Design Patterns (Software Engineering)

Design patterns are **typical solutions** to **common problems** in software design.
They are **templates**, not code, and help you write **clean, maintainable, and reusable code**.

Types of Design Patterns

1. **Creational Patterns**
    - Deal with object creation
    - Hide the complexity of instantiating objects
2. **Structural Patterns**
    - Deal with object composition
    - Help build flexible structures
3. **Behavioral Patterns**
    - Deal with object communication
    - Focus on responsibilities and algorithms

---

1. Creational Patterns

a) Singleton

Ensure only one instance of a class exists.

Use when:

- You need one global instance (e.g. config manager, logger)

C++ Example:

```
class Singleton {
private:
   static Singleton* instance;
   Singleton() {} // private constructor

public:
   static Singleton* getInstance() {
      if (!instance)
         instance = new Singleton();
      return instance;
   }
};
Singleton* Singleton::instance = nullptr;
```

---

b) Factory Method

Creates objects without exposing instantiation logic.

Use when:

- You want to delegate the creation logic to subclasses

C++ Example:

```
class Animal {
public:
   virtual void sound() = 0;
};

class Dog : public Animal {
public:
   void sound() override { std::cout << "Bark\n"; }
};

class Cat : public Animal {
public:
   void sound() override { std::cout << "Meow\n"; }
};

class AnimalFactory {
public:
   static Animal* createAnimal(std::string type) {
```

```
        if (type == "dog") return new Dog();
        if (type == "cat") return new Cat();
        return nullptr;
    }
};
```

---

2. Structural Patterns

a) Adapter

Convert one interface to another.

Use when:

- You want to reuse an existing class with an incompatible interface

C++ Example:

```
class OldPrinter {
public:
    void oldPrint() { std::cout << "Old print\n"; }
};

class Printer {
public:
    virtual void print() = 0;
};

class Adapter : public Printer {
    OldPrinter* old;
public:
    Adapter(OldPrinter* o) : old(o) {}
    void print() override { old->oldPrint(); }
};
```

---

3. Behavioral Patterns

a) Observer

Notify all observers when the subject changes.

Use when:

- You have a system where one change should affect many objects

C++ Example:

```
class Observer {
public:
    virtual void update(int value) = 0;
};

class Subject {
    std::vector<Observer*> observers;
    int state;
public:
    void attach(Observer* obs) { observers.push_back(obs); }

    void setState(int s) {
        state = s;
        for (auto obs : observers)
            obs->update(state);
    }
};

class ConcreteObserver : public Observer {
public:
    void update(int value) override {
        std::cout << "Value changed to: " << value << "\n";
    }
};
```

Practice

Implement these patterns:

- Logger (Singleton)
- Shape Factory (Factory)

- Voltage Adapter (Adapter)
- News Feed (Observer)

Questions

- When should you avoid Singleton?
- How does Factory improve maintainability?
- What's the difference between Adapter and Inheritance?

Now Practice These 4 Design Patterns

Write each one from scratch in C++. Keep code modular.

---

1. Singleton Practice

Task: Create a `Logger` class

- Only one instance allowed
- Method: `log(string msg)` that prints the message

Challenge:

- Prevent copy constructor and assignment

---

2. Factory Method Practice

Task: Create a `ShapeFactory`

- Input: "circle", "rectangle", "square"
- Output: pointer to `Shape`
- Each shape has a method `draw()`

```
Shape* s = ShapeFactory::createShape("circle");
s->draw();
```

---

3. Adapter Pattern Practice

Scenario: Old motor uses `rotateClockwise()`
New system expects `moveForward()`

Task:

- Create `MotorAdapter` that wraps `OldMotor`
- Call `moveForward()` internally calls `rotateClockwise()`

---

4. Observer Pattern Practice

Task:

- Class: `TemperatureSensor` (Subject)
- Observers: `Display`, `FanController`
- When temp changes, both get notified with new value

Optional Challenge (Real-World Embedded Style)

Embedded Factory:

- Create `SensorFactory` that returns `TemperatureSensor`, `IRSensor`, or `UltrasonicSensor` objects

- Each has a `read()` method

Observer Embedded:

- When sensor value crosses threshold, alert multiple modules like LCD, Buzzer, and Logger

---