Desarrollo de software

Refactorización

Ejercicio 1: Refactorización para mejorar la cohesión y reducir el acoplamiento

La cohesión se refiere a la medida en que los elementos de un módulo pertenecen juntos. Un alto nivel de cohesión dentro de una clase facilita la mantenibilidad y la comprensibilidad. Por otro lado, el acoplamiento se refiere a la medida en que las clases están dependientes unas de otras. Un bajo acoplamiento aumenta la posibilidad de reutilizar una clase.

Sea la siguiente aplicación Java con clases excesivamente acopladas y con baja cohesión. Debes identificar problemas específicos de cohesión y acoplamiento y refactorizar el código para mejorar estas métricas. Deberán proporcionar un breve informe explicando qué cambios se realizaron y por qué.

Código inicial

```
public class EmployeeManager {

    public void addEmployee(String name, String department) {

        // Añade un empleado al departamento

        System.out.println("Empleado añadido");

    }

    public void removeEmployee(String name) {

        // Elimina un empleado

        System.out.println("Empleado eliminado");

    }

    public void changeDepartment(String employeeName, String newDepartment) {

        // Cambia un empleado de departamento
```

```
        System.out.println("Departamento cambiado");
    }
    public void printDepartmentReport(String department) {
        // Imprime un informe del departamento
        System.out.println("Informe del departamento " + department);
    }
    public void printAllDepartments() {
        // Imprime todos los departamentos
        System.out.println("Lista de todos los departamentos");
    }
}
```

Ejercicio 2: aplicación de patrones de diseño para refactorización

Los patrones de diseño son soluciones típicas a problemas comunes en el diseño de software. La aplicación de patrones de diseño adecuados puede mejorar la estructura del código y su mantenibilidad.

Sea un segmento de código que realiza múltiples funciones utilizando una estructura monolítica. Los estudiantes deben identificar y aplicar al menos dos patrones de diseño (como Singleton, Factory, Observer, etc.) para refactorizar el código de manera que se mejore su estructura y mantenibilidad. Deben explicar su elección de patrones y cómo estos patrones ayudan en el contexto de la aplicación.

Código inicial

Supongamos que tenemos un sistema que gestiona notificaciones para una aplicación. El código inicial está diseñado de tal manera que maneja diferentes tipos de notificaciones de forma monolítica.

```
public class NotificationManager {
    private String user;
    private String message;
```

```
    public NotificationManager(String user, String message) {
        this.user = user;
        this.message = message;
    }

    public void sendNotification(String type) {
        if ("email".equals(type)) {
            System.out.println("Enviando email a " + user + " con mensaje: " + message);
        } else if ("sms".equals(type)) {
            System.out.println("Enviando sms a " + user + " con mensaje: " + message);
        } else if ("app".equals(type)) {
            System.out.println("Enviando notificación de app a " + user + " con mensaje: " + message);
        }
    }
}
```

Ejercicio 3: Refactorización para introducir principios SOLID

Los principios SOLID son un conjunto de cinco principios de diseño orientado a objetos que ayudan a desarrollar software más entendible, flexible y mantenible.

Tienes una base de código que viola varios de los principios SOLID. Debes analizar el código, identificar las violaciones, y refactorizar el código para adherirse a estos principios. Además, deben documentar cada cambio realizado y la justificación en términos del principio SOLID correspondiente.

Código inicial

Imaginemos que tenemos una clase en un sistema de gestión de contenido que maneja tanto la lógica de usuario como la lógica de la base de datos para artículos de blog.

```
public class BlogManager {

   private List<String> articles = new ArrayList<>();



   public void addArticle(String article) {

      if (article != null && !article.isEmpty()) {

         articles.add(article);

         System.out.println("Artículo añadido: " + article);

         saveArticleToDatabase(article);

      }

   }

   private void saveArticleToDatabase(String article) {

      // Simulación de guardar en la base de datos

      System.out.println("Guardando en la base de datos: " + article);

   }



   public void printAllArticles() {

      for (String article : articles) {

         System.out.println("Artículo: " + article);

      }
```

```
    }
}
```

Ejercicio 4: Refactorización de código para mejorar la prueba y la cobertura del código

La refactorización puede ser crucial para mejorar la capacidad de prueba del código. Código bien estructurado permite una segmentación más efectiva de las pruebas y una cobertura más completa.

Debes trabajar con un código que es difícil de probar debido a sus dependencias y su estructura complicada y poder refactorizar el código para hacerlo más amigable para las pruebas, implementar pruebas unitarias básicas, y luego usar una herramienta de cobertura de código para evaluar su trabajo. Debes explicar cómo sus cambios facilitaron una mejor prueba y qué estrategias podrían implementarse para aumentar la cobertura del código.

Código inicial

Supongamos que tenemos una clase que gestiona un sistema de inventario para una tienda, pero está diseñada de una manera que dificulta la realización de pruebas debido a la alta dependencia en recursos externos directamente en los métodos.

```
public class InventoryManager {

    private List<Product> products = new ArrayList<>();

    public void addProduct(int id, String name, double price, int quantity) {

        Product product = new Product(id, name, price);

        product.setQuantity(quantity);

        updateInventoryDatabase(product);

        System.out.println("Producto añadido: " + product);

    }



    private void updateInventoryDatabase(Product product) {

        // Lógica para actualizar la base de datos
```

```
        System.out.println("Actualizando base de datos para: " + product);

    }

}
```

Problemas en el código inicial

1. **Alta dependencia en recursos externos**: La lógica de actualizar la base de datos está fuertemente acoplada con la lógica de negocio de agregar productos, haciendo que las pruebas unitarias sean difíciles.
2. **Difícil de brobar**: Los métodos tienen efectos secundarios (como impresiones en consola y actualizaciones de base de datos) que dificultan la verificación de su comportamiento en pruebas unitarias sin una base de datos real o la interacción con el sistema de archivos.