**Design Patterns**

* In 1994, four authors **Erich Gamma**, **Richard Helm**, **Ralph Johnson** and **John Vlissides** published a book titled Design Patterns - Elements of Reusable Object-Oriented Software which initiated the concept of Design Pattern in Software development. These authors are collectively known as **Gang of Four (GOF).**
* A design pattern systematically names, motivates and explains a general design that addresses a recurring design problem in object-oriented systems. Design patterns are needed to represent some of the best practices followed and adopted in software development.
* Types of Design Patterns

1. **Creational Patterns**

* Singleton Pattern
* Factory Pattern
* Abstract Factory Pattern
* Prototype Pattern
* Builder Pattern

1. **Structural Design Patterns**

* Adapter Pattern
* Bridge Pattern
* Composite Pattern
* Decorator Pattern
* Facade Pattern
* Flyweight Pattern
* Proxy Pattern

1. **Behavioral Design Patterns**

* Chain Of Responsibility Pattern
* Command Pattern
* Interpreter Pattern
* Iterator Pattern
* Mediator Pattern
* Memento Pattern
* Observer Pattern
* State Pattern
* Strategy Pattern
* Template Pattern
* Visitor Pattern

**Creation Design Patterns**

* Creational patterns deal with the creation of objects.
* **Singleton design pattern**
* Singleton Pattern defines a class with only one instance and provides a global point of access.
* Singleton pattern is used for logging, driver objects, caching, and thread pool.
* Use the Singleton pattern when you need stricter control over global variables.
* Subclassing/Inheritance must not be allowed
* Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Façade, etc.
* Using a synchronized method makes sure that only one thread at a time can execute **getInstance()**. The main disadvantage of this method is that using synchronized every time while creating the singleton object is expensive and may decrease the performance of your program. However, if the performance of ***getInstance()*** is not critical for your application this method provides a clean and simple solution.

***public static synchronized Singleton getInstance() {***

***if (obj==null)***

***obj = new Singleton();***

***return obj;***

***}***

* Double-checked locking principle is used to create instances. In that, the synchronized block is used *inside* the if condition with an additional check to ensure that only one instance of a singleton class is created.

***public static ThreadSafeSingleton getInstanceUsingDoubleLocking() {***

***if (instance == null) {***

***synchronized (ThreadSafeSingleton.class) {***

***if (instance == null) {***

***instance = new ThreadSafeSingleton();***

***}***

***}***

***}***

***return instance;***

***}***

* **Reflection** can be used to destroy the singleton implementation approaches. To overcome this situation with Reflection, ***Joshua Bloch*** suggests the use of ***enum*** to implement the singleton design pattern as Java ensures that any ***enum*** value is instantiated only once in a Java program.

*Constructor constructor = singleton.getClass().getDeclaredConstructor(new Class[0]);*

***constructor.setAccessible(true);***

* ***Java Enum*** values are globally accessible, and so is the singleton. The drawback is that the enum type is somewhat inflexible for example; it does ***not allow lazy initialization*** and when serializing an enum, field variables are not getting serialized.
* Sometimes in distributed systems, we need to implement a Serializable interface in the singleton class so that we can store its state in the file system and retrieve it later. The problem with serialized singleton class is that whenever we deserialize it, it will create a new instance of the class. So it destroys the singleton pattern. To overcome this scenario, all we need to do is provide the implementation of the ***readResolve()*** method, which is called when preparing the deserialized object before returning it to the caller.

*public class Singleton implements Serializable{*

*public static final Singleton INSTANCE = new Singleton();*

*private Singleton() {*

*}*

*protected Object readResolve() {*

*return INSTANCE;*

*}*

*}*

* Singleton design pattern is used in core Java classes also (for example, ***java.lang.Runtime***, ***java.awt.Desktop***).
* **Factory Pattern**
* The factory design pattern is used when we have a superclass with multiple subclasses and based on input, we need to return one of the subclasses. This pattern takes out the responsibility of the instantiation of a class from the client program to the factory class. We can apply a singleton pattern on the factory class or make the factory method static.
* Super class in factory design pattern can be *an interface, abstract class, or a normal java class*.
* We can keep Factory class Singleton or we can keep the method that returns the subclass as static.
* Factory Design Pattern Examples in JDK : ***valueOf()*** method in wrapper classes like Boolean, Integer etc.
* *java.util.Calendar*, *ResourceBundle* and *NumberFormat* ***getInstance()*** methods uses Factory pattern.
* **Abstract Factory Pattern**
* The abstract factory pattern is similar to the factory pattern and is a factory of factories.
* The Abstract Factory Pattern is a way of organizing how you create groups of things that are related to each other. It provides a set of rules or instructions that let you create different types of things without knowing exactly what those things are. This helps you keep everything organized and lets you switch between different types easily.
* So at runtime, the abstract factory is coupled with any desired concrete factory which can create objects of the desired type.
* Abstract Factory Design Pattern Examples in JDK
* javax.xml.parsers.DocumentBuilderFactory#newInstance()
* javax.xml.xpath.XPathFactory#newInstance()
* **Prototype Pattern**
* Prototype Pattern is the cloning of an existing object instead of creating a new one and can also be customized as per the requirement.
* Prototype pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs.
* This pattern should be followed if the cost of creating a new object is expensive and resource-intensive.
* An object that supports cloning is called a prototype.
* **Example**: Suppose we have an Object that loads data from the 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 a new keyword and load all the data again from the database. The better approach would be to clone the existing object into a new object and then do the data manipulation. Prototype design pattern mandates that the Object which you are copying should provide the copying feature. It should not be done by any other class. However whether to use shallow or deep copy of the Object properties depends on the requirements and its a design decision.
* **Object Pool Pattern**
* Object pool pattern is a software creational design pattern which is used in situations where the cost of initializing a class instance is very high.
* Basically, an Object pool is a container which contains some amount of objects. So, when an object is taken from the pool, it is not available in the pool until it is put back.
* By using an object pool, an application can avoid the overhead of creating and destroying objects, making it more efficient and scalable.
* Objects in the pool have a lifecycle:
* Creation
* Validation
* Destroy
* **Reference**
* https://medium.com/javarevisited/java-developer-guide-to-start-with-object-pool-design-pattern-3d5fb121b327
* **Builder Pattern**
* 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.

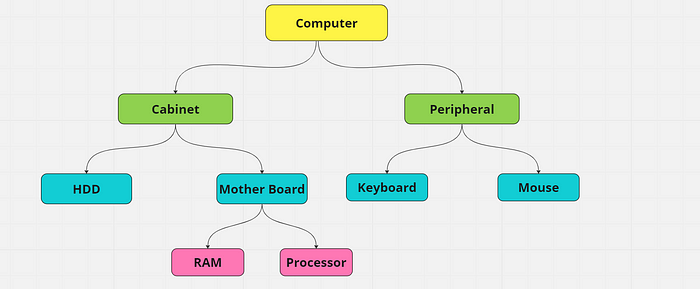
1. Too Many arguments to pass from the client program to the Factory class that can be error prone because most of the time, the type of arguments are the same and from the client side its hard to maintain the order of the arguments.
2. Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to be sent 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 a large number of parameters by providing a constructor with the 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 unless all the attributes are set explicitly.
* Builder pattern solves the issue with a 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.
* It is used to construct a complex object step by step and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.
* **Advantages of Builder Design Pattern**
* The parameters to the constructor are reduced and are provided in highly readable method calls.
* Builder design pattern also helps in minimizing the number of parameters in the constructor and thus there is no need to pass in null for optional parameters to the constructor.
* Object is always instantiated in a complete state
* Immutable objects can be built without much complex logic in the object building process.
* **Disadvantages of Builder Design Pattern**
* The number of lines of code increases at least to double in builder pattern, but the effort pays off in terms of design flexibility and much more readable code.
* Requires creating a separate ConcreteBuilder for each different type of Product.

**Structural Design Pattern**

Structural design patterns provide different ways to create a Class structure (for example, using inheritance and composition to create a large Object from small Objects).

* **Adapter Design Pattern**
* Adapter design pattern is one of the structural design pattern and its used so that two unrelated interfaces can work together. The object that joins these unrelated interface is called an **Adapter**.
* The Adapter Pattern is also known as **Wrapper**.
* The Adapter acts as a wrapper between two objects. It catches calls for one object and transforms them to format and interface recognizable by the second object.
* While implementing Adapter pattern, there are two approaches - class adapter and object adapter - however both these approaches produce same result.
* **Class Adapter** - This form uses java inheritance and extends the source interface, in our case Socket class.
* **Object Adapter** - This form uses Java Composition and adapter contains the source object.
* Some of the adapter design pattern example we could easily find in JDK classes are;
* java.util.Arrays#asList()
* java.io.InputStreamReader(InputStream) (returns a Reader)
* java.io.OutputStreamWriter(OutputStream) (returns a Writer)
* **Reference**
* <https://www.scaler.com/topics/design-patterns/adapter-design-pattern/>
* **Bridge Pattern**
* Bridge Design Pattern there are two layers ***i.e. Abstraction and Implementation.***
* ***In Bridge Design Pattern, if we do make any changes in the Implementation layer then it won’t affect the Abstraction Layer and in the same way, if we do any changes in the abstraction layer then it won’t affect the Implementation layer.***
* Bridge design pattern can be used when both abstraction and implementation can have different hierarchies independently and we want to hide the implementation from the client application.
* Points in Bridge Pattern:
* The bridge pattern allows the Abstraction and the Implementation to be developed independently and the client code can access only the Abstraction part without being concerned about the Implementation part.
* The abstraction is an interface or abstract class and the implementer is also an interface or abstract class.
* The abstraction contains a reference to the implementer. Children of the abstraction are referred to as refined abstractions, and children of the implementer are concrete implementers. Since we can change the reference to the implementer in the abstraction, we are able to change the abstraction’s implementer at run-time. Changes to the implementer do not affect client code.
* It increases the loose coupling between class abstraction and it’s implementation.
* **Composite Pattern**
* Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
* When we need to create a structure in a way that the objects in the structure have to be treated the same way, we can apply a composite design pattern.
* Composite Pattern consists of the following objects.
* **Base Component** - Base component is the interface for all objects in the composition, client program uses the base component to work with the objects in the composition. It can be an interface or an abstract class with some methods common to all the objects.
* **Leaf** - Defines the behavior of the elements in the composition. It is the building block for the composition and implements the base component. It doesn’t have references to other Components.
* **Composite** - It consists of leaf elements and implements the operations in the base component.
* **Composite Pattern Important Points**
* Composite pattern should be applied only when the group of objects should behave as a single object.
* Composite design patterns can be used to create a tree-like structure.



* **Decorator Pattern**
* Decorator design pattern 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 the individual object gets the modified behavior.
* Decorator patterns allow a user to add new functionality to an existing object without altering its structure. So, there is no change to the original class. It acts as a wrapper to the existing class.
* Suppose we want to implement different kinds of cars - we can create interface Car to define the assemble method and then we can have a Basic car, furthermore, we can extend it to Sports car and Luxury Car. But if we want to get a car at runtime that has both the features of a sports car and a luxury car, then the implementation gets complex, and if furthermore, we want to specify which features should be added first, it gets even more complex. To solve this kind of programming situation, we apply a decorator pattern in Java.
* Decorator design pattern helps provide runtime modification abilities and hence is more flexible. It's easy to maintain and extend when the number of choices is more.
* Decorator pattern is used a lot in Java IO classes, such as *FileReader*, *BufferedReader* etc.
* The disadvantage of the decorator design pattern is that it uses a lot of similar kinds of objects (decorators).
* The decorator design pattern is structurally almost like the chain of responsibility pattern.
* **Façade Design Pattern**
* It hides the complexity of the underlying system and provides a simple interface that clients can use to interact with the system. Facade Method Design Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a high-level interface that makes the subsystem easier to use.
* **Flyweight Design Pattern**
* Flyweight design pattern is a Structural design pattern used when we need to create a lot of Objects for 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.
* One important feature of flyweight objects is that they are immutable. This means that they cannot be modified once they have been constructed.
* Flyweight saves RAM by caching the same data used by different objects.
* Before we apply flyweight design pattern, we need to consider the following factors:
* The number of Objects to be created by the application should be huge.
* The object creation is heavy on memory and it can be time-consuming too.
* The object properties can be divided into intrinsic and extrinsic properties, extrinsic properties of an Object should be defined by the client program.
* To apply flyweight pattern, we need to divide Object property into intrinsic and extrinsic properties. Intrinsic properties make the Object unique whereas extrinsic properties are set by client code and used to perform different operations.
* All the wrapper classes valueOf() method uses cached objects showing use of Flyweight design pattern. The best example is Java String class String Pool implementation.’
* **Proxy Design Pattern**
* Proxy means ‘in place of’, representing’ or ‘in place of’ or ‘on behalf of’ are literal meanings of proxy and that directly explains Proxy Design Pattern.
* Proxies are also called surrogates, handles, and wrappers. They are closely related in structure, but not purpose, to Adapters and Decorators.
* 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. And that’s exactly what the Proxy pattern does – ***“Controls and manage access to the object they are protecting“.***
* Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.
* The proxy object has the same interface as a service, which makes it interchangeable with a real object when passed to a client.

**Behavioral Design Pattern**

* **Chain of Responsibility Design Pattern**
* Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides to process the request or pass it to the next handler in the chain.
* Chain of responsibility pattern is used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them. Then the object in the chain will decide who will be processing the request and whether the request is required to be sent to the next object in the chain or not.
* **Example in JDK:** We know that we can have multiple catch blocks in a try-catch block 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 sent to the first catch block to process. If the catch block is not able to process it, it forwards the request to the next object in the 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.
* **Real-world example:** One of the great examples of the Chain of Responsibility pattern is the ATM Dispensing machine. The user enters the amount to be dispensed and the machine dispenses the 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.
* **Command Design Pattern**
* Command design pattern is used to implement loose coupling in a request-response model.
* The request is sent to the *invoker* and the invoker passes 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 creates the receiver object and then attaches it to the Command. Then it creates the invoker object and attaches the command object to perform an action. Now when client program executes the action, it’s processed based on the command and receiver object.
* ***Runnable*** interface (java.lang.Runnable) and Swing Action (javax.swing.Action) uses command pattern
* **Template Method Pattern**
* Template method defines the steps to execute an algorithm and it can provide default implementation that might be common for all or some of the subclasses
* An abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class.
* It’s okay if all the steps end up being abstract. However, some steps might benefit from having a default implementation. Subclasses don’t have to implement those methods.
* **Observer Design Pattern**
* 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***.
* According to GoF, observer design pattern intent is***; Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically***.
* ***Java Message Service (JMS)*** uses *Observer design pattern* along with *Mediator pattern* to allow applications to subscribe and publish data to other applications. 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.
* **Memento Pattern**
* Memento pattern is used to restore state of an object to a previous state. It is also known as ***snapshot pattern***.
* The Memento Design Pattern offers a solution to implement undoable actions. We can do this by saving the state of an object at a given instant and restoring it if the actions performed since need to be undone.
* A memento is is like a restore point during the life cycle on the object, which the client application can use to restore the object state to its state. Conceptually, it is very much like we create restore points for operating systems and use to restore the system if something breaks or system crashes.
* When to use:
* Memento pattern shall be be used in any application in which object’s state is continuously changing and the user of the application may decide to rollback or undo the changes changes at any point.
* A memento can also be used in applications which must be restarted from their last known working state or draft. An example of this can be an IDE which restart from changes, user made before closing the IDE.
* In code editors, we can revert or apply any code change with simple commands to undo and redo.
* In calculator applications, we can revisit all the calculations in memory with simple button press.
* In programming, memento can be used to create checkpoints during database transactions. If any operation fails, we just rollback everything to last known stable database state.