Factory Design Pattern

```cpp
// 3. Factory
class NotificationFactory {
public:
    static std::unique_ptr<INotification> create(const std::string& channel) {
        if (channel == "email") {
            return std::make_unique<EmailNotification>();
        } else if (channel == "sms") {
            return std::make_unique<SmsNotification>();
        }
        // In real code, handle this better (maybe return null or use enum).
        throw std::invalid_argument("Unknown channel: " + channel);
    }
};

// 4. Client code
int main() {
    auto emailNotif = NotificationFactory::create("email");
    emailNotif->send("user@example.com", "Welcome to our platform!");

    auto smsNotif = NotificationFactory::create("sms");
    smsNotif->send("+911234567890", "Your OTP is 123456");

    return 0;
}
```

Abstract Factory

```java
// Products
interface Button {
    void render();
}

interface Checkbox {
    void render();
}

// Concrete Products - Light
class LightButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering Light Button");
    }
}

class LightCheckbox implements Checkbox {
    @Override
    public void render() {
        System.out.println("Rendering Light Checkbox");
    }
}
```

```java
// Concrete Products - Dark
class DarkButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering Dark Button");
    }
}

class DarkCheckbox implements Checkbox {
    @Override
    public void render() {
        System.out.println("Rendering Dark Checkbox");
    }
}
```

```
39
40 // Abstract Factory
41 interface UIFactory {
42     Button createButton();
43     Checkbox createCheckbox();
44 }
45
46 // Concrete Factories
47 class LightThemeFactory implements UIFactory {
48     @Override
49     public Button createButton() {
50         return new LightButton();
51     }
52
53     @Override
54     public Checkbox createCheckbox() {
55         return new LightCheckbox();
56     }
57 }
58
59 class DarkThemeFactory implements UIFactory {
60     @Override
61     public Button createButton() {
62         return new DarkButton();
63     }
64
65     @Override
66     public Checkbox createCheckbox() {
67         return new DarkCheckbox();
68     }
69 }
70
```

```
71 // Client
72 public class Main {
73     public static void main(String[] args) {
74         boolean darkMode = true;
75
76         UIFactory factory = darkMode
77                 ? new DarkThemeFactory()
78                 : new LightThemeFactory();
79
80         Button button = factory.createButton();
81         Checkbox checkbox = factory.createCheckbox();
82
83         button.render();
84         checkbox.render();
85     }
86 }
87
```

Singleton

```cpp
class Logger {
private:
    Logger() = default;                         // private ctor
    Logger(const Logger&) = delete;             // no copy
    Logger& operator=(const Logger&) = delete; // no copy assign

public:
    static Logger& getInstance() {
        static Logger instance; // created once, thread-safe in C++11+
        return instance;
    }

    void info(const std::string& msg) {
        std::cout << "[INFO] " << msg << '\n';
    }

    void error(const std::string& msg) {
        std::cerr << "[ERROR] " << msg << '\n';
    }
};


// For usage
Logger::getInstance().info("Server started");
Logger::getInstance().error("Connection failed");

```

Adapter Pattern

```java
interface PaymentProcessor {
    void processPayment(double amountInRupees);
}

// Adaptee
class OldPaymentService {
    public void makePaymentInPaise(long paise) {
        System.out.println("OldPaymentService: paid " + paise + "
paise");
    }
}
```

```java
// Adapter
class OldPaymentAdapter implements PaymentProcessor {
    private final OldPaymentService service;

    public OldPaymentAdapter(OldPaymentService service) {
        this.service = service;
    }

    @Override
    public void processPayment(double amountInRupees) {
        long paise = (long) (amountInRupees * 100);
        service.makePaymentInPaise(paise);
    }
}

// Client
public class Main {
    public static void main(String[] args) {
        OldPaymentService legacy = new OldPaymentService();
        PaymentProcessor processor = new OldPaymentAdapter(legacy);

        processor.processPayment(499.99);
    }
}
```

Decorator

```cpp
class Notification {
public:
    virtual ~Notification() = default;
    virtual void send(const string& to, const string& message) = 0;
};
```

```cpp
class EmailNotification : public Notification {
public:
    void send(const string& to, const string& message) override {
        cout << "[EMAIL] To: " << to << " | Message: " << message << endl;
    }
};
```

```cpp
class NotificationDecorator : public Notification {
protected:
    Notification* wrappee;

public:
    explicit NotificationDecorator(Notification* wrappee)
        : wrappee(wrappee) {}

    void send(const string& to, const string& message) override {
        wrappee->send(to, message);
    }
};
```

```cpp
class LoggingNotification : public NotificationDecorator {
public:
    explicit LoggingNotification(Notification* wrappee)
        : NotificationDecorator(wrappee) {}

    void send(const string& to, const string& message) override {
        cout << "[LOG] Sending to " << to << endl;
        NotificationDecorator::send(to, message);
        cout << "[LOG] Done" << endl;
    }
};
```

```cpp
class PrefixNotification : public NotificationDecorator {
    string prefix;

public:
    PrefixNotification(Notification* wrappee, const string& prefix)
        : NotificationDecorator(wrappee), prefix(prefix) {}

    void send(const string& to, const string& message) override {
        NotificationDecorator::send(to, prefix + ": " + message);
    }
};
```

```cpp
int main() {
    Notification* notification =
        new PrefixNotification(
            new LoggingNotification(
                new EmailNotification()
            ),
            "[IMPORTANT]"
        );

    notification->send("user@example.com", "Complete your profile");

    // Cleanup (important in C++)
    delete notification;

    return 0;
}
```

Strategy

```java
interface PaymentStrategy {
    void pay(int amount);
}

class CardPayment implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("Paying " + amount + " using Card");
    }
}

class UpiPayment implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("Paying " + amount + " using UPI");
    }
}
```

```java
class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void checkout(int amount) {
        if (paymentStrategy == null) {
            throw new IllegalStateException("No payment strategy set");
        }
        paymentStrategy.pay(amount);
    }
}

public class StrategyDemo {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        cart.setPaymentStrategy(new CardPayment());
        cart.checkout(500);

        cart.setPaymentStrategy(new UpiPayment());
        cart.checkout(300);
```

Observer

```java
import java.util.ArrayList;
import java.util.List;

interface Observer {
    void update(String news);
}

interface Subject {
    void addObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}
```

```java
class NewsAgency implements Subject {
    private final List<Observer> observers = new ArrayList<>();
    private String news;

    public void setNews(String news) {
        this.news = news;
        notifyObservers();
    }

    @Override
    public void addObserver(Observer o) {
        observers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(news);
        }
    }
```

```java
class NewsChannel implements Observer {
    private final String name;

    public NewsChannel(String name) {
        this.name = name;
    }

    @Override
    public void update(String news) {
        System.out.println("Channel " + name + " received: " +
news);
    }
}

public class ObserverDemo {
    public static void main(String[] args) {
        NewsAgency agency = new NewsAgency();

        Observer ch1 = new NewsChannel("A");
        Observer ch2 = new NewsChannel("B");

        agency.addObserver(ch1);
        agency.addObserver(ch2);
```

```java
public class ObserverDemo {
    public static void main(String[] args) {
        NewsAgency agency = new NewsAgency();

        Observer ch1 = new NewsChannel("A");
        Observer ch2 = new NewsChannel("B");

        agency.addObserver(ch1);
        agency.addObserver(ch2);

        agency.setNews("Breaking: Design Patterns Made Easy!");
        agency.setNews("Update: Observer pattern in action.");
    }
}

```