UNIVERSIDAD DON BOSCO

Asignatura:

Diseño y Programación de Software Multiplataforma

Estudiantes:

Jairo Jubynni Soto García SG080419

Ronald Gerardo Ramírez Guardado RG110604

Alejandro Ernesto Velasco Crespín VC161941

Actividad

Principios SOLID

Índice

Introducción

SOLID es un conjunto de principios de diseño de software que se utilizan ampliamente en el desarrollo de aplicaciones para mejorar la calidad, la mantenibilidad y la escalabilidad del código. Estos principios fueron introducidos por el reconocido ingeniero de software Robert C. Martin, también conocido como "Uncle Bob".

La palabra "SOLID" es un acrónimo que representa cinco principios individuales:

- **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)**

En este documento se brindará un resumen de cada uno de los principios.

Principios SOLID

S – Single Responsibility Principle (SRP)

O conocido en español como **"Principio de Responsabilidad Única",** el cual establece que una clase o componente solo debe tener una única responsabilidad, es decir que cada componente tendrá una sola función o tarea claramente definida y no deberá tener responsabilidades adicionales no relacionadas a la tarea principal.

Ejemplo:

Un ejemplo típico es el de un objeto que necesita ser renderizado de alguna forma, por ejemplo, imprimiéndose por pantalla. Podríamos tener una clase como esta:

```
1.  class Vehicle(
2.      val wheelCount: Int,
3.      val maxSpeed: Int
4.  ) {
5.      fun print() {
6.          println("wheelCount=$wheelCount, maxSpeed=$maxSpeed")
7.      }
8.  }
```

Como se observa en el código, se mezclan dos conceptos, los cuales son la lógica del negocio y la lógica de presentación, lo cual daría complicaciones al querer, por ejemplo, mostrar los resultados de manera diferente, o si queremos mostrar dos datos no se podría al tener un solo Print( ), entonces utilizando el principio, el código quedaría asi:

```
1.  class VehiclePrinter {
2.      fun print(vehicle: Vehicle) {
3.          println(
4.              "wheelCount=${vehicle.wheelCount}, " +
5.                      "maxSpeed=${vehicle.maxSpeed}"
6.          )
7.      }
```

En donde ya se tiene de una manera separada la lógica para imprimir el resultado.

O – Open/Closed Principle (OCP)

En español **“Principio de Abierto/Cerrado”,** sugiere que el software, ya sea clases, componentes, módulos, entre otros, deben ser abiertas para su extensión, pero cerradas para su modificación. Esto significa que los componentes deben poder extenderse y personalizarse sin necesidad de modificar el código fuente original.

Ejemplo:

Imaginemos que tenemos una clase con un método que se encarga de dibujar un vehículo por pantalla. Por supuesto, cada vehículo tiene su propia forma de ser pintado. Nuestro vehículo tiene la siguiente forma:

```
1.  class Vehicle(val type: VehicleType)
```

Es una clase que especifica el tipo del vehículo mediante un enumerado, asi:

```
1.  enum class VehicleType{
2.      CAR, MOTORBIKE
3.  }
```

Y el método de la clase con la lógica de pintura es:

```
1.  fun draw(vehicle: Vehicle) {
2.      when(vehicle.type){
3.          VehicleType.CAR -> drawCar(vehicle)
4.          VehicleType.MOTORBIKE -> drawMotorbike(vehicle)
5.      }
6.  }
```

Pero esto dificulta si en algún punto se requiere agregar mas tipos a ese enumerado, lo que implicaría un nuevo case y un nuevo método para dibujarlo, pero al aplicar el principio de Open/Closed se puede aplicar mediante herencia o polimorfismo, y el código quedaría asi:

```
interface Vehicle {
    fun draw()
}

class Car : Vehicle {
    override fun draw() {
        // Draw the car
    }
}

class Motorbike : Vehicle {
    override fun draw() {
        // Draw the motorbike
    }
}
```

Y el método anterior queda reducido a:

```
fun draw(vehicle: Vehicle) {
    vehicle.draw()
}
```

Y el añadir nuevos tipos es tan sencillo como crear la clase correspondiente a vehicle:

```
class Truck: Vehicle {
    override fun draw() {
        // Draw the truck
    }
}
```

L – Liskov Substitution Principle (LSP)

En español **“Principio de Sustitución de Liskov”**, Este principio establece que los objetos de un programa deben ser reemplazables por instancias de sus subtipos sin afectar la corrección del programa. Esto se traduce en asegurarse de que los componentes heredados o subclases se puedan utilizar en lugar de los componentes base sin causar errores o comportamientos inesperados

Ejemplo:

Supongamos que tenemos una clase Animal, que representa a un animal, y tiene propiedades como caminar o saltar:

```
1.  open class Animal {
2.      open fun walk() { ... }
3.      open fun jump() { ... }
4.  }
```

Y se requiere, en un momento determinado, hacer que el animal salte:

```
1.  fun jumpHole(a: Animal){
2.      a.walk()
3.      a.jump()
4.      a.walk()
5.  }
```

Y tomamos como el animal a saltar a un elefante, el cual sabemos que no puede hacerlo, pero utilizamos una excepción para detectar si el hecho ocurre:

```
1.  class Elephant : Animal() {
2.      override fun jump() =
3.          throw Exception("Los elefantes no pueden saltar")
4.
5.  }
```

Pero con esto se llega a un punto en donde se utilice el jumpHole, si el animal es un elefante se tendría una excepción, y esto se soluciona si creamos un tipo de animal de peso ligero que, si puede saltar, de esta manera:

```
1.  open class Animal {
2.      open fun walk() {  }
3.  }
4.
5.  open class LightweightAnimal : Animal() {
6.      open fun jump() {  }
7.  }
```

Esto permite tener animales que si pueden saltar y otros que no:

```
1.  class Dog: LightweightAnimal()
2.
3.  class Elephant: Animal()
```

Y podríamos limitar la función JumpHole a animales que si puedan saltar:

```
1.  fun jumpHole(a: LightweightAnimal){
2.      a.walk()
3.      a.jump()
4.      a.walk()
5.  }
```

Elegir las abstracciones correctas muchas veces no es fácil, pero tenemos que intentar limitar al máximo cuál es su alcance para no pedir más de lo que se necesita ni menos. Esta solución es al aplicar el principio de Liskov.

I – Interface Segregation Principle (ISP)

En español **“Principio de Segregación de la Interfaz”,** establece que los clientes de un módulo o componente no deben depender de interfaces que no utilizan. Esto es aplicable al diseño de componentes, asegurándose que los componentes tengan interfaces específicas y coherentes con su funcionalidad, evitando la dependencia innecesaria de interfaces más amplias.

Ejemplo:

Supongamos que en una tienda de discos de música sede tiene el siguiente modelado de productos:

```
1.  interface Product {
2.      val name: String
3.      val stock: Int
4.      val numberOfDisks: Int
5.      val releaseDate: Int
6.  }
7.
8.  class CD : Product {
9.      ...
10. }
```

El producto tiene una serie de propiedades que nuestra clase CD sobrescribirá de algún modo.

Pero ahora has decidido ampliar mercado, y empezar a vender DVDs también.

El problema es que para los DVDs necesitas almacenar también la clasificación por edades, porque tienes que asegurarte de que no vendes películas no adecuadas según la edad del cliente.

Lo más directo sería simplemente añadir la nueva propiedad a la interfaz:

```
1.  interface Product {
2.      ...
3.      val recommendedAge: Int
4.  }
```

Esto obliga a colocar esa opción los CD, pero al no tener ese dato, se aplica una excepción:

```
1. class CD : Product {
2.     ...
3.     override val recommendedAge: Int
4.         get() = throw UnsupportedOperationException()
5. }
```

Esto complica el código cada vez que se requiera agregar algo nuevo a Product, por lo que la solución que plantea este principio es la de segregar las interfaces, y asi cada clase utilizara la que necesite:

```
1.  interface AgeAware {
2.      val recommendedAge: Int
3.  }
```

Y asi, por ejemplo, para agregar un nuevo DVD, se utilizarán dos interfaces:

```
1. class CD : Product {
2.     ...
3. }
4.
5. class DVD : Product, AgeAware {
6.     ...
7. }
```

D – Dependency Inversion Principle (DIP)

En español **“Principio de inversión de dependencia”,** Este principio establece que los módulos de alto nivel no deben depender de módulos de bajo nivel, sino de abstracciones. Esto significa que los componentes de nivel superior deben depender de interfaces y abstracciones en lugar de depender directamente de componentes de nivel inferior. Esto permite una mayor flexibilidad y facilita la sustitución de componentes sin modificar el código que los utiliza.

Ejemplo:

Imaginemos que tenemos una cesta de la compra que lo que hace es almacenar la información y llamar al método de pago para que ejecute la operación. Nuestro código sería algo así:

```
1.  class Shopping { ... }
2.
3.  class ShoppingBasket {
4.      fun buy(shopping: Shopping?) {
5.          val db = SqlDatabase()
6.          db.save(shopping)
7.          val creditCard = CreditCard()
8.          creditCard.pay(shopping)
9.      }
10. }
11.
12. class SqlDatabase {
13.     fun save(shopping: Shopping?) {
14.         // Saves data in SQL database
15.     }
16. }
17.
18. class CreditCard {
19.     fun pay(shopping: Shopping?) {
20.         // Performs payment using a credit card
21.     }
22. }
```

Aquí estamos incumpliendo todas las reglas que impusimos al principio. Una clase de más alto nivel, como es la cesta de la compra, está dependiendo de otras de bajo nivel, como cuál es el mecanismo para almacenar la información o para realizar el método de pago. Se encarga de crear instancias de esos objetos y después utilizarlas.

Piensa ahora qué pasa si quieres añadir métodos de pago, o enviar la información a un servidor en vez de guardarla en una base de datos local. No hay forma de hacer todo esto sin desmontar toda la lógica. ¿Cómo lo solucionamos?

Primer paso, dejar de depender de concreciones. Vamos a crear interfaces que definan el comportamiento que debe dar una clase para poder funcionar como mecanismo de persistencia o como método de pago:

```
1.  interface Persistence {
2.      fun save(shopping: Shopping?)
3.  }
4.
5.  class SqlDatabase : Persistence {
6.      override fun save(shopping: Shopping?) {
7.          // Saves data in SQL database
8.      }
9.  }
10.
11. interface PaymentMethod {
12.     fun pay(shopping: Shopping?)
13. }
14.
15. class CreditCard : PaymentMethod {
16.     override fun pay(shopping: Shopping?) {
17.         // Performs payment using a credit card
18.     }
19. }
```

Nuestro segundo paso es invertir las dependencias. Vamos a hacer que estos objetos se pasen por constructor:

```
1. class ShoppingBasket(
2.     private val persistence: Persistence,
3.     private val paymentMethod: PaymentMethod
4. ) {
5.     fun buy(shopping: Shopping?) {
6.         persistence.save(shopping)
7.         paymentMethod.pay(shopping)
8.     }
9. }
```

Y asi mejoramos nuestro código aplicando los principios SOLID.

Ejemplo utilizando principios SOLID:

En este ejemplo, tenemos un componente reutilizable llamado Button que se encarga únicamente de renderizar un botón y manejar su acción de presionado. Este componente tiene una única responsabilidad, que es representar visualmente un botón y ejecutar una acción cuando se presiona.

Luego, en el componente HomeScreen, utilizamos el componente Button y definimos una función llamada handleButtonPress que contiene la lógica específica de manejo cuando el botón es presionado. El componente HomeScreen se enfoca en mostrar el texto de bienvenida y proporcionar el contexto para utilizar el componente Button. Esto mantiene una separación clara de responsabilidades y asegura que cada componente tenga una única razón para cambiar.

El uso del principio de responsabilidad única en este ejemplo permite que el componente Button sea reutilizable en diferentes partes de la aplicación, ya que su funcionalidad no está acoplada a un contexto específico. Además, la separación de la lógica de manejo del botón en el componente HomeScreen facilita la comprensión y el mantenimiento del código.

Conclusión

En resumen, los principios SOLID son un conjunto de pautas de diseño de software que promueven la creación de sistemas robustos, flexibles y mantenibles. Estos principios se aplican en React Native y en otros lenguajes de programación para ayudar a los desarrolladores a construir componentes y aplicaciones de manera eficiente.

Al seguir los principios SOLID, los desarrolladores pueden lograr una serie de beneficios:

- **Modularidad:** Los principios SOLID fomentan la creación de componentes independientes y cohesivos. Esto permite un mejor mantenimiento, ya que cada componente se encarga de una sola responsabilidad y es más fácil de entender y modificar.
- **Flexibilidad:** La aplicación de SOLID promueve el diseño de componentes extensibles y personalizables. Esto significa que se pueden agregar nuevas funcionalidades o modificar comportamientos existentes sin necesidad de alterar el código fuente original.
- **Reutilización de código**: Al seguir el principio de responsabilidad única, se crea código más reutilizable. Los componentes pueden ser fácilmente extraídos y utilizados en diferentes partes de una aplicación o incluso en proyectos futuros, lo que ahorra tiempo y esfuerzo en el desarrollo.
- **Mantenibilidad:** Los principios SOLID facilitan el mantenimiento del código a lo largo del tiempo. Debido a la claridad en la responsabilidad de cada componente y la reducción de acoplamiento entre ellos, es más sencillo corregir errores, realizar mejoras y realizar pruebas unitarias.
- **Escalabilidad:** Al diseñar aplicaciones basadas en los principios SOLID, se establece una base sólida para el crecimiento y la escalabilidad. Los componentes bien diseñados y desacoplados permiten agregar nuevas funcionalidades y adaptarse a los cambios de requerimientos sin afectar el resto del sistema.

En general, los principios SOLID son herramientas valiosas para los desarrolladores de React Native y otros entornos de desarrollo. Al aplicarlos, se fomenta un diseño de software de alta calidad que facilita la creación de aplicaciones robustas, mantenibles y escalables.

Referencias

- https://devexperto.com/principio-responsabilidad-unica/
- https://profile.es/blog/principios-solid-desarrollo-software-calidad/