Behavioral Patterns

Factory Method

Design Patterns

CHEAT SHEET

SCALER Topics

Abstract Factory

KISS, DRY

What's a Design Pattern?

Design patterns are recognized approaches to solve commonly occurring problems in software design. They are like pre-made blue-prints that you can customize to solve a recurring object-oriented design problem in your code.

Categories of Design Patterns

Creational Patterns

Creational design patterns simplify object creation in software design, allowing for flexibility and code reuse. By providing mechanisms to handle varying object nature, these patterns reduce complexity and enhance the overall design.

Structural Patterns

Structural design patterns organize objects and classes to achieve multiple objectives. They illustrate how system components can be combined in a flexible and extensible manner, facilitating targeted modifications without impacting the entire structure.

Behavioral Patterns

Behavioral design patterns address object interaction in software applications and offer solutions for common interaction issues in object-oriented design. They identify communication patterns among objects and provide guidelines to solve frequently encountered problems.

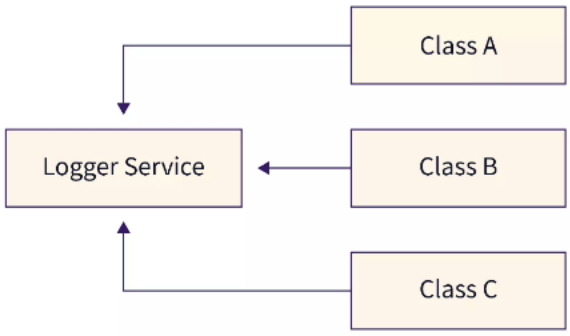
Creational Patterns

Singleton

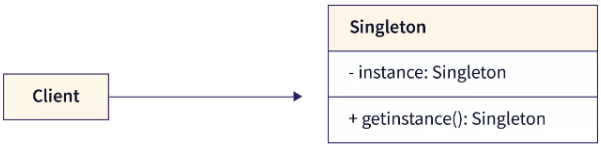
Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

Analogy

The LoggerService class, using the Singleton pattern, ensures efficient memory use in a project by creating a single, globally accessible instance for logging information, warnings, and errors, thus preventing out-of-memory errors.



Structure

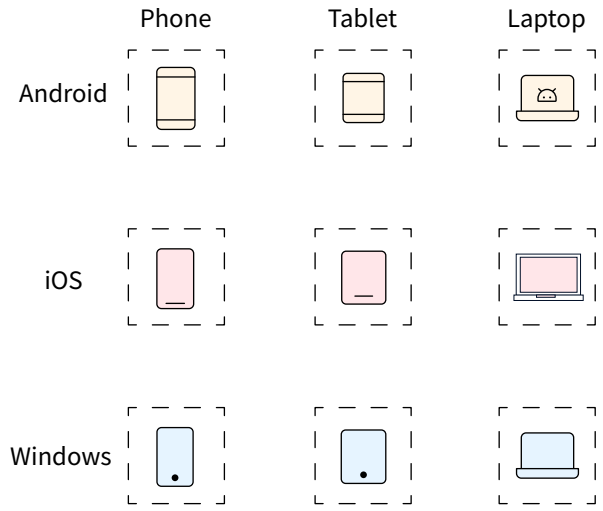


Abstract Factory

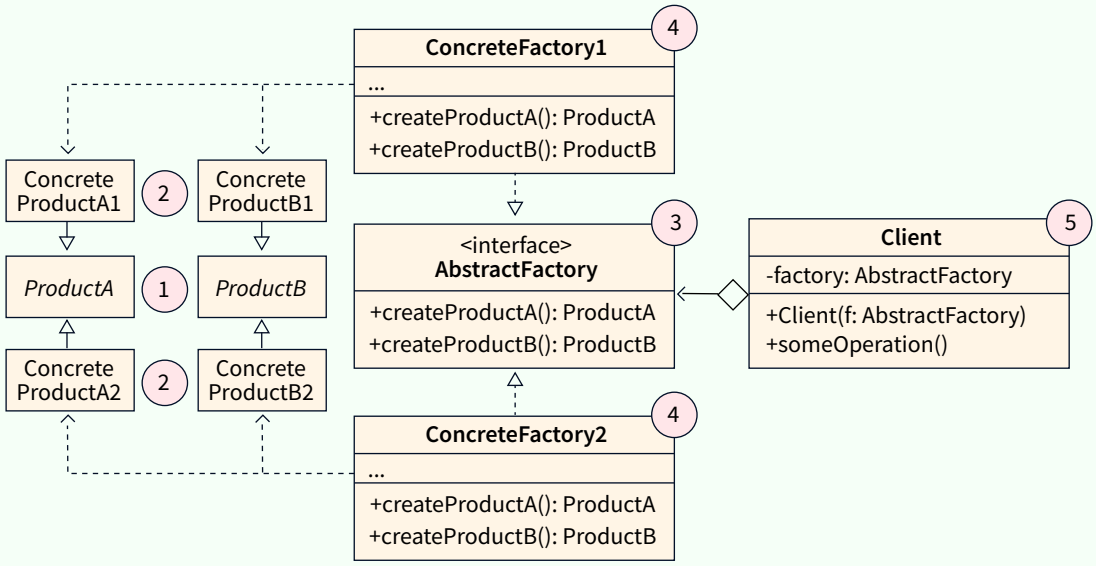
The Abstract Factory pattern is an extension of the Factory pattern. It allows the creation of object families without specifying concrete classes, resembling a hierarchical factory of factories.

Analogy

To ensure customer satisfaction, we create individual products that match others within the family of Phone, Tablet, and Laptop. Variants such as Android, iOS, and Windows are available for these devices. Inconsistent combinations of food disappoint customers.



Structure

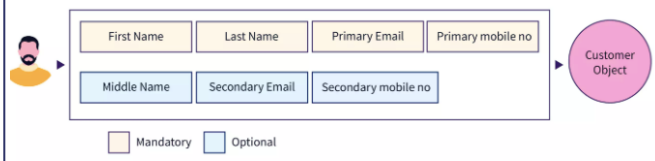


Builder

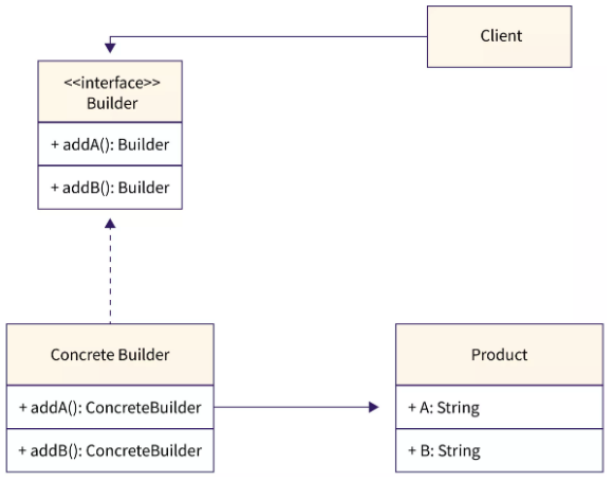
The Builder pattern constructs complex objects incrementally, similar to building a multi-layered cake. Each step adds an ingredient or component, allowing customization and diversity using a consistent building process.

Analogy

Customer details are stored in a database using a Customer class with mandatory fields (First Name, Last Name, Primary Email, and Primary Mobile Number) and optional fields (Middle Name, Secondary Email, and Secondary Mobile Number) passed as null values using the builder design pattern for flexible object creation.



Structure



Prototype

The Prototype pattern creates new objects by copying existing ones, ensuring code independence from their classes. It enables the creation of prototype objects that replicate the original ones.

Analogy

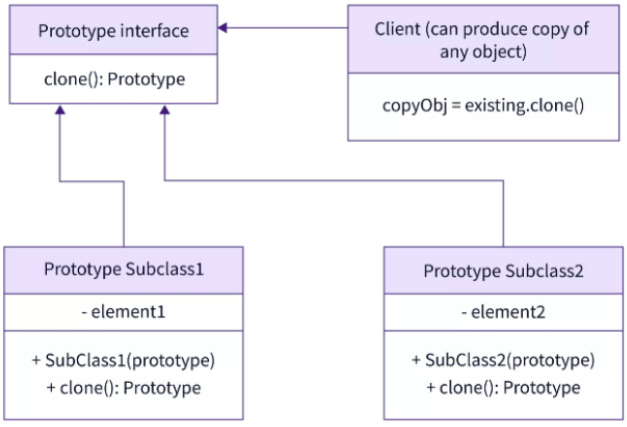
Say, you want to create an exact copy of an object. To perform this, you need to know the class of the object and its fields which may be private. Prototype delegates the work of cloning objects to the actual object itself.

" Do you think we'd be able to copy it? "

One Month After

"Perfect!"

Structure

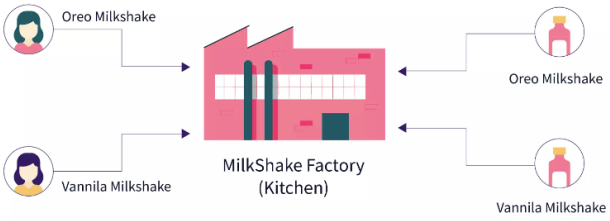


Factory Method

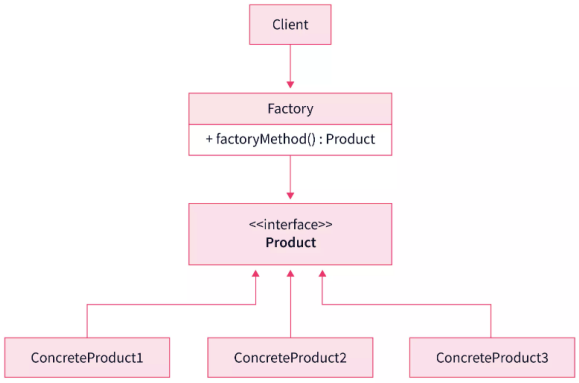
The Factory pattern is a creational design pattern that separates object creation logic from client code. A factory class uses a method to create objects based on client requests.

Analogy

Consider a milkshake bar. Customers order from a waiter, the order goes to the kitchen, and the kitchen prepares the milkshakes. The kitchen is like a shake factory, taking and fulfilling orders.



Structure



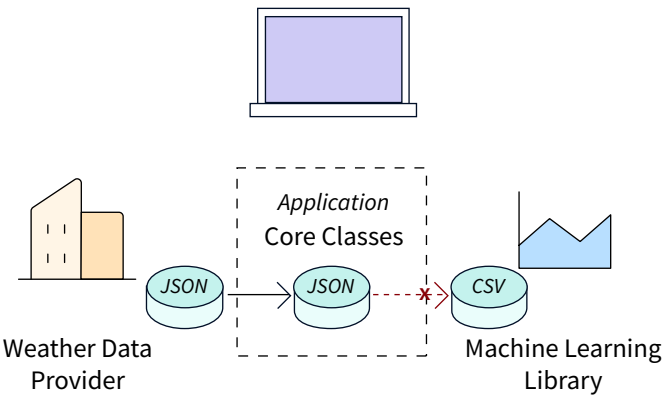
Structural Patterns

Adapter

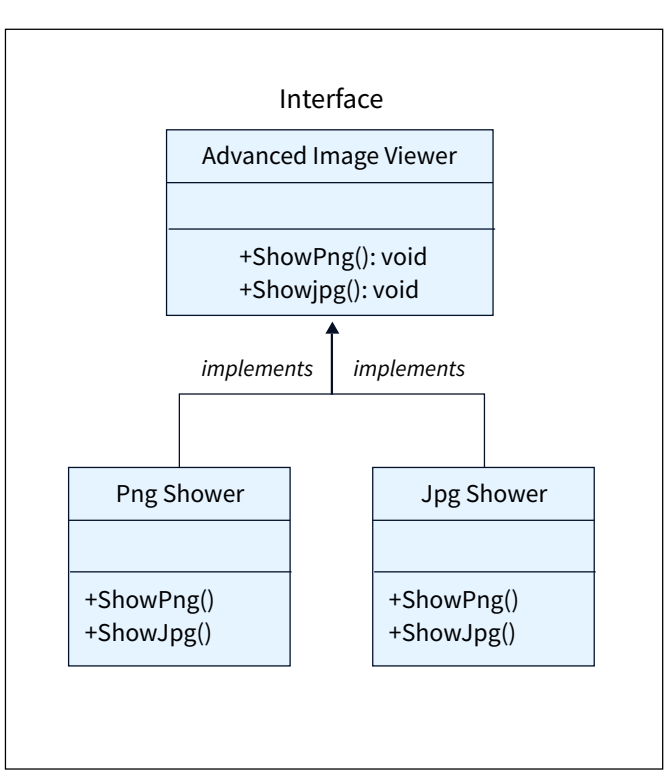
The Adapter pattern acts as a bridge between initially incompatible interfaces, facilitating collaboration between objects.

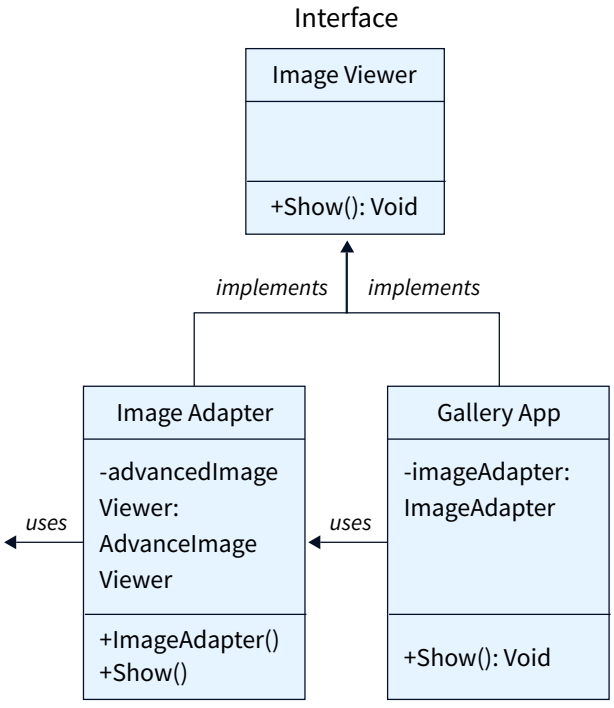
Analogy

A weather forecasting application retrieves JSON-formatted weather data from multiple sources and displays visually appealing forecasts and maps. To improve functionality, integrating a machine learning library that only supports CSV data format becomes a challenge.



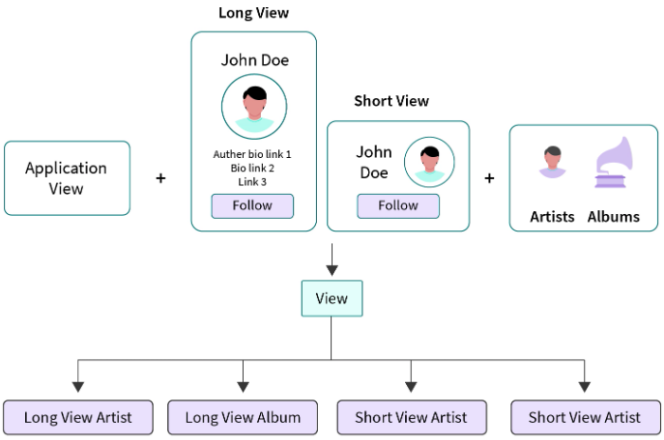
Structure





Bridge

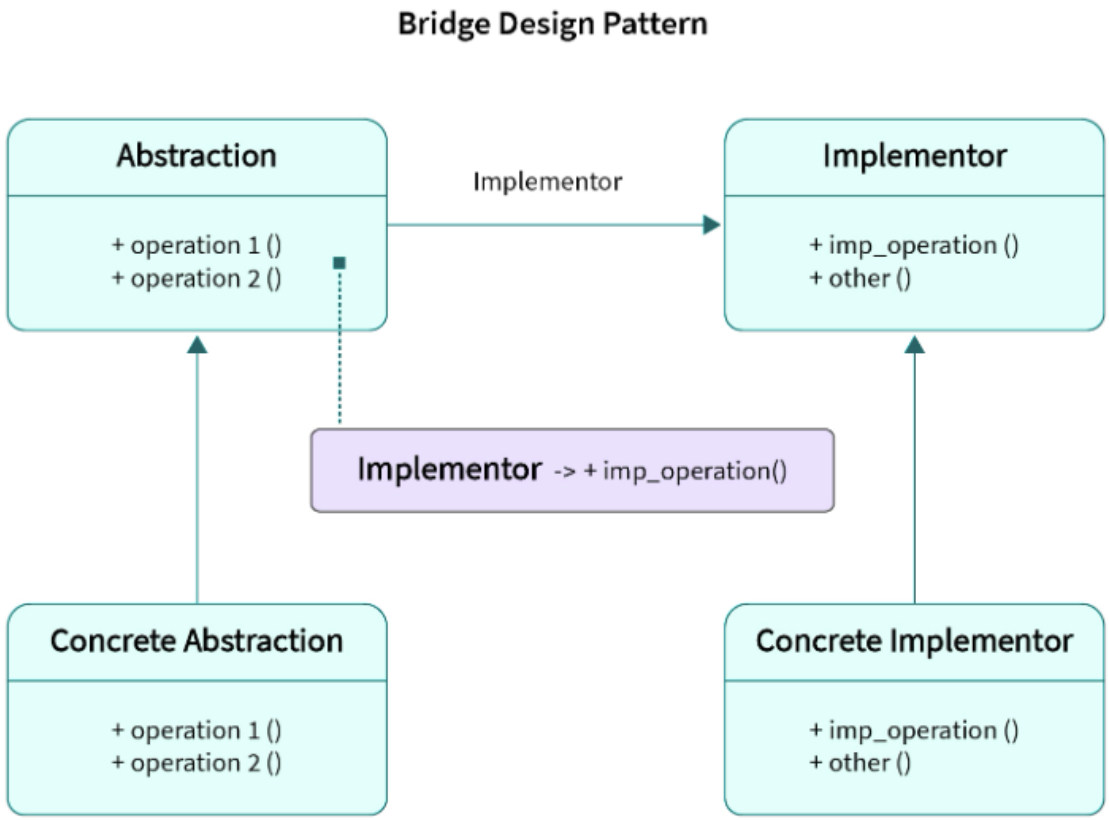
The Bridge pattern separates complex classes into abstraction and implementation hierarchies, allowing flexible connectivity through object composition for easier development and maintenance.



Analogy

In designing a multimedia application, different views like Long and Short views are implemented to present media content such as artist information and music albums.

Structure

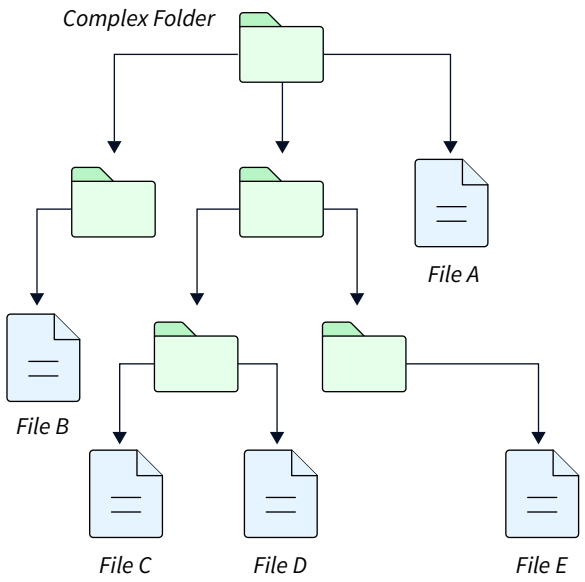


Composite

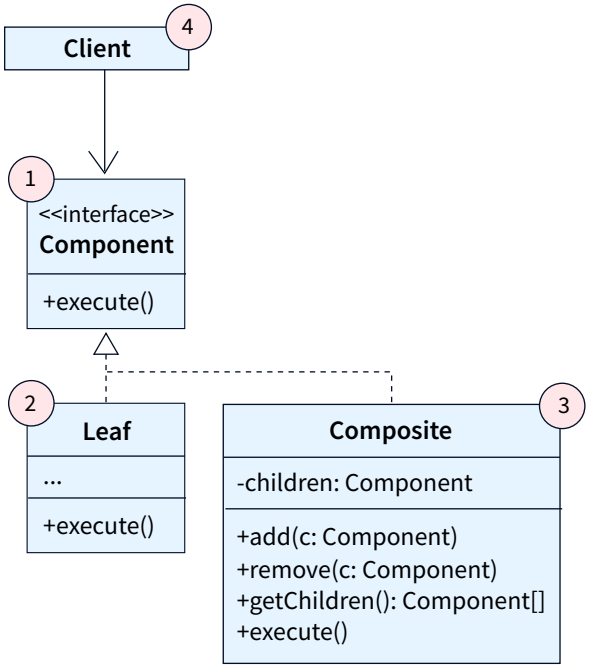
The Composite pattern creates tree-like structures by combining objects, allowing uniform manipulation of object collections regardless of hierarchy. It enhances code extensibility and modularity when working with complex object arrangements.

Analogy

Consider a scenario where you have two types of objects: Files and Folders. A Folder can contain multiple Files as well as smaller Folders. These smaller Folders can also hold Files or even smaller Folders, forming a hierarchical structure.



Structure



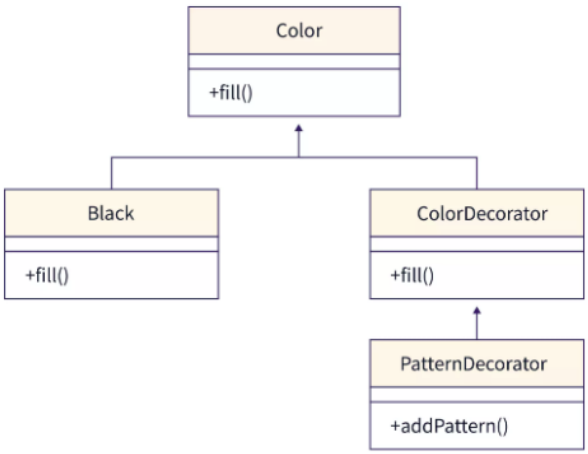
Decorator

The Decorator design pattern enables the addition of new functionalities to objects by wrapping them in special wrapper objects. This approach allows for dynamic behavior extension without impacting the behavior of other objects in the same class.

Analogy

In the decorator design pattern, a pizza servesas the original class while various toppings represent added functionalities. Customers can customize their pizza by adding toppings without altering the base structure.

Structure

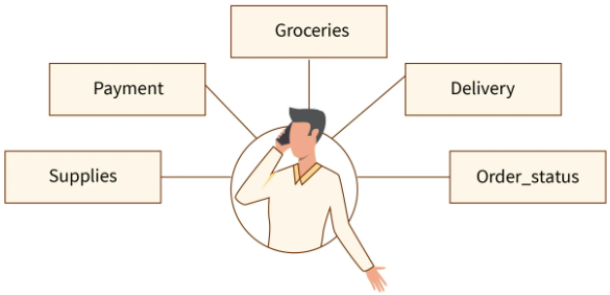


Facade

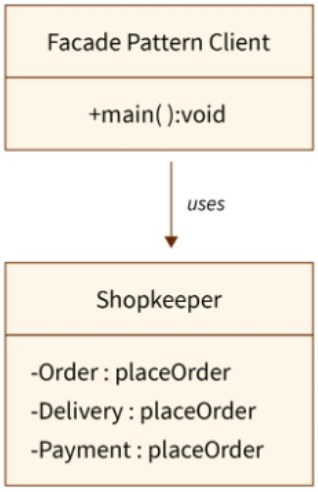
The Facade pattern provides a simple interface to interact with a complex system, shielding clients from underlying complexities and dependencies. It acts as a bridge, offering easy access to desired functionalities without revealing internal workings.

Analogy

In the Facade design pattern, when placing an order over the phone, the operator acts as a simplified interface to access various services offered by the shop. The facade layer assigns tasks to the appropriate subsystem based on user input.



Structure

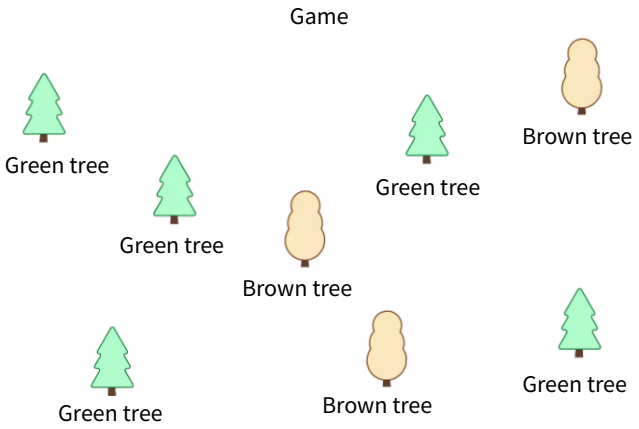


Flyweight

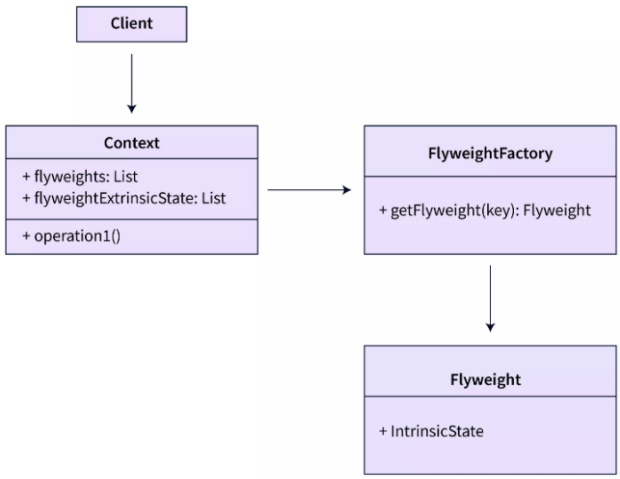
The Flyweight design pattern optimizes memory usage by sharing common state among multiple objects instead of duplicating it, allowing for efficient utilization of available resources when creating numerous objects.

Analogy

In the open-world game, the Flyweight designpattern minimizes memory usage by storing shared states (color and height) within the Flyweight object and unshared states (position) externally. This optimization eliminates redundant storage of duplicate values for efficient memory utilization.



Structure



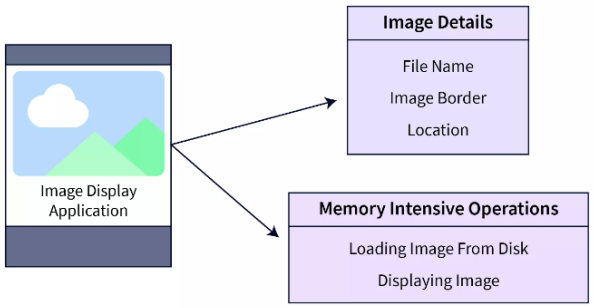
Facade

The Proxy pattern acts as a substitute for a real object, serving as an intermediary between the client and the actual object. It provides control over actual object-access, allows for additional functionality, and enables client access restrictions.

Analogy

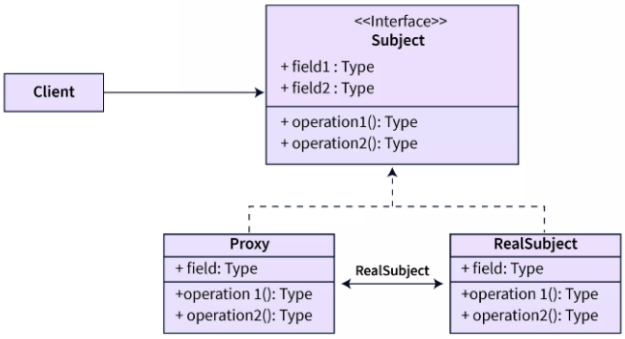
For an Instagram-like app on limited-resource smartphones, we design an Image interface with operations like border determination, rendering, and screen display. This interface ensures efficient image loading and display functionality in the application.

Application for Resource-Bound Device



Structure

Proxy Design Pattern



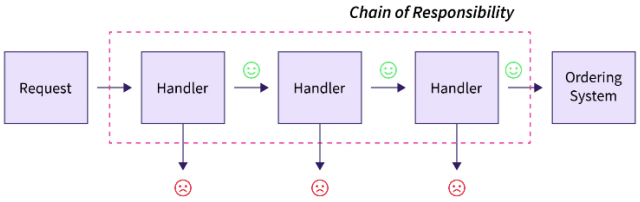
Behavioral Patterns

Chain of Responsibility

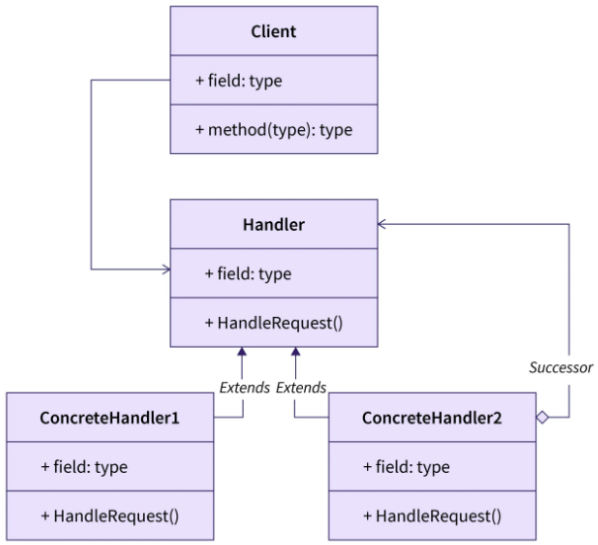
Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Analogy

The ATM Machine exemplifies the Chain of Responsibility pattern: it handles the responsibility chain, from card insertion to cash withdrawal, delegating tasks to appropriate handlers. Requests are processed systematically, achieving the desired result.



Structure

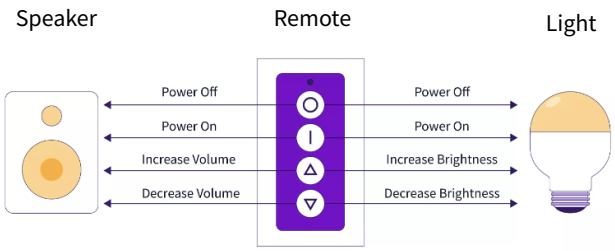


Command

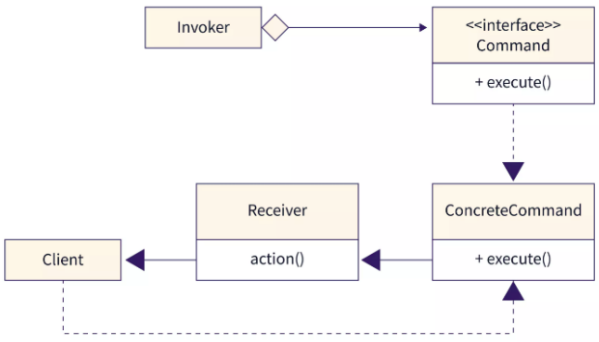
The Command pattern transforms requests or operations into self-contained objects, encapsulating information such as the method and its arguments. This abstraction enables method parameterization, queued request execution, and support for undoable operations.

Analogy

The Command design pattern is illustrated by a universal remote with configurable On, Off, Up, and Down buttons. These buttons can be assigned actions based on the device, like adjusting brightness or volume, implemented through command objects.



Structure



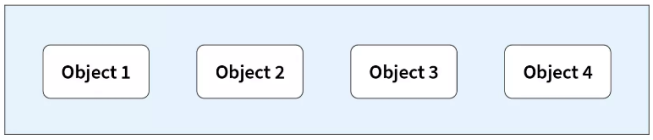
Iterator

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

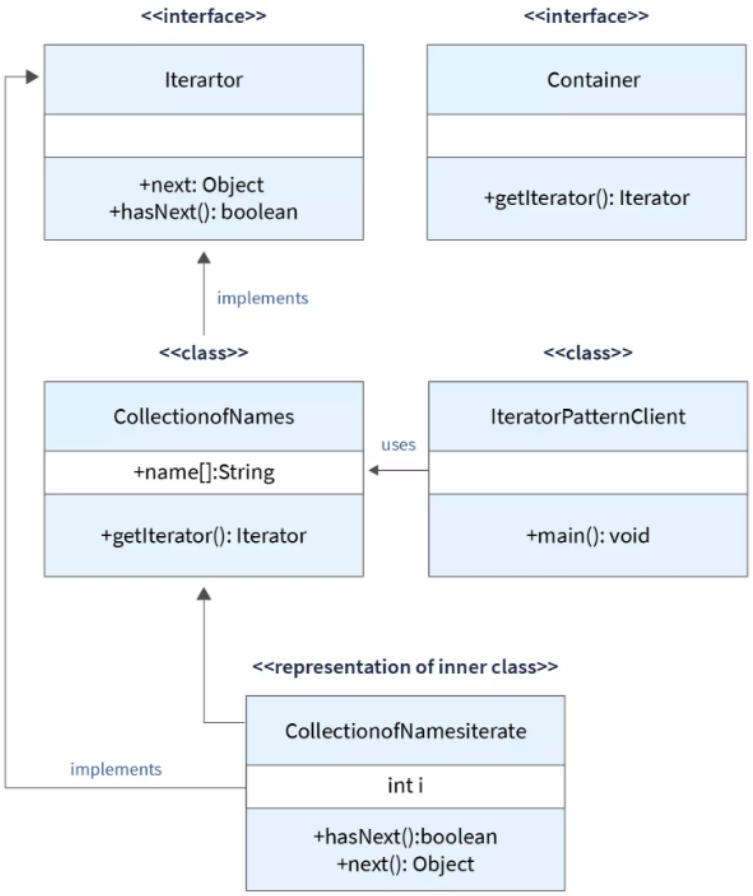
Analogy

In an application that tracks a collection of alerts, the Iterator design pattern enables iteration through the alerts without exposing the underlying structure. This pattern ensures seamless and controlled traversal of the collection.

Collections(List,ArrayList,Array,etc)



Structure



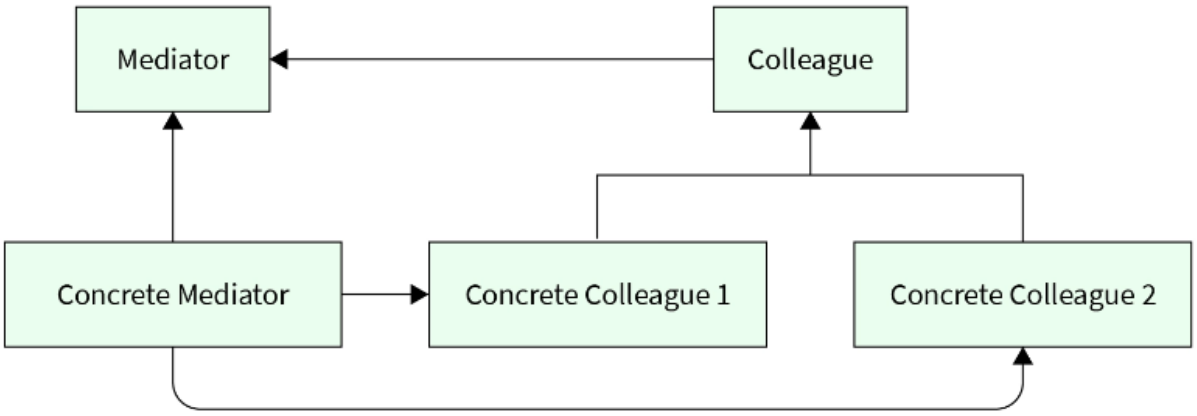
Mediator

The Mediator pattern reduces dependencies and promotes organized communication between objects. It introduces a mediator object to restrict direct communication and enforce interactions through the mediator.

Analogy

In a restaurant kitchen, chefs rely on an expediter who acts as a central communicator. The expediter relays orders, ensuring smooth coordination. Without this, chefs face potential confusion and delays.

Structure

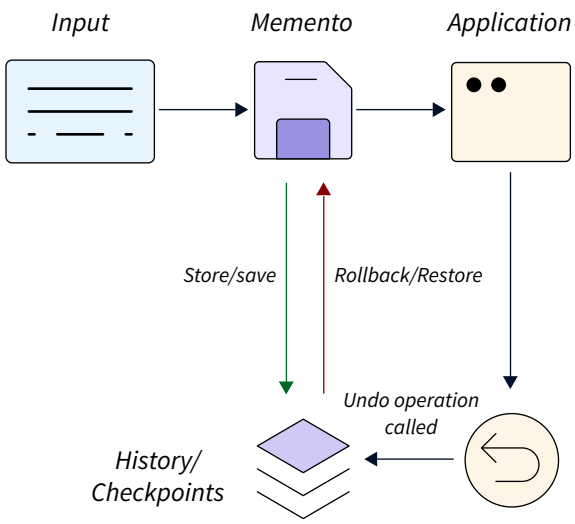


Memento

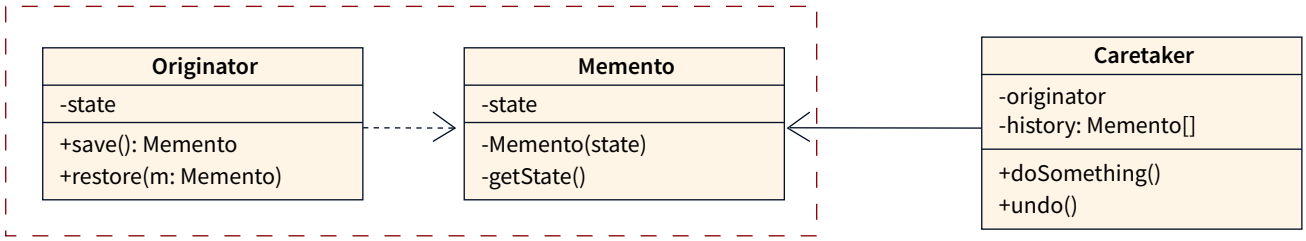
Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

Analogy

The Memento pattern delegates state snapshot creation to the originator object. Snapshots are stored in mementos, accessible only to the originator. Other objects have limited access, obtaining metadata but not the original state.



Structure

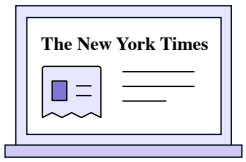


Observer

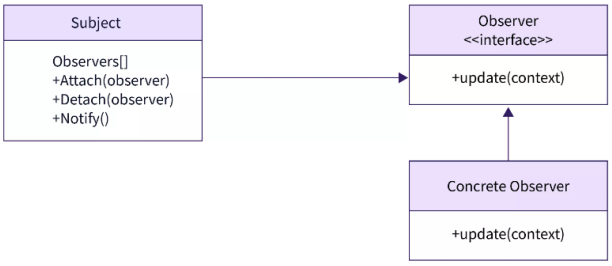
Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Analogy

Subscribing to a newspaper or magazine ensures direct delivery to your mailbox, eliminating the need to check availability at a store. Publishers send new issues promptly or in advance for convenience.



Structure



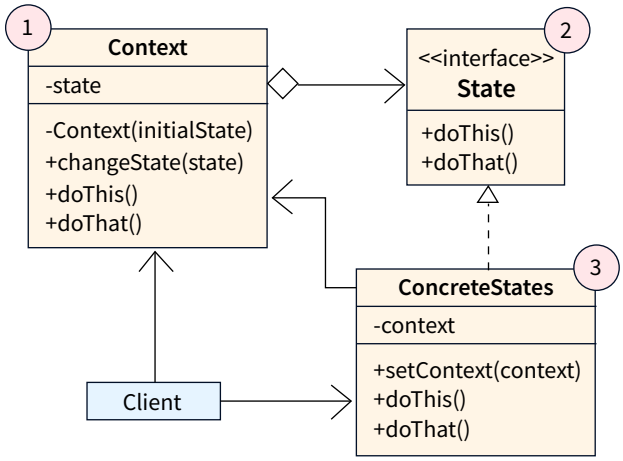
State

The State pattern allows an object to change its behavior based on internal state changes, resembling finite state machines. It avoids excessive conditional statements and provides flexibility in altering an object's state.

Analogy

In a video game, the behavior of buttons and controls changes depending on the game state. Such as during gameplay, pressing buttons triggers various in-game actions, when the game is paused, pressing any button brings up the pause menu and so on.

Structure

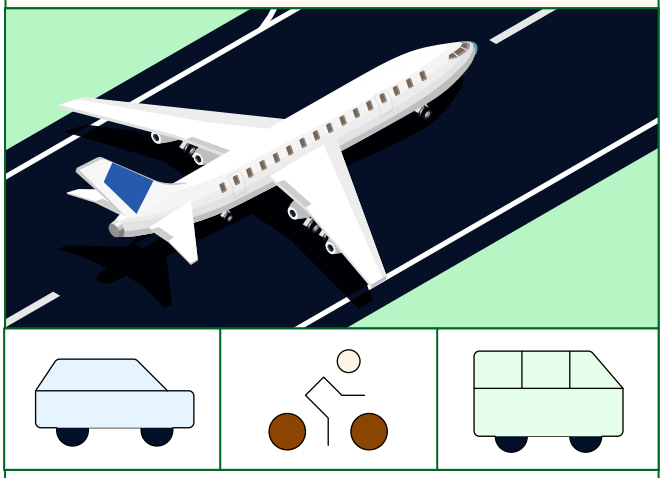


Strategy

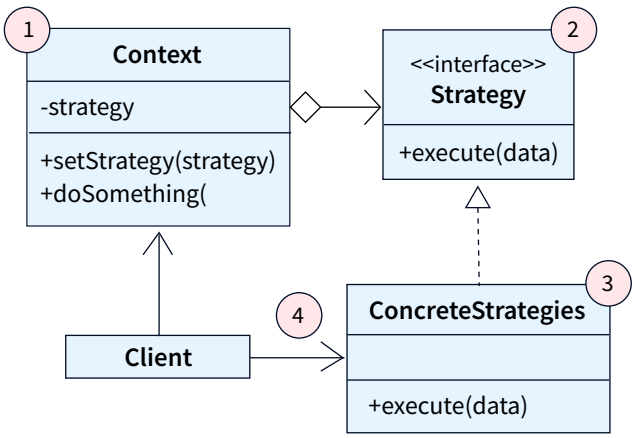
Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Analogy

When going to the airport, transportation strategies such as bus, cab, or bicycle offer options based on factors like budget and time constraints, enabling you to choose the most suitable approach.



Structure

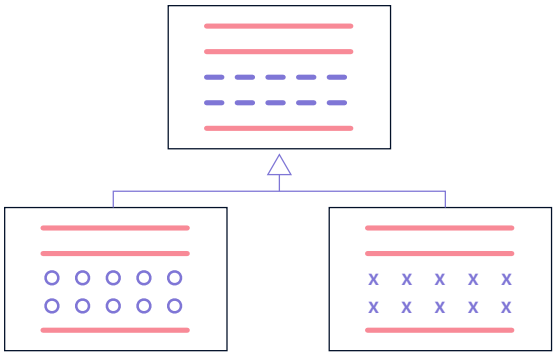


Template Method

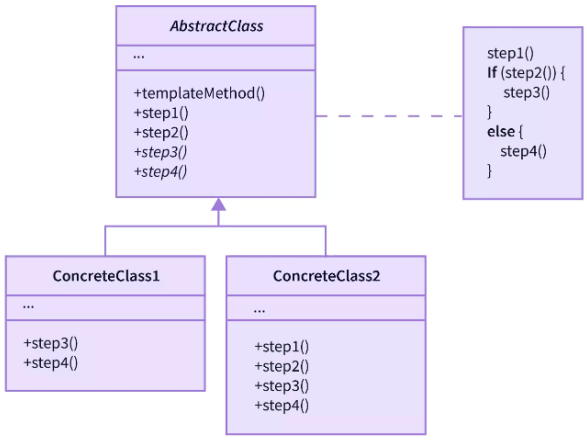
The Template Method pattern defines the structure of an algorithm in a superclass, allowing subclasses to override specific steps without changing the overall structure. It provides a skeleton of functions, enabling child classes to implement method details while the parent class maintains the algorithm's framework and sequencing.

Analogy

Frameworks extensively utilize the Template Method Design Pattern, executing immutable domain components and providing placeholders for user modifications. The framework becomes central, while client modifications are secondary.

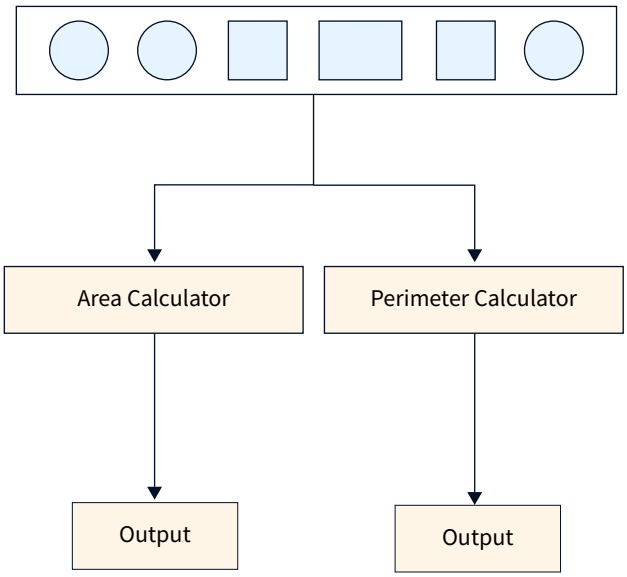


Structure



Visitor

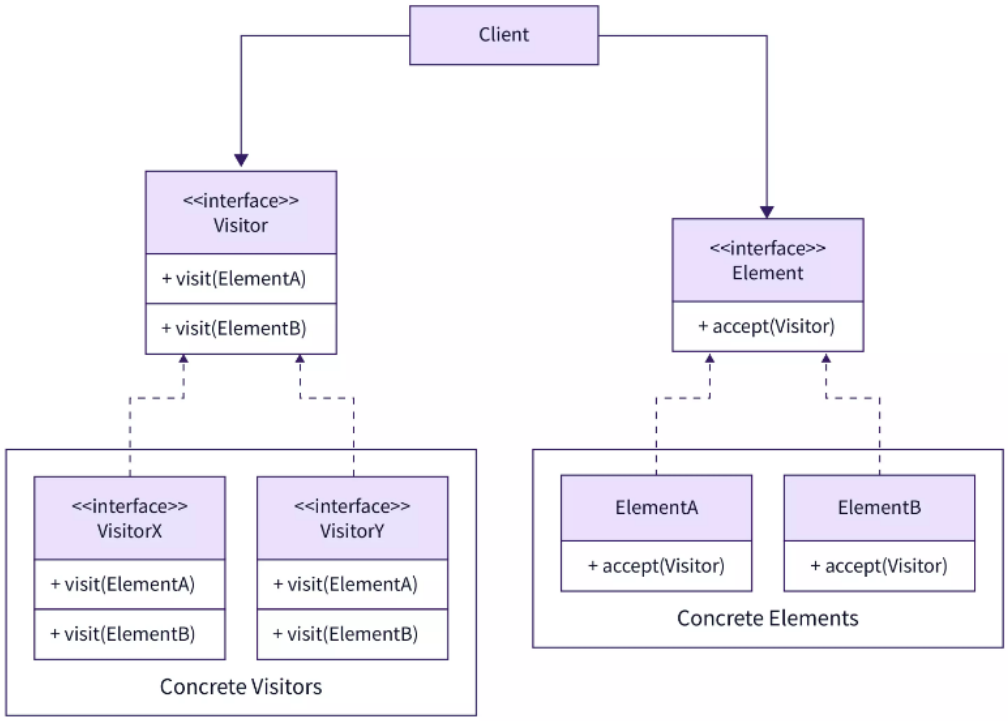
The Visitor pattern separates algorithms from the objects they operate on. It uses visitor classes to handle specific operations on objects. This decoupling enhances flexibility and extensibility, enabling different operations on objects without modifying their structure.



Analogy

The Visitor design pattern decouples algorithms from shape objects, allowing easy addition of new algorithms without modifying the objects. This promotes extensibility and adheres to the Open/Closed principle.

Structure



VI. Design Pattern Principles

- **SOLID principles:** Guide for object-oriented design and programming.
- **DRY (Don't Repeat Yourself) principle:** Avoid repetition of code.
- **KISS (Keep It Simple, Stupid) principle:** Simplicity should be a key goal.
- **YAGNI (You Aren't Gonna Need It) principle:** Don't add functionality until needed.

VII. Choosing the Right Design Pattern

- Understand the problem context thoroughly.
- Match problem characteristics with design patterns.
- Analyze the trade-offs and benefits of each pattern.
- Check pattern compatibility with the existing system.

VIII. Design Pattern Best Practices

- Don't overuse design patterns - keep it simple.
- Adapt patterns to fit your needs; they are not strict rules.
- Aim for simplicity and maintainability in your code.
- Continuously refactor and improve your code.

SCALER TOPICS

Unlock your potential in software development with
FREE COURSES from **SCALER TOPICS!**

Register now and take the first step towards your future Success!

C++ for Beginners

5.9k enrolled Free

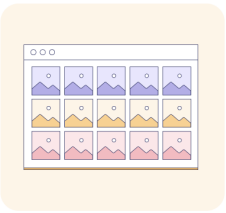
Java for Beginners

6.8k enrolled Free

That's not it. Explore 20+ Courses by clicking below

Explore Other Courses

Practice **CHALLENGES**
and become 1% better everyday



CIFAR-10 Image Classification Using PyTorch

Article

No. Of Questions : 3

Go to Challenge >

How to Build a Snake Game in JavaScript?

Article

No. Of Questions : 3

Go to Challenge >

Explore Other Challenges