Software Design Patterns

...

Cameron Wheeless II, Carson Hill & Rajveer Singh

What is a software design pattern?

Software design patterns are conceptual frameworks (or representations of relationships) that provide a structured solution to an everyday problem encountered when developing software.

The patterns are generally guidance from senior, seasoned developers for assistance based on how to tackle and address recurring challenges in system interaction, code design, and architecture.

These are **not** solutions or algorithms, but rather blueprints to improve reusability, flexibility, and maintainability.

Why use them?

The same way someone does not want to wash all their dishes or clothes individually, the average person does not want to do each function repeatedly and individually when writing a program.

Using design patterns helps with not having to recreate the wheel every time you want to use a wheel.

It promotes consistency, improving efficiency in the future of the current or newer projects. Systems become reliable and scalable, with practice becoming repetitive.

Communication increases when everyone involved is on the same path of understanding.

Gang of Four

Often referred to as “Gang of Four”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. These four co-authored the book “*Design patterns: Elements of Reusable Object-Oriented Software*” that was published in 1994. This book would popularize and introduce design patterns to many software engineers.

Their work would focus on defining and cataloging design patterns to create reusable solutions to common problems, mainly object-oriented programming.

When going over their design patterns, the patterns can sometimes be referenced as “GoF design patterns” to reference those four basic fundamentals.

Gang of Four, Erich Gamma & Richard Helm

- Erich Gamma
  - Swiss Computer Scientist
  - Contributed to JUnit testing framework.
  - Major role in developing the Eclipse Integrated Development Environment.
- Richard Helm
  - Australian Computer Scientist
  - Extensive work in distributed object systems.
  - Focused an expertise in object-oriented design and software architecture.

Gang of Four: Ralph Johnson and John Vlissides

- Ralph Johnson
  - American Computer Scientist
  - Contributed to frameworks for Smalltalk programming.
- John Vlissides
  - American Computer Scientist & Software Engineer
  - Known for work on design patterns and software engineering methodologies.

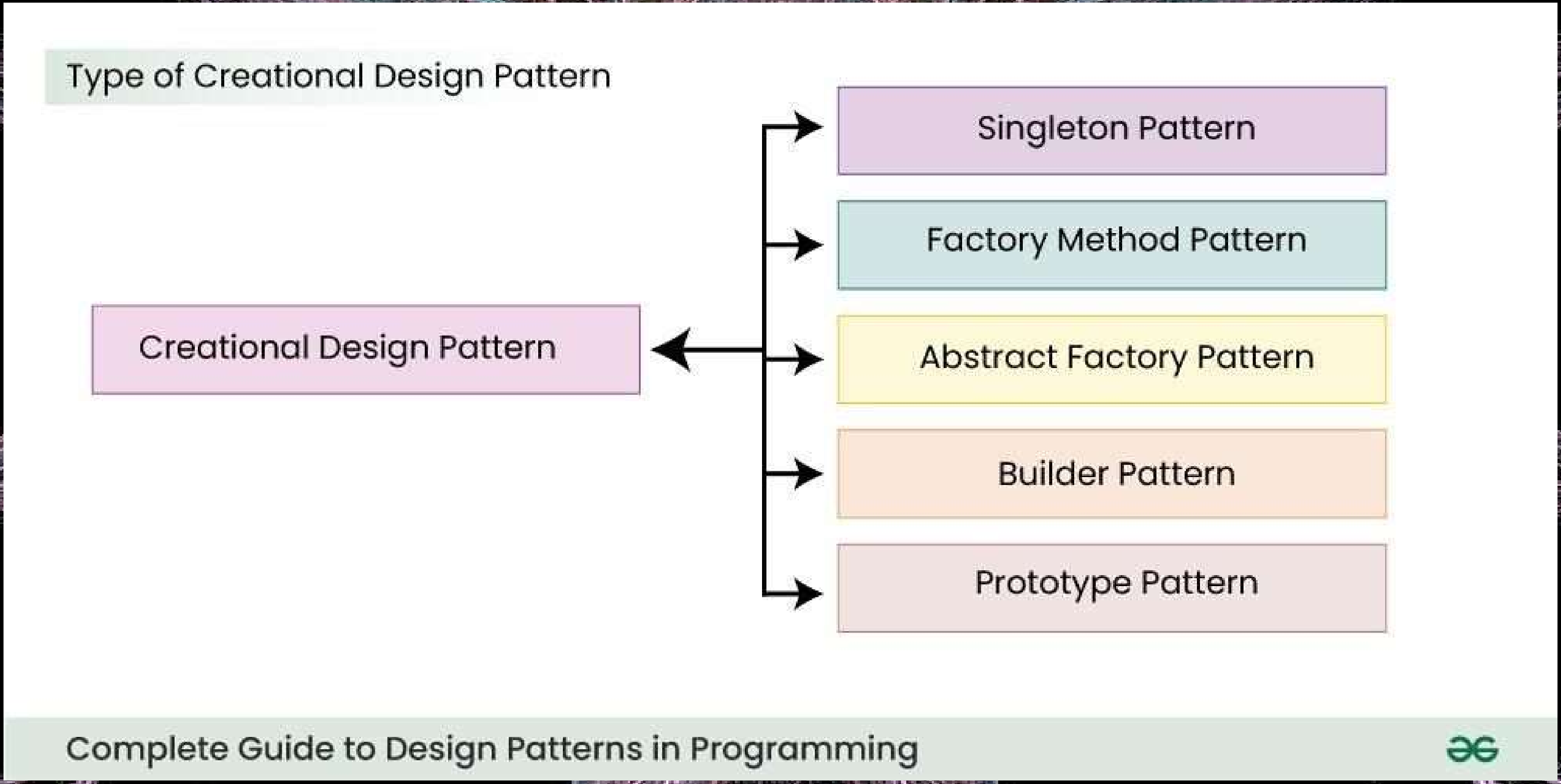
Types of Design Patterns

- Creational
    - Patterns involving the "creation" of an object, in order to increase flexibility and reuse of existing code.
    - Focused on making a system independent of how the objects are composed and represented.
- Structural
    - Use of inheritance to compose interfaces and implementations.
    - Helps independently developed class libraries work together.
- Behavioral
    - There is a need to model and make interactions between objects in a modular and easy to understand way.
    - Shifts away from the idea and focuses more on the way objects are connected.

When would someone use a specific design pattern?

- Creational
  - Step-by-step construction of complex objects.
  - Inner patterns, such as Singleton, prevent duplication by having a global class.
- Structural
  - Adapting to a pre-built interface.
  - Simplifying a complex system to provide a simplified and unified interface to a subsystem.
- Behavioral
  - Dynamic changes to an object's behavior are needed at runtime without changing the code.
  - Encapsulating algorithms or behaviors, allowing them to be independent of objects that use them.

Creational Patterns in a Broad View


Type of Creational Design Pattern
Creational Design Pattern
Singleton Pattern
Factory Method Pattern
Abstract Factory Pattern
Builder Pattern
Prototype Pattern
Complete Guide to Design Patterns in Programming

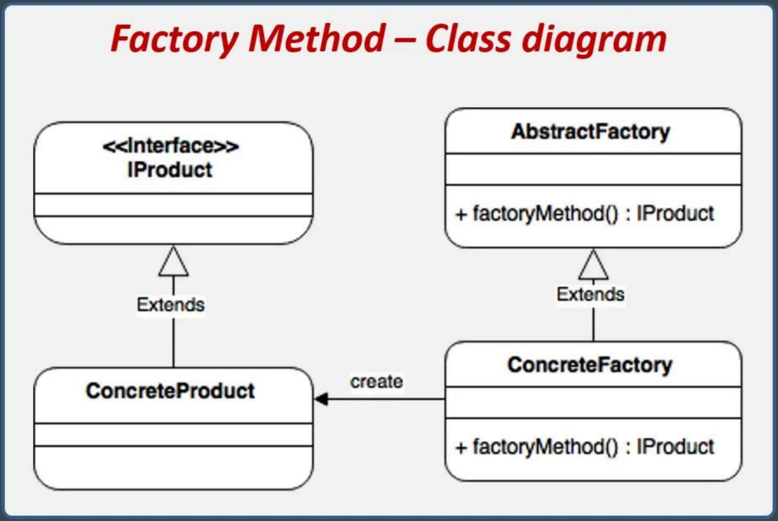
Creational Patterns in more depth: Singleton & Prototype

- Singleton methods/patterns solidify that a class only has one instance and that there is global access to it. These are primarily effective for managing connections to a database or controlling access to a resource. These are commonly considered anti-patterns as they are generally restrictive and not able to be used broadly.
- Prototype pattern lets an object be created by copying an existing object instance. This is useful when creating another object would be complex and costly, allowing there to be an effective “prototype” for modifying rather than creating from scratch.

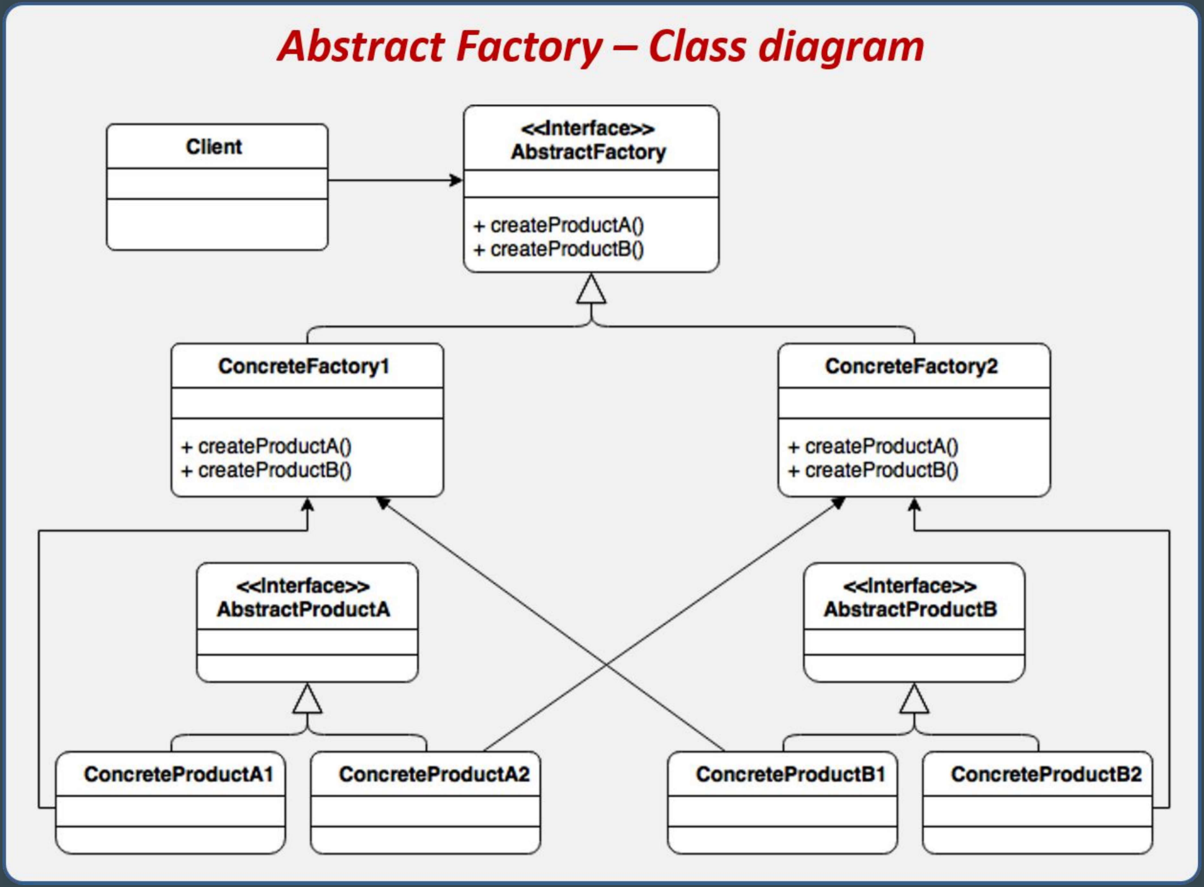
Creational Patterns in more depth: Factory and Abstract Factory

- Factory pattern deals with creating objects without having the specify what class exactly will be created. This hides the construction of a singular object from the client, keeping the output cleaner. The pattern also helps with putting the responsibility of instantiating objects to subclasses promoting flexibility and extensibility.
- Abstract factory pattern builds on factory by encapsulating factories into a similar theme. This helps produce different types of related objects while not worrying about concrete implementations. The different family of objects would be made by a separate factory, but the client would interact with the abstract interface.

Factory vs Abstract Factory



Factory Method – Class diagram
<<Interface>>
IProduct
AbstractFactory
+ factoryMethod() : IProduct
Extends
Extends
ConcreteProduct
create
ConcreteFactory
+ factoryMethod() : IProduct



Creational Patterns in more depth: Builder

The builder pattern separates the construction of a complex object and its representation. This allows the same construction process to now make different representations.

Builder patterns are very useful when it comes to complex configurations or numerous patterns, simplifying it down to what is needed specifically instead of broadly.

This pattern is similar to a constructor, even being called a “glorified constructor” at times, but it is not the same. Compared to a constructor, a builder can be more readable and easier to maintain. Instead of creating with every parameter, a builder can choose certain parameters to ‘build’ with, making it more specific.

Structural Patterns

Adaptor

The Adapter design pattern is a structural pattern that allows the interface of an existing class to be used as another interface.

<u>Components:</u>

- Target Interface
- Adaptee
- Adapter
- Client

Adaptor

Target Interface: The interface expected by the client. It represents the set of operations that the client code can use.

Adaptee: The existing class or system with an incompatible interface that needs to be integrated into the new system.

Adaptor: It acts as a bridge, adapting the interface of the adaptee to match the target interface.

Client: It’s the code that benefits from the integration of the adaptee into the system through the adapter.

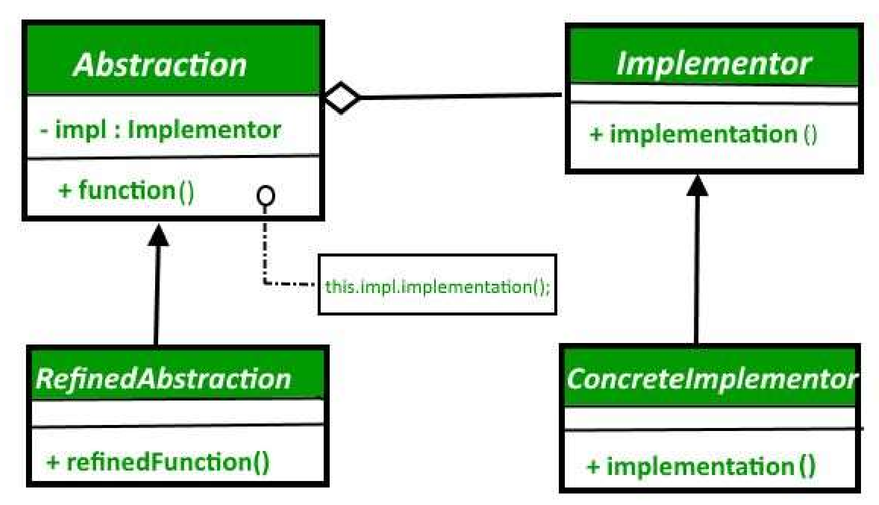
Bridge

The Bridge design pattern simply allows you to separate the abstraction from the implementation.

- The bridge pattern allows the Abstraction and the Implementation to be developed independently and the client code can access only the Abstraction part without being concerned about the Implementation part.

- The abstraction is an interface or abstract class and the implementer is also an interface or abstract class. It increases the loose coupling between class abstraction and its implementation.

Bridge


Abstraction
- impl : Implementor
+ function()
this.impl.implementation();
Implementor
+ implementation ()
RefinedAbstraction
+ refinedFunction()
ConcreteImplementor
+ implementation ()

Composite

Allows you to compose objects into tree structures to represent part-whole hierarchies.

- The main reason is to build a tree structure of objects, where individual objects and composite objects share a common interface.
- Composite is best used GUI libraries, File systems, and Organization Structures

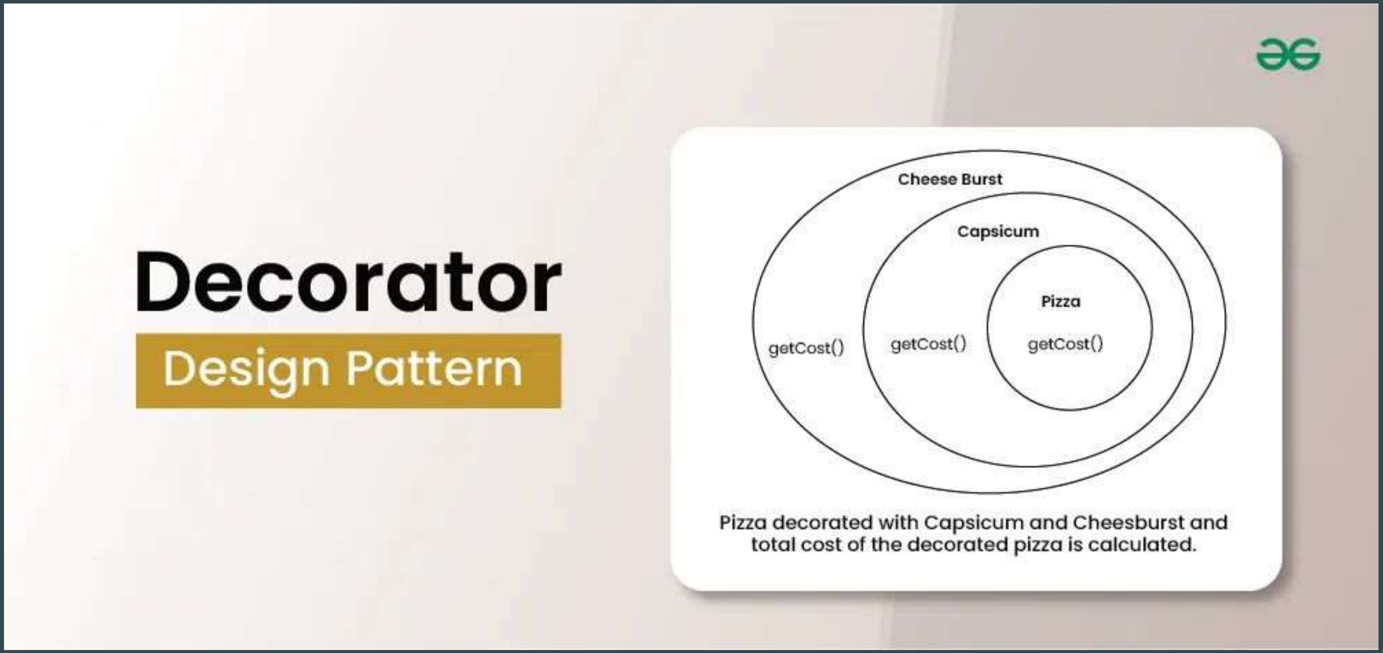
Components:

Leaf: The Leaf is the individual object that does not have any children. It implements the component interface and provides the specific functionality for individual objects.

Composite: The Composite is the container object that can hold Leaf objects as well as the other Composite objects. It implements the Component

Decorator

Allows behavior to be added to individual objects dynamically, without affecting the behavior of other objects from the same class


Decorator
Design Pattern
Cheese Burst
Capsicum
Pizza
getCost()
getCost()
getCost()
Pizza decorated with Capsicum and Cheesburst and total cost of the decorated pizza is calculated.

Decorator

- This pattern promotes flexibility and extensibility in software systems by allowing developers to compose objects with different combinations of functionalities at runtime.

- Powerful tool for building modular and customizable software components.

- Commonly used in scenarios where a variety of optional features or behaviors need to be added to objects in a flexible and reusable manner, such as in text formatting, graphical user interfaces, or customization of products

Decorator

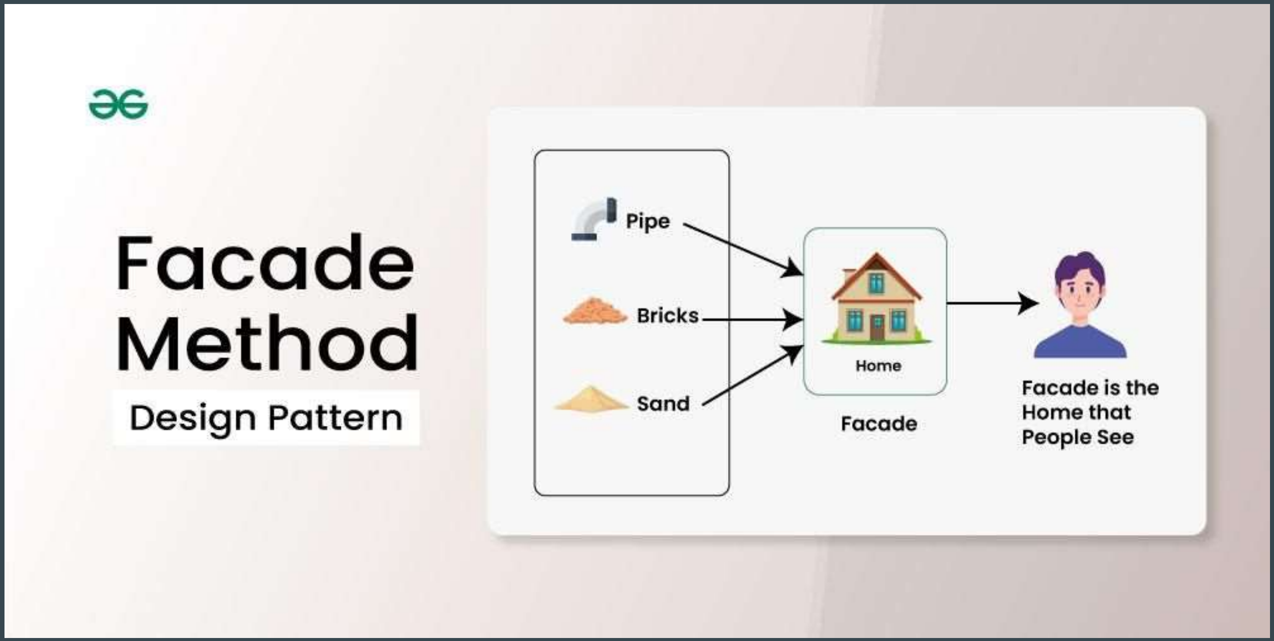
- One very common use of Decorator is the integration of legacy software. It is used to extend the functionality of existing objects without altering their implementation.

- It is also used often in GUI development. It can be used to add additional visual effects, such as borders, shadows, or animations, to GUI components like buttons, panels, or windows.

Facade

Hides the complexity of the underlying system and provides a simple interface that clients can use to interact with the system.

*** It is a part of the gang of four design patterns ***



Facade

- A Facade provides a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.

- Facade defines an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

Flyweight

Focuses on optimizing memory usage by sharing a common state among multiple objects. It aims to reduce the number of objects created and to decrease memory footprint

Key Concepts of Flyweight:

- Shared State
- Object Reuse
- Factory for Flyweights

Flyweight

- Shared State: Objects are divided into intrinsic (shared) state and extrinsic (context-dependent) state. The intrinsic state is shared among multiple objects, while the extrinsic state can vary and is passed to the flyweight objects as needed.
- Object Reuse: Instead of creating new objects each time they are needed, Flyweight objects are reused from a pool of existing objects. This reduces the amount of memory used and improves performance.
- Factory for Flyweights: A factory class manages the creation and reuse of flyweight objects. It ensures that clients get a shared flyweight object when they request one, or it creates a new one only if necessary.

Flyweight

<u>Components:</u>

1. Flyweight Interface/Class:
    - Defines the interface through which flyweight objects can receive and act on extrinsic state.
2. Concrete Flyweight Classes:
    - Stores intrinsic state (state that can be shared) and provides methods to manipulate intrinsic state if needed.
3. Flyweight Factory:
    - Manages a pool of flyweight objects.
    - Provides methods for clients to retrieve or create flyweight objects.
    - Ensures flyweight objects are shared appropriately to maximize reusability.

Flyweight

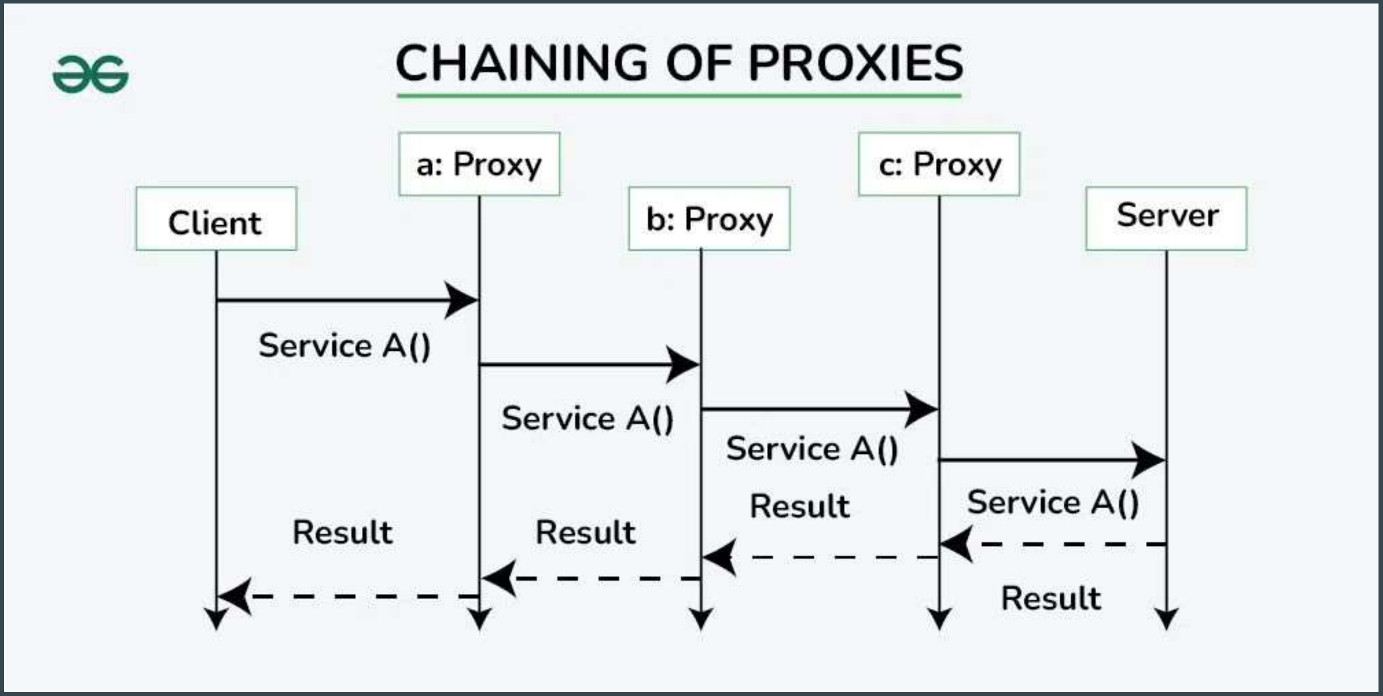
Uses for Flyweight:

- To create a large number of similar objects: For example, if you need to create thousands of graphical objects (like icons, buttons) that share common properties (like image, color) but have different positions or other variable attributes.

- When memory consumption is a concern: If your application is memory-intensive and creating multiple instances of similar objects would consume significant memory, the Flyweight pattern can help in reducing the memory footprint by sharing common state.

Proxy

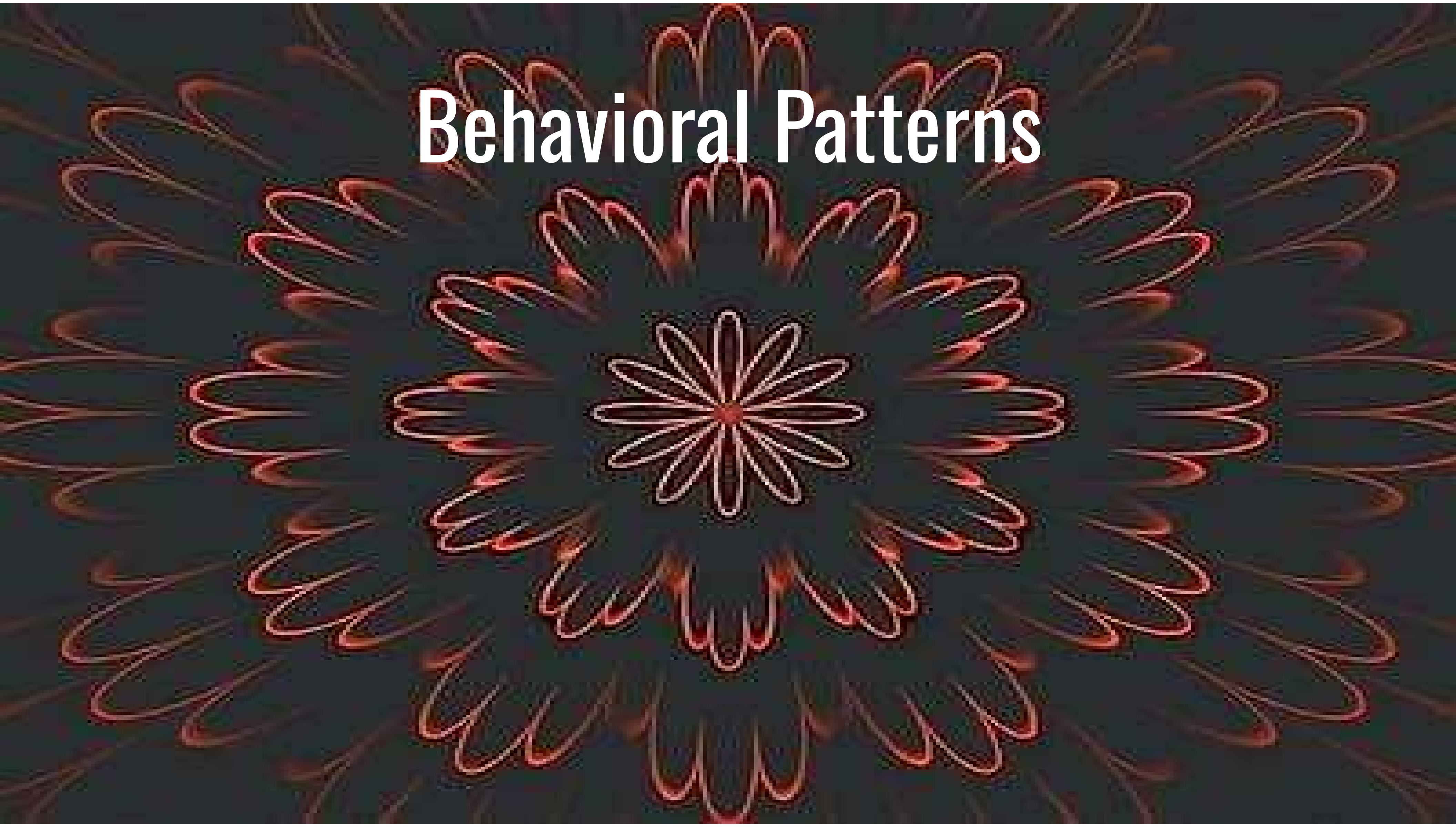
Provides a placeholder for another object to control access to it. The proxy acts as an intermediary, controlling access to the real object.



Proxy

Uses for Proxy:

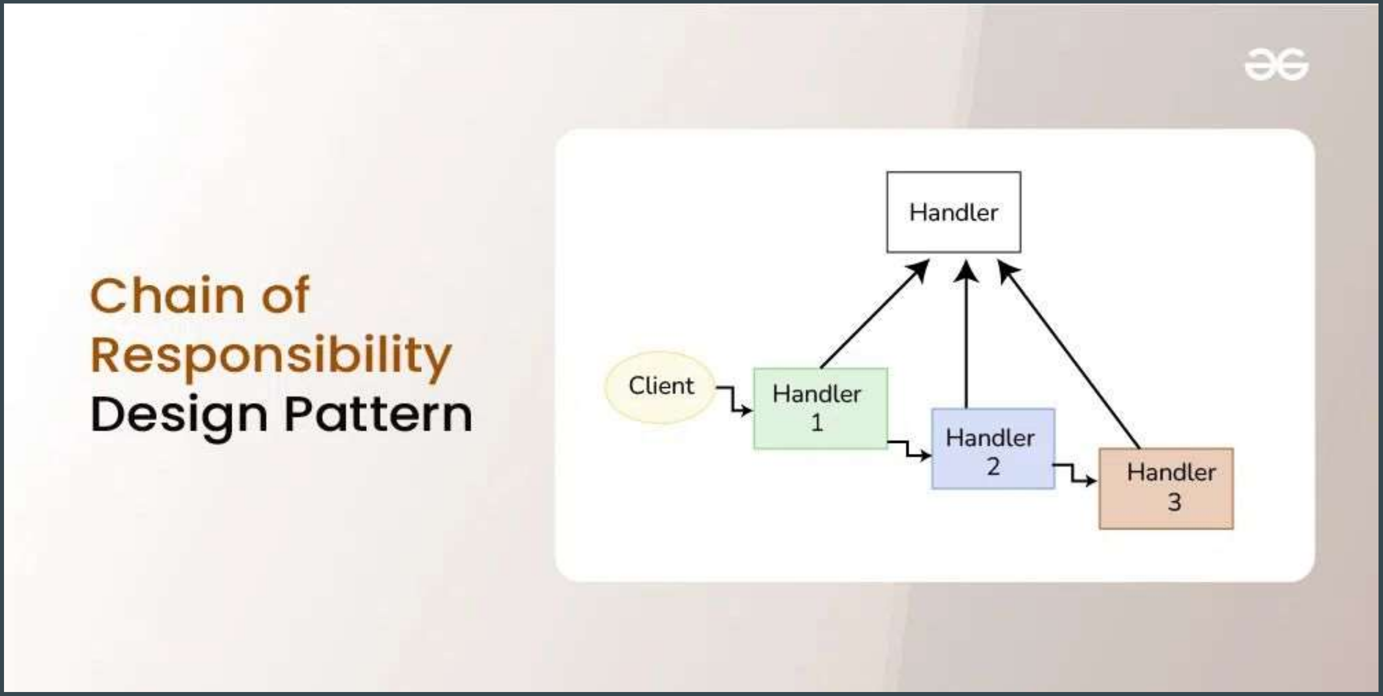
- When you want to postpone the creation of a resource-intensive object until it’s actually needed.
- When you need to control and manage access to an object, ensuring that certain conditions or permissions are met before allowing clients to interact with the real object.
- When dealing with distributed systems and you want to interact with objects located in different addresses or systems.



Behavioral Patterns

Chain of Responsibility

Allows an object to pass a request along a chain of handlers. Each handler in the chain decides either to process the request or to pass it along the chain to the next handler.



Chain of Responsibility

Key Characteristics:

- Dynamic Chain: The chain can be modified dynamically at runtime, allowing for flexibility in adding or removing handlers without affecting the client code.
- Single Responsibility Principle: Each handler in the chain has a single responsibility, either handling the request or passing it to the next handler, which helps in maintaining a clean and modular design.
- Sequential Order: Requests are processed sequentially along the chain, ensuring that each request is handled in a predefined order.
- Fallback Mechanism: The chain can include a mechanism to handle requests that are not handled by any handler in the chain, providing a fallback or default behavior.

Chain of Responsibility

Components:

1. Handler Interface or Abstract Class

This is the base class that defines the interface for handling requests and, in many cases, for chaining to the next handler in the sequence.

2. Concrete Handlers

These are the classes that implement how the requests are going to be handled. They can handle the request or pass it to the next handler in the chain if it is unable to handle that request.

Command

Turns a request into a stand-alone object, allowing parameterization of clients with different requests, queuing of requests, and support for undoable operations(action or a series of actions that can be reversed or undone in a system).

Components:

- Command Interface
- Concrete Command Classes
- Invoker (Remote Control)
- Receiver (Devices)

Command

- Command Interface- like a rulebook that all command classes follow. It declares a common method ensuring that every concrete command knows how to perform its specific action
- Concrete Command Classes- These classes act as executable instructions that the can trigger without worrying about the nitty-gritty details of how each command accomplishes its task.
- Invoker (Remote Control)- Responsible for initiating command execution. It holds a reference to a command but doesn't delve into the specifics of how each command works.
    - Usually a remote control of some sort
- Receiver (Devices)- Device that knows how to perform the actual operation associated with a command

Interpreter

Facilitates the interpretation and evaluation of expressions or language grammars. It provides a mechanism to evaluate sentences in a language by representing their grammar as a set of classes. Each class represents a rule or expression in the grammar, and the pattern allows these classes to be composed hierarchically to interpret complex expressions.

- The tree structure of the Interpreter design pattern is somewhat similar to that defined by the composite design pattern with terminal expressions being leaf objects and non-terminal expressions being composites.

Interpreter

Components:

1. Abstract Expression- This is an abstract class or interface that declares an abstract method. It represents the common interface for all concrete expressions in the language.

2. Terminal Expression- These are the concrete classes that implement the Abstract Expression interface. Terminal expressions represent the terminal symbols or leaves in the grammar. These are the basic building blocks that the interpreter uses to interpret the language.

3. Nonterminal Expression- These are also concrete classes that implement the Abstract Expression interface. Non-terminal expression classes are responsible for handling composite expressions, which consist of multiple sub-expressions. These classes are tasked to provide the interpretation logic for such composite expressions.

Interpreter

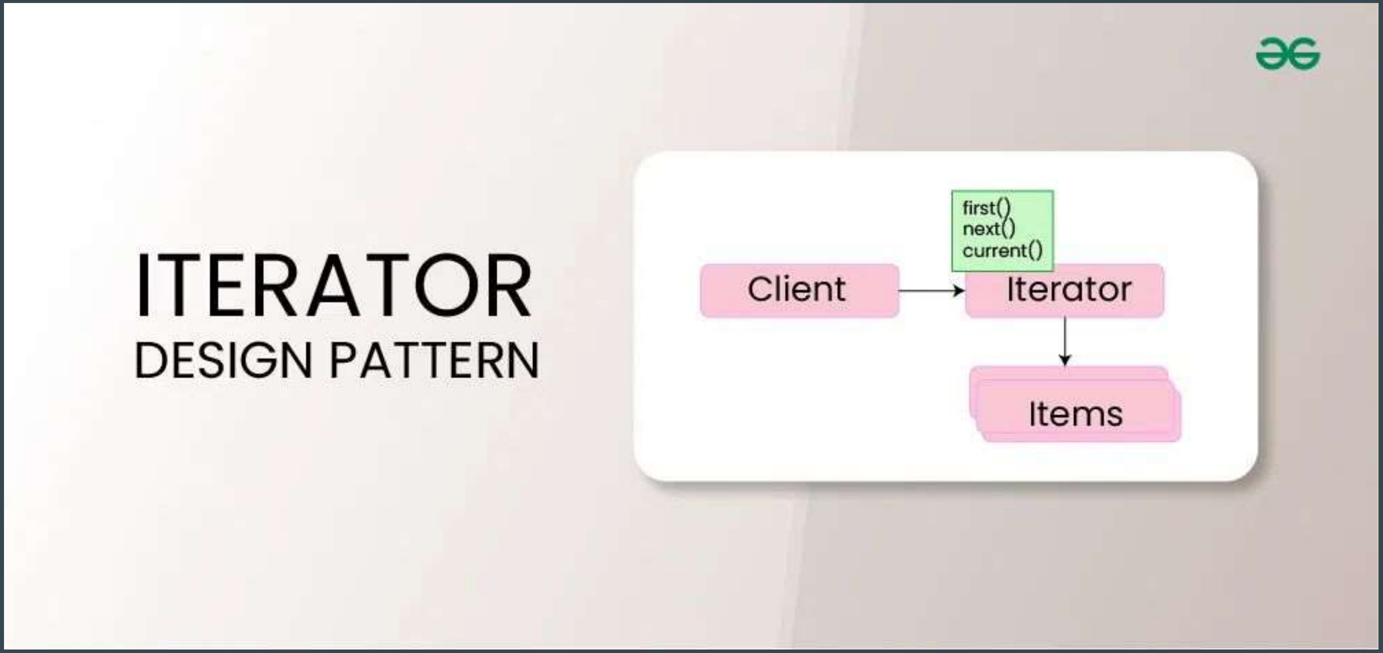
Components Continued:

4. Context- This class contains information that is global to the interpreter and is maintained and modified during the interpretation process. The context may include variables, data structures, or other state information that the interpreter needs to access or modify while interpreting expressions.

5. Interpreter-The interpreter is responsible for coordinating the interpretation process. It manages the context, creates expression objects representing the input expression, and interprets the expression by traversing and evaluating the expression tree.

Iterator

Provides a way to access the elements of an aggregate object (such as a list or collection) sequentially without exposing its underlying representation.



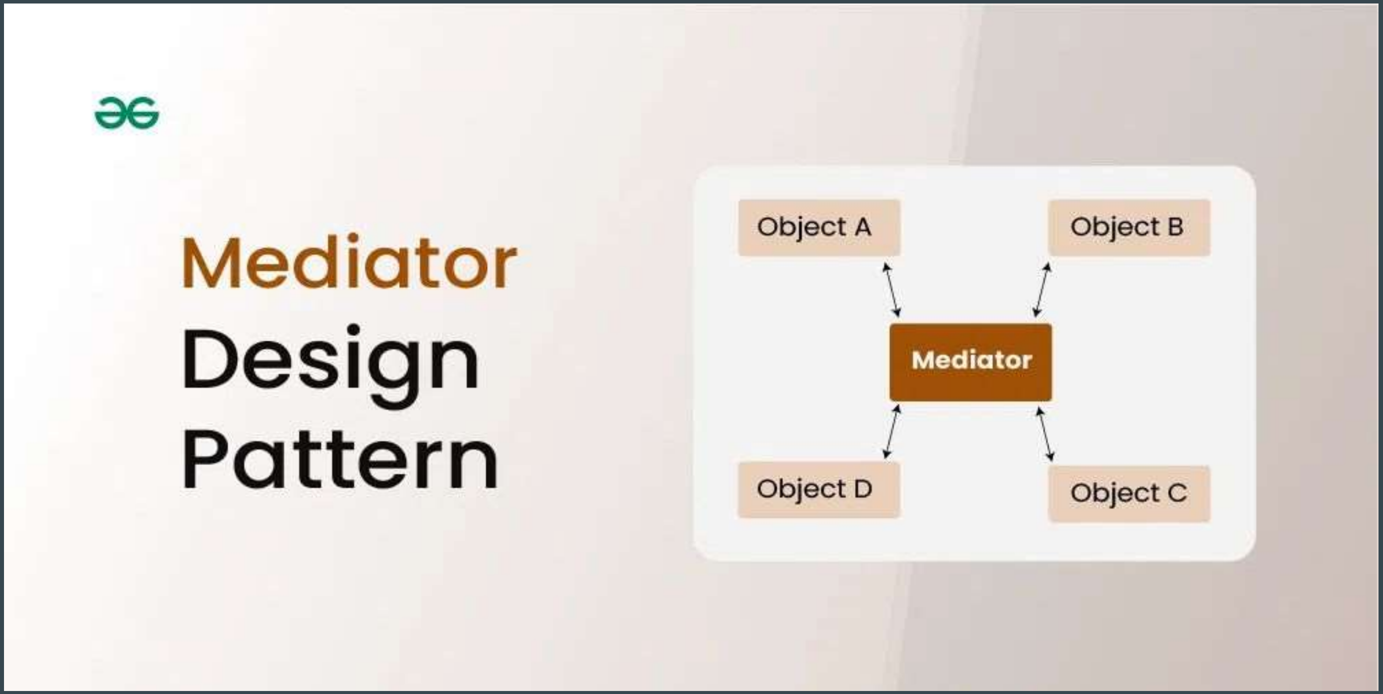
Iterator

Uses of Iterator:

- When you need to support multiple iterators over the same collection. Each iterator maintains its own iteration state, allowing multiple iterations to occur concurrently.
- When you need to access elements of a collection sequentially without exposing its underlying representation. This pattern provides a uniform way to iterate over different types of collections.
- When you want to decouple the iteration logic from the collection. This allows the collection to change its internal structure without affecting the way its elements are accessed.

Mediator

Defines an object, the mediator, to centralize communication between various components or objects in a system.



Mediator

Components:

1. Mediator- The Mediator interface defines the communication contract, specifying methods that concrete mediators should implement to facilitate interactions among colleagues.. It encapsulates the logic for coordinating and managing the interactions between these objects, promoting loose coupling and centralizing control over their communication.

2. Colleague- Colleague classes are the components or objects that interact with each other. They communicate through the Mediator, and each colleague class is only aware of the mediator, not the other colleagues. This isolation ensures that changes in one colleague do not directly affect others.

3. Concrete Mediator- Concrete Mediator is a specific implementation of the Mediator interface. It coordinates the communication between concrete colleague objects, handling their interactions and ensuring a well-organized collaboration while keeping them decoupled.

4. Concrete colleague- Concrete Colleague classes are the specific implementations of the Colleague interface. They rely on the Mediator to communicate with other colleagues, avoiding direct dependencies and promoting a more flexible and maintainable system architecture.

Mediator

Uses of Mediator:

- When your system involves a set of objects that need to communicate with each other in a complex manner, and you want to avoid direct dependencies between them.
- You need a centralized mechanism to coordinate and control the interactions between objects, ensuring a more organized and maintainable system.
- Changes in Behavior: You anticipate changes in the behavior of components, and you want to encapsulate these changes within the mediator, preventing widespread modifications.
- Enhanced Reusability: You want to reuse individual components in different contexts without altering their internal logic or communication patterns.

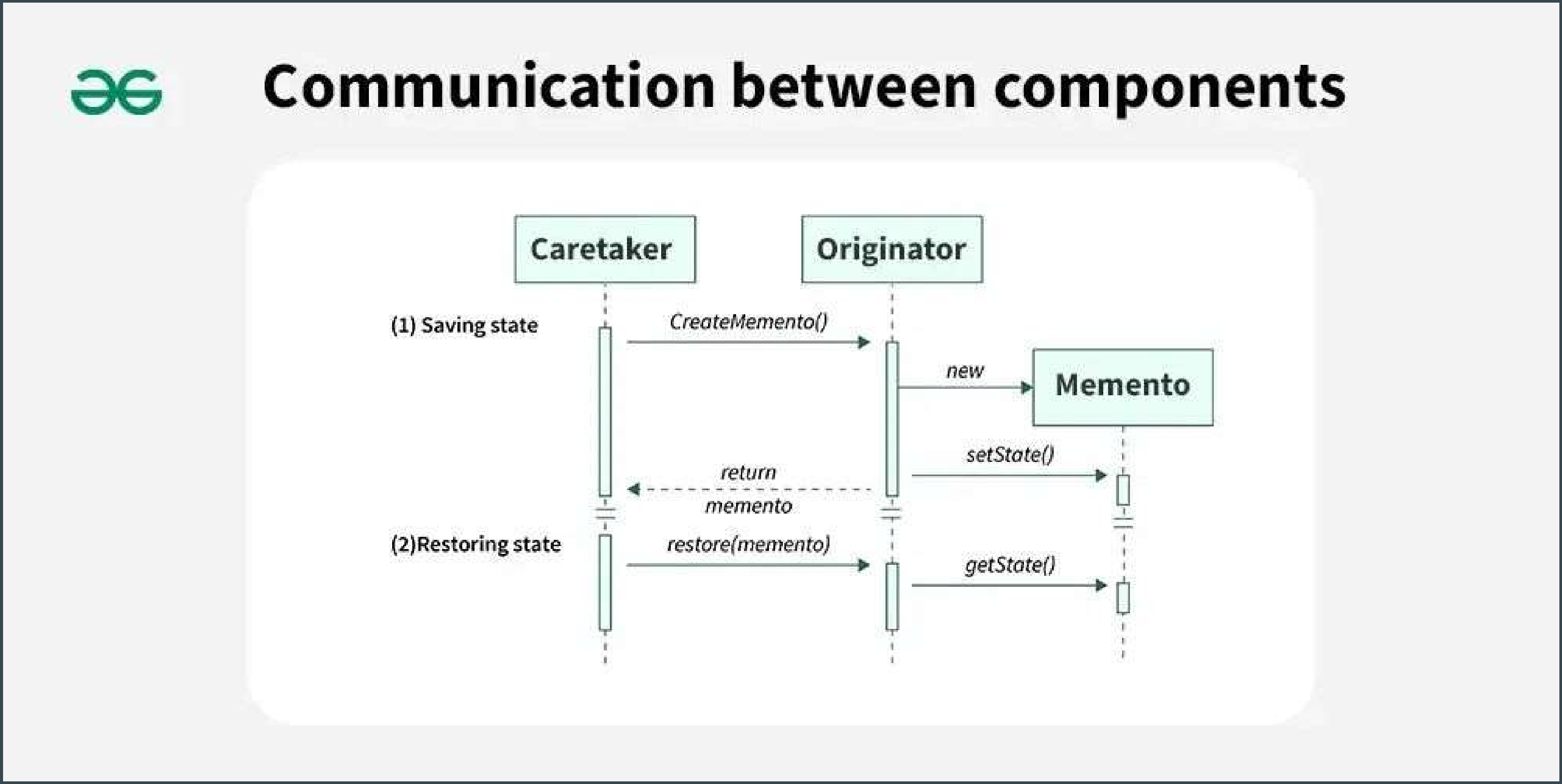
Memento

Used to capture and restore an object's internal state without violating encapsulation. It allows you to save and restore the state of an object to a previous state, providing the ability to undo or roll back changes made to the object.

<u>Components:</u>

- <u>Originator</u>- Responsible for creating and managing the state of an object. It has methods to set and get the object's state, and it can create Memento objects to store its state. The Originator communicates directly with the Memento to create snapshots of its state and to restore its state from a snapshot.
- <u>Memento</u>- Object that stores the state of the Originator at a particular point in time. It only provides a way to retrieve the state, without allowing direct modification. This ensures that the state remains
- <u>Caretaker</u>- Responsible for keeping track of Memento objects. It doesn't know the details of the state stored in the Memento but can request Mementos from the Originator to save or restore the object's state.

Memento


Communication between components
Caretaker
Originator
Memento
(1) Saving state
CreateMemento()
new
setState()
return memento
(2)Restoring state
restore(memento)
getState()

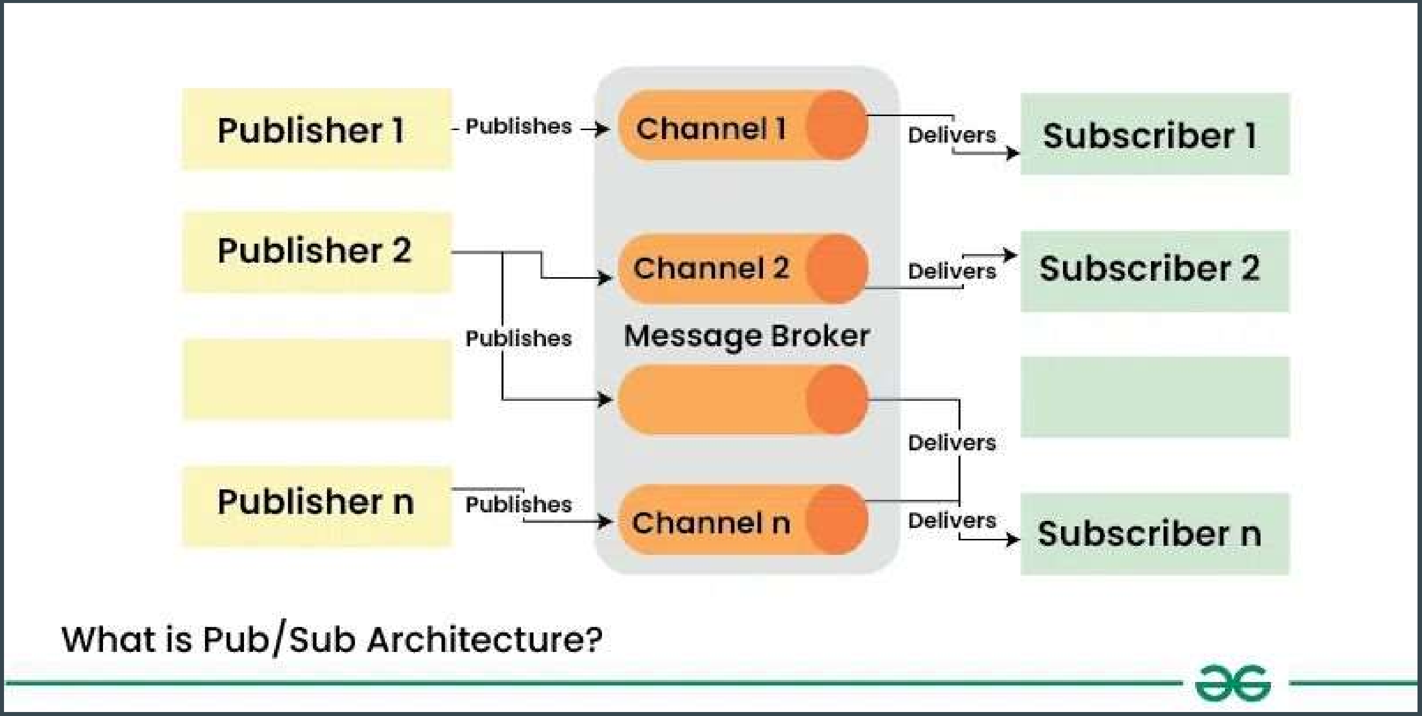
Memento

Uses for Memento:

- Undo functionality: When you need to implement an undo feature in your application that allows users to revert changes made to an object's state.
- Snapshotting: When you need to save the state of an object at various points in time to support features like versioning or checkpoints.
- Transaction rollback: When you need to rollback changes to an object's state in case of errors or exceptions, such as in database transactions.
- Caching: When you want to cache the state of an object to improve performance or reduce redundant computations.

Publish/Subscribe(observer)

A messaging pattern used in software architecture to facilitate asynchronous communication between different components or systems. In this model, publishers produce messages that are then consumed by subscribers.



Publish/Subscribe(observer)

Components:

1. Publisher- The Publisher is responsible for creating and sending messages to the Pub/Sub system. Publishers categorize messages into topics or channels based on their content. They do not need to know the identity of the subscribers.

2. Subscriber- The Subscriber is a recipient of messages in the Pub/Sub system. Subscribers express interest in receiving messages from specific topics. They do not need to know the identity of the publishers. Subscribers receive messages from topics to which they are subscribed.

3. Topic- A Topic is a named channel or category to which messages are published. Publishers send messages to specific topics, and subscribers can subscribe to one or more topics to receive messages of interest. Topics help categorize messages and enable targeted message delivery to interested subscribers.

4. Message Broker- The Message Broker is an intermediary component that manages the routing of messages between publishers and subscribers. It receives messages from publishers and forwards them to subscribers based on their subscriptions. The Message Broker ensures that messages are delivered to the correct subscribers and can provide additional features such as message persistence, scalability, and reliability.

Publish/Subscribe(observer)

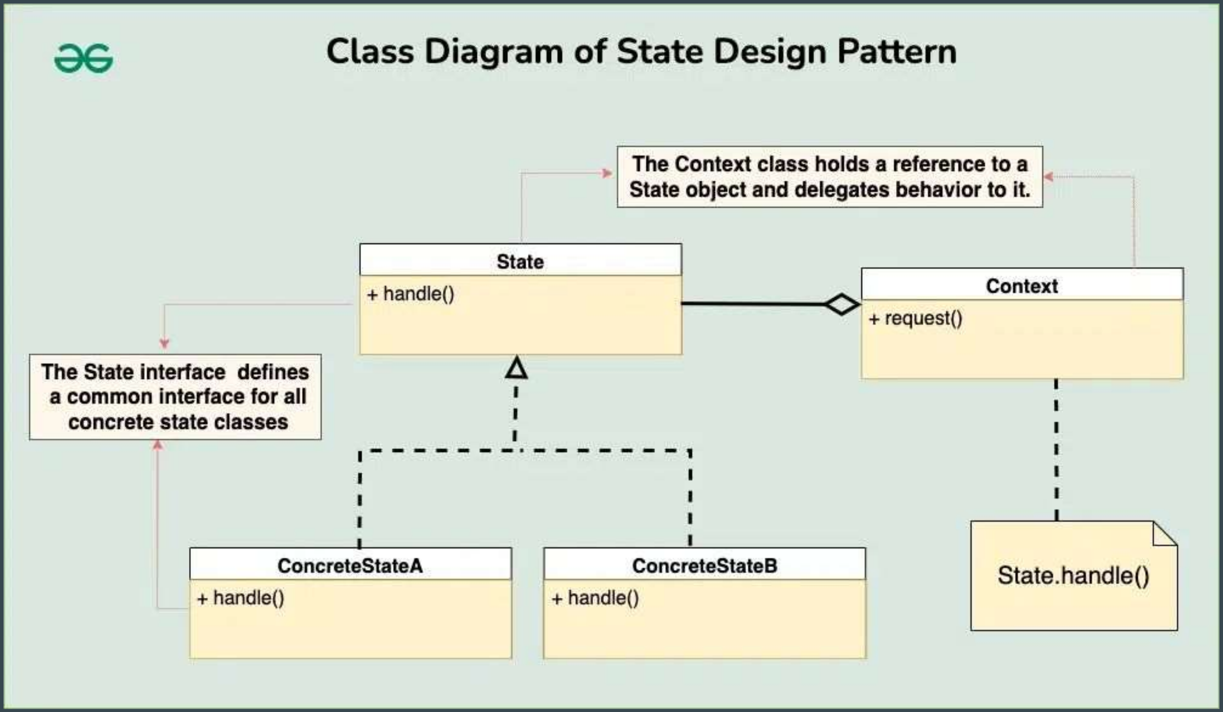
Components Continued:

5. Message- A Message is the unit of data exchanged between publishers and subscribers in the Pub/Sub system. Messages can contain any type of data, such as text or binary data. Publishers create messages and send them to the Pub/Sub system, and subscribers receive and process these messages.

6. Subscription- A Subscription represents a connection between a subscriber and a topic. Subscriptions define which messages a subscriber will receive based on the topics to which it is subscribed. Subscriptions can have different configurations, such as message delivery guarantees (e.g., at-most-once, at-least-once) and acknowledgment mechanisms.

State

Allows an object to alter its behavior when its internal state changes. It achieves this by encapsulating the object's behavior within different state objects, and the object itself dynamically switches between these state objects depending on its current state.


Class Diagram of State Design Pattern
The Context class holds a reference to a State object and delegates behavior to it.
State
+ handle()
Context
+ request()
The State interface defines a common interface for all concrete state classes
ConcreteStateA
+ handle()
ConcreteStateB
+ handle()
State.handle()

State

Components:

1. Context- The Context is the class that contains the object whose behavior changes based on its internal state. It maintains a reference to the current state object that represents the current state of the Context. The Context provides an interface for clients to interact with and typically delegates state-specific behavior to the current state object.

2. State Interface or Base Class- The State interface or base class defines a common interface for all concrete state classes. This interface typically declares methods that represent the state-specific behavior that the Context can exhibit. It allows the Context to interact with state objects without knowing their concrete types.

3. Concrete States- Concrete state classes implement the State interface or extend the base class. Each concrete state class encapsulates the behavior associated with a specific state of the Context. These classes define how the Context behaves when it is in their respective states.

State

Uses for State:

The State design pattern is beneficial when you encounter situations with objects whose behavior changes dynamically based on their internal state. Here are some key indicators:

- **Multiple states with distinct behaviors:** If your object exists in several states (e.g., On/Off, Open/Closed, Started/Stopped), and each state dictates unique behaviors, the State pattern can encapsulate this logic effectively.
- **Complex conditional logic:** When conditional statements (if-else or switch-case) become extensive and complex within your object, the State pattern helps organize and separate state-specific behavior into individual classes, enhancing readability and maintainability.
- **Frequent state changes:** If your object transitions between states frequently, the State pattern provides a clear mechanism for managing these transitions and their associated actions.
- **Adding new states easily:** If you anticipate adding new states in the future, the State pattern facilitates this by allowing you to create new state classes without affecting existing ones.

Strategy

Defines a family of algorithms, encapsulates each one, and makes them interchangeable, allowing clients to switch algorithms dynamically without altering the code structure.

- Allows the behavior of an object to be selected at runtime
- One of the Gang of Four Software patterns

Strategy

Components:

1. Context- The Context is a class or object that holds a reference to a strategy object and delegates the task to it.

- It acts as the interface between the client and the strategy, providing a unified way to execute the task without knowing the details of how it's done.

2. Strategy Interface- The Strategy Interface is an interface or abstract class that defines a set of methods that all concrete strategies must implement.

- It serves as a contract, ensuring that all strategies adhere to the same set of rules and can be used interchangeably by the Context.

3. Concrete Strategies- Concrete Strategies are the various implementations of the Strategy Interface. Each concrete strategy provides a specific algorithm or behavior for performing the task defined by the Strategy Interface.

- Concrete strategies encapsulate the details of their respective algorithms and provide a method for executing the task.
- They are interchangeable and can be selected and configured by the client based on the requirements of the task.

Strategy

Uses for Strategy:

Multiple Algorithms : When you have multiple algorithms that can be used interchangeably based on different contexts, such as sorting algorithms (bubble sort, merge sort, quick sort), searching algorithms, compression algorithms, etc.

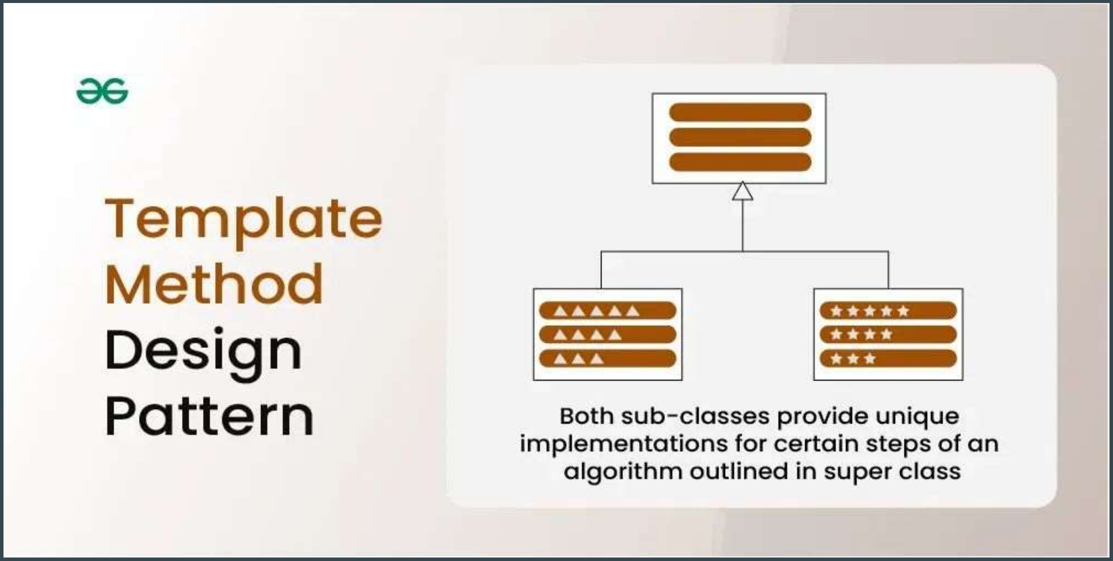
Runtime Selection : When you need to dynamically select and switch between different algorithms at runtime based on user preferences, configuration settings, or system states.

Reducing Conditional Statements : When you have a class with multiple conditional statements that choose between different behaviors, using the Strategy pattern helps in eliminating the need for conditional statements and making the code more modular and maintainable.

Template Method

Defines the skeleton of an algorithm in a superclass but allows subclasses to override specific steps of the algorithm without changing its structure.

- It promotes code reuse by encapsulating the common algorithmic structure in the superclass while allowing subclasses to provide concrete implementations for certain steps, thus enabling customization and flexibility.


Template Method Design Pattern
Both sub-classes provide unique implementations for certain steps of an algorithm outlined in super class

Template Method

- The overall structure and sequence of the algorithm are preserved by the parent class.
- This is one of the easiest to understand and implement. This design pattern is used popularly in framework development and helps to avoid code duplication.

Components:

- Abstract Class
- Template Method
- Abstract Methods
- Concrete Subclasses

Template Method

Abstract Class: Superclass that defines the template method. It provides a skeleton for the algorithm, where certain steps are defined but others are left abstract or defined as hooks that subclasses can override. It may also include concrete methods that are common to all subclasses and are used within the template method.

Template Method: This is the method within the abstract class that defines the overall algorithm structure by calling various steps in a specific order. It's often declared as final to prevent subclasses from changing the algorithm's structure.

Abstract Methods: Methods declared within the abstract class but not implemented. They serve as placeholders for steps in the algorithm that should be implemented by subclasses.

Concrete Subclasses: Subclasses that extend the abstract class and provide concrete implementations for the abstract methods defined in the superclass. Each subclass can override certain steps of the algorithm to customize the behavior without changing the overall structure.

Template Method

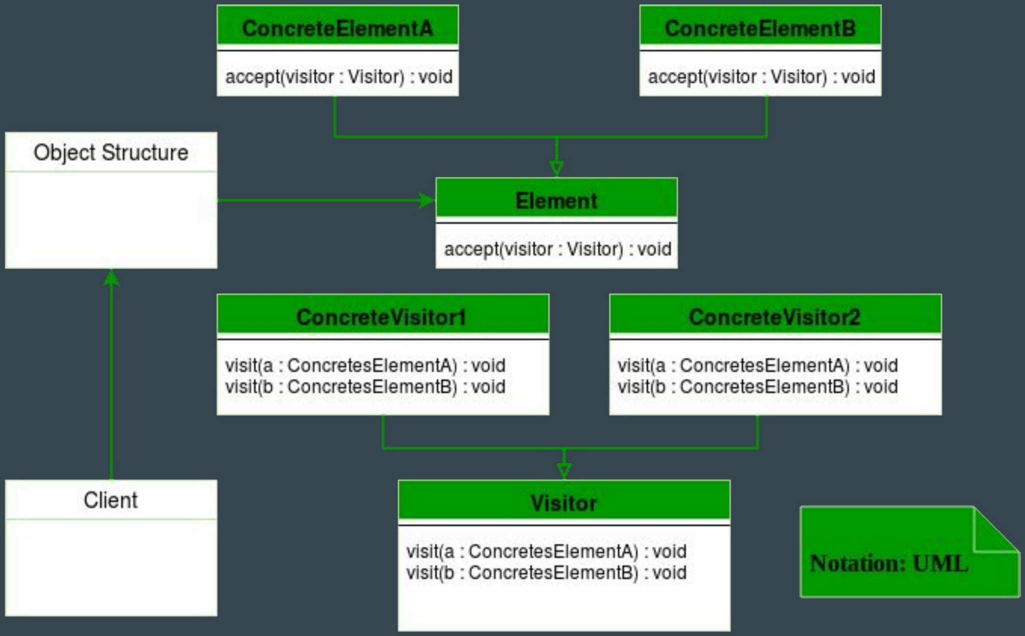
<u>Uses for Template:</u>

- **Common Algorithm with Variations** : When you have an algorithm with a common structure but with some varying steps or implementations, the Template Method pattern helps to encapsulate the common steps in a superclass while allowing subclasses to override specific steps.

- **Code Reusability** : If you have similar tasks or processes that need to be performed in different contexts, the Template Method pattern promotes code reuse by defining the common steps in one place.

Visitor

It is used when we have to perform an operation on a group of similar kind of Objects. With the help of visitor pattern, we can move the operational logic from the objects to another class.


ConcreteElementA
accept(visitor : Visitor) : void
ConcreteElementB
accept(visitor : Visitor) : void
Object Structure
Element
accept(visitor : Visitor) : void
ConcreteVisitor1
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
ConcreteVisitor2
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
Client
Visitor
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
Notation: UML

Visitor

Components:

- **Visitor :** This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes.
- **Concrete Visitor :** For each type of visitor all the visit methods, declared must be implemented. Each Visitor will be responsible for different operations.
- **Visitable :** This is an interface which declares the accept operation. This is the entry point which enables an object to be “visited” by the visitor object.
- **Concrete Visitable :** These classes implement the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.

Other Design Patterns

Concurrency and Lazy Initialization

- Concurrency
  - The pattern to deal with multi-threaded programming
  - Ensures multiple threads or processes can operate concurrently without conflicts.
  - Helps manage tasks like synchronization and task scheduling.

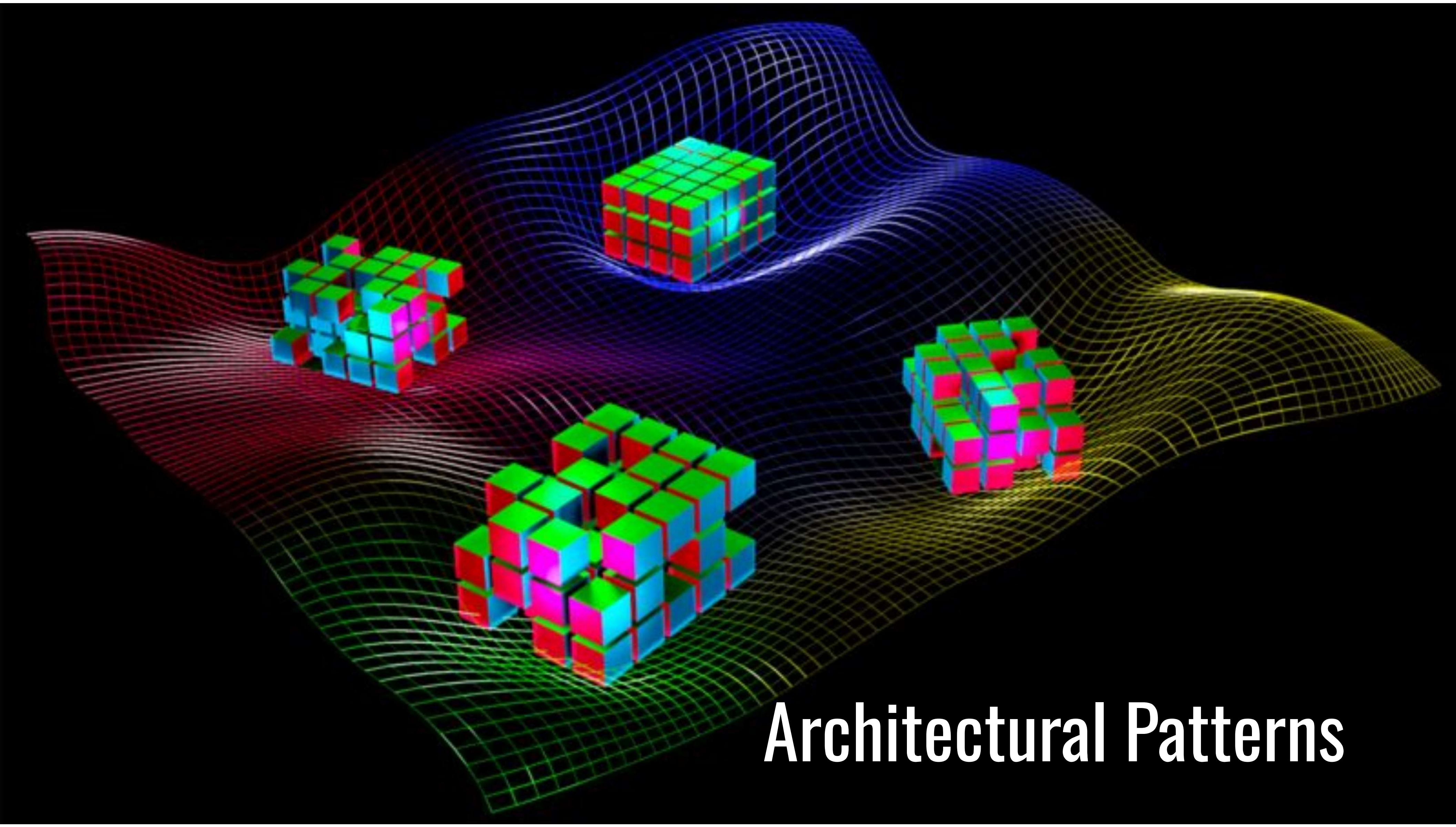
- Lazy Initialization
  - Pattern that delays the creation/initialization of an object/resource until it is needed.
  - Similar to a student studying computer science, it tries to improve performance by avoiding any unnecessary computations or memory usage.
  - This is generally done when the initialization of a resource is rather expensive or not always required rather than a resource that will be constantly required.

Null Object & Module

- Null Object
  - The default “couch potato” or “do nothing” behavior for objects.
  - This uses the idea of using a null object to provide a neutral implementation without checking for a ‘null’ value.
  - Particularly helpful in avoiding null reference errors.
- Module
  - Pattern to encapsulate related functions, objects, or variables into a single ‘module’.
  - Helps organize code, separate concerns, and avoid global variables.
  - An example would be a module that authenticates login functions.

RAII (Resource Acquisition is Initialization)

- A resource management pattern commonly used in C++ (and other similar languages, but originated in C++)
  - Resources are acquired during object creation/destruction.
  - Resource lifecycle is tied to its lifetime, limiting resource leaks.
- Poorly named, programmers and engineers sometimes try to refer to this pattern as “Scope-Bound Resource Management.”
  - It is bound to the scope of a variable rather
- An example is a constructor and destructor for a class.



Architectural Patterns

What is an Architectural Pattern in Software?

- An **architectural pattern** in software is a reusable, high-level solution that addresses common structural challenges in software design. It defines how software components should be organized and interact, offering a blueprint for designing complex systems.

- Architectural patterns provide guidelines for structuring components like user interfaces, business logic, and databases, ensuring they are scalable, maintainable, and easy to manage.

Why use Architectural Patterns?

- **1. Scalability:** Architectural patterns allow systems to grow smoothly by organizing components in a way that supports increased load or complexity.
- **2. Maintainability:** They separate concerns, making systems easier to update or debug without impacting other components.
- **3. Reusability:** Patterns are reusable and can be applied across multiple projects, speeding up development.
- **4. Efficiency:** Patterns provide pre-built solutions, reducing time spent on design and improving team collaboration by using familiar structures.
- **5. Flexibility:** Architectural patterns ensure systems are adaptable to future changes, allowing easy extension and modification.

Architectural Patterns

Layered

Client-Server

MVC

MVP

Representational State Transfer

Microservices

Controler-Responder

Multi-Tier

Microkernel

Map Reduce

MVVM

Layered Architecture

- Definition: The Layered Architecture, also known as the n-tier architecture, divides an application into multiple layers, each responsible for a specific aspect of the application.
- Layers:
  - Presentation Layer: Handles user interactions and displays the output (UI).
  - Business Logic Layer: Contains the core application logic (rules, operations).
  - Data Access Layer: Manages interaction with the database or external services.
  - Database Layer: Stores and retrieves data.
- Benefits:
  - Separation of Concerns: Each layer focuses on a specific task, making the system modular and easier to maintain.
  - Scalability: Layers can be independently modified or replaced.

Example: Traditional web applications where the front-end (UI), back-end logic (business), and database (data) are separated.

Client-Server Architecture

- **Definition** : The **Client-Server Architecture** splits the system into two main components: the **client** (which requests services) and the **server** (which provides services).
- **How It Works** :
  - **Client** : Sends requests to the server for data or services.
  - **Server** : Processes requests and returns appropriate responses.
- **Benefits** :
  - Centralized control on the server-side, which ensures data integrity and security.
  - Scalable as multiple clients can connect to the server simultaneously.

Model-View-Controller (MVC) Architecture

Definition : The **MVC** architecture divides an application into three interconnected components:

1. **Model** : Manages the data and business logic.
2. **View** : Displays the data (UI).
3. **Controller** : Receives user input and updates the Model and View accordingly.

Benefits :

- **Separation of Concerns** : Each component focuses on a specific responsibility, making the system easier to develop and test.
- **Maintainability** : Each component can be updated independently.

Example : Frameworks like **Ruby on Rails** and **ASP.NET MVC** follow the MVC pattern.

Model-View-Presenter (MVP) Architecture

Definition: The **MVP** architecture is a refinement of MVC, where the **Presenter** takes on the role of both the controller and the view model. It interacts with the **Model** and **View**, handling all the logic and acting as a middle layer between them.
Components:

1. **Model**: Contains the business logic and data.
2. **View**: A passive view that only displays information and interacts with the user.
3. **Presenter**: Handles the presentation logic and updates the View and Model.

Benefits:

- Provides better separation of concerns than MVC.
- The View is more passive, which improves testability.

Example: Common in Android development and WinForms applications.

Model-View-ViewModel (MVVM) Architecture

Definition : **MVVM** separates development into three components:

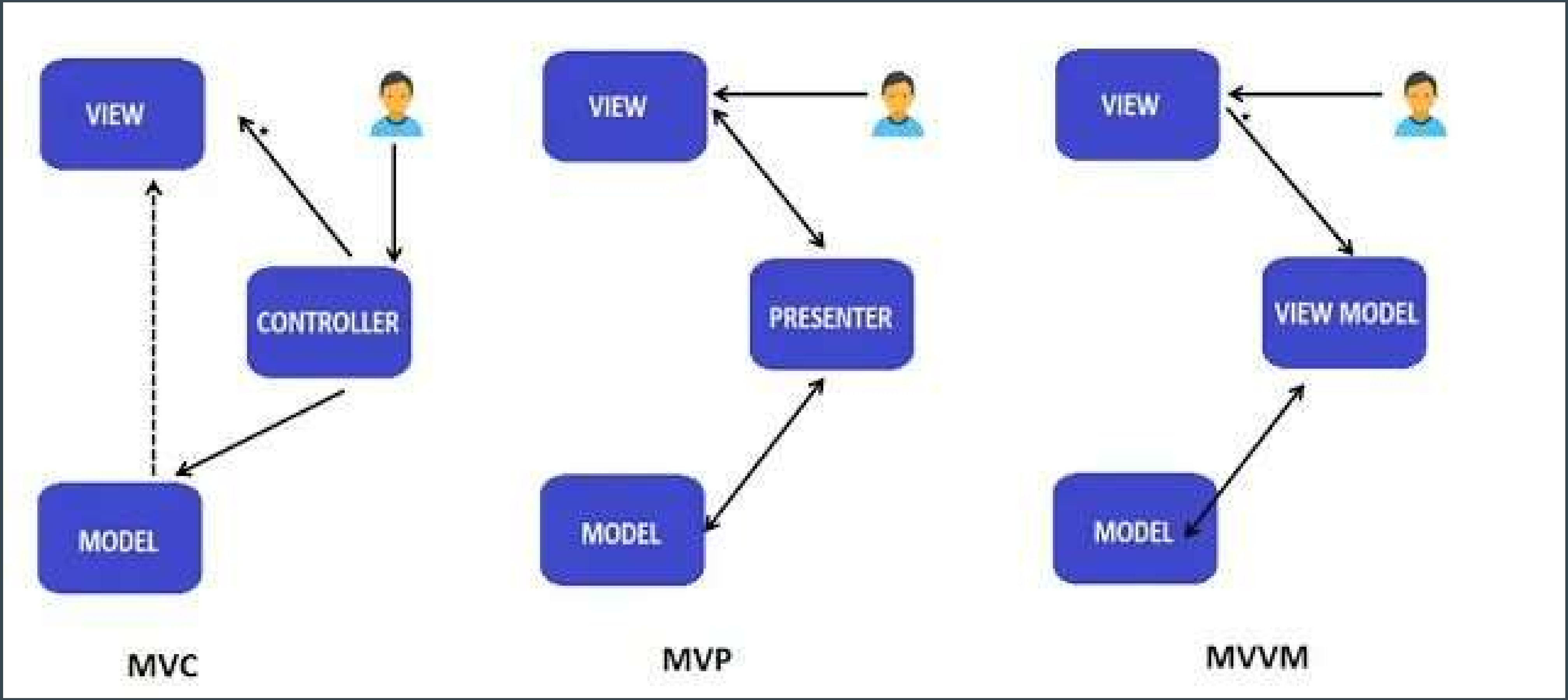
1. **Model** : Represents the data and business rules.
2. **View** : The UI that displays the data.
3. **ViewModel** : Acts as a mediator between the View and the Model, holding presentation logic and state.

Benefits :

- **Data Binding** : Automatically updates the View when the Model changes, and vice versa.
- **Testability** : The ViewModel can be tested independently from the UI.

Example : Common in **WPF** (Windows Presentation Foundation) applications and frameworks like **Angular** .

Too many acronyms? Here is a picture:



Microservices Architecture

Definition : The **Microservices** architecture divides an application into small, loosely coupled services. Each service is responsible for a specific piece of functionality and communicates with other services over APIs.

Benefits :

- **Scalability** : Each service can be scaled independently based on demand.
- **Independence** : Services can be developed, deployed, and maintained independently.
- **Flexibility** : Services can be written in different programming languages and use different technologies.

Example : **Netflix** uses microservices for different components like recommendation engines, streaming services, and payment gateways.

Controller-Responder Pattern

Definition : The **Controller-Responder Pattern** is an architectural pattern where the **Controller** receives input from a client, processes it, and hands off the response to a **Responder** which formats and sends the response back.

How It Works :

- **Controller** : Handles the logic and interaction with the model.
- **Responder** : Prepares and sends a response to the client.

Benefits :

- Clear separation of responsibilities.
- Easy to extend or modify the response format without changing core logic.

Example : Common in **RESTful APIs** , where the controller processes HTTP requests and the responder formats the output (e.g., JSON or XML).

Multi-tier Architecture

Definition : **Multi-tier architecture** is a more physical manifestation of the layered architecture, where each layer is hosted on a separate server. The typical tiers are:

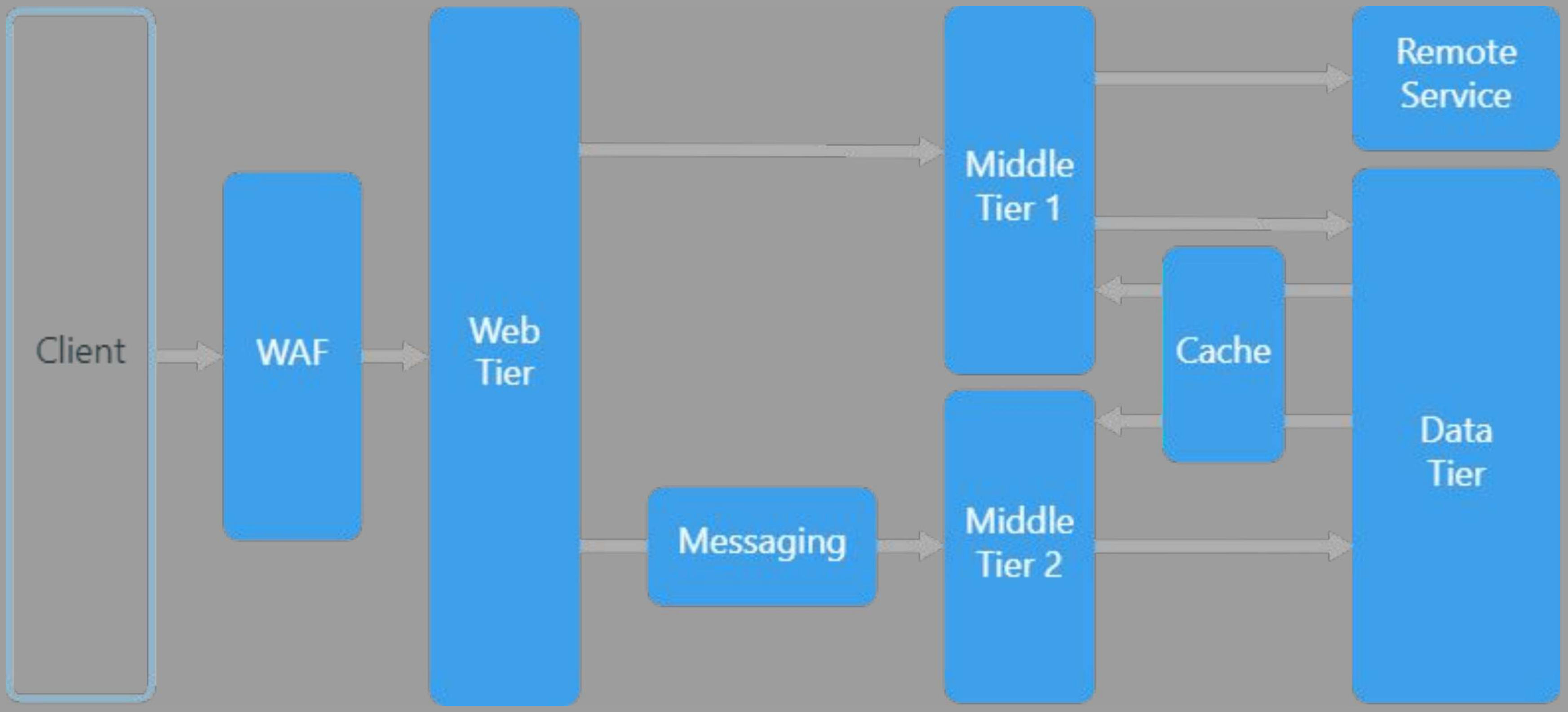
1. **Client Tier** : The user interface.
2. **Application Tier** : The business logic.
3. **Database Tier** : The data storage and retrieval system.

Benefits :

- **Scalability** : Each tier can be scaled independently.
- **Security** : Different security measures can be applied to each tier.

Example : A web application where the front-end (UI) runs on a web server, the business logic runs on an application server, and the database runs on a separate server.

Multi-Tier/N-Tier



Microkernel Architecture

Definition : The **Microkernel Architecture** structures an application around a small core system (kernel) that provides essential services, with additional functionality provided by plug-ins or extensions.
How It Works :

- **Core System (Kernel)** : Provides essential services like resource management, task scheduling, etc.
- **Plug-ins** : Add specialized functionality to the system without modifying the core.

Benefits :

- **Extensibility** : New features can be added as plug-ins without altering the core system.
- **Flexibility** : The core remains lightweight, and only necessary extensions are loaded.

Example : Operating systems like **Linux** use a microkernel approach, as do applications like the **Eclipse IDE** , which allows developers to add new plug-ins.

Representational State Transfer (REST) Architecture

Definition : **REST** is an architectural style for building scalable web services. It is based on stateless communication, typically using HTTP to transfer data between client and server.
Principles :

- **Stateless** : Each request from a client to a server must contain all the information needed to process the request, and the server doesn't store any session information.
- **Resource-Based** : Everything is treated as a resource (e.g., user, product), and resources are accessed using standard HTTP methods like GET, POST, PUT, DELETE.

Benefits :

- **Scalability** : Because REST is stateless, each request is processed independently, making it easy to scale across multiple servers.
- **Simplicity** : Built on top of HTTP, which is already widely understood.

Example : Web services like **Twitter** or **Facebook** expose REST APIs to allow developers to interact with their platform.

MapReduce Architecture

Definition : **MapReduce** is a programming model used for processing large datasets across distributed clusters. It breaks down the processing into two phases:

1. **Map**: Splits the dataset into smaller parts and processes them in parallel.
2. **Reduce** : Aggregates the results of the map phase to produce a final output.

How It Works :

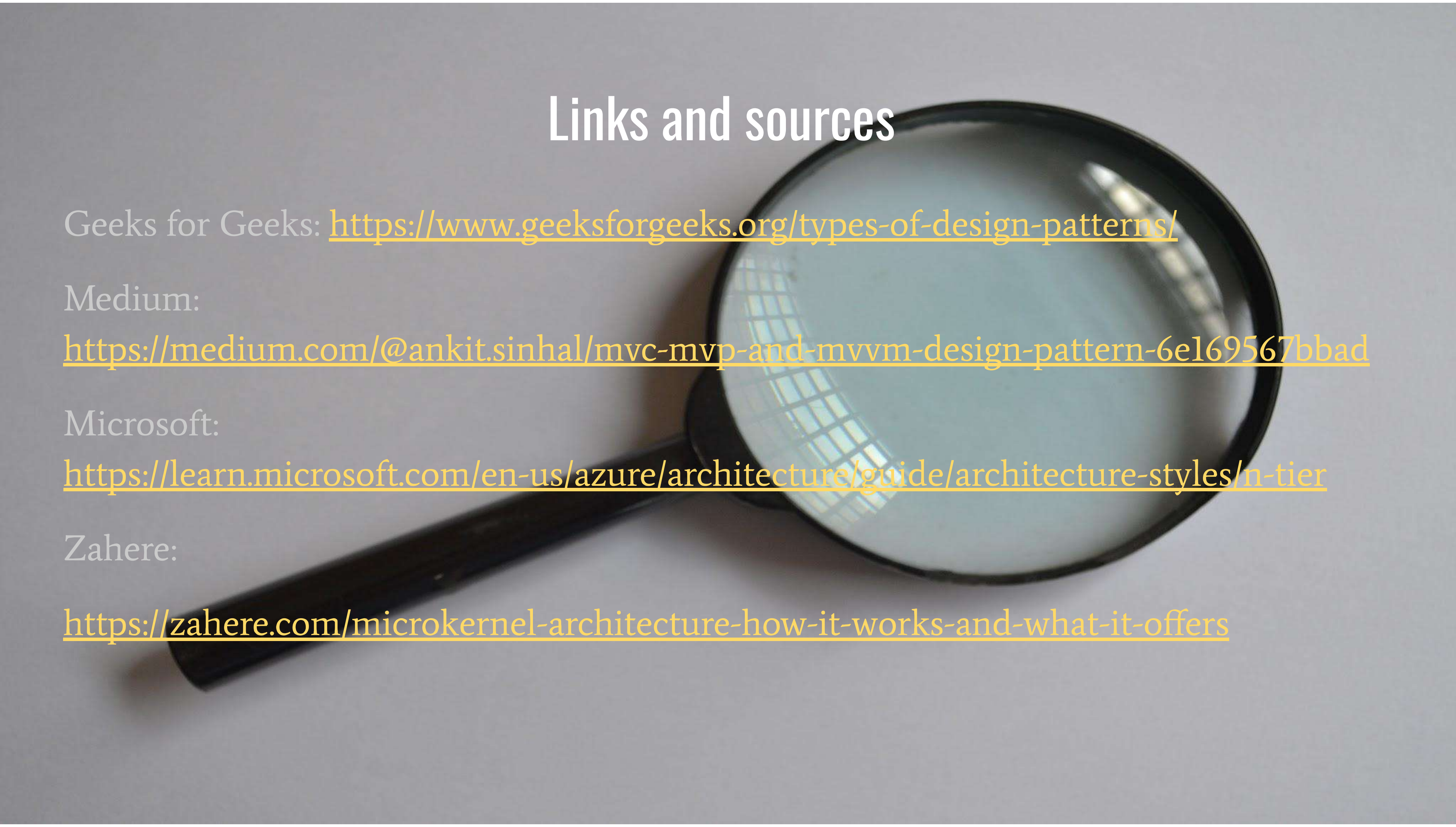
- **Map Phase** : The input is divided into smaller chunks, and the map function processes each chunk in parallel.
- **Reduce Phase** : The reduce function aggregates the intermediate results produced by the map functions to produce the final result.

Benefits :

- **Scalability** : Can process large datasets by distributing the work across multiple machines.
- **Fault Tolerance** : Built-in mechanisms for dealing with machine failures, rerunning tasks on other machines as needed.

Example : Google uses MapReduce for processing vast amounts of web data to index pages for their search engine.

Links and sources



Geeks for Geeks: https://www.geeksforgeeks.org/types-of-design-patterns/

Medium:
https://medium.com/@ankit.sinhal/mvc-mvp-and-mvvm-design-pattern-6e169567bbad

Microsoft:
https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier

Zahere:

https://zahere.com/microkernel-architecture-how-it-works-and-what-it-offers