Actividad 8: Informe Final Integrado

Resumen Ejecutivo

Este informe tiene como objetivo detallar el diseño arquitectónico de una plataforma de gestión de proyectos con inteligencia artificial, centrándose sobre todo en la aplicación de principios de desarrollo eficiente de software, patrones de arquitectura, y estrategias de control de calidad. Esta solución se ha diseñado manteniendo una serie de cualidades clave que el programa debe exhibir: el modularidad, escalabilidad, y mantenibilidad del código.

Para lograr esto se han aplicado principios de desarrollo como S.O.L.I.D., DRY, KISS y YAGNI, lo que nos ayuda diseñar una arquitectura limpia y extensible. Se han usado patrones de diseño como Factory Method, Adapter, y Observer, los cuales son esenciales para estructurar el código de manera clara y facilitar la reutilización de componentes y la comunicación entre los mismos. Además, se han implementado pruebas unitarias para asegurar el comportamiento de cada componente, e integraciones automatizadas mediante GitHub Actions, garantizando la calidad del software y reduciendo la deuda técnica desde la primera fase del desarrollo.

Justificación de las Decisiones Técnicas y Arquitectónicas

El diseño de la arquitectura debe adaptarse a las cualidades que debe tener el código y a los objetivos que estamos buscando. Debe ser un código adaptable y flexible, que sea capaz de integrarse con cualquier herramienta externa y escalar sin problemas.

Aplicación de Principios de Diseño:

- Principios SOLID: Garantizan un código desacoplado y fácil de mantener.
- Principio DRY: Minimiza las redundancias en el código, facilitando su legibilidad y mantenimiento.

- Principio KISS: Se aplica para evitar la complejidad innecesaria y mantener el producto lo más simple posible sin perder funcionalidad, facilitando el mantenimiento del código.
- Principio YAGNI: Nos recuerda que debemos priorizar las características esenciales, y evitar desarrollar funcionalidades hasta que sean necesarias.

Selección de Patrones de Diseño:

- Factory Method: Nos facilita la creación flexible de objetos sin necesidad de clases concretas para crearlos, lo que mejora la flexibilidad del programa. En nuestro caso hemos usado este patrón para crear tareas sin especificar el tipo de tareas siendo creadas, delegando esta responsabilidad a las subclases. Nuestra arquitectura contiene el siguiente modulo FabricaTareas.java:

```
package plataforma.tareas;

public abstract class FabricaTareas {
   public abstract Tarea crearTarea(String nombre);
}
```

La clase abstracta *FabricaTareas* actúa como una plantilla que define el método crearTarea pero no la implementa. Esto significa que cualquier clase que herede de *FabricaTareas* podrá crear tareas de una manera específica. en el código he usado como ejemplo el módulo FabricaTareasSimples.java:

```
package plataforma.tareas;

public class FabricaTareasSimples extends FabricaTareas {
   @Override
   public Tarea crearTarea(String nombre) {
      return new Tarea(nombre);
   }
}
```

La clase *FabricaTareasSimples* hereda de *FabricaTareas*, y tan solo crea una tarea con un nombre. Sin embargo, podríamos crear otras clases que hereden de FabricaTareas que creen distintos tipos de tareas. Por ejemplo:

```
package plataforma.tareas;

public class FabricaTareasComplejas extends FabricaTareas {
   @Override
   public Tarea crearTarea(String nombre) {
      Tarea tarea = new Tarea(nombre);
      tarea.setPrioridadAlta();
      return tarea;
   }
}
```

Esta clase creará tareas complejas, y establecerá su nivel de prioridad como alto, una vez implementado un sistema de prioridades en la clase *Tarea*. Otro ejemplo seria una clase que cree tareas recurrentes, para aquellas tareas que se repiten cada cierto tiempo o cada vez que se cumpla una condición:

```
package plataforma.tareas;

class FabricaTareasRecurrentes extends FabricaTareas {
   @Override
   public Tarea crearTarea(String nombre) {
      Tarea tarea = new Tarea(nombre);
      tarea.hacerRecurrente();
      return tarea;
   }
}
```

Por esta razón es por la que he optado por implementar este patrón de diseño, ya que podemos añadir diferentes tipos de fabricas en el futuro sin modificar el código existente. Esto favorece la extensibilidad y mantenibilidad del sistema.

- Adapter: El patrón adapter nos permite integrar la plataforma con herramientas externas sin necesidad de que sean compatibles. En nuestro código lo implementamos con una interfaz *HerramientaExterna*:

```
package plataforma.integracion;

public interface HerramientaExterna {
  void obtenerDatos();
}
```

Para cada herramienta que queramos integrar tendremos que crear una clase que implemente esta interfaz. Como ejemplos en el código he optado por crear clases que permitan integrar Trello y Google Drive, ya que son plataformas ampliamente usadas para la gestión de proyectos y almacenamiento de documentos a nivel empresarial. El código de los adaptadores tendría el siguiente formato:

```
package plataforma.integracion;

public class AdaptadorGoogleDrive implements
HerramientaExterna {
  @Override
  public void obtenerDatos() {
    System.out.println("Obteniendo datos desde Google Drive...");
  }
}

public class AdaptadorTrello implements HerramientaExterna {
  @Override
  public void obtenerDatos() {
    System.out.println("Obteniendo datos desde Trello...");
  }
}
```

Con el patrón Adapter, podemos comunicarnos con diferentes servicios sin modificar su código principal, lo que aumenta la modularidad de la arquitectura.

- Observer: Permite la notificación automática de cambios en las tareas mediante un servicio de suscripción entre módulos, y que estos reacciones

ante un cambio en el sistema en tiempo real. Esto es particularmente útil en nuestro caso, ya que nos permitirá que los miembros del equipo sean notificados instantáneamente sobre el progreso de las tareas, así evitando la necesidad de consultas manuales contantes. En nuestro código se implementa con una interfaz *IObservador*:

```
package plataforma.notificaciones;

public interface IObservador {
   void actualizar();
}
```

Esta interfaz será implementada por una clase *ObservadorTarea*, que mostrara un mensaje por consola cuando una tarea sea actualizada:

```
public class ObservadorTarea implements IObservador {
   @Override
   public void actualizar() {
      System.out.println("Notificación recibida: Se ha actualizado una
tarea.");
   }
}
```

Sin embargo, el verdadero potencial del patrón Observer se vería reflejado en clases como *ObservadorCorreo* o *ObservadorSMS* que usarían servicios externos de comunicación para notificar a los usuarios de actualizaciones en las tareas. Se pueden ir agregando diferentes opciones para notificar a los miembros del equipo sin necesidad de modificar el código. Los objetos de las clases de observadores, como *ObservadorTarea* u *ObservadorCorreo* serán almacenados en una lista de la clase *ServicioNotificaciones*. Esta clase nos permitirá agregar o eliminar observadores de la lista, y notificar a todos ellos cuando se actualize una tarea:

```
package plataforma.notificaciones;

import java.util.ArrayList;
import java.util.List;

public class ServicioNotificaciones {
  private List<IObservador> observadores = new ArrayList<>();

  public void agregarObservador(IObservador IObservador) {
    observadores.add(IObservador);
  }

  public void eliminarObservador(IObservador IObservador) {
    observadores.remove(IObservador);
  }

  public void notificarObservadores() {
    for (IObservador o : observadores) {
      o.actualizar();
    }
  }
}
```

Estrategia de Control de Calidad:

Para asegurar la calidad y funcionamiento correcto del código se han usado tests unitarios y de integración. Esto nos ayudara a identificar y corregir errores en el código desde las primeras etapas del desarrollo y así evitar las complicaciones y costes adicionales que conllevaría remediar estos errores en etapas posteriores. Las pruebas unitarias se han llevado a cabo usando JUnit, integrado con Maven. Aquí hay un ejemplo de lo que consistiría una prueba unitaria para comprobar que la clase *FabricaTareas* funciona correctamente:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import plataforma.tareas.*;

class FabricaTareasTest {
   @Test
   void testCrearTarea() {
      FabricaTareas fabrica = new FabricaTareasSimples();
      Tarea tarea = fabrica.crearTarea("Diseñar UI");
      assertNotNull(tarea);
      assertEquals("Pendiente", tarea.getEstado());
   }
}
```

Además de comprobar los módulos individuales, es necesario diseñar tests de integración para probar las interacciones entre las clases e interfaces. Para esto se han implementado pruebas de integración integradas en el pipeline de CI/CD mediante GitHub Actions.

Documentación

Repositorio de GitHub: https://github.com/jyustrod/Gestion-Proyectos.git

Video: Feedback_Tema5_YustresJavier_Actividad6.mp4

Conclusiones y aprendizajes obtenidos

El desarrollo de esta plataforma ha permitido aplicar de manera practica los principios y patrones de diseño esenciales en el desarrollo de software, y ha resultado en una solución modular, extensible, y mantenible. Se ha demonstrado que la correcta separación de responsabilidades y el uso de patrones adecuados

puede mejorar de manera significativa la estructura, eficiencia, e interoperabilidad del software.

Como mejoras futuras se podría explorar la implementación de microservicios para mejorar la escalabilidad horizontal del sistema, o la adopción de contenedorizacion con Docker y orquestación con Kubernetes.