**Adapter Pattern :**

Acts as a connector between two incompatible interfaces

or Converts the interface of a class into another interface that a client wants

**Advantage :**

1) It allows two or more previously incompatible objects to interact

2) It allows re-useability of existing functionality.

3) Adapter compared to Class Adapter is loose coupling of client and Adaptee

**Disadvantage :**

1) One of the drawbacks of using the adapter pattern is that it introduces additional overhead and complexity to your code.

2) The adapter class acts as an intermediary between the two interfaces, which means that it has to perform extra operations to convert the data, handle exceptions, and delegate calls. This can increase the memory usage, execution time, and maintenance cost of your code.

3) It creates coupling and dependency between the two interfaces. The adapter class depends on both the original interface and the adapted interface, which means that any changes in either of them can affect the adapter class and the rest of the code that uses it. For example, if the original interface adds a new method or modifies an existing one, the adapter class may need to be updated or extended to reflect the change. Similarly, if the adapted interface changes its specification or implementation, the adapter class may need to be modified or replaced to ensure compatibility.

4) It can make testing and debugging more difficult. The adapter class adds another layer of abstraction and indirection to your code, which can make it harder to trace the source of errors, identify the root cause of problems, or verify the correctness of the output. Moreover, the adapter class may have its own logic and behavior that need to be tested and validated separately from the original and adapted interfaces. This can increase the scope and complexity of your testing and debugging process.

**When to use :**

1) When an object needs to utilize an existing class with an incompatible interface.

2) When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.

**Example scenarios :**

1) The adapter design pattern is useful in situations where existing code needs to be reused with a new system or library that has a different interface. For example, if an application is designed to work with a specific database library, but a new library with a different interface is introduced, an adapter class can be used to translate between the two interfaces without having to modify the existing application code.

2) Another suitable scenario to use the adapter design pattern is when working with legacy code that cannot be easily modified. An adapter class can be created to adapt the legacy code to a new system without having to modify the existing codebase.

**class** AnalyticLibrary{

**public** **void** displayGraph(CustomLibraryObject customLibraryObject) {

System.***out***.println("Displayed graph based on the custom object");

}

}

**class** CustomLibraryObject{

**private** String jsonData;

CustomLibraryObject(String jsonData){

**this**.jsonData = jsonData;

}

**public** String getJsonData() {

**return** jsonData;

}

**public** **void** setJsonData(String jsonData) {

**this**.jsonData = jsonData;

}

}

**interface** DataVisulizer{

**public** **void** displayGraph(String jsonData);

}

**class** LibraryAdapter **implements** DataVisulizer{

AnalyticLibrary analyticLibrary; //adaptee

**public** LibraryAdapter(AnalyticLibrary analyticLibrary){

**this**.analyticLibrary = analyticLibrary;

}

**public** **void** displayGraph(String jsonData) {

CustomLibraryObject customLibraryObject = getCustomLibraryObject(jsonData);

analyticLibrary.displayGraph(customLibraryObject);

}

**private** CustomLibraryObject getCustomLibraryObject(String jsonData) {

//convert json data to library object

**return** **new** CustomLibraryObject(jsonData);

}

}

**public** **class** AdapterPattern {

**public** **static** **void** main(String[] args) {

String jsonData = "{}";

AnalyticLibrary library = **new** AnalyticLibrary();

LibraryAdapter adapter = **new** LibraryAdapter(library);

adapter.displayGraph(jsonData);

}

}

**Bridge Pattern :**

Bridge design pattern that uses OOP principles to decouple an abstraction from its implementation so that the two can vary independently.

**When to use :**

The Bridge design pattern can help you avoid the problems of inheritance, such as tight coupling, code duplication, and inflexibility. By separating the abstraction from the implementation, you can vary them independently and reduce the complexity of your code.

The Bridge design pattern can also improve the readability, maintainability, and testability of your web development framework. You can easily swap or add new implementations without modifying the existing abstractions.

**Advantages :**

1) The Bridge design pattern can be a great asset to your web development framework, as it increases the modularity and reusability of your code by separating the concerns of the abstraction and the implementation.

2) It also enhances the extensibility and scalability of your framework, allowing you to add new abstractions and implementations without breaking the existing ones. Moreover, it can improve the performance and efficiency of your framework by avoiding unnecessary inheritance and subclassing.

3) It facilitate the testing and debugging by isolating the abstraction and the implementation.

**Disadvantages :**

1) The Bridge design pattern can pose challenges to your web development framework, such as introducing complexity and overhead to your code, requiring more communication and coordination between the abstraction and the implementation, making your code less intuitive and harder to understand, and making it difficult to apply the pattern to existing frameworks that use inheritance extensively. These issues can affect the speed and reliability of your framework.

//Here we are creating a bridge between engine and vehicle

**interface** Engine{

**public** String reFill();

}

//Separating abstraction from implementation(we separate the engine from vehicle)

**abstract** **class** Vehical{

Engine engine;

**public** Vehical(Engine engine) {

**this**.engine = engine;

}

**abstract** **public** **void** reFill();

}

**class** Car **extends** Vehical{

Car(Engine engine){

**super**(engine);

}

**public** **void** reFill() {

System.***out***.println("Car "+engine.reFill());

}

}

**class** Bike **extends** Vehical{

**public** Bike(Engine engine) {

**super**(engine);

}

**public** **void** reFill() {

System.***out***.println("Bike "+engine.reFill());

}

}

**class** ElectricEngine **implements** Engine{

**public** String reFill() {

**return** "Charged with 100%";

}

}

**class** PetrolEngine **implements** Engine{

**public** String reFill() {

**return** "refilled with 5 liters";

}

}

**public** **class** BridgePattern {

**public** **static** **void** main(String[] args) {

//Engine engine = new ElectricEngine();

Engine engine = **new** PetrolEngine();

Vehical car = **new** Car(engine);

car.reFill();

}

}

**Composite Pattern :**

It allows you to compose objects into tree structures and then work with these structures as if they were individual objects.

It is useful when you want to represent part-whole hierarchies of objects and treat them uniformly.

**When to use :**

Using the Composite pattern makes sense only when the core model of your app can be represented as a tree.

**For example**: imagine that you have two types of objects: Folders and files. A Folder can contain several files as well as a number of sub folders. These sub folder can also hold some files or even sub folder, and so on.Say you decide to create an file system that uses these classes. file system could contain files with size, as well as sub folder may be or may not be with size and other files.... How would you determine the total size of such an file system?

**Advantages :**

1) You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.

2) Open/Closed Principle. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

**Disadvantages :**

1) It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you’d need to overgeneralize the component interface, making it harder to comprehend.

2) The delegation of operations adds additional run-time costs

//In below example we composed the object into tree structure(file system)

**interface** FileSystem{

**public** **double** getSize();

}

**class** File **implements** FileSystem {

**private** **final** **double** size;

**public** File(**double** size){

**this**.size = size;

}

**public** **double** getSize() {

**return** size;

}

}

**class** Folder **implements** FileSystem {

**private** **double** size = 0;

**private** List<FileSystem> childrens = **new** ArrayList<FileSystem>();

**public** Folder(**double** size){

**this**.size = size;

}

**public** **void** addChild(FileSystem file) {

childrens.add(file);

}

**public** **double** getSize() {

**double** size = 0;

**for** (FileSystem fileSystem : childrens) {

size += fileSystem.getSize();

}

**return** size + **this**.size;

}

}

**public** **class** CompositePattern {

**public** **static** **void** main(String[] args) {

Folder root = **new** Folder(0.5);

root.addChild(**new** File(5));

root.addChild(**new** File(2));

Folder subFolder1 = **new** Folder(0);

Folder subFolder2 = **new** Folder(0);

Folder subFolder3 = **new** Folder(0);

subFolder3.addChild(**new** File(8));

subFolder2.addChild(subFolder3);

subFolder2.addChild(**new** File(5));

subFolder1.addChild(subFolder2);

root.addChild(subFolder1);

System.***out***.println("Total size is:"+root.getSize());

}

}

**Decorator Pattern :**

A decorator pattern can be used to attach additional responsibilities to an object

either statically or dynamically. It is also known as Wrapper class

**Advantage :**

1) The decorator pattern also allows you to compose different decorators at runtime, creating flexible and dynamic combinations of behaviors.So better than static inheritance.

2) It enhances the extensibility of the object, because changes are made by coding new classes.

3) Which states that software entities should be open for extension but closed for modification.This means that you can extend the functionality of an existing object without changing its code or breaking its clients.For example,

Ex: you can wrap a file stream object with a compression decorator and an encryption decorator, depending on the user's preferences or security requirements.

4) It avoids the need for creating complex subclasses or inheritance hierarchies, which can lead to code duplication and maintenance issues.

**Disadvantage :**

1) it can introduce a lot of small objects in your system, which can increase the memory usage and the complexity of debugging.

2) You also need to make sure that the decorators are compatible with the original object's interface and behavior, otherwise you might encounter unexpected errors or inconsistencies.

3) The decorator pattern can make your code less readable and understandable, especially if you use too many decorators or nest them too deeply. You might also lose some type information or functionality of the original object, since the decorators only expose the common interface and hide the specific details.

**When to use :**

1) The decorator pattern is an ideal solution when you need to add or modify functionality of an object without altering its core behavior or structure, or when creating different variations of an object's behavior without creating a multitude of subclasses or inheritance hierarchies.

**Ex:** Examples of scenarios where the decorator pattern can be applied include adding logging, caching, or validation to an existing service or component; adding encryption, compression, or encoding to an existing data stream or file; and adding graphical effects, borders, or scrollbars to a user interface component.

**interface** Pizza {

**public** **void** prepare();

}

**class** BasePizza **implements** Pizza {

**public** BasePizza() {

}

**public** **void** prepare() {

System.***out***.println("Base pizza prepared");

}

}

**abstract** **class** PizzaDecorator **implements** Pizza {

**private** Pizza pizza;

**public** PizzaDecorator(Pizza pizza) {

**this**.pizza = pizza;

}

**public** **void** prepare() {

pizza.prepare();

}

}

**class** PepperoniPizza **extends** PizzaDecorator {

PepperoniPizza(Pizza pizza) {

**super**(pizza);

}

**public** **void** prepare() {

**super**.prepare();

System.***out***.println("Adding pepperoni on pizza");

}

}

**class** CapsicumPizza **extends** PizzaDecorator {

CapsicumPizza(Pizza pizza) {

**super**(pizza);

}

**public** **void** prepare() {

**super**.prepare();

System.***out***.println("Adding capsicum on pizza");

}

}

**class** OnionPizza **extends** PizzaDecorator {

OnionPizza(Pizza pizza) {

**super**(pizza);

}

**public** **void** prepare() {

**super**.prepare();

System.***out***.println("Adding onion on pizza");

}

}

**public** **class** DecoratorPattern {

**public** **static** **void** main(String[] args) {

Pizza pizza = **new** OnionPizza(**new** PepperoniPizza(**new** BasePizza()));

pizza.prepare();

}

}

**Façade Pattern :**

Facade is a structural design pattern that provides a simplified interface to a library a framework or any other complex set of classes

**Advantage :**

1) We can reduce the third party complexity by providing interface

2) Besides a much simpler interface, there’s one more benefit of using this design pattern.

It decouples a client implementation from the complex subsystem.

**Disadvantages :**

1) The facade pattern doesn’t force us to unwanted tradeoffs, because it only adds additional layers of abstraction.

2) Sometimes the pattern can be overused in simple scenarios, which will lead to redundant implementations.

**When to uses :**

1) When you want to provide simple interface to a complex sub-system.

2) When several dependencies exist between clients and the implementation classes of an abstraction.

**Best practice :**

1) You should identify the subsystems that have a high degree of complexity, diversity, or variability, and that can benefit from being encapsulated behind a facade.

2) You should design the facade to provide a simple, consistent, and coherent interface that meets the needs and expectations of the clients.

3) You should ensure that the facade does not expose or depend on any internal details or implementation of the subsystems, and that it does not add any unnecessary functionality or logic.

4)You should document and communicate the purpose, scope, and limitations of the facade and the subsystems, and make them clear and transparent to the clients.

**interface** MobileShop{

**public** **void** getModel();

**public** **void** getPrice();

}

**class** Iphone **implements** MobileShop {

**public** **void** getModel() {

System.***out***.println("Iphone 14");

}

**public** **void** getPrice() {

System.***out***.println("70000");

}

}

**class** Samsung **implements** MobileShop {

**public** **void** getModel() {

System.***out***.println("Galaxy tab");

}

**public** **void** getPrice() {

System.***out***.println("45000");

}

}

**class** Oneplus **implements** MobileShop {

**public** **void** getModel() {

System.***out***.println("Oneplas AZ");

}

**public** **void** getPrice() {

System.***out***.println("35000");

}

}

//Facade class

**class** ShopKeeper {

**private** Iphone iphone;

**private** Samsung samsung;

**private** Oneplus oneplus;

**public** ShopKeeper() {

**this**.iphone = **new** Iphone();

**this**.samsung = **new** Samsung();

**this**.oneplus = **new** Oneplus();

}

**public** **void** iphoneSale() {

iphone.getModel();

iphone.getPrice();

}

**public** **void** samsungSale() {

samsung.getModel();

samsung.getPrice();

}

**public** **void** oneplusSale() {

oneplus.getModel();

oneplus.getPrice();

}

}

**public** **class** FacadePattern {

**public** **static** **void** main(String[] args) {

ShopKeeper shopKeeper = **new** ShopKeeper();

shopKeeper.iphoneSale();

shopKeeper.samsungSale();

shopKeeper.oneplusSale();

}

}

**Flyweight Pattern:**

The flyweight pattern is based on the idea of reusing existing objects instead of creating new ones whenever possible. The objects that can be shared are called flyweights. The objects that use the flyweights are called contexts, and they contain the extrinsic state that is specific and variable for each instance.

The flyweight pattern separates the intrinsic and extrinsic state of the objects, and provides a factory that manages the creation and access of the flyweights.

So that flyweight Design Pattern reduces the objects creation, decreases memory footprint and increase the performance.

**Advantages :**

The main benefit of the flyweight pattern is that it can improve the system performance by reducing the memory usage and the number of objects that need to be created, transferred, and processed.

Ex: If you are designing a text editor that needs to display a large document with different fonts and styles, you can use the flyweight pattern to create a single object for each character with its intrinsic state (such as ASCII code and font size), and use the contexts to store the extrinsic state (such as position and color). This way, you can avoid creating thousands of objects for each character, and save memory and rendering time.

**Disavdvantages :**

1) It can increase the complexity of the system design, as you need to separate the intrinsic and extrinsic state of the objects, and implement a factory that manages the flyweights.

2) It can introduce some overhead in accessing and updating the extrinsic state of the contexts, as you need to pass it as a parameter or store it in an external data structure. Moreover, the flyweight pattern is not suitable for every system, as it depends on the nature and frequency of the objects and their state changes.

**When to use :**

To implement the flyweight pattern, you need to identify the objects that can be shared as flyweights and extract their intrinsic state. Then, create a flyweight class that encapsulates the intrinsic state and defines the common behavior of the flyweights. Additionally, create a flyweight factory that creates and manages the flyweights, and ensures that only one instance of each flyweight exists. Moreover, identify the objects that use the flyweights as contexts, and store their extrinsic state separately.Finally, modify the contexts to use the flyweight factory to access and manipulate the flyweights.

**Example :**

let's consider a system that needs to display a map with thousands of trees of different types and locations. Without the flyweight pattern, you would need to create an object for each tree with its type and location, which would consume a lot of memory and rendering time. With the flyweight pattern, you can create a flyweight class for the tree type, which contains the intrinsic state such as the name, shape, and texture of the tree. Then, you can create a flyweight factory that creates and caches the flyweight objects for each tree type.Finally, you can create a context class for the tree location, which contains the extrinsic state such as the coordinates and altitude of the tree. The context class uses the flyweight factory to get

the flyweight object for the tree type, and renders the tree accordingly.

**interface** VehicleI{

**public** **void** start();

**public** **void** stop();

**public** Color getColor();

}

**class** Color{

**private** String name;

**public** Color(String name) {

**this**.name = name;

}

**public** String getColorName() {

**return** name;

}

}

**class** EngineI {

**public** **void** start() {

System.***out***.println("Start the Engine");

}

**public** **void** stop() {

System.***out***.println("Stop the Engine");

}

}

**class** VehicalFactory{

**private** **static** Map<Color, VehicleI> *vehiclesCache* = **new** HashMap<>();

**public** **static** VehicleI createVehicle(Color color) {

VehicleI newVehicle = *vehiclesCache*.computeIfAbsent(color, newColor -> {

EngineI newEngine = **new** EngineI();

**return** **new** Truck(newEngine, newColor);

});

**return** newVehicle;

}

}

**class** Truck **implements** VehicleI{

**private** EngineI engine;

**private** Color color;

**public** Truck(EngineI engine,Color color) {

**this**.engine = engine;

**this**.color = color;

}

**public** **void** start() {

engine.start();

}

**public** **void** stop() {

engine.stop();

}

**public** Color getColor() {

**return** **new** Color(color.getColorName());

}

}

**public** **class** FlyweightPattern {

**public** **static** **void** main(String[] args) {

VehicalFactory factory = **new** VehicalFactory();

Color color = **new** Color("Blue");

VehicleI newVehicle = factory.*createVehicle*(color);

System.***out***.println(newVehicle.getColor().getColorName());

newVehicle.start();

Color color2 = **new** Color("Blue");

VehicleI newVehicle2 = factory.*createVehicle*(color);

//Both are same object because blue vehicle already created and stored in cache

System.***out***.println(newVehicle);

System.***out***.println(newVehicle2);

}

}

**Proxy Pattern:**

The Proxy design pattern provides a placeholder for another object in order to control access to it. The main purpose of this pattern is to add an additional level of interface to the target object, allowing us to perform operations before or after the target object is accessed, without changing its interface.

**The Proxy pattern solves several problems :**

1) Controlling access to an object: The Proxy pattern provides a way to control access to an object by encapsulating it and providing a surrogate object to the client. This allows us to restrict or grant access to the object based on specific conditions.

2) Reducing resource usage: The Proxy pattern can be used to reduce the resource usage of an application by creating lightweight proxy objects that consume fewer resources than the actual objects they represent. This is useful when we have a large number of expensive objects that we want to limit the number of.

3) Enhancing functionality: The Proxy pattern allows us to enhance the functionality of an object by adding additional features such as caching, logging, or lazy initialization without modifying its original implementation.

**When to use :**

1) Proxy pattern can be useful in cases where sessions are provided to an application, by allowing the use of a proxy session in place of a real session.

**For example**: consider a web application that allows users to login and access their personal account information. When a user logs in, a session is created that contains information about the user, such as their name, email address, and preferences. This session is stored on the server and is accessed by the application whenever the user interacts with the system.

In this scenario, we can use a Proxy pattern to provide a proxy session object that can be used in place of the real session object. The proxy session object would contain a reference to the real session object, and would be responsible for controlling access to it.

2) One use case for a proxy session object would be to limit the amount of access that a user has to their account information.

**For example**: if a user logs in from a public computer or a device that they don't own, the application might want to restrict access to certain parts of their account information, such as their billing information or their purchase history.

The proxy session object could be designed to provide this functionality by checking the user's IP address or device type, and only allowing access to certain parts of their account information based on those factors. The proxy session object could also be designed to log out the user automatically after a certain period of inactivity, or to limit the number of login attempts that are allowed. Using a proxy session object in this way allows the application to provide a more secure and flexible user experience, while still maintaining control over the user's access to their account information.

**Advatages :**

1) Controlled access to the target object: The Proxy allows us to control access to the target object, by adding additional functionality before or after the target object is accessed.

2) Reduced memory usage: The Proxy can delay the creation of the target object until it is actually needed, reducing memory usage in situations where the target object is expensive to create.

3) Improved performance: The Proxy can cache the results of expensive operations, improving performance in situations where the same operation is called repeatedly.

**Disadvantages :**

1) **Additional complexity**: Introducing a proxy layer adds an additional layer of complexity to the system. This additional complexity can be a burden on the system, especially if the proxy object is handling multiple tasks.

2) **Performance overhead**: Proxy design pattern can add additional overheads to the system in terms of performance. The extra processing required by the proxy can slow down the system.

3) **Maintenance overhead**: The proxy design pattern can add to the maintenance overhead of the system. Any changes made to the underlying real object must also be reflected in the proxy, which can be a challenging task.

4) **Limited functionality**: Proxies can be limited in terms of the functionality they offer compared to the real object. This can be a problem if the proxy does not support all of the features required by the system.

5) **Security risks**: The use of a proxy can introduce additional security risks, as it adds another point of access to the system. If not designed and implemented properly, a proxy can be exploited by malicious users.

**interface** Session {

**public** **boolean** login(String username,String password);

**public** **void** logout();

**public** **void** sendData(String data) **throws** Exception;

}

**class** RealSession **implements** Session {

**public** **boolean** login(String username,String password) {

System.***out***.println("Logging in with username: "+username +" and password: "+ password);

// Do actual login work here

**return** **true**;

}

**public** **void** logout() {

System.***out***.println("Logging out");

}

**public** **void** sendData(String data) **throws** Exception {

System.***out***.println("Sending data: " + data);

}

}

**class** ProxySession **implements** Session {

**private** Session realSession;

**private** **boolean** isLoggedIn = **false**;

**public** ProxySession(Session realSession) {

**this**.realSession = realSession;

}

**public** **boolean** login(String username,String password) {

**if** (realSession.login(username, password)) {

isLoggedIn = **true**;

**return** **true**;

}

**return** **false**;

}

**public** **void** logout() {

isLoggedIn = **false**;

realSession.logout();

}

**public** **void** sendData(String data) **throws** Exception {

**if** (isLoggedIn) {

realSession.sendData(data);

} **else** {

**throw** **new** Exception("Cannot send data without logging in first.");

}

}

}

**class** WebApp {

**private** Session session;

**public** WebApp(Session session) {

**this**.session = session;

}

**public** **boolean** login(String username,String password) {

**return** session.login(username, password);

}

**public** **void** logout() {

session.logout();

}

**public** **void** sendData(String data) **throws** Exception {

session.sendData(data);

}

}

**public** **class** ProxyPattern {

**public** **static** **void** main(String[] args) {

Session realSession = **new** RealSession();

Session proxySession = **new** ProxySession(realSession);

**try** {

//With session

WebApp webApp = **new** WebApp(realSession);

webApp.login("user123", "password123");

webApp.sendData("Hello world!");

webApp.logout();

//With proxy session

WebApp webAppWithProxy = **new** WebApp(proxySession);

webAppWithProxy.login("user456", "password456");

webAppWithProxy.sendData("Hello proxy!");

webAppWithProxy.logout();

}**catch** (Exception ex) {

System.***out***.println("Login failed "+ ex);

}

}

}