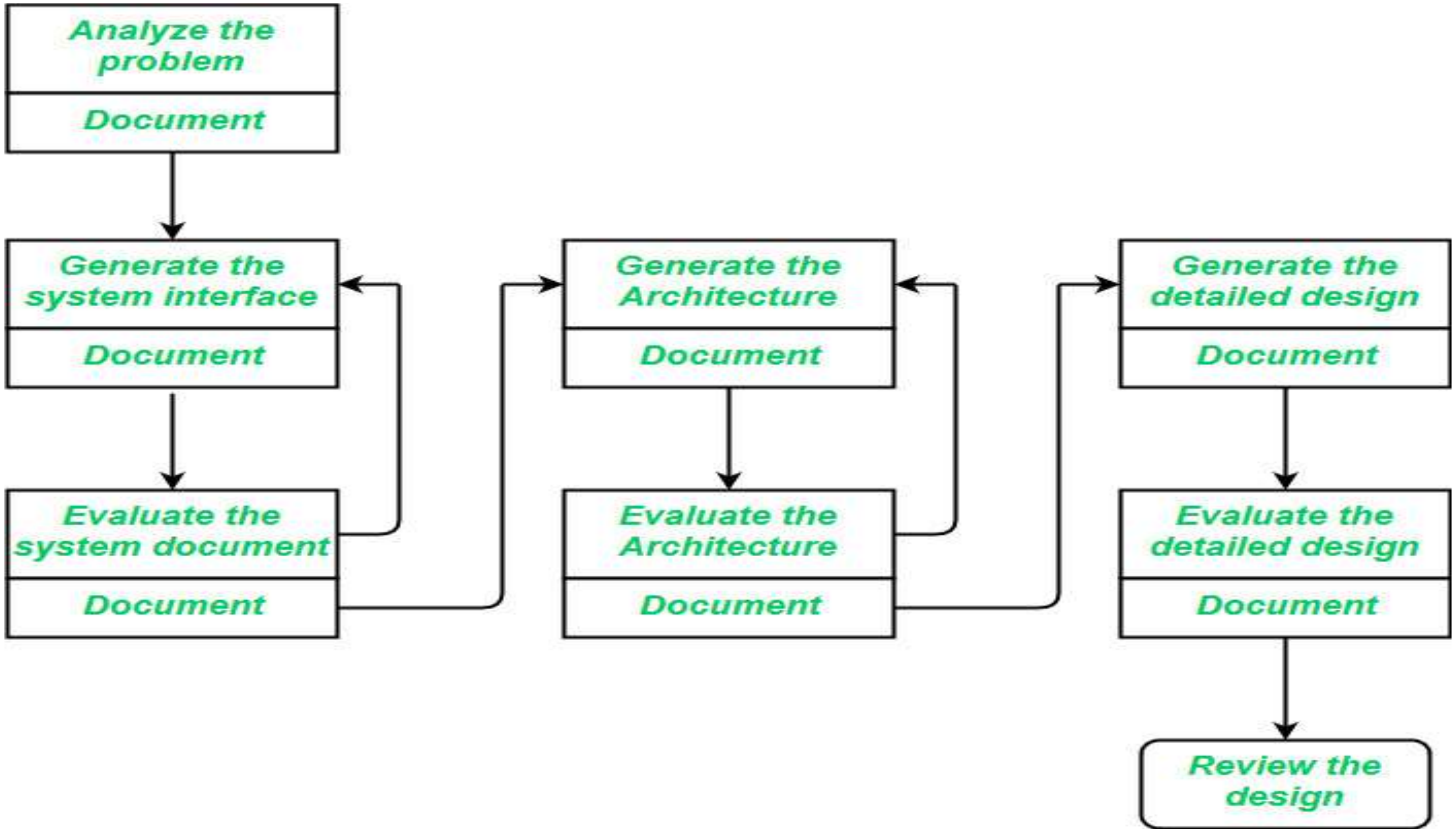
Topic 2: Software Design

❖Design Process

- Transforming the customer requirements as described in the SRS documents into a form implementable using a programming language.
- Iterative process through which requirements are translated into "blueprint" for constructing the software.
- The software design process can be divided into the following three levels or phases of design:

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


Analyze the problem
Document
Generate the system interface
Document
Evaluate the system document
Document
Generate the Architecture
Document
Evaluate the Architecture
Document
Generate the detailed design
Document
Evaluate the detailed design
Document
Review the design

❑Interface Design

- Specification of the interaction between a system and its environment.
- Proceeds at a high level of abstraction with respect to the inner workings of the system.
- for eg: During interface design, the internal of the systems are completely ignored, and the system is treated as a black box.
- Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts.
- The design problem statement produced during the problem analysis step should identify the people, other systems, and devices which are collectively called agents.

- Interface design should include the following details:

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

Focus Areas:

- **User Interface (UI)**: Design of the front-end components, defining how the end-user will interact with the system (e.g., screens, forms, buttons, etc.).
- **Application Programming Interface (API)**: Design of how the software will communicate with other systems or software components.
- **User Experience (UX)**: Considerations for making the software intuitive and user-friendly.

❑Architectural Design

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

1. Gross decomposition of the systems into major components.
2. Allocation of functional responsibilities to components.
3. Component Interfaces.
4. Component scaling and performance properties, resource consumption properties, reliability properties.
5. Communication and interaction between components.

- The architectural design adds important details ignored during the interface design. Design of the internals of the major components is ignored until the last phase of the design.

Focus Areas

- **System Organization**: Overall layout of components, including databases, middleware, and client-server architectures.
- **Module Decomposition**: Breaking down the system into modules or subsystems that will handle specific responsibilities.
- **Design Patterns**: Use of established design patterns to ensure scalability, maintainability, and performance.
- **Technology Stack**: Defining the platforms, languages, and frameworks that will be used.

❑ Detailed Design

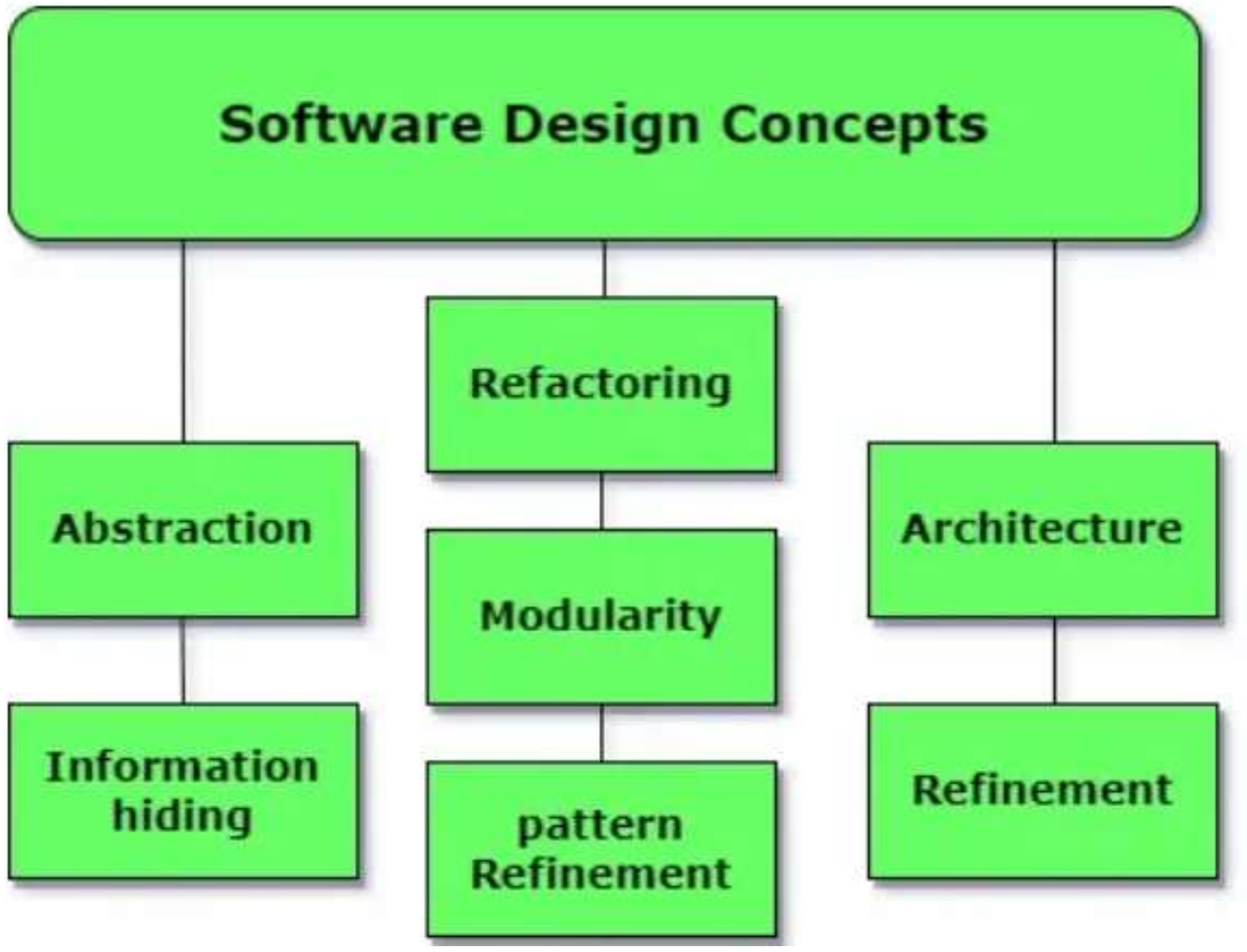
- Detailed design is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures.
- The detailed design may include:

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

Focus Areas:

- **Class Diagrams and Data Structures**: Designing classes, objects, data structures, and their interactions.
- **Algorithms**: Defining the algorithms and logic required for each functionality.
- **Component Interfaces**: Detailed design of how modules will interact, including parameter definitions, input/output specifications, and method calls.
- **Error Handling and Security**: Defining how the system will handle errors, exceptions, and security concerns.

❖ Design concepts



❖ Design concepts

- 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, and the logic, or thinking behind how you will design software.
- It allows the software engineer to create the model of the system 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:

➢ **Abstraction** :Data, Procedure and Control

➢ **Architecture** :The overall structure of the software

➢ **Patterns** :Coveys the essence of the proven design solution

- **Separation of Concerns** :Complex problem can be easily handled if it subdivided into smaller parts.
- **Modularity** :Compartmentalization of data and functions.
- **Information Hiding** :Controlled interfaces.
- **Functional Independence** :Single minded function and low coupling.
- **Refinement** :Elaboration of detail for all abstraction.
- **Aspects** :Mechanism for understanding how global requirement affect the design.
- **Refactoring** :A reorganization techniques that simplifies the design.
- **Object-Oriented Design Concepts**
- **Design Classes**: Provide design detail that will enable analysis class to be enabled.

❖ Design Modes

- Design modes or design patterns are standard solutions to common problems in software design. They provide templates for how to solve a problem that can be used in many different situations. Some well-known design patterns include:

1. **Creational Patterns:** Deal with object creation mechanisms, aiming to create objects in a manner suitable for the situation.
    - **Singleton:** Ensures a class has only one instance and provides a global point of access to it.
    - **Factory Method:** Defines an interface for creating an object but subclasses alter the type of objects that will be created.

2. Behavioral Patterns: Deal with communication between objects, making the flow of control more manageable.

- **Observer:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.
- **Strategy:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

3. Structural Patterns: Deal with object composition or structure, simplifying the design by identifying a simple way to realize relationships between entities.

- **Adapter:** Allows incompatible interfaces to work together.
- **Composite:** Composes objects into tree structures to represent part-whole hierarchies.

❖ Design Heuristics

- Design heuristics are rules of thumb or guidelines that help designers make decisions during the design process. They are less formal than design patterns and are used to improve the quality of the design.
- These heuristics and design patterns are tools that can help software developers create systems that are robust, maintainable, and scalable. Some common are

1. **Encapsulation:** Hide implementation details and expose only what is necessary. This reduces complexity and increases modularity.
2. **Separation of Concerns:** Divide the system into distinct sections, each addressing a separate concern. This makes the system easier to understand, develop, and maintain.
3. **DRY (Don't Repeat Yourself):** Avoid duplication of code. Each piece of knowledge must have a single, unambiguous representation in the system.

4. KISS (Keep It Simple, Stupid): Simplicity should be a key goal in design and unnecessary complexity should be avoided.

5. YAGNI (You Aren't Gonna Need It): Don't add functionality until it is necessary. This prevents overengineering and reduces the potential for bugs.

6.Open/Closed Principle: Software entities should be open for extension but closed for modification. This means you should be able to extend the behavior of a system without altering its existing code.

7. Single Responsibility Principle: A class should have only one reason to change means it should have only one job or responsibility.

8. Liskov Substitution Principle: Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

9. Interface Segregation Principle: No client should be forced to depend on methods it does not use. This means providing small, specific interfaces rather than a large, general-purpose one.

❖ Architectural design Decision

- Concerned with understanding how a system should be organized and designing the overall structure of that system.
- Architectural design is the first stage in the software design process.
- Identifies the main structural components in a system and the relationships between them.
- Architectural model that describes how the system is organized as a set of communicating components.

Architectural design decisions:

- Creative process where you design a system organization that will satisfy the functional and non-functional requirements of a system.

- The activities within the process depend on the type of system being developed, the background and experience of the system architect and the specific requirements for the system. It is therefore useful to think of architectural design as a series of decisions to be made rather than a sequence of activities.
- The system architects have to consider the following fundamental questions about the system:

➢ Is there generic application architecture that can act as a template for the system that is being designed?

➢ What strategy will be used to control the operation of the components in the system?

➢ What architectural organization is best for delivering the non-functional requirement of the system?

➢ How will the architectural design be evaluated?

➢ How should the architecture of the system be documented?

- The close relationship between nonfunctional requirements and software architecture, the particular architectural style and structure that choose for a system depend on the nonfunctional system requirements:

1. Performance: The architecture should be designed to localize critical operations within a small number of components, with these components all deployed on the same computer rather than distributed across the network.

2. Security: A layered structure for the architecture should be used, with the most critical assets protected in the innermost layers with a high level of security validation applied to these layers.

3. Safety: The architecture should be designed so that safety related operations are all located in either a single component or in a small number of components.

4. Availability: The architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system.

5. Maintainability: The system architecture should be designed using fine-grain, self-contained components that may readily be changed. Producers of data should be separated from consumers and shared data structures should be avoided.

❖System organization

- Reflects the basic strategy that is used to structure a system.
- Three organizational styles are widely used:

1. Shared Data Repository Style:

- ➢Multiple systems or components share a common data repository.
- ➢All components interact with the same dataset, which acts as a central point of data management.
- ➢This approach ensures consistency and reduces data redundancy but may create a bottleneck if the repository becomes a single point of failure or a performance constraint.
- ➢Common in database-centric applications and environments where data integrity and consistency are paramount.

2.Shared Services and Servers Style:

- This style involves sharing common services or servers among different components or systems.
- Services can include authentication, logging, data processing, etc., and servers can be web servers, application servers, etc.
- Promotes reusability and modularity, as components can leverage existing services without duplicating functionality.
- Often used in microservices architectures and environments with a Service-Oriented Architecture (SOA).

3.Abstract Machine or Layered Style:

- In this style, the system is organized into layers, each providing a set of services to the layer above and using services from the layer below.
- This abstraction helps in managing complexity by separating concerns and promoting a clear structure of responsibilities.
- Each layer operates as an abstract machine, hiding the details of lower-level operations and exposing a simpler interface to the upper layers.
- Commonly seen in operating system design, network protocol stacks, and software frameworks.

❖Modular decomposition styles:

- Styles of decomposing sub-systems into modules.
- No rigid distinction between system organization and modular decomposition.

Sub-systems and modules

- A sub-system is a system whose operation is independent of the services provided by other sub-systems.
- A module is a system component that provides services to other components but would not normally be considered as a separate system.

Modular decomposition

- Another structural level where sub-systems are decomposed into modules.
- Two modular decomposition models covered

➢An object model where the system is decomposed into interacting objects,

➢A pipeline or data-flow model where the systems is decomposed into functional modules which transform inputs to outputs.

- If possible, decisions about concurrency should be delayed until modules are implemented.

❖ Control styles

- Are concerned with the control flow between sub-systems. Distinct from the system decomposition model.

1. Centralized control

➢ One sub-system has overall responsibility for control and starts and stops other sub-systems.

Types

✓**Call-return model**

- Top-down subroutine model where control starts at the top of a subroutine hierarchy and moves downwards. Applicable to sequential systems.

✓**Manager model**

- Applicable to concurrent systems. One system component controls the stopping, starting and coordination of other system processes. Can be implemented in sequential systems as a case statement.

2. Event-based control

- ➢Each sub-system can respond to externally generated events from other sub-systems or the system's environment.

Types

✓ **Broadcast models**

- An event is broadcast to all sub-systems. Any subsystem which can handle the event may do so.

✓**Interrupt- driven models**

- Used in real-time systems where interrupts are detected by an interrupt handler and passed to some other component for processing.

❖ Reference Architecture

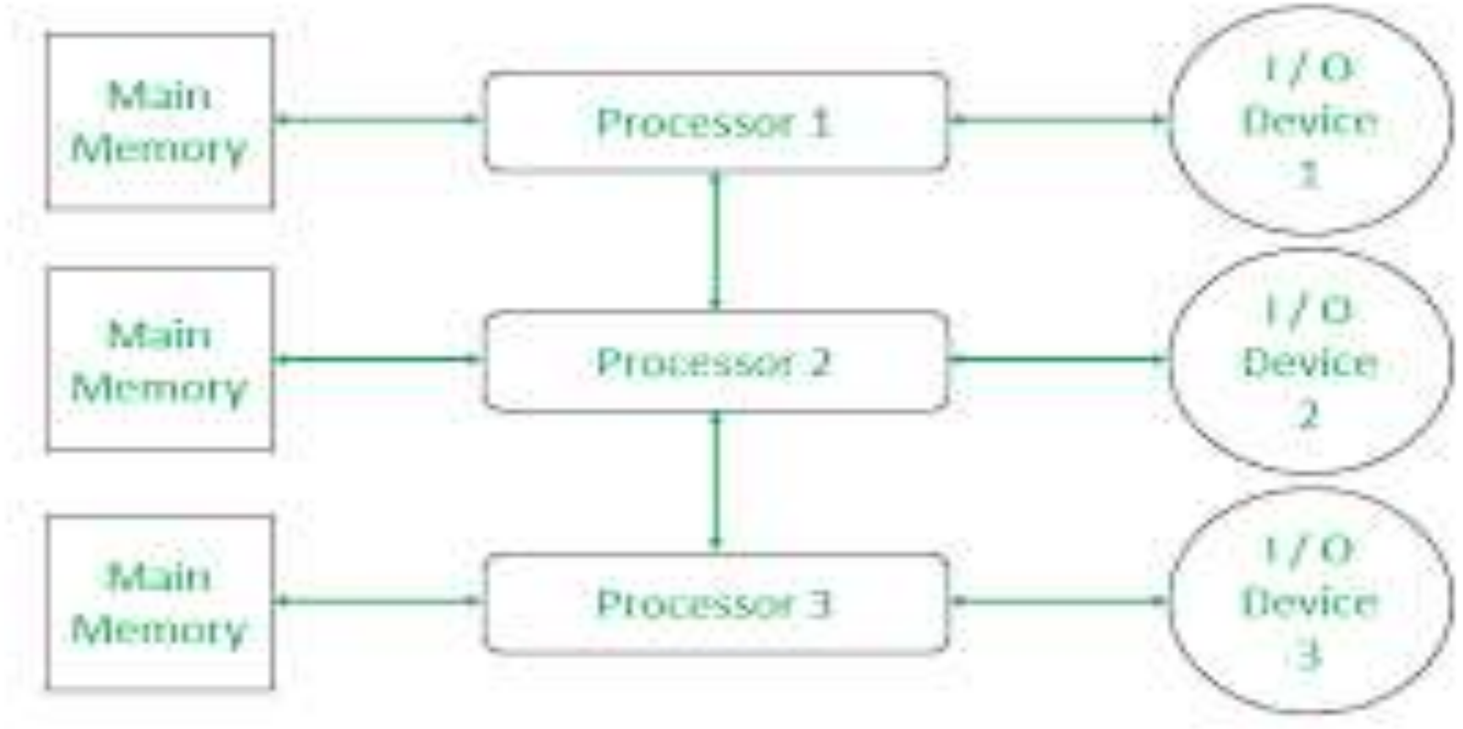
- Reference architecture captures important features of system architectures in a domain.
- Purpose of reference architectures is to evaluate and compare design proposal and educate people about architectural characteristics in that domain.
- Architectural models may be applicable to some application domain:

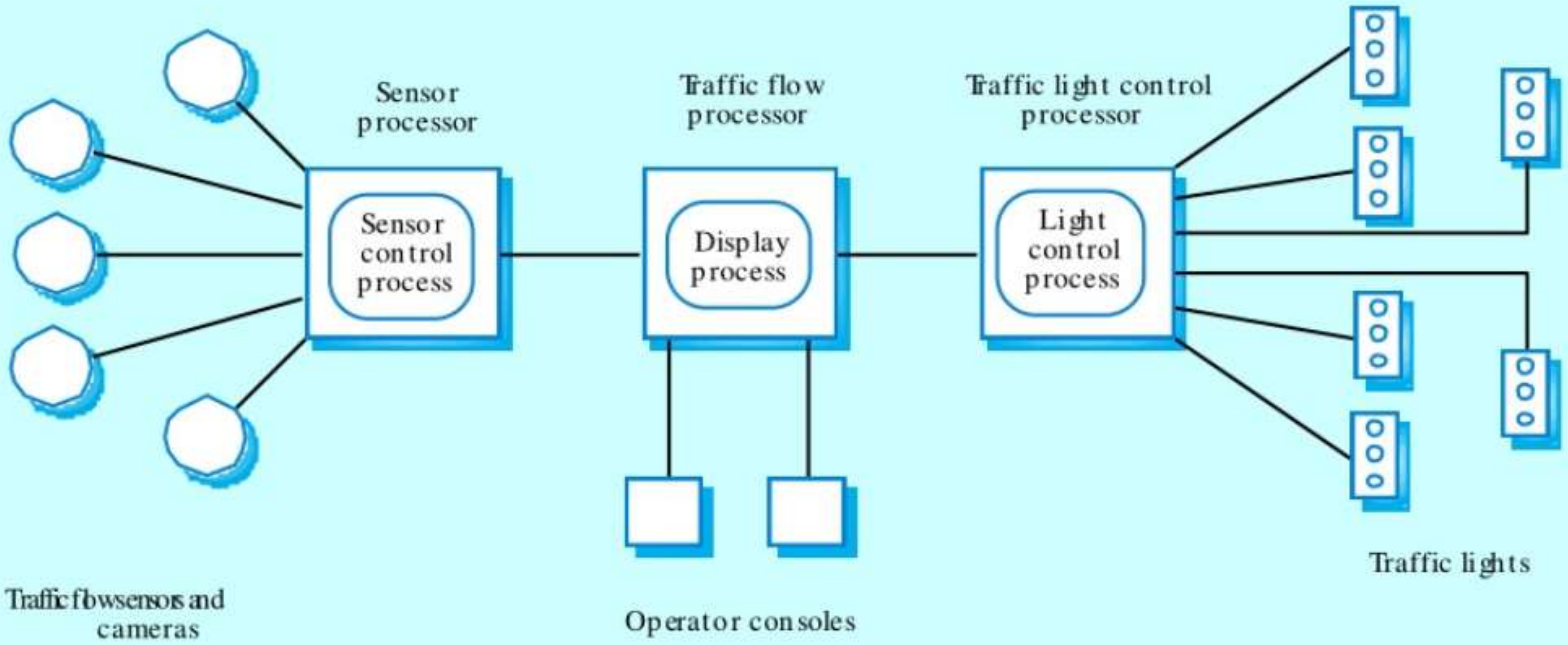
1. **Generic model:** bottom-top model
2. **Reference model:** top-down approach

- Reference models are derived from study of application domain rather than from existing system- maybe used as a basic system implementation or to compare different system.
- It acts as a standard against which system can be evaluated.
- OSI reference model is a layered model of communication system

❖ Multiprocessor Architecture

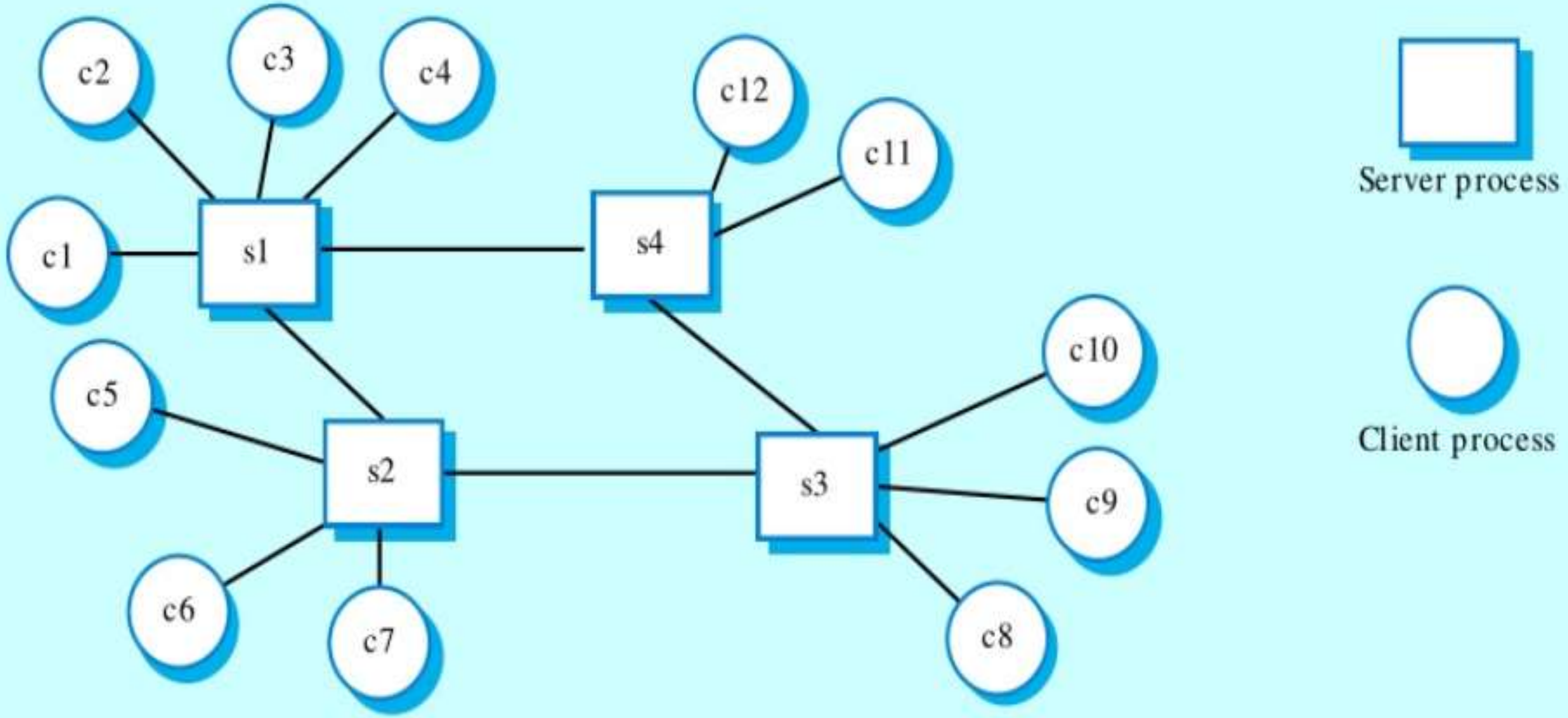
- It is the simplest distributed system model. The system is composed of multiple processor processes which may execute on different processor.
- It is Architectural model of many large real time systems. In this architecture, distribution of process to processor may be preordered or may be under the control of the dispatcher.




Sensor processor
Traffic flow processor
Traffic light control processor
Sensor control process
Display process
Light control process
Traffic flow sensors and cameras
Operator consoles
Traffic lights

❖ Client server Architecture

- This architectural model is used for the set of services that are provided by server and a set of clients that use this service.
- Client should know about the server, but the server need not know about clients in client-server architecture.
- Mapping of processor to process is not necessarily 1:1 in case of client server architecture.
- Client and server are logical processors in client-server architecture.


c1
c2
c3
c4
s1
c5
c6
c7
s2
s4
c12
c11
s3
c10
c9
c8
Server process
Client process

1. Two-Tier Architecture

- **Characteristics:**
  - **Client**: Directly interacts with the server.
  - **Server**: Provides resources or services to the client.
  - **Communication**: Typically uses protocols like HTTP, TCP/IP, or UDP.
- **Use Cases:**
  - Small-scale applications.
  - Simple database applications.
  - Local area network (LAN) applications.
- **Examples:**
  - Traditional desktop applications with a database server.
  - Simple web applications.

2. Three-Tier Architecture

➢**Characteristics:**

- **Client**: User interface (UI) tier, which interacts with the user.
- **Application Server**: Middle tier that processes business logic.
- **Database Server**: Data tier where data is stored and managed.

➢**Use Cases:**

- Enterprise applications.
- Web applications requiring separation of concerns.
- Scalable and maintainable systems.

➢**Examples:**

- Web applications using a web server, an application server, and a database server.
- E-commerce applications.

3. N-Tier Architecture (Multitier Architecture)

➢**Characteristics:**

- Extends the three-tier architecture to more tiers.
- Allows for greater separation of concerns and more scalability.
- Additional tiers might include additional business logic layers, data processing layers, etc.

➢**Use Cases:**

- Large-scale enterprise applications.
- Complex web services.
- Systems requiring high scalability and flexibility.

➢**Examples:**

- Applications using microservices architecture.

4. Peer-to-Peer (P2P) Architecture

➢ **Characteristics:**

- Each node (peer) can act as both client and server.
- Peers share resources directly with each other without a centralized server.
- Often used for file sharing and distributed computing.

➢ **Use Cases:**

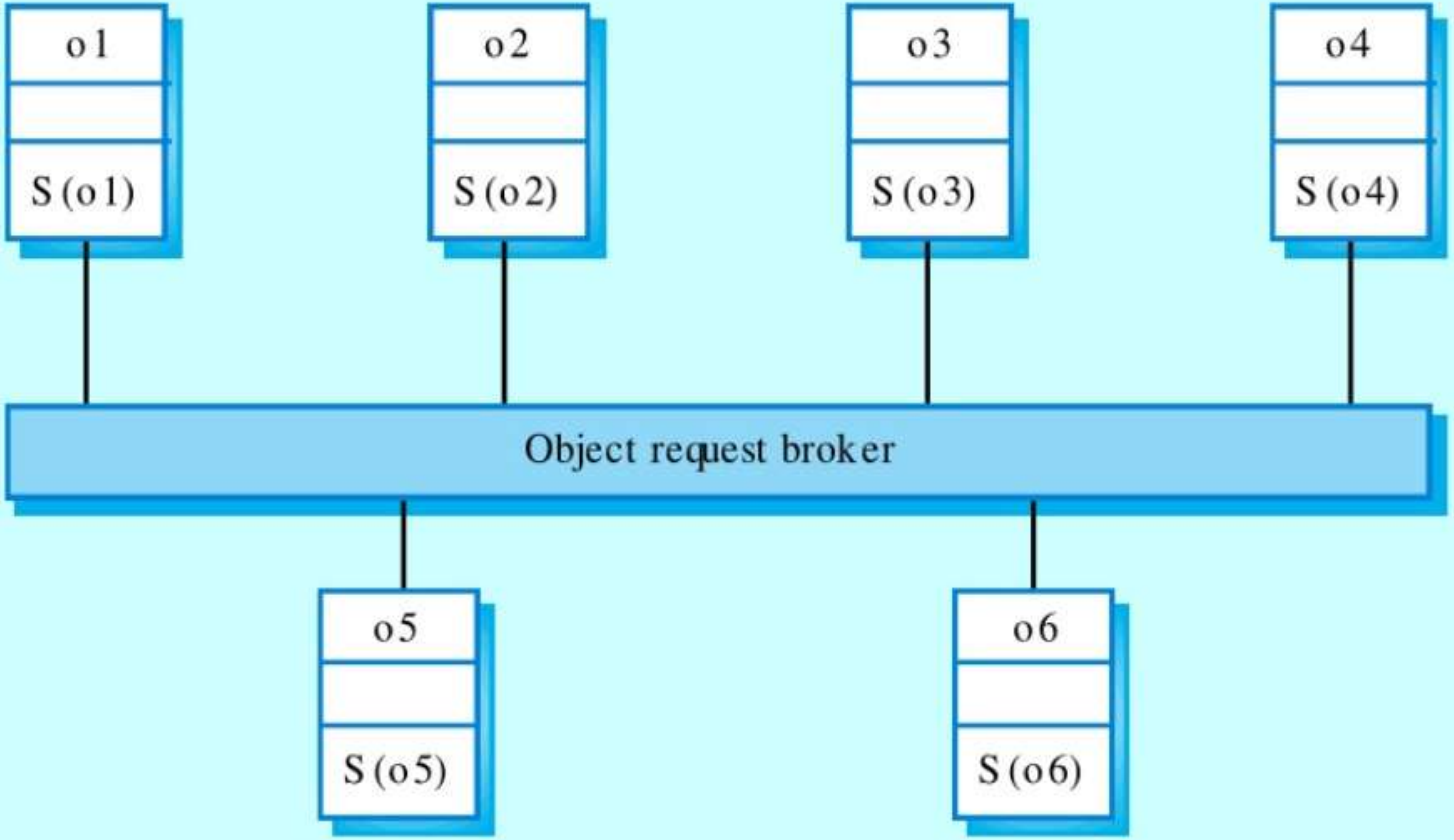
- File-sharing applications.
- Distributed computing projects.
- Decentralized applications (DApps).

➢ **Examples:**

- BitTorrent.
- Blockchain networks.

❖ Distributed object Architecture

- No distinction between client and server.
- Each distributed entity is objects that provide service other object and receive from other objects.
- Object communication is through middleware system called an object request booker or software bus.
- However, more complexes to design the client server architecture.
- It allows system designer to delay decision on where and how services should be provided.
- It is very open system architecture that allows to new resources to be as required.
- System is flexible and scalable.


o1
S (o1)
o2
S (o2)
o3
S (o3)
o4
S (o4)
Object request broker
o5
S (o5)
o6
S (o6)

❖ Inter-organizational distributed computing

- Used for security and interoperability reason.
- Local standards management and operational processes apply for such interorganizational distributed computing.
- Newer model of distributed computing have been designed to support interorganizational
- Computing where different node server are located at different organization.

❖ Real –time software design

- A real-time software system design means that the system is subjected to real-time, i.e., the response should be guaranteed within a specified timing constraint, or the system should meet the specified deadline.
- For eg: flight control systems, real-time monitors, etc.

Types

➢ **Hard real-time system:**

- Can never miss its deadline.
- Missing the deadline may have disastrous consequences.
- Example: Flight controller system.

➢ **Soft real-time system:**

- This type of system can miss its deadline occasionally with some acceptably low probability.
- Missing the deadline have no disastrous consequences.
- Example: Telephone switches.

➢ **Firm Real-Time Systems:**

- Lie between hard and soft real-time systems.
- In firm real-time systems, missing a deadline is tolerable, but the usefulness of the output decreases with time.
- Examples of firm real-time systems include online trading systems, online auction systems, and reservation systems.

❖ Component Based Software Engineering

- Process that focuses on the design and development of computer-based systems with the use of reusable software components.
- It not only identifies candidate components but also qualifies each component’s interface, adapts components to remove architectural mismatches, assembles components into a selected architectural style, and updates components as requirements for the system change.
- The process model for component-based software engineering occurs concurrently with *component-based development*.

CBSE framework activities

- **Component Qualification:** This activity ensures that the system architecture defines the requirements of the components for becoming a reusable components.
- **Component Adaptation:** This activity ensures that the architecture defines the design conditions for all components and identifies their modes of connection.
- **Component Composition:** This activity ensures that the Architectural style of the system integrates the software components and forms a working system.
- **Component Update:** This activity ensures update of reusable components.

THANK YOU