Texto

Descripción generada automáticamente

**Nombre del trabajo: Investigación - Principios SOLID - 4%**

**Materia: Diseño y Programación de Software Multiplataforma DPS941 G01T**

**Alumno: María Dolores Martínez León**

**Docente: Ing. Alexander Alberto Siguenza Campos**

**Fecha de entrega:10/6/2023**

**Los cinco principios SOLID:**

* Principio de Responsabilidad Única,
* Principio de Abierto/Cerrado,
* Principio de Sustitución de Liskov,
* Principio de Segregación de Interfaces
* Principio de Inversión de Dependencias.

Los principios SOLID aplicados en el desarrollo de aplicaciones con React Native.

**Principio de Responsabilidad Única (SRP - Single Responsibility Principle):**

Este principio establece que una clase o componente debe tener una única responsabilidad. Significa que cada clase debe tener un único motivo para cambiar. Al seguir este principio, se logra un código más modular y cohesivo. Cada componente se enfoca en una tarea específica y no tiene dependencias innecesarias. Esto facilita el mantenimiento del código, ya que los cambios en una responsabilidad no afectan a otras partes del sistema.

**Principio de Abierto/Cerrado (OCP - Open/Closed Principle):**

El principio de abierto/cerrado establece que las entidades de software deben estar abiertas para la extensión pero cerradas para la modificación. Esto significa que el código debe ser flexible para permitir la adición de nuevas funcionalidades sin tener que modificar el código existente. Al aplicar este principio, se fomenta la reutilización de código y se reduce el riesgo de introducir errores en áreas ya funcionales. Además, facilita la evolución y el mantenimiento del software a medida que los requisitos cambian.

**Principio de Sustitución de Liskov (LSP - Liskov Substitution Principle):**

El principio de sustitución de Liskov establece que los objetos de una clase base deben ser reemplazables por instancias de sus clases derivadas sin alterar la integridad del programa. Esto significa que una clase derivada debe poder usarse en lugar de la clase base sin que se rompa el funcionamiento del programa. Al seguir este principio, se garantiza la consistencia del comportamiento de los objetos en todo el sistema y se evitan sorpresas o comportamientos inesperados al utilizar la herencia.

**Principio de Segregación de Interfaces (ISP - Interface Segregation Principle):**

El principio de segregación de interfaces establece que los clientes no deben depender de interfaces que no utilicen. Propone que las interfaces sean cohesivas y proporcionen solo los métodos necesarios para cada cliente. Esto evita la dependencia de código innecesario y reduce el acoplamiento entre componentes. Al aplicar este principio, se mejora la modularidad del código y se facilita la evolución y el mantenimiento, ya que los cambios en una interfaz no afectarán a los clientes que no la utilizan.

**Principio de Inversión de Dependencias (DIP - Dependency Inversion Principle):**

El principio de inversión de dependencias establece que los módulos de alto nivel no deben depender de los módulos de bajo nivel, sino de abstracciones. Esto implica que los componentes de nivel superior deben depender de interfaces o abstracciones en lugar de depender directamente de los componentes de nivel inferior. Al aplicar este principio, se logra un código más flexible, modular y fácil de mantener. Permite intercambiar fácilmente las implementaciones de bajo nivel sin afectar a los componentes de alto nivel, lo que facilita las pruebas y la reutilización del código.

La importancia de aplicar estos principios SOLID radica en lograr un código limpio, mantenible y escalable. Al seguir estos principios, se mejora la estructura del código, se reducen las dependencias innecesarias y se facilita la evolución del software a medida que los requisitos

**Ejemplo Practico**

**El principio de Responsabilidad Única (SRP) en React Native**

Como ejemplo suponemos que estamos desarrollando una aplicación de notas en React Native. tenemos un componente llamado NoteItem que se encarga de mostrar una nota en la interfaz de usuario. Actualmente, este componente también se encarga de manejar la lógica para eliminar una nota cuando se realiza una acción de deslizamiento hacia la izquierda.

Al aplicar el principio de Responsabilidad Única, separamos la responsabilidad de mostrar la nota y la lógica de eliminación en dos componentes diferentes.

Aquí está el ejemplo:

*// NoteItem.js*

*import React from 'react';*

*import { View, Text, TouchableOpacity } from 'react-native';*

*const NoteItem = ({ note, onDeleteNote }) => {*

*return (*

*<TouchableOpacity onPress={() => onDeleteNote(note.id)}>*

*<View>*

*<Text>{note.title}</Text>*

*<Text>{note.content}</Text>*

*</View>*

*</TouchableOpacity>*

*);*

*};*

*export default NoteItem;*

*// NoteItemDeleteButton.js*

*import React from 'react';*

*import { TouchableOpacity, Text } from 'react-native';*

*const NoteItemDeleteButton = ({ onDeleteNote }) => {*

*return (*

*<TouchableOpacity onPress={onDeleteNote}>*

*<Text>Delete</Text>*

*</TouchableOpacity>*

*);*

*};*

*export default NoteItemDeleteButton;*

En el ejemplo anterior, hemos creado dos componentes: NoteItem y NoteItemDeleteButton. El componente NoteItem se encarga únicamente de mostrar la nota en la interfaz de usuario. Por otro lado, el componente NoteItemDeleteButton se encarga exclusivamente de mostrar el botón de eliminación y manejar la lógica para eliminar la nota.

Al dividir estas responsabilidades, cumplimos con el principio de Responsabilidad Única. El componente NoteItem ahora se centra solo en presentar la nota, mientras que el componente NoteItemDeleteButton se centra únicamente en la lógica de eliminación. Esto hace que el código sea más modular, fácil de entender y mantener.

Además, al separar la lógica de eliminación en un componente independiente, podemos reutilizar fácilmente este botón de eliminación en otros lugares de nuestra aplicación si es necesario.

Este es solo un ejemplo de cómo aplicar el principio de Responsabilidad Única en React Native. La idea es separar las responsabilidades de los componentes para que cada uno se encargue de una única tarea, lo que mejora la legibilidad, el mantenimiento y la reutilización del código.

**El principio de Inversión de Dependencias (DIP) en React Native**

Otro ejemplo, suponiendo que estámos desarrollando una aplicación de gestión de tareas en React Native. Tenemos un componente llamado TaskList que muestra una lista de tareas en la interfaz de usuario. Actualmente, este componente depende directamente de una implementación concreta de una clase TaskService para obtener las tareas desde una API.

Para aplicar el principio de Inversión de Dependencias, debemos invertir la dependencia de TaskList hacia una abstracción, en lugar de depender directamente de una implementación concreta.

Aquí está el ejemplo:

*// TaskList.js*

*import React, { useEffect, useState } from 'react';*

*import { View, Text } from 'react-native';*

*import { TaskService } from './services';*

*const TaskList = ({ taskService }) => {*

*const [tasks, setTasks] = useState([]);*

*useEffect(() => {*

*const fetchTasks = async () => {*

*const fetchedTasks = await taskService.getTasks();*

*setTasks(fetchedTasks);*

*};*

*fetchTasks();*

*}, []);*

*return (*

*<View>*

*{tasks.map((task) => (*

*<Text key={task.id}>{task.title}</Text>*

*))}*

*</View>*

*);*

*};*

*export default TaskList;*

*// TaskService.js*

*class TaskService {*

*async getTasks() {*

*// Lógica para obtener las tareas desde una API*

*const response = await fetch('https://api.example.com/tasks');*

*const data = await response.json();*

*return data;*

*}*

*}*

*export default TaskService;*

En el ejemplo anterior, hemos creado una clase TaskService que encapsula la lógica para obtener las tareas desde una API. En lugar de depender directamente de esta implementación concreta en el componente TaskList, hemos introducido una abstracción a través del parámetro taskService. Esto permite que TaskList sea independiente de la implementación concreta de TaskService.

Al invertir la dependencia, podemos proporcionar diferentes implementaciones de TaskService según sea necesario. Por ejemplo, podríamos crear una implementación de TaskService para obtener las tareas desde el almacenamiento local en lugar de una API, y simplemente pasársela al componente TaskList.

Esta inversión de dependencias facilita la prueba y la reutilización del componente TaskList, ya que podemos proporcionar implementaciones simuladas de TaskService durante las pruebas y cambiar fácilmente la fuente de datos sin modificar el componente principal.

El principio de Inversión de Dependencias se basa en depender de abstracciones en lugar de implementaciones concretas. Al aplicar este principio, logramos un código más flexible, modular y fácil de mantener, permitiendo una mayor escalabilidad y facilitando las pruebas y la evolución del software.

**El principio de Abierto/Cerrado (OCP) en React Native:**

Supongamos que estamos desarrollando una aplicación de comercio electrónico en React Native. Tenemos un componente llamado ProductCard que se encarga de mostrar los detalles de un producto en la interfaz de usuario. Actualmente, este componente tiene la funcionalidad de agregar productos al carrito de compras.

Sin embargo, en lugar de modificar directamente el código del componente ProductCard cada vez que se requiere agregar una nueva funcionalidad, como añadir un producto a la lista de deseos, podemos aplicar el principio de Abierto/Cerrado para extender la funcionalidad sin modificar el componente existente.

Aquí está el ejemplo:

*// ProductCard.js*

*import React from 'react';*

*import { View, Text, TouchableOpacity } from 'react-native';*

*const ProductCard = ({ product, onAddToCart }) => {*

*return (*

*<View>*

*<Text>{product.name}</Text>*

*<Text>{product.price}</Text>*

*<TouchableOpacity onPress={() => onAddToCart(product)}>*

*<Text>Add to Cart</Text>*

*</TouchableOpacity>*

*</View>*

*);*

*};*

*export default ProductCard;*

*// WishlistButton.js*

*import React from 'react';*

*import { TouchableOpacity, Text } from 'react-native';*

*const WishlistButton = ({ product, onAddToWishlist }) => {*

*return (*

*<TouchableOpacity onPress={() => onAddToWishlist(product)}>*

*<Text>Add to Wishlist</Text>*

*</TouchableOpacity>*

*);*

*};*

*export default WishlistButton;*

En el ejemplo anterior, hemos creado un componente separado llamado WishlistButton que se encarga exclusivamente de la funcionalidad de añadir un producto a la lista de deseos. Este componente puede ser utilizado en otros lugares de la aplicación donde sea necesario agregar productos a la lista de deseos, sin necesidad de modificar el componente ProductCard original.

Al seguir el principio de Abierto/Cerrado, hemos mantenido el componente ProductCard cerrado para modificaciones, pero abierto para la extensión. Si en el futuro se requiere agregar más funcionalidades, como añadir a comparación o compartir en redes sociales, simplemente creamos nuevos componentes independientes y los integramos donde sea necesario sin modificar el código existente.

Al aplicar este principio, logramos un código más modular, mantenible y escalable. Cada componente se enfoca en una única responsabilidad, y la extensión de funcionalidades se realiza a través de la composición de componentes, lo que facilita la reutilización, el mantenimiento y la evolución del software.

**Referencias**

* <https://reactnative.dev/>
* <https://scholar.google.com/>
* <https://ieeexplore.ieee.org/>
* <https://pubmed.ncbi.nlm.nih.gov/>