



Universidad Don Bosco, El Salvador

EXPERIENCIA DE APRENDIZAJE 5

Principios SOLID (4%)

Ingeniero: Alexander Siguenza

Alumnos:

Apellido	Nombre	N de carné
Guerrero Zelaya	Diego Benjamín	GZ210369
Novoa Velásquez	Yesenia Nicole	NV210134

Fecha de entrega:

10 de junio del 2023

Esta actividad estaría reponiendo el 4% restante de las U4 Y Y5 de las preguntas de Retroalimentación.

PRINCIPIOS SOLID


Los 5 principios SOLID de diseño de aplicaciones de software son:

- **S** – Single Responsibility Principle (**SRP**)
- **O** – Open/Closed Principle (**OCP**)
- **L** – Liskov Substitution Principle (**LSP**)
- **I** – Interface Segregation Principle (**ISP**)
- **D** – Dependency Inversion Principle (**DIP**)



Los objetivos principales para tener en cuenta de los 5 principios a la hora de escribir código tenemos:

1. Crear un software eficaz: que cumpla con su cometido y que sea robusto y estable.
2. Escribir un código limpio y flexible ante los cambios: que se pueda modificar fácilmente según necesidad, que sea reutilizable y mantenible.
3. Permitir escalabilidad: que acepte ser ampliado con nuevas funcionalidades de manera ágil.

 **Principio de Responsabilidad Única:** Este principio establece que un componente o clase debe tener una responsabilidad única, sencilla y concreta. Esto ayuda a simplificar el código al evitar que existan clases que cumplan con múltiples funciones.

Esto significa que, si una clase es un contenedor de datos, como una clase Libro o una clase Estudiante, y tiene algunos campos relacionados con esa entidad, debería cambiar solo cuando cambiamos el modelo de datos. Facilita el control de versiones. Por ejemplo, digamos que tenemos una clase de persistencia que maneja las operaciones de la base de datos y vemos un cambio en ese archivo en las confirmaciones de GitHub. Al seguir el PRU, sabremos que está relacionado con

el almacenamiento o con cosas relacionadas con la base de datos.

- ✚ **Principio de Abierto/Cerrado:** Este principio establece que los componentes del software deben estar abiertos para extender a partir de ellos, pero cerrados para evitar que se modifiquen.

Se puede agregar nuevas funciones sin tocar el código existente para la clase. Esto se debe a que cada vez que modificamos el código existente, corremos el riesgo de crear errores potenciales. Se puede modificar utilizando interfaces y clases abstractas.

- ✚ **Principio de Sustitución de Liskov:** Este principio establece que una subclase puede ser sustituida por su superclase. Podemos crear una subclase llamada Auto, la cual deriva de la superclase Vehículo. Si al usar la superclase el programa falla, este principio no se cumple. cuando una clase no obedece este principio, genera algunos errores desagradables que son difíciles de detectar.

- ✚ **Principio de Segregación de Interfaces:** Este principio establece que los clientes no deben ser forzados a depender de interfaces que no utilizan. Es importante que cada clase implemente las interfaces que va a utilizar. De este modo, agregar nuevas funcionalidades o modificar las existentes será más fácil. La segregación significa mantener las cosas separadas, y el Principio de Segregación de Interfaces se trata de separar las interfaces.

- ✚ **Principio de Inversión de Dependencias:** Este principio establece que los módulos de alto nivel no deben depender de los de bajo nivel. En ambos casos deben depender de las abstracciones. Alto nivel se refiere a operaciones cuya naturaleza es más amplia o abarca un

contexto más general y bajo nivel son componentes individuales más específicos. El principio de inversión de dependencia establece que nuestras clases deben depender de interfaces o clases abstractas en lugar de clases y funciones concretas.

Ejemplo

Supongamos que estás desarrollando una aplicación de clima que muestra la temperatura actual y el pronóstico para los próximos días. Para aplicar el principio de Responsabilidad Única, puedes separar las responsabilidades relacionadas con la obtención de datos y la presentación de la información en diferentes componentes.

1. Obtención de datos:

Crea un componente llamado WeatherAPI que se encargue exclusivamente de obtener los datos del clima desde una API externa. Este componente puede utilizar una biblioteca como axios para realizar la solicitud HTTP.

```
1  import axios from 'axios';
2
3  const WeatherAPI = async () => {
4    try {
5      const response = await axios.get('https://api.weather.com/forecast');
6      return response.data;
7    } catch (error) {
8      console.error('Error al obtener los datos del clima:', error);
9      return null;
10   }
11 };
12
13 export default WeatherAPI;
14
```

2. Presentación de la información:

Crea otro componente llamado WeatherDisplay este se encargará de mostrar toda la información del clima en pantalla. Este componente recibe los datos del clima como una “prop” y los renderiza de manera adecuada.

```

1  import React from 'react';
2  import { Text, View } from 'react-native';
3
4  const WeatherDisplay = ({ weatherData }) => {
5    if (!weatherData) {
6      return <Text>No se pudieron obtener los datos del clima.</Text>;
7    }
8
9    return (
10     <View>
11       <Text>Temperatura actual: {weatherData.currentTemperature}°C</Text>
12       <Text>Pronóstico para los próximos días:</Text>
13       {weatherData.forecast.map((day) => (
14         <Text key={day.date}>
15           {day.date}: {day.temperature}°C
16         </Text>
17       ))}
18     </View>
19   );
20 };
21
22 export default WeatherDisplay;
23

```

3. Componente principal:

Por último, crea un componente llamado App que integre los componentes WeatherAPI y WeatherDisplay. Este componente manejará el estado de los datos del clima y la lógica de obtener y pasar los datos al componente de visualización.

```

1  import React, { useEffect, useState } from 'react';
2  import WeatherAPI from './WeatherAPI';
3  import WeatherDisplay from './WeatherDisplay';
4
5  const App = () => {
6    const [weatherData, setWeatherData] = useState(null);
7
8    useEffect(() => {
9      const fetchData = async () => {
10        const data = await WeatherAPI();
11        setWeatherData(data);
12      };
13
14      fetchData();
15    }, []);
16
17    return <WeatherDisplay weatherData={weatherData} />;
18  };
19
20 export default App;

```