SOFTWARE ARCHITECTURE

"Software Architecture refers to the high-level structures of a software system and the discipline of creating such structures and systems. Each structure comprises software elements, relations among them, and properties of both elements and relations."
It is the fundamental organization of a system embodied in its components. It acts as the blueprint or the plan of making one software. It represents the core functions of every software. It is the basic structure of every software that you see.

An **architectural pattern** is a general, reusable solution to a commonly occurring problem in software architecture within a given context. Architectural patterns are similar to software design pattern but have a broader scope.

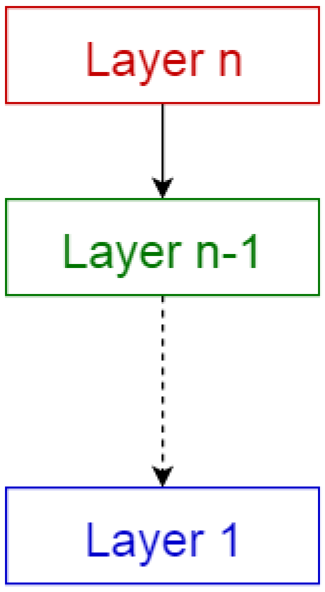
1. LAYERED PATTERN

- This pattern can be used to structure programs that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction. Each layer provides services to the next higher layer.
- It is also known as the n-tier software architecture pattern.
- It is a standard pattern for most Java EE applications and most architects, developers and designers use it. It is a common choice for business application development.
- The most commonly found 4 layers of a general information system are as follows.

  - **Presentation layer** (also known as **UI layer**)
  - **Application layer** (also known as **service layer**)
  - **Business logic layer** (also known as **domain layer**)
  - **Data access layer** (also known as **persistence layer**)

Usage

- General desktop applications.
- E commerce web applications.



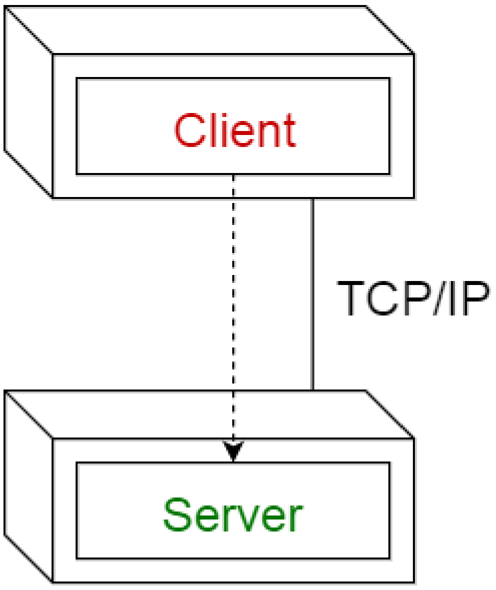
Layered pattern

2. CLIENT-SERVER PATTERN

- ➢ It is a distributed application structure. It partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters called clients.
- ➢ Clients and servers communicate over a computer network on separate hardware. Both the client and server might reside in the same system.
- ➢ Server host runs server programs which share the resources with clients. A client does not share any of its resources but requests the server's content or service function in software architecture.
- ➢ Servers further are classified as stateless or stateful. Clients of the stateful server make composite requests, enabling more conversational or transactional interactions between the client and the server.
- ➢ The stateful server keeps the record of requests from every client, and these records are called sessions.
- ➢ This pattern consists of two parties; a **server** and multiple **clients**. The server component will provide services to multiple client components. Clients request services from the server and the server provides relevant services to those clients. Furthermore, the server continues to listen to client requests.

Usage

- Online applications such as email, document sharing and banking.



Client-server pattern

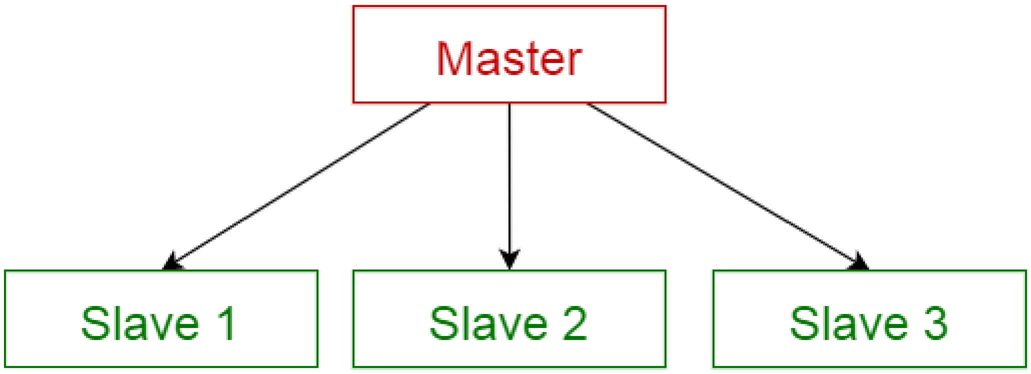
3. MASTER-SLAVE PATTERN

- ➢ This pattern consists of two parties; **master** and **slaves**. The master component distributes the work among identical slave components, and computes a final result from the results which the slaves return.
- ➢ In the master-slave pattern, all the subtasks are identical. It helps in coordinating efficiently.
- ➢ Transparency between master and slave helps clients. A client requests the master to perform the task, and thus the work is given to the slaves for completion.
- ➢ The Master-Slave pattern is a fundamental software architecture that many developers use.
- ➢ If there is a need for implementing two or more processes to be run simultaneously with different rates, a master-slave pattern works the best.
- ➢ A Master-Slave pattern has multiple parallel loops, and each of these loops executes different tasks at disparate rates. The master loop is in control of the slave loops. It communicates with them through messaging architectures.

Usage

- In database replication, the master database is regarded as the authoritative source, and the slave databases are synchronized to it.

- Peripherals connected to a bus in a computer system (master and slave drives).



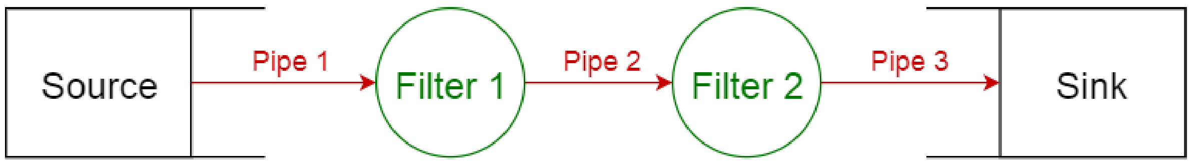
Master-slave pattern

4. PIPE-FILTER PATTERN

- This pattern can be used to structure systems which produce and process a stream of data. Each processing step is enclosed within a **filter** component. Data to be processed is passed through **pipes**. These pipes can be used for buffering or for synchronization purposes.
- A pipe-filter pattern is powerful in use and is robust software architectural pattern. It has 'n' number of components that filter data, before passing it on via connectors to other components in the structure.
- Every filter or component works at any given time. It helps in the simple sequence structure and also in extremely complex structures.
- There are four main components in this pattern; pump, filter, pipe and sink. The pipe-filter pattern works best for Unix programs, workflows in bioinformatics and compilers.
- When you have a lot of transformations to perform and need to be very flexible in using them, yet you want them to be robust, a pipe-filter pattern helps.

Usage

- Compilers. The consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation.
- Workflows in bioinformatics.



Pipe-filter pattern

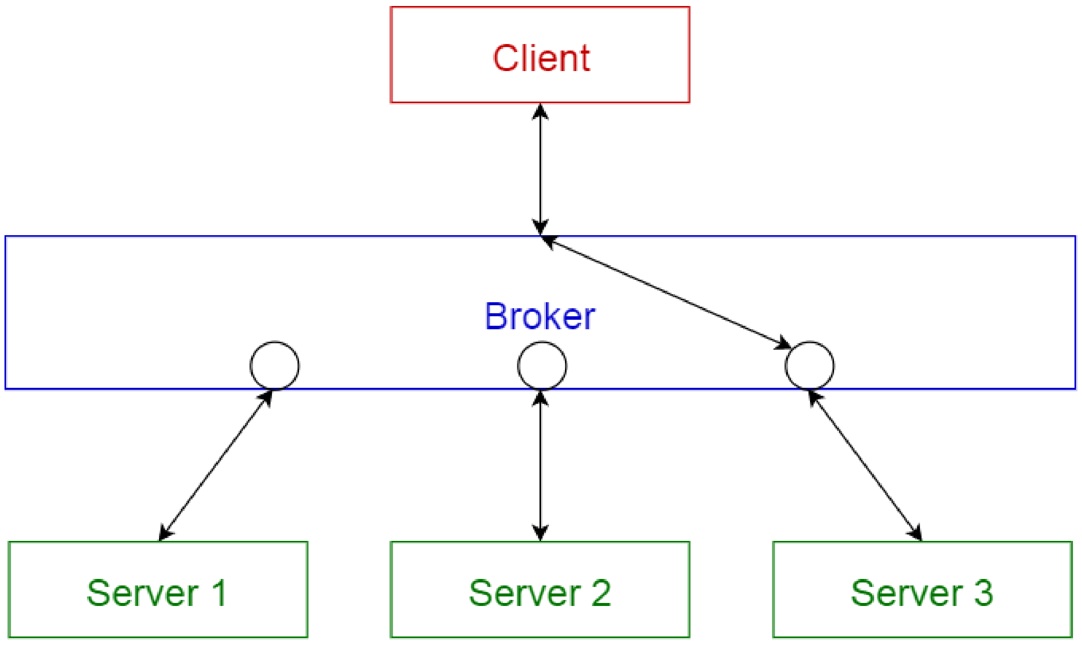
5. BROKER PATTERN

- This pattern is used to structure distributed systems with decoupled components. These components can interact with each other by remote service invocations. A **broker** component is responsible for the coordination of communication among **components**.
- The main components of the broker pattern are the broker, the server, and the client. A broker pattern also features proxies and bridges.

- Servers publish their capabilities (services and characteristics) to a broker. Clients request a service from the broker, and the broker then redirects the client to a suitable service from its registry.
- Brokers are the message-routing components of the software architecture. It sends and receives messages from the client and the server.
- The messages are mostly requests for services and replies to those requests. A broker pattern maintains the registry of the servers.

Usage

- Message broker software such as **Apache ActiveMQ**, **Apache Kafka**, **RabbitMQ** and **JBoss Messaging**.



Broker pattern

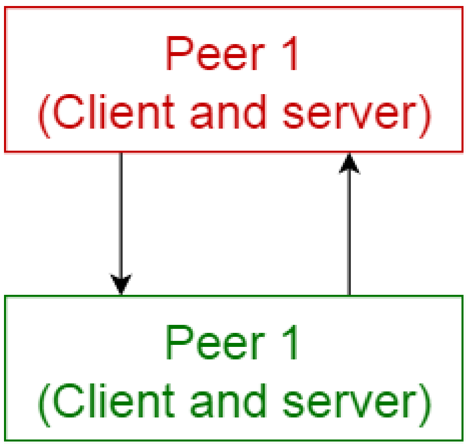
6. PEER-TO-PEER PATTERN

- In this pattern, individual components are known as **peers**. Peers may function both as a **client**, requesting services from other peers, and as a **server**, providing services to other peers. A peer may act as a client or as a server or as both, and it can change its role dynamically with time.
- Peers are dynamic in nature.
- Peer-to-peer patterns have many software applications and the most common amongst them is content distribution.
- It includes software publication and distribution, content delivery networks, streaming media and peercasting for multicasting streams, facilitating on-demand content delivery.

Usage

- File-sharing networks such as **Gnutella** and **G2**)
- Multimedia protocols such as **P2PTV** and **PDTP**.

- Cryptocurrency-based products such as **Bitcoin** and **Blockchain**



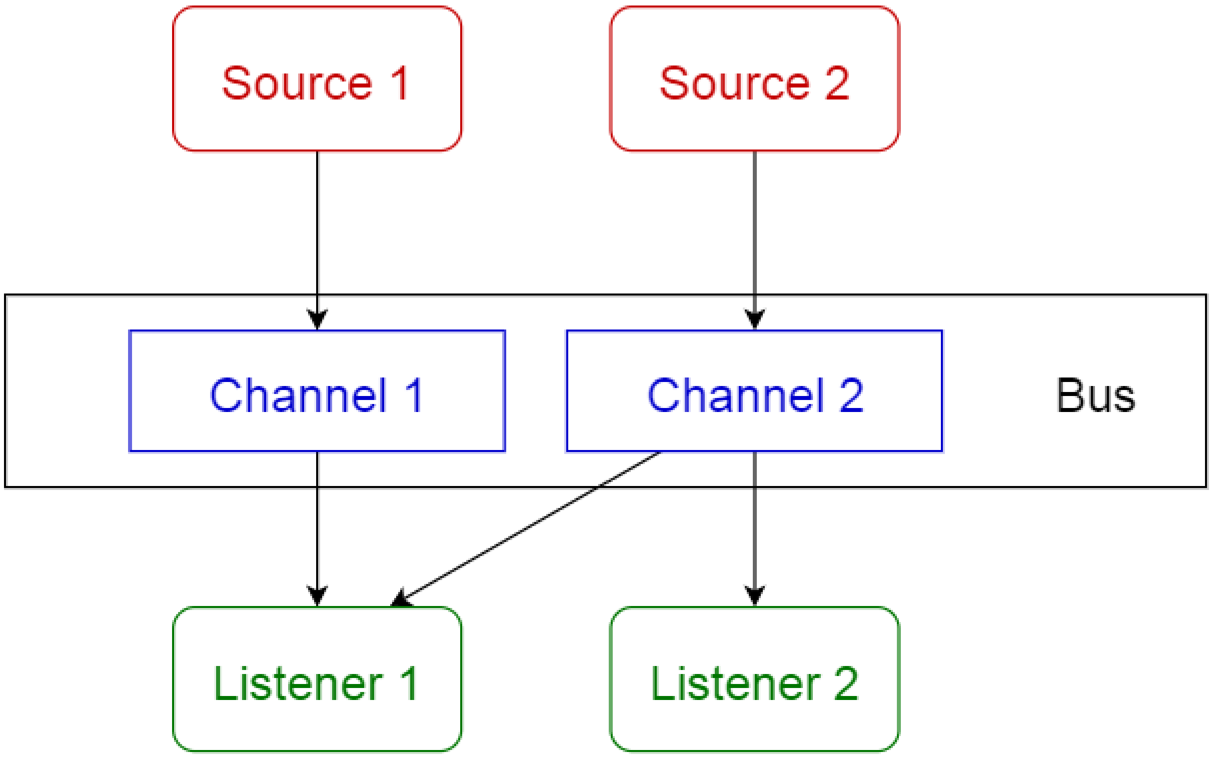
Peer-to-peer pattern

7. EVENT-BUS PATTERN

- ➢ An event-bus pattern allows different components to communicate with one another without knowing each other. Any component can send events to the Event-Bus without knowing who will pick it up or how many others will pick it up.
- ➢ Event-Bus Pattern works for events based structure
- ➢ This pattern primarily deals with events and has 4 major components; **event source**, **event listener**, **channel** and **event bus**. Sources publish messages to particular channels on an event bus. Listeners subscribe to particular channels. Listeners are notified of messages that are published to a channel to which they have subscribed before.
- ➢ references to other components, it can send events to an event-bus.
- ➢ It doesn't have to worry about who takes care of them. It allows easy development and splitting up of an application into several independent parts.

Usage

- Android app development
- Notification services



Event-bus pattern

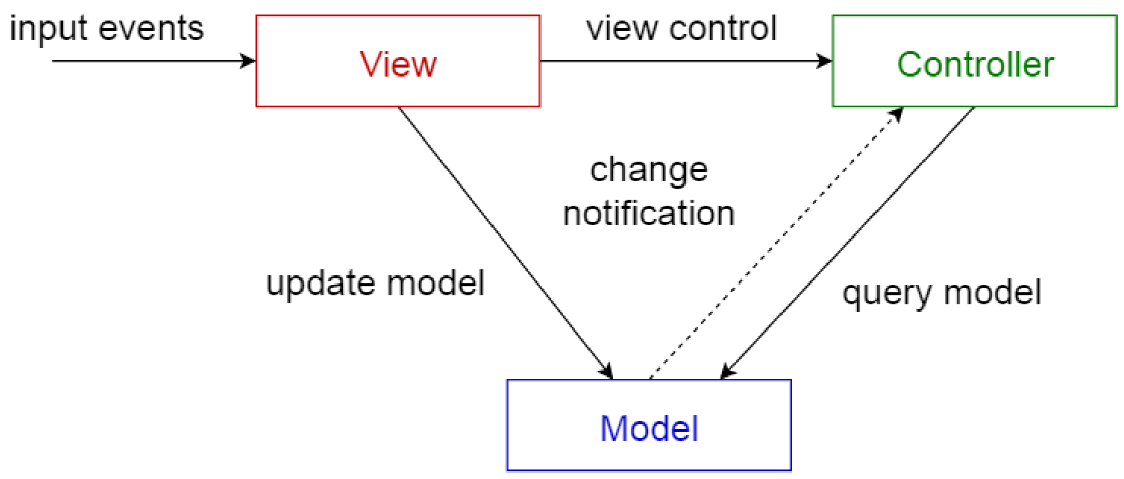
8. MODEL-VIEW-CONTROLLER PATTERN

- It is a design pattern that specifies an application consists of a data model, presentation and control information. Model View Controller pattern is also known as the MVC pattern.
- It is used to conduct operations separately. Model View Controller patterns work for web development applications.
- MVC pattern divides an interactive application into 3 parts; Model, View and Controller.
  1. **model** — Contains the core functionality and data. It is the lowest level of pattern responsible for maintaining data. It represents objects or JAVA POJO carrying data. It has logic to update controller if its data changes. A model consists of core functionality and data of an application. It is pure application data and contains no logic describing how to present the data to a user.
  2. **view** — View is responsible for the display of all or a portion of the data. It represents the visualization of data that the model contains and displays information to users.
  3. **controller** — handles the input from the user. It is the Software Code that controls the interactions between the Model and View. It acts on both model and view. It controls the data flow into the Model object and updates the View whenever there is any data change. It keeps the View and Model separate.

This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user. It decouples components and allows efficient code reuse.

Usage

- Architecture for World Wide Web applications in major programming languages.
- Web frameworks such as **Django** and **Rails**.



Model-view-controller pattern

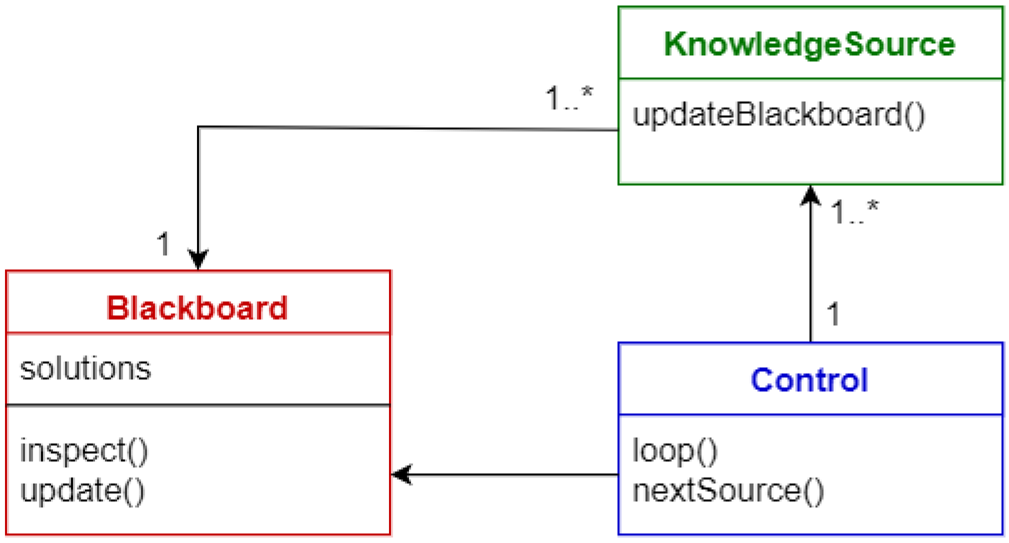
9. BLACKBOARD PATTERN

- It is also a behavioural design pattern. A Blackboard pattern provides a computational framework for the design and implementation of systems. The integration of large and diverse modules and complex, non-deterministic control strategies are in the blackboard pattern.
- It has three main components.
  - **blackboard** — a structured global memory containing objects from the solution space
  - **knowledge source** — specialized modules with their own representation. These are modules or set of algorithms that read information from the world and post the results to the Blackboard
  - **control component** — selects, configures and executes modules that read the results from the Blackboard and update the software accordingly.

All the components have access to the blackboard. Components may produce new data objects that are added to the blackboard. Components look for particular kinds of data on the blackboard, and may find these by pattern matching with the existing knowledge source.

Usage

- Speech recognition
- Vehicle identification and tracking
- Protein structure identification
- Sonar signals interpretation.



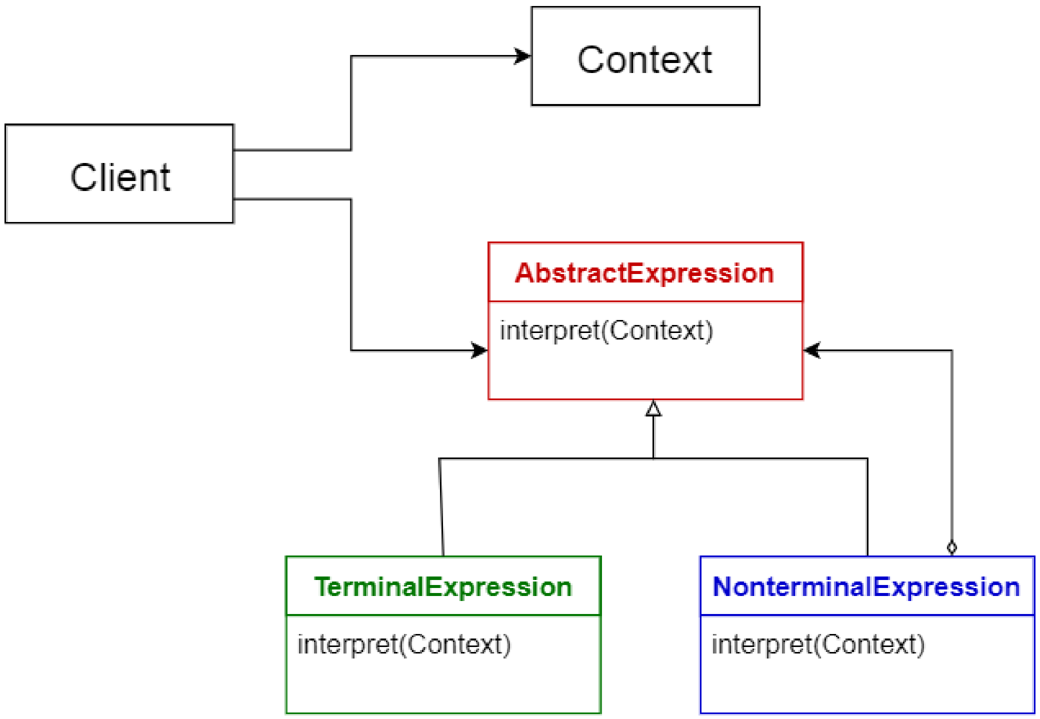
Blackboard pattern

10. INTERPRETER PATTERN

- It is one of the behavioural design patterns which analyses grammar and number representations. Interpreter pattern defines a domain language as a simple language grammar, representing the domain rules as language sentences, and interpreting these sentences to find a solution.
- Interpreter pattern uses a class to represent every grammar rule. It involves implementing the expression interface which interprets any given context. Interpreter pattern works in symbol processing engine, SQL parsing, etc.
- It depends on a hierarchy of expressions with each expression being terminal or non-terminal.
- It is slightly similar to the composite design pattern. Terminal expressions in the tree structure refer to the leaf objects and non-terminal expressions as composites.
- Google translator, Java compiler are some of the examples of the Interpreter Design pattern.

Usage

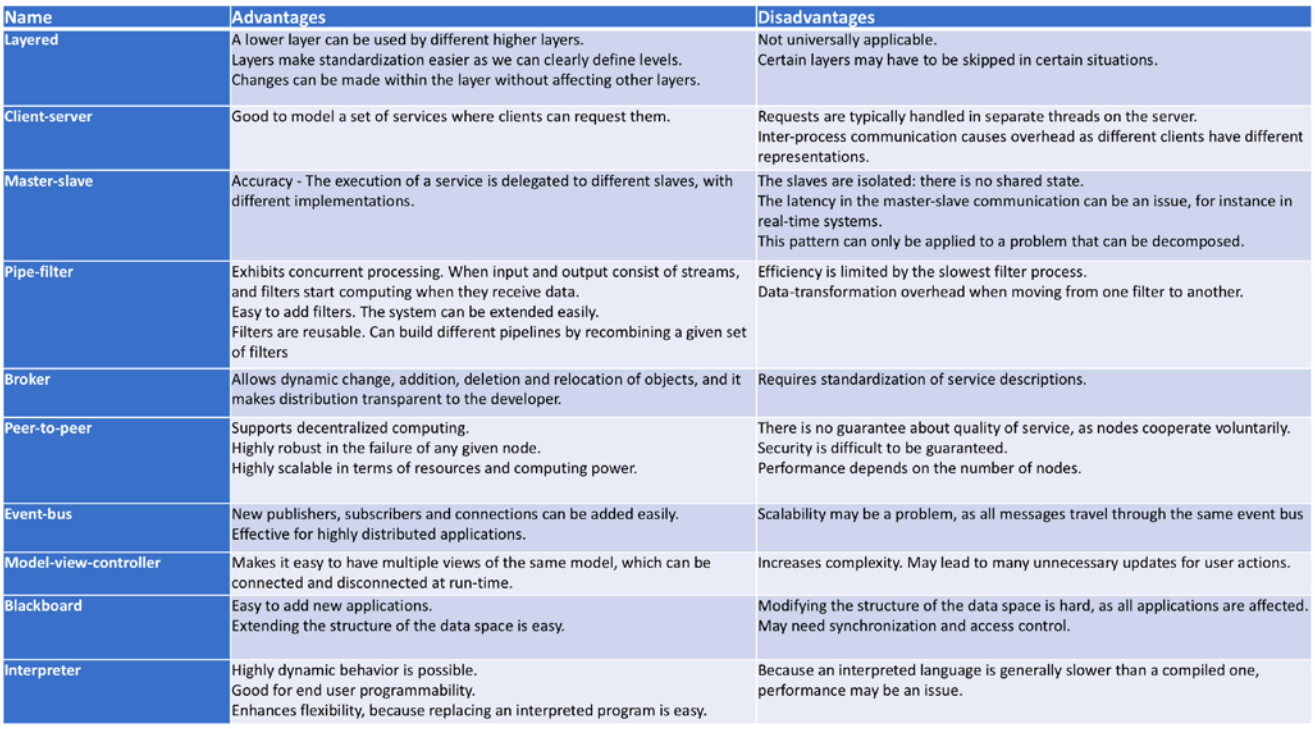
- Database query languages such as SQL.
- Languages used to describe communication protocols.



Interpreter pattern

Comparison of Architectural Patterns

The table given below summarizes the pros and cons of each architectural pattern.



Name	Advantages	Disadvantages
Layered	A lower layer can be used by different higher layers. Layers make standardization easier as we can clearly define levels. Changes can be made within the layer without affecting other layers.	Not universally applicable. Certain layers may have to be skipped in certain situations.
Client-server	Good to model a set of services where clients can request them.	Requests are typically handled in separate threads on the server. Inter-process communication causes overhead as different clients have different representations.
Master-slave	Accuracy - The execution of a service is delegated to different slaves, with different implementations.	The slaves are isolated: there is no shared state. The latency in the master-slave communication can be an issue, for instance in real-time systems. This pattern can only be applied to a problem that can be decomposed.
Pipe-filter	Exhibits concurrent processing. When input and output consist of streams, and filters start computing when they receive data. Easy to add filters. The system can be extended easily. Filters are reusable. Can build different pipelines by recombining a given set of filters	Efficiency is limited by the slowest filter process. Data-transformation overhead when moving from one filter to another.
Broker	Allows dynamic change, addition, deletion and relocation of objects, and it makes distribution transparent to the developer.	Requires standardization of service descriptions.
Peer-to-peer	Supports decentralized computing. Highly robust in the failure of any given node. Highly scalable in terms of resources and computing power.	There is no guarantee about quality of service, as nodes cooperate voluntarily. Security is difficult to be guaranteed. Performance depends on the number of nodes.
Event-bus	New publishers, subscribers and connections can be added easily. Effective for highly distributed applications.	Scalability may be a problem, as all messages travel through the same event bus
Model-view-controller	Makes it easy to have multiple views of the same model, which can be connected and disconnected at run-time.	Increases complexity. May lead to many unnecessary updates for user actions.
Blackboard	Easy to add new applications. Extending the structure of the data space is easy.	Modifying the structure of the data space is hard, as all applications are affected. May need synchronization and access control.
Interpreter	Highly dynamic behavior is possible. Good for end user programmability. Enhances flexibility, because replacing an interpreted program is easy.	Because an interpreted language is generally slower than a compiled one, performance may be an issue.