**What are Design Patterns?**

**The design pattern**is a reusable solution to a common problem that occurs in software design.

There are many different types of design patterns, each addressing a specific problem or set of problems.

As shown in the image below, some common types of design patterns include **creational patterns**, which deal with **object creation**; **structural patterns**, which focus on **object composition and organization**; and **behavioural patterns**, which deal with the **communication between objects and classes**.

**1.Creational Design Patterns :**

These deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

[*Creational Design Patterns*](https://www.geeksforgeeks.org/system-design/creational-design-pattern/) *focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed and represented.*

**Singleton:** Ensures a class has only one instance and provides a global point of access to it.

**Factory Method:** Provides an interface for creating objects in a **superclass**, but allows **subclasses** to alter the type of objects that will be created.

**Abstract Factory:** Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

**Builder:** Separates the construction of a complex object from its representation.

**Prototype:** Creates new objects by copying an existing object, known as the prototype.

**2. Structural Patterns**

**These deal with object composition or the structure of classes.**

[*Structural Design Patterns*](https://www.geeksforgeeks.org/system-design/structural-design-patterns/) *solves problems related to how classes and objects are composed/assembled to form larger structures which are efficient and flexible in nature. Structural class patterns use inheritance to compose interfaces or implementations.*

**Adapter:** Allows incompatible interfaces to work together.

**Bridge:** Separates an object’s abstraction from its implementation.

**Composite:** Composes objects into tree structures to represent part-whole hierarchies.

**Decorator:** Adds new functionality to an object dynamically.

**Facade:** Provides a simplified interface to a complex subsystem.

**Flyweight:** Reduces the cost of creating and manipulating a large number of similar objects.

**Proxy:** Provides a surrogate or placeholder for another object to control access to it.

**3. Behavioural Patterns**

**These are concerned with algorithms and the assignment of responsibilities between objects.**

[*Behavioral Patterns*](https://www.geeksforgeeks.org/system-design/behavioral-design-patterns/) *are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that’s difficult to follow at run-time.*

**Observer:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.

**Strategy:** Enables selecting an algorithm’s behaviour at runtime.

**Command:** Encapsulates a request as an object, thereby allowing for parameterization and queuing of requests.

**Chain of Responsibility:** Passes a request along a chain of handlers.

**Mediator:** Defines an object that encapsulates how a set of objects interact.

**Memento:** Captures and restores an object’s internal state.

**State:** Allows an object to alter its behaviour when its internal state changes.

**Template Method:** Defines the skeleton of an algorithm in a method, deferring some steps to subclasses.

**Visitor:** Separates an algorithm from the object structure it operates on.

**Interpreter:** Implements a grammar for a language and interprets sentences in that language.

[**https://capgemini.udemy.com/course/java-design-patterns/learn/lecture/24967100#overview**](https://capgemini.udemy.com/course/java-design-patterns/learn/lecture/24967100#overview)

**Singleton:** Ensures a class has only one instance and provides a global point of access to it.

----------

Singleton pattern is one of the simplest design patterns in Java.

The **Singleton design pattern** is a creational pattern that ensures a class has only **one instance** and provides a **global point of access** to that instance.

This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

**public** **class** DateUtil **implements** Serializable, Cloneable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** **static** **volatile** DateUtil *instance*;

**private** DateUtil() {

}

**public** **static** DateUtil getInstace() {

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

**synchronized** (DateUtil.**class**) {

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

*instance*=**new** DateUtil();

}

}

} **return** *instance*;

}

**protected** Object readResolve() {

**return** *instance*;

}

@Override

**protected** Object clone() **throws** CloneNotSupportedException {

**throw** **new** CloneNotSupportedException();

}

}

**Factory Method:**

The **Factory Method design pattern** is a *creational pattern* that provides an interface for creating objects in a **superclass**, but allows **subclasses** to alter the type of objects that will be created.

It's all about delegating the instantiation logic to subclasses, so the code becomes more flexible and decoupled

In the Factory pattern, we don’t expose the creation logic to the client and refer the created object using a standard interface.

The Factory Pattern is also known as Virtual Constructor.

Steps:

1.Create main class which call factory class.

2.Factory class returns required class instance.

A factory pattern is a creational pattern that abstracts or hides the object creation process.

**Subclass :**

* A **subclass** is any class that inherits from another class (called a superclass).
* It can be **abstract** or **concrete**.
* It may override or extend the behaviour of its superclass.

**Example :**

When you think of factory you can think of a car factory a chocolate factory or a toy factory.

A car factory is responsible for manufacturing the cars .A car dealer need not worry about how the car is manufactured.

He simply asks the car factory to deliver him some cars. The car factory is responsible for manufacturing them and delivering them to the dealer.

**public class PizzaFactory {**

**public static Pizza createPizza(String type) {**

**Pizza p = null;**

**if (type.equals("cheese")) {**

**p = new CheesePizza();**

**} else if (type.equals("chicken")) {**

**p = new ChickenPizza();**

**} else if (type.equals("veggie")) {**

**p = new VeggiePizza();**

**}**

**return p;**

**}**

**}**

**Abstract Factory:**

The Abstract Factory Pattern is a creational design pattern that provides an interface for creating **families of related or dependent objects without specifying their concrete classes**.

* Abstract Factory pattern is almost similar to Factory Pattern and is considered as another layer of abstraction over factory pattern.
* Abstract Factory patterns work around a super-factory which creates other factories.

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

In the Factory patterns we don’t expose the creation logic to the client and refer the created object using a standard interface .

The Factory Pattern is also known as Virtual Constructor.

Steps:

1.Create main class which call factory class.

2.Factory returns required class instance.

**Concreate Class :**

A concrete class is a class that provides complete implementations for all its methods. That means it's ready to be used to create objects, unlike abstract classes or interfaces, which may leave some methods unimplemented.

* A class that has a **complete implementation** of all its methods
* You can **instantiate** it directly (create objects from it)
* Unlike an abstract class or interface, it doesn’t have any unimplemented methods

| **Feature** | **Factory Method** | **Abstract Factory** |
| --- | --- | --- |
| **Purpose** | Creates *one* product | Creates *families* of related products |
| **Inheritance** | Uses inheritance: subclass overrides a method | Uses composition: encapsulates multiple factories |
| **Scalability** | Adds new product types via new subclasses | Adds new product families via new factory classes |
| **Complexity** | Simpler | More complex, but offers greater flexibility |
| **Coupling** | Loosely coupled to one product | Loosely coupled to product families |
| **Real-world Analogy** | A bakery making one type of pastry | A furniture factory producing chairs + tables |

**Prototype:** Creates new objects by copying an existing object, known as the prototype.

-----------

The **Prototype design pattern** is a *creational pattern* used when the cost of creating a new object is more expensive than copying an existing one. Instead of instantiating new objects, you **clone** existing ones—making the process faster and more efficient

* **Cloning**: Objects are duplicated using a copy mechanism (usually a clone() method).
* **Avoids Costly Initialization**: Handy when object creation is complex or resource-heavy.
* **Encapsulation of Copy Logic**: Each class defines how it should be cloned

**Builder:** Separates the construction of a complex object from its representation.

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

Builder pattern refer to approach that focuses on construction a complex object form simple

Objects using step-by-step approach.

**Builder Pattern:**

Builder pattern will give us an option to create an object for only specific fields.

Builder design pattern belongs to Creational design pattern which is a type of Design Patterns in java.

Builder pattern builds a complex object using simple objects and uses step by step approach. A Builder class builds the final object step by step. This builder is independent of other objects.

==============================

**package** com.builder;

**public** **class** HttpClient {

**private** String method;

**private** String url;

**private** String userName;

**private** String password;

**private** String headers;

**private** String body;

**public** HttpClient(String method, String url, String userName, String password, String headers,  String body) {

**super**();

**this**.method = method;

**this**.url = url;

**this**.userName = userName;

**this**.password = password;

**this**.headers = headers;

**this**.body = body;

                 }

@Override

**public** String toString() {

**return** "HttpClient [method=" + method + ", url=" + url + ", userName=" + userName + ", password=" + password

                                                                       + ", headers=" + headers + ", body=" + body + "]";

                 }

}

=============================================

**package** com.builder;

**public** **class** HttpClientBuilder {

**private** String method;

**private** String url;

**private** String userName;

**private** String password;

**private** String headers;

**private** String body;

**public** HttpClientBuilder setMethod(String method) {

**this**.method = method;

**returnthis**;

                 }

**public** HttpClientBuilder setUrl(String url) {

**this**.url = url;

**returnthis**;

                 }

**public** HttpClientBuilder setUserName(String userName) {

**this**.userName = userName;

**returnthis**;

                 }

**public** HttpClientBuilder setPassword(String password) {

**this**.password = password;

**returnthis**;

                 }

**public** HttpClientBuilder setHeaders(String headers) {

**this**.headers = headers;

**returnthis**;

                 }

**public** HttpClientBuilder setBody(String body) {

**this**.body = body;

**returnthis**;

                 }

**public** HttpClient getHttpClient() {

**returnnew** HttpClient(method, url, userName, password, headers, body);

                 }

}

**publicclass** Test {

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

         HttpClient builder = **new** HttpClientBuilder().setUrl("http://youtube.com").getHttpClient();

        System.***out***.println(builder).

   }

}

OUTPUT: HttpClient [method=null, url=<http://youtube.com>, userName=null, password=null, headers=null, body=null]

**2. Structural Design Patterns in Java**

Structural design patterns focus on how classes and objects are arranged to create larger, more complex structures in software development. They help organize relationships between objects, making systems more flexible, reusable, and maintainable. By using these patterns, developers can create easier-to-understand and modify systems.

Structural design patterns show you how to assemble different pieces of system together in a flexible and extensible fashion.

They help you guarantee that when one of the parts changes, the entire structure does not need to change.

**Proxy:**

The **Proxy design pattern** is a *structural pattern* that provides a substitute or placeholder for another object to **control access** to it. Think of it as a middleman that intercepts requests to the actual object, adding a layer of logic before or after forwarding the call.

**Roles in Proxy Design Patterns:**

**Subject :** It is an interface which expose the functionality available to used by the clients.

**Real Subject :** Is a class implementing subject and it is concreate implementation which need to be hidden behind a proxy.

**Proxy** **:** Hides the real object by extending it and clients communicate to real object via this proxy object. Usually, frameworks create this proxy objects when client request for real object.

**Advantages of Proxy Design Patterns :**

* Access control / Protection given by proxy object : This is when you want only specific clients to be able to use the service object.
* **Caching** request result(caching proxy). This is when you need to cache result of client requests and manage the life cycle of this cache, especially if results are quite large.
* **Logging** requests (logging proxy). This is when to keep a history of requests to the service object.

**Flyweight :**

It is Structural design patterns.

A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context.

**Where to Use Flyweight DP :**

When your application needs large number of objects that shares most of the common attributes called Intrinsic Attributes or properties and only few unique attributes called as Extrinsic attributes / properties.

We need to control the memory consumption by large number of objects – by creating fewer objects and sharing then across.

**When to Use Flyweight DP :**

The number of objects to be created by 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.