# Software Architecture and Design Patterns 02

## Software Architecture

### What is Software Architecture ?

- The fundamental structure of a software system.
- Provides a high-level overview of components, relationships and goals.
- Act as a blueprint during design and development.
- It defines the components, relationships and how they work together to achieve the goals of a software system.

### Key Elements of Architecture

01. Components

    Functional units of a system that perform specific tasks. (UI, Database)

02. Connectors

    Methods that enable interaction between components. (APIs, protocols)

03. Constraints

    Design rules or limitations that guide the system. (performance, security)

### Why is it important ?

01. Maintainability.

    Simplifies software maintenance and requirement fulfilment.

02. Risk Management.

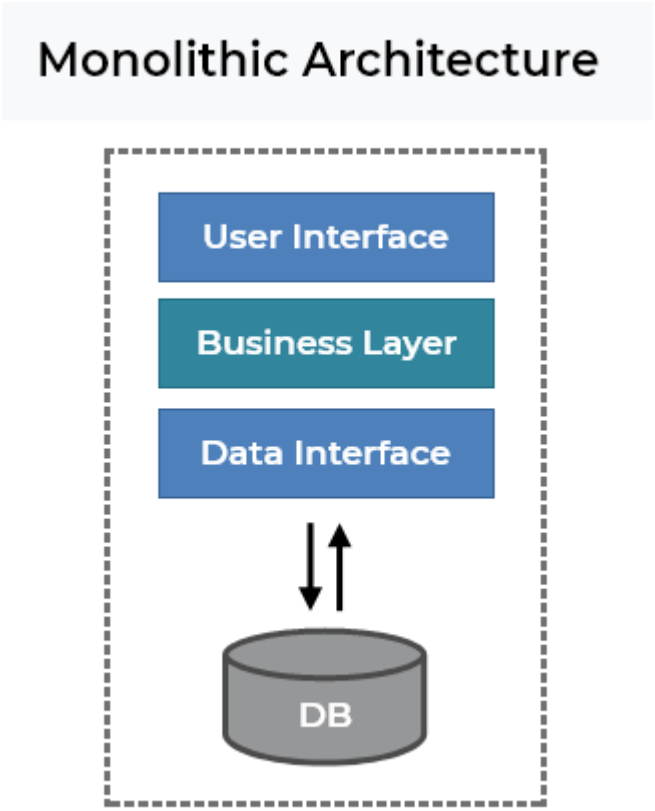
    Analyze risks before implementation.

03. Easy Team Collaboration.

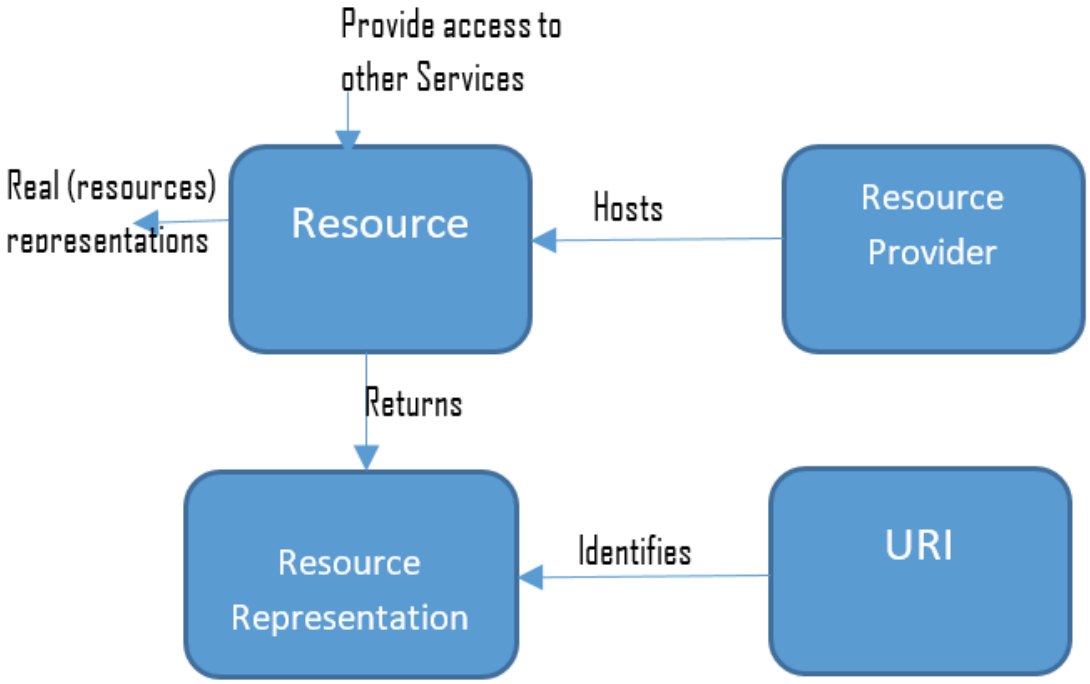
    Help team members understand the system.

[Types of Software Architectures]

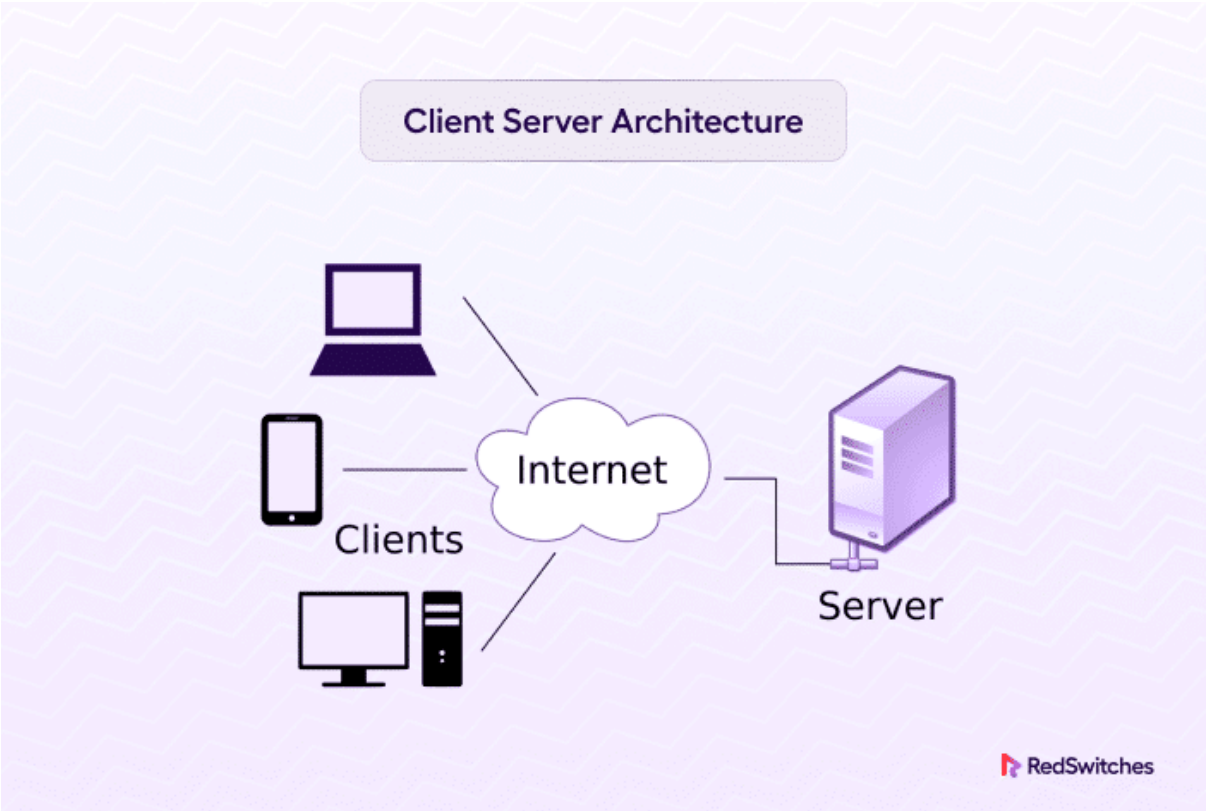
01. Monolithic Architecture
    a. A single self contained application.
    b. All components are tightly integrated into a single codebase.
    c. Easier to develop initially but harder to scale and maintain as the system grows.
    d. Design includes an illustration showing all components in a single block, representing tightly coupled elements.



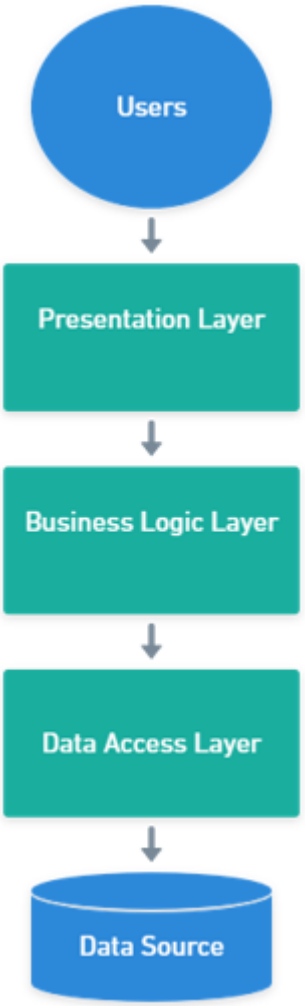

02. Resource-Oriented Architecture.
    a. Focuses on resources (data/ services) accessible via URLs.
    b. Commonly used in restful web services.



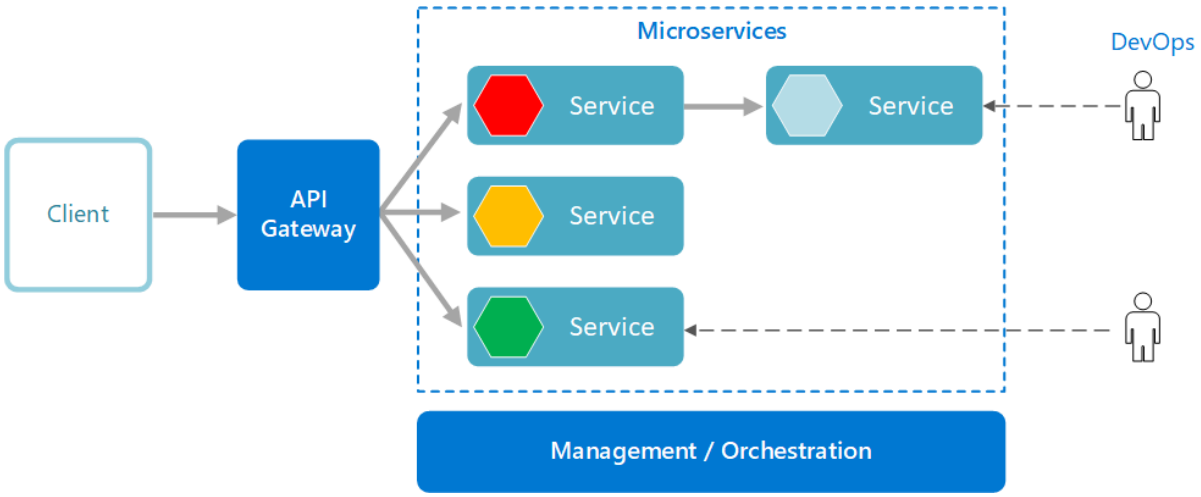

03. Client-Server Architecture.
    a. The system is split into clients and servers.
    b. Clients request services or resources from servers.



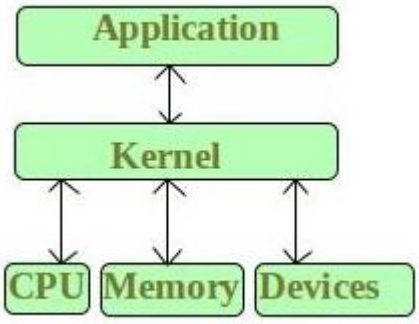
04. Layered Architecture.
    a. Organizes the software into layers. (presentation, business, persistence, data)
    b. Defined interactions between layers help separate concerns.




05. Microservices Architecture.
    a. Modular approach dividing the system into small, independent services.
    b. Services communicate through APIs, often via an API gateway.

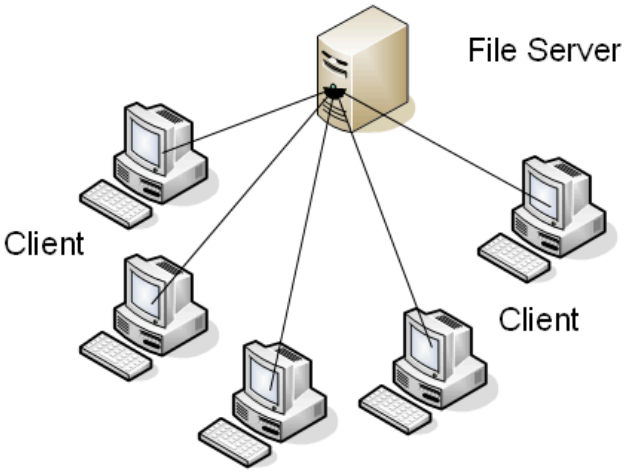



06. Microkernel Architecture.
    a. The system core maintains minimal functionality.
    b. Additional functionality is handled by independent modules or processes.
    c. Often used in embedded systems and IoT applications.

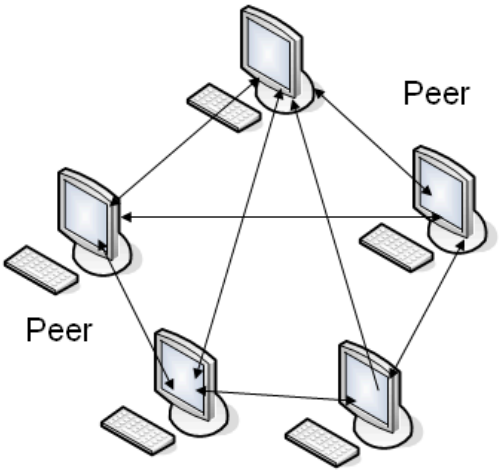



07. Peer to Peer Architecture.
    a. Decentralized network topology where each participants can act as both a client and server.
    b. Highly scalable and privacy focused.

    E.g. BitTorrent, Blockchain

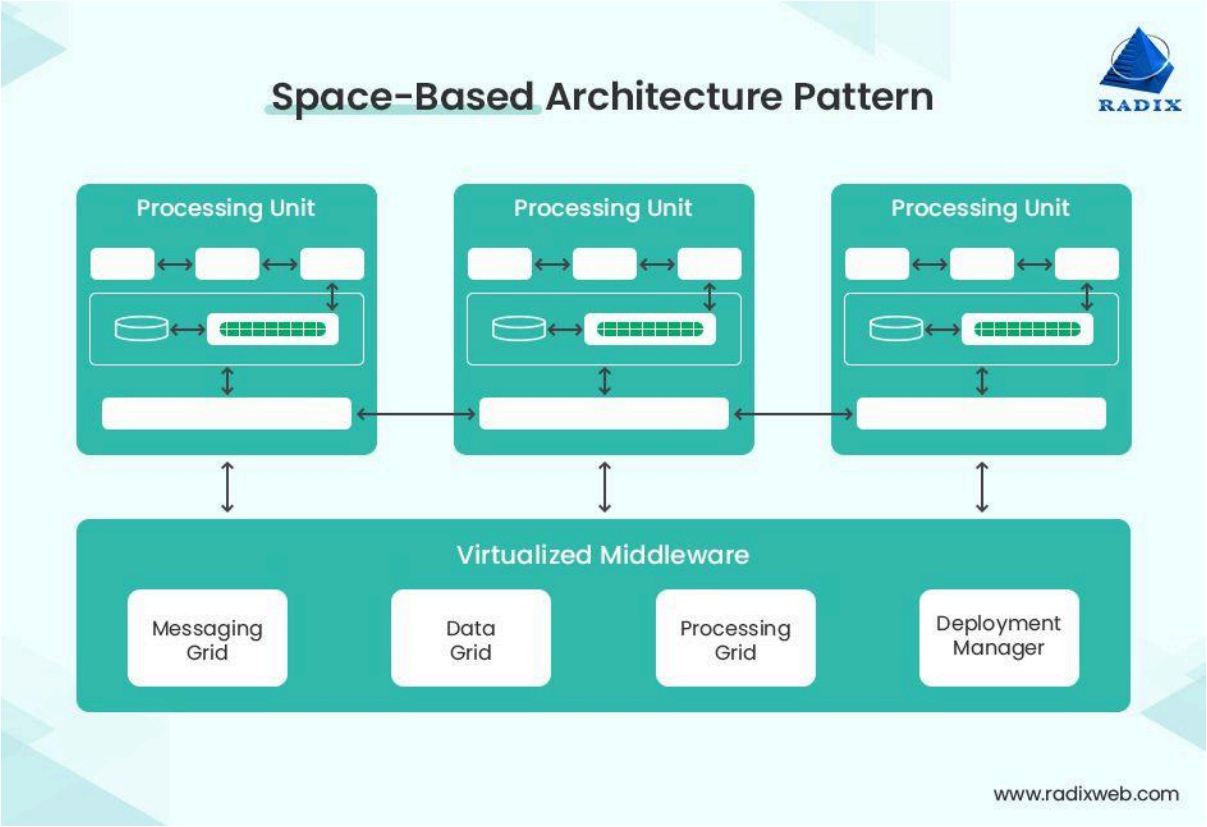



Client server mode

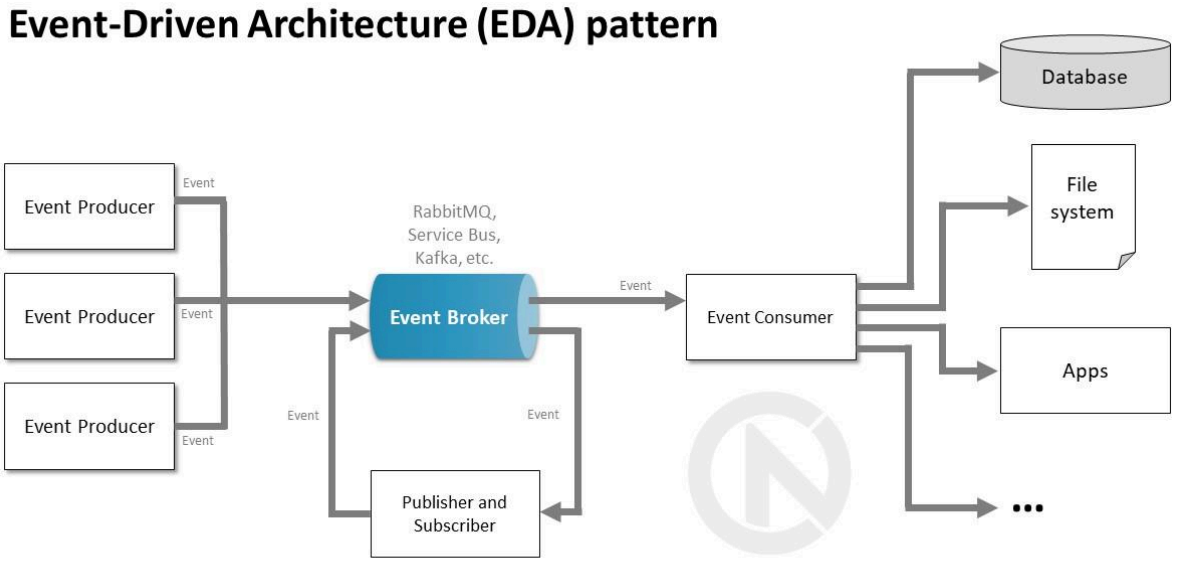



Peer to Peer mode

08. Space Based Architecture.
    a. Data and processing are distributed across multiple nodes or “spaces”.
    b. Ideal for large scale data systems like,
        Google file system.
        Amazon dynamoDB.
        Microsoft SQL server data services



09. Event Driven Architecture.
    a. An architecture style where components produce and respond to events, enabling real-time communication and decoupled, asynchronous interactions.
    b. This is commonly used in systems with real-time processing.
        Social media platforms.
        Messaging and notification systems.
        Online gaming.




[Role of Software Architects]

01. Requirements gathering.
    Defines project objectives and requirements, ensuring alignment with stakeholder needs.

02. High-level design.
    Creates a blueprint for the system's structure guiding the development process.

03. Scalability.
    Design systems to support growth and adaptability as user base or data increases.

04. Maintainability.
    Ensures the system is easy to update and debug overtime.

05. Reusability.
    Promotes component reuse to reduce redundancy and improve efficiency.

06. Testing support.
    Supports the creation of testable components and frameworks for thorough testing.

07. Technology adaptability.
    Design with flexibility to allow for easy updates and integration with new technologies.

08. Risk mitigation.
    Identifies potential risks early and implements strategies to migrate them.

09. Development efficiency.
    Streamline development tasks, minimizing complexity and reducing development time.

10. Team collaboration.
    Encourages effective teamwork, communication and alignment between developers and stakeholders.

11. User satisfaction.
    Focuses on creating a positive user experience through intuitive design and performance.

## Application Architecture vs Software Architecture

| Application Architecture | Software Architecture |
| --- | --- |
| Focuses on single application's design and structure. | Encompasses the entire ecosystem, including multiple applications and their interactions |

| | |
|---|---|
| Primarily concerned with the internal design of a single app. | Deals with system-wide design and the integration of various subsystems. |
| Typically smaller in scope. (for one application) | Larger scale, managing the interconnections of various systems and services. |
| Includes components like UI, business logic and databases of one app. | Involves multiple systems, services, middleware, data flows and interactions. |
| Aims to optimize a single application for performance, user experience and maintainability. | Focuses on ensuring scalability, maintainability and performance across a system of applications. |
| Primarily concerned with UI/UX design, application structure and responsiveness. | Focuses on modularity, scalability, security and ensuring that different applications can communicate effectively. |
| Deployed as a single unit. (application) | Involves multiple deployments across services and systems. |
| Generally, a single technology stack for one application. | Uses multiple technologies across different services and applications. |
| Less complex as it deals with one app. | More complex as it involves interdependent systems and services. |
| Maintenance focuses on one application's updates and performance. | Maintenance addresses system-wide updates and ensures all applications work seamlessly together. |
| Low interdependence, primarily self-contained. | High interdependence as it requires multiple systems to function together. |

## Software Architecture Characteristics

Characteristics define the qualities and properties that a software system should have to meet non-functional requirements.

01. Structural Characteristics

    How the system is organized and designed.

- Modularity

  Divides the system into independent modules for easier maintenance.

- Layering
  Organizes the system into distinct layers to separate concerns.
- Hierarchical decomposition
  Breaks the systems into manageable subsystems.
- Information hiding
  Hides internal details to reduce complexity.
- Encapsulation
  Controls data and functionality access, improving modularity.

02. Quality Attributes
    Non-functional aspects like performance, scalability and security.

- Performance
  Efficient use of resources and fast response times.
- Scalability
  Ability to handle increased load by adding resources.
- Reliability
  Consistency in system performance with minimal failures.
- Security
  Protects data and system from unauthorized access.
- Maintainability
  Easy to modify and extend overtime.

## Quality-Driven Attributes

Quality-driven attributes are essential for ensuring the overall quality of a software system.

- Performance
  System's ability to run efficiently.
- Scalability
  Ability to handle increased load.
- Reliability
  Consistent operation with minimal failures.
- Security
  Protection of data and resources.
- Maintainability
  Ease to make updates and fixes.
- Usability
  User-friendly interfaces and experience.

- Safety
  Ensuring system and user safety.

## Design Principles & Patterns

### Design Principles

Design principles represent high-level guidelines or best practices that software developers should consider while designing system architecture.They are broad concepts that can guide decisions, rather than specific solutions to problems.

There are many design principles, but some of the most popular ones in object-oriented design are the SOLID principles:

01. Single Responsibility Principle
    A class should have only one reason to change.

02. Open/Closed Principle
    Software entities should be open for extension, but closed for modification.

03. Liskov Substitution Principle
    Objects of a superclass should be able to be replaced with objects of a subclass without affecting correctness.

04. Interface Segregation Principle
    Clients should not be forced to implement interfaces they don't use.

05. Dependency Inversion Principle
    High-level modules shouldn't depend on low-level modules; both should depend on abstractions.

### Design Patterns

Design patterns are general repeatable solutions to commonly occurring problems in software design. They aren't templates or actual code, but more like guidelines to solve a particular problem in a certain way.

Design patterns can be seen as templates for how to solve software problems. They represent solutions that have been developed and evolved over a long period by experienced software developers.

Design patterns are usually categorized into three types:

01. Creational Patterns.
    Concerned with object creation.

    - Singleton Design Pattern
    - Factory Design Pattern
    - Prototype Design Pattern
    - Builder Design Pattern

02. Structural Patterns.
    Concerned with how classes and objects are composed to form larger structures.

    - Facade Design Pattern
    - Decorator Design Pattern
    - Proxy Design Pattern
    - Adapter Design Pattern
    - Bridge Design Pattern
    - Composite Design Pattern

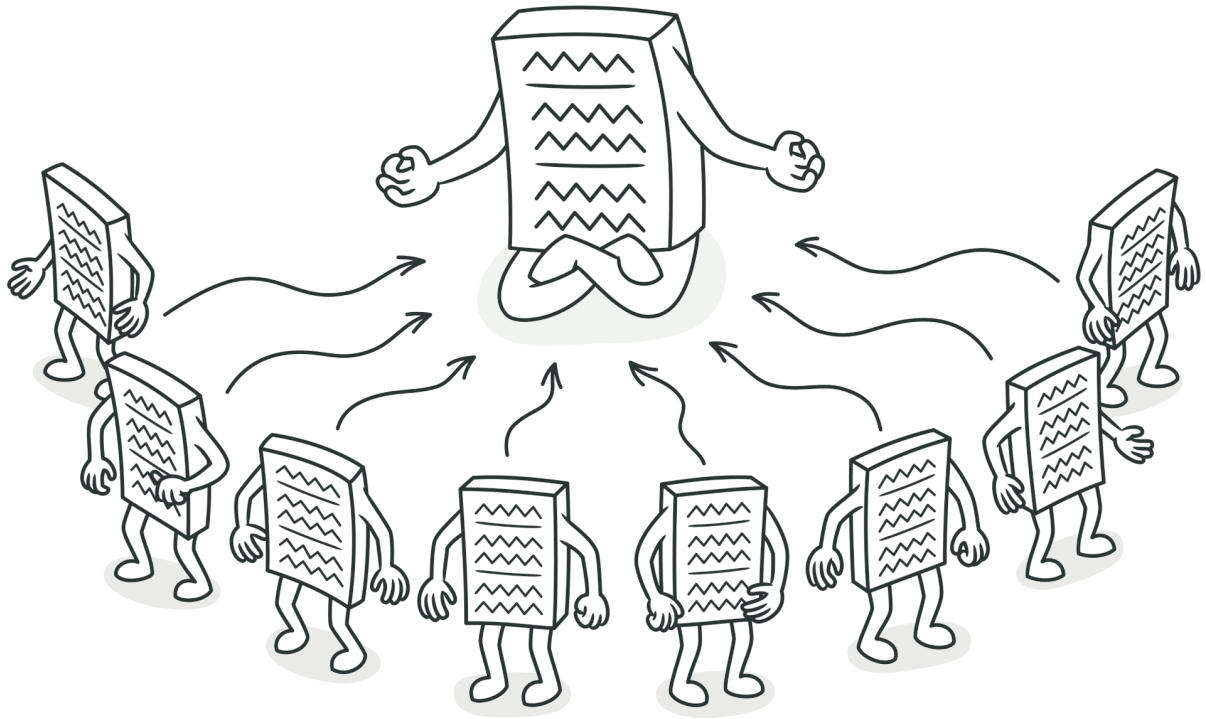
03. Behavioral Patterns.
    Define ways to communicate between objects.

    - Observer Design Pattern
    - Strategy Design Pattern
    - Mediator Design Pattern
    - State Design Pattern
    - Memento Design Pattern
    - Visitor Design Pattern
    - Iterator Design Pattern

Creational Patterns

## 01. Singleton Design Pattern

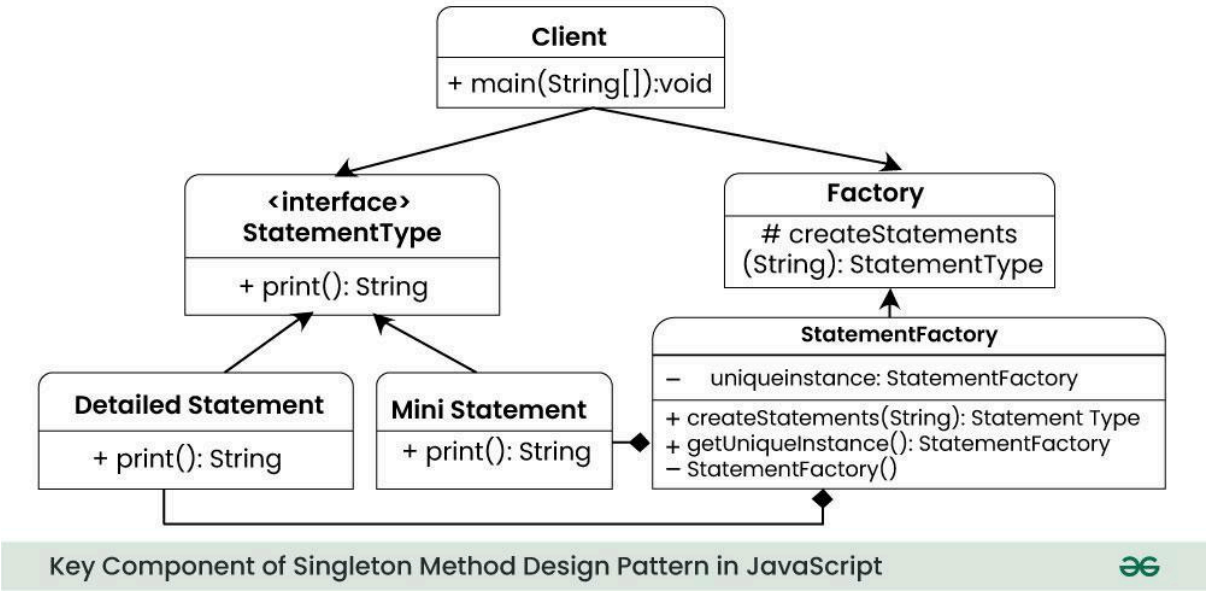
The Singleton design pattern ensures a class only has one instance, and provides a global point of access to it.



When to use Singleton

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Key Component of Singleton




- Private Static Instance

  The class should have a private static instance of itself. This instance is the single instance that the pattern ensures.

- Private Constructor

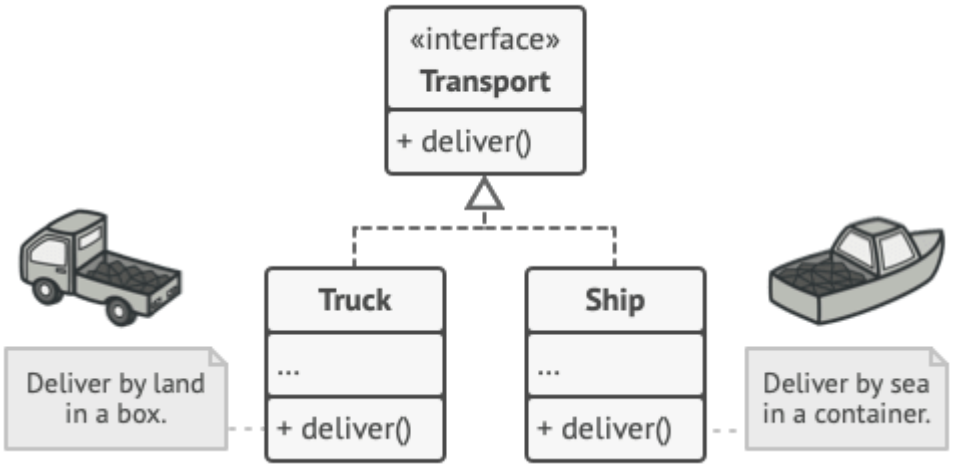
  The class should have a private constructor to prevent instantiation of the class from external classes.

- Public Static Method to Access the Instance

  The class provides a public static method that returns the instance of the class. If an instance does not exist, it creates one; otherwise, it returns the existing instance.

**02. Factory Design Pattern**

This pattern is typically helpful when it's necessary to separate the construction of an object from its implementation. With the use of this design pattern, objects can be produced without having to define the exact class of object to be created




## When to use Factory

- A class can't anticipate the class of objects it must create.
- A class wants its subclass to specify the objects it creates.
- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

## Key Component of Factory

- Creator

  This is an abstract class or an interface that declares the factory method. The creator typically contains a method that serves as a factory for creating objects. It may also contain other methods that work with the created objects.

- Concrete Creator

  Concrete Creator classes are subclasses of the Creator that implement the factory method to create specific types of objects. Each Concrete Creator is responsible for creating a particular product.

- Product

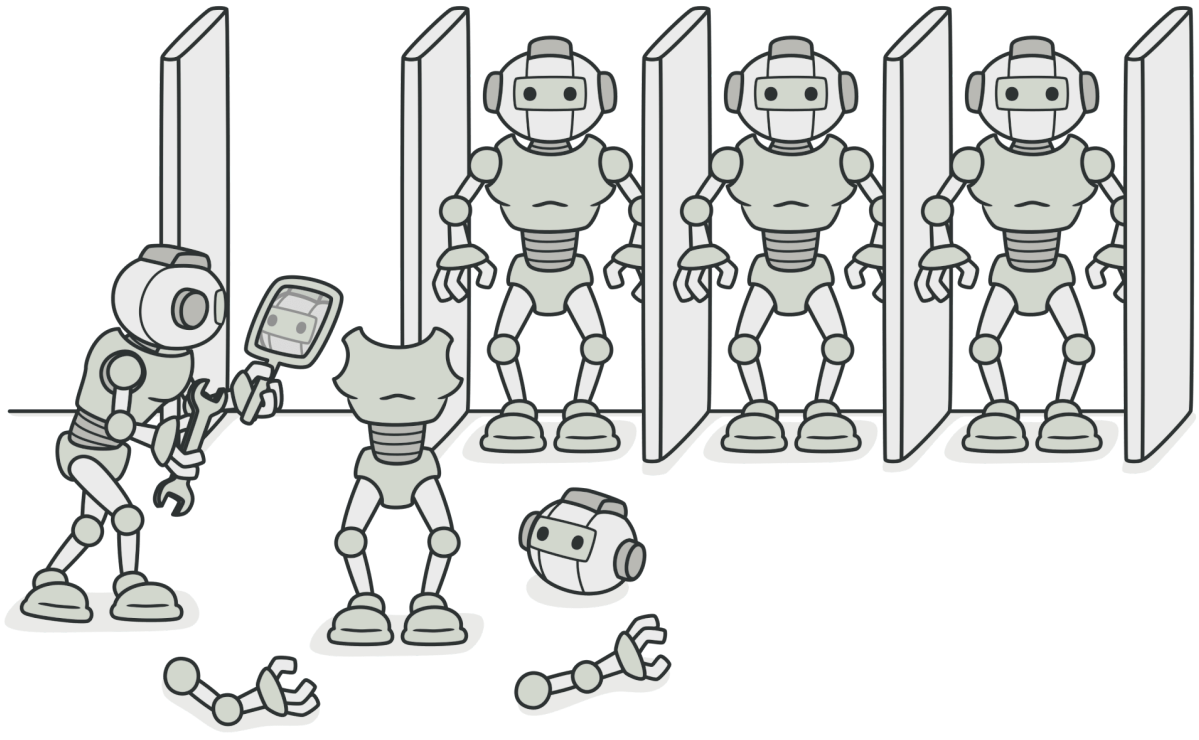
This is the interface or abstract class for the objects that the factory method creates. The Product defines the common interface for all objects that the factory method can create.

- Concrete Product

  Concrete Product classes are the actual objects that the factory method creates. Each Concrete Product class implements the Product interface or extends the Product abstract class.

### 03. Prototype Design Pattern

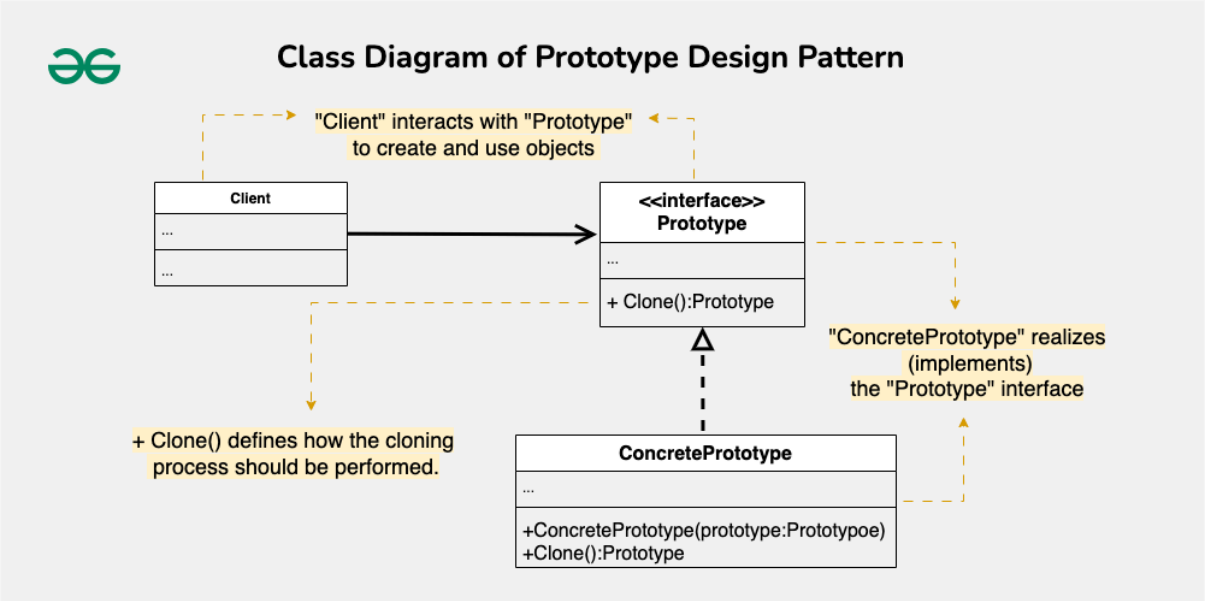
Prototype allows us to hide the complexity of making new instances from the client. The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations



## When to use Prototype

- Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and
- When the classes to instantiate are specified at run-time, for example, by dynamic loading.
- To avoid building a class hierarchy of factories that parallels the class hierarchy of products.

## Key Component of Prototype




- Prototype Interface or Abstract Class
  - This defines the method for cloning objects and sets a standard that all concrete prototypes must follow. Its main purpose is to serve as a blueprint for creating new objects by outlining the cloning contract.
  - It includes a clone method that concrete prototypes will implement to create copies of themselves.
- Concrete Prototype

    - This class implements the prototype interface or extends the abstract class. It represents a specific type of object that can be cloned.

    - The Concrete Prototype details how the cloning process should work for instances of that class and provides the specific logic for the clone method.

- Client
    - The Client is the code or module that requests new object creation by interacting with the prototype.

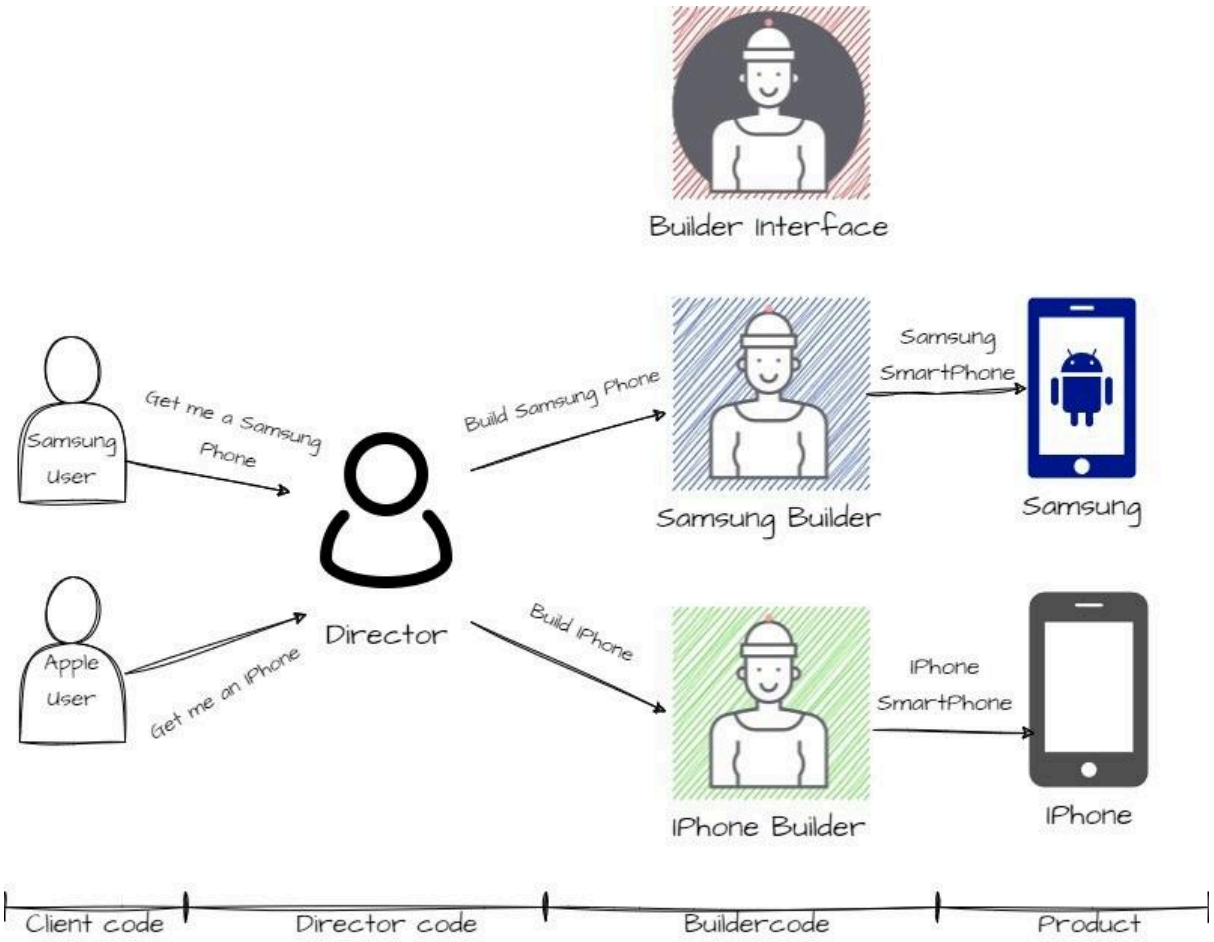
    - It initiates the cloning process without needing to know the specifics of the concrete classes involved.

- Clone Method
    - This method is declared in the prototype interface or abstract class and outlines how an object should be copied.

    - Concrete prototypes implement this method to define their specific cloning behavior, detailing how to duplicate the object's internal state to create a new, independent instance.

**04. Builder Design Pattern**

Builder Design Pattern is used to Separate the construction of a complex object from its representation so that the same construction process can create different representations. It helps in constructing a complex object step by step and the final step will return the object.



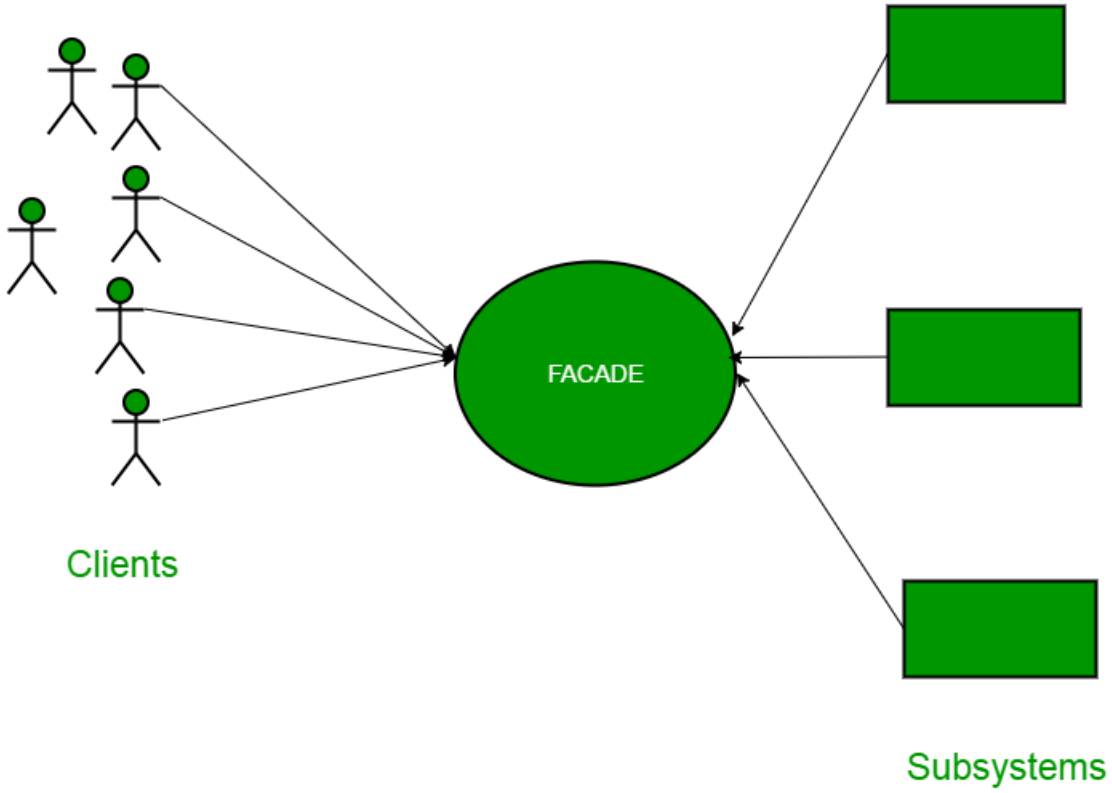

When to use Builder

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- The construction process must allow different representations for the object that's constructed.

Structural Patterns

**01. Facade Design Pattern**

Facade Method Design Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



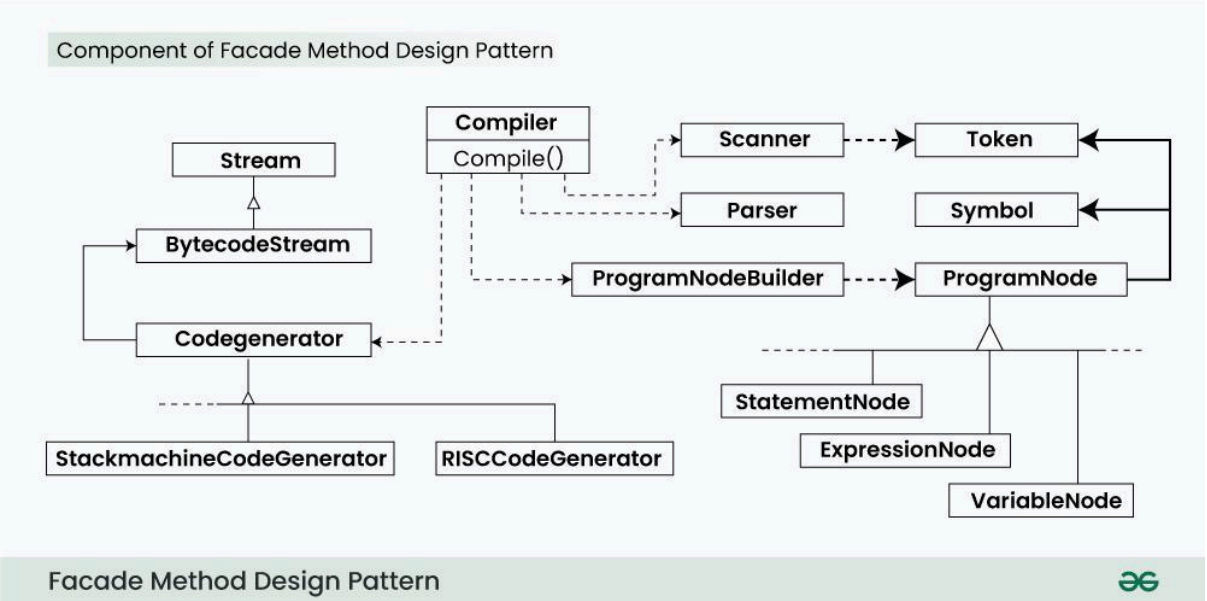

[When to use Facade]

- You want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don't need to customize it.

- There are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems.

- You want to layer your subsystems. Use a facade to define 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 through their facades.

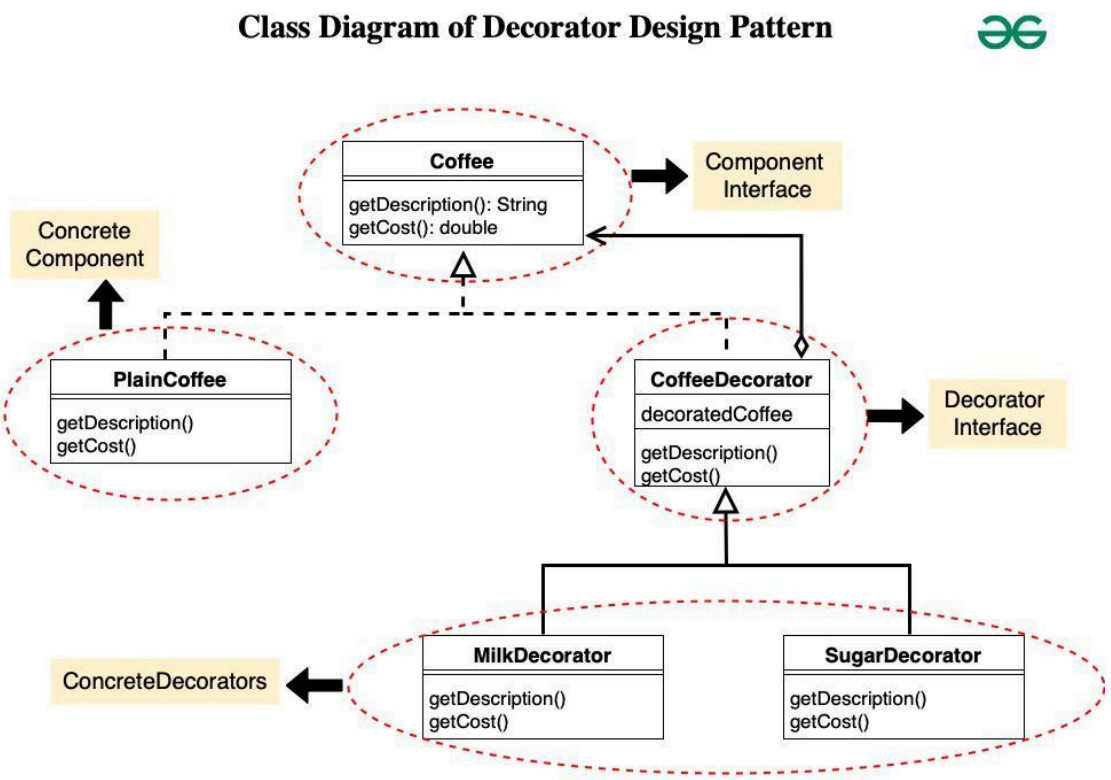
[Key Component of Facade]




- Facade
  - Facade knows which subsystem classes are responsible for a request.
  - It delegates client requests to appropriate subsystem objects.
- Subsystem classes (Scanner, Parser, ProgramNode, etc.)
  - It implements subsystem functionality.
  - It handles work assigned by the Facade object.
  - It has no knowledge of the facade; that is, they keep no references to it.
- Interface
  - The Interface in the Facade Design Pattern refers to the set of simplified methods that the facade exposes to the client.
  - It hides the complexities of the subsystem, ensuring that clients interact only with high-level operations, without dealing with the underlying details of the system.

## 02. Decorator Design Pattern

Adding additional features or actions to an item without changing its structure is possible with the Decorator Method Design Pattern. Consider that you want to add things like milk, sugar, or whipped cream to your simple coffee. Instead of creating a whole new coffee type for every possible combination, the decorator pattern lets you "wrap" the plain coffee with add-ons.




[When to use Decorator]

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- For responsibilities that can be withdrawn.
- When extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.

## Key Component of Decorator

- Component Interface

  This is an abstract class or interface that defines the common interface for both the concrete components and decorators. It specifies the operations that can be performed on the objects.

- Concrete Component

  These are the basic objects or classes that implement the Component interface. They are the objects to which we want to add new behavior or responsibilities.

- Decorator

  This is an abstract class that also implements the Component interface and has a reference to a Component object. Decorators are responsible for adding new behaviors to the wrapped Component object.

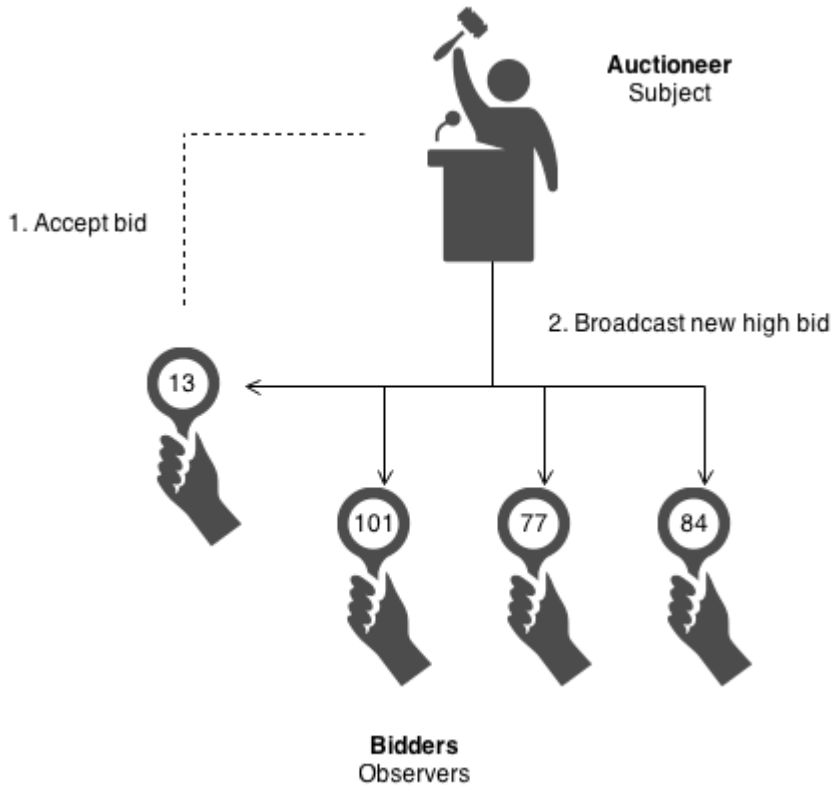
- Concrete Decorator

  These are the concrete classes that extend the Decorator class. They add specific behaviors or responsibilities to the Component. Each Concrete Decorator can add one or more behaviors to the Component.

## Behavioral Patterns

### 01. Observer Design Pattern

Observer method or Observer design pattern also known as dependents and publish-subscribe. It define a one to many dependency between objects so that when one objects so that when one object change state, all its dependents are notified and updated automatically




When to use Observer

- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are.

Key Component of Observer

- Subject
  - The subject maintains a list of observers (subscribers or listeners).
  - It Provides methods to register and unregister observers dynamically and defines a method to notify observers of changes in its state.

- Observer
    - Observer defines an interface with an update method that concrete observers must implement and ensures a common or consistent way for concrete observers to receive updates from the subject.

- ConcreteSubject
    - ConcreteSubjects are specific implementations of the subject. They hold the actual state or data that observers want to track. When this state changes, concrete subjects notify their observers.

    - For instance, if a weather station is the subject, specific weather stations in different locations would be concrete subjects.

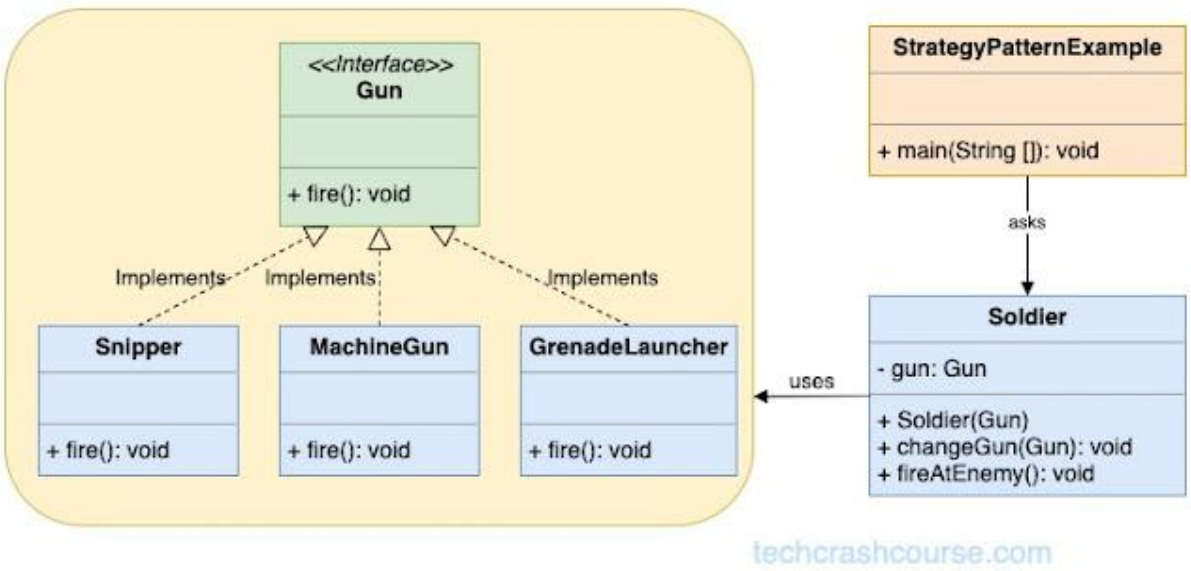
- ConcreteObserver
    - Concrete Observer implements the observer interface. They register with a concrete subject and react when notified of a state change.

    - When the subject's state changes, the concrete observer's update() method is invoked, allowing it to take appropriate actions.

    - For example, a weather app on your smartphone is a concrete observer that reacts to changes from a weather station.

**02. Strategy Design Pattern**

Strategy method or Strategy Design Pattern also known as Policy, it defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.




## When to use Strategy

- 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.

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

## Key Component of Strategy

- Context

  A class or object known as the Context assigns the task to a strategy object and contains a reference to it.

  - It serves as an intermediary between the client and the strategy, offering an integrated approach for task execution without exposing every detail of the process.

    - The Context maintains a reference to a strategy object and calls its methods to perform the task, allowing for interchangeable strategies to be used.

- Strategy Interface

  An abstract class or interface known as the Strategy Interface specifies a set of methods that all concrete strategies must implement.

    - As a kind of agreement, it guarantees that all strategies follow the same set of rules and are interchangeable by the Context.

    - The Strategy Interface promotes flexibility and modularity in the design by establishing a common interface that enables decoupling between the Context and the specific strategies.

- 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.

- Client

  The Client is responsible for selecting and configuring the appropriate strategy and providing it to the Context.

    - It knows the requirements of the task and decides which strategy to use based on those requirements.

- The client creates an instance of the desired concrete strategy and passes it to the Context, enabling the Context to use the selected strategy to perform the task.