Universidad de Costa Rica
Escuela de Ciencias de la Computación e Informática
Diseño de Software (CI-0136)
Documento Final CHIKI

Buenas prácticas de ingeniería de software que cumple el código

Se sigue una arquitectura hexagonal, separando responsabilidades correctamente, reduciendo el acoplamiento y facilitando las pruebas unitarias. Se cumple con los 5 principios SOLID, y con buenas prácticas de clean code. Se cumple con el estándar de estilos de código PEP8.

Las funcionalidades reutilizables están bien documentadas, como el framework de logs, y todas las funcionalidades están cubiertas con pruebas unitarias.

Patrones de diseño que se utilizaron

- Iterator
    - Funcionalidad que usa el patrón: Mostrar logs en pantalla con paginación
    - Descripción clara del patrón o principio: Es una clase que abstrae el acceso a los elementos de una colección y permite acceder secuencialmente a los elementos de la colección para la cual se implementa, independientemente de su estructura interna.
    - Motivación de uso: Para hacer paginación es necesario recorrer los elementos de una colección mediante rangos secuenciales de elementos, para lo cual Iterator es ideal.
    - Explicación de su implementación y guía sobre cómo extenderlo si es necesario: La clase LogIterator tiene un método llamado load_logs que recibe como parámetro un índice y una cantidad de logs a extraer a partir de ese índice, y como resultado guarda esos logs en su atributo "logs" que es una lista. Para obtener esos logs que cargó se accede directamente a esa lista de solo lectura. Para extenderlo, se le pueden agregar cuantos métodos se deseen mientras se trabaje con un repositorio que cumpla la interfaz IFirebaseLogRepository, o se puede cambiar esta dependencia si contiene los métodos que se están usando actualmente en el código, sin mucha complicación

- Factory

    - Funcionalidad que usa el patrón: Interfaces de tipo de notificación de baneo
    - Descripción clara del patrón o principio: El patrón Factory es un patrón de creación que proporciona una interfaz para crear objetos en una superclase, pero permite que las subclases decidan qué clase instanciar. Su objetivo es delegar la creación de objetos a una clase derivada, evitando así acoplar el código directamente a clases concretas.
    - Motivación de uso: Se decidió utilizar este patrón debido a nuestra intención de apegarnos lo más posible al cumplimiento de los principios SOLID, en este caso específicamente a Open/Closed. Tener una interfaz de tipo de mensaje de baneo nos permitió que la lógica del código pueda ser extendida con nuevos tipos de notificación sin necesidad de modificar lo ya existente.
    - Explicación de su implementación y guía sobre cómo extenderlo si es necesario: Se creó una interfaz llamada BanNotificaction que posee un método llamado notify que envía un correo electrónico notificando el baneo y su motivo. Distintas clases heredan de esta interfaz y modifican el cuerpo de su mensaje para luego ejecutar notify.
- Singleton
    - Funcionalidad que usa el patrón: servicio de registro de logs de la aplicación
    - Descripción clara del patrón o principio: Singleton es un patrón de diseño de software creacional, definido por primera vez en el conjunto original de 23 patrones de diseño de software de la Banda de los Cuatro (GdF). Garantiza que, durante la ejecución de un programa, solo exista una instancia de una clase determinada. Se utiliza cuando, por cualquier motivo, no debe existir más de una instancia de dicha clase.
    - Motivación de uso: al tener una clase Audit Subject, por razones prácticas producto de la esencia y funcionalidad de la misma, es mejor garantizar que se mantenga una única instancia de esta para evitar inconsistencia y desperdicio de recursos. De esta forma en diversas partes de la aplicación es posible tener un punto centralizado de registro de logs para tareas de auditoría y administración.
    - Explicación de su implementación y guía sobre cómo extenderlo si es necesario: En la clase Audio Subject se tiene un miembro estático el cuál corresponde a una instancia de sí misma. El constructor de la clase incluye lógica que verifica si dicha variable de instancia es nula o no, para garantizar la existencia de una

única instancia. En caso de estar vacía, se crea y retorna la instancia, sino, simplemente se retorna la instancia.

Reflexiones y trabajo pendiente

El equipo de trabajo de gestión administrativa y moderación en general no tuvo inconvenientes durante el desarrollo de las funcionalidades asignadas. La gran mayoría de peticiones asignadas por el product owner pudieron ser implementadas. A pesar de que algunos miembros del equipo son nuevos en temas de desarrollo web, estos pudieron adaptarse a las condiciones y aprendieron bastante del proceso.

Respecto a aspectos de mejora, se pudo gestionar el tiempo más efectivamente para poder haber implementado la totalidad de peticiones e ideas del PO. Por ejemplo, quedó pendiente añadir el feature de banear temporalmente a usuarios, las estadísticas en el dashboard de administrador sobre la cantidad de estudiantes han contratado a un tutor específico, y hacer que se registren logs cuando se editan o eliminan usuarios.

También se pudieron hacer más reutilizables y genéricas algunas piezas de software, por ejemplo las relacionadas a paginación