“UNIVERSIDAD DON BOSCO”

Tema:
“Principios SOLID”

Asignatura:
Diseño y Programación de Software Multiplataforma

Facultad:
Ing. En Ciencias De La Computación (Virtual)

Integrante

Gabriela María Pineda González	**PG120866**
Martínez Alvarado Yosselin Beatriz	**MM110166**
Cabezas Vaquero Gerardo Antonio	**CV152055**

Docente:
Ing. Alexander Sigüenza

Grupo Teórico:
G01 T

Contenido

INTRODUCCIÓN:

En el ámbito del desarrollo de aplicaciones con React Native, es fundamental comprender y aplicar los principios SOLID. Estos principios proporcionan pautas clave para el diseño de software de calidad, permitiendo crear aplicaciones más limpias, mantenibles y escalables. Con la creciente popularidad de React Native como framework para el desarrollo móvil, es crucial conocer cómo aplicar estos principios en este contexto específico.

El objetivo de esta investigación es explorar y comprender en profundidad los cinco principios SOLID y su aplicación en el desarrollo de aplicaciones con React Native. Mediante la comprensión de estos principios, podremos mejorar nuestras habilidades de diseño y desarrollo, generando código más robusto y eficiente.

OBJETIVOS:

Comprender los cinco principios SOLID (Principio de Responsabilidad Única, Principio de Abierto/Cerrado, Principio de Sustitución de Liskov, Principio de Segregación de Interfaces y Principio de Inversión de Dependencias) y su relevancia en el desarrollo de software.

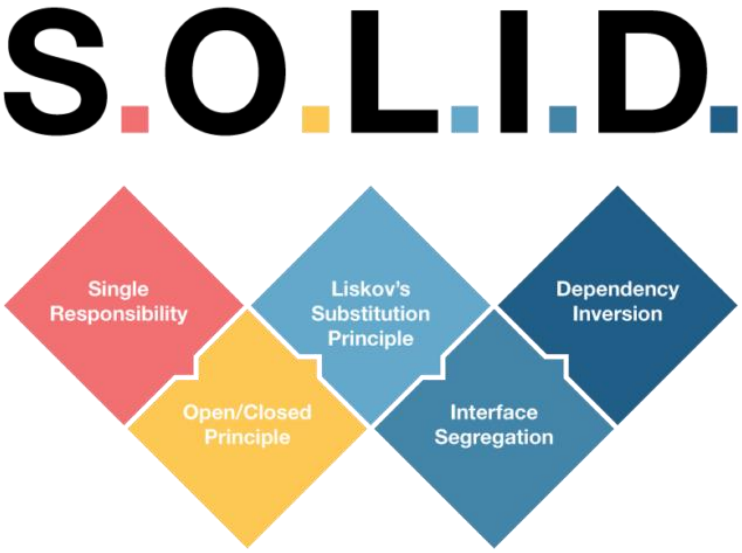
Analizar y describir en detalle cada principio SOLID, explorando su significado y propósito específico en el contexto de React Native.

Destacar la importancia de aplicar los principios SOLID en el desarrollo de aplicaciones con React Native, enfatizando los beneficios de un código limpio, mantenible y escalable.

Presentar ejemplos prácticos de aplicación de al menos uno de los principios SOLID en el desarrollo de aplicaciones móviles con React Native. Estos ejemplos deben ser claros y fáciles de entender, demostrando cómo se pueden implementar los principios SOLID en la práctica.

Proporcionar una investigación organizada y bien estructurada, utilizando fuentes confiables y citando adecuadamente cualquier recurso utilizado. Esto asegurará la validez y credibilidad de la información presentada.

PRINCIPIOS SOLID



PRINCIPIO DE RESPONSABILIDAD ÚNICA (SINGLE RESPONSIBILITY PRINCIPLE):

El principio de Responsabilidad Única establece que una clase o módulo debe tener una sola responsabilidad. Esto significa que una clase debe tener un único motivo para cambiar. Al adherirse a este principio, se logra un código más modular y cohesivo, ya que cada clase se enfoca en una tarea específica. Esto facilita el mantenimiento, ya que los cambios relacionados con una responsabilidad no afectarán a otras partes del sistema. Además, la claridad de las responsabilidades mejora la comprensión y reutilización del código.

Aplicación en React Native: En el desarrollo de aplicaciones con React Native, este principio implica que cada componente debe ser responsable de una única funcionalidad o comportamiento específico. Esto mejora la modularidad y facilita el mantenimiento del código, ya que cada componente tiene un propósito claro y bien definido.

Por ejemplo, imaginemos que queremos implementar el registro de un nuevo usuario a nuestra aplicación:

```
1class UserController
2{
3    function register(array $data)
4    {
5        if (isset($data['password'])) {
6            $data['password'] = password_hash($data['password'], PASSWORD_DEFAULT);
7        }
8        User::create($data);
9    }
10}
```

Podemos destacar que en la función register primero creamos un hash para la contraseña y después el usuario, pero si quisiéramos cumplir este principio deberíamos crear una clase para crear el hash.

```
1class PasswordHasher
2{
3    static function hash($password): string
4    {
5        return password_hash($data['password'], PASSWORD_DEFAULT);
6    }
7}
```

```
1class UserController
2{
3    function register(array $data):
4    {
5        if (isset($data['password'])) {
```

```
 6            PasswordHasher::hash($data['password']))
 7        }
 8        User::create($data);
 9    }
10}
```

De esta forma hemos separado la lógica de encriptación de contraseña de la creación del usuario, así cada una tiene su función o responsabilidad facilitando su cambio, mantenimiento o extensión en un futuro.

PRINCIPIO DE ABIERTO/CERRADO (OPEN/CLOSED PRINCIPLE):

El principio de Abierto/Cerrado establece que las entidades del software deben estar abiertas para su extensión pero cerradas para su modificación. Esto significa que el código existente no debe modificarse para agregar nuevas funcionalidades, en su lugar, se deben introducir nuevas clases o módulos que extiendan el comportamiento existente. Al seguir este principio, se evita el riesgo de introducir errores en el código existente y se promueve la reutilización de componentes. Esto facilita la evolución del software sin comprometer su estabilidad.

Aplicación en React Native: En el desarrollo de aplicaciones con React Native, se busca que los componentes sean fácilmente extensibles sin necesidad de modificar el código existente. Esto se logra utilizando técnicas como la herencia, la composición y la creación de componentes reutilizables. De esta manera, se pueden agregar nuevas funcionalidades sin alterar el código base, lo que mejora la estabilidad y la escalabilidad del sistema.

Por ejemplo, si quisiéramos enviar distintas notificaciones por diferentes canales a nuestros usuarios:

```
1class NotificationController
2{
3    function send(array $notifications)
4    {
5        foreach ($notifications as $notification) {
6            $type = $notification['type'];
7            if ($type == 'email') {
8                $this->sendEmailNotification();
9            } elseif ($type == 'telegram') {
10               $this->sendTelegramNotification();
11           }
12       }
13   }
14}
```

Pero si en futuro necesitamos enviar por Slack, Whatsapp o sms tendriamos que modificar la función send, sin embargo si aplicamos dicho principio crearíamos la interfaz NotificationSender

```
1interface NotificationSender
2{
3    function notify(array $data);
4}
```

y los diferentes canales implementarían la interfaz

```php
class EmailSender implements NotificationService
{
    function notify(array $data)
    {
        $this->send($data);
    }
}
```

```php
class TelegramSender implements NotificationService
{
    function notify(array $data)
    {
        $this->send($data);
    }
}
```

De esta forma nuestro controlador quedaría abierto para nuevas modificaciones sin tener que modificarlo directamente, favoreciendo su mantenimiento y escalabilidad en el tiempo.

```php
class NotificationController
{
    function send(array $notifications)
    {
        foreach ($notifications as $notification) {
            $notification->notify();
        }
    }
}
```

PRINCIPIO DE SUSTITUCIÓN DE LISKOV (LISKOV SUBSTITUTION PRINCIPLE):

El principio de Sustitución de Liskov establece que las clases derivadas deben ser sustituibles por sus clases base sin alterar el funcionamiento del programa. Esto implica que las clases hijas deben cumplir con las mismas expectativas y contratos que sus clases padres. Al adherirse a este principio, se garantiza la coherencia del programa y se evitan comportamientos inesperados cuando se utilizan polimorfismo. Esto facilita el mantenimiento y la extensibilidad del código.

Aplicación en React Native: En el desarrollo de aplicaciones con React Native, se busca garantizar que los componentes derivados puedan utilizarse en lugar de los componentes base sin causar efectos adversos en el comportamiento del sistema. Esto implica que los componentes derivados deben cumplir con las mismas interfaces y contratos que los componentes base, para que puedan ser intercambiables sin problemas.

Por ejemplo imaginemos una clase pato:

```
1class Duck
2{
3    function fly() {}
4    function swim() {}
5    function cuack() {}
6}
```

En la que sus hijos heredarían de ella los métodos fly, swim y cuack. Pero si quisiéremos crear una clase pato de goma heredando de la clase pato anterior no se cumpliría dicho principio ya que cuando usemos dichas clases tendremos que añadir una lógica adicional, por ejemplo:

```
1class RubberDuck
2{
3    function fly() {
4        throw new \Exception("can't fly");
5    }
6    function swim() {}
7    function cuack() {}
8}
```

```php
class DuckProcesser
{
    function makeDucksFly(array $ducks) {
        foreach($ducks as $duck) {
            try {
                $duck->fly();
            } catch (\Exception $e) {
                throw $e;
            }
        }
    }
}
```

Para cumplir este principio podemos rediseñar las clases creando interfaces para cada acción de nuestro pato:

```php
interface IFly
{
    function fly(): void;
}

interface ICuack
{
    function cuack(): void;
}

interface ISwim
{
    function swim(): void;
}

class RubberDuck implements ICuak, ISwim
{
    function swim() {}
    function cuack() {}
}

class Duck implements ICuak, ISwim, IFly
{
    function fly() {}
    function swim() {}
    function cuack() {}
}
```

```php
class DuckProcesser
{
    function makeDucksFly(array $ducks) {
        foreach($ducks as $duck) {
            $duck->fly();
        }
    }
}
```

De esta forma cada tipo de pato implementa lo que puede hacer, este diseño es aún más escalable en el tiempo y nos evitaría tener funciones que no hicieran nada como en el primer diseño propuesto que incumplía el principio.

Principio de Segregación de Interfaces (Interface Segregation Principle):

El principio de Segregación de Interfaces establece que las interfaces de un sistema deben ser específicas para los clientes que las utilizan. En lugar de tener una única interfaz grande y genérica, se deben tener varias interfaces más pequeñas y especializadas. Esto permite que los clientes solo dependan de las interfaces que necesitan, evitando la dependencia de funcionalidades innecesarias. Al aplicar este principio, se reduce el acoplamiento entre los componentes del sistema y se promueve la cohesión y la modularidad.

Aplicación en React Native: En el desarrollo de aplicaciones con React Native, se busca dividir las interfaces en partes más pequeñas y específicas, adaptadas a las necesidades de los diferentes clientes o componentes que las utilizan. Esto evita la dependencia de interfaces no utilizadas y promueve la cohesión y el modularidad del código. Al definir interfaces más específicas, se reduce la complejidad y se mejora la reutilización de componentes.

Supongamos un sistema de impresión en que la clase Job se encargará de imprimir, grapar las hojas, enviar fax…

```
1class Job
2{
3    function print() {}
4    function fax() {}
5    function staple() {}
6    ...
7}
```

El problema de este diseño se presentaría cada vez que quisiéramos añadir nuevas funcionalidades o refactorizar las existentes de nuestro sistema. Cada cambio se volvería más costoso que el anterior, sin embargo si aplicamos este principio podemos mejorar el diseño creando clases que especializan cada tipo de trabajo:

```
1class PrintJob extend Job
2{
3    function print() {}
4}
```

```
1class StapleJob extend Job
2{
3    function staple() {}
4}
```

```
1class FaxJob extend Job
2{
3    function fax() {}
4}
```

```
1private class Job
2{
3    ...
4}
```

De esta forma quedan definidas mucho mejor las necesidades, y el sistema tiene mayor escalabilidad. No se podría utilizar la clase Job, pero sí las clases que extienden de ella, definidas para desempeñar funciones específicas

PRINCIPIO DE INVERSIÓN DE DEPENDENCIAS (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 directamente, sino a través de abstracciones. Además, las abstracciones no deben depender de los detalles, sino que los detalles deben depender de las abstracciones. Al seguir este principio, se logra una arquitectura flexible y desacoplada, donde los cambios en los módulos de bajo nivel no afectan a los módulos de alto nivel. Esto permite la fácil sustitución de componentes y facilita las pruebas y el mantenimiento del código.

Aplicación en React Native: En el desarrollo de aplicaciones con React Native, se busca evitar las dependencias directas entre los diferentes componentes y módulos. En su lugar, se utilizan abstracciones o interfaces para establecer una capa de indirección entre los componentes. Esto permite una mayor flexibilidad y facilita la modificación y el reemplazo de componentes

Es decir las implementaciones concretas no deberían depender de otras implementaciones concretas si no de capas abstractas. Veámoslo en el siguiente ejemplo:

```
 1class MySQLConnection
 2{
 3    public function connect()
 4    {
 5        // handle the database connection
 6        return 'Database connection';
 7    }
 8}
 9
10class PasswordReminder
11{
12    private $dbConnection;
13
14    public function __construct(MySQLConnection $dbConnection)
15    {
16        $this->dbConnection = $dbConnection;
17    }
18}
```

Vemos que la clase PasswordReminder (alto nivel) depende directamente de MySQLConnection (bajo nivel), pero si en futuro quisiéramos cambiar nuestro gestor de base de datos este diseño no sería optimo, pero si aplicamos el principio sí:

```php
interface DBConnectionInterface
{
    public function connect();
}

class MySQLConnection implements DBConnectionInterface
{
    public function connect()
    {
        // handle the database connection
        return 'Database connection';
    }
}

class PasswordReminder
{
    private $dbConnection;

    public function __construct(DBConnectionInterface $dbConnection)
    {
        $this->dbConnection = $dbConnection;
    }
}
```

De esta forma nuestro sistema depende de una abstracción lo que nos ofrece poder cambiar fácilmente nuestro gestor de base de datos.

CONCLUSIÓN

En conclusión, la aplicación de los principios SOLID en el desarrollo de aplicaciones con React Native es fundamental para crear software de calidad y garantizar la escalabilidad y mantenibilidad a largo plazo. A lo largo de esta investigación, hemos explorado en detalle los cinco principios SOLID: Principio de Responsabilidad Única, Principio de Abierto/Cerrado, Principio de Sustitución de Liskov, Principio de Segregación de Interfaces y Principio de Inversión de Dependencias, comprendiendo su significado y propósito en el contexto de React Native.

Hemos destacado la importancia de aplicar estos principios para lograr un código limpio y estructurado, que sea fácil de mantener y evolucionar. Al adherir al principio de Responsabilidad Única, podemos asegurar que cada componente o módulo tenga una única responsabilidad, lo que facilita su comprensión y reutilización. El principio de Abierto/Cerrado nos permite extender la funcionalidad sin modificar el código existente, lo que reduce el riesgo de introducir errores.

El principio de Sustitución de Liskov nos guía en la creación de clases que puedan ser sustituidas por sus subclases sin afectar la integridad del programa. Mediante el principio de Segregación de Interfaces, evitamos la dependencia de interfaces demasiado generales y promovemos interfaces específicas para cada cliente. Por último, el principio de Inversión de Dependencias nos ayuda a desacoplar componentes y a depender de abstracciones en lugar de implementaciones concretas.

Además, hemos presentado ejemplos prácticos que demuestran cómo aplicar estos principios en la práctica en el desarrollo de aplicaciones con React Native.

Estos ejemplos nos han mostrado cómo mejorar la estructura y la modularidad del código, facilitando el mantenimiento y fomentando la reutilización de componentes.

REFERENCIAS:

- Martin, R. C. (2003). Agile software development: principles, patterns, and practices. Prentice Hall. En este libro, Robert C. Martin (también conocido como Uncle Bob) explora los principios SOLID y su aplicación en el desarrollo de software ágil.
- Freeman, A., Robson, E., Bates, B., & Sierra, K. (2021). Head First Design Patterns: A Brain-Friendly Guide. O'Reilly Media. Este libro aborda los principios SOLID y su relación con los patrones de diseño en el desarrollo de software. Proporciona ejemplos prácticos y fáciles de entender.
- Chinnathambi, A. (2017). React Native Blueprints. Packt Publishing. Este libro se centra en el desarrollo de aplicaciones móviles con React Native. Si bien no se centra específicamente en los principios SOLID, ofrece información valiosa sobre la implementación de buenas prácticas en el desarrollo de aplicaciones con React Native.
- "The SOLID Principles in React Native" por Cory House. Disponible en: https://www.pluralsight.com/guides/the-solid-principles-in-react-native Este artículo en línea aborda los principios SOLID en el contexto de React Native, proporcionando ejemplos y explicaciones claras de cómo aplicar estos principios en el desarrollo de aplicaciones móviles con React Native.
- "SOLID Principles with React Native" por Maximiliano Contieri. Disponible en: https://www.amazon.com/SOLID-Principles-React-Native-Contieri-ebook/dp/B09831GMH9 Este libro se enfoca específicamente en la aplicación de los principios SOLID en el desarrollo de aplicaciones con React Native. Ofrece ejemplos prácticos y consejos para implementar estos principios de manera efectiva.