Design Pattern

1. Creational Design Pattern

1.1. Singleton design pattern in Java

Singleton Pattern says that just **"define a class that has only one instance and provides a global point of access to it".**

In other words, a class must ensure that only single instance should be created and single object can be used by all other classes

* **Early Instantiation:** creation of instance at load time.
* **Lazy Instantiation:** creation of instance when required.

#### **Advantage of Singleton design pattern**

* Saves memory because object is not created at each request. Only single instance is reused again and again.

#### **Usage of Singleton design pattern**

* Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

Git Url:

<https://git.epam.com/krishna_singh1/rd-training/tree/master/Design%20Patterns/Singleton%20Pattern>

# 1.2. Factory Method Pattern

A Factory Pattern or Factory Method 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 Method Pattern is also known as **Virtual Constructor.**

#### **Advantage of Factory Design Pattern**

* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

#### **Usage of Factory Design Pattern**

* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes choose the creation of objects to its sub-classes.

Git Url:

<https://git.epam.com/krishna_singh1/rd-training/tree/master/Design%20Patterns/Factory%20Method>

# 1.3. Abstract Factory Pattern

Abstract Factory Pattern says that just **define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.**That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

An Abstract Factory Pattern is also known as **Kit.**

#### **Advantage of Abstract Factory Pattern**

* Abstract Factory Pattern isolates the client code from concrete (implementation) classes.
* It eases the exchanging of object families.
* It promotes consistency among objects.

#### **Usage of Abstract Factory Pattern**

* When the system needs to be independent of how its object are created, composed, and represented.
* When the family of related objects has to be used together, then this constraint needs to be enforced.
* When you want to provide a library of objects that does not show implementations and only reveals interfaces.
* When the system needs to be configured with one of a multiple family of objects.

Git Url:

<https://git.epam.com/krishna_singh1/rd-training/tree/master/Design%20Patterns/Abstract%20Factory%20Pattern>

# 1.4. Builder Design Pattern

Builder Pattern says that **"construct a complex object from simple objects using step-by-step approach"**

It is mostly used when object can't be created in single step like in the de-serialization of a complex object.

#### **Advantage of Builder Design Pattern**

The main advantages of Builder Pattern are as follows:

* It provides clear separation between the construction and representation of an object.
* It provides better control over construction process.
* It supports to change the internal representation of objects.

Git Url:

<https://git.epam.com/krishna_singh1/rd-training/tree/master/Design%20Patterns/Builder%20Pattern>

# Behavioral Design Pattern

# 2.1. Strategy Pattern

A Strategy Pattern says that "defines a family of functionality, encapsulate each one, and make them interchangeable".

The Strategy Pattern is also known as Policy.

## **Benefits:**

* It provides a substitute to subclassing.
* It defines each behavior within its own class, eliminating the need for conditional statements.
* It makes it easier to extend and incorporate new behavior without changing the application.

## **Usage:**

* When the multiple classes differ only in their behaviors.e.g. Servlet API.
* It is used when you need different variations of an algorithm.

Git Url:

<https://git.epam.com/krishna_singh1/rd-training/tree/master/Design%20Patterns/Strategy%20Pattern>

# 2.2. Chain Of Responsibility Pattern

In chain of responsibility, sender sends a request to a chain of objects. The request can be handled by any object in the chain.

A Chain of Responsibility Pattern says that just **"avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request".** For example, an ATM uses the Chain of Responsibility design pattern in money giving process.

In other words, we can say that normally each receiver contains reference of another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

#### **Advantage of Chain of Responsibility Pattern**

* It reduces the coupling.
* It adds flexibility while assigning the responsibilities to objects.
* It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.

#### **Usage of Chain of Responsibility Pattern:**

It is used:

* When more than one object can handle a request and the handler is unknown.
* When the group of objects that can handle the request must be specified in dynamic way.

Git Url:

<https://git.epam.com/krishna_singh1/rd-training/tree/master/Design%20Patterns/Chain%20of%20Responsibility%20Pattern>

# 2.3. Observer Pattern

An Observer Pattern says that "just define a one-to-one dependency so that when one object changes state, all its dependents are notified and updated automatically".

The Memento pattern is also known as Dependents or Publish-Subscribe.

## **Benefits:**

* It describes the coupling between the objects and the observer.
* It provides the support for broadcast-type communication.

## **Usage:**

* When the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
* When the framework we writes and needs to be enhanced in future with new observers with minimal chamges.

Git Url:

<https://git.epam.com/krishna_singh1/rd-training/tree/master/Design%20Patterns/Observer%20Pattern>

**2.4. Iterator Pattern**

According to GoF, Iterator Pattern is used **"to access the elements of an aggregate object sequentially without exposing its underlying implementation".**

The Iterator pattern is also known as **Cursor.**

In collection framework, we are now using Iterator that is preferred over Enumeration.

#### **Advantage of Iterator Pattern**

* It supports variations in the traversal of a collection.
* It simplifies the interface to the collection.

#### **Usage of Iterator Pattern:**

It is used:

* When you want to access a collection of objects without exposing its internal representation.
* When there are multiple traversals of objects need to be supported in the collection.

Git Url:

<https://git.epam.com/krishna_singh1/rd-training/tree/master/Design%20Patterns/Iterator%20Pattern>

1. **Structural Design Pattern** 
   1. **Adapter Pattern**

An Adapter Pattern says that just **"converts the interface of a class into another interface that a client wants".**

In other words, to provide the interface according to client requirement while using the services of a class with a different interface.

The Adapter Pattern is also known as **Wrapper.**

#### **Advantage of Adapter Pattern**

* It allows two or more previously incompatible objects to interact.
* It allows reusability of existing functionality.

#### **Usage of Adapter pattern:**

It is used:

* When an object needs to utilize an existing class with an incompatible interface.
* When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.
* When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.

Git Url:

<https://git.epam.com/krishna_singh1/rd-training/tree/master/Design%20Patterns/Adapter%20Pattern>

# Proxy Pattern

Simply, proxy means an object representing another object.

According to GoF, a Proxy Pattern **"provides the control for accessing the original object".**

So, we can perform many operations like hiding the information of original object, on demand loading etc.

Proxy pattern is also known as **Surrogate or Placeholder.**

#### **Advantage of Proxy Pattern**

* It provides the protection to the original object from the outside world.

#### **Usage of Proxy Pattern:**

It is used:

* It can be used in **Virtual Proxy** scenario---Consider a situation where there is multiple database call to extract huge size image. Since this is an expensive operation so here we can use the proxy pattern which would create multiple proxies and point to the huge size memory consuming object for further processing. The real object gets created only when a client first requests/accesses the object and after that we can just refer to the proxy to reuse the object. This avoids duplication of the object and hence saving memory.
* It can be used in **Protective Proxy** scenario---It acts as an authorization layer to verify that whether the actual user has access the appropriate content or not. For example, a proxy server which provides restriction on internet access in office. Only the websites and contents which are valid will be allowed and the remaining ones will be blocked.
* It can be used in **Remote Proxy** scenario---A remote proxy can be thought about the stub in the RPC call. The remote proxy provides a local representation of the object which is present in the different address location. Another example can be providing interface for remote resources such as web service or REST resources.
* It can be used in **Smart Proxy** scenario---A smart proxy provides additional layer of security by interposing specific actions when the object is accessed. For example, to check whether the real object is locked or not before accessing it so that no other objects can change it.

Git Url:

<https://git.epam.com/krishna_singh1/rd-training/tree/master/Design%20Patterns/Proxy%20Pattern>

* 1. **Decorator Pattern**

A Decorator Pattern says that just **"attach a flexible additional responsibilities to an object dynamically".**

In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

The Decorator Pattern is also known as **Wrapper.**

#### **Advantage of Decorator Pattern**

* It provides greater flexibility than static inheritance.
* It enhances the extensibility of the object, because changes are made by coding new classes.
* It simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object.

#### **Usage of Decorator Pattern**

It is used:

* When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
* When you want to add responsibilities to an object that you may want to change in future.
* Extending functionality by sub-classing is no longer practical.

Git Url:

<https://git.epam.com/krishna_singh1/rd-training/tree/master/Design%20Patterns/Decorator%20Pattern>