Patrones de diseño

Integrantes:
ARCIA QUINTERO CRISTIAN DAVID
SANCHEZ MORA JUAN JOSE
SAMACA GONZALEZ DANIEL MAURICIO
SOLANO GARCES JOSE ALEJANDRO

Proyecto: My building App
Grupo: LUDOPATAS

1. Introducción

Dentro del desarrollo de software, los **patrones de diseño** son soluciones comprobadas que permiten resolver problemas recurrentes en la arquitectura y organización del código.
Su uso mejora la **mantenibilidad, escalabilidad y reutilización** del sistema, además de garantizar una estructura coherente entre los distintos módulos del proyecto:

- El patrón **Singleton**, implementado directamente sobre la gestión de base de datos
- El patrón **State**, implementado directamente para manejar los roles que tienen los usuarios.

El propósito principal es demostrar cómo estos patrones pueden **integrarse naturalmente**, optimizando tanto el rendimiento como la claridad arquitectónica del proyecto **BuildingAppProject**.

2. Contexto del Proyecto

El proyecto BuildingAppProject tiene como objetivo gestionar la información de un conjunto residencial, incluyendo **usuarios, apartamentos, reservas, asambleas, zonas comunes** y más.

La estructura del sistema se basa en el patrón **Modelo-Vista-Controlador (MVC)** adaptado a Django (Modelo-Vista-Template), en donde las vistas (views.py) controlan la interacción con los modelos (models.py) y las plantillas HTML (templates/).

3. Patrón Singleton: Justificación y Aplicación

3.1. Descripción del patrón

El patrón **Singleton** garantiza que **solo exista una instancia única de una clase** en todo el sistema, y que esta sea accesible de forma global.
Se utiliza comúnmente para manejar recursos compartidos, como conexiones a base de datos, registros de configuración o controladores de logs.

3.2. Por qué usar Singleton en Django

Aunque Django ya implementa de forma interna un sistema de conexión persistente con la base de datos (a través de su ORM y el archivo settings.py), en algunos contextos puede ser necesario **reforzar este control de acceso único**, especialmente cuando se realizan operaciones específicas o consultas repetitivas fuera del ORM (por ejemplo, para auditorías o reportes).

En este proyecto, la vista views.py puede beneficiarse de un **gestor Singleton** que controle el acceso a las operaciones sobre la base de datos, evitando la creación innecesaria de múltiples objetos de consulta.

4. Implementación del Singleton en views.py

La siguiente adaptación del archivo core/views.py aplica el patrón Singleton **sin crear nuevos archivos**, utilizando únicamente el código existente:

```
from django.shortcuts import render
from django.http import HttpResponse
from .models import Usuario

# Clase Singleton para manejar acceso centralizado
class UsuarioManager:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def listar_todos(self):
        return Usuario.objects.all()
```

```
# Vista que utiliza el Singleton
def listar_usuarios(request):
    manager = UsuarioManager()
    usuarios = manager.listar_todos()
    contexto = {'usuarios': usuarios}
    return render(request, 'core/listar_usuarios.html', contexto)

def hola(request):
    return HttpResponse("Hola Django, la app core está funcionando ")
```

4.1. Explicación paso a paso

1. **Definición de la clase UsuarioManager:**
   Centraliza las operaciones sobre el modelo Usuario.
   Esta clase se instancia solo una vez gracias al método especial __new__().
2. **Uso del atributo _instance:**
   Permite almacenar una referencia única al objeto creado. Si la clase ya fue instanciada, simplemente devuelve la misma instancia.
3. **Método listar_todos:**
   Realiza la consulta ORM Usuario.objects.all() encapsulada dentro del Singleton.
4. **En la vista listar_usuarios:**
   Se obtiene la instancia del Singleton (manager = UsuarioManager()), garantizando que todas las consultas al modelo Usuario compartan la misma referencia administradora.
5. **Ventajas obtenidas:**
   - Control único de acceso a los datos del modelo.
   - Reducción de sobrecarga al evitar múltiples instancias.
   - Facilita la extensión futura (por ejemplo, para logging, caché o estadísticas de acceso).

5. Singleton implícito en settings.py

Además del Singleton implementado en las vistas, **Django ya utiliza internamente el patrón Singleton en su sistema de configuración**, particularmente en el archivo settings.py.

Cada vez que se ejecuta el proyecto, Django carga el módulo de configuración y lo mantiene en memoria como una **instancia única global**, accesible mediante:

```
from django.conf import settings
print(settings.DATABASES)
```

Esto demuestra que el propio framework garantiza una única instancia de configuración (Singleton natural).

Por tanto, al aplicar el patrón en las vistas, se refuerza esta idea: tanto la **configuración general** como el **acceso a datos** se gestionan de manera controlada y centralizada.

6. Segundo patrón: State

6.1. Descripción del patrón

El patrón State permite que un objeto cambie su comportamiento cuando cambia su estado interno, delegando la lógica a clases especializadas que representan cada estado posible.
En lugar de manejar múltiples condiciones (if/elif) dentro de una sola clase, el patrón distribuye los comportamientos específicos en clases de estado, haciendo que el objeto principal sea más limpio, flexible y fácil de mantener.

6.2. Escenario de aplicación en *BuildingAppProject*

En este proyecto, el patrón State puede aplicarse al manejo de los diferentes roles de usuario dentro del conjunto residencial (Administrador, Propietario, Residente).

En lugar de que el modelo Usuario maneje múltiples condiciones para determinar su comportamiento según el rol, cada tipo de rol puede representarse como un estado con su propia lógica y responsabilidades.

Por ejemplo:

```
# core/states.py
class EstadoRol:
    def get_permission(self):
        raise NotImplementedError("Debe implementarse en la subclase.")

class Administrador(EstadoRol):
    def get_permission(self):
        return "Acceso total a la administración del sistema."

class Propietario(EstadoRol):
    def get_permission(self):
        return "Acceso a la gestión de su propiedad y comunicación con
administración."

class Residente(EstadoRol):
    def get_permission(self):
        return "Acceso a reservas, notificaciones y áreas comunes."
```

Y en el modelo Usuario se integraría el patrón así:

```
from core.states import Administrador, Propietario, Residente

class Usuario(models.Model):
    nombre = models.CharField(max_length=100)
    correo = models.EmailField()
    contrasena = models.CharField(max_length=100)
    rol = models.CharField(max_length=20)

    _estado_rol=None

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self._estado_rol = self.get_state()

    def  get_state(self):
        state = {
            'Admin': Administrador(),
            'Propietario': Propietario(),
            'Residente': Residente()
        }
        return state[self.rol]

    def get_permission(self):
        return self._estado_rol.get_permission()
```

De esta forma, el objeto Usuario delegará el comportamiento al estado actual según su rol, sin condicionales extensos dentro de su lógica principal.

6.3. Beneficios

- Evita el uso de múltiples condicionales (if o match) para determinar el comportamiento del usuario según su rol.
- Favorece la extensibilidad: se pueden añadir nuevos roles simplemente creando nuevas clases de estado, sin modificar la lógica existente del modelo Usuario.
- Cumple con el principio Open/Closed (abierto a extensión, cerrado a modificación).
- Aísla los comportamientos específicos de cada tipo de usuario, facilitando su mantenimiento y pruebas unitarias.
- Promueve la coherencia entre el estado del usuario y las acciones que puede ejecutar dentro del sistema