**Simple Factory DP:**

Simple Factory DP is a creational design pattern that encapsulates the object creation logic into a centralised factory class.

All object creation happens in one place (the centralised factory). Client code is decoupled from logic to create objects.

Having if-else part in client code is combining object creation logic with business logic. If if-else part is kept in client code then, if Chair class is renamed then you have to modify client code so move the logic to factory class.

Now client just need value to pass to factory and factory take care of creating objects. Client code depends on interface(Furniture) rather than concrete implementations (Chair etc.). SRP is followed but OCP is violated in Factory class.

Abstract Product, Concrete Product, Factory and Client are the components.

Examples: Shape(Circle, Rectangle), Account(Current, Saving, Business), Logger(ErrorLogger, DebugLogger).

**Factory Method DP:**

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

Unlike the Simple Factory, which uses a single class to create objects, the Factory Method delegates object creation to subclasses. Factory Method DP is not centralised.

It includes all advantages of Simple Factory DP and is an extension for it. Both SRP and OCP are not violated here.

Used when you want to add flexibility to system by introducing new products without changing the existing code.

Instead of object creation we can directly do Chair chair; chair.display(); but client code need to be modified after each new product and it has direct knowledge about concrete classes.

Abstract Product, Concrete Product, Abstract Factory, Concrete Factory and Client are the components.

Examples: Shape(Circle, Rectangle), Account(Current, Saving, Business), Logger(ErrorLogger, DebugLogger).

**Abstract Factory DP:**

The Abstract Factory DP is a creational design pattern used to create families of related objects without specifying concrete classes. (Here Modern and Traditional are the families)

Used when you want to build complex system with interrelated objects and decouple the creation logic.

We can make it follow OCP by simply avoid passing string & remove if-else condition & create object for family that is needed. It has all advantages of SF and FM design patterns.

Abstract Product, Concrete Product, Family Factory, Concrete Factory and Client are the components.

**Builder DP:**

The Builder DP is a creational design pattern that separates the construction of a complex object from its representation, allowing the same construction process to create different representations. It's useful when an object requires many fields to be initialized.

Any number of new variants can be introduced by creating classes and we can use same buildDestop() to call it’s functions which are implementations of DesktopBuilder class. Here DesktopDirector is the boilerplate code.

Here we are avoiding passing parameters to initialize fields by not creating objects and hardcoding them in methods itself. So objects once created cannot be modified. Also we are avoiding telescoping constructor issue as more parameters make it unreadable/difficult.

If customization of object is needed then we will go for Factory DP. If in creation of each object type, steps to be followed is same then Builder DP is preferred. Product, Abstract Builder, Concrete Builder, Director and Client are the components.

**Prototype DP:**

The Prototype is a creational design pattern that allows cloning existing objects by making the code independent on their classes. It is useful when object creation is costly (e.g., updating database) and you need similar but copies of an object.

Avoids creating object from scratch. Reduces subclassing as cloning replaced inheritance. Allows Dynamic object configuration (modifiable at runtime) by customizing properties at runtime by introducing methods(setName()) without affecting the original.

If I have to create branches for my restaurant and need to copy same menu for different cuisines. Now I will clone it for each new branch. Within an item we need to add description (name, price, rating), image etc, so we can just clone which makes it simpler.

Abstract Prototype, Concrete Prototype and Client are the components. Deep vs Shallow copy?

**Singleton DP:**

Singleton DP is a creational DP which ensures that only one instance of a class is created and it provides a global point of access to that instance.

It’s useful when exactly one object is needed to coordinate actions across a system and you want to control access to shared resources(mutual exclusion) by providing centralised access. Ex: One shared access point to manage DB.

Does not follow SRP as object creation logic is combined with normal class logic. If mutex is removed then program is not thread safe. Private constructor, Private static instance, Public static method and client are the components.

A private constructor prevents external classes(using Singleton obj; or new Singleton();) from directly creating new instances of the class using the new keyword. Only the class itself can create an instance.

Private static instance variable as external classes should not be able to modify(null) and same location has to be checked for each instance. Whenever a new instance is trying to get created then we should check for same field so we are making it static so it will be common for every instance.

Public static method as we should be able to access method from external classes without object(as Singleton obj; or new Singleton(); are not allowed due to private constructor).

I have an UI which has modules like recordings, notes, code etc and if I change configuration say light theme or dark theme it should be reflected in all modules. Similarly if a restaurant is closed then delivery partner and user both should know.

Double-Checked Locking is a performance optimization used to avoid unnecessary locking in multi-threaded environment once the Singleton instance is initialized. If if-null checks are removed then code is thread safe without double-checked locking and is expensive.

Acquiring lock every time getInstance() is called, even if the instance is already created, will be very expensive, especially if getInstance() is called frequently after the Singleton is created.

Early/Eager Initialization - The object is created at the time of class loading/application startup, whether needed or not. Instance is always available, so consume resources even when instance is not needed.

Late/Lazy Initialization - The object is created only when it is first needed or requested. This conserves resources and used when creation/maintenance of instance is costly.

Examples: Centralised logging across the application, Device driver – one instance to control all hardware, Resource manager – controlling access to file system.