Types of Software Engineering Patterns:

**Software Architecture :  
Software architecture is the blueprint of building software. It shows the overall structure of the software, the collection of components in it, and how they interact with one another while hiding the implementation.**

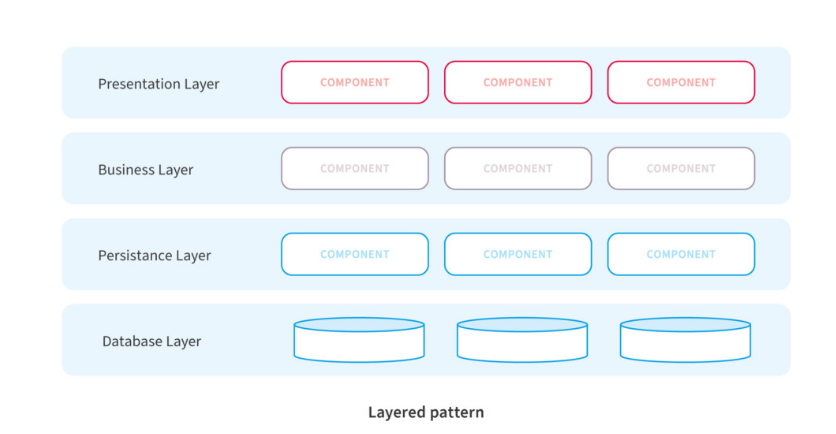
**1. Layered Pattern :**  
As the name suggests, components(code) in this pattern are separated into layers of subtasks and they are arranged one above another.

Each layer has unique tasks to do and all the layers are independent of one another. Since each layer is independent, one can modify the code inside a layer without affecting others.

It is the most commonly used pattern for designing the majority of software. This layer is also known as ‘N-tier architecture’. Basically, this pattern has 4 layers.

1. Presentation layer (The user interface layer where we see and enter data into an application.)
2. Business layer (this layer is responsible for executing business logic as per the request.)
3. Application layer (this layer acts as a medium for communication between the ‘presentation layer’ and ‘data layer’.
4. Data layer (this layer has a database for managing data.)

Ideal for:

E-commerce web applications development like Amazon.

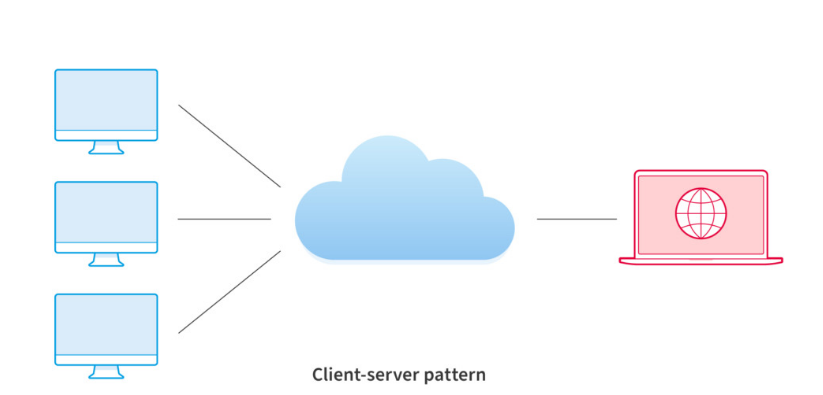
**2. Client-Server Pattern :**  
The client-server pattern has two major entities. They are a server and multiple clients.

Here the server has resources(data, files or services) and a client requests the server for a particular resource. Then the server processes the request and responds back accordingly.

Examples of software developed in this pattern:

* Email.
* WWW.
* File sharing apps.
* Banking, etc…

So this pattern is suitable for developing the kind of software listed in the examples.



### Master-slave:

“Master-slave architecture pattern” is useful when clients make multiple instances of the same request. The requests need simultaneous handling. Following are its’ key characteristics:

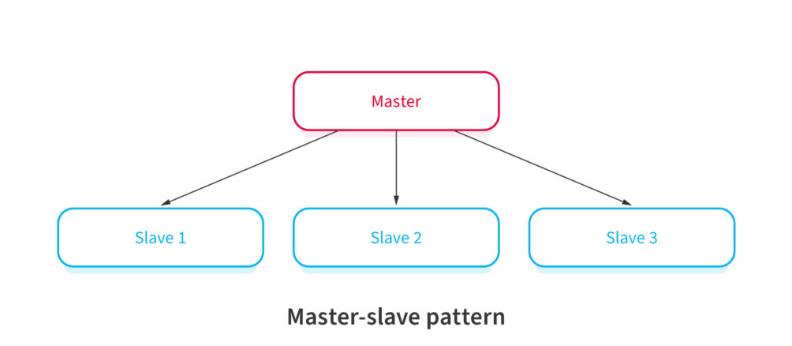
* The master launches slaves when it receives simultaneous requests.
* The slaves work in parallel, and the operation is complete only when all slaves complete processing their respective requests.

Advantages of this pattern are the following:

* Applications read from slaves without any impact on the master.
* Taking a slave offline and the later synchronization with the master requires no downtime.

Any application involving multi-threading can make use of this pattern, e.g., monitoring applications used in electrical energy systems.

There are a few disadvantages to this pattern, e.g.:

* This pattern doesn’t support automated fail-over systems since a slave needs to be manually promoted to a master if the original master fails.
* Writing data is possible in the master only.
* Failure of a master typically requires downtime and restart, moreover, data loss can happen in such cases.

### Pipe-filter :

Suppose you have complex processing in hand. You will likely break it down into separate tasks and process them separately. This is where the “Pipe-filter” architecture pattern comes into use. The following characteristics distinguish it:

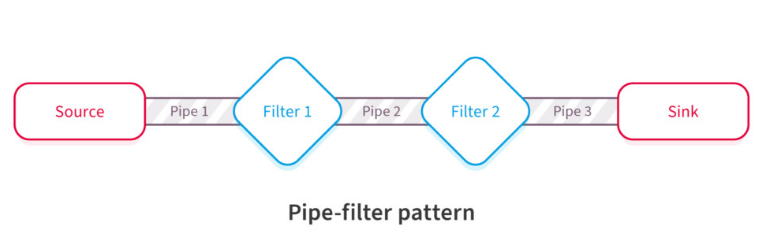
* The code for each task is relatively small. You treat it as one independent ‘filter’.
* You can deploy, maintain, scale, and reuse code in each filter.
* The stream of data that each filter processes pass through ‘pipes’.

Compilers often use this pattern, due to the following advantages:

* There are repetitive steps such as reading the source code, parsing, generating code, etc. These can be easily organized as separate filters.
* Each filter can perform its’ processing in parallel if the data input is arranged as streams using pipes.
* It’s a resilient model since the pipeline can reschedule the work and assign to another instance of that filter.

Watch out for a few disadvantages:

* This pattern is complex.
* Data loss between filters is possible in case of failures unless you use a reliable infrastructure.



### Broker :

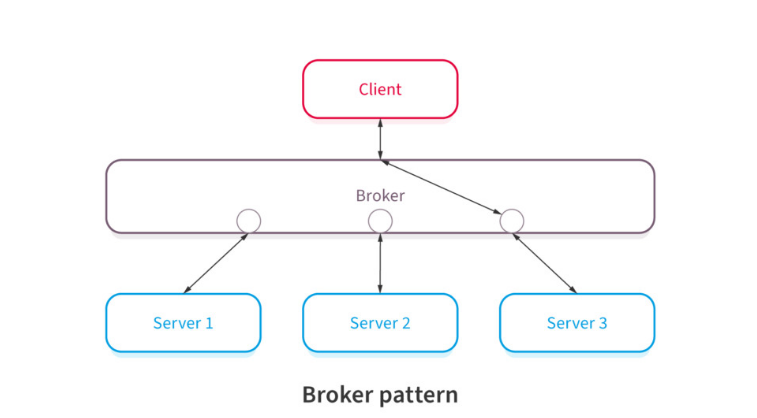
Consider distributed systems with components that provide different services independent of each other. Independent components could be heterogeneous systems on different servers, however, clients still need their requests serviced. “Broker architecture pattern” is a solution to this.

It has the following broad characteristics:

* A broker component coordinates requests and responses between clients and servers.
* The broker has the details of the servers and the individual services they provide.
* The main components of the broker architectural pattern are clients, servers, and brokers. It also has bridges and proxies for clients and servers.
* Clients send requests, and the broker finds the right server to route the request to.
* It also sends the responses back to the clients.

Message broker software like IBM MQ uses this pattern. The pattern has a few distinct advantages, e.g.:

* Developers face no constraints due to the distributed environment, they simply use a broker.
* This pattern helps using object-oriented technology in a distributed environment.



### Peer-to-peer (P2P) :

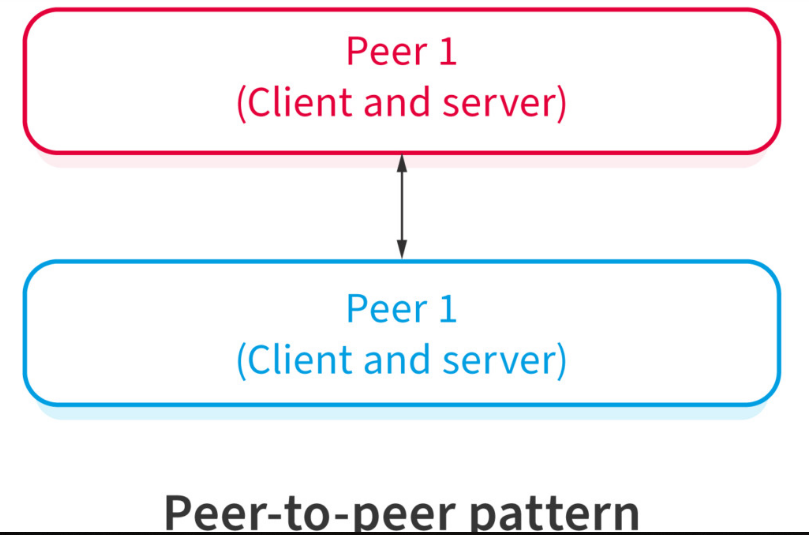
“Peer-to-peer (P2P) pattern” is markedly different from the client-server pattern since each computer on the network has the same authority. Key characteristics of the P2P pattern are as follows:

* There isn’t one central server, with each node having equal capabilities.
* Each computer can function as a client or a server.
* When more computers join the network, the overall capacity of the network increases.

File-sharing networks are good examples of the P2P pattern. Bitcoin and other cryptocurrency networks are other examples. The advantages of a P2P network are as follows:

* P2P networks are decentralized, therefore, they are more secure. You must have already heard a lot about the security of the Bitcoin network.
* Hackers can’t destroy the network by compromising just one server.

Under heavy load, the P2P pattern has performance limitations, as the questions surrounding the Bitcoin transaction throughout show.

e.x: “bitcoin”

### Event-bus pattern :

There are applications when components act only when there is data to be processed. At other times, these components are inactive. “Event-bus pattern” works well for these, and it has the following characteristics:

* A central agent, which is an event-bus, accepts the input.
* Different components handle different functions, therefore, the event-bus routes the data to the appropriate module.
* Modules that don’t receive any data pertaining to their function will remain inactive.

Think of a website using JavaScript. Users’ mouse clicks and keystrokes are the data inputs. The event-bus will collate these inputs and it will send the data to appropriate modules. The advantages of this pattern are as follows:

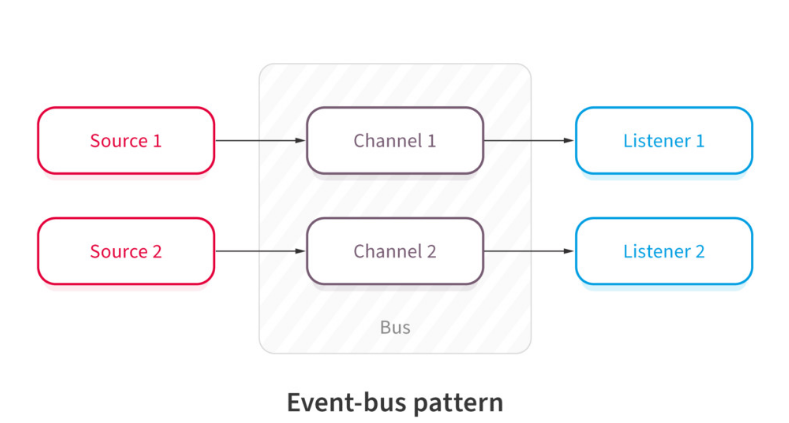
* This pattern helps developers handle complexity.
* It’s a scalable architecture pattern.
* This is an extensible architecture, new functionalities will only require a new type of events.

This software architecture pattern is also used in Android development.

Some disadvantages of this pattern are as follows:

* Testing of interdependent components is an elaborate process.
* If different components handle the same event require complex treatment to error-handling.
* Some amount of messaging overhead is typical of this pattern.

The development team should make provision for sufficient fall-back options in the event the event-bus has a failure.



### Model-View-Controller (MVC):

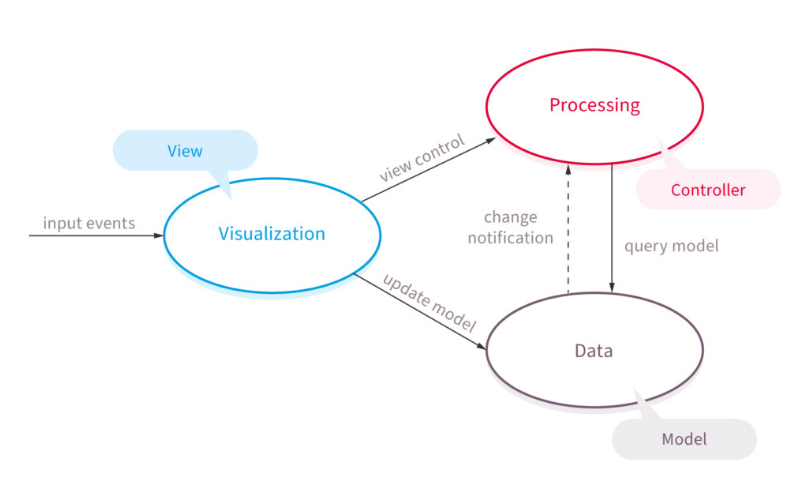
“Model-View-Controller (MVC) architecture pattern” involves separating an applications’ data model, presentation layer, and control aspects. Following are its’ characteristics:

* There are three building blocks here, namely, model, view, and controller.
* The application data resides in the model.
* Users see the application data through the view, however, the view can’t influence what the user will do with the data.
* The controller is the building block between the model and the view. View triggers events, subsequently, the controller acts on it. The action is typically a method call to the model. The response is shown in the view.

This pattern is popular. Many web frameworks like Spring and Rails use it, therefore, many web applications utilize this pattern. Its’ advantages are as follows:

* Using this model expedites the development.
* Development teams can present multiple views to users.
* Changes to the UI is common in web applications, however, the MVC pattern doesn’t need changes for it.
* The model doesn’t format data before presenting to users, therefore, you can use this pattern with any interface.

There are also a few disadvantages, for e.g.:

* With this pattern, the code has new layers, making it harder to navigate the code.
* ****There is typically a learning curve for this pattern, and developers need to know multiple technologies.

### Blackboard :

Emerging from the world of ‘Artificial Intelligence’ (AI) development, the “Blackboard architecture pattern” is more of a stop-gap arrangement. Its’ noticeable characteristics are as follows:

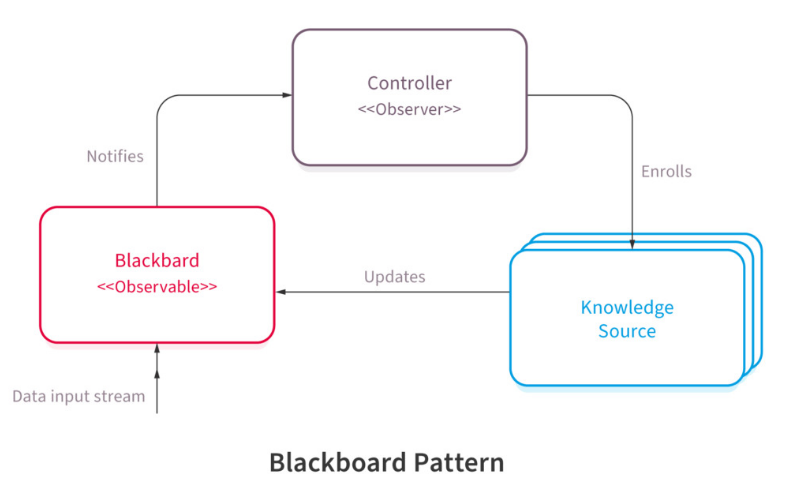
* When you deal with an emerging domain like AI or ‘Machine Learning’ (ML), you don’t necessarily have a settled architecture pattern to use. You start with the blackboard pattern, subsequently, when the domain matures, you adopt a different architecture pattern.
* There are three components, namely, the blackboard, a collection of knowledge resources, and a controller.
* The application system stores the relevant information in the blackboard.
* The knowledge resources could be algorithms in the AI or ML context that collect information and updates the blackboard.
* The controller reads from the blackboard and updates the application ‘assets’, for e.g., robots.

Image recognition, speech recognition, etc. use this architecture pattern. It has a few advantages, as follows:

* The pattern facilitates experiments.
* You can reuse the knowledge resources like algorithms.

There are also limitations, for e.g.:

* It’s an intermediate arrangement. Ultimately, you will need to arrive at a suitable architecture pattern, however, you don’t have certainty that you will find the right answer.
* All communication within the system happens via the blackboard, therefore, the application can’t handle parallel processing.
* Testing can be hard.



**Microservices Pattern :**

The collection of small services that are combined to form the actual application is the concept of microservices pattern. Instead of building a bigger application, small programs are built for every service (function) of an application independently. And those small programs are bundled together to be a full-fledged application.

So adding new features and modifying existing microservices without affecting other microservices are no longer a challenge when an application is built in a microservices pattern.

Modules in the application of microservices patterns are loosely coupled. So they are easily understandable, modifiable and scalable.

**Example** Netflix is one of the most popular examples of software built-in microservices architecture. This pattern is most suitable for websites and web apps having small components.