PATRONEES DE DISEÑO PROYECTO

PRESENTADO POR:

DAVID ALEJANDRO PEÑALOSA VAZQUEZ

BRIAN STEVEN CUBILLOS CUBILLOS

KEVIN STEVEN ORTIZ VALLEJO

DANIELA MARTIN ORTIZ

INGENIERIA DE SISTEMAS Y COMPUTACIÓN

UNIVERSIDAD DE CUNDINAMARCA

OCTAVO SEMESTRE

FUSAGASUGÁ

2024

Introducción

Los patrones de diseño son enfoques estandarizados para resolver problemas recurrentes en el desarrollo de software. Se popularizaron con el libro "Design Patterns: Elements of Reusable Object-Oriented Software" de los autores conocidos como el **Gang of Four**. Estos patrones permiten crear sistemas que sean robustos, extensibles y mantenibles. Existen tres categorías principales:

- **Patrones Creacionales**: Enfocados en la creación de objetos, permitiendo un mayor control sobre el proceso de instanciación.
- **Patrones Estructurales**: Facilitan la composición de clases y objetos para formar estructuras más complejas.
- **Patrones de Comportamiento**: Definen cómo los objetos interactúan y se comunican entre sí.

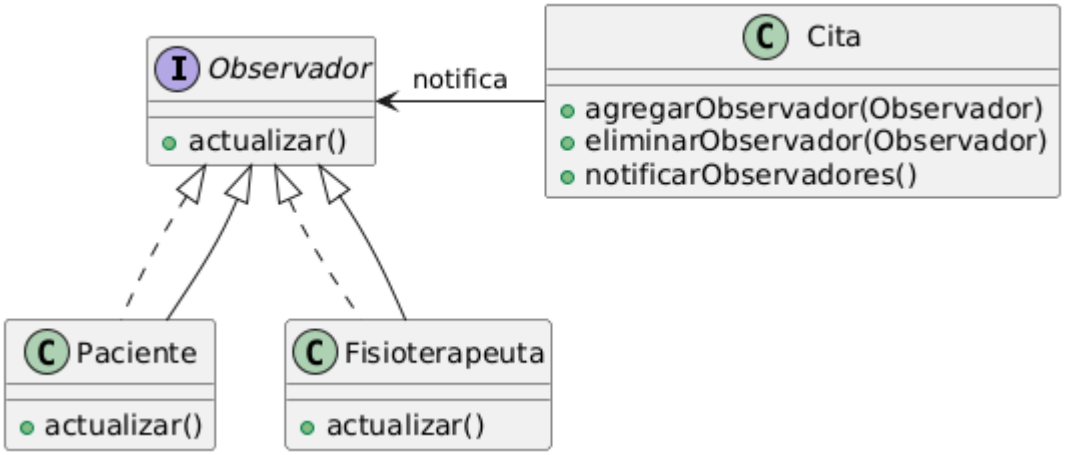
El patrón **Observador** y el **Fachada** son ejemplos de patrones de comportamiento y estructurales, respectivamente, que encajan perfectamente en la arquitectura del proyecto de fisioterapia online.

2. Patrón de Diseño Observador

a. Concepto

El patrón Observador se utiliza cuando es necesario que varios objetos sean informados de cambios en el estado de otro objeto sin acoplamiento rígido. En un sistema como el de fisioterapia, en el que múltiples usuarios (pacientes y fisioterapeutas) necesitan estar al tanto de las actualizaciones en tiempo real, este patrón es esencial.

b. Diagrama UML del Patrón Observador



c. Detalles de la Implementación

La clase `Cita` actúa como el **sujeto**, mientras que los objetos `Paciente` y `Fisioterapeuta` son los **observadores**. Cuando se realiza un cambio en una cita, todos los observadores registrados se notifican automáticamente. Esto se hace usando el sistema de señales de Django.

Ejemplo de código adicional:

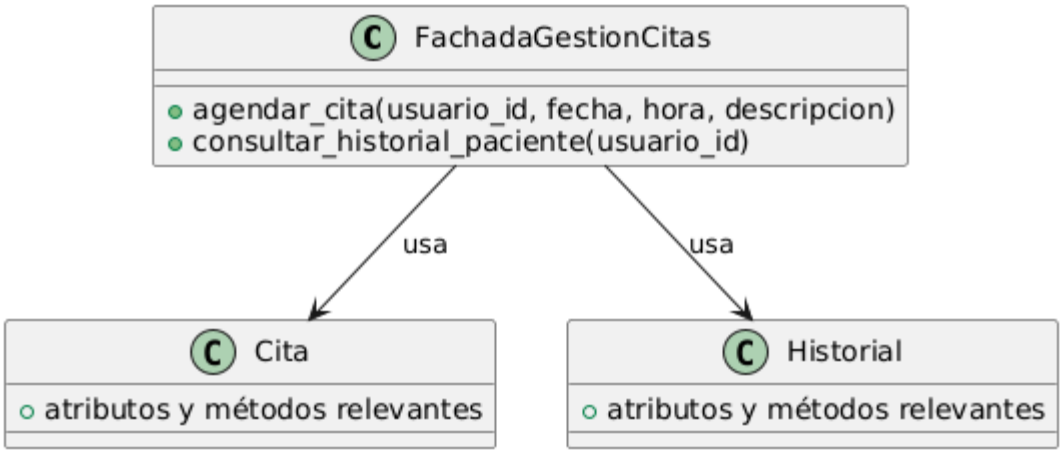
```
from django.db.models.signals import post_delete


@receiver(post_delete, sender=Cita)
def notificar_cancelacion_cita(sender, instance, **kwargs):
    print(f"Notificación: La cita de {instance.paciente.nombre} el {instance.fecha} ha sido cancelada.")
```

3. Patrón de Diseño Fachada

a. Concepto

El patrón Fachada es ideal para sistemas donde la complejidad interna debe ser ocultada detrás de una interfaz simplificada. Este patrón permite que las operaciones de la gestión de citas e historiales se encapsulen y se presenten de una manera clara al resto de la aplicación.

b. Diagrama UML del Patrón Fachada



c. Casos de Uso en el Proyecto de Fisioterapia

La clase `FachadaGestionCitas` se encarga de manejar las operaciones de citas y consultas de historial, proporcionando una única interfaz para interacciones complejas. Esto reduce la carga en el controlador y simplifica la lógica de negocio.

Ejemplo de uso extendido:

```
# archivo: servicios/fachada_citas.py

class FachadaGestionCitas:

    def actualizar_estado_cita(self, cita_id, nuevo_estado):

        cita = Cita.objects.get(id=cita_id)

        cita.estado = nuevo_estado

        cita.save()

        print(f"Estado de la cita actualizado a {nuevo_estado}.")


    def eliminar_cita(self, cita_id):
```

```
Cita.objects.filter(id=cita_id).delete()
print("Cita eliminada exitosamente.")
```

4. Comparación con Otros Patrones de Diseño

a. Comparación del Patrón Observador con el Patrón Mediador

El **patrón Mediador** centraliza la comunicación entre objetos para evitar dependencias directas. Aunque podría usarse en el proyecto de fisioterapia, el **patrón Observador** es más adecuado ya que permite la notificación directa de múltiples observadores sin necesidad de un mediador central.

b. Comparación del Patrón Fachada con el Patrón Adaptador

El **patrón Adaptador** se usa para convertir la interfaz de una clase en otra interfaz que el cliente espera. Aunque es útil en sistemas que integran componentes externos, el **patrón Fachada** es preferible aquí porque oculta la complejidad interna y proporciona un punto de acceso simplificado a las funcionalidades del sistema.

5. Beneficios y Desafíos en el Proyecto de Fisioterapia

a. Beneficios de Implementar el Patrón Observador

- **Reactivo y en tiempo real**: Los cambios en las citas se reflejan automáticamente en los pacientes y fisioterapeutas.
- **Desacoplamiento**: Reduce la dependencia entre la clase `Cita` y los objetos que la observan.

b. Beneficios de Implementar el Patrón Fachada

- **Simplificación**: Proporciona una única interfaz para manejar citas y historiales, mejorando la claridad y usabilidad del código.
- **Flexibilidad**: Facilita la introducción de cambios en los subsistemas sin afectar a otros componentes.

c. Desafíos y Soluciones

- **Complejidad añadida en grandes sistemas**: En aplicaciones más extensas, gestionar múltiples observadores puede complicar la depuración.
    - **Solución**: Implementar logs y monitoreo para rastrear las notificaciones.
- **Interfaz limitada de la Fachada**: Puede ocultar funcionalidades avanzadas que algunos módulos podrían necesitar.
    - **Solución**: Diseñar la fachada de manera modular, permitiendo acceso directo a funcionalidades avanzadas si es necesario.

Webgrafía

- https://docs.djangoproject.com/en/stable/
- https://refactoring.guru/design-patterns
- https://martinfowler.com/
- https://c4model.com/
- https://www.geeksforgeeks.org/software-design-patterns/
- https://www.dofactory.com/net/design-patterns
- https://www.programiz.com/uml-diagrams/class-diagram

Conclusión

La implementación de los patrones de diseño **Observador** y **Fachada** en el proyecto de fisioterapia ofrece una estructura más sólida, mantenible y escalable. Estos patrones permiten manejar notificaciones automáticas y simplificar las interacciones con el sistema, mejorando tanto la experiencia del usuario como la eficiencia del desarrollo. En futuras expansiones del proyecto, se podrían considerar otros patrones, como el **Patrón Estrategia** para gestionar diferentes métodos de pago o el **Patrón Comando** para implementar acciones de usuario deshacer/rehacer.

Con esta estructura, tienes el contenido suficiente para desarrollar un documento de hasta 10 páginas en Word, agregando explicaciones más detalladas, ejemplos adicionales de código, y gráficos UML de alta calidad.