Understandings:

**Singleton Pattern – UML Diagram**

📌 **Ensures only one instance of a class exists.**

**Diagram:**

+-----------------+

| Singleton |

+-----------------+

| - instance |

| + GetInstance() |

+-----------------+

**Factory Method Pattern – UML Diagram**

📌 **Creates objects without specifying their exact class.**

+-------------------+

| Product |

+-------------------+

| + Operation() |

+-------------------+

▲

│

+------------------+ +-----------------+

| ConcreteProductA | | ConcreteProductB |

+------------------+ +-----------------+

| + Operation() | | + Operation() |

+------------------+ +-----------------+

▲

│

+------------------+

| Creator |

+------------------+

| + FactoryMethod() |

+------------------+

▲

│

+----------------------+

| ConcreteCreator |

+----------------------+

| + FactoryMethod() |

+----------------------+

Builder Pattern – UML Diagram - Constructs a complex object step by step.

+-----------------+

| Product |

+-----------------+

| - PartA |

| - PartB |

| - PartC |

+-----------------+

▲

│

+-----------------+

| Builder |

+-----------------+

| + BuildPartA() |

| + BuildPartB() |

| + BuildPartC() |

| + GetResult() |

+-----------------+

▲

│

+------------------+ +------------------+

| ConcreteBuilder1 | | ConcreteBuilder2 |

+------------------+ +------------------+

| + BuildPartA() | | + BuildPartA() |

| + BuildPartB() | | + BuildPartB() |

| + GetResult() | | + GetResult() |

+------------------+ +------------------+

▲

│

+-----------------+

| Director |

+-----------------+

| + Construct() |

+-----------------+

Prototype Pattern – UML Diagram Creates objects by copying existing ones.

+-----------------+

| Prototype |

+-----------------+

| + Clone() |

+-----------------+

▲

│

+-----------------+ +-----------------+

| ConcreteProto1 | | ConcreteProto2 |

+-----------------+ +-----------------+

| + Clone() | | + Clone() |

+-----------------+ +-----------------+

Creational Design Patterns - Lab Exercise Assignment

**Task 1: Singleton Pattern**

**Problem Statement:**  
Create a **Logger** class that ensures only one instance of the logger exists. This logger should be able to write logs to a file.

**Implementation Steps:**

1. Create a Logger class that prevents multiple instances.
2. Use a private constructor and a static method to return the single instance.
3. Implement a method LogMessage(string message) that writes to a file.

**Task 2: Factory Method Pattern**

**Problem Statement:**  
Develop a program that generates different types of **notifications** (Email, SMS, Push Notification) using a Factory pattern.

**Implementation Steps:**

1. Create an abstract class/interface INotification.
2. Implement concrete classes: EmailNotification, SMSNotification, PushNotification.
3. Create a NotificationFactory that returns an appropriate instance based on input.

**Task 3: Abstract Factory Pattern**

**Problem Statement:**  
Create an **Automobile Factory** that produces different types of vehicles (Car, Bike) with different brands (e.g., Toyota, Honda).

**Implementation Steps:**

1. Create an IVehicle interface with a method GetDetails().
2. Implement Car and Bike classes that implement IVehicle.
3. Create ToyotaFactory and HondaFactory, both implementing IVehicleFactory.
4. Use an AbstractFactory to create vehicle objects dynamically.

**Task 4: Builder Pattern**

**Problem Statement:**  
Develop a **Meal Builder** system where users can create customized meals with different combinations (Burger, Drink, Fries).

**Implementation Steps:**

1. Create a Meal class to represent the meal order.
2. Implement a MealBuilder class that allows step-by-step construction.
3. Add different meal combos (Veg Meal, Non-Veg Meal).

**Task 5: Prototype Pattern**

**Problem Statement:**  
Design a system where employees can **clone** their profiles instead of manually creating new ones.

**Implementation Steps:**

1. Create an Employee class with properties like Name, Department, Salary.
2. Implement the ICloneable interface in the class.
3. Use the Clone() method to duplicate employee objects instead of manually copying attributes.

**Task 6: Factory Method Pattern – Payment Processing System**

🔹 **Scenario:**  
You are developing an **E-commerce platform** where users can pay using **Credit Card, PayPal, or Google Pay**. Each payment method has its own implementation.

🔹 **Requirements:**

1. Create an interface IPaymentProcessor with a method ProcessPayment(amount).
2. Implement **three concrete classes**: CreditCardPayment, PayPalPayment, and GooglePayPayment.
3. Develop a **PaymentFactory** that returns an appropriate payment processor based on user selection.

**Task 7: Builder Pattern – Online Pizza Ordering System**

🔹 **Scenario:**  
A **pizza delivery service** allows customers to customize their pizza with different sizes, toppings, and crust types.

🔹 **Requirements:**

1. Create a Pizza class with properties: Size, Crust, Toppings.
2. Implement a PizzaBuilder class that constructs a pizza step by step.
3. Add pre-defined **combo pizzas** (Veg Deluxe, Chicken Supreme, etc.).

**Task 8: Prototype Pattern – Employee Profile Cloning**

🔹 **Scenario:**  
A company wants to **duplicate employee profiles** when hiring new employees with similar job roles.

🔹 **Requirements:**

1. Create an Employee class with properties (Name, Role, Salary).
2. Implement **cloning** using the **Prototype Pattern**.
3. Ensure that cloning is **deep copy**, not shallow.

**9. Singleton Pattern – API Request Rate Limiter**

🔹 **Scenario:**  
To prevent excessive API usage, an **API Gateway** should implement a **rate limiter** that ensures users can make only a limited number of requests per minute.

🔹 **Requirements:**

1. Create a RateLimiter class using the **Singleton Pattern**.
2. Implement logic to track requests and restrict access if the limit is exceeded.
3. Store request data using an **in-memory cache** (e.g., **Redis, In-Memory Dictionary**).