Semana 4 y 5 – Patrones Creacionales: Singleton, Factory, Builder

OBJETIVO

Fortalecer el dominio de los patrones creacionales en la Programación Orientada a Objetos mediante implementaciones en **C++ y Python**, destacando la importancia de la gestión de instancias, la reutilización de código y la escalabilidad en sistemas complejos.

MARCO TEÓRICO

- **Singleton**: asegura una única instancia global para un recurso compartido (ejemplo: configuración de sistema, conexión a base de datos).
- **Factory Method y Abstract Factory**: centralizan la creación de objetos, permitiendo independencia de clases concretas.
- **Builder**: simplifica la construcción de objetos complejos dividiéndola en pasos, útil en objetos con múltiples configuraciones (ejemplo: creación de un "Documento PDF" o "Vehículo").

Los patrones creacionales son esenciales en sistemas grandes, ya que desacoplan la lógica de creación de la lógica de negocio.

CÓDIGO

Ejemplo 1: Singleton en C++

```cpp
#include <iostream>
using namespace std;

class Config {
private:
    static Config* instance;
    Config() {}  // Constructor privado
public:
    static Config* getInstance() {
        if (!instance) instance = new Config();
        return instance;
    }
    void showMessage() { cout << "Configuración global cargada.\n"; }
};
Config* Config::instance = nullptr;

int main() {
    Config* obj1 = Config::getInstance();
    Config* obj2 = Config::getInstance();
    obj1->showMessage();
    cout << "¿Son iguales? " << (obj1 == obj2) << endl;
}
```

Ejemplo 2: Factory en Python

```
class Transporte:
    def entregar(self):
        pass

class Camion(Transporte):
    def entregar(self):
        return "Entrega por carretera"

class Barco(Transporte):
    def entregar(self):
        return "Entrega por mar"

class Factory:
    @staticmethod
    def get_transporte(tipo):
        if tipo == "camion": return Camion()
        elif tipo == "barco": return Barco()

t = Factory.get_transporte("barco")
print(t.entregar())
```

Ejemplo 3: Builder en Python

```
class Computadora:
    def __init__(self):
        self.cpu = None
        self.ram = None
        self.gpu = None

class Builder:
    def __init__(self):
        self.computadora = Computadora()

    def add_cpu(self, cpu):
        self.computadora.cpu = cpu
        return self

    def add_ram(self, ram):
        self.computadora.ram = ram
        return self

    def add_gpu(self, gpu):
        self.computadora.gpu = gpu
        return self

    def build(self):
        return self.computadora

pc = Builder().add_cpu("Intel i9").add_ram("32GB").add_gpu("RTX
4090").build()
print(vars(pc))
```

DESARROLLO

1. Implementar **Singleton** en C++ y probar que se crea una única instancia.
2. Usar **Factory** en Python para simular diferentes medios de transporte.
3. Crear un **Builder** en Python que arme un "combo de fast food" (hamburguesa, bebida, papas fritas).

TRABAJO DE INVESTIGACIÓN

- Investigar cómo **Django ORM** usa Factory para construir modelos.
- Identificar cómo **Spring Boot** usa Singleton en el manejo de Beans.
- Comparar uso de Builder en bibliotecas de UI como **Tkinter** y **Qt**.

RESULTADOS

El estudiante presentará código documentado en repositorios (GitHub/GitLab), un informe comparativo y diagramas UML de los tres patrones.

Semana 6 – Patrones Estructurales: Adapter, Decorator, Composite

OBJETIVO

Aplicar patrones estructurales en C++ y Python para mejorar la modularidad, integración de componentes y flexibilidad de software.

MARCO TEÓRICO

- **Adapter**: convierte una interfaz existente en otra compatible sin modificarla.
- **Decorator**: extiende funcionalidades en tiempo de ejecución.
- **Composite**: maneja jerarquías de objetos (ej. menús en interfaces gráficas).

Estos patrones facilitan el mantenimiento de proyectos a largo plazo y promueven el **principio abierto/cerrado (SOLID)**.

CÓDIGO

Ejemplo 1: Adapter en Python

```python
class EnchufeEuropeo:
    def conectar(self):
        return "220V conectado"

class Adaptador:
    def __init__(self, enchufe):
        self.enchufe = enchufe
    def conectar(self):
        return f"Adaptado a 110V -> {self.enchufe.conectar()}"

e = EnchufeEuropeo()
a = Adaptador(e)
print(a.conectar())
```

Ejemplo 2: Decorator en Python

```python
def log(func):
    def envoltura(*args, **kwargs):
        print("Ejecutando función:", func.__name__)
        return func(*args, **kwargs)
    return envoltura

@log
def saludar(nombre):
    return f"Hola {nombre}"

print(saludar("Aldo"))
```

Ejemplo 3: Composite en C++

```cpp
#include <iostream>
#include <vector>
```

```cpp
using namespace std;

class Figura {
public:
    virtual void dibujar() = 0;
};

class Circulo : public Figura {
public:
    void dibujar() override { cout << "Círculo\n"; }
};

class Grupo : public Figura {
    vector<Figura*> figuras;
public:
    void add(Figura* f) { figuras.push_back(f); }
    void dibujar() override {
        for (auto f : figuras) f->dibujar();
    }
};

int main() {
    Circulo c1, c2;
    Grupo g;
    g.add(&c1);
    g.add(&c2);
    g.dibujar();
}
```

DESARROLLO

1. Construir un **adaptador de pagos** que convierta dólares a soles.
2. Aplicar un **decorator** para añadir validación extra a funciones matemáticas.
3. Usar **Composite** para crear una estructura jerárquica de carpetas y archivos.

TRABAJO DE INVESTIGACIÓN

- Revisar cómo **Qt** y **Java Swing** aplican Composite.
- Investigar Decorator en **Flask (Python)** para manejo de rutas.

RESULTADOS

Informe con ejemplos ejecutados, diagrama UML de cada patrón y aplicaciones reales en frameworks.

Semana 7 – Patrones de Comportamiento: Observer, Strategy, Command

OBJETIVO

Implementar patrones de comportamiento que gestionan la comunicación entre objetos, optimizando interacción y desacoplamiento.

MARCO TEÓRICO

- **Observer**: mantiene sincronización entre objetos (ej. sistemas de notificación).
- **Strategy**: selecciona algoritmos dinámicamente.
- **Command**: encapsula operaciones en objetos independientes, favoreciendo el undo/redo.

CÓDIGO

Ejemplo 1: Observer en Python

```python
class Observador:
    def actualizar(self, mensaje):
        print("Notificado:", mensaje)

class Sujeto:
    def __init__(self):
        self.obs = []
    def registrar(self, o):
        self.obs.append(o)
    def notificar(self, mensaje):
        for o in self.obs:
            o.actualizar(mensaje)

s = Sujeto()
o1, o2 = Observador(), Observador()
s.registrar(o1); s.registrar(o2)
s.notificar("Se actualizó el sistema.")
```

Ejemplo 2: Strategy en C++

```cpp
#include <iostream>
using namespace std;

class Estrategia {
public:
    virtual void ejecutar() = 0;
};

class EstrategiaA : public Estrategia {
public:
    void ejecutar() override { cout << "Algoritmo A\n"; }
};

class EstrategiaB : public Estrategia {
public:
```

```
    void ejecutar() override { cout << "Algoritmo B\n"; }
};

class Contexto {
    Estrategia* estrategia;
public:
    Contexto(Estrategia* e) : estrategia(e) {}
    void setEstrategia(Estrategia* e) { estrategia = e; }
    void operar() { estrategia->ejecutar(); }
};

int main() {
    EstrategiaA a; EstrategiaB b;
    Contexto ctx(&a);
    ctx.operar();
    ctx.setEstrategia(&b);
    ctx.operar();
}
```

Ejemplo 3: Command en Python

```
class Command:
    def ejecutar(self):
        pass

class ImprimirCommand(Command):
    def ejecutar(self):
        print("Imprimiendo documento...")

class GuardarCommand(Command):
    def ejecutar(self):
        print("Guardando documento...")

class Invocador:
    def __init__(self):
        self.historial = []
    def ejecutar(self, comando):
        comando.ejecutar()
        self.historial.append(comando)

i = Invocador()
i.ejecutar(ImprimirCommand())
i.ejecutar(GuardarCommand())
```

DESARROLLO

1. Crear un sistema de **notificaciones Observer** para usuarios de un chat.
2. Usar **Strategy** para elegir diferentes algoritmos de ordenamiento (burbuja, quicksort).
3. Implementar **Command** para un editor de texto con comandos “guardar” y “deshacer”.

TRABAJO DE INVESTIGACIÓN

- Investigar cómo **Android LiveData** implementa Observer.

- Analizar Strategy en librerías de IA (ej. selección de optimizadores en PyTorch).
- Ver cómo Command se usa en videojuegos para gestionar acciones del jugador.

RESULTADOS

Un informe escrito, repositorios con los tres patrones implementados, y presentación oral sobre su aplicación en sistemas reales.