# Ref :- <https://www.journaldev.com/1827/java-design-patterns-example-tutorial>

# Design Patterns

A design patterns are **well-proved solution** for solving the specific problem/task.

#### Creational Design Patterns

* [Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples)
* [Factory](https://www.journaldev.com/1392/factory-design-pattern-in-java)
* [Abstract Factory](https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java)
* [Builder](https://www.journaldev.com/1425/builder-design-pattern-in-java)
* [Prototype](https://www.journaldev.com/1440/prototype-design-pattern-in-java)

#### Structural Design Patterns

* [Adapter](https://www.journaldev.com/1487/adapter-design-pattern-java)
* [Composite](https://www.journaldev.com/1535/composite-design-pattern-in-java)
* [Proxy](https://www.journaldev.com/1572/proxy-design-pattern)
* [Flyweight](https://www.journaldev.com/1562/flyweight-design-pattern-java)
* [Facade](https://www.journaldev.com/1557/facade-design-pattern-in-java)
* [Bridge](https://www.journaldev.com/1491/bridge-design-pattern-java)
* [Decorator](https://www.journaldev.com/1540/decorator-design-pattern-in-java-example)

#### Behavioral Design Patterns

* [Template Method](https://www.journaldev.com/1763/template-method-design-pattern-in-java)
* [Mediator](https://www.journaldev.com/1730/mediator-design-pattern-in-java-example-tutorial)
* [Chain of Responsibility](https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java)
* [Observer](https://www.journaldev.com/1739/observer-design-pattern-in-java)
* [Strategy](https://www.journaldev.com/1754/strategy-design-pattern-in-java-example-tutorial)
* [Command](https://www.journaldev.com/1624/command-design-pattern)
* [State](https://www.journaldev.com/1751/state-design-pattern-java)
* [Visitor](https://www.journaldev.com/1769/visitor-design-pattern-java)
* [Interpreter](https://www.journaldev.com/1635/interpreter-design-pattern-java)
* [Iterator](https://www.journaldev.com/1716/iterator-design-pattern-java)
* [Memento](https://www.journaldev.com/1734/memento-design-pattern-java)

#### Miscellaneous Design Patterns

* [Dependency Injection](https://www.journaldev.com/2394/java-dependency-injection-design-pattern-example-tutorial)
* [Thread Safety in Java Singleton](https://www.journaldev.com/171/thread-safety-in-java-singleton-classes-with-example-code)

**-----------------------Creational Design Pattern ----------------**

**Factory design pattern** is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-class.

--------------------------------------------------------

If you are familiar with [**factory design pattern in java**](https://www.journaldev.com/1392/factory-design-pattern-in-java), you will notice that we have a single Factory class. This factory class returns different subclasses based on the input provided and factory class uses if-else or switch statement to achieve this.

In the **Abstract Factory pattern**, we get rid of if-else block and have a factory class for each sub-class. Then an Abstract Factory class that will return the sub-class based on the input factory class.

**--------------------------------------------------**

**Builder pattern** was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

There are three major issues with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

1. Too Many arguments to pass from client program to the Factory class that can be error prone because most of the time, the type of arguments are same and from client side its hard to maintain the order of the argument.
2. Some of the parameters might be optional but in [Factory pattern](https://www.journaldev.com/1392/factory-design-pattern-in-java), we are forced to send all the parameters and optional parameters need to send as NULL.
3. If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing.

We can solve the issues with large number of parameters by providing a constructor with required parameters and then different setter methods to set the optional parameters. The problem with this approach is that the Object state will be **inconsistent** until unless all the attributes are set explicitly.

Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.

**------------------------------------------------------------**

**Prototype design pattern** is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing.

Prototype pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. Prototype design pattern uses java cloning to copy the object.

Suppose we have an Object that loads data from database. Now we need to modify this data in our program multiple times, so it’s not a good idea to create the Object using new keyword and load all the data again from database.

The better approach would be to clone the existing object into a new object and then do the data manipulation.

**----------------------------------Structural------------------------------**

**Adapter**[**design pattern**](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) is one of the **structural design pattern** and it used so that two unrelated interfaces can work together. The object that joins these unrelated interface is called an **Adapter**.

One of the great real life example of Adapter design pattern is mobile charger. Mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket.

-------------------------------------------------------------------------

**Composite design pattern -** When we need to create a structure in a way that the objects in the structure has to be treated the same way, we can apply composite design pattern.

Lets understand it with a real life example – A diagram is a structure that consists of Objects such as Circle, Lines, Triangle etc. When we fill the drawing with color (say Red), the same color also gets applied to the Objects in the drawing. Here drawing is made up of different parts and they all have same operations.

**------------------------------------------------------------------**

**Proxy design pattern - Provide a surrogate or placeholder for another object to control access to it.**

Let’s say we have a class that can run some command on the system. Now if we are using it, its fine but if we want to give this program to a client application, it can have severe issues because client program can issue command to delete some system files or change some settings that you don’t want.

Here a proxy class can be created to provide controlled access of the program.

**------------------------------------------------------------------**

**Flyweight design pattern** is used when we need to create a lot of Objects of a class. Since every object consumes memory space that can be crucial for low memory devices, such as mobile devices or embedded systems, flyweight design pattern can be applied to reduce the load on memory by sharing objects.

**------------------------------------------------------------------**

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

Suppose we have an application with set of interfaces to use MySql/Oracle database and to generate different types of reports, such as HTML report, PDF report etc.

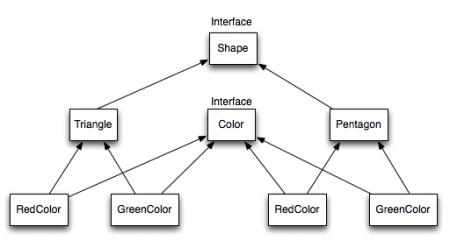
So we will have different set of interfaces to work with different types of database. Now a client application can use these interfaces to get the required database connection and generate reports.

But when the complexity increases or the interface behavior names are confusing, client application will find it difficult to manage it.

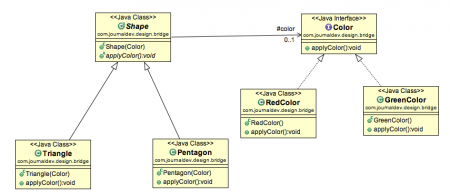
So we can apply Facade design pattern here and provide a [wrapper](https://www.journaldev.com/1002/wrapper-class-in-java) interface on top of the existing interface to help client application.

**-------------------------------------------------------------------------**

**Bridge**[**Design Pattern**](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) - When we have interface hierarchies in both interfaces as well as implementations, then **bridge design pattern** is used to decouple the interfaces from implementation and hiding the implementation details from the client programs.



**By applying Builder Design Pattern to simplify this,**



**We can see the transformation above.**

**--------------------------------------------------------------------------**

**Decorator**[**design pattern**](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior.

We use [inheritance](https://www.journaldev.com/644/inheritance-java-example) or composition to extend the behavior of an object but this is done at compile time and its applicable to all the instances of the class. We can’t add any new functionality of remove any existing behavior at runtime – this is when Decorator pattern comes into picture.

**--------------------------Behavioral Design Pattern---------------**

**Template method design pattern** defines the steps to execute an algorithm and it can provide default implementation that might be common for all or some of the subclasses.

Let’s understand this pattern with an example, suppose we want to provide an algorithm to build a house. The steps need to be performed to build a house are – building foundation, building pillars, building walls and windows. The important point is that the we can’t change the order of execution because we can’t build windows before building the foundation. So in this case we can create a template method that will use different methods to build the house.

Now building the foundation for a house is same for all type of houses, whether its a wooden house or a glass house. So we can provide base implementation for this, if subclasses want to override this method, they can but mostly it’s common for all the types of houses.

To make sure that subclasses don’t override the template method, we should make it final.

**----------------------------------------------------------------**

**Mediator design pattern** is used to provide a centralized communication medium between different objects in a system.

Mediator design pattern is very helpful in an enterprise application where multiple objects are interacting with each other. If the objects interact with each other directly, the system components are tightly-coupled with each other that makes higher maintainability cost and not hard to extend. Mediator pattern focuses on provide a mediator between objects for communication and help in implementing lose-coupling between objects.

Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights. Mediator works as a router between objects and it can have it’s own logic to provide way of communication.

---------------------------------------------------------------------

**Chain of responsibility** **pattern** is used to achieve loose coupling in software design where a request from client is passed to a chain of objects to process them. Then the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

For example, in [try-catch block](https://www.journaldev.com/592/java-try-with-resources) code. Here every catch block is kind of a processor to process that particular exception.

So when any exception occurs in the try block, its send to the first catch block to process. If the catch block is not able to process it, it forwards the request to next object in chain i.e next catch block. If even the last catch block is not able to process it, the exception is thrown outside of the chain to the calling program.

One of the great example of Chain of Responsibility pattern is **ATM Dispense machine**. The user enters the amount to be dispensed and the machine dispense amount in terms of defined currency bills such as 50$, 20$, 10$ etc.

If the user enters an amount that is not multiples of 10, it throws error. We will use Chain of Responsibility pattern to implement this solution.

----------------------------------------------------------------------

**Observer Pattern** is one of the **behavioral**[**design pattern**](https://www.journaldev.com/1827/java-design-patterns-example-tutorial). Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change. In observer pattern, the object that watch on the state of another object are called **Observer** and the object that is being watched is called **Subject**.

**Subject** contains a list of observers to notify of any change in it’s state, so it should provide methods using which observers can register and unregister themselves. Subject also contain a method to notify all the observers of any change and either it can send the update while notifying the observer or it can provide another method to get the update.

Observer should have a method to set the object to watch and another method that will be used by Subject to notify them of any updates.

Java provides inbuilt platform for implementing Observer pattern through *java.util.Observable* class and *java.util.Observer* interface. However it’s not widely used because the implementation is really simple and most of the times we don’t want to end up extending a class just for implementing Observer pattern as java doesn’t provide multiple inheritance in classes.

Model-View-Controller (MVC) frameworks also use Observer pattern where Model is the Subject and Views are observers that can register to get notified of any change to the model.

--------------------------------------------------------------------------

**Strategy**[**design pattern**](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) is one of the **behavioral design pattern**. Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.

Strategy pattern is also known as **Policy Pattern**. We define multiple algorithms and let client application pass the algorithm to be used as a parameter.

One of the best example of strategy pattern is Collections.sort() method that takes Comparator parameter. Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways.

--------------------------------------------------------------------------

In **Command design pattern**, the request is send to the invoker and invoker pass it to the encapsulated command object.

Command object passes the request to the appropriate method of Receiver to perform the specific action.

The client program create the receiver object and then attach it to the Command. Then it creates the invoker object and attach the command object to perform an action.

Now when client program executes the action, it’s processed based on the command and receiver object.

Let’s say we want to provide a File System utility with methods to open, write and close file. This file system utility should support multiple operating systems such as Windows and Unix.

To implement our File System utility, first of all we need to create the receiver classes that will actually do all the work.

Since we code in terms of [interface in java](https://www.journaldev.com/1601/interface-in-java), we can have FileSystemReceiver interface and it’s implementation classes for different operating system flavors such as Windows, Unix, Solaris etc.

--------------------------------------------------------------

**State design pattern** is used when an Object change its behavior based on its internal state.

If we have to change the behavior of an object based on its state, we can have a state variable in the Object. Then use **if-else** condition block to perform different actions based on the state. State design pattern is used to provide a systematic and loosely coupled way to achieve this through Context and State implementations.

Suppose we want to implement a TV Remote with a simple button to perform action. If the State is ON, it will turn on the TV and if state is OFF, it will turn off the TV.

----------------------------------------------------------------

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

For example, think of a Shopping cart where we can add different type of items (Elements). When we click on checkout button, it calculates the total amount to be paid. Now we can have the calculation logic in item classes or we can move out this logic to another class using visitor pattern.

-------------------------------------------------

**Interpreter design pattern** is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.

The best example of interpreter design pattern is java compiler that interprets the java source code into byte code that is understandable by [JVM](https://www.journaldev.com/546/difference-jdk-vs-jre-vs-jvm). Google Translator is also an example of interpreter pattern where the input can be in any language and we can get the output interpreted in another language.

---------------------------------------------------

**Iterator design pattern** is used to provide a standard way to traverse through a group of Objects. Iterator pattern is widely used in [Java Collection Framework](https://www.journaldev.com/1260/collections-in-java-tutorial). Iterator interface provides methods for traversing through a collection.

Iterator design pattern hides the actual implementation of traversal through the collection and client programs just use iterator methods.

Suppose we have a list of Radio channels and the client program want to traverse through them one by one or based on the type of channel. For example some client programs are only interested in English channels and want to process only them, they don’t want to process other types of channels.

So we can provide a collection of channels to the client and let them write the logic to traverse through the channels and decide whether to process them. But this solution has lots of issues such as client has to come up with the logic for traversal. We can’t make sure that client logic is correct. Furthermore if the number of client grows then it will become very hard to maintain.

Here we can use Iterator pattern and provide iteration based on type of channel. We should make sure that client program can access the list of channels only through the iterator.

---------------------------------------------------

**Memento design pattern** is used when we want to save the state of an object so that we can restore later on. Memento pattern is used to implement this in such a way that the saved state data of the object is not accessible outside of the object, this protects the integrity of saved state data.

Memento design pattern is implemented with two objects – Originator and Caretaker.

Originator is the object whose state needs to be saved and restored and it uses an [inner class](https://www.journaldev.com/996/java-inner-class) to save the state of Object. The inner class is called **Memento** and it’s private, so that it can’t be accessed from other objects.

Caretaker is the helper class that is responsible for storing and restoring the Originator’s state through Memento object. Since Memento is private to Originator, Caretaker can’t access it and it’s stored as an Object within the caretaker.

One of the best real life example is the text editors where we can save it’s data anytime and use undo to restore it to previous saved state.

--------------Miscellaneous Design Pattern---------------------

**Java Dependency Injection** [**design pattern**](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable. We can implement **dependency injection in java** to move the dependency resolution from compile-time to runtime.