**Design Pattern**

It is re-useable solution to commonly occurring problem in software design.

Composition is better than inheritance.

Design pattern are divided into different types which are further divided into sub-parts:

1. Creational Design pattern - They are concerned with method of creating objects. They are preferred during the process of class-initiation.

a. Factory Pattern.

b. Abstract Factory Pattern

c. Singleton Pattern

d. Prototype Pattern

e. Builder Pattern

f. Object pool Pattern

2. Structural Design Pattern - They deal with composition of classes and objects which form larger structures. They simplify the structure by identifying the relationship b/w classes and objects.

a. Adapter Pattern

b. Bridge Pattern

c. Composite Pattern

d. Decorator Pattern

e. Facade Pattern

f. Flyweight Pattern

g. Proxy Pattern

3. Behavioural Design Pattern - They are concerned with interaction and responsibility of objects. The interaction b/w the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.

a. Chain of Responsibility Pattern

b. Command Pattern

c. Interpreter Pattern

d. Iterator Pattern

e. Mediator Pattern

f. Memento Pattern

g. Observer Pattern

h. State Pattern

i. Strategy Pattern

j. Template Pattern

k. Visitor Pattern.

4. JEE Design Pattern - They are concerned with providing solutions to the Java-EE based applications and framework.

a. MVC Design Pattern

b. Dependency Injection Pattern

c. DAO Design Pattern

d. Business delegate Pattern

e. Intercepting Filter Pattern

f. Service Locator Pattern

g. Transfer Object Pattern

**Singleton Design Pattern**

It is design pattern that ensures a class has only 1 instance and provides global point of access to this instance. Key Points while defining a class as a singleton class.

1. Make Constructor Private to prevent instantiation from outside.
2. Write a static method that has the return type object of this singleton class. Here, the concept of Lazy initialization is used to write this static method.

Use case of design pattern.

1. Configuration Management: For example, if our application needs to read some configuration settings from a file or database, you can create a singleton class to read the settings and make them available to other parts of application.
2. Logging: If our application needs to log information to file or database, we can create singleton logging class that handles all the logging operations.
3. Database Connection: Instead of creating a new database connection each time our application needs to interact with the database, we can create a Singleton database connection class that manages the connection.
4. Caching: If our application needs to fetch data from web, we can create a Singleton caching class that caches the data to reduce the number of requests to the web service.

Difference between Normal class and Singleton Class

1. To instantiate normal class, we use java constructor. On the other hand, to instantiate a singleton class, we use getInstance method.

Example  of Lazy way

*public class Singleton {*

*private static Singleton instance;*

*private Singleton() {} // private constructor to prevent creation of instance outside this class.*

*public static synchronized Singleton getInstance() {  // synchronized ensures thread safety*

*if (instance == null) instance = new Singleton();*

*return instance;*

*}*

*}*

'getInstance() method is synchronized, which means only one thread can execute it at a time. This ensures multiple threads can't instantiate their own instances.

Also, the ‘instance' variable is static in nature which means it belongs to class and ‘getInstance() will return the same instance of the singleton class for every call.

Example of Eager Way

*public class Singleton{*

*private static Singleton instance = new Singleton();*

*private Singleton(){}*

*public static Singleton getInstance() {*

*return instance;*

*}*

*}*

Example of Singleton class java.lang.Runtime  
Singleton Design pattern can be breaken down using Reflection API, Serialization and deserialization & Clone Method.

**Observer Design Pattern**

It is a behavioural design pattern that allows an object, called the 'Subject', to notify other objects, called the 'Observers' when it's state changes. This pattern is used to establish one to many relationship between the subject and it's observer, such that when the subject's state changes, all its observer are notified and updated automatically.

This design pattern consists of following components:

1. **Subject**: It is the object that is being observed. It maintains list of observers and provides methods for adding and removing them.
2. **Observer**: It is an interface or abstract class that defines the methods that observer should implement to receive notifications from the subject.
3. **Concrete Subject**: A sub class of Subject that implements the specific behaviour of the Subject. It maintains its own state, which can change over time.
4. **Concrete Observer**: A sub class of Observer that implements the specific behaviour of the observer. It registers with the subject to receive notifications and implements the update method to handle the notifications.

The key advantage of observer design pattern is that it promotes loose coupling between the subject and its observers. The subject doesn't need to know anything about observers without affecting the subject's behavior.

Additionally, the pattern supports broadcasting notifications to all the observers simultaneously, which can be very useful in certain scenarios.

Use case of observer Design Pattern

1. Stock market monitoring: It can be used to keep track of stock prices and notify users when a particular stock reaches a certain price level. The Stock price can be subject, and the users can be observers.
2. Message Broadcasting: It can be used to broadcast message to multiple recipients. The sender will be subject and message recipient can be observer.
3. Traffic Monitoring: It can be used to monitor traffic flow and notify drivers of any road closures or traffic accidents. Traffic flow data can be the subject and the drivers can be observers.
4. Weather Monitoring: It can be used to keep track of whether conditions and notify users of any change in temperature, humidity, or precipitation.
5. Online auction system: It can be used to notify bidders of new bids or when an auction is about to end. The auction object can be the subject and bidders can be observers.

Do implement practical implementation.

**Decorator Design Pattern**

The decorator design pattern is a structural design pattern. It is based on the idea of wrapping a class with another class to enhance or modify its behaviour.

In this pattern, a decorator is a wrapper class that is used to add or modify functionality of an object at runtime.

Decorator design pattern involve screating :

1. **Component Interface**: It is the interface for the objects that will be decorated (wrapped).
2. **Concrete Component Class**: It is the class that implements the component interface.
3. **Abstract Decorator Class**: It is the abstract class that implements the component interface and contains a reference to an instance of the component.
4. **Concrete Decorator Class**: It is the class that extend the decorator abstract class and add additional functionality to the component.

*// Component interface*

*public interface Pizza {*

*public String getDescription();*

*public double getCost();*

*}*

*// Concrete component class*

*public class BasicPizza implements Pizza {*

*public String getDescription() { return "Pizza with tomato sauce and cheese";}*

*public double getCost() { return 10.00;}*

*}*

*// Decorator abstract class*

*public abstract class PizzaDecorator implements Pizza {*

*protected Pizza pizza;*

*public PizzaDecorator(Pizza pizza) { this.pizza = pizza;}*

*public String getDescription() { return pizza.getDescription();}*

*public double getCost() { return pizza.getCost(); }*

*}*

*// Concrete decorator class*

*public class Pepperoni extends PizzaDecorator {*

*public Pepperoni(Pizza pizza) { super(pizza);}*

*public String getDescription() { return pizza.getDescription() + ", with pepperoni";}*

*public double getCost() { return pizza.getCost() + 2.50;}*

*}*

*// Client code*

*Pizza basicPizza = new BasicPizza();*

*Pizza pepperoniPizza = new Pepperoni(basicPizza);*

*System.out.println(basicPizza.getDescription() + ", cost: $" + basicPizza.getCost());*

*System.out.println(pepperoniPizza.getDescription() + ", cost: $" + pepperoniPizza.getCost());*

In this example Pizza interface is the component interface, the Basic Pizza class is the concrete component class, the PizzaDecorator class is the decorator abstract class, and the Pepperoni class is the concrete decorator class.

The client code creates an instance of the Basic Pizza class and then creates a new instance of the Pepperoni pizza class, passing the Basic Pizza Instance as parameter. The Pepperoni class "decorates" the BasicPizza Object by adding pepperoni to it, and the resulting object is printed out with its description and cost. The same approach can be used to add other toppings or ingredients to the pizza, without changing the original pizza object or class.

Another Example

Suppose we are working on the notification service of a food delivery app. The service notifies customer about important events such as delivery departure and arrival time, order receipts etc.

The initial version of notifier class houses single send method which sends notifications via email.With additional functionality we want to add Whatsapp and sms notification. For doing that the first thing that comes to our mind is inheritance. So we extended the notifier class with new subclass, the *‘WhatsAppNotifier’* and ‘*SmsNotifier’.* The client is supposed to initiate the desired notifier class and use it while notifying the particular customer.

However the customer want to be notified via SMS and WhatApp at same time. With current design the solution will be to create a new class *‘WhatsAppAndSmsNotifier’ .*

But now with this approach if we add another notifier (which notifies via facebook), We might lead ourselves into ‘class explosion’. More classes like *FacebookNotifier, FacebookAndWhatsappNotifier, FacebookAndSmsNotifier, FacebookWhatsappAndSmsNotifier* have to be created. Thus by adding a single additional choice, the number of subclasses mutifolds.

The solution to such problem is Decorator Design Pattern.

If we apply Decorator design pattern to our case then we will keep the main component of notifier class intact,

public class Notifier {  
 public void send(String msg) {  
 System. out. println ( "Sending by Mail"+msg);  
 }  
}   
Also in the former example we used inheritance which lead to Class Explosion, In decorator design pattern we use Composition to avoid class explosion. Using composition we can substitute the objects with one another because all of them belongs to same interface. This allows us to change behaviour of object at run time.  
We create new classes for SMS, Whatsapp and Facebook Notification   
public interface INotifier{  
 public void send(String message);  
}

public class FacebookDecorator extends BaseNotifierDecorator {  
 public FacebookDecorator(INotifier wrapped) {super(wrapped) ;}  
 public void send(String msg) {   
 super. send(msg) ;  
 System. out. println( "Sending on Facebook”+msg);  
 }  
}

public class WhatsappDecorator extends BaseNotifierDecorator {  
 public WhatsappDecorator(INotifier wrapped) {super(wrapped) ;}  
 public void send(String msg) {   
 super. send(msg) ;  
 System. out. println( "Sending on WhatsApp”+msg);  
 }  
}

public class SMSDecorator extends BaseNotifierDecorator {  
 public SMSDecorator(INotifier wrapped) {super(wrapped) ;}  
 public void send(String msg) {   
 super. send(msg) ;  
 System. out. println( "Sending on SMS”+msg);  
 }  
}

Note that all the decorators are extending BaseNotifierDecorator

public abstract class BaseNotifierDecorator {  
 private final INotifier wrapped;  
 BaseNotifierDecorator(INotifier wrapped) {  
 this.wrapped = wrapped;  
 }  
 public void send(String msg) {  
 wrapped. send(msg) ;  
 }  
}

public static void main(String[] Args) {  
 INotfier notifier = new SMSNotifier(new WhatsAppNotifier(new Notifier()));  
 notifier.send(“Food is Ready”);  
}

For More information watch <https://www.youtube.com/watch?v=v6tpISNjHf8>

Example code to create run time composition of the object  
for (String channel : channels) {  
 switch (channel) {  
 case "sms": notifier = new SMSNotifier(notifier);  
 break;  
 case "whatsapp": notifier = new WhatsAppNotifier(notifier);  
 break;

// Add more cases for other notification types

default:

System.out.println("Unknown channel: " + channel);

}

}

**Factory Design Pattern**

Factory Design Pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. In other words, subclasses are responsible to create the instance of the class.

The factory pattern involves defining a separate class or method, called the factory, that is responsible for creating objects. The factory class or method typically accepts a set of parameters that describe the object to be created, and then returns an instance of that object. By encapsulating the creation of objects in a separate class or method, the factory pattern makes it easier to modify or extend the types of objects that can be created, without having to modify the code that uses those objects.

*interface Drink {void prepare();}*

*class Coffee implements Drink {public void prepare() {System.out.println("Preparing coffee...");} }  
class Tea implements Drink {public void prepare() {System.out.println("Preparing tea...");} }  
class Soda implements Drink {public void prepare() {System.out.println("Preparing soda...");}}*

*class DrinkFactory {  
 public Drink createDrink(String type) {  
 if ( type.equals("coffee") ) { return new Coffee(); }   
 else if ( type.equals("tea") ) { return new Tea(); }   
 else if ( type.equals("soda") ) { return new Soda(); }   
 else { throw new IllegalArgumentException("Invalid drink type");}  
 }  
}*In this example, we have an interface Drink and three classes Coffee, Tea, and Soda that implement the Drink interface and define their own prepare() method. We also have a DrinkFactory class, which has a createDrink() method that takes in the type of drink to create and returns an instance of the corresponding drink class.

To use this implementation, you would simply create a DrinkFactory object and call the createDrink() method with the desired drink type:

*DrinkFactory factory = new DrinkFactory();  
Drink drink1 = factory.createDrink("coffee");  
drink1.prepare();  
Drink drink2 = factory.createDrink("tea");  
drink2.prepare();  
Drink drink3 = factory.createDrink("soda");  
drink3.prepare();*

This way, you can easily add new types of drinks by creating a new class that implements the Drink interface, without having to modify the existing code.

**Adapter Design Pattern**

Adapter design pattern is a design pattern that allows incompatible interfaces to work together. It is used when two classes or components have different interfaces, and they need to work together in the same system. The adapter acts as a bridge between the two interfaces, allowing them to communicate with each other.

For example: consider a USB to Ethernet adapter. We need this when we have an Ethernet interface on one end and USB on the other. Since they are incompatible with each other, we use an adapter that converts one to other.

In the adapter pattern, an adapter class is created that implements the target interface, which is the interface that the client expects to use. The adapter class also contains an instance of the adaptee class, which is the class that has the incompatible interface. The adapter class then maps the methods of the target interface to the methods of the adaptee interface, allowing the client to use the adaptee class through the adapter.

Here's an example of how the adapter pattern can be used:  
Suppose you have an application that reads data from a CSV file and displays it on a web page. You are using a third-party CSV library that has an interface for reading data, but your web page expects a different format of data. You can create an adapter that maps the interface of the CSV library to the interface expected by your web page.

*// target interface  
interface DataProvider {void displayData();}*

*// adaptee interface  
interface CSVReader {void readCSV();}*

*// adaptee class  
class ThirdPartyCSVReader implements CSVReader {  
 public void readCSV() {System.out.println("Reading data from CSV file");}  
}*

*// adapter class  
class CSVAdapter implements DataProvider {  
 private CSVReader csvReader;  
 public CSVAdapter(CSVReader csvReader) {this.csvReader = csvReader;}  
 public void displayData() {  
 csvReader.readCSV();  
 System.out.println("Formatting data for web page");  
 }  
}  
// client code  
public class Application {  
 public static void main(String[] args) {  
 CSVReader csvReader = new ThirdPartyCSVReader();  
 DataProvider dataProvider = new CSVAdapter(csvReader);  
 dataProvider.displayData();  
 }  
}*

In this example, we have a target interface DataProvider, which represents the interface that the client code expects to use. We also have an adaptee interface CSVReader, which represents the interface of the third-party CSV library that we are using. The adaptee class ThirdPartyCSVReader implements the CSVReader interface and has a method for reading data from a CSV file.

We then create an adapter class CSVAdapter that implements the DataProvider interface and takes an instance of CSVReader in its constructor. The displayData() method of the adapter first calls the readCSV() method of the CSVReader instance, and then formats the data for the web page.

Finally, in the client code, we create an instance of the ThirdPartyCSVReader class and pass it to the CSVAdapter constructor to create a DataProvider instance. We can then call the displayData() method on the DataProvider instance to display the data on the web page.

Use Case :

1. **Legacy System Integration**: In a software system, you may have an old legacy system that uses an outdated interface, and you want to integrate it with a new system that uses a modern interface. You can use the adapter pattern to create a wrapper around the old system's interface and make it compatible with the new system's interface.
2. **Third-Party Library Integration**: In an application, you may need to use a third-party library that has a different interface than the one you are currently using. You can use the adapter pattern to create a wrapper around the third-party library's interface and make it compatible with your application's interface.
3. **Device Drivers**: In an operating system, you may have device drivers that communicate with hardware devices using a specific interface, and you want to make them work with a different interface. You can use the adapter pattern to create a wrapper around the device driver's interface and make it compatible with the operating system's interface.
4. **Internationalization**: In an application that supports multiple languages, you may need to translate text between different character sets or encodings. You can use the adapter pattern to create a wrapper around the text conversion libraries and make them compatible with your application's encoding.

**Strategy Design Pattern**

Strategy design pattern allows us to define a family of algorithms, encapsulate each one as an object, and make them interchangeable at runtime. This pattern allows us to select an algorithm at runtime without tightly coupling the algorithm with the client code. It is also known as Policy.  
In the strategy pattern, we have a context object that contains a reference to a strategy object. The strategy object encapsulates the algorithm that needs to be performed. The context object is responsible for setting the strategy object and executing the algorithm.   
The strategy pattern is useful in situations where you need to switch between multiple algorithms dynamically, or when you want to provide multiple implementation options for a single behavior.

Here's an example of how the strategy pattern can be used:

Suppose you have an application that needs to calculate the total cost of a shopping cart. The total cost can be calculated using different algorithms depending on the type of products in the cart. For example, if the cart contains books, the cost can be calculated by applying a discount to the total price. If the cart contains electronics, the cost can be calculated by adding a sales tax to the total price. You can use the strategy pattern to encapsulate each algorithm as a separate strategy object.

*// strategy interface  
interface PricingStrategy { double calculatePrice(double price); }*

*// concrete strategies  
class BookDiscount implements PricingStrategy {  
 public double calculatePrice(double price) { return price \* 0.8;}  
}*

*class ElectronicsSalesTax implements PricingStrategy {  
 public double calculatePrice(double price) {return price \* 1.1;}  
}*

*// context object  
class ShoppingCart {  
 private PricingStrategy pricingStrategy;  
 public void setPricingStrategy(PricingStrategy pricingStrategy) {this.pricingStrategy = pricingStrategy;}  
 public double calculateTotal(double price) {return pricingStrategy.calculatePrice(price);}  
}*

*// client code  
public class Application {  
 public static void main(String[] args) {  
 ShoppingCart cart = new ShoppingCart();  
 PricingStrategy bookDiscount = new BookDiscount();  
 PricingStrategy electronicsSalesTax = new ElectronicsSalesTax();  
 cart.setPricingStrategy(bookDiscount);  
 double total = cart.calculateTotal(100);  
 System.out.println("Total cost with book discount: " + total);  
 cart.setPricingStrategy(electronicsSalesTax);  
 total = cart.calculateTotal(100);  
 System.out.println("Total cost with electronics sales tax: " + total);  
 }  
}*

In this example, we have a PricingStrategy interface that defines the algorithm for calculating the total price. We also have two concrete strategy classes, BookDiscount and ElectronicsSalesTax, that implement the PricingStrategy interface and provide the algorithms for calculating the total cost.

We then have a ShoppingCart class, which is the context object that contains a reference to a PricingStrategy object. The ShoppingCart class is responsible for setting the PricingStrategy object and executing the algorithm for calculating the total price.

In the client code, we create an instance of the ShoppingCart class and set the PricingStrategy object to the BookDiscount object. We then call the calculateTotal() method of the ShoppingCart instance to calculate the total cost with the book discount. We repeat the same steps with the ElectronicsSalesTax object to calculate the total cost with the electronics sales tax.

Use Case

1. **Sorting Algorithms**: In a sorting application, you may have different algorithms that can be used to sort a list of items. You can use the strategy pattern to encapsulate each sorting algorithm as a separate strategy object, and then allow the user to switch between different sorting algorithms at runtime.
2. **Payment Gateway Integration**: In an e-commerce application, you may need to integrate with multiple payment gateways, each with its own set of APIs and protocols. You can use the strategy pattern to encapsulate each payment gateway integration as a separate strategy object, and then allow the user to switch between different payment gateways at runtime.
3. **Image Processing**: In an image processing application, you may have different algorithms that can be used to perform operations such as scaling, cropping, and filtering. You can use the strategy pattern to encapsulate each image processing algorithm as a separate strategy object, and then allow the user to switch between different image processing algorithms at runtime.
4. **Encryption and Decryption**: In an application that involves encryption and decryption, you may have different algorithms that can be used to encrypt and decrypt data. You can use the strategy pattern to encapsulate each encryption and decryption algorithm as a separate strategy object, and then allow the user to switch between different encryption and decryption algorithms at runtime.

**Façade Design Pattern**

The facade design pattern involves creating a single interface that encapsulates and simplifies the interactions with the underlying system. This interface is implemented as a separate class that sits between the client and the underlying subsystems. The facade class provides a simple set of methods that the client can use to interact with the system. Behind the scenes, the facade class delegates the requests to the appropriate classes in the subsystems.

The main benefits of using the facade pattern are:

1. Simplifying the usage of complex subsystems: By providing a simplified interface, the facade pattern makes it easier for clients to use complex subsystems. Clients don't need to understand the details of how the subsystems work, they can simply use the methods provided by the facade.
2. Reducing coupling: The facade pattern decouples the client from the subsystems. Clients only need to interact with the facade, which shields them from changes in the underlying subsystems.
3. Promoting modularity: The facade pattern promotes modularity by encapsulating the subsystems behind a single interface. This makes it easier to modify, extend, or replace the subsystems without affecting the clients.

For Example

Let's say we have a complex subsystem of classes that provide various functionalities such as calculating taxes, generating invoices, and sending emails. We can create a facade class to simplify the interactions with this subsystem.

*public class TaxInvoiceSystemFacade {  
 private TaxCalculator taxCalculator;  
 private InvoiceGenerator invoiceGenerator;  
 private EmailSender emailSender;  
 public TaxInvoiceSystemFacade() {  
 this.taxCalculator = new TaxCalculator();  
 this.invoiceGenerator = new InvoiceGenerator();  
 this.emailSender = new EmailSender();  
 }*

*public void generateInvoiceAndSendEmail(String customerName, String customerEmail, double amount) {  
 double tax = taxCalculator.calculateTax(amount);  
 double totalAmount = amount + tax;  
 String invoice = invoiceGenerator.generateInvoice(customerName, totalAmount);  
 emailSender.sendEmail(customerEmail, invoice);  
 }  
}*

In this example, we have a facade class TaxInvoiceSystemFacade that encapsulates the functionality of the subsystem. The facade has references to the TaxCalculator, InvoiceGenerator, and EmailSender classes, which are part of the subsystem.

The generateInvoiceAndSendEmail method is a simplified interface that the client can use to generate an invoice and send it via email. Behind the scenes, the facade class delegates the requests to the appropriate classes in the subsystem.

The client can use the facade class as follows:

*TaxInvoiceSystemFacade facade = new TaxInvoiceSystemFacade();   
facade.generateInvoiceAndSendEmail("John Doe", "johndoe@example.com", 1000.0);*

This code generates an invoice for John Doe with a total amount of $1100 (assuming a tax rate of 10%) and sends it to the email address provided.

Using the facade design pattern in this way simplifies the interactions with the subsystem, hides the underlying complexity from clients, and reduces coupling between the subsystem and the client.

Use Case

1. **Complex API or library**: In many software development projects, there are complex APIs or libraries that have a large number of classes and objects. In such cases, the facade design pattern can be used to simplify the usage of the API or library by providing a simple interface to the clients.
2. **Legacy code integration**: When integrating legacy code into a new system, the legacy code might be complex and difficult to use. The facade design pattern can be used to provide a simplified interface to the legacy code, making it easier to integrate it into the new system.
3. **Multiple systems integration**: In distributed systems or service-oriented architectures, multiple systems might need to be integrated to provide a unified service. The facade design pattern can be used to provide a simple interface to the clients, hiding the complexity of the interactions between the systems.
4. **User interface design**: In user interface design, the facade design pattern can be used to simplify the interactions between the user interface and the underlying business logic. The facade class can provide a simplified interface to the business logic, making it easier to develop and maintain the user interface.
5. **Testing**: The facade design pattern can be used to simplify the testing of complex systems by providing a simple interface to the test cases. The facade class can be used to encapsulate the complex logic of the system, making it easier to test different scenarios and edge cases.

**Builder Design Pattern**

The Builder pattern is useful when we need to create an object that requires a complex initialization process, or when there are multiple ways to construct the object. Rather than exposing the complex initialization process to the client, we can use a Builder class to encapsulate the process and provide a simplified interface for the client to use.

The Builder pattern involves the following components:

1. **Product**: the object being constructed.
2. **Builder**: an interface or abstract class that defines the steps involved in constructing the object.
3. **ConcreteBuilder**: a class that implements the Builder interface and provides an implementation for each of the steps involved in constructing the object.
4. **Director**: a class that uses the Builder interface to construct the object.

*public class Car {  
 private String make, model;  
 private int year, horsepower  
 private Car() {}  
 public String getMake() { return make; }  
 public String getModel() { return model; }  
 public int getYear() { return year; }  
 public int getHorsepower() { return horsepower; }  
 public static class Builder {  
 private String make, model;  
 private int year, horsepower;  
 public Builder(String make, String model, int year) {  
 this.make = make;  
 this.model = model;  
 this.year = year;  
 }*

*public Builder withHorsepower(int horsepower) {  
 this.horsepower = horsepower;  
 return this;  
 }*

*public Car build() {  
 Car car = new Car();  
 car.make = this.make;  
 car.model = this.model;  
 car.year = this.year;  
 car.horsepower = this.horsepower;  
 return car;}}}*

In this example, we have a Car class with four attributes: make, model, year, and horsepower. The Car class has a private constructor to prevent clients from directly creating Car objects.

We also have a Builder class that is used to create Car objects. The Builder class has methods to set the make, model, year, and horsepower attributes of the Car. The Builder class also has a build method that creates and returns the fully constructed Car object.

Clients can use the builder to create Car objects like this:

*Car car = new Car.Builder("Ford", "Mustang", 2022).withHorsepower(450).build();*

This code creates a Car object with the make "Ford", model "Mustang", year 2022, and horsepower 450.

Using the builder pattern in this way provides several benefits:

1. **Flexibility**: The builder pattern allows you to create different variations of an object by setting different attributes. This is more flexible than having a large number of constructors for different variations.
2. **Encapsulation**: The builder pattern encapsulates the construction process and object representation within the builder class, making it easier to modify and maintain the object creation process.
3. **Validation**: The builder pattern allows you to enforce certain constraints or validations on the object being created, ensuring that it is valid and consistent.

**Proxy Design Pattern**

Proxy means an object representing another object. A Proxy Pattern provides the control for accessing the original object. In other words, Proxy object acts as an intermediary between a client object and a real object, allowing the proxy to control access to the real object and provide additional functionalities or behavior. A real world example can be a cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash, and provides a means of accessing that cash when required.  
The proxy pattern typically consists of three main components:

1. **Real Subject**: This is the actual object that the proxy represents and provides a substitute for. It defines the interface that the proxy and the client both implement, allowing them to interact with the real subject.
2. **Proxy**: This is the substitute object that stands in for the real subject. It implements the same interface as the real subject, allowing clients to interact with it in the same way. The proxy may perform additional tasks before or after forwarding requests to the real subject, such as logging, caching, or authentication.
3. **Client**: This is the object that interacts with the proxy and requests operations on the real subject. The client is usually unaware of the presence of the proxy and interacts with it in the same way as it would with the real subject.

Use Case

1. **Remote Proxy**: Suppose you have a distributed system where objects are spread across different servers or networks. In this case, you can use a remote proxy to encapsulate the communication between the client and the remote objects. The remote proxy acts as a local representative of the remote object and handles tasks such as marshaling and unmarshaling data, managing network connections, and handling remote method invocations. This can help improve performance by reducing the amount of data transferred over the network and hiding the complexity of remote communication from the client.
2. **Virtual Proxy**: Imagine you have a large collection of images or other expensive resources that are not loaded into memory unless they are actually needed. In this case, you can use a virtual proxy to represent these resources and delay their loading until they are requested by the client. The virtual proxy acts as a placeholder for the real object and loads it into memory only when necessary, helping to optimize resource usage and improve performance.
3. **Security Proxy**: Consider a scenario where you need to restrict access to certain resources or operations based on user permissions or authentication status. In this case, you can use a security proxy to enforce access control rules. The security proxy intercepts requests from the client, verifies the user's credentials or permissions, and only forwards the request to the real object if the access is allowed. This can help enhance security and prevent unauthorized access to sensitive resources.
4. **Caching Proxy**: Suppose you have a resource-intensive operation that is expensive to compute or fetch, such as fetching data from a remote database or performing complex calculations. In this case, you can use a caching proxy to store the results of previous requests and serve them directly to subsequent requests without re-computing or re-fetching the data. The caching proxy can significantly improve performance by reducing the amount of time and resources needed to fetch or compute the same data repeatedly.

*// Subject interface  
interface Image { void display();}*

*// RealSubject class  
class RealImage implements Image {  
 private String fileName;  
 public RealImage(String fileName) {  
 this.fileName = fileName;  
 loadFromDisk();  
 }  
  
 @Override  
 public void display() { System.out.println("Displaying image: " + fileName); }  
 private void loadFromDisk() { System.out.println("Loading image from disk: " + fileName); }  
}*

*// Proxy class  
class ProxyImage implements Image {  
 private String fileName;  
 private RealImage realImage;  
 public ProxyImage(String fileName) { this.fileName = fileName; }*

*@Override  
 public void display() {  
 if (realImage == null) { realImage = new RealImage(fileName);}  
 realImage.display();  
 }  
}*

*// Client class  
public class Client {  
 public static void main(String[] args) {  
 // Client uses the ProxyImage to display images  
 Image image1 = new ProxyImage("image1.jpg");  
 Image image2 = new ProxyImage("image2.jpg");  
 // Images are loaded and displayed only when needed  
 image1.display();  
 image2.display();  
 }  
}*

In this example, we have an interface Image that defines the common methods for displaying images. The RealImage class is the real object that loads and displays the actual image from disk. The ProxyImage class is the proxy object that acts as a placeholder for the real image and only loads and displays the image when needed. The Client class uses the ProxyImage objects to display images, and the images are loaded and displayed only when requested by the client, thus providing a level of abstraction and control over the image loading process.

**State Design Pattern**

State design pattern allows an object to change its behavior based on its internal state. It is used to model objects that can exist in different states and have different behaviors based on their current state. The state pattern involves three main components:

1. **Context**: This is the object whose behavior is influenced by its state. It maintains a reference to the current state object and delegates requests to it. The context object can also define methods to allow clients to change its state.
2. **State**: This is an interface or an abstract class that defines the common interface for all concrete states. It typically contains methods that represent the actions or behaviors that can be performed in a particular state.
3. **Concrete State**: These are the different classes that implement the State interface or inherit from the State abstract class. Each concrete state represents a specific state of the context and defines the behavior or actions associated with that state. The concrete state objects are responsible for changing the state of the context object.

**Use Case**

1. **Traffic Light System**: A traffic light can be in several states such as "red", "green", and "yellow". The behavior of the traffic light, such as changing colors and managing timers, depends on its current state. The state design pattern can be used to model the traffic light system with separate concrete state classes for each state, and the context object being the traffic light itself, which changes its state based on the rules of a traffic light system.
2. **Order Processing System**: An order in an e-commerce system can have different states, such as "pending", "processing", "shipped", and "delivered". The behavior of the order, such as processing payments, updating inventory, and sending notifications, depends on its current state. The state design pattern can be used to model the order processing system with separate concrete state classes for each state, and the context object being the order, which changes its state based on events or actions performed on the order.
3. **Game Character Behavior**: In a video game, a game character can have different states, such as "idle", "walking", "running", "attacking", and "defending". The behavior of the game character, such as movement, animations, and interactions, depends on its current state. The state design pattern can be used to model the behavior of the game character with separate concrete state classes for each state, and the context object being the game character, which changes its state based on player inputs or game events.
4. **Document Editing System**: In a document editing system, a document can have different states, such as "editing", "reviewing", and "published". The behavior of the document, such as editing capabilities, review process, and publishing actions, depends on its current state. The state design pattern can be used to model the document editing system with separate concrete state classes for each state, and the context object being the document, which changes its state based on user actions or workflow rules.

**Command Design Pattern**

The Command design pattern is a behavioral design pattern that encapsulates a request as an object, allowing clients to parameterize and queue requests, as well as support undoable operations. It separates the requester of an operation from the object that performs the operation, providing greater flexibility in managing requests.

The key components of the Command design pattern are:

1. Command: This is the interface or abstract class that defines the common methods for executing an operation. It typically includes an execute() method that encapsulates the operation to be performed.
2. Concrete Command: These are the concrete implementations of the Command interface or class that encapsulate specific operations to be performed. They hold a reference to the Receiver object that will perform the actual operation when the command is executed.
3. Receiver: This is the object that performs the actual operation requested by the Command. It knows how to perform the operation and carries out the requested action.
4. Invoker: This is the object that requests the execution of a command. It holds a reference to a Command object and can invoke the execute() method on the Command to perform the operation. The Invoker can also maintain a history of executed commands and support undo/redo functionality.
5. Client: This is the object that creates and configures the Command objects, as well as sets up the relationship between the Invoker, Command, and Receiver objects. It typically initiates the execution of commands by invoking the execute() method on the Command object.

Use Case

1. **Implementing Undo/Redo functionality**: The Command pattern allows you to encapsulate operations as objects, which can be easily stored and managed. This makes it possible to implement undo and redo functionality by keeping a history of executed commands and their parameters, and simply reversing or re-executing the commands as needed.
2. **Implementing a Queue of Operations**: The Command pattern allows you to queue operations as objects, which can be executed in a specific order. This is useful in situations where you need to manage a sequence of operations, such as in job scheduling, task management, or workflow systems.
3. **Implementing a Callback System**: The Command pattern can be used to implement callback systems, where you can encapsulate a request as an object and pass it to another object to be executed at a later time or in a different context. This is commonly used in event handling, asynchronous programming, and callback-based APIs.
4. **Implementing a Remote Control or Macro System**: The Command pattern can be used to implement a remote control system, where you can encapsulate different commands as objects and bind them to buttons or keys. It can also be used to implement a macro system, where you can record a sequence of operations as commands and replay them later.
5. **Decoupling Requesters from Executors**: The Command pattern allows you to decouple the object that requests an operation from the object that performs the operation. This provides flexibility in managing requests and allows for extensibility and maintainability in the code, as requesters and executors can be changed independently without affecting each other.
6. **Implementing Transactional Operations**: The Command pattern can be used to implement transactional operations, where a sequence of operations need to be executed as a single atomic transaction. If any of the operations fail, the entire transaction can be rolled back by simply undoing the commands that were executed.

**Abstract Factory Design Pattern**

Abstract Factory design pattern allows you to create objects of different classes that share a common interface or belong to a common family of objects, without specifying the concrete class names directly in your code. This provides a way to encapsulate object creation and create families of objects that are interchangeable, extensible, and maintainable.

The key components of the Abstract Factory design pattern are:

1. **Abstract Factory**: This is the interface that defines the methods for creating objects. It provides a common interface for creating different types of objects, but does not specify the concrete classes.
2. **Concrete Factories**: These are the concrete classes that implement the Abstract Factory interface. They are responsible for creating objects of specific concrete classes that belong to a family or group of related objects.
3. **Abstract Products**: These are the abstract classes or interfaces that define the common interface for the products (objects) created by the Abstract Factory. These products belong to a family or group of related objects.
4. **Concrete Products**: These are the concrete classes that implement the Abstract Products. They are the actual objects created by the Concrete Factories.

**Use Case**

1. **UI Component Creation**: When building a user interface (UI) for an application, you may need to create different types of UI components, such as buttons, text boxes, checkboxes, etc., with different styles or themes. The Abstract Factory pattern can be used to create families of related UI components, where each family represents a different style or theme, and the Abstract Factory provides methods for creating these components without specifying their concrete classes directly.
2. **Database Abstraction**: When working with databases in an application, you may need to support different types of databases, such as MySQL, Oracle, SQLite, etc., with different implementations. The Abstract Factory pattern can be used to create families of database-related objects, such as database connections, queries, and transactions, where each family represents a different type of database, and the Abstract Factory provides methods for creating these objects without specifying their concrete classes directly.
3. **Operating System Abstraction**: When developing a system or application that needs to work with different operating systems, such as Windows, macOS, Linux, etc., with different system-specific operations, the Abstract Factory pattern can be used to create families of related objects, such as system-specific file system operations, process management, and network communication, where each family represents a different operating system, and the Abstract Factory provides methods for creating these objects without specifying their concrete classes directly.
4. **Localization and Internationalization**: When developing an application that needs to support different languages, regions, or cultures, the Abstract Factory pattern can be used to create families of related objects, such as localized strings, date/time formats, number formats, etc., where each family represents a different language, region, or culture, and the Abstract Factory provides methods for creating these objects without specifying their concrete classes directly.
5. **Game Development**: In game development, you may need to create different types of game objects, such as characters, weapons, enemies, etc., with different behaviors or appearances based on the game level, game mode, or player preferences. The Abstract Factory pattern can be used to create families of related game objects, where each family represents a different type of game object, and the Abstract Factory provides methods for creating these objects without specifying their concrete classes directly.