Principios SOLID en Acción: Diseñando con Mario Bros

Guía Completa para Desarrolladores iOS

Arturo González

INTRODUCCIÓN

¡Hola! Mi nombre es Arturo y soy desarrollador iOS con algunos años de experiencia. Siempre me ha apasionado compartir mis conocimientos, ya que creo que enseñar no solo ayuda a otros, sino que también refuerza nuestro propio entendimiento de los conceptos.

Por eso, he creado esta guía para desarrolladores. Aunque los ejemplos están en Swift, los principios se pueden aplicar a otros lenguajes de programación.

A continuación, encontrarás ejemplos y explicaciones conectados con el popular juego Mario Bros. Creo que de esta manera puedo ilustrar mejor cada uno de los conceptos.

INTRODUCCIÓN A LOS PRINCIPIOS SOLID ⛩

Los principios SOLID son cinco directrices esenciales para el diseño de software que facilitan la creación de código comprensible, flexible y fácil de mantener. Fueron popularizados por Robert C. Martin, también conocido como 'Uncle Bob'. Estos principios, clave en la programación orientada a objetos, están diseñados para resolver problemas frecuentes en el diseño de software, tales como el alto acoplamiento y la baja cohesión.

El término SOLID es un acrónimo donde cada letra representa uno de los principios:

- S: Principio de Responsabilidad Única (Single Responsibility Principle).
- O: Principio de Abierto/Cerrado (Open/Closed Principle).
- L: Principio de Sustitución de Liskov (Liskov Substitution Principle).
- I: Principio de Segregación de Interfaces (Interface Segregation Principle).
- D: Principio de Inversión de Dependencias (Dependency Inversion Principle).

Alto acoplamiento: Se refiere a la situación en la que dos o más módulos, clases o componentes de un sistema están estrechamente vinculados, de tal manera que los cambios en uno podrían requerir cambios en los otros.

S - PRINCIPIO DE RESPONSABILIDAD ÚNICA (SINGLE RESPONSIBILITY PRINCIPLE)

Principio de Responsabilidad Única: Este principio establece que una clase debe tener una sola responsabilidad o razón para cambiar. Cada clase debe encargarse de una única tarea o funcionalidad, lo que facilita el mantenimiento y la evolución del código. Así, los cambios en una funcionalidad no impactarán otras áreas.

Para ilustrar este y otros principios, usemos el ejemplo de un videojuego de Mario Bros. En este juego, Mario puede realizar acciones como saltar, correr y lanzar bolas de fuego.

Antes de aplicar estos principios, la práctica común (basada en mi propia experiencia) sería crear una clase que englobe todas las acciones de Mario:

```
class MarioActions {
    func jump() {
        // Lógica para el salto de Mario
        print("Mario salta")
    }

    func run() {
        // Lógica para la carrera de Mario
        print("Mario corre")
    }

    func launchingFireballs() {
        // Lógica para los disparos de Mario
        print("Mario dispara")
    }
}
```

Enfoque Tradicional

Desventajas del Enfoque Tradicional: ¿Cuál es el inconveniente de este enfoque? El problema principal es que cada vez que necesitamos modificar la lógica de una acción, corremos el riesgo de introducir errores en otras partes del código. Estos errores pueden deberse a un error tipográfico o a una falla en la lógica que afecte el estado interno de la clase.

Así, un cambio o error en una acción específica, como en la función launchingFireballs, podría afectar negativamente otras partes de MarioActions, haciendo que métodos como jump o run se comporten incorrectamente, incluso si no están directamente relacionados con launchingFireballs.

Solución con el Principio de Responsabilidad Única: Entonces, ¿cómo resolvemos esto? Aplicando el SRP (Principio de Responsabilidad Única), donde cada acción es gestionada por una clase diferente.

```
// Clase para manejar el salto de Mario
class MarioJump {
    func jump() {
        // Lógica para el salto de Mario
        print("Mario salta")
    }
}

// Clase para manejar la carrera de Mario
class MarioRun {
    func run() {
        // Lógica para la carrera de Mario
        print("Mario corre")
    }
}

// Clase para manejar los disparos de Mario
class MarioShoot {
    func launchingFireballs() {
        // Lógica para los disparos de Mario
        print("Mario dispara")
    }
}
```

Al aplicar el SRP, hemos optado por segmentar MarioActions en varias clases, cada una dedicada exclusivamente a una acción particular—MarioJump para saltar, MarioRun para correr, y MarioShoot para disparar. Así, si es necesario ajustar la lógica del salto, simplemente modificamos la clase MarioJump, sin el riesgo de afectar otras acciones.

O - PRINCIPIO DE ABIERTO/CERRADO (OPEN/CLOSED PRINCIPLE).

Principio de Abierto/Cerrado: Este principio dicta que una clase debe estar abierta para la extensión, pero cerrada para la modificación. Es decir, deberíamos ser capaces de añadir nuevas funcionalidades a una clase sin necesidad de alterar su código existente. Este enfoque no solo fomenta la reutilización del código, sino que también minimiza el riesgo de errores al modificar clases que ya han sido probadas y están en uso.

Supongamos que queremos añadir nuevas habilidades a Mario, como la capacidad de volar y bucear. Sin adherirnos al Principio de Abierto/Cerrado, podríamos vernos tentados a integrar estas nuevas habilidades directamente en la clase original MarioActions:

```
class MarioActions {
    func jump() {
        print("Mario salta")
    }

    func run() {
        print("Mario corre")
    }

    func launchingFireballs() {
        print("Mario dispara")
    }

    // Nuevas funcionalidades
    func fly() {
        print("Mario vuela")
    }

    func swim() {
        print("Mario nada")
    }
}
```

Enfoque Tradicional

Limitaciones del Enfoque Tradicional: A primera vista, modificar directamente la clase MarioActions para añadir nuevas habilidades puede parecer práctico. Sin embargo, cada adición requiere cambios en la clase, lo que aumenta el riesgo de errores y complica el mantenimiento del código.

Implementando el Principio de Abierto/Cerrado: ¿Cómo podemos manejar mejor las expansiones de funcionalidad sin alterar el código existente?

La solución es evitar modificar la clase original. En su lugar, optamos por crear nuevas clases que extiendan las capacidades de Mario mediante herencia o, preferiblemente, mediante composición. Veamos cómo se puede lograr esto último:

```
// Clase base de Mario con las habilidades básicas
class Mario {
    func jump() {
        print("Mario salta")
    }

    func run() {
        print("Mario corre")
    }
}
```

```
// Protocolo para definir una habilidad de Mario
protocol MarioAbility {
    func useAbility()
}
```

```
// Clase para la habilidad de disparar
class MarioShoot: MarioAbility {
    func useAbility() {
        print("Mario dispara")
    }
}

// Clase para la habilidad de volar
class MarioFly: MarioAbility {
    func useAbility() {
        print("Mario vuela")
    }
}

// Clase para la habilidad de nadar
class MarioSwim: MarioAbility {
    func useAbility() {
        print("Mario nada")
    }
}
```

Extendiendo Funcionalidades con Protocolos: Para este ejemplo, creamos un protocolo MarioAbility que define las posibles habilidades adicionales para Mario. Seguidamente, implementamos clases específicas para cada habilidad, todas siguiendo este protocolo. Así, extendemos las funcionalidades de Mario sin modificar la clase original, lo que mantiene el código más organizado y sencillo de mantener.

Uso de Protocolos en Swift: En Swift, los protocolos se utilizan para definir interfaces que pueden ser adoptadas por clases, estructuras o enumeraciones, facilitando la extensibilidad y reusabilidad del código. En otros lenguajes, como Java y C#, este rol es desempeñado por las interfaces, que definen contratos para las clases. Similarmente, TypeScript utiliza interfaces para estructurar objetos.

Beneficios del Principio de Abierto/Cerrado: Aplicando este principio, alcanzamos un diseño de software más flexible y fácil de mantener. Permite la adición de nuevas funcionalidades a nuestro código sin alterar las clases existentes, optimizando así la escalabilidad y la gestión del mismo."

L - PRINCIPIO DE SUSTITUCIÓN DE LISKOV (LSP)

Principio de Sustitución de Liskov (LSP): Este principio sostiene que las clases derivadas deben poder sustituir a sus clases base sin afectar la funcionalidad del programa. Es decir, si tenemos una subclase B de la clase A, debe ser posible usar objetos de B en lugar de A sin alterar el comportamiento del software.

Imaginemos que en nuestro juego contamos con distintos personajes, como Mario y Luigi, que comparten acciones básicas tales como correr y saltar. Bajo el LSP, cualquier subclase de Character debe poder usarse donde se espera un objeto de este tipo sin comprometer la funcionalidad del juego.

Para implementar esto, empezamos definiendo una clase base Character y, a continuación, desarrollamos subclases específicas para Mario y Luigi:

```
class Character {
    func run() {
        print("El personaje corre")
    }

    func jump() {
        print("El personaje salta")
    }
}
```

```
class Mario: Character {
    override func run() {
        print("Mario corre")
    }

    override func jump() {
        print("Mario salta")
    }
}

class Luigi: Character {
    override func run() {
        print("Luigi corre")
    }

    override func jump() {
        print("Luigi salta")
    }
}
```

Aplicación del LSP en Funciones: Podemos diseñar funciones que acepten objetos del tipo Character. De acuerdo con el Principio de Sustitución de Liskov (LSP), estas funciones deberían operar correctamente con cualquier subclase de Character, asegurando la interoperabilidad y la flexibilidad del código:

```
func performActions(character: Character) {
    character.run()
    character.jump()
}

// Usamos Mario y Luigi con la función que espera un Character
let mario = Mario()
let luigi = Luigi()

performActions(character: mario)
performActions(character: luigi)
```

En este caso, la función performActions está diseñada para aceptar un parámetro del tipo Character. Esto permite que la función reciba objetos tanto de la clase base Character como de cualquiera de sus subclases, como Mario o Luigi. Al invocar performActions con un objeto de tipo Mario o Luigi, el programa mantiene su funcionalidad intacta y ejecuta correctamente los métodos específicos de cada subclase

---

Ejemplo de Violación del Principio de Sustitución de Liskov (LSP): Para ilustrar una violación del LSP, examinemos el caso de una subclase nueva, FlyingMario, cuyo comportamiento difiere significativamente del de su clase base.

```
class FlyingMario: Character {
    override func run() {
        print("Flying Mario no corre, vuela")
    }

    override func jump() {
        // No hace nada porque Flying Mario no salta, solo vuela
    }
}
```

Si intentamos usar FlyingMario con la función performActions, el comportamiento no sería el esperado:

```
let flyingMario = FlyingMario()
performActions(character: flyingMario)
```

En este caso, FlyingMario no se comporta como un Character estándar porque ha alterado la lógica de los métodos run y jump, violando el LSP. Los usuarios de la clase Character esperan que todos los personajes corran y salten de manera consistente, pero FlyingMario no cumple con estas expectativas, lo que podría provocar problemas en el programa.

I - PRINCIPIO DE SEGREGACIÓN DE INTERFACES (ISP)

Principio de Segregación de Interfaces (ISP): Este principio establece que los clientes no deben ser forzados a depender de interfaces que no utilizan. Es más efectivo y eficiente disponer de múltiples interfaces pequeñas y específicas, adecuadas para cada tipo de cliente, en lugar de una única interfaz grande y monolítica. Esto contribuye a eliminar el 'peso muerto' en las clases, manteniéndolas más organizadas y enfocadas en sus responsabilidades específicas.

Consideremos el ejemplo de nuestro videojuego, donde los personajes poseen habilidades diversas: algunos pueden saltar, otros volar, y otros nadar. Diseñar una interfaz única que abarque todas estas habilidades resultaría en una estructura sobredimensionada y obligaría a muchos personajes a implementar métodos irrelevantes para sus funciones.

Ejemplo de una Interfaz Grande y Poco Específica:
Vamos a explorar cómo se vería esta interfaz demasiado amplia:

```
protocol Character {
    func run()
    func jump()
    func fly()
    func swim()
}
```

Implementar esta interfaz en diferentes personajes obligaría a cada uno a proporcionar implementaciones para métodos que no utilizan, lo cual no es ideal:

```
class Mario: Character {
    func run() {
        print("Mario corre")
    }

    func jump() {
        print("Mario salta")
    }

    func fly() {
        // Mario no vuela, pero tiene que implementar el método
    }

    func swim() {
        print("Mario nada")
    }
}

class Luigi: Character {
    func run() {
        print("Luigi corre")
    }

    func jump() {
        print("Luigi salta")
    }

    func fly() {
        // Luigi no vuela, pero tiene que implementar el método
    }

    func swim() {
        print("Luigi nada")
    }
}
```

La implementación de métodos que no se utilizan puede causar confusión y errores.

En lugar de esto, aplicamos el ISP dividiendo la interfaz grande en varias interfaces más pequeñas y específicas:

```
protocol Runnable {
    func run()
}

protocol Jumpable {
    func jump()
}

protocol Flyable {
    func fly()
}

protocol Swimmable {
    func swim()
}
```

Ahora podemos implementar estas interfaces específicas solo en los personajes que realmente necesitan esas habilidades:

```
class Mario: Runnable, Jumpable, Swimmable {
    func run() {
        print("Mario corre")
    }

    func jump() {
        print("Mario salta")
    }

    func swim() {
        print("Mario nada")
    }
}

class Luigi: Runnable, Jumpable, Swimmable {
    func run() {
        print("Luigi corre")
    }

    func jump() {
        print("Luigi salta")
    }

    func swim() {
        print("Luigi nada")
    }
}
```

Podemos crear funciones que acepten objetos que implementen estas interfaces específicas, haciendo que el código sea más flexible y modular:

```
func performRunAction(runner: Runnable) {
    runner.run()
}

func performJumpAction(jumper: Jumpable) {
    jumper.jump()
}

func performFlyAction(flyer: Flyable) {
    flyer.fly()
}

let mario = Mario()
let luigi = Luigi()
let flyingMario = FlyingMario()

performRunAction(runner: mario)
performJumpAction(jumper: luigi)
performFlyAction(flyer: flyingMario)
```

El Principio de Segregación de Interfaces nos ayuda a mantener nuestras clases y sus dependencias más claras y específicas. Al evitar interfaces monolíticas y dividirlas en varias interfaces más pequeñas y enfocadas, reducimos la complejidad y aumentamos la mantenibilidad de nuestro código. Esto permite que las clases implementen solo lo que realmente necesitan, promoviendo un diseño más limpio y eficiente.

D - PRINCIPIO DE INVERSIÓN DE DEPENDENCIA (DIP)

Principio de Inversión de Dependencias (DIP): Este principio dicta que los módulos de alto nivel no deben depender de módulos de bajo nivel. En su lugar, ambos tipos de módulos deben basarse en abstracciones. Además, estas abstracciones no deben depender de los detalles implementativos; más bien, los detalles deben depender de las abstracciones. Este enfoque sugiere que el diseño del software debe ser guiado por interfaces o abstracciones, y no por implementaciones concretas, facilitando así la creación de un código más flexible y fácil de mantener.

Para ilustrar esto, consideremos un sistema diseñado para que Mario interactúe con distintos tipos de enemigos, como Goombas y Koopas. Sin el Principio de Inversión de Dependencias, podríamos terminar diseñando un sistema en el que la clase Mario dependa directamente de las clases Goomba y Koopa, resultando en un acoplamiento demasiado rígido:

```
class Goomba {
    func defeat() {
        print("Goomba derrotado")
    }
}

class Koopa {
    func defeat() {
        print("Koopa derrotado")
    }
}

class Mario {
    func defeatGoomba(goomba: Goomba) {
        goomba.defeat()
    }

    func defeatKoopa(koopa: Koopa) {
        koopa.defeat()
    }
}
```

En este diseño, Mario depende directamente de las clases Goomba y Koopa. Si en el futuro queremos agregar nuevos enemigos, tendríamos que modificar la clase Mario, lo que va en contra del Principio de Abierto/Cerrado (OCP).
Para aplicar el DIP, introducimos una abstracción, como un protocolo Enemy, que define una interfaz común para todos los enemigos. Luego, hacemos que Goomba y Koopa implementen esta interfaz y cambiamos la clase Mario para que dependa de la abstracción en lugar de las implementaciones concretas:

```
protocol Enemy {
    func defeat()
}

class Goomba: Enemy {
    func defeat() {
        print("Goomba derrotado")
    }
}

class Koopa: Enemy {
    func defeat() {
        print("Koopa derrotado")
    }
}

class Mario {
    func defeatEnemy(enemy: Enemy) {
        enemy.defeat()
    }
}
```

Para aplicar el DIP, introducimos una abstracción, como un protocolo Enemy, que define una interfaz común para todos los enemigos.

Luego, hacemos que Goomba y Koopa implementen esta interfaz y cambiamos la clase Mario para que dependa de la abstracción en lugar de las implementaciones concretas:

```
protocol Enemy {
    func defeat()
}

class Goomba: Enemy {
    func defeat() {
        print("Goomba derrotado")
    }
}

class Koopa: Enemy {
    func defeat() {
        print("Koopa derrotado")
    }
}

class Mario {
    func defeatEnemy(enemy: Enemy) {
        enemy.defeat()
    }
}
```

En este diseño, Mario ya no depende de las clases concretas Goomba y Koopa, sino de la abstracción Enemy. Esto hace que el sistema sea más flexible y fácil de extender. Ahora, podemos agregar nuevos tipos de enemigos sin modificar la clase Mario:

```
class PiranhaPlant: Enemy {
    func defeat() {
        print("Piranha Plant derrotada")
    }
}

// Ejemplo de uso
let mario = Mario()
let goomba = Goomba()
let koopa = Koopa()
let piranhaPlant = PiranhaPlant()

mario.defeatEnemy(enemy: goomba)
mario.defeatEnemy(enemy: koopa)
mario.defeatEnemy(enemy: piranhaPlant)
```

El Principio de Inversión de Dependencia nos ayuda a diseñar sistemas más flexibles y fáciles de mantener al hacer que tanto los módulos de alto nivel como los de bajo nivel dependan de abstracciones en lugar de implementaciones concretas.

CONCLUSIÓN

A lo largo de este ensayo, hemos explorado cómo los principios SOLID actúan como pilares fundamentales en el diseño de software. Mediante ejemplos prácticos, basados en el icónico videojuego de Mario Bros, hemos demostrado que aplicar estos principios no solo optimiza la calidad del código, sino que también potencia su flexibilidad y escalabilidad.


Arturo7thDev


arturo@7th.dev