UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA

Facultad de Ciencias de la Ingeniería y Tecnología

Unidad Valle de las Palmas

Patrones de Software

Meta 3.5 Desarrollar un software para los patrones de diseño: plantilla, estado, iterador y composición, que sean utilizados en casos prácticos.

José Humberto Moreno Mejía

OCTUBRE 2024

El objetivo de este programa es mostrar la implementación y aplicación práctica de cuatro patrones de diseño: Plantilla, Estado, Iterador y Composición. A través de ejemplos concretos, se ilustra cómo cada patrón mejora la estructura del código, la legibilidad y la gestión de cambios. El uso de estos patrones fomenta buenas prácticas en el desarrollo de software orientado a objetos, permitiendo una mejor organización y reutilización del código. También se incluyen pruebas unitarias para asegurar el correcto funcionamiento de cada patrón.

1. **Patrón Plantilla (Template Method)**

```python
# Patrón Plantilla
class Documento:
    def crear_documento(self):
        self.crear_titulo()
        self.crear_cuerpo()
        self.crear_firma()

    def crear_titulo(self):
        pass

    def crear_cuerpo(self):
        pass

    def crear_firma(self):
        pass


class Carta(Documento):
    def crear_titulo(self):
        print("Título: Carta Formal")

    def crear_cuerpo(self):
        print("Cuerpo: Estimado Sr. Perez...")

    def crear_firma(self):
        print("Firma: Atentamente, José")
```

El **Patrón Plantilla** define un esquema general para crear un documento en la clase base Documento.
Subclases como Carta personalizan las partes del proceso, como el título, cuerpo y firma, pero siguen el mismo flujo de ejecución. La clase base asegura que todas las subclases usen el mismo orden para construir el documento.

Salida:

```
Ejemplo Plantilla:
Título: Carta Formal
Cuerpo: Estimado Sr. Perez...
Firma: Atentamente, José
```

El patrón se muestra en cómo el método crear_documento() sigue una estructura fija, pero el contenido es específico de la subclase.

2. **Patrón Estado (State)**

```python
class Estado:
    def manejar(self, dispositivo):
        pass


class EstadoActivo(Estado):
    def manejar(self, dispositivo):
        print("Dispositivo activado.")
        dispositivo.cambiar_estado(EstadoInactivo())


class EstadoInactivo(Estado):
    def manejar(self, dispositivo):
        print("Dispositivo desactivado.")
        dispositivo.cambiar_estado(EstadoActivo())


class Dispositivo:
    def __init__(self):
        self.estado = EstadoInactivo()

    def cambiar_estado(self, estado):
        self.estado = estado

    def presionar_boton(self):
        self.estado.manejar(self)
```

El **Patrón Estado** permite cambiar el comportamiento de un objeto (Dispositivo) en función de su estado actual.

Dispositivo tiene dos estados: EstadoActivo y EstadoInactivo. Cada vez que se presiona el botón, el dispositivo cambia de estado y ajusta su comportamiento. Este patrón permite manejar diferentes situaciones sin tener condicionales complejos.

Salida:

```
Ejemplo Estado:
Dispositivo desactivado.
Dispositivo activado.
```

La salida refleja cómo cambia el estado del dispositivo al presionar el botón, usando el patrón **Estado** para controlar el flujo.

3. **Patrón Iterador (Iterator)**

```python
# Patrón Iterador
class IteradorNumeros:
    def __init__(self, numeros):
        self.numeros = numeros
        self.indice = 0

    # El iterador debe tener el método __iter__
    def __iter__(self):
        return self

    # Y el método __next__
    def __next__(self):
        if self.indice < len(self.numeros):
            numero = self.numeros[self.indice]
            self.indice += 1
            return numero
        else:
            raise StopIteration
```

```python
# Ejemplo del Patrón Iterador
print("\nEjemplo Iterador:")
coleccion = [10, 20, 30]
iterador = IteradorNumeros(coleccion)

for numero in iterador:
    print(numero)
```

El **Patrón Iterador** permite recorrer una colección de manera secuencial sin exponer su estructura interna. IteradorNumeros implementa los métodos __iter__() y __next__(), lo que permite utilizar un bucle for para recorrer la lista de números (coleccion). El iterador devuelve un elemento a la vez y lanza StopIteration cuando ya no hay más elementos.

Salida:

```
Ejemplo Iterador:
10
20
30
```

El patrón **Iterador** se aplica al recorrer la lista de números sin necesidad de gestionar manualmente los índices, delegando esa tarea al objeto iterador.

4. **Patrón Composición (Composite)**

```python
# Patrón Composición
class Grafico:
    def dibujar(self):
        pass


class Circulo(Grafico):
    def dibujar(self):
        print("Dibujar un círculo.")


class Rectangulo(Grafico):
    def dibujar(self):
        print("Dibujar un rectángulo.")


class GrupoGraficos(Grafico):
    def __init__(self):
        self.graficos = []

    def agregar(self, grafico):
        self.graficos.append(grafico)

    def dibujar(self):
        for grafico in self.graficos:
            grafico.dibujar()
```

```python
# Ejemplo del Patrón Composición
print("\nEjemplo Composición:")
grupo = GrupoGraficos()
circulo = Circulo()
rectangulo = Rectangulo()

grupo.agregar(circulo)
grupo.agregar(rectangulo)
grupo.dibujar()
```

El **Patrón Composición** permite tratar objetos individuales y compuestos de la misma forma. GrupoGraficos puede contener varios objetos gráficos (Circulo, Rectangulo), y al llamar a dibujar(), ejecuta el método dibujar() de cada uno de ellos. Esto permite componer gráficos complejos a partir de objetos más simples.

Salida:

```
Ejemplo Composición:
Dibujar un círculo.
Dibujar un rectángulo.
Dibujar un círculo.
Dibujar un rectángulo.
```

La salida refleja cómo el grupo de gráficos gestiona múltiples objetos como un conjunto, usando el patrón **Composición** para tratarlos uniformemente.

Conclusión

Este programa demuestra la utilidad de los patrones de diseño en el desarrollo de software. Al implementar los patrones de Plantilla, Estado, Iterador y Composición, se logra una mayor claridad y organización del código, facilitando su mantenimiento y evolución. Las pruebas unitarias aseguran que cada patrón funcione correctamente, lo que refuerza la confiabilidad del software. En conjunto, estos elementos contribuyen a mejorar la calidad y eficiencia en el desarrollo de aplicaciones orientadas a objetos.