PATRONES DE DISEÑO CREACIONALES

MARTÍNEZ COSIO JOSÉ ALFREDO

Aarón Hernández García

PROYECTO FINAL PATRONES DE DISEÑO

(Sistema de Gestión de Vehículos)

05/12/2024

Este proyecto implementa un sistema de gestión de vehículos sencillo utilizando patrones de diseño para garantizar una solución escalable y bien estructurada. Se emplean los patrones **Singleton** y **Factory Method**:

- **Singleton:** Para gestionar una instancia única del gestor de vehículos, que administra una lista centralizada de los vehículos creados.
- **Factory Method:** Para simplificar la creación de objetos de diferentes tipos de vehículos, como automóviles, motocicletas y camiones.

El sistema está diseñado para ejecutarse en consola y permite al usuario:

1. Crear vehículos de diferentes tipos.
2. Agregarlos a una lista centralizada.
3. Mostrar los detalles de todos los vehículos registrados.

Código →

```python
from abc import ABC, abstractmethod


#  --> Clase base: Vehículo <--
class Vehiculo(ABC):
    def __init__(self, marca, modelo, año):
        self.marca = marca
        self.modelo = modelo
        self.año = año

    @abstractmethod
    def detalles(self):
        pass


#  --> Clases derivadas <--
class Automovil(Vehiculo):
    def detalles(self):
        return f"Automóvil: {self.marca} {self.modelo}, Año: {self.año}"


class Motocicleta(Vehiculo):
    def detalles(self):
        return f"Motocicleta: {self.marca} {self.modelo}, Año: {self.año}"


class Camion(Vehiculo):
    def detalles(self):
        return f"Camión: {self.marca} {self.modelo}, Año: {self.año}"
```

1. **Vehículo es una clase base abstracta:**
    - ABC (Abstract Base Class): Es una clase que no puede instanciarse directamente. Se utiliza como plantilla para crear otras clases derivadas.
    - Especificada al importar ABC del módulo abc en Python.
2. **Atributos comunes:**
    - marca, modelo, y año son propiedades comunes a todos los vehículos, por lo que se definen en el constructor (__init__).

3. **Método abstracto:**
    - @abstractmethod: Indica que todas las clases derivadas de Vehículo deben implementar este método.
    - El método detalles() no tiene implementación aquí porque su comportamiento depende del tipo de vehículo específico.

Clase: VehiculoFactory

La clase VehiculoFactory implementa el patrón de diseño Factory Method, que se utiliza para centralizar y simplificar la creación de objetos de diferentes tipos de vehículos.

```python
# --> Factory Method: Creamos Vehiculos según el tipo que deseemos agregar. <--
class VehiculoFactory:
    @staticmethod
    def crear_vehiculo(tipo, marca, modelo, año):
        tipo = tipo.lower()
        if tipo == "automovil":
            return Automovil(marca, modelo, año)
        elif tipo == "motocicleta":
            return Motocicleta(marca, modelo, año)
        elif tipo == "camion":
            return Camion(marca, modelo, año)
        else:
            raise ValueError(f"Tipo de vehículo '{tipo}' no es válido.")
```

1. **Método crear_vehiculo**
    - Es un **método estático**:
        - Declarado con @staticmethod, lo que significa que puede ser llamado directamente desde la clase, sin necesidad de crear una instancia de VehiculoFactory.
    - **Parámetros de entrada**:
        - tipo: Tipo de vehículo a crear (por ejemplo, "automóvil", "motocicleta", "camión").
        - marca, modelo, año: Detalles específicos del vehículo.
    - **Proceso de creación**:
        - Se evalúa el valor de tipo:

        - Si es "automóvil", crea y devuelve una instancia de la clase Automóvil.
        - Si es "motocicleta", crea y devuelve una instancia de la clase Motocicleta.
        - Si es "camión", crea y devuelve una instancia de la clase Camión.
        - Si el valor no coincide con ninguno de los tipos, lanza una excepción ValueError.
- **Conversión a minúsculas**:
    - Convierte el tipo a minúsculas (tipo.lower()) para evitar problemas si el usuario escribe "AUTOMOVIL" o "Automovil", por ejemplo.

2. **Uso del patrón Factory Method**
    - El **Factory Method** abstrae la lógica de creación de objetos:
        - En lugar de instanciar directamente las clases (Automovil, Motocicleta, Camion), el usuario solo necesita llamar a crear_vehiculo() y proporcionar el tipo y atributos.
        - Esto hace que el código sea más flexible y fácil de mantener.

El patrón **Singleton** se utiliza para asegurar que solo exista una única instancia de la clase GestorVehiculos durante la ejecución del programa. Esto es útil cuando se necesita una única fuente de verdad para gestionar los datos, en este caso, los vehículos registrados.

```python
#  --> Singleton: Gestor de Vehículos <--
class GestorVehiculos:
    _instancia = None

    def __new__(cls, *args, **kwargs):
        if not cls._instancia:
            cls._instancia = super().__new__(cls, *args, **kwargs)
            cls._instancia._vehiculos = []
        return cls._instancia

    def agregar_vehiculo(self, vehiculo):
        self._vehiculos.append(vehiculo)
        print(f"Vehículo agregado: {vehiculo.detalles()}")

    def mostrar_vehiculos(self):
        print("\nListado de Vehículos:")
        for vehiculo in self._vehiculos:
            print(vehiculo.detalles())
```

1. **Atributo de Clase: _instancia**

   _instancia = None

   - **_instancia**:
     - Es un atributo de clase que almacena la única instancia creada de GestorVehiculos.
     - Se inicializa en None para indicar que no existe ninguna instancia al principio.
   - **Propósito de __new__:**
     - Este método se llama antes del constructor (__init__) para controlar la creación de instancias.
     - Aquí se asegura que solo se cree una única instancia de GestorVehiculos.

- **Lógica en __new__:**
  1. Comprueba si _instancia es None.
     - Si es None, significa que no existe una instancia y se crea una nueva usando super().__new__(cls, *args, **kwargs).
     - Luego, se inicializa la lista _vehiculos para almacenar los objetos vehículo.
  2. Si ya existe una instancia, simplemente la devuelve, evitando que se cree una nueva.

2. **Método __new__**

- **Propósito de __new__:**
  - Este método se llama antes del constructor (__init__) para controlar la creación de instancias.
  - Aquí se asegura que solo se cree una única instancia de GestorVehiculos.
- **Lógica en __new__:**
  1. Comprueba si _instancia es None.
     - Si es None, significa que no existe una instancia y se crea una nueva usando super().__new__(cls, *args, **kwargs).
     - Luego, se inicializa la lista _vehiculos para almacenar los objetos vehículo.
  2. Si ya existe una instancia, simplemente la devuelve, evitando que se cree una nueva.

La función **main()** orquesta el funcionamiento del sistema al combinar los patrones **Singleton** y **Factory Method** implementados previamente. Es el punto de entrada principal del programa cuando se ejecuta.

```python
def main():
    #  --> Crear el gestor de vehículos (Singleton) <--
    gestor = GestorVehiculos()

    # --> Crear vehículos usando Factory <--
    auto = VehiculoFactory.crear_vehiculo("automovil", "Toyota", "Corolla", 2020)
    moto = VehiculoFactory.crear_vehiculo("motocicleta", "Yamaha", "R6", 2018)
    camion = VehiculoFactory.crear_vehiculo("camion", "Volvo", "FH16", 2022)

    #  --> Agregar vehículos al gestor <--
    gestor.agregar_vehiculo(auto)
    gestor.agregar_vehiculo(moto)
    gestor.agregar_vehiculo(camion)

    #  --> Mostrar los vehículos registrados <--
    gestor.mostrar_vehiculos()


if __name__ == "__main__":
    main()
```

1. **Crear el gestor de vehículos (Singleton)**
    - **Se crea una instancia de GestorVehiculos:**
        - Debido al patrón Singleton, esta instancia será la única que exista en todo el programa.
        - Se usará para gestionar los vehículos creados.
2. **Crear vehículos usando el Factory Method**
    - **Uso de la fábrica (VehiculoFactory):**
        - Se llama al método estático crear_vehiculo() para crear objetos de diferentes tipos (Automovil, Motocicleta, Camion).
        - Los detalles específicos (marca, modelo, año) se pasan como argumentos al método.

- **Proceso subyacente:**
    - El Factory Method determina qué tipo de objeto crear basándose en el parámetro tipo.
    - Devuelve la instancia correspondiente.

3. **Agregar vehículos al gestor**

- **Se llaman a los métodos del gestor (GestorVehiculos) para agregar los vehículos creados.**
- **agregar_vehiculo():**
    - Recibe un objeto de tipo vehículo.
    - Lo almacena en la lista interna _vehiculos.
    - Imprime un mensaje indicando que el vehículo fue agregado exitosamente.

4. **Mostrar los vehículos registrados**

- **Llama al método mostrar_vehiculos() del gestor:**
    - Recorre la lista interna _vehiculos.
    - Imprime los detalles de cada vehículo utilizando el método detalles() de cada objeto.

Resultado en Terminal →