INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

Segundo Parcial

Principios de Diseño Orientado a Objetos

Análisis y diseño orientado a objetos

GRUPO: 2CV14

Alumnos:

Estrada Botello Oscar Eduardo	**2019630411**
Pastrana Torres Victor Norberto	**2019630349**
Pereda Reyes Alfredo	**2019630500**
Romero Escogido Donovan	**2019630460**

PROFESORA:

Reyna Elia Melara Abarca

FECHA DE ENTREGA: viernes 21 de mayo de 2021

Principios de diseño orientado a objetos

1. **¿A qué se refiere el término Diseño Pobre?**

El Diseño Pobre carece de ciertas características y/o estructuras que provocan que el sistema tenga fallas y dificulta la lectura y comprensión del usuario.

2. **¿Cuáles son síntomas del Diseño Pobre?**

Rigidez: Es difícil cambiar el sistema porque cada cambio forzar muchos otros cambios en otras partes del sistema.

Fragilidad: Los cambios causan que el sistema se rompa en lugares que no tienen relación con la parte que fue cambiada.

Inmovilidad: Es difícil separar las partes del sistema para que estas puedan ser reutilizadas en otros sistemas.

Viscosidad: Hacer cosas correctas es más difícil que hacer cosas incorrectas.

Complejidad Innecesaria: El diseño contiene infraestructura que no añade ningún beneficio directo.

Repetición innecesaria: El diseño contiene estructuras repetidas que podrían unificarse bajo una sola abstracción.

Opacidad: Es difícil de leer y entender. No expresa bien su intención.

3. **Explique a qué se refiere el concepto Bad Design Smell o Bad Smell.**

A Desing Smell es un síntoma de algo que no puede ser medido de forma subjetiva. Algunas veces esto suele ser ocasionado por la violación de uno o más principios de programación.

Los principios de programación son:

SRP: Principio de la responsabilidad simple

OCP: Principio Open-Close

LSP: Principio Sustitución Liskov

DIP: Principio de Inversión de Dependencias

SIP: Principio de la Segregación de interfaz

4. **¿Qué significa SOLID?**
   Enlace a Infografía SOLID

5. **El equipo selecciona uno de los principios SOLID e incluye un ejemplo en código en el que se vea explicado.**
    - **S**ingle Responsibility
    - **O**pen/Closed
    - **L**iskov Substitution
    - **I**nterface Segregation
    - **D**ependency Inversion

    Principio de **responsabilidad única.** Este principio establece que una clase sólo debe tener una responsabilidad. Además, sólo debería tener una razón para cambiar.

    Beneficios:

    - ➔ **Pruebas:** una clase con una responsabilidad tendrá mucho menos casos de prueba.
    - ➔ **Acoplamiento inferior:** menos funcionalidad de una sola clase tendrá menos dependencias.
    - ➔ **Organización:** las clases más pequeñas y bien organizadas son más fáciles de buscar que las monolíticas.

```java
public class Car {
    private String name;
    private String model;
    private String Description;

    //constructor, getters and setters
}
```

    En este código, almacenamos el nombre, el modelo y una descripción a una instancia de car.

    Ahora si agregamos un par de métodos para consultar el texto.

```java
public class Car {
    private String name;
    private String model;
    private String Description;

    //constructor, getters and setters

    //methods that directly relate to the book properties
    public String replaceWordInText(String word){
        return text.replaceAll(word, text);
    }

    public boolean isWordInText(String word){
        return text.contains(word);
    }
}
```

Ahora en la clase car funciona bien, y se puede almacenar tantos carros se requieran.

Pero para enviar el texto a la consola y poder leerla, añadimos un método de impresión.

```java
public class Car {
    //...
    void printTextToConsole(){
        //our code for formatting and printing the text
    }
}
```

Sin embargo, este código infringe el principio de responsabilidad única que hemos descrito anteriormente.

Para arreglar este problema, se implementó una clase separada que se ocupa únicamente de imprimir nuestros textos.

```java
public class CarPrinter {

    //methods for outputting text
    void printTextToConsole(String text){
        //our code for formatting and printing the text
    }

    void printTextToAnotherMedium(String text){
        //code for writting to any other location..
    }
}
```


De esta forma podemos cumplir con el principio de responsabilidad.

6. **Existen otros principios además de SOLID. En este apartado el equipo podría proporcionar algún otro ejemplo, su propósito y un ejemplo.**

GRASP (Principios de software de asignación de responsabilidad general)

GRASP es un conjunto de 9 Patrones Generales de Responsabilidad de Asignación Software.

1.- Alta cohesión
Información que almacena una clase, debe de ser coherente y debe estar relacionada con la clase.

2.- Bajo acoplamiento
Consiste en tener las clases lo menos ligadas entre sí, para cuando modifiquemos alguna clase, el resto de clases no se vean afectadas.

3.- Controlador
Patrón que realiza la función de intermediario entre una determinada interfaz y la clase que la implementa, con la capacidad de ser reutilizada.

4.- Creador
Es responsable de la creación o quien instancia nuevos objetos y/o clases.

```
public class Customer : Entity, IAggregateRoot
{
    private readonly List<Order> _orders;

    public void AddOrder(List<OrderProduct> orderProducts)
    {
        var order = new Order(orderProducts); // Creator

        if (this._orders.Count(x => x.IsOrderedToday()) >= 2)
        {
            throw new BusinessRuleValidationException("You cannot order more than 2
        }

        this._orders.Add(order);

        this.AddDomainEvent(new OrderAddedEvent(order));
    }
}
```

Como puede ver arriba, la clase Cliente agrega de forma compuesta Órdenes (no hay Orden sin Cliente), registra Órdenes, usa Órdenes de cerca y tiene datos de inicialización pasados por parámetros de método. Candidato ideal para "Creador de pedidos".

5.- Experto en información

La responsabilidad de crear un objeto o su implementación, debe realizarla la clase que sabe toda la información necesaria para hacerlo, de este modo obtenemos una mayor cohesión la información esta encapsulada disminuyendo el acoplamiento.

6.- Fabricación pura

Práctica que ayuda con una clase poco cohesiva y no tiene otra clase donde implementar algunos métodos.

Crea una clase “inventada” pero que al añadirla mejora la estructura del sistema.

7.- Indirección

La indirecta admite un bajo acoplamiento, pero reduce la legibilidad y el razonamiento sobre todo el sistema. No sabe qué clase maneja el comando de la definición del controlador. Esta es la compensación a tener en cuenta.