**day24\_107856406\_dsdipt\_sudipto\_6august2025**

**Employee Code:** 107856406

**Login ID:** dsdipt

**Email :** dsdipt@amazon.com

**Name:** Sudipto Das

**Date:** 6 August 2025 (Day 24)

### ***Task 01: Implement Factory Method Design Pattern***

The **Factory Method Pattern** is a **creational design pattern** that provides a **way to create objects** without exposing the creation logic to the client.

Instead of calling new SomeObject() directly, we **delegate that responsibility to a method** (a factory method) that returns the object. This gives you more flexibility when creating objects, especially when there are multiple types or variations.

Imagine we run a **pizza store**  
We have a factory method called createPizza(type).

* If someone asks for "cheese" → you return a CheesePizza
* If someone asks for "pepperoni" → you return a PepperoniPizza

But we never expose how you actually make the pizza. That’s the chef’s (factory method's) job.

**When to use:**

* Have **many related classes** (like different types of pizzas, documents, or buttons).
* Want to **decide the object at runtime** instead of hardcoding it.
* Expect the code to grow or **support new types in the future** (Open-Closed Principle).
* Want to **separate the creation logic from the core logic**.

├── src/

│ └── com/

│ ├── factory/

│ │ ├── AbstractPizzaProduct.java

│ │ ├── ConcreteCheezyPizzaProduct.java

│ │ ├── ConcretePepperoniPizzaProduct.java

│ │ ├── ConcretePizzaFactory.java

│ │ └── FactoryPatternDemo.java

### ***Task 02: Implement Singleton Design Pattern***

The **Singleton Pattern** ensures that a class has **only ONE instance**, and provides a **global access point** to that instance.

One class. One object.

Instead of creating multiple objects with new, you just **ask the class**:

MySingleton obj = MySingleton.getInstance();

We will **always get the same object** no matter how many times you call it.

**When to use:**

* Only need one object of a class across the whole app
* That object needs to be globally accessible
* Want to control resource usage, like memory or file handles
* Need centralized management (like config or service coordination)

├── src/

│ └── com/

│ └── singleton/

│ ├── SingletonConfigManager.java

│ └── SingletonDemo.java

### ***Task 03: Implement Abstract Factory Method Design Pattern***

It's a **factory of factories**.  
It creates **related families of objects** – **without specifying their concrete classes**.

The **abstract factory** knows **how to make all the related products**, and we don’t care about **which exact class** is used.

**When to use:**

* Need to create families of related objects
* Code should be independent of concrete classes
* Want to enforce consistency between objects
* Need scalability (new themes, styles, environments, brands)

├── src/

│ └── com/

│ ├── abstractfactory/

│ │ ├── AbstractFactoryDemo.java

│ │ ├── AbstractBookFactory.java

│ │ ├── ConcreteNoteBookFactory.java

│ │ ├── ConcreteTextBookFactory.java

│ │ ├── AbstractNoteBookProduct.java

│ │ ├── AbstractTextBookProduct.java

│ │ ├── ConcreteLongNoteBookProduct.java

│ │ ├── ConcreteShortNoteBookProduct.java

│ │ ├── ConcreteLongTextBookProduct.java

│ │ ├── ConcreteShortTextBookProduct.java

### ***Task 04: Implement Prototype Method Design Pattern***

It’s a **creational pattern** that lets you **clone** existing objects **instead of creating new ones from scratch**.

**When to use:**

* Need many copies of similar objects.
* Creating an object is expensive (e.g., has lots of setup/config).
* Want to avoid complex constructors.
* Using a registry of prebuilt objects (e.g.,SuperHeroCloneCache).
* Want to customize copies slightly without full rebuild.

├── src/

│ └── com/

│ ├── prototype/

│ │ ├── AbstractSuperHeroProduct.java

│ │ ├── ConcreteBatManProduct.java

│ │ ├── ConcreteIronManProduct.java

│ │ ├── ConcreteSpiderManProduct.java

│ │ ├── SuperHeroCloneCache.java

│ │ └── PrototypeDemo.java

### ***Task 05: Practice below UML diagram***

@startuml

participant Pax as p

actor Actor as a #Green

boundary Boundary as b

control Ctrl as ctrl

database Dbase as db

entity table as t

participant Collection as col

participant Queue as q

autonumber

p -[#red]> a : pax to actor

a -> p: ohh is it!

autonumber 100

b ->> p: boundary to pax

note left: plz make a note on left

ctrl -\ p: control to pax

db \\- p: db to pax

autonumber 50 10

t //-- db: table to db

ctrl ->o db: collection to db

note right: plz make a note

q <->o p: q to pax

Prasunamba -> Batch1 : learning uml

Batch1 -> Prasunamba: ok will learn

@enduml

