Actividad 8: Informe Final Integrado

Unax Aller

15/03/2025

Índice del Informe Final Integrado

1.Resumen Ejecutivo

1.1 Síntesis del Proyecto

El proyecto trata sobre hacer una Plataforma Inteligente para gestionar proyectos, que sirve para mejorar como se trabaja, evitar que haya retrasos y dar consejos para que el equipo sea más productivo. Se ha hecho de forma que todo esté bien organizado siguiendo el modelo MVC, así cada parte está separada, como lo que hace la lógica, lo que se muestra en pantalla y lo que controla todo. Para que el código esté bien hecho y no sea un lío, se han seguido normas modernas como S.O.L.I.D., DRY, KISS y YAGNI, además de usar formas conocidas de programar como Factory Method, Facade y Observer. Se ha programado en Java con Maven en IntelliJ y todo lo que se ha hecho se ha guardado en GitHub para que todos los cambios y el trabajo en equipo sean más fáciles de llevar.

1.2 Resultados Clave

El código está hecho para que sea modular y fácil de mantener. Se han seguido normas como S.O.L.I.D. y formas de programar que ayudan a que el sistema esté compuesto por partes independientes y que se puedan mejorar sin problema. Se ha optimizado todo para que los procesos sean más eficientes. Separar bien lo que hace cada parte y meter pruebas unitarias ha servido para detectar fallos rápido y reducir los problemas técnicos. El sistema es escalable porque usa una estructura basada en MVC y el método Factory Method, lo que hace que se puedan añadir cosas nuevas sin que el sistema se rompa. La interfaz se ha simplificado con el patrón Facade, lo que ayuda a que las funciones complicadas sean más fáciles de usar y mejoran la experiencia del usuario. La sincronización funciona en tiempo real gracias a Observer, que hace que cualquier cambio se vea al instante en la pantalla, haciendo que todo vaya más rápido. Estos resultados demuestran que lo que se ha hecho funciona bien y deja una buena base para mejorar y ampliar el sistema en el futuro, siguiendo lo que se considera lo mejor en la industria.

2. Justificación de las Decisiones Técnicas y Arquitectónicas

2.1 Aplicación de Principios de Diseño (S.O.L.I.D., DRY, KISS, YAGNI)

Aplicar estos principios es clave para tener un sistema fuerte, que se pueda mejorar y que sea fácil de arreglar si hace falta. Con S.O.L.I.D. conseguimos que cada clase haga solo una cosa, lo que permite que el sistema crezca sin tener que cambiar lo que ya está hecho. Además, usar LSP, ISP y DIP hace que se puedan cambiar unas partes por otras sin problema y que dependamos más de ideas generales en vez de cosas muy concretas. También usamos DRY para que no haya código repetido y KISS

para que todo sea fácil de entender y no se complique más de la cuenta. Por último, YAGNI nos sirve para centrarnos solo en lo importante, sin meter cosas que no hacen falta y así reducimos los problemas técnicos.

2.2 Selección de Patrones de Diseño (Factory, Facade, Observer)

Elegir bien los patrones de diseño ha sido una decisión clave para solucionar problemas típicos y hacer que el sistema esté mejor organizado. El patrón Factory Method se encarga de crear objetos, lo que permite hacer diferentes tipos de proyectos como web o móvil sin que el cliente tenga que saber cómo se construyen, haciendo que en el futuro sea más fácil añadir más cosas. El patrón Facade ayuda a que usar partes complicadas del sistema, como la gestión de tareas y el envío de notificaciones, sea más sencillo, ya que da una única forma de acceder a ellas y así se evita que todo esté demasiado conectado. Por último, el patrón Observer se asegura de que cuando algo cambia en el sistema, ese cambio se vea al instante en todos los elementos que lo necesiten, lo que es muy importante para que la sincronización entre el modelo y la vista en aplicaciones en tiempo real sea correcta.

2.3 Estrategia de Control de Calidad y Pruebas Unitarias

Para asegurarse de que el software tenga una calidad alta, se ha puesto en marcha un plan completo que usa pruebas automáticas, integración continua y revisión del código. Se usan programas como JUnit para crear y hacer pruebas que comprueben si todo funciona como debería y también si se manejan bien los errores, lo que ayuda a encontrar fallos desde el principio. Además, se han metido sistemas de CI/CD como Jenkins o GitHub Actions para que cada vez que se haga un cambio en el código, estas pruebas se hagan solas, reduciendo los problemas técnicos. También se usan herramientas como SonarQube, PMD y Checkstyle para revisar el código y ver si cumple con las normas, lo que facilita que el sistema pueda mejorar sin complicarse demasiado. Todo esto hace que el software sea más estable y fácil de mantener, evitando problemas en el futuro.

3. Documentación de Actividades

3.1 Actividad 1: Análisis y Selección de Principios de Diseño

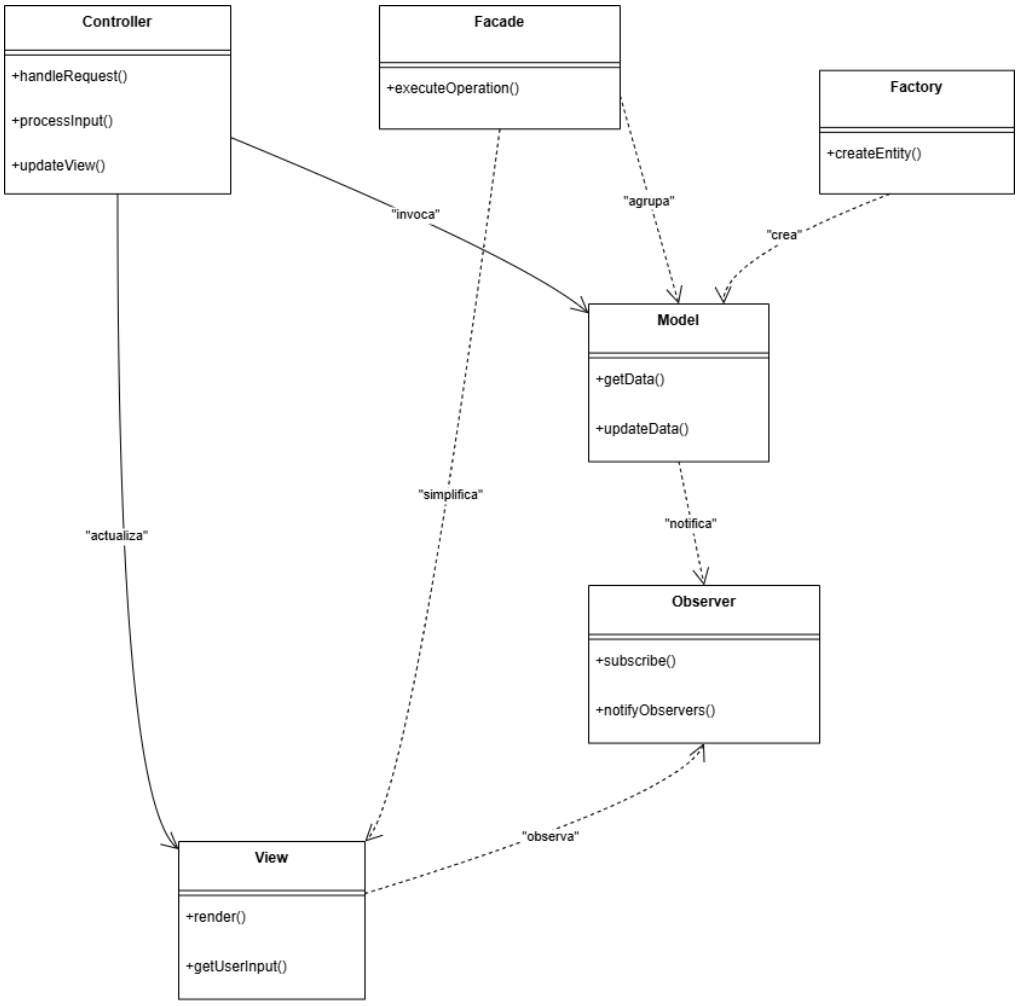
Se hizo una investigación muy completa sobre los principios más importantes del diseño, sobre todo S.O.L.I.D., DRY, KISS y YAGNI. Se preparó un documento donde se explicó cómo estos principios se iban a usar en la estructura de la plataforma, asegurando que cada clase hiciera solo una cosa, evitando repetir código, manteniendo un diseño simple y centrándose solo en lo realmente necesario. Esta forma de trabajar garantiza que el código sea modular, fácil de mantener y que se pueda mejorar sin problemas en el futuro.

3.2 Actividad 2: Identificación y Justificación de Patrones de Diseño

Se eligieron tres patrones de diseño muy importantes. El Factory Method se usó para que la creación de objetos estuviera bien organizada sin que se tuviera que mostrar cómo se crean por dentro. El Facade sirvió para dar una única forma de interactuar con partes complicadas del sistema, haciendo que todo fuera más fácil de usar. El Observer se utilizó para asegurarse de que cuando algo cambiaba en el sistema, todos los elementos que lo necesitaban fueran avisados automáticamente. Cada uno de estos patrones se eligió porque ayudan a que el sistema sea más organizado, fácil de mejorar y más sencillo de mantener.

3.3 Actividad 3: Esquema de la Arquitectura del Software

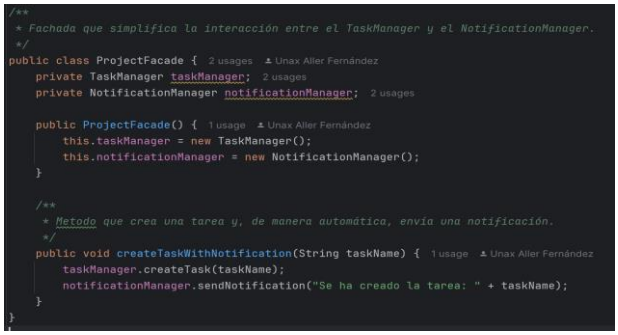
Se hizo un diagrama UML que muestra cómo está organizada la estructura general del sistema. En este esquema se puede ver claramente cómo se separan la lógica de negocio, la presentación y el control siguiendo el modelo MVC. También se han incluido los patrones de diseño que se han usado, lo que ayuda a entender mejor cómo interactúan y dependen entre sí los diferentes componentes. Esto hace que el sistema sea más fácil de analizar y mejorar en el futuro.



3.4 Actividad 4: Implementación Práctica de Patrones de Diseño

Se hicieron fragmentos de código en Java usando Maven en IntelliJ para demostrar cómo se aplican en la práctica los patrones que se eligieron como Factory Method, Facade y Observer. Cada parte de código tiene comentarios que explican bien cómo funciona y qué ventajas aporta al sistema, mostrando claramente cómo se consigue que el diseño sea modular y fácil de mejorar. Esto ayuda a entender mejor cómo se usan estos patrones en un proyecto real y por qué son útiles.

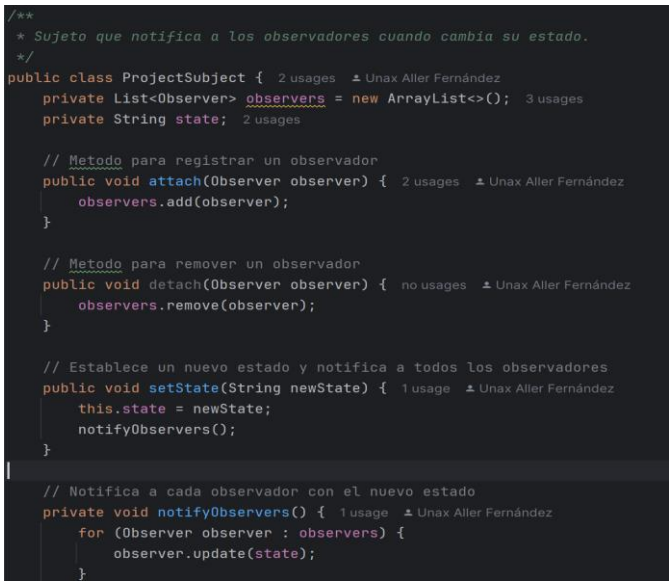
```java
/**
 * Fábrica que crea instancias de Project según el tipo solicitado.
 */
public class ProjectFactory {  2 usages  Unax Aller Fernández
    public static Project createProject(String type) {  2 usages  Unax Aller Fernández
        if (type.equalsIgnoreCase( anotherString: "web")) {
            return new WebProject();
        } else if (type.equalsIgnoreCase( anotherString: "mobile")) {
            return new MobileProject();
        }
        throw new IllegalArgumentException("Tipo de proyecto no soportado: " + type);
    }
}
```



```java
/**
 * Fachada que simplifica la interacción entre el TaskManager y el NotificationManager.
 */
public class ProjectFacade {  2 usages  Unax Aller Fernández
    private TaskManager taskManager;  2 usages
    private NotificationManager notificationManager;  2 usages

    public ProjectFacade() {  1 usage  Unax Aller Fernández
        this.taskManager = new TaskManager();
        this.notificationManager = new NotificationManager();
    }

    /**
     * Metodo que crea una tarea y, de manera automática, envía una notificación.
     */
    public void createTaskWithNotification(String taskName) {  1 usage  Unax Aller Fernández
        taskManager.createTask(taskName);
        notificationManager.sendNotification("Se ha creado la tarea: " + taskName);
    }
}
```



```java
/**
 * Sujeto que notifica a los observadores cuando cambia su estado.
 */
public class ProjectSubject {  2 usages  Unax Aller Fernández
    private List<Observer> observers = new ArrayList<>();  3 usages
    private String state;  2 usages

    // Metodo para registrar un observador
    public void attach(Observer observer) {  2 usages  Unax Aller Fernández
        observers.add(observer);
    }

    // Metodo para remover un observador
    public void detach(Observer observer) {  no usages  Unax Aller Fernández
        observers.remove(observer);
    }

    // Establece un nuevo estado y notifica a todos los observadores
    public void setState(String newState) {  1 usage  Unax Aller Fernández
        this.state = newState;
        notifyObservers();
    }

    // Notifica a cada observador con el nuevo estado
    private void notifyObservers() {  1 usage  Unax Aller Fernández
        for (Observer observer : observers) {
            observer.update(state);
        }
```

3.5 Actividad 5: Estrategia de Control de Calidad y Pruebas Unitarias

Se pensó un plan completo para controlar la calidad del software usando pruebas automáticas con JUnit, integración continua con pipelines como Jenkins o GitHub Actions y revisión del código con herramientas como SonarQube, PMD y Checkstyle. Gracias a esto, se pueden encontrar errores desde el principio, evitando problemas más adelante y asegurando que el software sea fuerte y fácil de mejorar. Todo esto hace que el sistema esté mejor preparado para seguir creciendo sin complicarse demasiado.

3.6 Actividad 6: Presentación en Video

Se hizo un vídeo de un máximo de cinco minutos donde se explica con detalle cómo el uso de los principios y patrones de diseño ha ayudado a que el software sea de mejor calidad, más organizado y fácil de mantener. En la presentación se enseñaron ejemplos prácticos, imágenes de la pantalla y casos concretos con partes del código que se ha desarrollado. Esto sirvió para que todo quedara más claro y se pudiera ver de forma visual cómo funcionan estos conceptos en la práctica.

https://drive.google.com/file/d/1g2vr2qxE18lZP3yZAfwsS8oJftHhOJ_Y/view?usp=drive_link

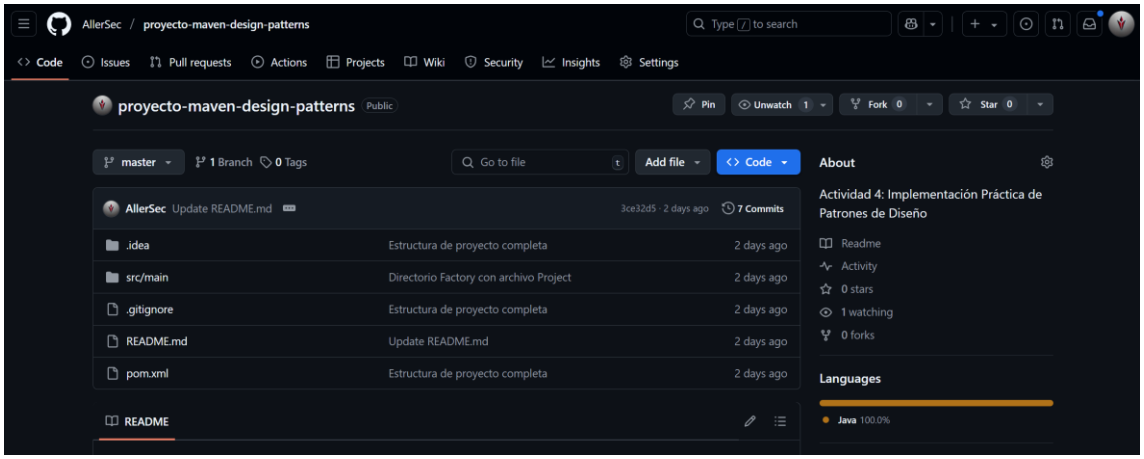
3.7 Actividad 7: Análisis Comparativo con Arquitecturas Industriales

Se hizo un análisis comparativo donde se comparó la arquitectura que se diseñó con otras soluciones que ya están muy usadas en la industria, como las arquitecturas monolíticas, las que usan microservicios y la Clean Architecture. Se encontraron puntos fuertes en cuanto a que el sistema es modular y fácil de mantener, pero también se vieron cosas que se pueden mejorar, como la necesidad

de que el sistema pueda crecer en horizontal y separar mejor las partes usando métodos como Domain-Driven Design. Esto ayudó a entender mejor en qué aspectos la arquitectura funciona bien y en cuáles todavía se puede mejorar.

3.8 Enlaces al Repositorio y al Video

Se añadieron enlaces directos al repositorio de GitHub donde está todo el código del proyecto y también al vídeo de presentación. Esto hace que sea más fácil comprobar y evaluar todo el proceso que se ha llevado a cabo, permitiendo ver de forma clara cómo se ha desarrollado el trabajo.



https://github.com/AllerSec/proyecto-maven-design-patterns.git

https://drive.google.com/file/d/1g2vr2qxE18lZP3yZAfwsS8oJftHhOJ_Y/view?usp=drive_link

4. Final

El desarrollo de la Plataforma Inteligente de Gestión de Proyectos ha servido para comprobar que un enfoque basado en el uso correcto de principios y patrones de diseño funciona bien. Durante el proyecto, se han aplicado S.O.L.I.D., DRY, KISS y YAGNI, lo que ha ayudado a que el código sea más limpio, modular y fácil de mantener. Además, usar los patrones Factory Method, Facade y Observer ha hecho que sea más sencillo añadir nuevas funciones e integrar cambios sin problemas. Gracias a esto, se ha conseguido reducir la deuda técnica, mejorar la calidad del software y crear una base sólida para seguir mejorándolo en el futuro. También se han metido pruebas unitarias y herramientas de análisis estático para encontrar y corregir errores desde el principio, haciendo que el sistema sea todavía más estable. En resumen, todo el trabajo realizado no solo ha cumplido los objetivos del proyecto, sino que también ha dejado aprendizajes importantes y posibles mejoras para sistemas más grandes, como el uso de arquitecturas basadas en microservicios o Clean Architecture.

5. Referencias y Anexos

5.1 Bibliografía y Fuentes

- Documentos teóricos de la asignatura:
    - Contenido teórico Tema1.pdf
    - Contenido teórico Tema2.pdf
    - Contenido teórico Tema3.pdf
    - Contenido teórico Tema4.pdf
    - Contenido teórico Tema5.pdf
    - Contenido teórico Tema6.pdf
- Artículos y publicaciones sobre S.O.L.I.D., DRY, KISS, YAGNI y patrones de diseño (e.g., recursos en Adictos al Trabajo).
- Referencias en línea y manuales oficiales de Java, Maven y frameworks de pruebas (JUnit, SonarQube, etc.).

5.2 Diagramas y Capturas de Pantalla

- Diagramas UML que ilustran la estructura de la solución y la interacción de los patrones de diseño (por ejemplo, los diagramas de ProjectFacade y el flujo de notificación entre ProjectSubject y UserObserver).
- Capturas de pantalla del entorno de desarrollo en IntelliJ, mostrando fragmentos de código representativos y resultados de ejecución en consola.

5.3 Documentación Complementaria

- Enlaces a repositorios de código y al video de presentación.