Architectural styles

What is software architectural styles?

- The **architectural style** is a very specific solution to a particular software, which typically focuses on how to organize the code created for the software. ... The architectural pattern is the description of relationship types and elements along with a set of constraints to implementing a software system.
- The **architectural pattern** is the description of relationship types and elements along with a set of constraints to implementing a software system. The patterns are usually reusable solutions for common problems or models.

Software Design Levels

Software design yields three levels of results:

- **Architectural Design -** The architectural design is the highest abstract version of the system.
- It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.

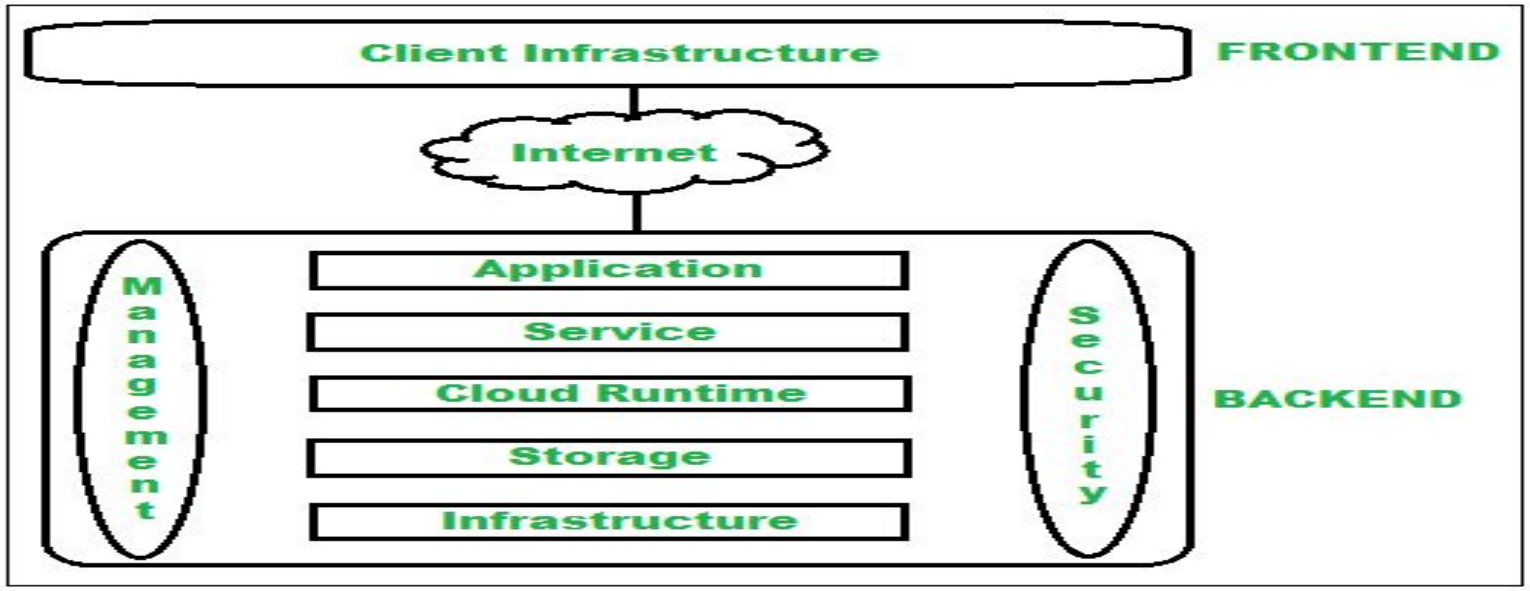
High-level Design

- **High-level Design-** The high-level design breaks the ‘single entity-multiple component’ concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other.
- High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.

Detailed Design-

- **Detailed Design-** Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs.
- It is more detailed towards modules and their implementations.
- It defines logical structure of each module and their interfaces to communicate with other modules.

- Architecture of cloud computing is the **combination of both SOA (Service Oriented Architecture) and EDA (Event Driven Architecture)**. Client infrastructure, application, service, runtime, storage, infrastructure, management and security all these are the components of cloud computing architecture.



What's the difference between an architectural style, an architectural pattern, and a system metaphor?

- An architectural style is a central, organizing concept for a system.
- An architectural pattern describes a coarse-grained solution at the level of subsystems or modules and their relationships.
- A system metaphor is more conceptual and it relates more to a real-world concept over a software engineering concept.

Architecture style	Description
Client-Server	Segregates the system into two applications, where the client makes a service request to the server.
Component-Based Architecture	Decomposes application design into reusable functional or logical components that are location-transparent and expose well-defined communication interfaces.
Layered Architecture	Partitions the concerns of the application into stacked groups (layers).
Message-Bus	A software system that can receive and send messages that are based on a set of known formats, so that systems can communicate with each other without needing to know the actual recipient.
N-tier / 3-tier	Segregates functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer.
Object-Oriented	An architectural style based on division of tasks for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object.
Separated Presentation	Separates the logic for managing user interaction from the user interface (UI) view and from the data with which the user works.
Service-Oriented Architecture (SOA)	Refers to Applications that expose and consume functionality as a service using contracts and messages.

Each of those architectural styles are applied to specific areas of interest:

Category	**Architecture styles**
Communication	Service-Oriented Architecture (SOA), Message Bus, Pipes and Filters
Deployment	Client/server, 3-Tier, N-Tier
Domain	Domain Model, Gateway
Interaction	Separated Presentation
Structure	Component-Based, Object-Oriented, Layered Architecture

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

1. Data-centered architecture

- The data store in the file or database is occupying at the center of the architecture.
- Store data is access continuously by the other components like an update, delete, add, modify from the data store.
- Data-centered architecture helps integrity.
- Pass data between clients using the blackboard mechanism.
- The processes are independently executed by the client components.

2. Data-flow architecture

- This architecture is applied when the input data is converted into a series of manipulative components into output data.
- A pipe and filter pattern is a set of components called as filters.
- Filters are connected through pipes and transfer data from one component to the next component.
- The flow of data degenerates into a single line of transform then it is known as batch sequential.

3. Call and return architectures

This architecture style allows to achieve a program structure which is easy to modify.

Following are the sub styles exist in this category:

1. Main program or subprogram architecture. The program is divided into smaller pieces hierarchically.

- The main program invokes many of program components in the hierarchy that program components are divided into subprogram.
- **2. Remote procedure call architecture.** The main program or subprogram components are distributed in network of multiple computers.
- The main aim is to increase the performance.

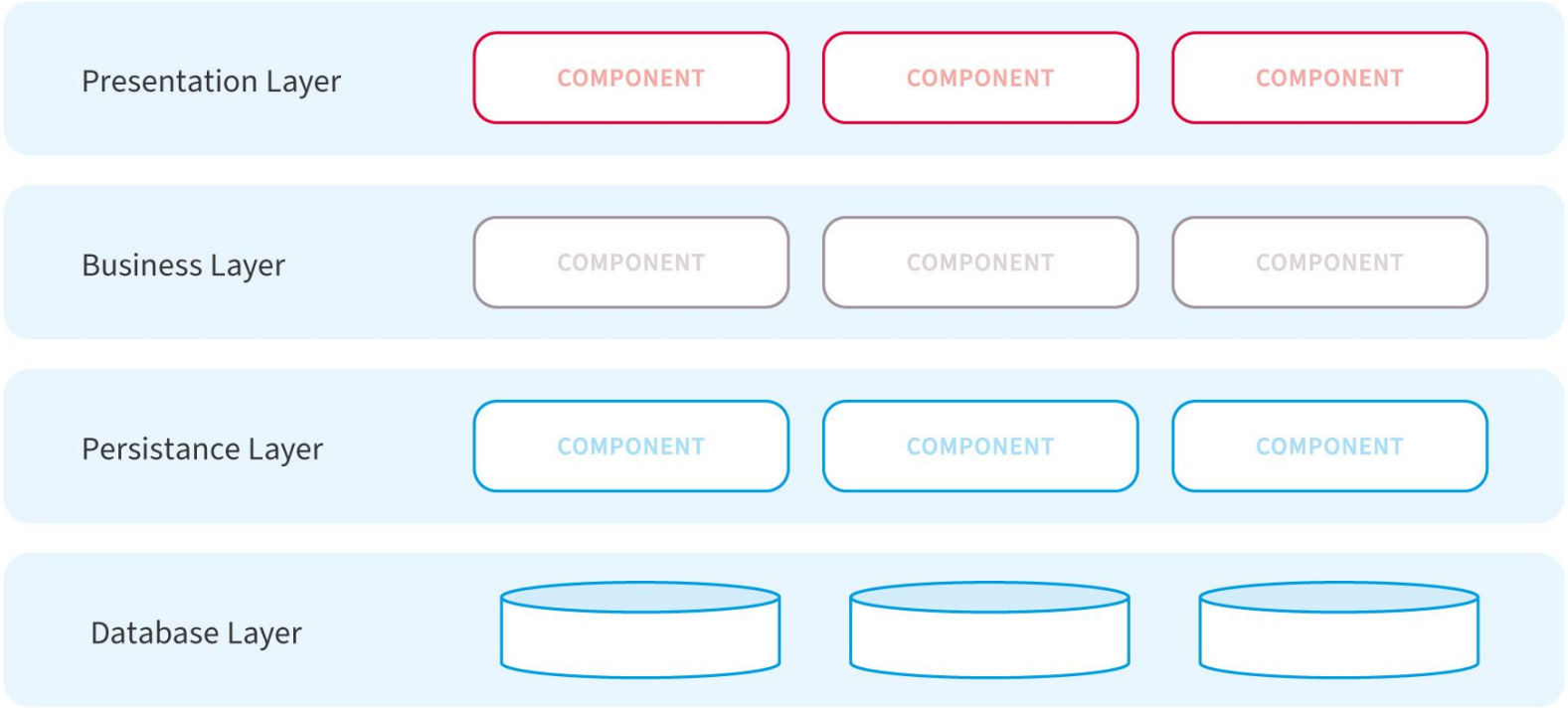
4. Object-oriented architectures

- This architecture is the latest version of call-and-return architecture.
- It consist of the bundling of data and methods.

5. Layered architectures

- The different layers are defined in the architecture. It consists of outer and inner layer.
- The components of outer layer manage the user interface operations.
- Components execute the operating system interfacing at the inner layer.
- The inner layers are application layer, utility layer and the core layer.
- In many cases, It is possible that more than one pattern is suitable and the alternate architectural style can be designed and evaluated.

Pattern #1: Layered architecture



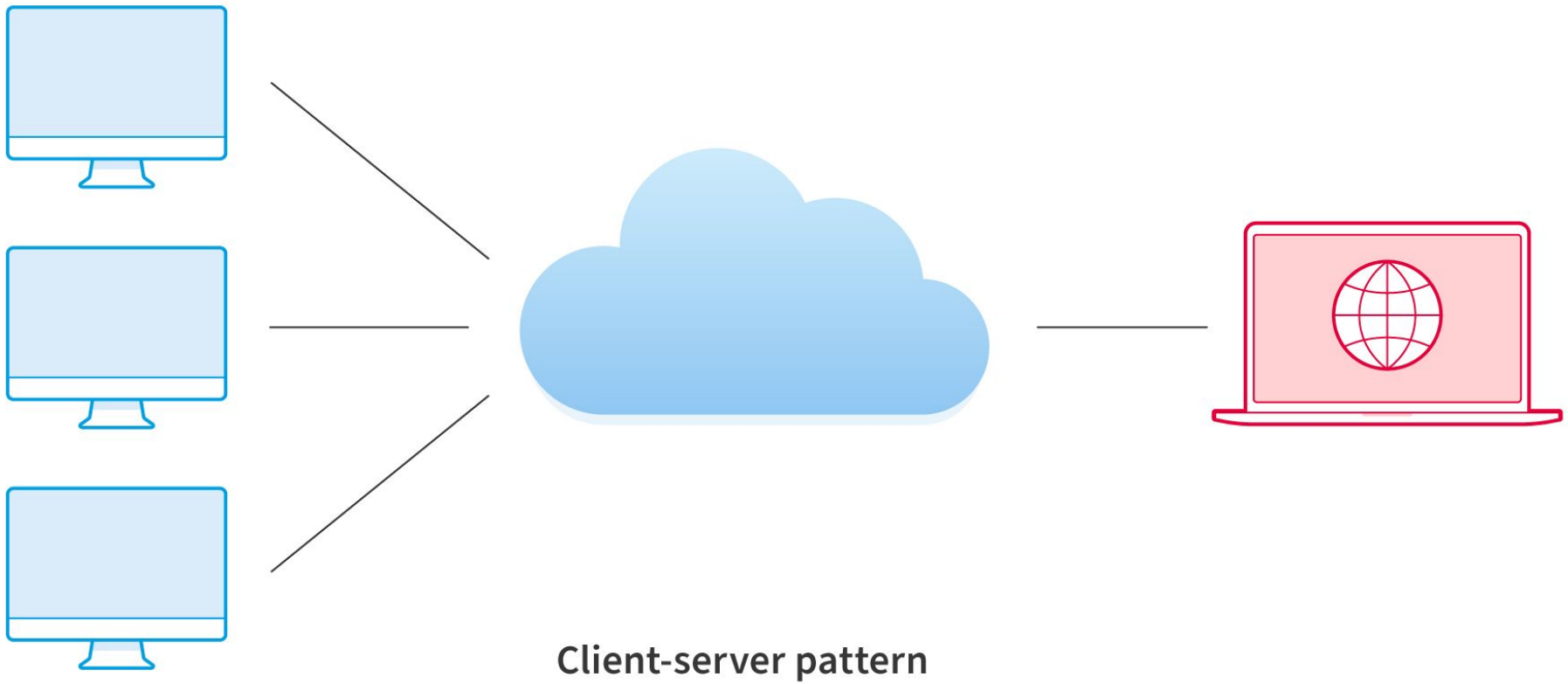
Layered pattern

- We often use ‘N-tier architecture’, or ‘Multi-tiered architecture’ to denote “layered architecture pattern”. It’s one of the most commonly used patterns where the code is arranged in layers. Key characteristics of this pattern are as follows:
- The outermost layer is where the data enters the system. The data passes through the subsequent layers to reach the innermost layer, which is the database layer.
- Simple implementations of this pattern have at least 3 layers, namely, a presentation layer, an application layer, and a data layer.
- Users access the presentation layer using a GUI, whereas the application layer runs the business logic. The data layer has a database for the storage and retrieval of data.

This pattern has the following advantages:

- Maintaining the software is easy since the tiers are segregated.
- Development teams find it easy to manage the software infrastructure, therefore, it’s easy to develop large-scale web and cloud-hosted apps.
- Popular frameworks like Java EE use this pattern.
- There are a few disadvantages too, as follows:
- The code can become too large.
- A considerable part of the code only passes data between layers instead of executing any business logic, which can adversely impact performance.

Pattern #2: Client-server



Client-server pattern

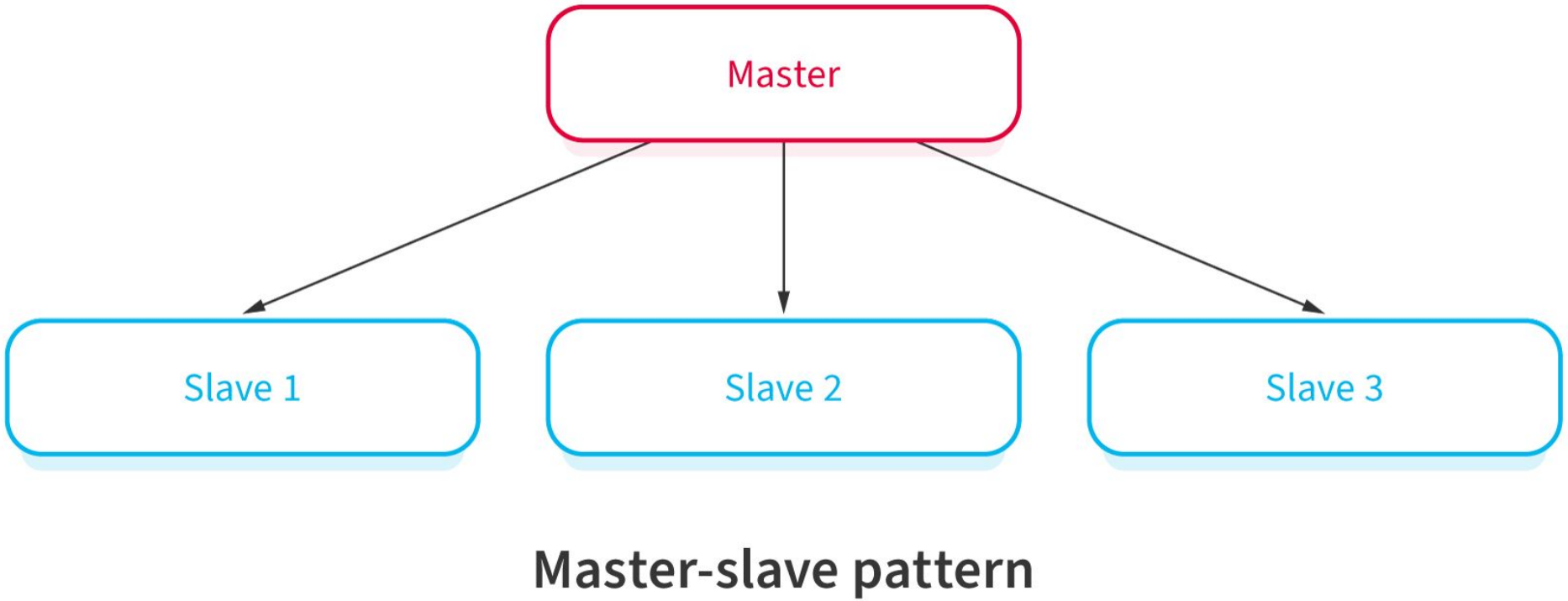
- "Client-server software architecture pattern" is another commonly used one, where there are 2 entities. It has a set of clients and a server. The following are key characteristics of this pattern:
- Client components send requests to the server, which processes them and responds back.
- When a server accepts a request from a client, it opens a connection with the client over a specific protocol.
- Servers can be stateful or stateless. A stateful server can receive multiple requests from clients. It maintains a record of requests from the client, and this record is called a 'session'.
- Email applications are good examples of this pattern. The pattern has several advantages, as follows:

- Clients access data from a server using authorized access, which improves the sharing of data.
- Accessing a service is via a ‘user interface’ (UI), therefore, there’s no need to run terminal sessions or command prompts.
- Client-server applications can be built irrespective of the platform or technology stack.
- This is a distributed model with specific responsibilities for each component, which makes maintenance easier.

Some disadvantages of the client-server architecture are as follows:

- The server can be overloaded when there are too many requests.
- A central server to support multiple clients represents a ‘single point of failure’.

Pattern #3: Master-slave



Master-slave pattern

- “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:

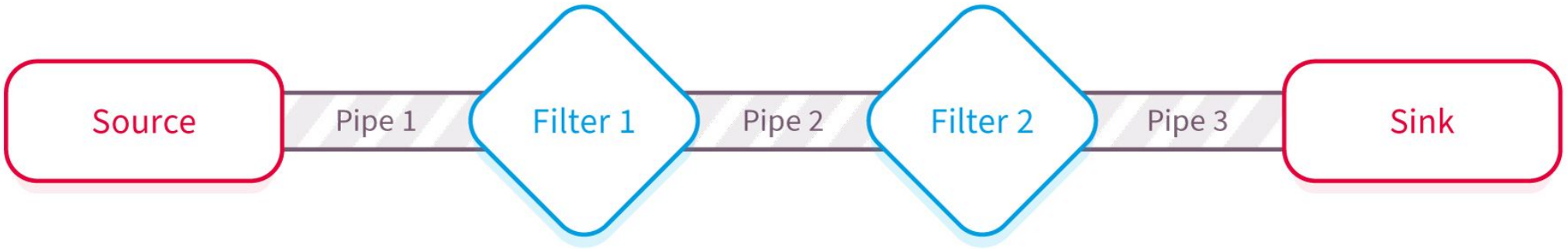
- Applications read from slaves without any impact on the master.
- Taking a slave offline and the later synchronization with the master requires no downtime.

- 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,

- 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.

Pattern #4: Pipe-filter

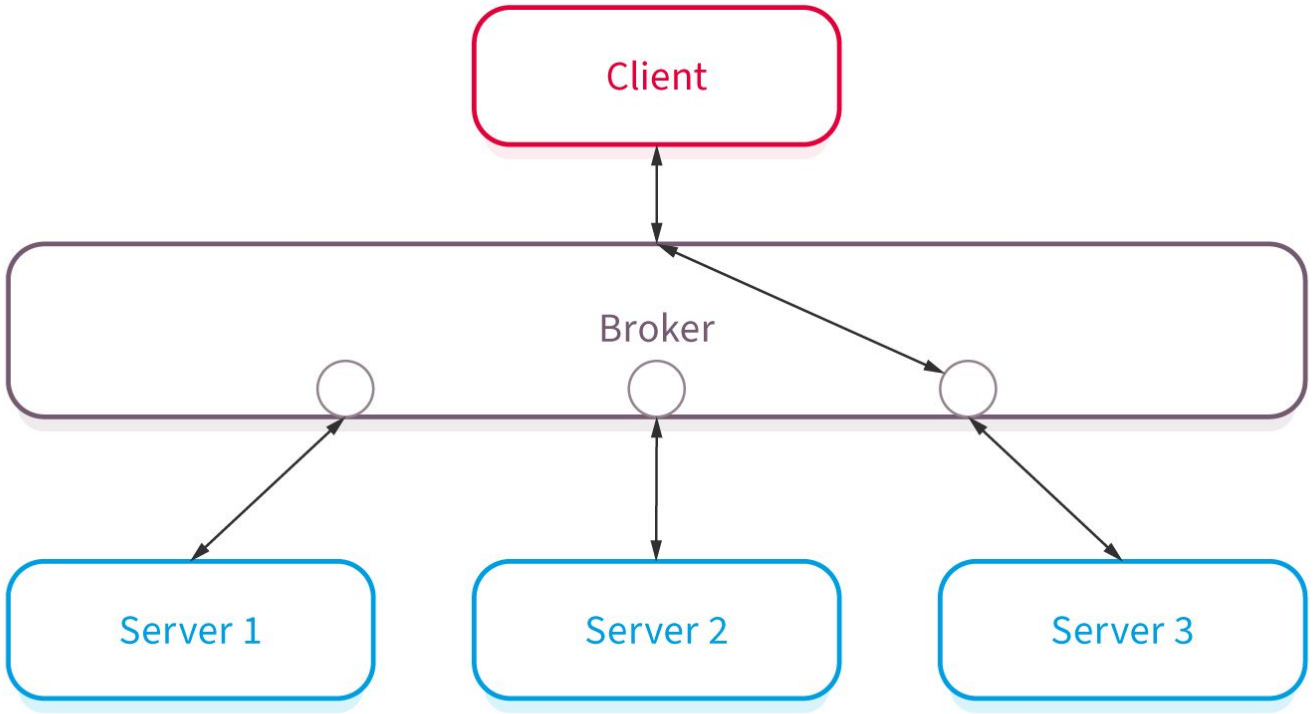


Pipe-filter pattern

- 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 in use. 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.

Pattern #5: Broker



Broker pattern

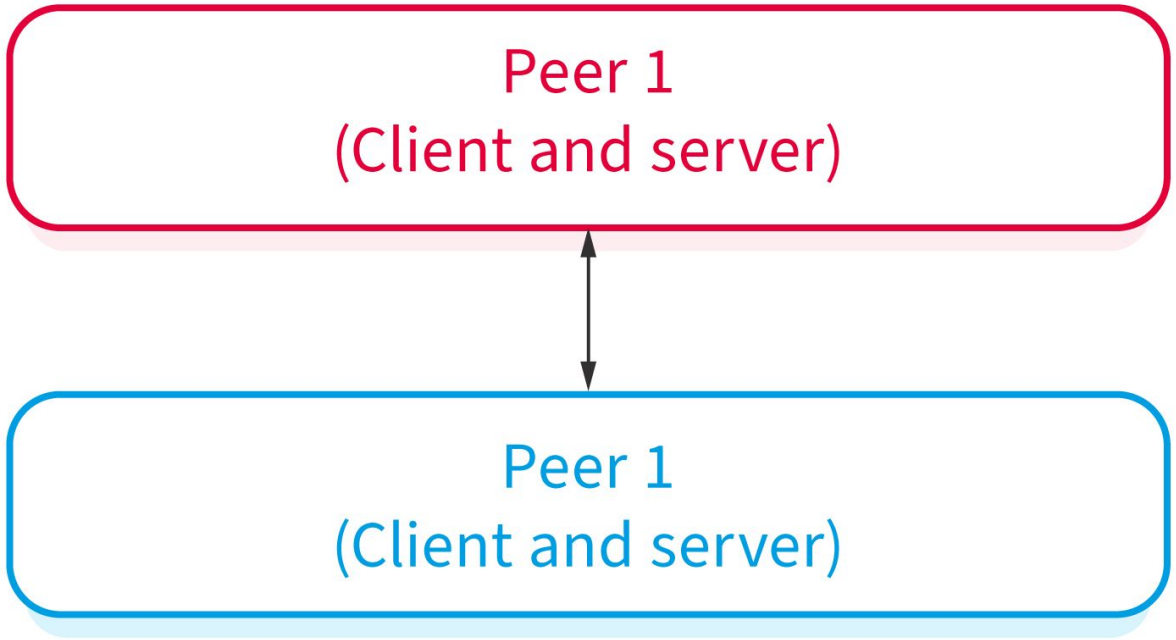
- Consider distributed systems with components that provide different services independent of each other. Independent components could be heterogeneous systems on different servers, however, clients still need their requests serviced. “Broker architecture pattern” is a solution to this.

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.

Pattern #6: Peer-to-peer (P2P)

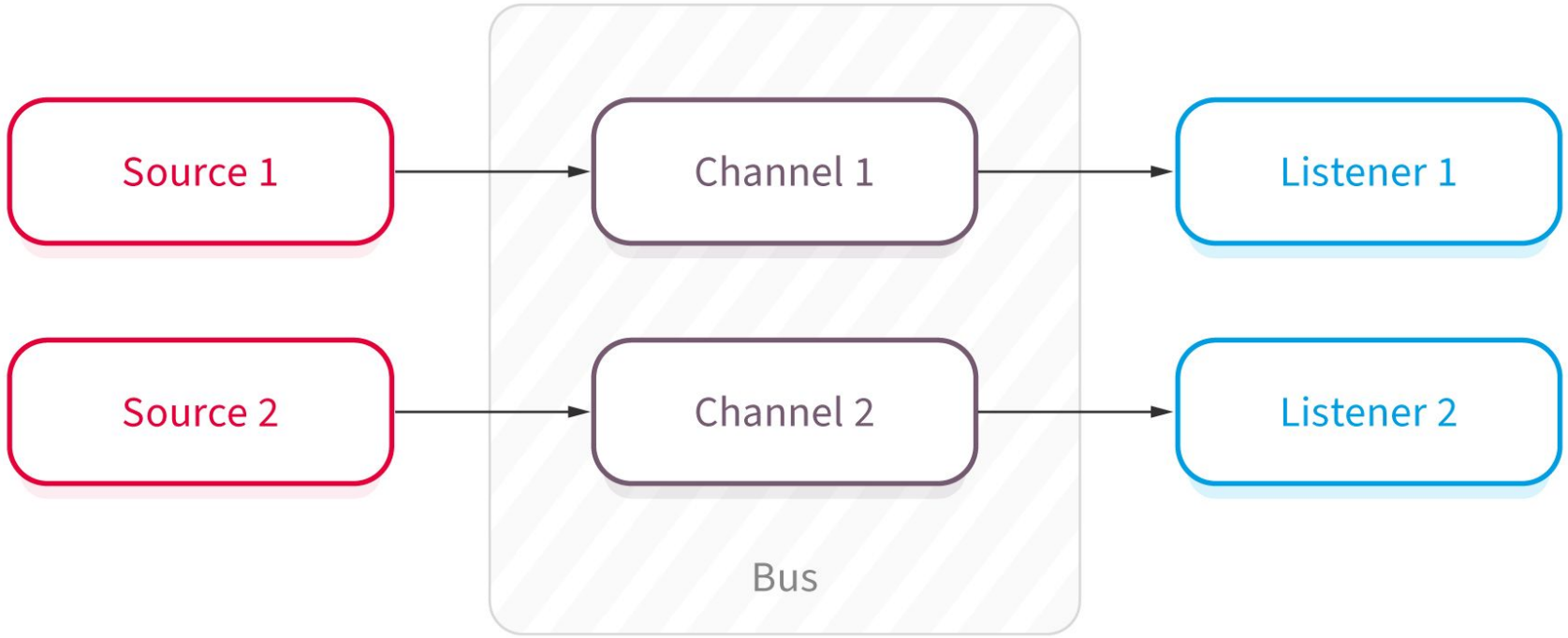


Peer-to-peer pattern

- “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.

Pattern #7: Event-bus pattern



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.

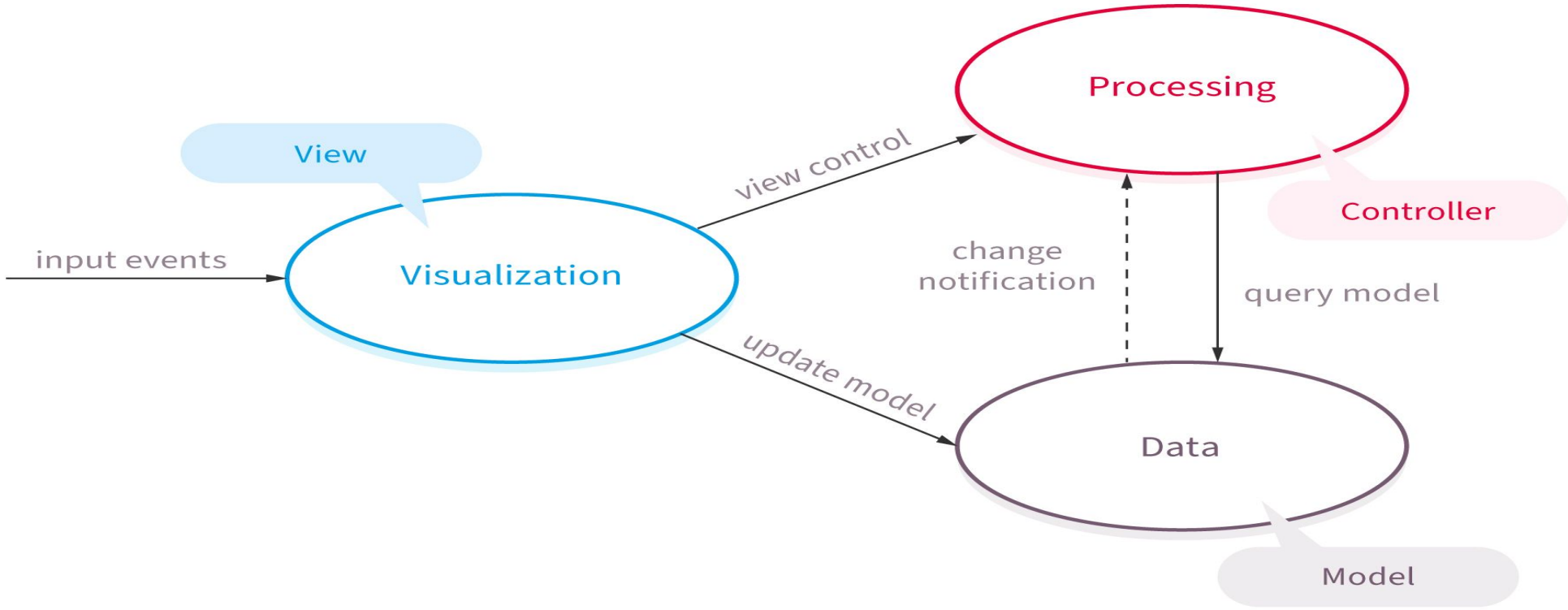
The advantages of this pattern are as follows:

- 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.

Pattern #8: Model-View-Controller (MVC)



Model-view-controller pattern

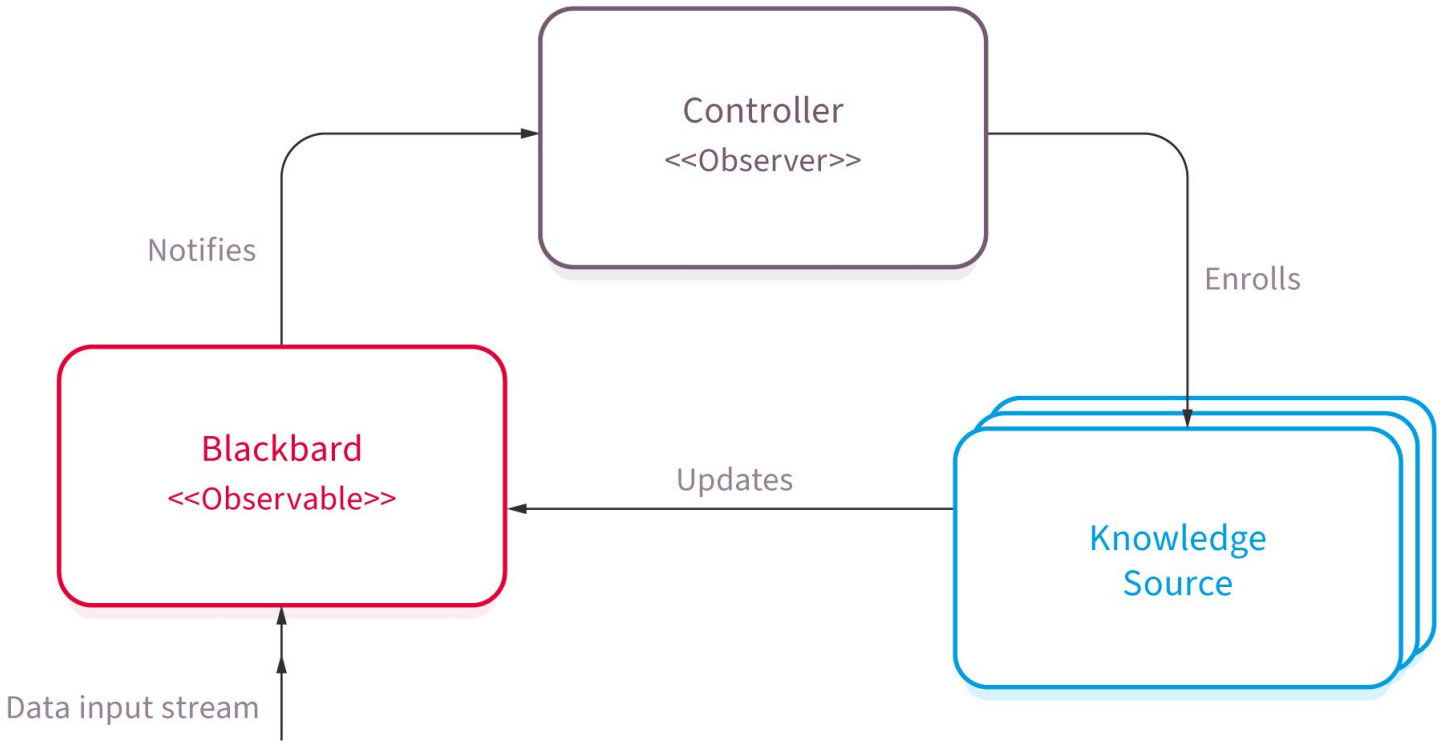
- “Model-View-Controller (MVC) architecture pattern” involves separating an applications’ data model, presentation layer, and control aspects. Following are its’ characteristics:
- There are three building blocks here, namely, model, view, and controller.
- The application data resides in the model.
- 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.

Pattern #9: Blackboard

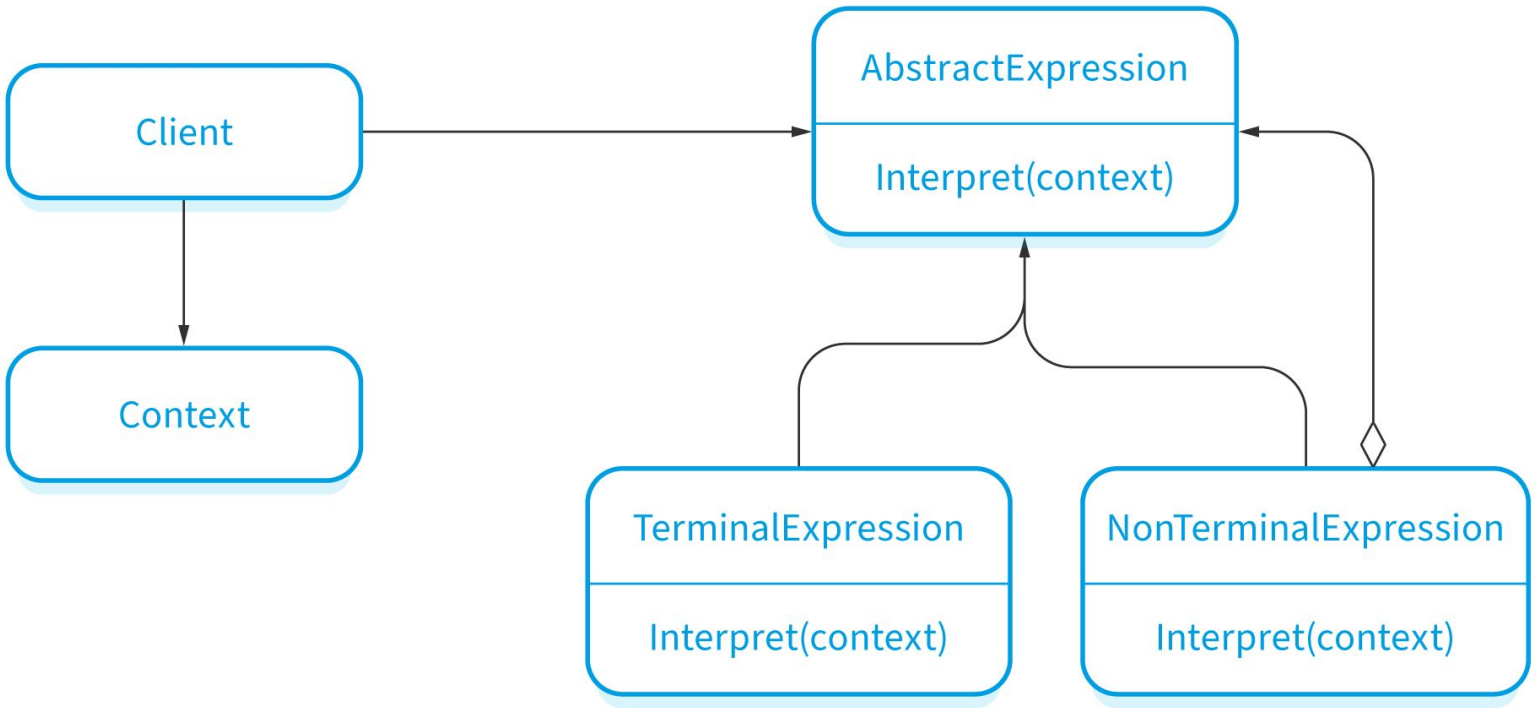


Blackboard Pattern

- 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.

Pattern #10: Interpreter



Interpreter Pattern

- A pattern specific to certain use cases, the “Interpreter pattern” deals with the grammar of programming languages. It offers an interpreter for the language. It works as follows:
- You implement an interface that aids in interpreting given contexts in a programming language.
- The pattern uses a hierarchy of expressions.
- It also uses a tree structure, which contains the expressions.
- A parser, external to the pattern, generates the tree structure for evaluating the expressions.
- The use of this pattern is in creating “Classes” from symbols in programming languages. You create grammar for the language, so that interpretation of sentences becomes possible. Network protocol languages and SQL uses this pattern.

Evaluating Software Architecture Patterns for Your Strategic Application?

- Are you trying to find the best software architectural pattern for your key application?
- You need to make the right choice the first time, however, you need qualified software architects for that. It’s a hot skill. You might need a professional.

Requirement Engineering Process