TALLER SOLID

LABORATORIO INGENIERÍA DE SOFTWARE II

ESTUDIANTES:

Juan Manuel Ceron

Juan Carlos Manquillo Tibanta

Sharyth Yaliny Velasco

Freddy Daniel Botia Calle

UNIVERSIDAD DEL CAUCA

FACULTAD DE INGENIERÍA ELECTRÓNICA Y TELECOMUNICACIONES

INGENIERÍA DE SISTEMAS

POPAYÁN - CAUCA

2024

PARTE I

ÚNICA RAZÓN:

La clase Service parece tener varias responsabilidades. Maneja la lógica de negocio (calculateProductTax), la lógica de acceso a datos (saveProduct y listProducts), y la inicialización de la base de datos (initDatabase). Esto viola el SRP, ya que la clase debería tener una única razón para cambiar. Podría considerar dividir estas responsabilidades en clases separadas.

Solucion: para solventar este problema, se dividieron las responsabilidades en 2 clases:

ProductRepository: se encarga de manejar el acceso a la base y proporcionar procedimientos SQL

Service: se encarga de implementar el cálculo del impuesto e invocar procedimientos SQL validando restricciones

EXTIENDE NO MODIFIQUES:

Viola el principio OCP dado que tenemos una clase enum y en ella se hace un llamado de diferentes países la cual conduce a modificar la clase lo cual viola el principio.

Solución: Para cada país se implementó una clase diferente lo que al momento de extender se va agrega en la clase factory y se agrega una clase externa.

TAL PADRE TAL HIJO:

La clase SpecializedTruck extiende la clase Truck y anula el método addTrip, pero cambia su comportamiento al actualizar el odómetro, lo cual no es esperado en una relación de herencia.

Solución: la clase SpecializedTruck evita la actualización del odómetro en su implementación del método addTrip, manteniendo así la consistencia con la clase padre Truck. Esto permite que las instancias de SpecializedTruck se utilicen como sustitutos válidos de instancias de Truck, cumpliendo así con el principio de sustitución de Liskov.

NO DEPENDAS NO NECESITES:

Se está violando el Principio de Segregación de Interfaces (ISP) de los principios

SOLID. La interfaz IRepository tiene demasiadas responsabilidades, ya que incluye métodos relacionados con diferentes entidades (usuarios, project, task).

Solución: Se puede dividir la interfaz en interfaces más pequeñas y específicas, cada una relacionada con una entidad concreta. Esto mejorará la cohesión y evitará que las clases tengan que implementar métodos que no necesitan.

ABSTRACTO BUENO:

En las clases Service y ProductRepository, ProductRepository tiene una dependencia directa de service. Esto implica que Service está dependiendo de un módulo de nivel inferior (la implementación concreta de ProductRepository), lo que viola el principio de inversión de dependencias.

La solución para adherirse al principio DIP es introducir una abstracción (por ejemplo, una interfaz) que ProductRepository implementará. Luego, la clase Service dependerá de esta abstracción en lugar de depender directamente de la implementación concreta de ProductRepository. Esto permitirá que Service sea más flexible y no esté acoplado directamente a la implementación específica de ProductRepository.

PARTE II

PRINCIPIOS SOLID IMPLEMENTADOS INCORRECTAMENTE:	**SOLUCIÓN INTEGRADA AL PRINCIPIO:**
Principio única responsabilidad (En el archivo de ProductRepository se estaba implementando simultáneamente la lógica del negocio y del acceso a datos, además dependían mucho de esta clase porque de esta se desprendían la mayoría de funcionalidades del programa lo cual el código era poco mantenible)	Se creó una clase ProductService en la que se invocó un objeto de dicha clase para así implementar correctamente el principio de única responsabilidad.
Principio de inversión de dependencias (Al haber implementado la solución anterior, se tenía que a clase ProductService y ProductRepository dependían estrechamente la una de la otra)	Se creó una interfaz para cada una de las clases anteriormente mencionadas, aplicando así el principio de inversión de dependencia correctamente basándose en la abstracción.