CHAPTER 4 - SOFTWARE STRUCTURE AND ARCHITECTURE (PART 1)

LECTURE OUTLINE

- Software Architecture Structures
- Architecture Views
- Architectural Styles/Patterns

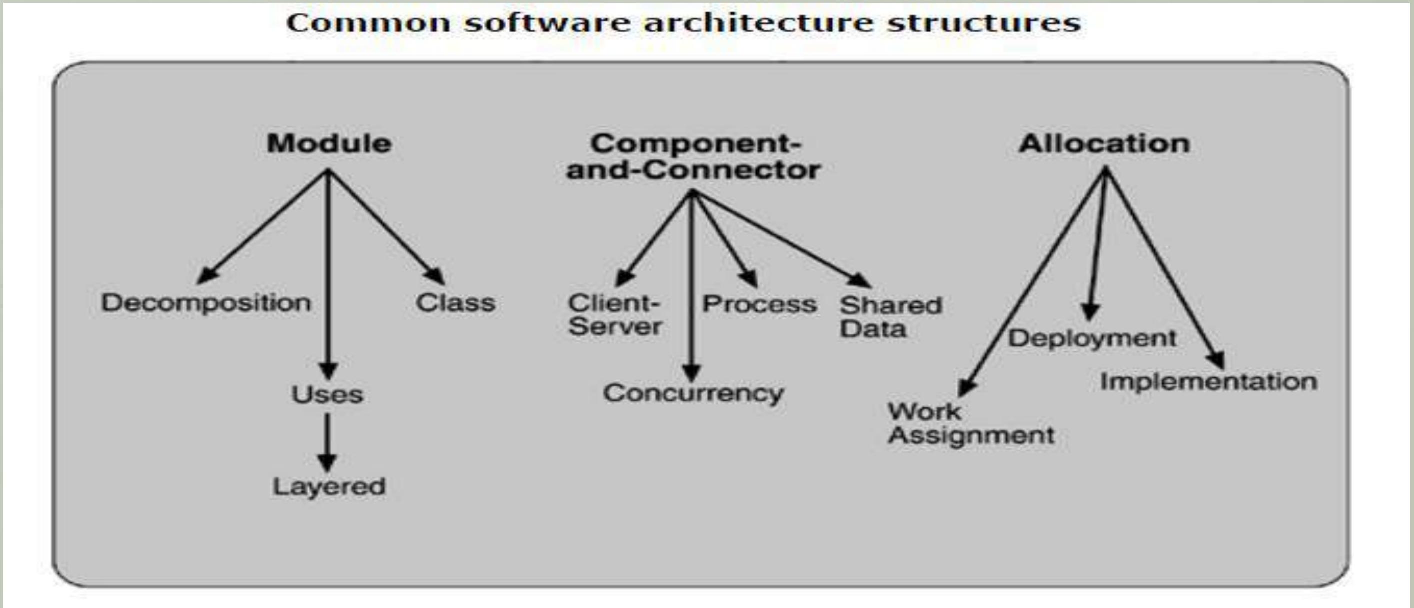
SOFTWARE ARCHITECTURE STRUCTURE

SOFTWARE ARCHITECTURE STRUCTURES

- Software systems are composed of many structures.
- A structure is simply a set of elements held together by a relation.
- **Software architecture** is the set of structures needed to reason about the system, which comprise of software elements, the externally visible properties of those elements and the relationships (interaction) among them.
  - Interaction: Pipes, Client Server, Peer-to-Peer etc.
  - System-wide Properties: Data rates, latencies, behavioral change propagation.

SOFTWARE ARCHITECTURE STRUCTURES

- Three categories of software structures:
  - **Modules** structures
  - **Component-and-connector** structures (Dynamic structures)
  - **Allocation** structures



MODULE-BASED STRUCTURES

- Structures that partition systems into implementation units
- How is the system to be structured as a set of code or data units? (functional responsibility)
- Assigned specific computational responsibilities.
- Basis of work assignments for programming teams
  - Team A works on Database
  - Team B works on Business rules
  - Team C works on the User Interface

MODULE-BASED STRUCTURES

- In large projects, modules are sub-divided for assignment to sub teams.
  - The Database for a large Enterprise Resource Planning (ERP) system might be very complex that its implementation is sub-divided into many parts.
  - The structure that captures the sub-division is a Module Structure or Module Decomposition Structures (emerged from OOA&D)

MODULE-BASED STRUCTURES

- Decomposition – the units are modules related by the "is a submodule of" relation.
- Uses – the units are modules, procedures or resources on the interfaces of modules related by the "uses" relation.
- Layered – when the uses relations are controlled in a particular way a system of layers emerges in which layer is a coherent set of related functionality.
- Class or Generalization – the module units are classes and the relation is "inherits-from or "is-an-instance of".

COMPONENT-AND-CONNECTOR STRUCTURES

- Dynamic structures
- How is the system to be structured as a set of elements that have running behavior and interactions?
- Focus on way the software elements interact with each other at runtime to carry out the system's functions.
- Example: System is a set of services
    - The services, the infrastructure: they interact with each other.

COMPONENT-AND-CONNECTOR STRUCTURES

- Synchronization and interaction relations among them (services and infrastructure) forms a software structure, used to describe the system.
- Component is always a runtime entity.
- Process or Communicating Processes – deals with the dynamic aspects of a running system and the relation is "attachment". It shows how processes (the units) are connected by communication, synchronization, and/or exclusion operations.

COMPONENT-AND-CONNECTOR STRUCTURES

- Currency – the units are components and the connectors are “logical threads”. It allows determination of opportunities for parallelism and locations of resource contention.
- Shared data or repository – comprises components and connectors that create store and access persistent data.
- Client-Server – components are the clients and servers and the connectors are the protocols and messages they share.

ALLOCATION STRUCTURES

- How is the system relates to non software structures in its environment?
- Describes the mapping from software structures to the system's organizational, developmental, installation and execution environments
- For example:
  - Modules are assigned to teams to develop (organizational and developmental mapping)
  - Modules are assigned to places in a file structure for implementation, integration and testing
  - Components are deployed onto hardware in order to execute

ALLOCATION STRUCTURES

- Deployment – shows how software is assigned to hardware-processing and communication elements and the relations are “allocated to” and “migrate to”.
- Implementation – shows how software elements (modules) are mapped to the file structure(s) in the system’s development, integration, or configuration control environments.
- Work Assignment – assigns responsibility for implementing and integrating the modules to the appropriate development teams.

RELATING STRUCTURES TO EACH OTHER

- Each of these structures provides a different perspective and design handle on a system.
- Each is valid and potentially useful in its own right
- They are not independent but highly interrelated.
- Often the module structure is dominant, but not always.
- Not all systems warrant consideration of many architectural structures.
- The larger the systems the more dramatic the differences between these structures tend to be.

RELATING STRUCTURES TO EACH OTHER

- Structures represent the primary engineering leverage point for satisfying quality attributes.
- Multiple structures represent a powerful separation of concerns approach for creating the architecture.

ARCHITECTURE VIEWS

ARCHITECTURAL VIEWS

- Modern systems are frequently too complex to grasp all at once.
- At one moment, we restrict our attention to limited software system’s structures – one view of the overall system.
- A **view** is a representation of a coherent set of architectural elements, as written by and read by system stakeholders.
- It consists of a representation of a set of elements and the relations among them.

ARCHITECTURAL VIEWS

- Each architectural model only shows one view or perspective of the system.
- It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network.
- For both design and documentation, you usually need to present multiple views of the software architecture.
  - **4 + 1 view model** advocated a multiple-view modeling approach for software architectures

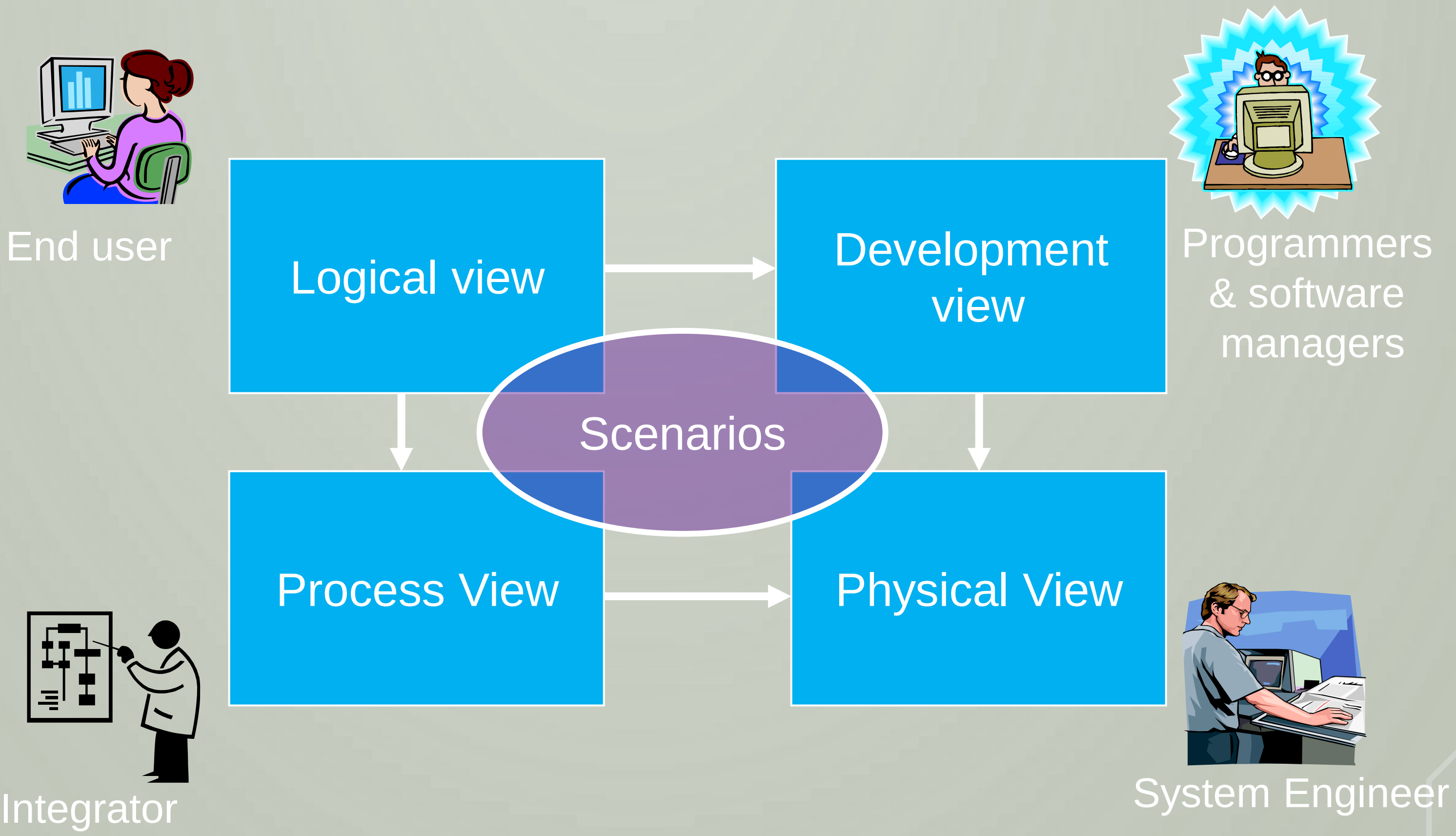
ARCHITECTURAL VIEWS

- Kruchten's "four plus one" approach recommends four **views** (structures):
  - Logical – the elements are "key abstractions" (a module structure)
  - Process – addresses concurrency and distribution of functionality (a component-and-connector structure)
  - Development (implementation) – organization of software modules, libraries, subsystems, and units of development (an allocation structure)
  - Physical (deployment) – maps elements onto processing and communication

ARCHITECTURAL VIEWS

- The “plus one” is a set of key use cases (scenarios) that serve to validate that the structures are not in conflict with each other and together describe a system meeting its requirements.

KRUCHTEN'S 4 + 1 VIEWS



LOGICAL VIEW – THE OBJECT-ORIENTED DECOMPOSITION

- A.k.a Design View or Conceptual View
- Which shows the key abstractions in the system as objects or object classes (static modeling view)
- Primarily supports the functional requirement – what the system should provide in term of services to its users.
- Exploits the principles of abstraction, encapsulation and inheritance.

LOGICAL VIEW – THE OBJECT-ORIENTED DECOMPOSITION

- **Viewer:** End-user
- **Concerns**: Primarily supports the functional requirements (What the system should provide in terms of services to its users).
- **Representation:**
  - Class diagrams (classes and logical relationships)
  - Class categories
  - Class utilities

PROCESS VIEW - THE PROCESS DECOMPOSITION

- Which shows how, at run-time, the system is composed of interacting processes – concurrent process or task view.
- Takes into account some non-functional requirements such as performance and availability.
- Addresses issues of concurrency and distribution, of system' integrity, of fault-tolerance and how the main abstractions from the logical view fit within the process architecture.
- Can be described at several levels of abstractions, each level addressing difference concerns.

PROCESS VIEW - THE PROCESS DECOMPOSITION

- **Viewer:** Integrators
- **Concerns:** Non-functional requirements (concurrency, performance, availability, etc.)
- **Representation:**
  - Different levels of abstractions
    - Processes and Threads
    - Major Tasks, Minor Tasks
    - Communication Mechanisms
      - Major tasks uses synchronous and asynchronous message communications, RPC, and event broadcasts, etc.
      - Minor tasks uses rendezvous or shared memory

DEVELOPMENT VIEW - SUBSYSTEM DECOMPOSITION

- A.k.a Implementation View
- Which shows how the software is decomposed for development – a subsystem and component design view.
- Focuses on the actual software module organization on the software development environment.
- The subsystems are organized in an hierarchy of layers, each layer providing a narrow and well-defined interface to the layers above it.

DEVELOPMENT VIEW - SUBSYSTEM DECOMPOSITION

- **Viewer**: Programmers and Software Managers
- **Concerns**: Actual software module organization (hierarchy of layers, software management, reuse, constraints of tools) on the software development environment.
- **Representation**:
  - Layered Style (depends on same levels or layers below)

PHYSICAL VIEW - MAPPING THE SOFTWARE TO HARDWARE

- A.k.a Deployment View
- Which shows the system hardware and how software components are distributed across the processors in the system.
- Takes into account primarily the non-functional requirements of the system such as availability, reliability (fault-tolerance), performance (throughput) and scalability.
- The software executes on a network of computers, or processing nodes.
- Various elements identified – networks, processes, tasks and objects.

PHYSICAL VIEW - MAPPING THE SOFTWARE TO HARDWARE

- **Viewer:** System Engineers
- **Concerns**: Primarily the non-functional requirements of the systems (like availability, reliability, scalability). Non-functional requirements regarding to underlying hardware (Topology, Communication)
- **Representation:**
  - Various forms (words, notations) over the process view
- There may be two architecture:
  - Test and development
  - Deployment

SCENARIOS/USE CASE VIEW (+1) - PUTTING IT ALL TOGETHER

- The unifying view – functional requirements view.
- Scenarios – instances of more general use cases.
- Serves:
  - As a driver to discover the architectural elements during the architecture design.
  - As a validation and illustration role after this architecture design is complete.
- Help illustrate and validate the document
- Help architect during the architecture design.

SCENARIOS/USE CASE VIEW (+1) - PUTTING IT ALL TOGETHER

- **Viewer:** All users of other views and Evaluators.
- **Concerns**: Redundant with other views (thus “+1”)
  - Drivers to discover architectural elements
  - Validation and illustration to show the design is complete
  - Considers**:** System consistency, validity
- **Representation**:
  - Similar to the logical view but a few variations.

CORRESPONDENCE BETWEEN THE VIEWS

- Views are interconnected.
- Start with Logical view (Requirements Documents) and move to Development or Process view and then finally go to Physical view.
- **From Logical to Process view**
  - Two strategy to analyse level of **concurrency**:
    - Inside-out: starting from Logical structure
    - Outside-in: starting from Physical structure

CORRESPONDENCE BETWEEN THE VIEWS

- **From Logical to Development**
  - They are very close, but the larger the project, the greater the distance
  - Grouping to subsystems based on:
    - Classes
    - Class packages
    - Line of codes
    - Team organization

4+1 VIEW MODEL UML DIAGRAMS

Views	Notations
Logical View	Class diagram, Object diagram, Sequence diagram, Communication diagram, State diagram
Process View	Activity diagram
Development / Implementation View	Component diagram, Package diagram
Physical View	Deployment diagram
Use case View / Scenario	Use case diagram

ARCHITECTURAL STYLES/PATTERNS

ARCHITECTURAL STYLES

- During software architecture, designers spend a great deal of time devising architectural solutions that provide the necessary components and interfaces to **achieve system requirements**.
- An architectural style is “a specialization of element and relation types, together with a set of constraints on how they can be used”. It is a conceptual way of how the system will be created/will work.
    - It contains a set of rules, constraints, and patterns of how to structure a system into a set of elements and connectors.
    - It governs runtime interaction of flow control and data transfer.

ARCHITECTURAL STYLES

- Common styles have emerged that describe elements of the system together with their interrelationships and quality characteristics.
- Identifying and designing using architectural styles can improve the efficiency of the development process and the quality of the final system.

ARCHITECTURAL STYLES

- For each view, the architects can pick a certain architectural style.
- The choice of applying architectural styles for designing some architectural elements depend on:
  - System type
  - Requirements
  - Desired quality attributes

ARCHITECTURAL STYLES

- An architecture style:
  - Describes a **class** of architectures or significant architecture pieces
  - Is **found repeatedly** in practice
  - Is a coherent package of **design decisions**
  - Has **known properties** that permit **reuse**
- In other words, architecture styles are like “design patterns” for the structure and interconnection within and between software systems

ARCHITECTURAL PATTERNS

- "**Pure**" architectural styles are **rarely** used in practice.
- Many of the **original** architectural styles have been **reformulated as patterns**.
- An architectural pattern describes a solution for implementing a style at the level of subsystems or modules and their relationships.
- The main difference is that a pattern can be seen as a **solution to a problem**, while a style is **more general** and does not require a problem to solve for its appearance.

ARCHITECTURAL PATTERNS

- Architects must understand the "**pure**" styles to understand the strength and weaknesses of the style as well as the consequences of deviating from the style.
- As many of the original architectural styles have been **reformulated** as architectural patterns, the use of both terms are interchangeable.

ARCHITECTURAL PATTERNS

- An architecture pattern is determined by:
  - A set of element types (such as data repository or a component)
  - A topological layout of the element indicating their interrelationships
  - A set of semantic constraints (e.g. filters in a pipe-and-filter style)
  - A set of interaction mechanisms that shows how the elements coordinate through the allowed topology
- Architectural patterns are similar to software design patterns but have a broader scope.

ARCHITECTURAL PATTERNS CLASSIFICATION

- Hierarchical – systems where components can be structured as a hierarchy to reflect different levels of abstraction and responsibility.
- Data-centered – systems that serve as a centralized repository for data, while allowing clients to access and perform work on data
- Data flow – systems oriented around the transport and transformation of a stream of data
- Distributed – systems that primarily involve interaction between several independent processing units connected via a network
- Interactive – systems that serve users or user-centric systems

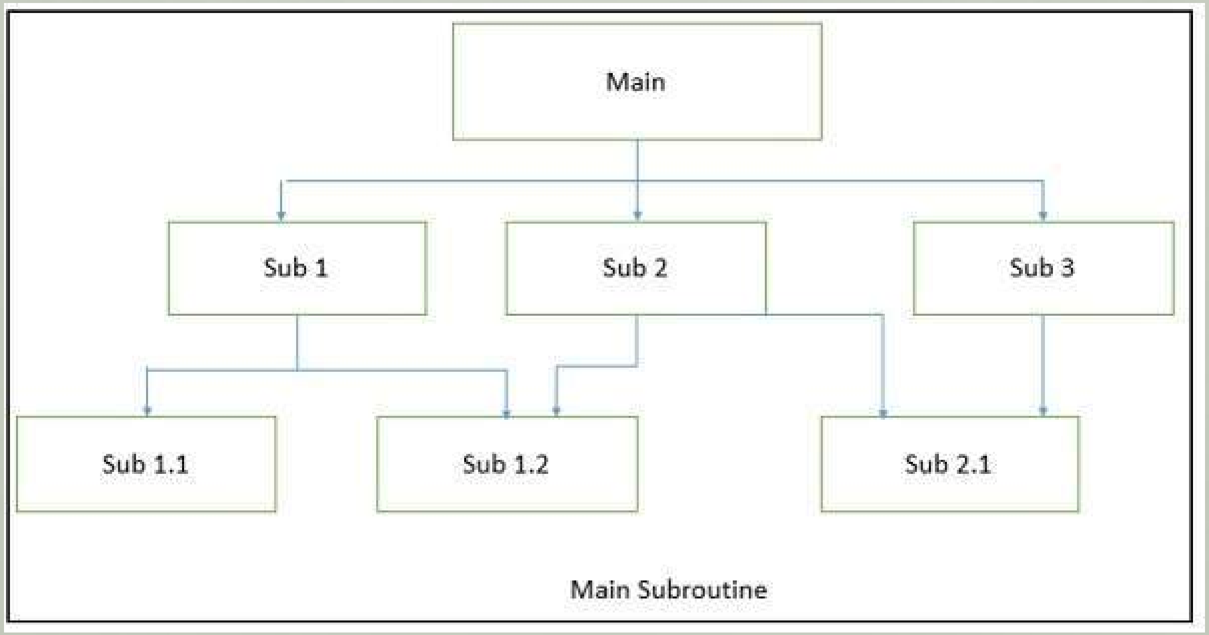
HIERARCHICAL SYSTEMS

- Components can be structured in hierarchical fashion so that components exists at different levels of abstraction and each level addresses a particular concern of the software system.
- For each level identified in the system, one or more can components can be identified, each possibly branching downward to other components necessary to carry out a particular operation.
- Two common styles/patterns:
  - Main program and subroutine
  - Layered

MAIN PROGRAM & SUBROUTINE

- Used in the structured (or functional) design strategy.
- Main component (or program) contains the main data for the program, which is shared among components residing at lower levels of the hierarchy.
- Quality Properties:
  - Modifiability – decomposing the system into independent components – easier to understand and manage
  - Reusability – independent components can be reused in other systems.

MAIN PROGRAM & SUBROUTINE



LAYERED ARCHITECTURE

- Layered architecture focuses on the grouping of related functionality within an application into distinct layers that are stacked vertically on top of each other.
- Each layer represents a different level of abstraction.
- Each layer provides services to the layer(s) ‘**above**’ and makes use of services from the layer(s) ‘**below**’.
- Examples:
  - Operating Systems.
  - Communication Protocols.

LAYERED ARCHITECTURE

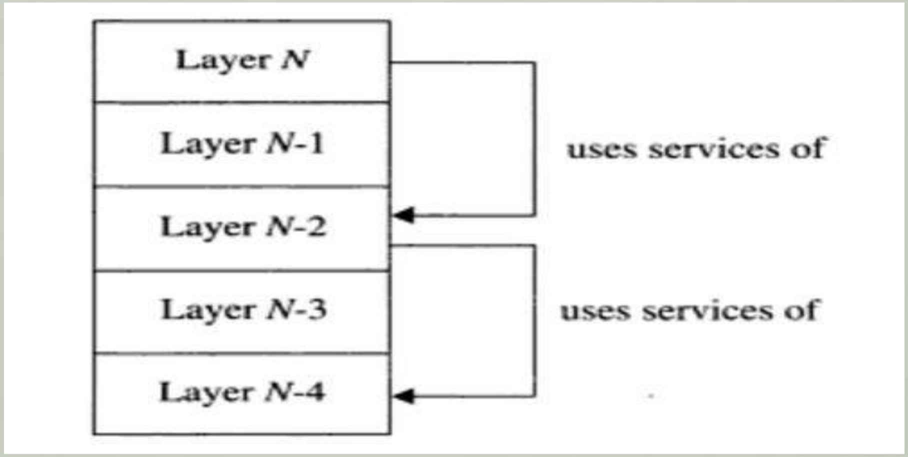
- Flow
  - requests from higher layer to lower layer
  - answers from lower layer to higher layer
  - incoming data or event notification from low to high
- Used when:
  - Building new facilities on top of existing systems
  - The development is spread across several teams with each team responsibility for a layer of functionality.
  - There is a requirement for multi-level security.

LAYERED ARCHITECTURE

- Each layer is a module that provides a cohesive set of *services* through a well-defined *interface.*
- Used to model the interfacing of sub-systems.
- Layered architectures are **open** (a.k.a *relaxed)* if each layer provides services to and/or makes use of services from two or more other layers
- Layered architectures are **closed** *(*a.k.a *strict)* if each layer only provides services to the layer immediately above it or makes use of services from the layer immediately below it.

OPEN LAYERED ARCHITECTURE

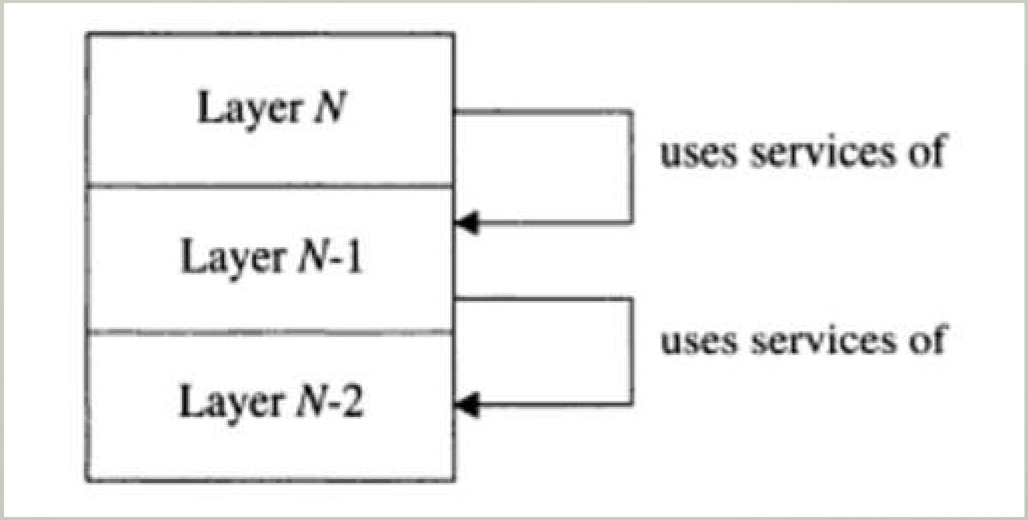
- A layer can use services from any lower layer.



- More compact code as the services of lower layers can be accessed directly - No need for extra code to pass messages through each layer
- Breaks the encapsulation of layers, so increase dependencies between layers and hence maintenance overhead.

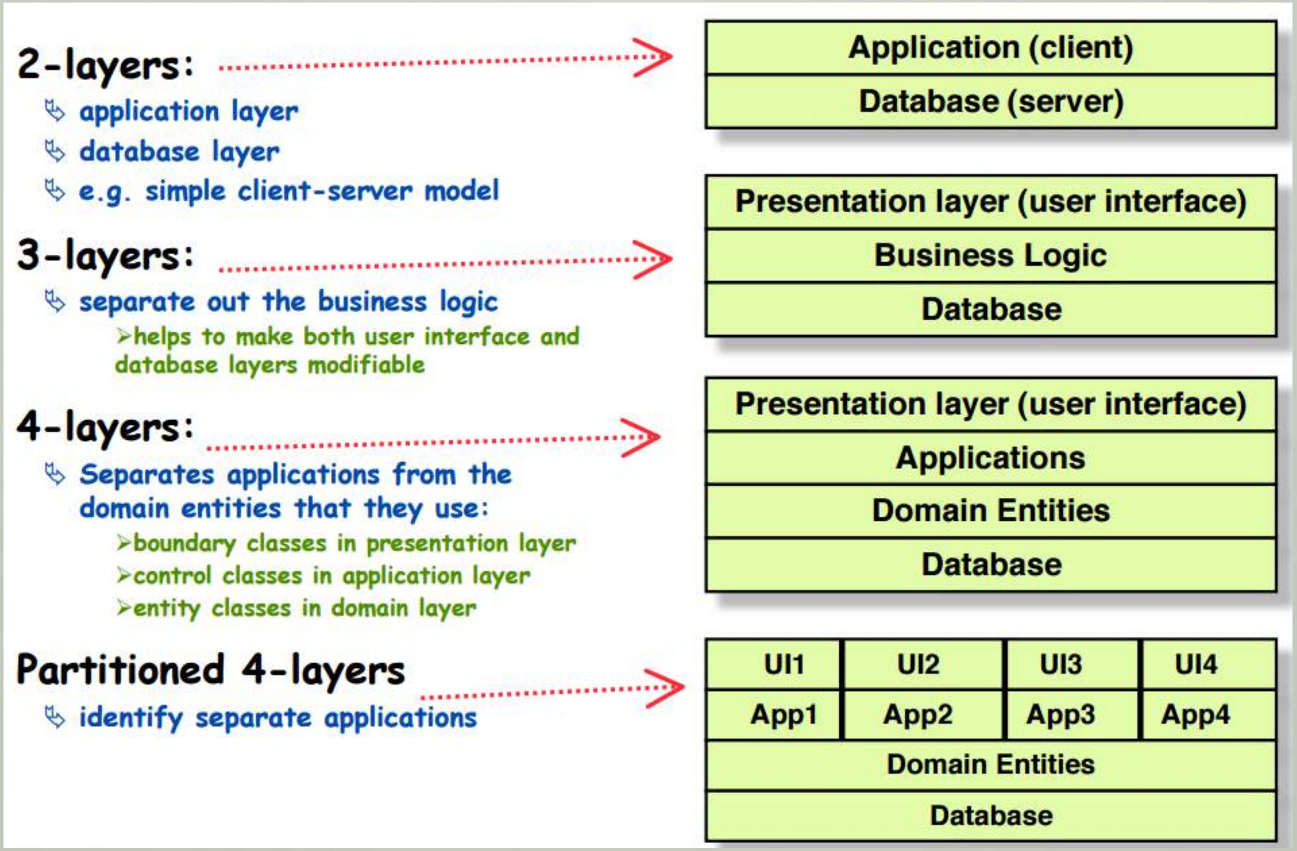
CLOSED LAYERED ARCHITECTURE

- Each layer only uses services of the layer immediately below.



- Minimises dependencies between layers
- Reduces the impact of a change to the interface of any one

HOW MANY LAYERS?



2-layers:
- application layer
- database layer
- e.g. simple client-server model

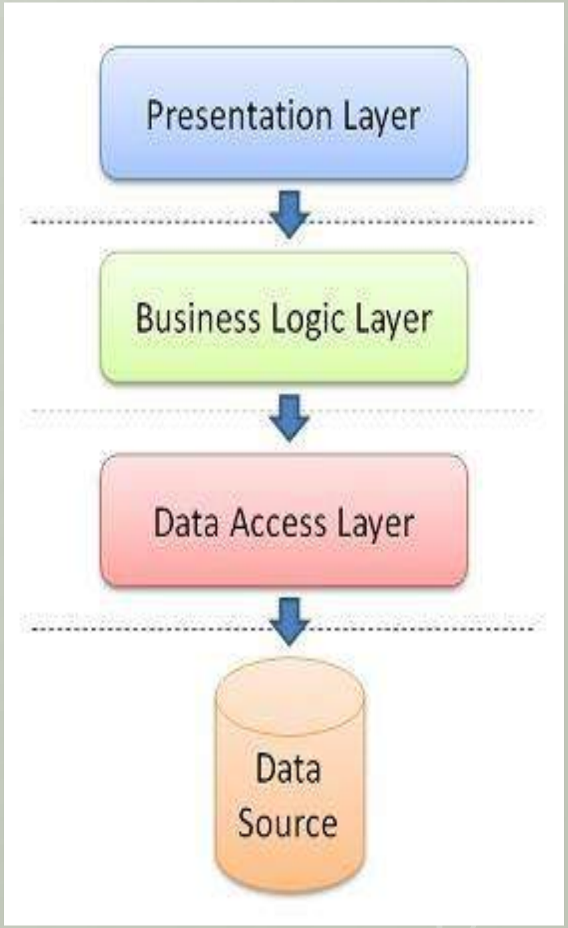
3-layers:
- separate out the business logic
  - helps to make both user interface and database layers modifiable

4-layers:
- Separates applications from the domain entities that they use:
  - boundary classes in presentation layer
  - control classes in application layer
  - entity classes in domain layer

Partitioned 4-layers
- identify separate applications

THREE-LAYERED ARCHITECTURE

- Commonly used for business-oriented information systems.
- Divided into:
  - *Presentation Layer which includes UI (User Interface), displaying data to the user, acceptance of input from the user.*
  - *Business Layer which includes business rules, data validation and task behaviour.*
  - *Data Access Logic which includes database communication, executing SQL Queries via the relevant API (manage the persistent storage of data).*

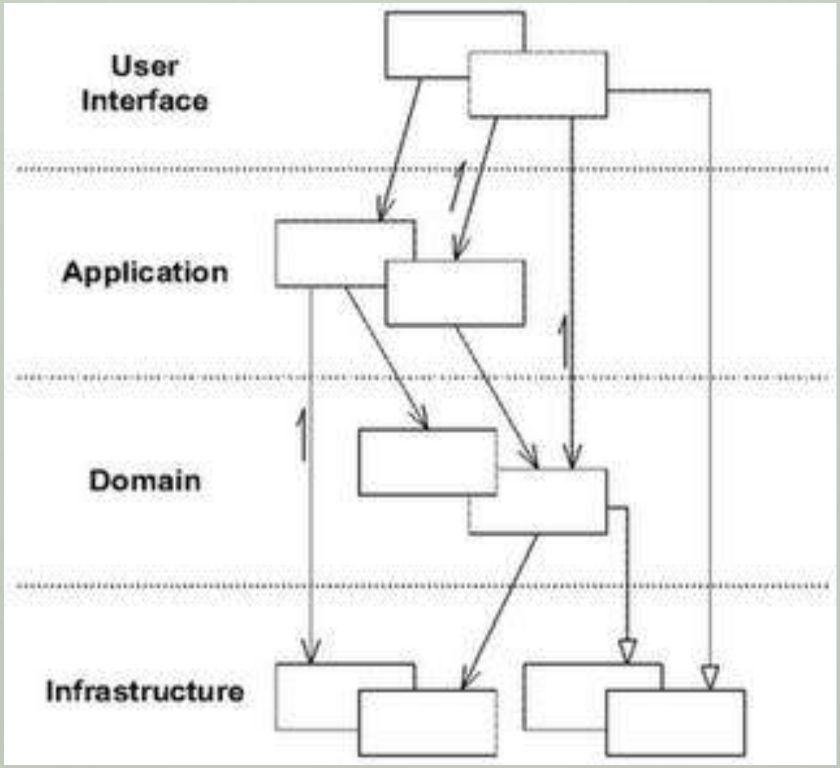


THREE-LAYERED ARCHITECTURE

- Benefits to Organization
  - More code can be sharing instead of duplication.
  - Developer can change the contents of any one of the layers without having to make changes. i.e.
    - UI Changes - For example – Desktop to the web, would only require changes to the components in Presentation Layer.
    - DB Changes – A change from one DB to another would only require changes to the Database Access Layer.
  - Possible to use different teams of developers to work on each due to having separate layers. i.e. PHP/Java developer for the business layer, SQL developer for the data access layer and HTML/CSS resource for the presentation layer.

FOUR-LAYERED ARCHITECTURE

- Separates business logic into application logic and domain layers so that several applications can share the same domain.



ADVANTAGES OF LAYERED ARCHITECTURE

- **Abstraction**. Layered architecture abstracts the view of the system as whole while providing enough detail to understand the roles and responsibilities of individual layers and the relationship between them.
- **Encapsulation**. No assumptions need to be made about data types, methods and properties, or implementation during design, as these features are not exposed at layer boundaries.

ADVANTAGES OF LAYERED ARCHITECTURE

- **Clearly defined functional layers**. The separation between functionality in each layer is clear. Upper layers such as the presentation layer send commands to lower layers, such as the business and data layers, and may react to events in these layers, allowing data to flow both up and down between the layers.
- **Reusable**. Lower layers have no dependencies on higher layers, potentially allowing them to be reusable in other scenarios.

ADVANTAGES OF LAYERED ARCHITECTURE

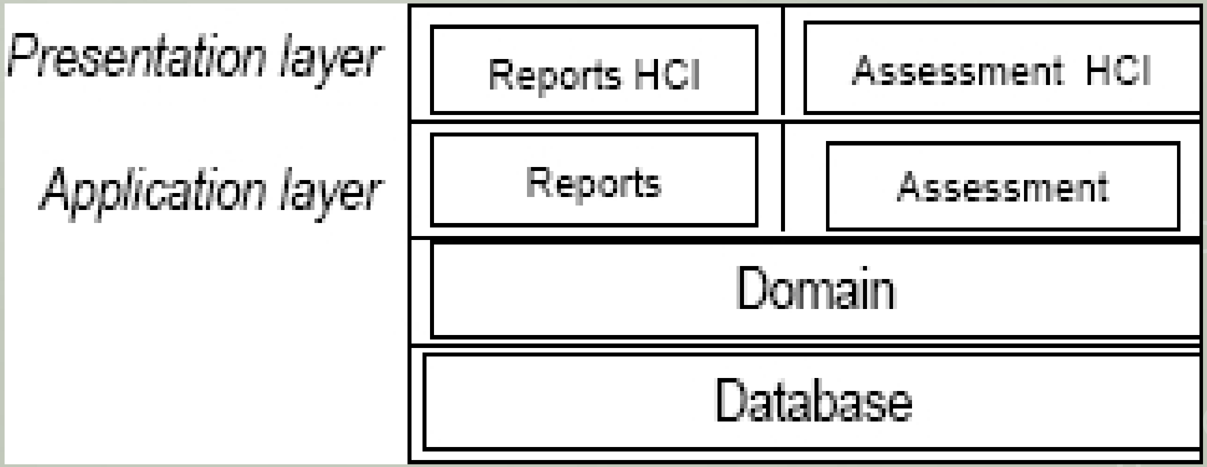
- **High cohesion**. Well-defined responsibility boundaries for each layer, and ensuring that each layer contains functionality directly related to the tasks of that layer, will help to maximize cohesion within the layer.
- **Loose coupling**. Communication between layers is based on abstraction and events to provide loose coupling between layers.

DISADVANTAGES OF LAYERED ARCHITECTURE

- Data often pass through many layers, thus reducing performance. The workaround of adopting an open layered architecture increases coupling, while decreasing simplicity and information hiding.
- Multi-layered programs can be hard to debug as operations tend to be implemented through calls across layers.
- Getting the layers right may be difficult. Too few layers may result in low individual cohesion, may be too big and complicated, and may discourage their reuse. Too many layers may result in high interlayer coupling and in degraded performance.

PARTITIONING

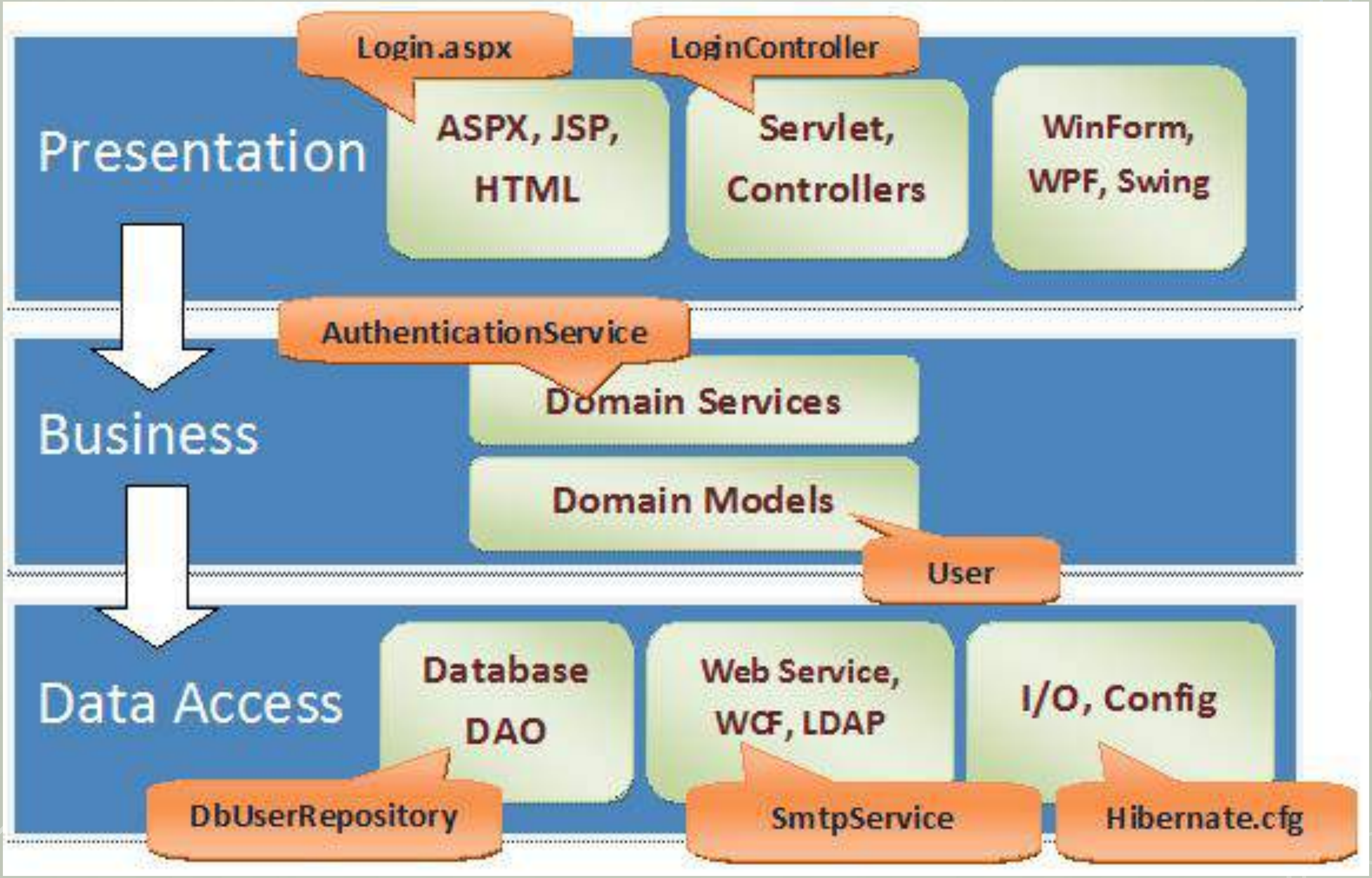
- Due to complexity, some layers within a layered architecture may have to be further decomposed (*e.g., into packages)*
- Sub-systems within a layer should have clearly defined boundaries and well specified interfaces to provide high levels of information hiding
- Each partition includes loosely coupled sub-systems, each delivering a single service or a coherent set of services



QUALITY PROPERTIES FOR LAYERED ARCHITECTURE

- Modifiability – dependencies are kept local within layer components
- Portability – services deal directly with a platform’s application programming interface (API)
- Security – encrypt and decrypt incoming or outgoing data.
- Reusability – easier to use

EXAMPLE LAYERED ARCHITECTURE



DATA-CENTERED SYSTEMS

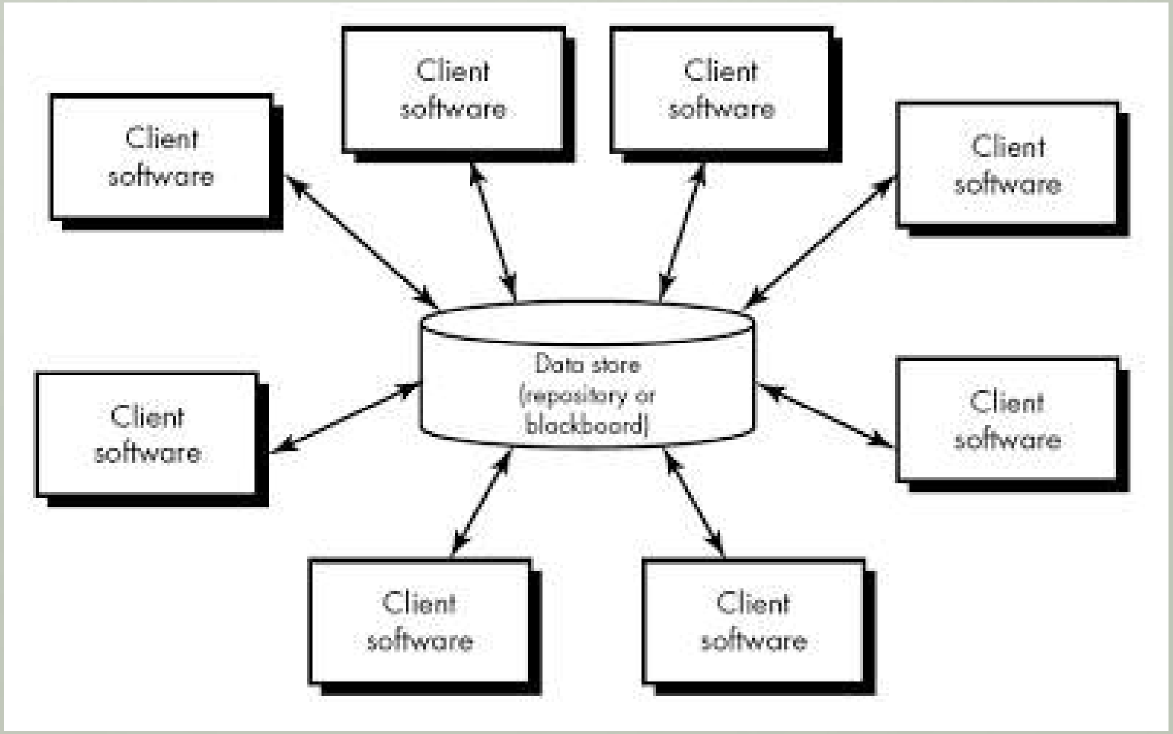
- A.k.a Shared Memory
- Primarily decomposed around a main central repository of data (Repository Architecture)
- Subsystems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems **(data is “global”)**;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used.

DATA-CENTERED SYSTEMS

- Example data-centered systems:
  - Expert systems – interact with a database management system for storing and retrieving knowledge information.
- Example architectural style/pattern:
  - Blackboard

BLACKBOARD

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



BLACKBOARD

- Client software accesses a central repository.
- In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software.
- A variation on this approach transforms the repository into a “blackboard” that sends notifications to client software when data of interest to the client change.

BLACKBOARD

- Existing components can be changed and new client components can be added to the architecture without concern about other clients (because the client components operate independently)
- Data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

ADVANTAGES OF BLACKBOARD

- Efficient way to share large amounts of data
- Sub-systems need not be concerned with how data is produced by centralised management e.g. backup, security, etc.
- Provides data integrity, backup and restore features.
- Provides scalability and reusability of agents as they do not have direct communication with each other.
- Reduces overhead of transient data between software components (Sharing model is published as the repository schema)

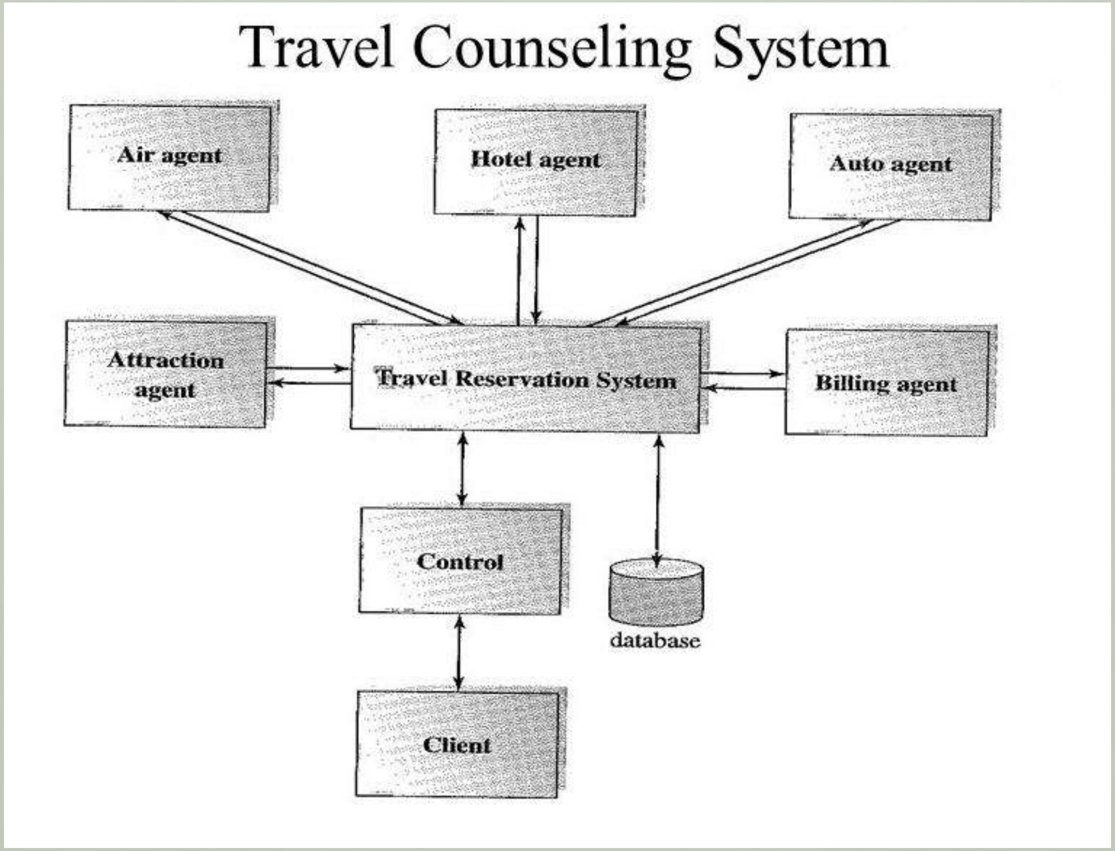
DISADVANTAGES OF BLACKBOARD

- Sub-systems must agree on a repository data model. Inevitably a compromise.
- Data evolution is difficult and expensive
- Changes in data structure highly affect the clients (No scope for specific management policies)
- Difficult to distribute efficiently.
- High dependency between data structure of data store and its agents.
- Cost of moving data on network for distributed data.

QUALITY PROPERTIES FOR BLACKBOARD

- Modifiability – easy to add or remove clients to fit new systems
- Reusability – specialized components can be reused easily on other applications
- Maintainability – allow separations of clients – maintaining existing components becomes easier.

EXAMPLE BLACKBOARD



DATA FLOW SYSTEMS

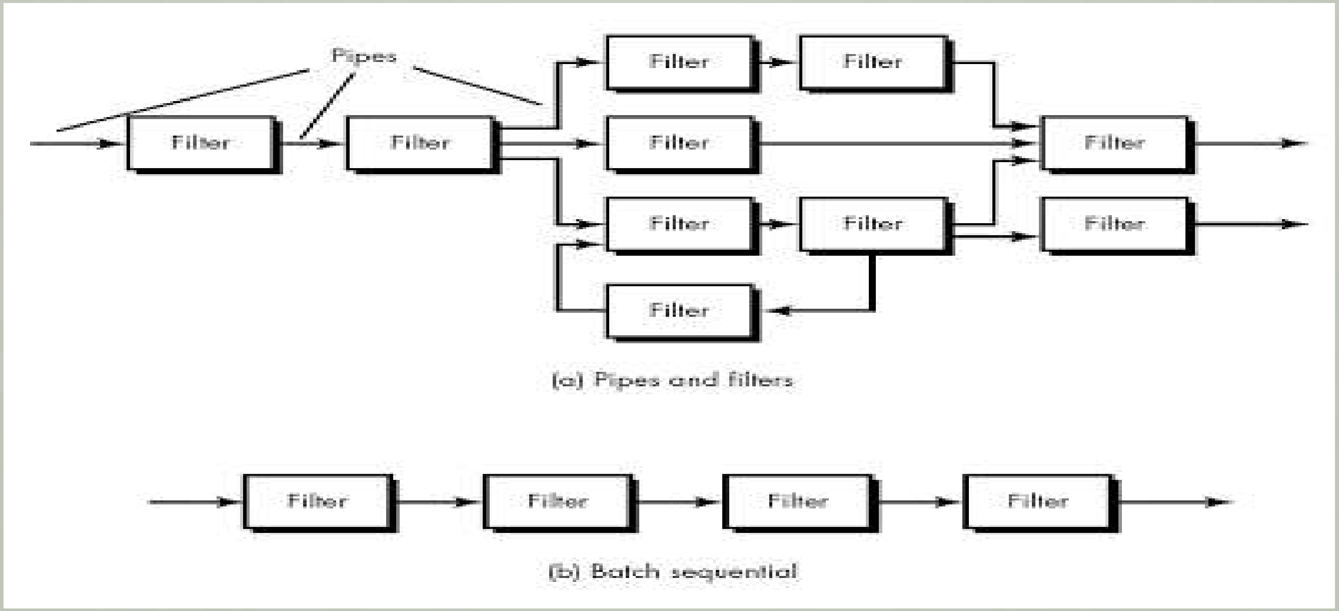
- Primarily decomposed around the central theme of transporting data and transforming the data along the way to meet application-specific requirements.
- Applied when input data are to be transformed through a series of computational or manipulative components into output data.
- Example architectural style:
  - Pipe and filter
  - Batch sequential

PIPE AND FILTER

- Has a set of components, called filters, connected by pipes that transmit data from one component to the next.
- **Filters** responsible for processing and transforming data.
- **Pipes** responsible for transferring data between components.
- 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.

PIPE AND FILTER

- If the data flow degenerates into a single line of transforms, it is termed **batch sequential**. This pattern accepts a batch of data and then applies a series of sequential components (filters) to transform it.



(a) Pipes and filters

(b) Batch sequential

QUALITY PROPERTIES PIPE AND FILTER

- Extensibility – processing filters can be added easily for more capabilities.
- Efficiency – by connecting filters in parallel, concurrency can be achieved to reduce latency in the system
- Reusability
- Modifiability – filters are independent from each other.
- Security – provide different types of security mechanisms to the data.
- Maintainability

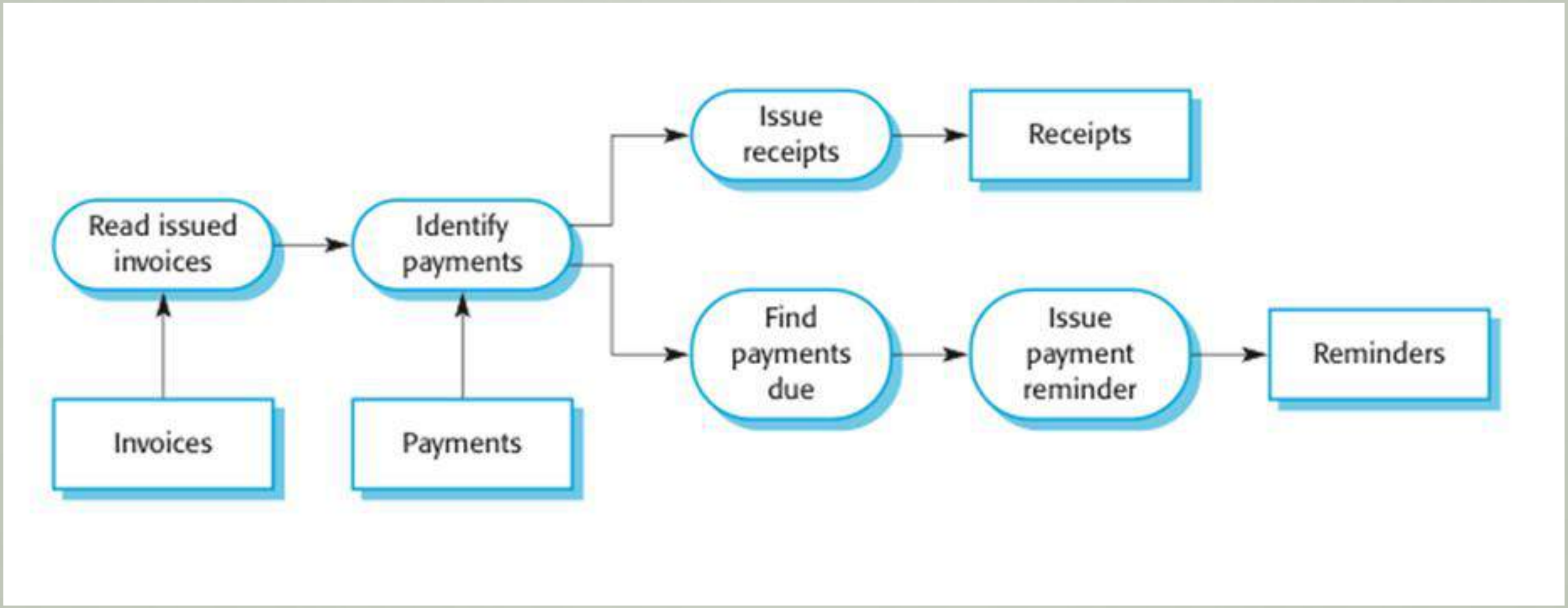
ADVANTAGES OF PIPE AND FILTER

- Filters can be modified or replaced easily, making it simple to change the module to fix a problem or modify its behaviour
- Filters can be rearranged with little effort, making it easy to develop several programs that do similar tasks
- Filters are highly reusable
- Concurrency is supported and is relatively easy to implement, provided synchronising pipes are available.

DISADVANTAGES OF PIPE AND FILTER

- Filters communicate only through pipes, making it difficult for them to coordinate their activities
- Filters usually consume and produce very simple data streams, such as streams of characters, which means that they may have to spend a lot of effort converting input into a usable form and then converting results back to a simple format for output
- Error handling is difficult; error information can only be output or passed along a pipe. The difficulty of error detection and recovery makes this style inappropriate when reliability and safety are important
- Gains from concurrency may be illusory. Pipes may not synchronise filters effectively, and some filters may need to wait for all their input before doing any output

EXAMPLE PIPE AND FILTER



DISTRIBUTED SYSTEMS

- Systems decomposed into multiple processes that collaborate through the network.
- Example architectural styles:
  - Client-server (2-tier Architecture)
  - N-tier
  - Broker
  - Peer-to-peer
  - Service-oriented

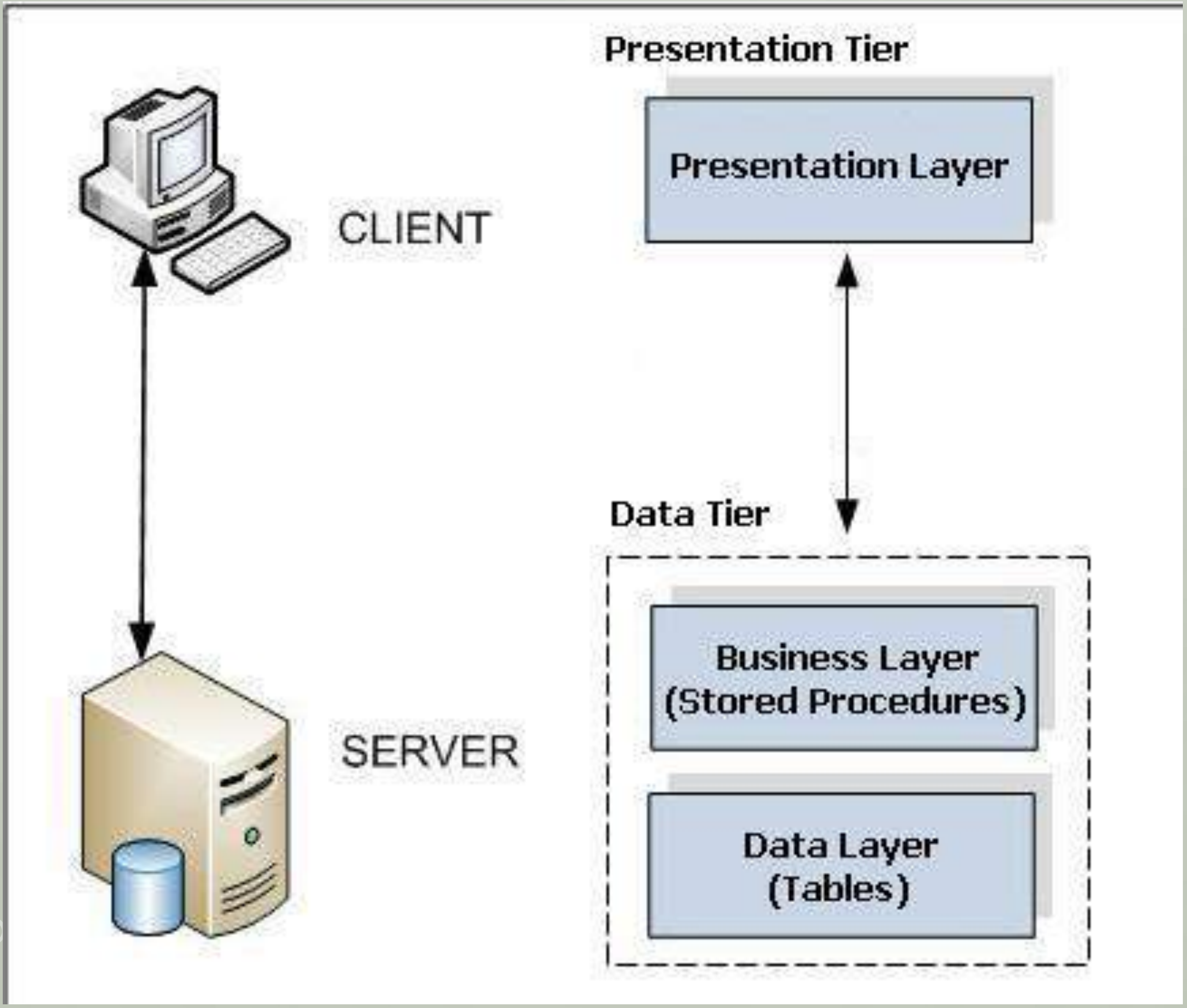
CLIENT-SERVER

- A.k.a 2-tier architecture.
- Decomposed of the **client** and the **server**.
- A server subsystem instance provides services to instances of other subsystem instances (the clients), which are responsible for interacting with the user.
- The request for a service is usually done via a remote procedure call mechanism or via a common object broker (*e.g.*, CORBA, Java RMI, or HTTP)
- Client depends on the services provided by the server.

CLIENT-SERVER

- Control flow in clients and servers is independent except for synchronisation to manage requests or to receive results
- Can be distributed over the network or within a single node.
- Useful for distributed systems with a large client base, since they provide localization of data in one central place.
- Client-server systems are not restricted to a single server:

CLIENT-SERVER



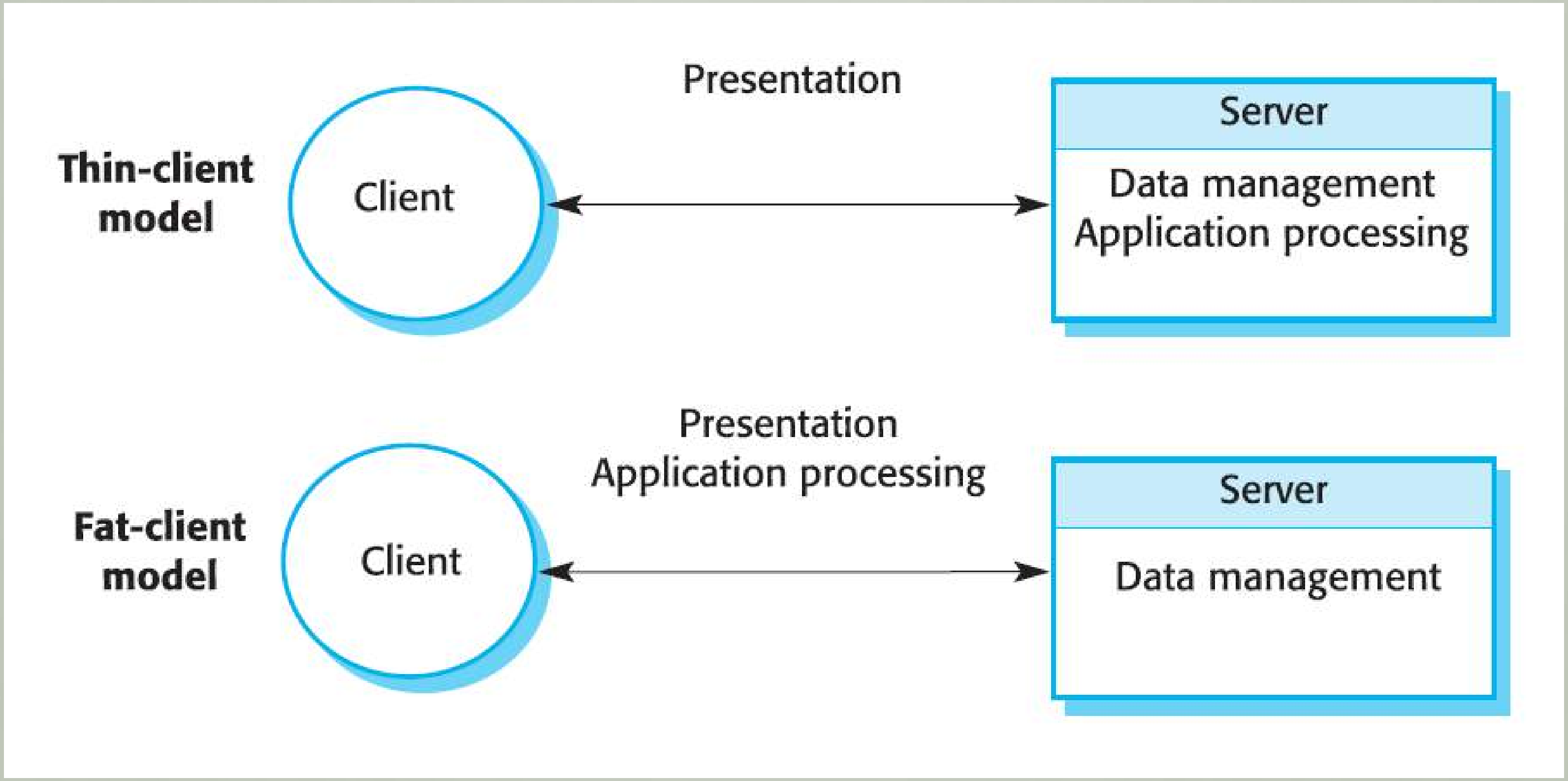
THIN AND FAT CLIENTS MODEL

- Thin-client model
  - In a thin-client model, all of the application processing and data management is carried out on the server. The client is simply responsible for running the presentation software.
  - Used when legacy systems are migrated to client server architectures.
    - The legacy system acts as a server in its own right with a graphical interface implemented on a client.
  - A major disadvantage is that it places a heavy processing load on both the server and the network.

THIN AND FAT CLIENTS MODEL

- Fat-client model (Thick-client)
  - In this model, the server is only responsible for data management. The software on the client implements the application logic and the interactions with the system user.
  - More processing is delegated to the client as the application processing is locally executed.
  - Most suitable for new client-server systems where the capabilities of the client system are known in advance.
  - More complex than a thin client model especially for management. New versions of the application have to be installed on all clients.

THIN AND FAT CLIENTS MODEL



ADVANTAGES OF CLIENT-SERVER

- Distribution of data is straightforward
- Makes effective use of networked systems. May require cheaper hardware
- Easy to add new servers or upgrade existing servers

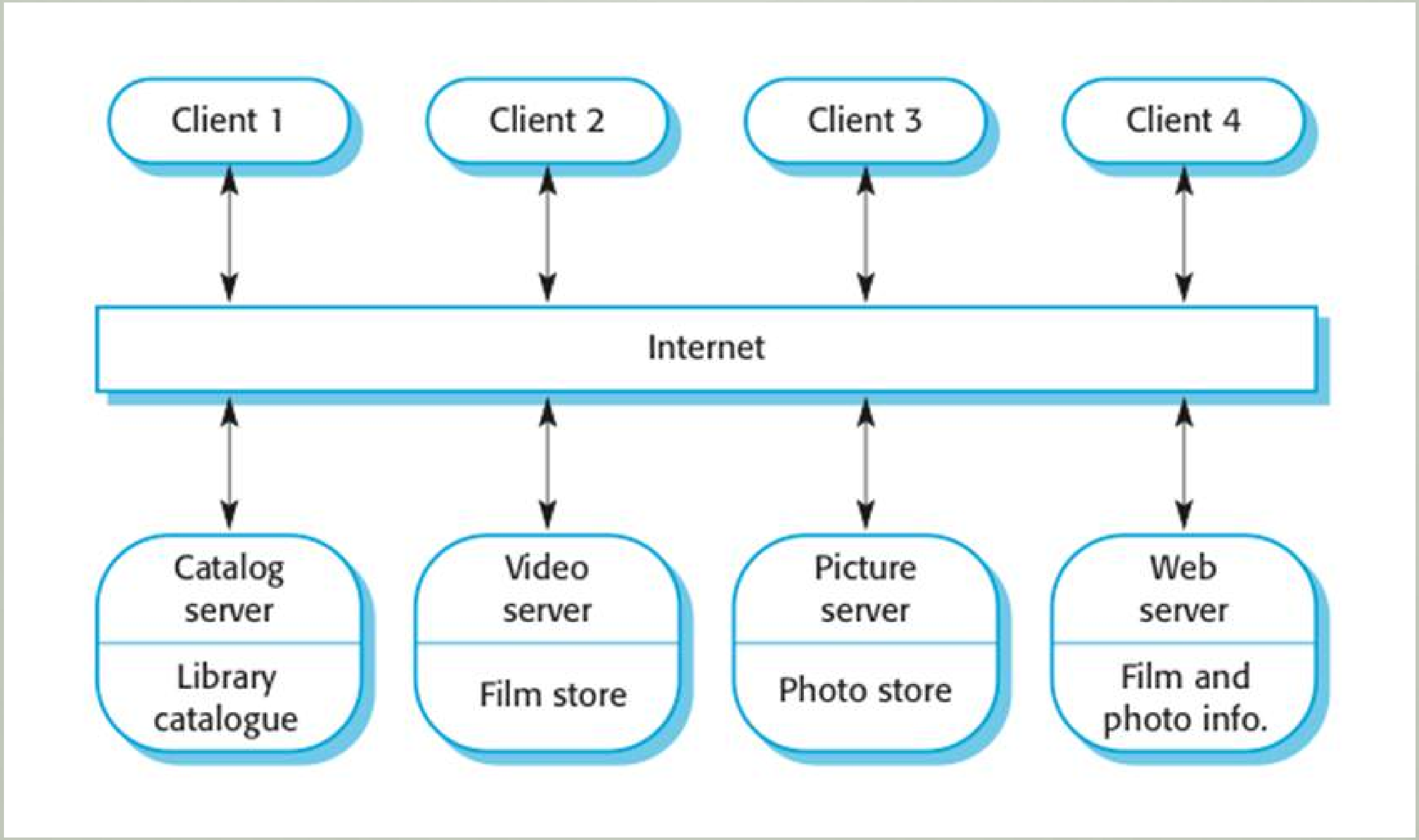
DISADVANTAGES OF CLIENT-SERVER

- No shared data model so sub-systems use different data organisation. Data interchange may be inefficient
- Redundant management in each server
- No central register of names and services -it may be hard to find out what servers and services are available

QUALITY PROPERTIES FOR CLIENT-SERVER

- Interoperability – allow clients on different platforms to interoperate with servers of different platforms.
- Modifiability – allows centralized changes in the server and quick distribution among many clients.
- Availability – by separating server data, multiple server nodes can be connected as backup to increase the server data or services' availability.
- Reusability – by separating server from clients, services or data provided by the server can be reused in different applications.

EXAMPLE CLIENT-SERVER



N-TIER

- **N-Tier** architecture usually has at least three separate logical parts, each located on separate physical server (A.k.a. 3-Tier Architecture).
- Each tier is responsible for a specific functionality.
- Each tier in the system is completely independent from all tiers except those immediately above and below it.
- Communication between tiers is typically asynchronous in order to support better scalability (allows the addition or removal of tiers).

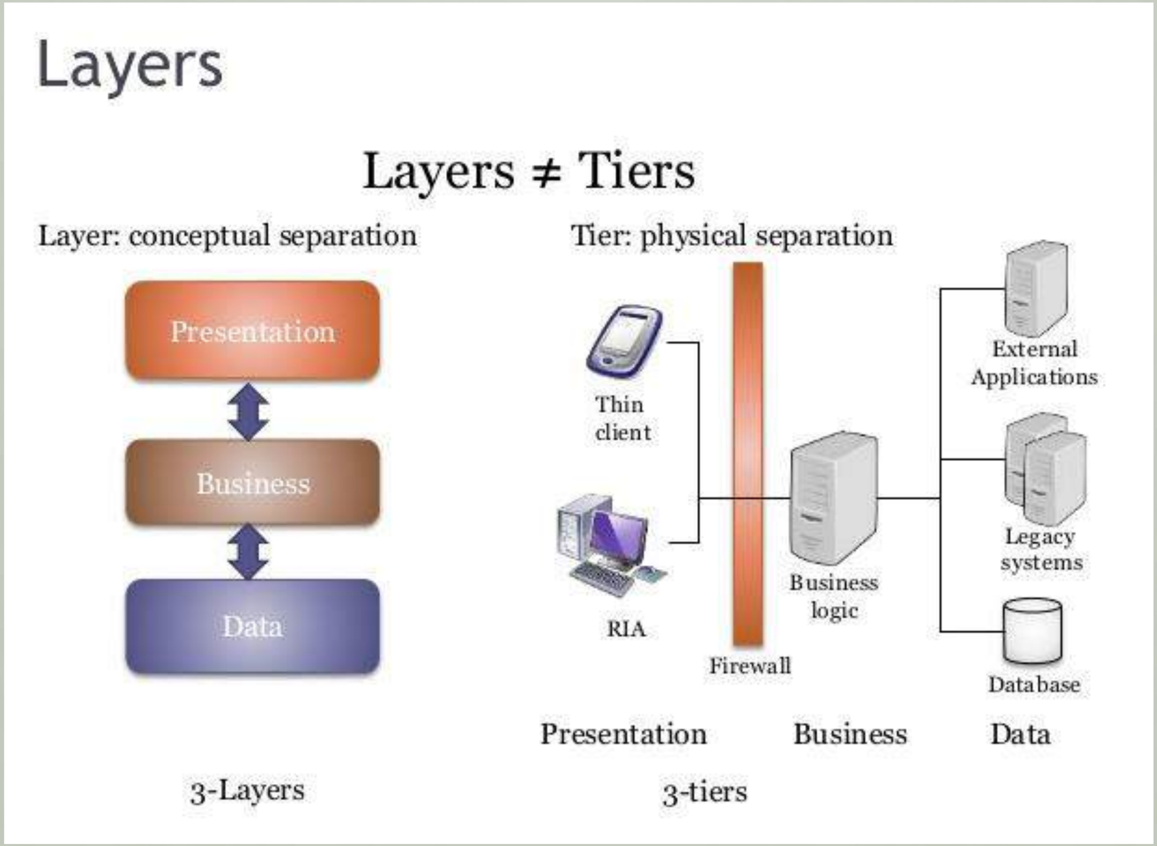
N-TIER & N-LAYER

- **N-tier** has close ties to the layered architecture
- **N-Layers** of application may reside on the same physical computer (same tier) and the components in each layer communicates with the components of other layer by well defined interfaces.
- Layered architecture focuses on the grouping of related functionality within an application into distinct layers that are stacked vertically on top of each other

N-TIER & N-LAYER

- Used N-tier if:
    - The layer in question requires an amount of resources that could lead to decreased performance of other components of the system if it were to be run on the same machine.
    - A layer needs access to sensitive information which is not needed in other layers, as it allows the data to be stored in a place where no other layers can possibly access it.

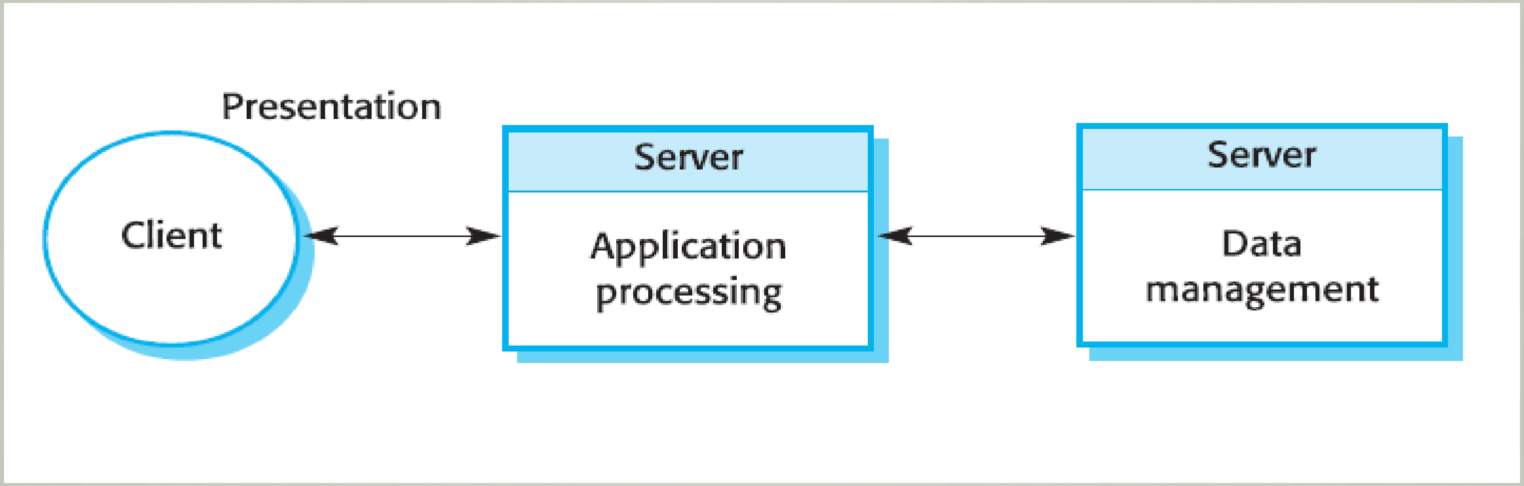
N-TIER & N-LAYER

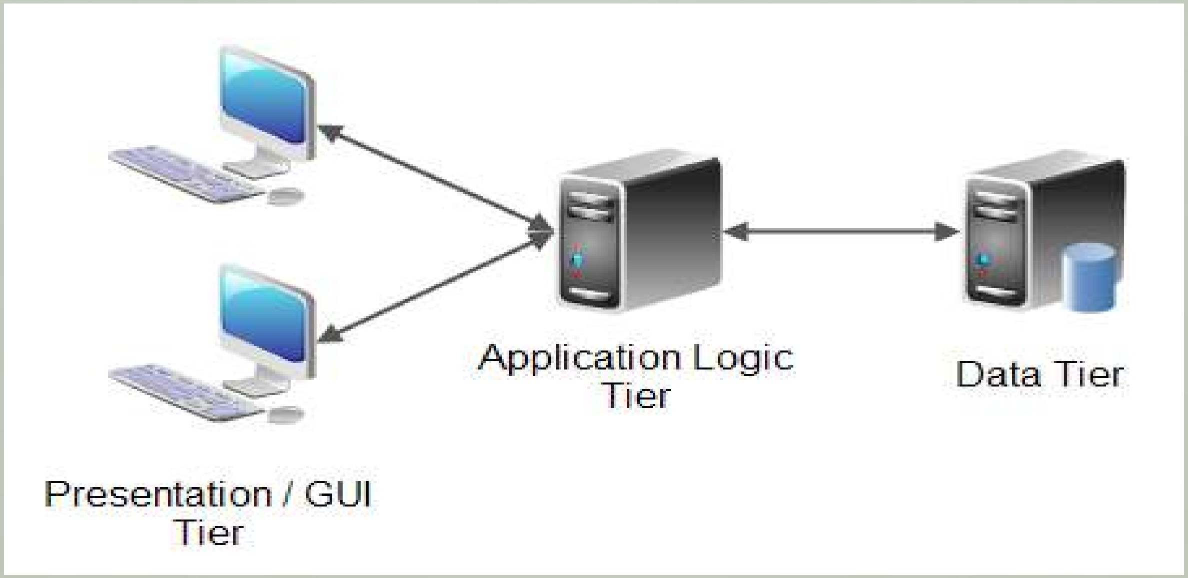


3-TIER ARCHITECTURE

- Each of the application layers may execute on a separate processor.
- Allows for better performance than a thin-client model and is simpler to manage than a fat-client model.
- A more scalable architecture - as demands increase, extra servers can be added.

3-TIER ARCHITECTURE





ADVANTAGES OF N-TIER

- Maintainability
    - Each tier is independent of the other tiers, which allows changes and patches to be performed on a part of the system without affecting the application as a whole.
- Scalability
    - Because tiers are designed based on the application's layers, scaling tiers is a simple process. The N-tier system is modular, allowing for easy scaling.

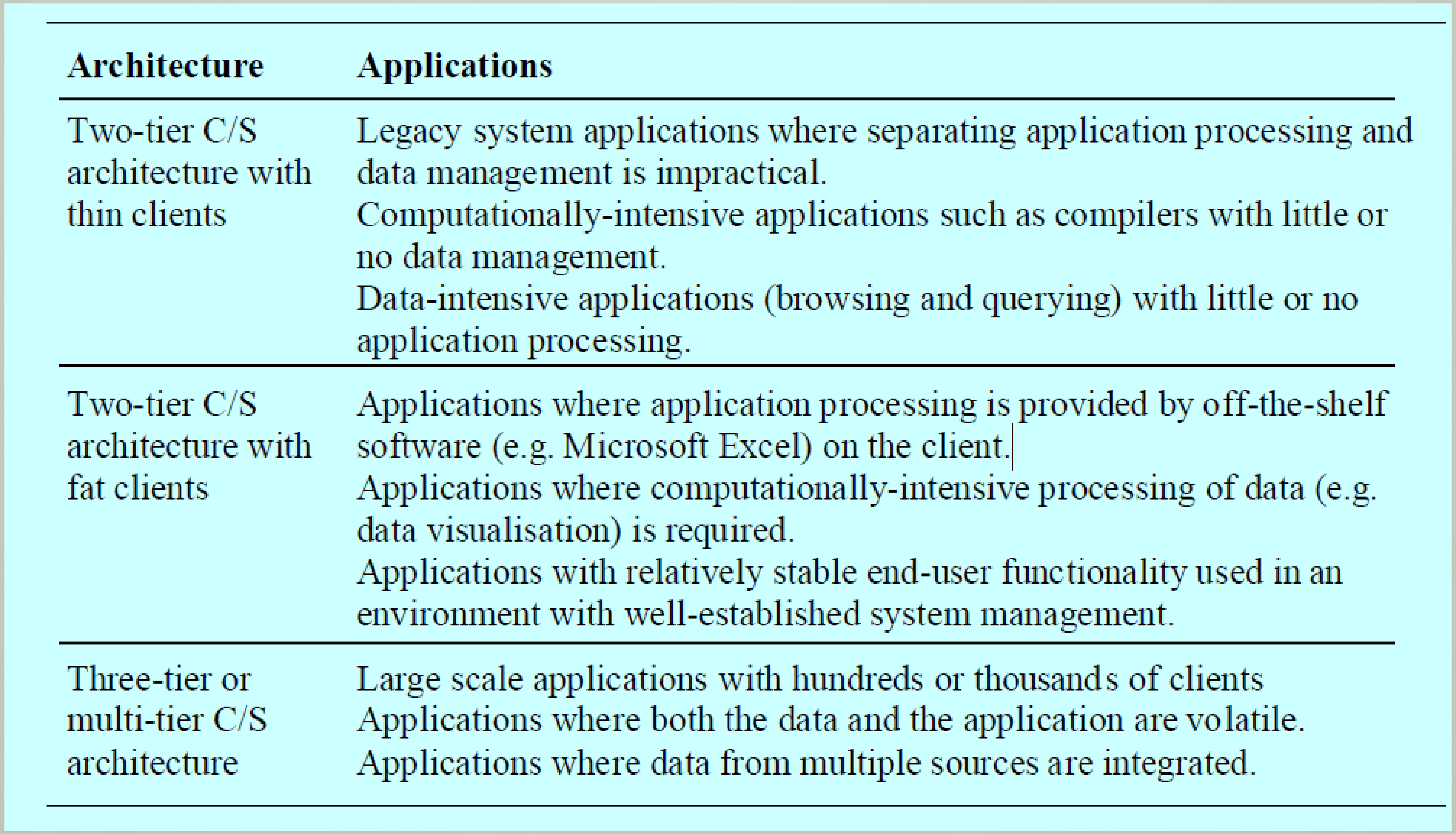
ADVANTAGES OF N-TIER

- Flexibility
  - Since each tier is mostly independent, they can be managed or changed independently, allowing for a flexible application.
- Availability
  - Since a tier is responsible for one component, it is easy for the application to increase the amount of a certain component, allowing higher availability.

DISADVANTAGES OF N-TIER

- Bloat
  - Since communication between tiers is needed and they are on different machines, programs can become much larger than they otherwise would be.
- Inflexibility
  - Although in most cases the use of the N-Tier architectural style increases flexibility, there are some instances in which having to communicate between multiple layers necessitates data be accessed in a specific way which might not be what is wanted.

USED OF CLIENT-SERVER



Architecture	**Applications**
Two-tier C/S architecture with thin clients	Legacy system applications where separating application processing and data management is impractical. Computationally-intensive applications such as compilers with little or no data management. Data-intensive applications (browsing and querying) with little or no application processing.
Two-tier C/S architecture with fat clients	Applications where application processing is provided by off-the-shelf software (e.g. Microsoft Excel) on the client. Applications where computationally-intensive processing of data (e.g. data visualisation) is required. Applications with relatively stable end-user functionality used in an environment with well-established system management.
Three-tier or multi-tier C/S architecture	Large scale applications with hundreds or thousands of clients Applications where both the data and the application are volatile. Applications where data from multiple sources are integrated.

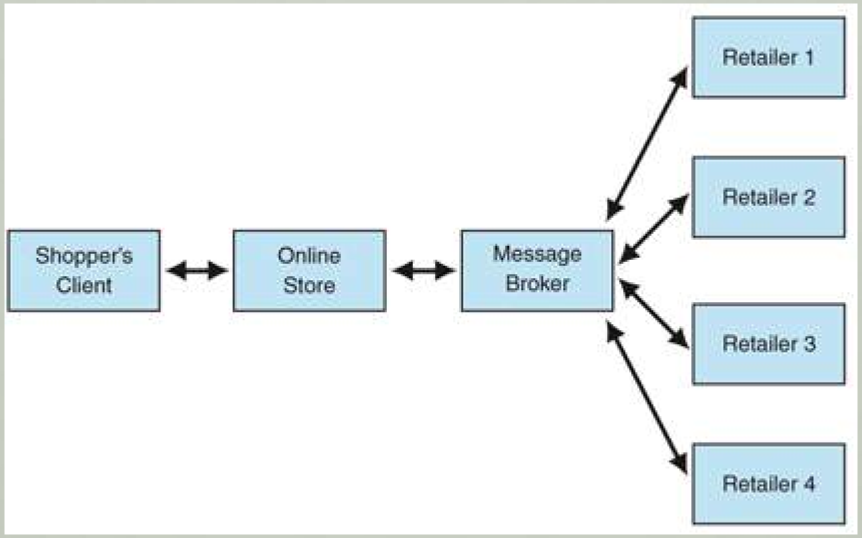
BROKER

- Provides mechanisms for achieving better flexibility between clients and servers in a distributed environment.
- Instead of accessing servers directly, clients access their functionality via a broker component, which locates appropriate servers, forwards requests, and relays responses back to clients.
- Decreases coupling between clients and servers.

QUALITY PROPERTIES FOR BROKER

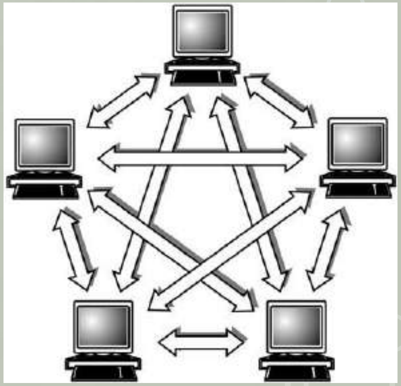
- Interoperability – allows clients on different platforms to interoperate with servers of different platforms
- Modifiability - allows centralized changes in the server and quick distribution among many clients
- Portability – by porting the broker to different platforms, services provided by the system can be easily acquired by new clients in different platforms.
- Reusability – when using broker, many complex services can be reused on other applications that require similar distributed operations.

EXAMPLE BROKER



PEER-TO-PEER (P2P)

- Type of decentralized and distributed architectural network that depends only individual nodes called “peers” as both suppliers and consumers of resources.
- No main server for receiving and delivering all the requests, instead the workload is distributed to all peers.
- It does not required a centralized server that hosts its files, since everyone has access to each other’s files through FTP (File Transfer Protocol)
- Examples:
  - USENET
  - File sharing services



PEER-TO-PEER (P2P)

- Advantages
  - More reliable as central dependency is eliminated
  - Overall cost of building and maintaining is comparatively less
- Disadvantages
  - The whole system is decentralized making it difficult to administer
  - Security in the system is very less
  - Data recovery and backup is difficult

SERVICE-ORIENTED ARCHITECTURE (SOA)

- As discussed in Chapter 2, a service-oriented architecture is essentially a collection of services.
    - These services communicate with each other.
    - The communication can involve either simple data passing or it could involve two or more services coordinating some activity.
    - Some means of connecting services to each other is needed.
- SOA is an evolution of distributed computing based on the request/reply design paradigm for synchronous and asynchronous applications.

SERVICE-ORIENTED ARCHITECTURE (SOA)

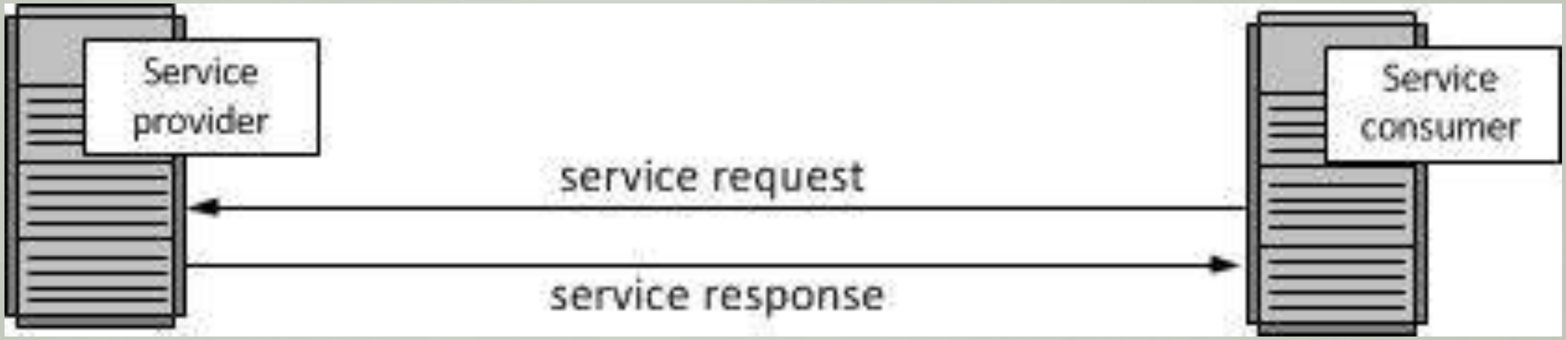
- Service:
  - is a function that is well-defined, self-contained, and does not depend on the context or state of other services.
  - A service is the endpoint of a connection.
  - Also, a service has some type of underlying computer system that supports the connection offered.
  - It is a program you interact with via message exchanges.

SERVICE-ORIENTED ARCHITECTURE (SOA)

- The technology of Web Services is the most likely connection technology of SOAs.
- Web Services refers to the technologies that allow for making connections.
- Services are what you connect together using Web Services.
- The combination of services—internal and external to an organization—make up a service-oriented architecture.

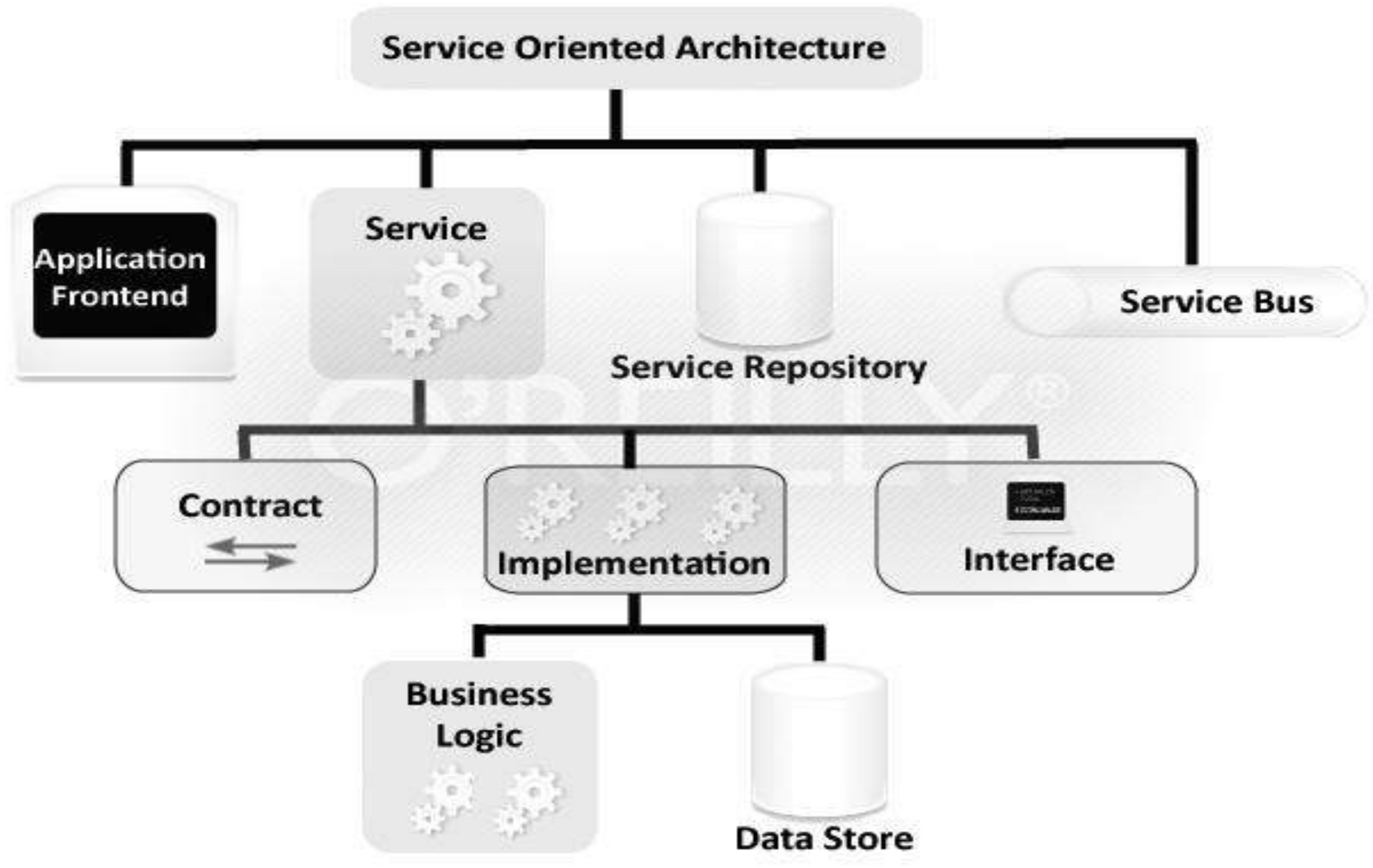
SERVICE-ORIENTED ARCHITECTURE (SOA)

- Basic SOA:



- It shows a service consumer at the right sending a service request message to a service provider at the left.
- The service provider returns a response message to the service consumer.
- The request and subsequent response connections are defined in some way that is understandable to both the service consumer and service provider.

SERVICE-ORIENTED ARCHITECTURE (SOA)



Layout of Service Oriented Architecture

ADVANTAGES OF SOA

- Organizational flexibility
  - Loosely coupled services ease the integration into other applications.
- Interoperability
  - Services are able to understand each other despite the language they were written in.
- Reusability & Maintainability
  - Changes made to one component in SOA will not ripple outward and affect others. Reusable services can be easily plugged into new applications without having to test them

DISADVANTAGES OF SOA

- Not suitable for Graphical User Interface functionalities. (i.e. Map manipulation)
- Stand-alone or short lived applications that do not require request and response (i.e. word processing)
- Increased complexity in basic understanding and managing of SOA
- Security is an issue as it must be considered at service level rather than application level

INTERACTIVE SYSTEMS

- Systems that support user interactions, typically through user interfaces.
- Concentrate on 2 main qualities:
  - Usability – the degree of complexity involved when learning or using the system.
  - Modifiability
- Example systems include:
  - Gaming systems
  - Simulations
  - Internet applications

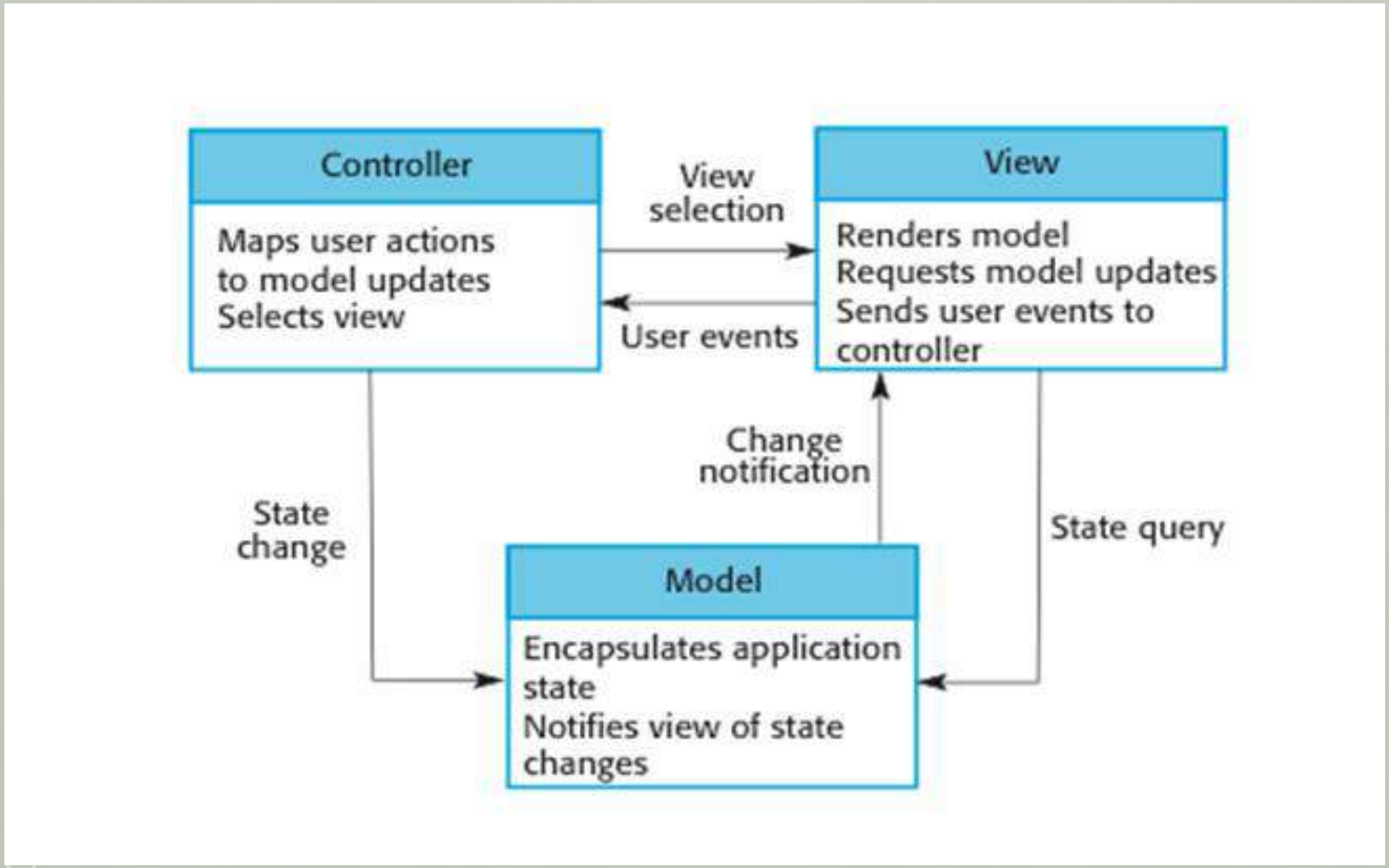
INTERACTIVE SYSTEMS

- The primary objective of is to separate the interaction of user from data abstraction and business data processing.
- Decomposes the system into three major partitions:
  - Data module − Data module provides the data abstraction and all business logic.
  - Control module − Control module identifies the flow of control and system configuration actions.
  - View presentation module − View presentation module is responsible for visual or audio presentation of data output and it also provides an interface for user input.

MODEL-VIEW-CONTROLLER (MVC)

- Require flexible incorporation of human-computer interfaces.
- Systems are decomposed into three main components:
  - Model – components that represents the system’s core including its major processing capabilities and data
  - View – components that represents the output representation of the system (e.g. graphical output or console based)
  - Controller – components(associated with a view) that handles user inputs (Respond to user action and direct the application flow)

MODEL-VIEW-CONTROLLER (MVC)



MODEL-VIEW-CONTROLLER (MVC)

- **Model**
  - Model is a central component of MVC that directly manages the data, logic, and constraints of an application. It consists of data components, which maintain the raw application data and application logic for interface.
  - It is an independent user interface and captures the behavior of application problem domain. Model is the domain-specific software simulation or implementation of the application's central structure.
  - If there is change in its state, it gives notification to its associated view to produce updated output, and the controller to change the available set of commands.

MODEL-VIEW-CONTROLLER (MVC)

- **View**
  - View can be used to represent any output of information in graphical form such as diagram or chart. It consists of presentation components which provide the visual representations of data.
  - Views request information from their model and generate an output representation to the user. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.

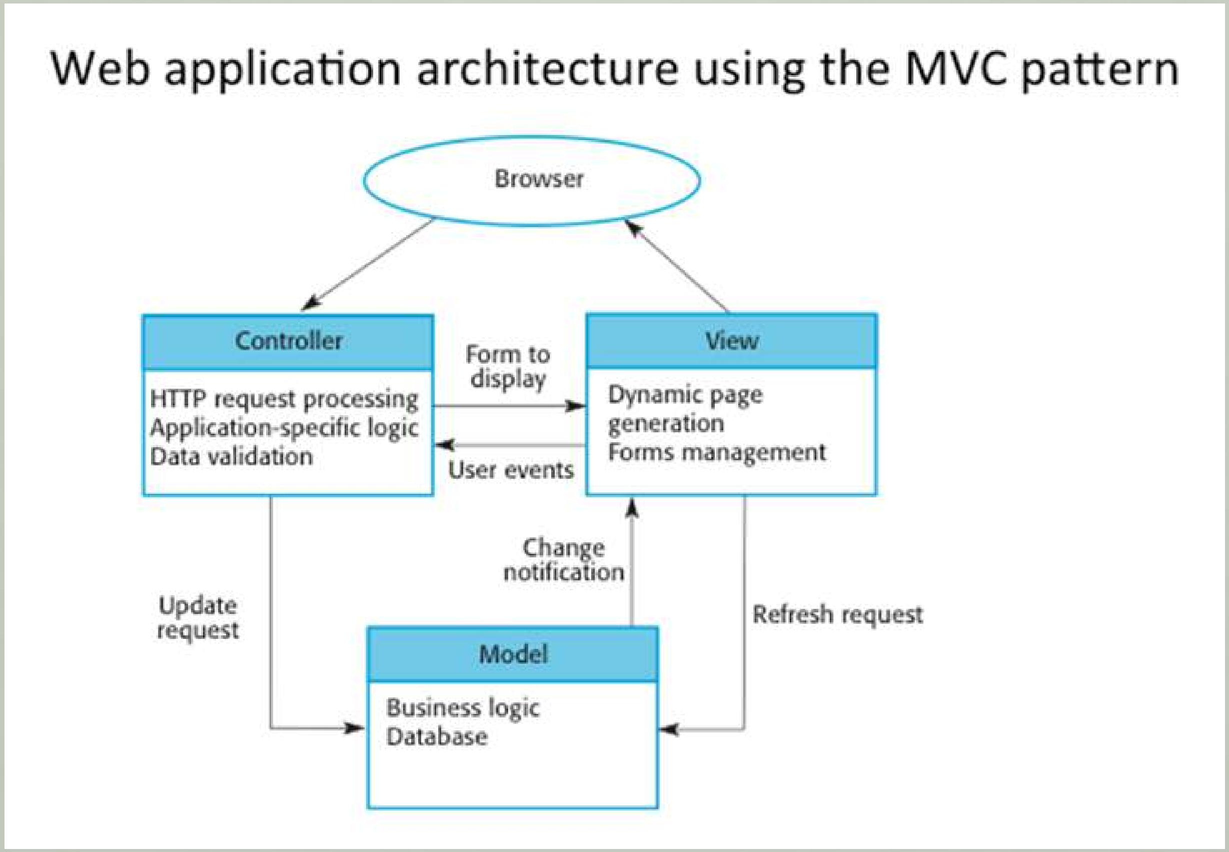
MODEL-VIEW-CONTROLLER (MVC)

- **Controller**
  - A controller accepts an input and converts it into commands for the model or view. It acts as an interface between the associated models and views and the input devices.
  - Controller can send commands to the model to update the model's state and to its associated view to change the view's presentation of the model.
  - It consists of input processing components, which handle input from the user by modifying the model.

QUALITY PROPERTIES OF MVC

- Modifiability – easy to exchange, enhance or add additional user interfaces
- Usability – by allowing easy exchangeability of user interfaces, systems can be configured with different user interfaces to meet different usability needs of particular groups of customers.
- Reusability – by separating the concerns of the model, view and controller components, they all can reused in other systems

EXAMPLE MVC


Web application architecture using the MVC pattern
Browser
Controller
HTTP request processing
Application-specific logic
Data validation
Form to display
User events
View
Dynamic page generation
Forms management
Update request
Change notification
Refresh request
Model
Business logic
Database

SUMMARY

- This chapter the types of architecture structures.
- It also described 4+1 view model of a software architecture.
- It also discuss the different categories of architectural patterns for different types of software systems such as layered, pipe and filter and MVC.