¡Te doy la bienvenida!

Muchas gracias por tu interés en esta guía. Para mí es un honor poder ayudarte a avanzar en tus conocimientos para que puedas crear código de calidad.

Tómate tu tiempo para sacarle el máximo partido. Si nunca habías visto estos conceptos antes, te puede llevar un tiempo.

¿Quién soy yo?

Mi nombre es Antonio Leiva, y llevo desde el 2012 ayudando a otros desarrolladores a evolucionar en su carrera profesional. Soy formador, Google Developer Expert en Android y Kotlin, y Kotlin Certified Trainer por JetBrains.

Escribí hace un tiempo un libro llamado **Kotlin for Android Developers**, en el que explico cómo pasar de Java a Kotlin. También existe el curso online y la formación para empresas.

Además **soy el creador del programa Architect Coders**, donde ayudo a desarrolladores Android a convertirse en verdaderos ingenieros, dominando todos los conocimientos que las grandes empresas y startups del sector demandan. Puedes apuntarte aquí para que te informe de la próxima edición.

Si quieres conectar conmigo en las redes sociales, puedes encontrarme en:

- YouTube
- Twitter
- Instagram
- LinkedIn

Y si has llegado a esta página por otra vía, puedes suscribirte aquí y llevarte de regalo la guía para iniciarte en Kotlin.

¡Empezamos!

Patrones de Diseño de Software

Los Patrones de Diseño son uno de los conceptos más populares dentro del diseño de software.

En esta guía te voy a mostrar qué son los patrones, cuándo utilizarlos y cuáles son los patrones más populares.

¿Qué son los patrones de diseño?

Hay una cosa que está clara: por muy específico que sea un problema al que te estés enfrentando durante el desarrollo de tu software, hay un 99% de posibilidades (cifra totalmente inventada, pero seguro que muy real) de que **alguien se haya enfrentado a un problema tan similar en el pasado, que se pueda modelar de la misma manera**.

Con modelado me estoy refiriendo a que la estructura de las clases que conforma la solución de tu problema puede estar ya inventada, porque estás resolviendo un problema común que otra gente ya ha solucionado antes. Si **la forma de solucionar ese problema se puede extraer, explicar y reutilizar** en múltiples ámbitos, entonces nos encontramos ante un patrón de diseño de software.

Un patrón de diseño es una forma reutilizable de resolver un problema común. Clic para tuitear

El concepto de patrón de diseño lleva existiendo desde finales de los 70, pero su verdadera popularización surgió en los 90 con el lanzamiento del libro de Design Pattern de la Banda de los

Cuatro (Gang of Four), que aunque parezca que estamos hablando de los Trotamúsicos, es el nombre con el que se conoce a los creadores de este libro: Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. En él explican 23 patrones de diseño, que desde entonces han sido un referente.

¿Por qué son útiles los patrones de diseño?

Puede parecer una tontería, pero si no encuentras utilidad a las cosas acabarás por no usarlas. Los patrones de diseño son muy útiles por los siguientes motivos:

1. Te ahorran tiempo

Sé que te encantará encontrar una solución ingeniosa a un problema cuando estás modelando tu software, y es normal, a mí también me pasa. Como he comentado alguna vez, el desarrollo es un proceso casi artístico, y ese reto mental que supone revierte en una satisfacción personal enorme una vez que consigues un buen resultado.

Pero hay que ser sinceros: **buscar siempre una nueva solución a los mismos problemas reduce tu eficacia como desarrollador**, porque estás perdiendo mucho tiempo en el proceso. No hay que olvidar que el desarrollo de software también es una ingeniería, y que por tanto en muchas ocasiones habrá reglas comunes para solucionar problemas comunes.

Buscar siempre una nueva solución a los mismos problemas reduce tu eficacia como desarrollador. Clic para tuitear

Los patrones de diseño atajan ese punto. Una vez los conozcas, contarás con un conjunto de «trucos», de reglas, de herramientas muy probadas, que te permitirán s**olucionar la mayor parte de tus problemas de forma directa**, sin tener que pensar en cómo de válidas son, o si puede haber una alternativa mejor.

2. Te ayudan a estar seguro de la validez de tu código

Un poco relacionado con lo anterior, siempre que creamos algo nuevo nos surge la duda de si realmente estamos dando con la solución correcta, o si realmente habrá una respuesta mejor. Y el tema es que es una duda muy razonable y que en muchos casos la respuesta sea la que no deseas: sí que hay una solución más válida, y has perdido tu valioso tiempo en implementar algo que, aunque funciona, podría haberse modelado mejor.

Los patrones de diseño son estructuras probadas por millones de desarrolladores a lo largo de muchos años, por lo que si eliges el patrón adecuado para modelar el problema adecuado, puedes estar seguro de que va a ser una de las soluciones más válidas (si no la que más) que puedas encontrar.

3. Establecen un lenguaje común

Todas las demás razones palidecen ante esta. Modelar tu código mediante patrones **te ayudará a explicar a otras personas, conozcan tu código o no, a entender cómo has atajado un problema**. Además ayudan a otros desarrolladores a comprender lo que has implementado, cómo y por qué, y además a descubrir rápidamente si esa era la mejor solución o no.

Los patrones de diseño establecen un lenguaje común entre todos los miembros de un equipo. Clic para tuitear

Pero también te servirá para sentarte con tus compañeros a pensar sobre cómo solucionar algo, y poneros de acuerdo mucho más rápido, explicar de forma más sencilla cuáles son vuestras ideas y que el resto lo comprenda sin ningún problema. Los patrones de diseño os ayudarán a ti y a tu equipo, en definitiva, a **avanzar mucho más rápido, con un código más fácil de entender para todos** y mucho más robusto.

¿Cómo identificar qué patrón encaja con tu problema?

Desafortunadamente, tengo malas noticias… Este es el punto más complicado, y la respuesta más evidente, que es también la que menos nos gusta, es que **se aprende practicando**. La experiencia es la única forma válida de ser más hábil detectando dónde te pueden ayudar los patrones de diseño.

Por supuesto, hay situaciones conocidas en las que un patrón u otro nos puede ayudar, y las iré comentando a lo largo de los artículos. Además te recomiendo que te leas el libro de Head First Design Patterns, en el que además de explicarte los patrones de forma muy amena, explican muy bien cómo usarlos en la vida real.

Pero a partir de ese punto estás solo. Necesitarás conocer **qué tipo de problemas soluciona cada uno y descubrir cómo aplicarlo a casos concretos**. Como comentaba en el artículo de

los miedos, en este caso lo mejor que te puede pasar es que encuentres a un compañero que los domine y que te haga de mentor. Pégate a él y exprímelo hasta que tengas todo su conocimiento. En caso contrario, practica, practica y practica.

Listado de patrones de diseño

Este es el listado de los patrones de diseño más conocidos.

Los patrones se dividen en **distintos grupos según el tipo de problema que resuelven**, a saber:

Patrones creacionales

Son los que **facilitan la tarea de creación de nuevos objetos**, de tal forma que el proceso de creación pueda ser desacoplado de la implementación del resto del sistema.

Los patrones creacionales están basados en dos conceptos:

1. Encapsular el conocimiento acerca de los tipos concretos que nuestro sistema utiliza. Estos patrones normalmente trabajarán con interfaces, por lo que la implementación concreta que utilicemos queda aislada.
2. Ocultar cómo estas implementaciones concretas necesitan ser creadas y cómo se combinan entre sí.

Los patrones creacionales facilitan la tarea de creación de nuevos objetos encapsulando el proceso. Clic para tuitear

Los patrones creacionales más conocidos son:

- **Abstract Factory**: Nos provee una interfaz que delega la creación de un conjunto de objetos relacionados sin necesidad de especificar en ningún momento cuáles son las implementaciones concretas.
- **Factory Method:** Expone un método de creación, delegando en las subclases la implementación de este método.
- **Builder**: Separa la creación de un objeto complejo de su estructura, de tal forma que el mismo proceso de construcción nos puede servir para crear representaciones diferentes.
- **Singleton:** limita a uno el número de instancias posibles de una clase en nuestro programa, y proporciona un acceso global al mismo.
- **Prototype:** Permite la creación de objetos basados en «plantillas». Un nuevo objeto se crea a partir de la clonación de otro objeto.

Patrones estructurales

Son patrones que nos facilitan la modelización de nuestros software **especificando la forma en la que unas clases se relacionan con otras**.

Los patrones estructurales especifican la forma en que unas clases se relacionan con otras. Clic para tuitear

Estos son los patrones estructurales que definió la Gang of Four:

- **Adapter**: Permite a dos clases con diferentes interfaces trabajar entre ellas, a través de un objeto intermedio con el que se comunican e interactúan.
- **Bridge**: Desacopla una abstracción de su implementación, para que las dos puedan evolucionar de forma independiente.
- **Composite**: Facilita la creación de estructuras de objetos en árbol, donde todos los elementos emplean una misma interfaz. Cada uno de ellos puede a su vez contener un listado de esos objetos, o ser el último de esa rama.
- **Decorator**: Permite añadir funcionalidad extra a un objeto (de forma dinámica o estática) sin modificar el comportamiento del resto de objetos del mismo tipo.
- **Facade**: Una facade (o fachada) es un objeto que crea una interfaz simplificada para tratar con otra parte del código más compleja, de tal forma que simplifica y aísla su uso. Un ejemplo podría ser crear una fachada para tratar con una clase de una librería externa.
- **Flyweight**: Una gran cantidad de objetos comparte un mismo objeto con propiedades comunes con el fin de ahorrar memoria.
- **Proxy**: Es una clase que funciona como interfaz hacia cualquier otra cosa: una conexión a Internet, un archivo en disco o cualquier otro recurso que sea costoso o imposible de duplicar.

Patrones de comportamiento

En este último grupo se encuentran la mayoría de los patrones, y se usan para **gestionar algoritmos, relaciones y responsabilidades entre objetos**.

Los patrones de comportamiento gestionan algoritmos, relaciones y responsabilidades entre objetos. Clic para tuitear

Los patrones de comportamiento son:

- **Command**: Son objetos que encapsulan una acción y los parámetros que necesitan para ejecutarse.
- **Chain of responsibility**: se evita acoplar al emisor y receptor de una petición dando la posibilidad a varios receptores de consumirlo. Cada receptor tiene la opción de consumir esa petición o pasárselo al siguiente dentro de la cadena.
- **Interpreter**: Define una representación para una gramática así como el mecanismo para evaluarla. El árbol de sintaxis del lenguaje se suele modelar mediante el patrón Composite.
- **Iterator**: Se utiliza para poder movernos por los elementos de un conjunto de forma secuencial sin necesidad de exponer su implementación específica.
- **Mediator**: Objeto que encapsula cómo otro conjunto de objetos interactúan y se comunican entre sí.
- **Memento**: Este patrón otorga la capacidad de restaurar un objeto a un estado anterior.

- Observer: Los objetos son capaces de suscribirse a una serie de eventos que otro objetivo va a emitir, y serán avisados cuando esto ocurra.
- State: Permite modificar la forma en que un objeto se comporta en tiempo de ejecución, basándose en su estado interno.
- Strategy: Permite la selección del algoritmo que ejecuta cierta acción en tiempo de ejecución.
- Template Method: Especifica el esqueleto de un algoritmo, permitiendo a las subclases definir cómo implementan el comportamiento real.
- Visitor: Permite separar el algoritmo de la estructura de datos que se utilizará para ejecutarlo. De esta forma se pueden añadir nuevas operaciones a estas estructuras sin necesidad de modificarlas.

Conclusión

Como ves, nos encontramos ante una lista muy variada de patrones que nos dan la oportunidad de **crear nuestro código de manera mucho más sencilla con estructuras probadas y que funcionan**. La mayor complejidad radica en saber cuándo utilizarlas, algo que nos dará la práctica.

Es lógico que sólo con la pequeña definición que he dado no termines de entender para qué sirven algunos, pero te sirven como referencia para ver cuál te puede encajar más y buscar más información sobre ese en particular.

Existen muchos otros patrones de diseño además de los que se definieron en el libro de *Design Patterns* de la *Gang of Four*, pero con estos ya tienes una buena base sobre la que empezar a trabajar.

Y a ti, ¿cuáles te parecen los patrones de diseño más útiles? ¿Ya los conocías todos?

Los 23 Patrones

Singleton

El patrón de diseño Singleton es uno de los patrones más utilizados en la programación orientada a objetos. Su objetivo es garantizar que una clase solo tenga una única instancia y proporcionar un punto de acceso global a ella.

Este patrón es útil en casos en los que necesitamos garantizar que solo exista una única instancia de una clase, como por ejemplo en el caso de un administrador de recursos compartidos o un administrador de configuración.

En Kotlin, podemos implementar el patrón Singleton de la siguiente manera:

```
class Singleton private constructor() {

    companion object {

        private var instance: Singleton? = null

        fun getInstance(): Singleton {

            if (instance == null) {

                instance = Singleton()

            }
```

```
            return instance!!
        }
    }
}
```

En este ejemplo, la clase Singleton tiene un constructor privado para evitar que se pueda instanciar directamente.

En su lugar, se utiliza el método `getInstance` de la companion object para obtener la única instancia de la clase.

Además, se utiliza una variable de instancia privada para almacenar la única instancia de la clase.
Para usar la clase Singleton, simplemente llamamos al método `getInstance`:

```
val singleton = Singleton.getInstance()
```

También es posible utilizar una expresión de objeto para implementar el patrón Singleton en Kotlin:

```
object Singleton {

    fun doSomething() {

        // Código aquí

    }
```

```
}
```

En este caso, la clase Singleton es implementada como una expresión de objeto, lo que garantiza que solo exista una única instancia de la misma.

En resumen, el patrón de diseño Singleton es una forma útil de garantizar que solo exista una única instancia de una clase en Kotlin. Su implementación es sencilla y permite un fácil acceso a la única instancia de la clase.

Abstract Factory

El patrón de diseño Abstract Factory es un patrón de diseño de software que se utiliza para proporcionar una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

Esto se logra mediante la creación de una fábrica abstracta que proporciona una interfaz para crear cada tipo de objeto de la familia, pero deja la implementación concreta de cada objeto a sus subclases.

Por ejemplo, imagina que estás desarrollando un juego de estrategia en el que los jugadores pueden construir diferentes tipos de edificios, como castillos, torres y murallas.

En lugar de tener una clase concreta para cada tipo de edificio, podría usar el patrón Abstract Factory para crear una fábrica abstracta de edificios que proporcione una interfaz para crear cada tipo de edificio.

Luego, podrías crear subclases concretas de la fábrica abstracta para cada tipo de edificio, como una fábrica de castillos, una fábrica de torres y una fábrica de murallas.

A continuación se muestra un ejemplo de cómo se podría implementar el patrón Abstract Factory en Kotlin:

La interfaz AbstractFactory proporciona una interfaz para crear cada tipo de objeto de la familia

```
interface AbstractFactory {

    fun createCastle(): Castle

    fun createWall(): Wall

    fun createTower(): Tower

}
```

Las subclases concretas de AbstractFactory implementan la interfaz para crear objetos concretos:

```
class CastleFactory: AbstractFactory {

    override fun createCastle(): Castle {

        return Castle()

    }
```

```
    override fun createWall(): Wall {
        return CastleWall()
    }
    override fun createTower(): Tower {
        return CastleTower()
    }
}

class WallFactory: AbstractFactory {
    override fun createCastle(): Castle {
        return Wall()
    }
    override fun createWall(): Wall {
        return Wall()
    }
    override fun createTower(): Tower {
        return WallTower()
    }
}

class TowerFactory: AbstractFactory {
```

```
    override fun createCastle(): Castle {
        return Tower()
    }
    override fun createWall(): Wall {
        return TowerWall()
    }
    override fun createTower(): Tower {
        return Tower()
    }
}
```

Las clases concretas representan cada tipo de objeto de la familia:

```
class Castle {
    // ...
}

class CastleWall: Wall {
    // ...
}

class CastleTower: Tower {
```

```
    // ...
}

class Wall {

    // ...
}

class WallTower: Tower {

    // ...
}

class Tower {

    // ...
}
```

En este ejemplo, la interfaz `AbstractFactory` proporciona una interfaz para crear cada tipo de objeto de la familia de edificios, como castillos, torres y murallas.

Las subclases concretas de `AbstractFactory`, como `CastleFactory`, `WallFactory` y `TowerFactory`, implementan esta interfaz para crear objetos concretos de cada tipo. Por último, las clases concretas como `Castle`, `CastleWall`, `CastleTower`, `Wall`, `WallTower` y `Tower` representan cada tipo de objeto de la familia de edificios.

El patrón Abstract Factory se utiliza para proporcionar una abstracción en la creación de objetos, lo que permite modificar la implementación concreta de los objetos sin afectar al código que los utiliza.

Además, permite crear familias de objetos relacionados que se pueden utilizar juntos sin especificar sus clases concretas, lo que facilita la extensión y el mantenimiento del código.

Factory Method

El patrón de diseño Factory Method es una de las técnicas de diseño de software más utilizadas en la actualidad.

Consiste en crear una interfaz o una clase abstracta que define los métodos necesarios para crear un objeto, pero deja la implementación de esos métodos en las subclases.

Esto permite a una clase crear objetos sin conocer la clase concreta del objeto que está creando.

Esto es muy útil cuando se trabaja con diferentes tipos de objetos que tienen una relación de jerarquía y se desea utilizar una clase para manejarlos de forma genérica.

Por ejemplo, supongamos que tenemos una aplicación que maneja diferentes tipos de vehículos, como coches, motocicletas y bicicletas.

En lugar de crear una clase diferente para cada tipo de vehículo, podemos utilizar el patrón Factory Method para crear una interfaz «Vehicle» que define los métodos necesarios para manejar un vehículo.

Luego, cada tipo de vehículo puede implementar esa interfaz y crear sus propios objetos.

En Kotlin, podemos implementar el patrón Factory Method de la siguiente manera.
Interfaz que define los métodos necesarios para manejar un vehículo:

```
interface Vehicle {

    fun startEngine()

    fun stopEngine()

    fun drive()

}
```

Clase concreta para manejar un coche:

```
class Car: Vehicle {

    override fun startEngine() {

        println("Starting car engine")
```

```
    }

    override fun stopEngine() {
        println("Stopping car engine")
    }

    override fun drive() {
        println("Driving car")
    }
}
```

Clase concreta para manejar una motocicleta:

```
class Motorcycle: Vehicle {

    override fun startEngine() {
        println("Starting motorcycle engine")
    }

    override fun stopEngine() {
        println("Stopping motorcycle engine")
    }

    override fun drive() {
        println("Driving motorcycle")
    }
```

```
}
```

Clase que utiliza el patrón Factory Method para crear objetos de tipo `Vehicle`:

```
class VehicleFactory {

    fun createVehicle(type: String): Vehicle? {

        when (type) {

            "car" -> return Car()

            "motorcycle" -> return Motorcycle()

            else -> return null

        }

    }

}
```

Utilización de la clase `VehicleFactory`:

```
val factory = VehicleFactory()

val car = factory.createVehicle("car")

car?.startEngine()

car?.drive()

car?.stopEngine()

// El resultado de la ejecución sería:
```

```
// Starting car engine

// Driving car

// Stopping car engine
```

Como se puede ver, la clase `VehicleFactory` utiliza el patrón Factory Method para crear objetos de tipo `Vehicle` de acuerdo al tipo especificado en el método `createVehicle`.

Esto permite manejar de forma genérica diferentes tipos de vehículos sin tener que crear una clase diferente para cada uno.

Otro ejemplo de uso del patrón Factory Method es en la creación de objetos de diferentes tipos de cajas de mensajes.

Por ejemplo, podemos tener una interfaz `MessageBox` que define los métodos necesarios para mostrar una caja de mensajes, y luego crear subclases concretas para cajas de mensajes de alerta, información y error.

La clase que utiliza el patrón Factory Method puede crear objetos de caja de mensajes de acuerdo al tipo especificado, lo que permite manejar de forma genérica diferentes tipos de cajas de mensajes.

En resumen, el patrón Factory Method es una técnica muy útil en la programación orientada a objetos ya que permite crear objetos de forma genérica sin tener que conocer la clase concreta del objeto.

Esto facilita la creación de código modular y reutilizable.

En Kotlin, podemos utilizar el patrón Factory Method mediante la definición de una interfaz o clase abstracta que defina los métodos necesarios para crear un objeto, y luego dejar la implementación de esos métodos en las subclases concretas.

Prototype

El patrón de diseño Prototype es un patrón creacional que se utiliza para crear nuevos objetos a partir de objetos ya existentes.

Este patrón se utiliza cuando el proceso de creación de un nuevo objeto puede ser costoso o complicado, ya sea porque el objeto es complejo o porque se requieren muchos pasos para crearlo.

En lugar de crear un nuevo objeto desde cero, el patrón Prototype permite crear una copia de un objeto ya existente, modificarla según sea necesario y devolver la copia como el nuevo objeto.

Un ejemplo de cómo se puede utilizar el patrón Prototype en Kotlin es la creación de una aplicación de dibujo.

Supongamos que en nuestra aplicación tenemos diferentes tipos de figuras geométricas, como círculos, cuadrados y triángulos, y que cada una de estas figuras tiene diferentes propiedades, como su posición en la pantalla, su tamaño y su color.

Cuando un usuario de la aplicación quiere dibujar una nueva figura en la pantalla, podemos utilizar el patrón Prototype para crear una copia de una figura ya existente y modificar sus propiedades para que se ajusten a lo que el usuario ha solicitado.
Para implementar el patrón Prototype en Kotlin, necesitamos crear una interfaz `Prototype` que defina un método `clone()` para crear una copia de un objeto.

Luego, cada una de las figuras geométricas de nuestra aplicación de dibujo debe implementar esta interfaz y proporcionar una implementación para el método `clone()`.

Por ejemplo, nuestra clase `Circle` podría tener la siguiente implementación:

```
class Circle(var x: Int, var y: Int, var radius: Int, var color:
String): Prototype {

    override fun clone(): Circle {

        return Circle(x, y, radius, color)

    }

}
```

En este caso, el método `clone()` simplemente devuelve una nueva instancia de la clase `Circle` con las mismas propiedades que el objeto original.

Luego, para crear una nueva figura basada en una figura existente, podemos utilizar el método `clone()` de esta manera:

```
val circle1 = Circle(10, 10, 5, "red")

val circle2 = circle1.clone()
```

En este ejemplo, primero creamos un círculo rojo en la posición (10, 10) con un radio de 5.

Luego, creamos una copia de este círculo utilizando el método `clone()` y almacenamos la copia en la variable `circle2`.

Ahora, podemos modificar las propiedades de `circle2` para que se ajusten a lo que el usuario ha solicitado, como cambiar su posición en la pantalla o su color.

Otro ejemplo de cómo se puede utilizar el patrón Prototype en Kotlin es en el contexto de una aplicación de sistema de archivos.

Supongamos que nuestra aplicación permite a los usuarios crear y modificar archivos y carpetas en un sistema de archivos virtual.

En lugar de tener que crear un nuevo archivo o carpeta desde cero cada vez que un usuario lo solicita, podemos utilizar el patrón Prototype para crear una copia de un archivo o carpeta ya existente y modificarla según sea necesario.

Para implementar el patrón Prototype en este contexto, podemos utilizar la misma interfaz `Prototype` que definimos anteriormente, pero en este caso necesitamos tener una clase `File` y una clase `Folder` que implementen esta interfaz.

La clase `File` podría tener una implementación similar a la siguiente:

```
class File(var name: String, var content: String): Prototype {

    override fun clone(): File {

        return File(name, content)

    }

}
```

En este caso, el método `clone()` devuelve una nueva instancia de la clase `File` con las mismas propiedades que el objeto original.

Luego, para crear una copia de un archivo existente, podemos utilizar el método `clone()` de esta manera:

```
val file1 = File("file1.txt", "This is the content of file1.txt")

val file2 = file1.clone()
```

En este ejemplo, primero creamos un archivo con el nombre «file1.txt» y el contenido «This is the content of file1.txt».

Luego, creamos una copia de este archivo utilizando el método `clone()` y almacenamos la copia en la variable `file2`. Ahora, podemos modificar las propiedades de `file2`, como su nombre o su contenido, para que se ajusten a lo que el usuario ha solicitado.

En resumen, el patrón Prototype es un patrón creacional que se utiliza para crear nuevos objetos a partir de objetos ya existentes.

Este patrón se utiliza cuando el proceso de creación de un nuevo objeto puede ser costoso o complicado, ya sea porque el objeto es complejo o porque se requieren muchos pasos para crearlo.

En Kotlin, podemos implementar el patrón Prototype mediante la definición de una interfaz `Prototype` con un método `clone()` y la implementación de esta interfaz en las clases que deseemos utilizar con este patrón.

Luego, podemos utilizar el método `clone()` para crear una copia de un objeto existente y modificarla según sea necesario.

El patrón Prototype nos permite ahorrar tiempo y esfuerzo en la creación de nuevos objetos, ya que nos permite utilizar objetos ya existentes como base para crear las copias.

Adapter

El patrón de diseño Adapter es una técnica muy utilizada en el desarrollo de software para adaptar una interfaz de una clase a otra interfaz que se espera utilizar.

Esto permite que las clases que no sean compatibles entre sí puedan trabajar juntas y ofrecer una funcionalidad similar.

Un ejemplo común de utilización de este patrón es cuando se desea utilizar una librería o una clase que no se ajusta completamente a nuestras necesidades y requerimientos.

En lugar de reescribir completamente la clase o librería, podemos utilizar un Adapter para adaptar su interfaz a la que necesitamos.

En Kotlin, podemos utilizar el patrón Adapter mediante la implementación de una clase adaptadora que herede de la clase o librería que queremos adaptar y sobreescribir los métodos necesarios para adaptarlos a nuestras necesidades.

A continuación, un ejemplo de cómo se podría implementar un Adapter en Kotlin:

```
// Clase original que queremos adaptar

class OriginalClass {

    fun originalMethod() {

        // Código de la función original
```

```
    }
}

// Adapter que sobrescribe la función original para adaptarla a nuestras necesidades
class Adapter : OriginalClass() {
    override fun originalMethod() {
        // Código adaptado de la función original
    }
}
```

Ejemplo concreto del patrón adapter

Un ejemplo concreto de utilización del patrón Adapter en Kotlin podría ser el siguiente:

Imaginémonos que tenemos una aplicación que utiliza una librería externa para gestionar la autenticación de usuarios.

La librería ofrece una interfaz que permite iniciar sesión, cerrar sesión y obtener información del usuario autenticado.

Sin embargo, la información que ofrece la librería no se ajusta exactamente a lo que necesitamos en nuestra aplicación, por lo que decidimos utilizar un Adapter para adaptar la interfaz de la librería a nuestras necesidades.

Para ello, creamos una clase `AuthenticationAdapter` que herede de la clase de la librería de autenticación y sobrescriba el método que nos devuelve la información del usuario autenticado.

La clase `AuthenticationAdapter` podría quedar de la siguiente manera:

```
// Clase de la librería de autenticación

class AuthenticationLibrary {

    fun login() {

        // Código para iniciar sesión

    }

    fun logout() {

        // Código para cerrar sesión

    }

    fun getUserInformation(): String {

        // Devuelve un string con la información del usuario autenticado

    }

}

// Adapter que adapta la interfaz de la librería de autenticación a nuestras necesidades
```

```
class AuthenticationAdapter : AuthenticationLibrary() {

    override fun getUserInformation(): User {

        // Devuelve un objeto Usuario con la información del
usuario autenticado

        val userInformation = super.getUserInformation()

        val data = userInformation.split(",")

        return User(data[0], data[1], data[2])

    }

}
```

De esta manera, podemos utilizar la clase `AuthenticationAdapter` en nuestra aplicación en lugar de la clase `AuthenticationLibrary` y aprovechar su funcionalidad, pero adaptada a nuestras necesidades.

Además, el código que utiliza la clase `AuthenticationAdapter` no necesita conocer los detalles de cómo se ha adaptado la función `getUserInformation()`, lo que permite una mayor reutilización del código.

En resumen, el patrón de diseño Adapter es una técnica muy útil en el desarrollo de software que permite adaptar una interfaz a otra que se espera utilizar, lo que facilita la interoperabilidad entre clases y librerías que no son compatibles entre sí.

En Kotlin, se puede implementar mediante la creación de una clase adaptadora que herede de la clase a adaptar y sobrescriba los métodos necesarios para adaptarlos a nuestras necesidades.

Bridge

El patrón de diseño Bridge es un patrón estructural que permite separar la implementación de una clase de su interfaz, de modo que ambos puedan variar de forma independiente.

Esto permite una mayor flexibilidad y reutilización del código, y es muy útil cuando se trabaja con clases que tienen muchas variantes y que pueden ser combinadas de diferentes maneras.

En Kotlin, el patrón Bridge se implementa mediante la creación de una interfaz que define la funcionalidad de la clase, y una clase abstracta que implementa dicha interfaz y que contiene una referencia a un objeto de otra clase que implementa la implementación concreta de la funcionalidad.

Esto se ilustra en el siguiente ejemplo:

```
interface DrawAPI {

    fun drawCircle(radius: Int, x: Int, y: Int)

}

abstract class Shape(val drawAPI: DrawAPI) {
```

```
    abstract fun draw()

}

class Circle(x: Int, y: Int, radius: Int, drawAPI: DrawAPI) :
Shape(drawAPI) {

    override fun draw() {

        drawAPI.drawCircle(radius, x, y)

    }

}
```

En este ejemplo, la interfaz `DrawAPI` define la funcionalidad para dibujar un círculo, y la clase abstracta `Shape` contiene una referencia a un objeto que implementa esta interfaz.

La clase concreta `Circle` hereda de `Shape` y utiliza su referencia a `DrawAPI` para dibujar un círculo en pantalla.

Esto permite que diferentes implementaciones de `DrawAPI` puedan ser utilizadas para dibujar círculos de diferentes maneras, como por ejemplo utilizando diferentes librerías gráficas o incluso utilizando diferentes medios como un canvas en una aplicación web o una superficie en una aplicación de escritorio.

```
class CanvasDrawAPI : DrawAPI {

    override fun drawCircle(radius: Int, x: Int, y: Int) {

        // Dibuja un círculo en un canvas
```

```
    }
}

class SurfaceDrawAPI : DrawAPI {

    override fun drawCircle(radius: Int, x: Int, y: Int) {

        // Dibuja un círculo en una superficie

    }
}
```

De esta forma, el patrón Bridge permite separar la lógica de la clase `Shape` de la implementación concreta de cómo se dibuja un círculo, lo que facilita la reutilización y extensión del código.

Por ejemplo, podríamos agregar una nueva clase `Square` que utilice la misma interfaz `DrawAPI` y la misma clase abstracta `Shape` para dibujar un cuadrado en lugar de un círculo, sin necesidad de modificar el código existente.

```
class Square(x: Int, y: Int, side: Int, drawAPI: DrawAPI) :
Shape(drawAPI) {

    override fun draw() {

        drawAPI.drawSquare(side, x, y)

    }
}
```

En resumen, el patrón Bridge es una herramienta muy útil en la programación orientada a objetos, ya que permite separar la implementación de una clase de su interfaz, lo que facilita la reutilización y extensión del código.

En Kotlin, se puede implementar mediante la creación de una interfaz que define la funcionalidad de la clase y una clase abstracta que contiene una referencia a un objeto que implementa dicha interfaz.

Composite

El patrón de diseño Composite es una técnica utilizada en programación orientada a objetos para tratar grupos de objetos de manera uniforme, como si fueran un solo objeto.

Este patrón permite a los desarrolladores construir estructuras jerárquicas y manejar los elementos de forma recursiva.

En Kotlin, el patrón Composite se implementa mediante la interface Composable y la clase Composite, que se utilizan para definir la estructura del grupo de objetos y sus operaciones respectivamente.

Estructura de implementación en Kotlin

```kotlin
// Interface que define la operación composable

interface Composable {

    fun operation(): String

}

// Clase que representa un objeto individual del grupo

class IndividualObject: Composable {

    override fun operation() = "Individual object operation"

}

// Clase que representa el grupo de objetos y sus operaciones

class Composite: Composable {

    private val objects = mutableListOf<Composable>()

    fun add(object: Composable) {

        objects.add(object)

    }

    fun remove(object: Composable) {

        objects.remove(object)

    }
```

```
    override fun operation() = objects.map { it.operation() }.joinToString(separator = "\n")
}

// Creación del grupo de objetos y su uso
fun main() {
    val composite = Composite()
    composite.add(IndividualObject())
    composite.add(IndividualObject())
    println(composite.operation()) // Imprime la operación de cada objeto del grupo
}
```

Ventajas del patrón Composite

- Permite tratar a un grupo de objetos de manera uniforme, lo que facilita su manejo y organización.
- Permite construir estructuras jerárquicas y manejarlas de manera recursiva.
- Provee una interfaz común para todos los objetos del grupo, lo que facilita su extensión y modificación.

Desventajas del patrón Composite

- Puede ser complejo de implementar y difícil de entender para programadores no familiarizados con el patrón.
- Puede aumentar la complejidad del código y dificultar su depuración.

Ejemplo

Un ejemplo concreto del uso del patrón Composite puede ser el desarrollo de una aplicación de dibujo en la que se puedan crear y manipular diferentes figuras geométricas.

Por ejemplo, se puede implementar una clase abstracta `Figure` que represente a una figura genérica y tenga métodos para dibujarla, moverla y obtener su área.

Luego, se pueden crear clases concretas como `Circle`, `Rectangle` y `Triangle` que hereden de `Figure` y implementen las operaciones específicas de cada figura.

Para manejar grupos de figuras, se puede utilizar el patrón Composite y crear una clase `Group` que implemente la interface `Composable` y permita agregar y eliminar figuras individuales del grupo.

De esta manera, se pueden tratar los grupos de figuras de manera uniforme, como si fueran una sola figura, y manejarlos de manera recursiva.

Aquí está un ejemplo de implementación en Kotlin:

```
// Clase abstracta que representa a una figura genérica

abstract class Figure {

    abstract fun draw()

    abstract fun move(x: Int, y: Int)

    abstract fun area(): Double

}

// Clases concretas que representan figuras específicas

class Circle(val radius: Double): Figure() {

    override fun draw() = println("Dibujando círculo de radio $radius")

    override fun move(x: Int, y: Int) = println("Moviendo círculo a ($x, $y)")

    override fun area() = Math.PI * radius * radius

}

class Rectangle(val width: Double, val height: Double): Figure() {

    override fun draw() = println("Dibujando rectángulo de ancho $width y alto $height")

    override fun move(x: Int, y: Int) = println("Moviendo rectángulo a ($x, $y)")
```

```
    override fun area() = width * height
}

class Triangle(val base: Double, val height: Double): Figure() {
    override fun draw() = println("Dibujando triángulo de base $base y altura $height")
    override fun move(x: Int, y: Int) = println("Moviendo triángulo a ($x, $y)")
    override fun area() = base * height / 2
}

// Clase que representa un grupo de figuras y sus operaciones
class Group: Composable {
    private val figures = mutableListOf<Figure>()
    fun add(figure: Figure) {
        figures.add(figure)
    }
    fun remove(figure: Figure) {
        figures.remove(figure)
    }
    override fun operation() = figures.forEach { it.draw() }
}
```

```
// Creación de un grupo de figuras y su uso

fun main() {

    val group = Group()

    group.add(Circle(5.0))

    group.add(Rectangle(10.0, 5.0))

    group.add(Triangle(10.0, 5.0))

    group.operation() // Dibuja cada figura del grupo

}
```

En este caso, se crea un grupo de figuras y se agrega un círculo, un rectángulo y un triángulo.

Luego, se llama al método operation() del grupo, que dibuja cada una de las figuras agregadas.

De esta manera, se puede tratar el grupo de figuras como si fuera una sola figura y manejarlo de manera uniforme.

También se pueden crear grupos de grupos y manejarlos de manera recursiva.

Conclusión

En conclusión, el patrón Composite es una técnica útil en programación orientada a objetos para tratar grupos de objetos de manera uniforme.

Su correcta implementación puede mejorar la organización y manejo de estructuras jerárquicas en el código.

Sin embargo, es importante tener en cuenta sus desventajas y utilizarlo de manera adecuada en cada caso.

Decorator

El patrón de diseño Decorator es uno de los patrones estructurales más utilizados en la programación orientada a objetos.

Este patrón permite agregar funcionalidades adicionales a un objeto de forma dinámica, sin necesidad de crear subclases para cada combinación de funcionalidades.

En Kotlin, el patrón Decorator se implementa mediante la creación de una interfaz que define la funcionalidad base del objeto a decorar, y una clase base que implementa esta interfaz y que servirá como punto de partida para crear las decoraciones.
Las decoraciones son clases que implementan la misma interfaz que el objeto base, y que contienen una referencia a un objeto de la clase base.

Por ejemplo, imagina que queremos crear una aplicación que permita a los usuarios comprar pizzas con diferentes ingredientes.

Para ello, podemos crear una interfaz `Pizza` que define la funcionalidad básica de una pizza, como su tamaño y su precio, y una clase base `PizzaBase` que implementa esta interfaz y que representa una pizza básica sin ingredientes adicionales.

```
interface Pizza {

    val size: Int

    val price: Double

}

class PizzaBase(override val size: Int) : Pizza {

    override val price: Double

        get() = 10.0 * size

}
```

Luego, podemos crear las diferentes decoraciones para agregar ingredientes a la pizza.

Por ejemplo, una clase `PepperoniPizzaDecorator` que agrega pepperoni a la pizza y aumenta su precio en 2 dólares.

```
class PepperoniPizzaDecorator(private val pizza: Pizza) : Pizza {
```

```
    override val size: Int

        get() = pizza.size

    override val price: Double

        get() = pizza.price + 2

}
```

De esta forma, podemos crear una pizza de tamaño mediano con pepperoni de la siguiente manera:

```
val pizza = PepperoniPizzaDecorator(PizzaBase(size = 2))

println(pizza.price) // imprime 14.0
```

Además, podemos crear combinaciones de ingredientes creando decoraciones que a su vez contengan otras decoraciones.

Por ejemplo, una clase `ExtraCheesePizzaDecorator` que agrega queso extra a la pizza y aumenta su precio en 1 dólar.

```
class ExtraCheesePizzaDecorator(private val pizza: Pizza) : Pizza
{

    override val size: Int

        get() = pizza.size

    override val price: Double

        get() = pizza.price + 1
```

```
}
```

De esta forma, podemos crear una pizza de tamaño grande con pepperoni y extra queso de la siguiente manera:

```
val pizza =
ExtraCheesePizzaDecorator(PepperoniPizzaDecorator(PizzaBase(size =
3)))

println(pizza.price) // imprime 17.0
```

Como se puede ver, el patrón Decorator nos permite agregar funcionalidades adicionales a un objeto de forma dinámica y sin necesidad de crear subclases para cada combinación posible.

Esto nos permite mantener un código limpio y fácil de extender en el futuro.

En resumen, el patrón Decorator es una herramienta muy útil en la programación orientada a objetos, y en Kotlin podemos implementarlo de forma sencilla utilizando interfaces y clases.

Facade

El patrón de diseño Facade es un patrón de diseño estructural que se utiliza para simplificar la interacción entre diferentes componentes de un sistema.

Este patrón se enfoca en proporcionar una interfaz única y fácil de usar para acceder a las funcionalidades de un sistema complejo.

En Kotlin, podemos implementar el patrón Facade mediante la creación de una clase de interfaz que se encargue de delegar las llamadas a los diferentes componentes del sistema.

Estructura del patrón Facade

A continuación, se muestra un ejemplo de cómo implementar el patrón Facade en Kotlin:

```
class Facade {

    private val component1 = Component1()

    private val component2 = Component2()

    private val component3 = Component3()

    fun operation1() {

        component1.operation1()

        component2.operation1()

    }

    fun operation2() {
```

```
        component1.operation2()

        component3.operation2()

    }
}
```

En el ejemplo anterior, la clase Facade tiene acceso a los componentes 1, 2 y 3, y se encarga de delegar las llamadas a sus respectivas operaciones.

De esta manera, los clientes que utilizan la clase Facade solo tienen que preocuparse por llamar a las operaciones expuestas por la interfaz, en lugar de tener que lidiar con los detalles de cada componente individualmente.

Ventajas del patrón Facade

- Simplifica la interacción entre componentes: el patrón Facade proporciona una interfaz única y fácil de usar para acceder a las funcionalidades de un sistema complejo.
- Mejora la legibilidad del código: al utilizar el patrón Facade, se pueden agrupar las operaciones relacionadas en una sola interfaz, lo que mejora la legibilidad del código y facilita su mantenimiento.

- Reduce la complejidad del código: el patrón Facade permite desacoplar los componentes del sistema, lo que reduce la complejidad del código y facilita su depuración y testing.

Desventajas del patrón Facade

- Puede aumentar la complejidad en algunos casos: en algunos casos, el patrón Facade puede generar una capa adicional de abstracción que aumente la complejidad del código.
- No es adecuado para todos los casos: el patrón Facade solo es adecuado para sistemas con una cierta complejidad, y no es recomendable utilizarlo en sistemas sencillos.

Ejemplo del patrón Facade

Un ejemplo concreto del patrón Facade podría ser el siguiente: supongamos que tenemos un sistema que se encarga de gestionar el inventario de una tienda.

Este sistema está compuesto por diferentes componentes, como una clase que se encarga de manejar el stock de productos, otra que se encarga de realizar pedidos a los

proveedores, y otra que se encarga de generar reportes del inventario.

Para simplificar la interacción con este sistema, podríamos crear una clase de interfaz que se encargue de delegar las llamadas a los diferentes componentes.

La clase de interfaz, que sería el patrón Facade en este caso, podría tener métodos como `addProduct()` para agregar un producto al inventario, `makeOrder()` para hacer un pedido a un proveedor, o `generateReport()` para generar un reporte del inventario.

De esta manera, los clientes del sistema solo tendrían que preocuparse por llamar a estos métodos de la interfaz, en lugar de tener que lidiar con los detalles de cada componente individualmente.

A continuación se muestra un ejemplo de cómo podría implementarse el patrón Facade en este caso en Kotlin:

```
class InventoryFacade {

    private val stockManager = StockManager()

    private val orderManager = OrderManager()

    private val reportGenerator = ReportGenerator()

    fun addProduct(product: Product) {

        stockManager.addProduct(product)
```

```
    }

    fun makeOrder(order: Order) {
        orderManager.makeOrder(order)
    }

    fun generateReport() {
        reportGenerator.generateReport()
    }
}
```

En el ejemplo anterior, la clase `InventoryFacade` es la interfaz que se encarga de delegar las llamadas a los diferentes componentes del sistema.

Los clientes del sistema pueden utilizar esta clase para realizar tareas como agregar productos al inventario, realizar pedidos a proveedores o generar reportes del inventario.

Conclusión

En resumen, el patrón de diseño Facade es una herramienta útil para simplificar la interacción entre componentes en un sistema complejo.
En Kotlin, podemos implementar el patrón Facade mediante la creación de una clase de interfaz que se encargue de delegar las llamadas a los diferentes componentes del sistema.

El patrón Facade tiene varias ventajas, como la simplificación de la interacción entre componentes, mejora de la legibilidad del código y reducción de la complejidad del código.

Sin embargo, también tiene algunas desventajas, como la posible aumento de la complejidad en algunos casos y no ser adecuado para todos los casos.

En conclusión, el patrón Facade es una opción a considerar cuando se trabaja con sistemas complejos y se desea simplificar la interacción entre sus componentes.

Su uso en Kotlin puede ser muy útil para mejorar la legibilidad y mantenibilidad del código.

Flyweight

El patrón de diseño Flyweight es un patrón de diseño estructural que se utiliza para optimizar la utilización de memoria en una aplicación.

Este patrón se basa en la idea de compartir objetos que se repiten en una aplicación, en lugar de crear una nueva instancia de cada uno de ellos.

Este patrón se aplica en situaciones donde una aplicación necesita manejar un gran número de objetos similares, como

por ejemplo en un juego donde se necesitan dibujar muchos enemigos en pantalla.

En lugar de crear una nueva instancia de cada enemigo, el patrón Flyweight permite compartir una única instancia entre todos los enemigos del mismo tipo.

En Kotlin, se puede implementar el patrón Flyweight utilizando una interfaz y una clase concreta que implemente esa interfaz.

La interfaz se utiliza para definir los métodos comunes que tendrán todas las instancias del objeto compartido, mientras que la clase concreta se encarga de implementar esos métodos.

Por ejemplo, supongamos que queremos crear una aplicación que dibuje enemigos en pantalla.

Podemos utilizar el patrón Flyweight para compartir una única instancia de cada tipo de enemigo.

La interfaz podría definirse de la siguiente manera:

```
interface Enemy {

    fun draw(x: Int, y: Int)

}
```

La clase concreta que implemente esta interfaz sería la encargada de dibujar el enemigo en pantalla en la posición indicada.

Por ejemplo, una clase que dibuje un enemigo rojo podría definirse de la siguiente manera:

```
class RedEnemy : Enemy {

    override fun draw(x: Int, y: Int) {

        // Dibujar enemigo rojo en la posición (x, y)

    }

}
```

Una vez que tenemos la interfaz y la clase concreta que implementa esa interfaz, podemos utilizar el patrón Flyweight para compartir una única instancia de cada tipo de enemigo.

Esto se puede hacer utilizando una clase que se encargue de almacenar las instancias compartidas y devolverlas cuando sean necesarias.

Esta clase se conoce como «factoría» y puede definirse de la siguiente manera:

```
class EnemyFactory {

    private val enemies = mutableMapOf<String, Enemy>()

    fun getEnemy(type: String): Enemy {

        if (enemies.containsKey(type)) {

            return enemies[type]!!
```

```
        } else {
            val enemy = when (type) {
                "red" -> RedEnemy()
                "blue" -> BlueEnemy()
                // etc.
                else -> throw IllegalArgumentException("Invalid enemy type")
            }
            enemies[type] = enemy
            return enemy
        }
    }
}
```

Esta clase almacena las instancias compartidas en un mapa, utilizando como clave el tipo de enemigo.
Cuando se solicita un enemigo, la clase verifica si ya existe una instancia compartida del tipo solicitado y, en caso contrario, crea una nueva instancia y la almacena en el mapa.

Para utilizar esta clase, basta con crear una instancia de la misma y utilizar el método `getEnemy()` para obtener un enemigo. Por ejemplo:

```
val factory = EnemyFactory()

val redEnemy = factory.getEnemy("red")

redEnemy.draw(10, 20)
```

De esta manera, se utiliza el patrón Flyweight para compartir una única instancia del enemigo rojo entre todos los lugares de la aplicación que lo necesiten.

Esto permite reducir el uso de memoria, ya que sólo se crea una única instancia del enemigo rojo, en lugar de crear una nueva instancia en cada lugar que se utilice.

En resumen, el patrón de diseño Flyweight es un patrón de diseño estructural que se utiliza para optimizar el uso de memoria en una aplicación.

Este patrón se basa en la idea de compartir objetos que se repiten en una aplicación, en lugar de crear una nueva instancia de cada uno de ellos.

En Kotlin, se puede implementar el patrón Flyweight utilizando una interfaz y una clase concreta que implemente esa interfaz, y una clase «factoría» que se encargue de almacenar y devolver las instancias compartidas.

Proxy

El patrón de diseño Proxy es una técnica comúnmente utilizada en programación orientada a objetos para proporcionar una interfaz de acceso a un objeto real.

Esto se logra mediante la creación de un objeto «proxy» que actúa como intermediario entre el usuario y el objeto real, permitiendo controlar y manipular el acceso a ese objeto.

Ejemplo Proxy 1: Calculadora

En Kotlin, podemos implementar el patrón de diseño Proxy utilizando la palabra clave `by` en la declaración de una interfaz.

Por ejemplo, si tenemos una interfaz llamada `Calculator` que tiene un método llamado `sum`:

```
interface Calculator {

    fun sum(a: Int, b: Int): Int

}
```

Podemos crear una clase llamada `CalculatorProxy` que implemente esta interfaz y actúe como intermediario para el objeto real que hace la operación de suma:

```
class CalculatorProxy(private val realCalculator: Calculator) :
Calculator by realCalculator
```

En este caso, la clase `CalculatorProxy` está delegando la implementación del método `sum` al objeto `realCalculator`, que es el objeto real que hará la operación de suma.

De esta manera, podemos controlar el acceso a la operación de suma a través del objeto `CalculatorProxy`, permitiendo por ejemplo validar los parámetros de entrada o realizar algún tipo de cálculo adicional antes de delegar la operación al objeto real.

Ejemplo Proxy 2: Servidor remoto

Otro ejemplo de utilización del patrón de diseño Proxy en Kotlin puede ser el acceso a un servicio remoto. Supongamos que tenemos una interfaz llamada `RemoteService` que tiene un método llamado `getData()`:

```
interface RemoteService {

    fun getData(id: Int): String

}
```

Podemos crear una clase llamada `RemoteServiceProxy` que implemente esta interfaz y actúe como intermediario para el objeto real que hace la llamada al servicio remoto:

```
class RemoteServiceProxy(private val realRemoteService:
RemoteService) : RemoteService by realRemoteService
```

En este caso, la clase `RemoteServiceProxy` está delegando la implementación del método «obtenerDatos» al objeto `realRemoteService`, que es el objeto real que hace la llamada al servicio remoto.

De esta manera, podemos controlar el acceso a la llamada al servicio remoto a través del objeto `RemoteServiceProxy`, permitiendo por ejemplo realizar validaciones de seguridad antes de delegar la llamada al objeto real.

Además, el patrón de diseño Proxy también nos permite implementar caché en la respuesta del servicio remoto, evitando realizar llamadas innecesarias si ya se ha obtenido previamente la misma información.

Conclusión

En resumen, el patrón de diseño Proxy nos permite proporcionar una interfaz de acceso controlado a un objeto real, permitiendo implementar funcionalidades adicionales como validaciones de seguridad o caché de respuestas.

Esto puede ser muy útil en diferentes escenarios, como en el acceso a servicios remotos o en la implementación de lógica de negocio compleja.

En Kotlin, podemos implementar el patrón de diseño Proxy utilizando la palabra clave `by` en la declaración de una interfaz, delegando la implementación de sus métodos a un objeto real.

Command

El patrón de diseño Command es un patrón de diseño de software que se utiliza para encapsular una solicitud como un objeto, de modo que se pueda parametrizar con diferentes solicitudes, enviar solicitudes por separado a colas o registrar y reproducir operaciones.

Este patrón se basa en la idea de tratar las operaciones de la misma manera que los datos, lo que permite tratar a las operaciones de forma genérica y separar la ejecución de una operación de su implementación.

Ejemplo 1: Sistema de luces inteligentes

Un ejemplo común de la utilización del patrón Command es en la implementación de un sistema de control de luces en una casa inteligente.

En este caso, cada luz podría ser representada por un objeto que puede ejecutar objetos de tipo `Command`, y cada acción que se pueda realizar sobre una luz (encender, apagar,

cambiar de color, etc.) se implementaría como una clase concreta que implemente la interfaz `Command`.

De esta forma, se pueden crear diferentes objetos `Command` que representen diferentes acciones sobre una luz, y enviarlos al objeto Luz para que se ejecuten.

En Kotlin, podemos implementar el patrón Command de la siguiente manera:

```
interface Command {

    fun execute()

}

class TurnOnLight: Command {

    override fun execute() {

        // Code to turn on a light

    }

}

class TurnOffLight: Command {

    override fun execute() {

        // Code to turn off a light

    }

}
```

```
class Light {

    private val commands = mutableListOf<Command>()

    fun addCommand(command: Command) {

        commands.add(command)

    }

    fun executeCommands() {

        commands.forEach { it.execute() }

    }

}

fun main() {

    val light = Light()

    light.addCommand(TurnOnLight())

    light.addCommand(TurnOffLight())

    light.executeCommands()

}
```

En este ejemplo, hemos creado una interfaz `Command` que define el método execute(), que será implementado por las clases concretas que representen diferentes acciones sobre una luz.

Luego, hemos creado dos clases concretas que implementan la interfaz Command: `TurnOnLight` y `TurnOffLight`.

Estas clases contienen la lógica para encender y apagar una luz, respectivamente.

Finalmente, hemos creado una clase `Light` que tiene una lista de objetos `Command` y un método que permite añadir nuevos objetos `Command` a esta lista.

La clase `Light` también tiene un método `executeCommands()` que itera sobre la lista de objetos `Command` y ejecuta el método `execute()` de cada uno de ellos.

De esta forma, se pueden enviar diferentes acciones a una luz y ejecutarlas en el orden en que se han agregado a la lista de commands.

Por ejemplo, en el código anterior podríamos crear una clase `ChangeLightColor` que implemente la interfaz Command y agregarla a la lista de commands en la luz.

Cuando se llame al método *executeCommands()*, se ejecutarán en orden las acciones de encender la luz, cambiarle el color y apagarla.

De esta forma, el patrón *Command* nos permite tratar a las operaciones de un sistema de forma genérica y separar su ejecución de su implementación, lo que nos permite crear sistemas flexibles y modulares.

Ejemplo 2: Sistema de ventanas de escritorio

En este ejemplo, hemos creado una interfaz `Command` que define el método `execute()`, que será implementado por las clases concretas que representen diferentes acciones sobre una ventana emergente.

Luego, hemos creado tres clases concretas que implementan la interfaz `Command`: `OpenWindow`, `CloseWindow` y `ResizeWindow`.

Estas clases contienen la lógica para abrir, cerrar y cambiar el tamaño de una ventana emergente, respectivamente.

Finalmente, hemos creado una clase `PopUpWindow` que tiene una lista de objetos `Command` y un método que permite añadir nuevos objetos `Command` a esta lista.

La clase `PopUpWindow` también tiene un método `executeCommands()` que itera sobre la lista de objetos `Command` y ejecuta el método `execute()` de cada uno de ellos.

De esta forma, se pueden enviar diferentes acciones a una ventana emergente y ejecutarlas en el orden en que se han agregado a la lista de commands.

Aquí puedes ver un ejemplo en Kotlin de cómo se podría implementar el patrón `Command` en este caso:

```
interface Command {

 fun execute()

}

class OpenWindow: Command {

 override fun execute() {

   // Código para abrir una ventana emergente

 }

}

class CloseWindow: Command {

 override fun execute() {

   // Código para cerrar una ventana emergente

 }

}

class ResizeWindow: Command {

 override fun execute() {

   // Código para cambiar de tamaño una ventana emergente

 }
```

```
}

class PopUpWindow {

    private val commands = mutableListOf<Command>()

    fun addCommand(command: Command) {

        commands.add(command)

    }

    fun executeCommands() {

        commands.forEach { it.execute() }

    }

}

fun main() {

    val window = PopUpWindow()

    window.addCommand(OpenWindow())

    window.addCommand(ResizeWindow())

    window.addCommand(CloseWindow())

    window.executeCommands()

}
```

Conclusión

En conclusión, el patrón de diseño Command es un patrón de diseño de software que nos permite encapsular una solicitud como un objeto y tratar a las operaciones de forma genérica y separada de su implementación.

Esto nos permite crear sistemas flexibles y modulares que permiten parametrizar diferentes solicitudes, enviar solicitudes por separado a colas o registrar y reproducir operaciones.

El patrón Command es ampliamente utilizado en diferentes contextos y puede ser implementado en cualquier lenguaje de programación, como lo hemos visto en nuestro ejemplo en Kotlin.

Chain of Responsibility

El patrón de diseño Chain of Responsibility es un patrón de diseño de software que permite a varios objetos intentar resolver una solicitud.

En lugar de enviar una solicitud directamente a un objeto específico, se envía a una cadena de objetos que pueden resolverla.

Cada objeto en la cadena tiene la oportunidad de manejar la solicitud.

Si un objeto no puede manejarla, se la pasa al siguiente objeto en la cadena hasta que se encuentra uno que pueda manejarla.

Este patrón es útil cuando hay varias posibles manejadores para una solicitud y no se sabe de antemano cuál es el adecuado.

También puede ser útil para evitar que una solicitud se envíe a un objeto que no puede manejarla, lo que puede mejorar el rendimiento y evitar errores.

Un ejemplo típico de la cadena de responsabilidad es un sistema de procesamiento de pagos.

Puede haber varios tipos de pagos, como pagos con tarjeta de crédito, pagos con PayPal y pagos en efectivo.

Cada uno de estos tipos de pago puede ser manejado por un objeto diferente.

En lugar de tener que comprobar cada tipo de pago y enviar la solicitud al objeto adecuado, se puede crear una cadena de objetos que manejan cada tipo de pago y enviar la solicitud a la cadena.

La cadena se encargará de enviar la solicitud al objeto adecuado.

En Kotlin, la cadena de responsabilidad se puede implementar de la siguiente manera.

Primero, nos creamos la interfaz que tiene que implementar un procesador de pago:

```
interface PaymentHandler {

    fun setNext(handler: PaymentHandler): PaymentHandler

    fun handle(request: PaymentRequest): PaymentResponse

}
```

Necesitamos varias clases que representen la petición de pago, la respuesta y los tipos de pago:

```
data class PaymentRequest(val paymentMethod: PaymentMethod)

data class PaymentResponse(val success: Boolean, val message:
String)

enum class PaymentMethod {

    CREDIT_CARD,

    PAYPAL,

    CASH

}
```

Aquí tienes tres posibles implementaciones: mediante tarjeta de crédito, PayPal, o efectivo:

```
class CreditCardHandler : PaymentHandler {

    private var next: PaymentHandler? = null

    override fun setNext(handler: PaymentHandler): PaymentHandler {

        next = handler

        return handler

    }

    override fun handle(request: PaymentRequest): PaymentResponse {

        if (request.paymentMethod == PaymentMethod.CREDIT_CARD) {

            // Procesar pago con tarjeta de crédito

            return PaymentResponse(true, "Pago procesado con
éxito")

        } else {

            if (next == null) {

                throw IllegalStateException("No se encontró un
manejador adecuado para el pago")

            }

            return next!!.handle(request)

        }
```

```
    }
}

class PaypalHandler : PaymentHandler {
    private var next: PaymentHandler? = null

    override fun setNext(handler: PaymentHandler): PaymentHandler {
        next = handler
        return handler
    }

    override fun handle(request: PaymentRequest): PaymentResponse {
        if (request.paymentMethod == PaymentMethod.PAYPAL) {
            // Procesar pago con PayPal
            return PaymentResponse(true, "Pago procesado con
éxito")
        } else {
            if (next == null) {
                throw IllegalStateException("No se encontró un
manejador adecuado para el pago")
            }
            return next!!.handle(request)
        }
```

```
    }
}

class CashHandler : PaymentHandler {

    private var next: PaymentHandler? = null

    override fun setNext(handler: PaymentHandler): PaymentHandler {

        next = handler

        return handler

    }

    override fun handle(request: PaymentRequest): PaymentResponse {

        if (request.paymentMethod == PaymentMethod.CASH) {

            // Procesar pago en efectivo

            return PaymentResponse(true, "Pago procesado con
éxito")

        } else {

            if (next == null) {

                throw IllegalStateException("No se encontró un
manejador adecuado para el pago")

            }

            return next!!.handle(request)

        }
```

```
    }

}
```

Y finalmente indicamos la cadena en con el orden en el que se ejecutará cada `handler`, con el fin de saber si puede o no consumir esa petición:

```
val chain: PaymentHandler = CreditCardHandler()

    .setNext(PaypalHandler())

    .setNext(CashHandler())
```

Y se le indica a la cadena que gestione la petición:

```
val response = chain.handle(PaymentRequest(PaymentMethod.PAYPAL))

println(response.message) // Imprimirá "Pago procesado con éxito"
```

En este ejemplo, se creó una cadena de manejadores para diferentes métodos de pago.

Cuando se envía una solicitud de pago a la cadena, se pasa a cada manejador hasta que se encuentra uno que pueda manejar el tipo de pago especificado en la solicitud.

Esto permite a la aplicación manejar diferentes tipos de pagos de manera flexible y eficiente.

En resumen, el patrón de diseño Chain of Responsibility es un patrón de diseño de software que permite a varios objetos intentar resolver una solicitud.

Esto puede ser útil cuando hay varios posibles manejadores para una solicitud y no se sabe de antemano cuál es el adecuado.

En Kotlin, se puede implementar mediante la creación de una interfaz de manejador de pagos y clases que implementan esta interfaz para manejar diferentes tipos de pagos.

Luego se pueden agregar estos manejadores a una cadena en el orden deseado y enviar solicitudes de pago a la cadena para ser manejadas por el manejador adecuado.

Interpreter

El patrón de diseño Interpreter es un patrón de diseño estructural que se utiliza para dar una interpretación a un lenguaje específico.

Esto se logra mediante la creación de un intérprete que se encarga de analizar y ejecutar las instrucciones del lenguaje.

Este patrón es útil cuando se desea crear un lenguaje de programación propio o cuando se necesita interpretar un lenguaje de programación existente de una manera diferente.

Además, permite cambiar la interpretación de un lenguaje sin afectar el código fuente del programa.

Ejemplo de Interpreter

Un ejemplo sencillo de uso del patrón Interpreter en Kotlin sería la creación de un intérprete para un lenguaje de operaciones matemáticas básico.

Primero, se crearía una clase abstracta denominada `Expression` que serviría como base para todas las expresiones del lenguaje.

Luego, se crearían clases concretas para cada tipo de operación matemática, como `SumExpression` y `MultiplyExpression`, que heredarían de la clase `Expression` y sobreescribirían el método `interpret()` para realizar la operación correspondiente.

Por último, se crearía una clase `Interpreter` que recibiría una cadena de texto con la expresión matemática a interpretar y utilizaría una estructura de datos, como una pila, para almacenar y evaluar las operaciones en el orden correcto.

La clase Interpreter podría verse así:

```
class Interpreter {
```

```
    fun interpret(expression: String): Double {

        // Crea una pila para almacenar las operaciones

        val stack = Stack<Expression>()

        // Divide la expresión en tokens

        val tokens = expression.split(" ")

        // Recorre cada token y crea la expresión correspondiente

        for (token in tokens) {

            when (token) {

                "+" -> stack.push(SumExpression())

                "*" -> stack.push(MultiplyExpression())

                else ->
stack.push(NumberExpression(token.toDouble()))

            }

        }

        // Realiza las operaciones en el orden correcto y devuelve
el resultado final

        return stack.pop().interpret()

    }

}
```

Con esto, se puede utilizar el intérprete de la siguiente manera:

```
val interpreter = Interpreter()

val result = interpreter.interpret("1 2 + 3 *") // 9.0
```

En este ejemplo, el intérprete analiza la expresión «1 2 + 3 *» y crea las expresiones correspondientes para cada token.

Luego, utiliza una pila para almacenar las expresiones en el orden correcto y realiza las operaciones matemáticas en el orden correcto, evaluando primero la multiplicación y luego la suma, dando como resultado el valor 9.0.

Este es un ejemplo básico del patrón Interpreter en Kotlin, pero se puede utilizar para interpretar lenguajes de programación más complejos y con más tipos de operaciones.

Además, el patrón permite cambiar la interpretación de un lenguaje sin afectar el código fuente del programa, lo que facilita la actualización y mantenimiento del código.

Conclusión

En resumen, el patrón de diseño Interpreter es una forma eficiente de interpretar lenguajes de programación y permite cambiar la interpretación de un lenguaje de manera sencilla.

Su uso en Kotlin es sencillo y permite crear intérpretes personalizados para diferentes lenguajes de programación.

Builder

El patrón de diseño Builder es un patrón de creación que permite construir objetos complejos de manera sencilla y organizada.

Este patrón se utiliza cuando tenemos una clase con una gran cantidad de parámetros, y queremos crear una interfaz clara y fácil de usar para construir objetos de esa clase.

En Kotlin, podemos implementar el patrón Builder mediante una clase interna que se encarga de construir el objeto y una clase externa que contiene los métodos de construcción.

Ejemplo del patrón Builder

Por ejemplo, supongamos que tenemos una clase Pizza que tiene varios parámetros, como el tamaño, el tipo de masa, los ingredientes y el precio.

Podemos implementar el patrón Builder para construir objetos de esta clase de la siguiente manera:

```
class Pizza {
```

```
    var size: String? = null

    var crust: String? = null

    var ingredients: List<String>? = null

    var price: Double? = null

    class Builder {

        private val pizza = Pizza()

        fun size(size: String) = apply { pizza.size = size }

        fun crust(crust: String) = apply { pizza.crust = crust }

        fun ingredients(vararg ingredients: String) = apply {
pizza.ingredients = ingredients.toList() }

        fun price(price: Double) = apply { pizza.price = price }

        fun build() = pizza

    }
}
```

La clase `Builder` tiene una propiedad privada llamada `pizza`, que es una instancia de la clase `Pizza`.

Los métodos `size`, `crust`, `ingredients` y `price` reciben sus respectivos parámetros y los establecen en el objeto pizza.

El método `build()` devuelve el objeto pizza completamente construido.

Para utilizar el patrón Builder, podemos crear un objeto de la clase `Pizza` de la siguiente manera:

```
val pizza = Pizza.Builder()
    .size("medium")
    .crust("thick")
    .ingredients("cheese", "pepperoni", "mushrooms")
    .price(12.50)
    .build()
```

En este ejemplo, estamos construyendo un objeto de la clase Pizza con un tamaño mediano, masa gruesa, ingredientes queso, pepperoni y champiñones, y un precio de 12.50 dólares.

El patrón Builder nos permite escribir código legible y organizado para crear objetos complejos.

¿Tiene sentido en Kotlin el patrón Builder, teniendo opciones como *apply*?

La realidad es que en Kotlin la función `apply()` hace las veces de Builder y prácticamente se puede usar de forma indistinta.

Pero en ocasiones nos puede interesar usar una implementación como la que hemos visto más arriba, si el

código que se produce en cada función es lo suficientemente complejo.

Aún así, muchas veces también podrá sustituirse por sobrescrituras de los getters de las properties.

Pero el patrón Builder es muy popular en cualquier lenguaje, y usarlo puede ayudar a razonar sobre el código.

Conclusión

En resumen, el patrón Builder es un patrón de creación que nos permite construir objetos complejos de manera sencilla y organizada.

En Kotlin, podemos implementar este patrón mediante una clase interna que se encarga de construir el objeto.

Iterator

El patrón de diseño Iterator es un patrón de diseño comúnmente utilizado en el desarrollo de software.

Este patrón permite recorrer una colección de elementos de manera secuencial sin conocer su implementación interna.

Implementación de Iterator: Opción 1

En Kotlin, podemos implementar el patrón de diseño Iterator utilizando la interfaz `Iterator<T>` y la clase `Iterable<T>`.

La interfaz `Iterator<T>` tiene dos métodos principales: `hasNext()` y `next()`.

El método `hasNext()` devuelve verdadero si hay más elementos en la colección para recorrer, mientras que el método `next()` devuelve el siguiente elemento en la colección.

Por ejemplo, supongamos que tenemos una clase que representa una lista de números enteros.

Podemos implementar la interfaz `Iterator<Int>` en esta clase para permitir el recorrido de los elementos de la lista de manera secuencial:

```
class IntList : Iterable<Int> {

    private val numbers = mutableListOf(1, 2, 3, 4, 5)

    override fun iterator(): Iterator<Int> {

        return object : Iterator<Int> {

            private var currentIndex = 0
```

```
            override fun hasNext(): Boolean {

                return currentIndex < numbers.size

            }

            override fun next(): Int {

                return numbers[currentIndex++]

            }

        }

    }
}
```

Ahora, podemos utilizar la clase `IntList` para recorrer sus elementos utilizando un ciclo `for-in`:

```
val list = IntList()

for (number in list) {

    println(number)

}
```

El código anterior imprimirá cada uno de los elementos de la lista de números enteros en consola.

Implementación de Iterator: Opción 2

Otra forma de utilizar el patrón de diseño Iterator en Kotlin es implementando la interfaz `Iterable<T>` en la clase que representa nuestra colección de elementos.

Esta interfaz tiene un único método: `iterator()`. Este método debe devolver un objeto que implemente la interfaz `Iterator<T>`.

Por ejemplo, podemos modificar nuestra clase `IntList` para implementar la interfaz `Iterable<Int>` en lugar de crear un objeto que implemente `Iterator<Int>` dentro de su método `iterator()`:

```
class IntList : Iterable<Int> {

    private val numbers = mutableListOf(1, 2, 3, 4, 5)

    override fun iterator(): Iterator<Int> {

        return IntListIterator(numbers)

    }

}

class IntListIterator(private val numbers: List<Int>) :
Iterator<Int> {

    private var currentIndex = 0

    override fun hasNext(): Boolean {

        return currentIndex < numbers.size

    }
```

```
    override fun next(): Int {

        return numbers[currentIndex++]

    }

}
```

Ahora, podemos utilizar nuestra clase `IntList` para recorrer sus elementos utilizando un ciclo `for-in`:

```
val list = IntList()

for (number in list) {

    println(number)

}
```

El código anterior imprimirá cada uno de los elementos de la lista de números enteros en consola.

Conclusión

En resumen, el patrón de diseño Iterator nos permite recorrer una colección de elementos de manera secuencial sin conocer su implementación interna.

En Kotlin, podemos implementar este patrón utilizando la interfaz `Iterator<T>` y la clase `Iterable<T>`.

Estas interfaces nos permiten crear clases que pueden ser recorridas utilizando un ciclo `for-in`, lo que nos ahorra tener que escribir código para recorrer la colección manualmente.

Mediator

El patrón de diseño Mediator es un patrón de diseño que se utiliza para controlar la comunicación entre diferentes objetos.

Esto permite que los objetos se comuniquen de manera eficiente sin tener que depender directamente uno del otro.

Un ejemplo de un caso en el que se puede utilizar el patrón Mediator es en un sistema de chat en línea.

Los usuarios del chat pueden enviar y recibir mensajes a través del mediador, que controla la comunicación entre ellos.

De esta manera, los usuarios no tienen que conocerse directamente y pueden enviarse mensajes de manera eficiente. En Kotlin, podemos implementar el patrón Mediator utilizando una interfaz y clases concretas para representar a los usuarios del chat y el mediador.

En este ejemplo, la interfaz `User` representa a los usuarios del chat, que pueden enviar y recibir mensajes.

```
interface User {
```

```
    fun sendMessage(message: String)

    fun receiveMessage(message: String)

}
```

La clase `ConcreteUser` implementa esta interfaz y mantiene una referencia al mediador para enviar mensajes a través de él.

```
class ConcreteUser(val mediator: Mediator, val name: String): User
{

    override fun sendMessage(message: String) {

        mediator.sendMessage(message, this)

    }

    override fun receiveMessage(message: String) {

        println("$name recibió: $message")

    }

}
```

La clase `Mediator`, por su parte, mantiene una lista de usuarios registrados y controla la comunicación entre ellos.

```
class Mediator {

    private val users = mutableListOf<User>()

    fun registerUser(user: User) {
```

```
        users.add(user)
    }

    fun sendMessage(message: String, user: User) {
        users.forEach {
            if (it != user) {
                it.receiveMessage(message)
            }
        }
    }
}
```

Para utilizar este código, podemos crear una instancia del mediador y registrar a los usuarios en él:

```
val mediator = Mediator()

val user1 = ConcreteUser(mediator, "User 1")

val user2 = ConcreteUser(mediator, "User 2")

mediator.registerUser(user1)

mediator.registerUser(user2)

user1.sendMessage("Hola, ¿cómo estás?")

user2.sendMessage("Bien, ¿y tú?")
```

En este caso, el mediador controla la comunicación entre los usuarios y se encarga de enviar los mensajes a los destinatarios adecuados.

De esta manera, se asegura que cada usuario reciba solo los mensajes que le envían, sin tener que conocer directamente a los demás usuarios del chat.

El patrón Mediator tiene varias ventajas en comparación con otros patrones de diseño de comunicación.

- Una de ellas es que permite controlar y centralizar la comunicación entre objetos, lo que facilita la organización y mantenimiento del código.
- También permite que los objetos se comuniquen de manera eficiente, sin tener que establecer relaciones directas entre ellos.

En resumen, el patrón Mediator es una herramienta útil en situaciones en las que se necesita controlar y centralizar la comunicación entre objetos.

En Kotlin, se puede implementar utilizando una interfaz y clases concretas para representar a los objetos que se comunican y al mediador que controla esta comunicación.

Memento

El patrón de diseño Memento es un patrón de comportamiento que se utiliza para permitir a un objeto volver a un estado anterior sin violar el principio de encapsulamiento.

Este patrón se utiliza a menudo en aplicaciones donde es necesario revertir un cambio o restaurar un objeto a un estado anterior.

Por ejemplo, en un editor de texto, podemos utilizar el patrón Memento para deshacer y rehacer cambios en un documento.

En Kotlin, podemos implementar el patrón Memento mediante la creación de una clase `Memento` que almacena el estado de un objeto y una clase `Caretaker` que se encarga de crear y restaurar el estado del objeto mediante la clase `Memento`.

Ejemplo de Memento: Editor de texto

Por ejemplo, supongamos que tenemos una clase `Editor` que representa un editor de texto y tiene una propiedad de texto que almacena el contenido del documento.

Podemos implementar el patrón Memento de la siguiente manera:

```
class Memento(val text: String)

class Caretaker {
```

```
    private val mementos = mutableListOf<Memento>()

    fun save(editor: Editor) {

        mementos.add(Memento(editor.text))

    }

    fun restore(editor: Editor) {

        editor.text = mementos.last().text

    }

}

class Editor {

    var text = ""

    val caretaker = Caretaker()

    fun write(text: String) {

        this.text = text

    }

    fun save() {

        caretaker.save(this)

    }

    fun restore() {

        caretaker.restore(this)
```

```
    }
}
```

En este ejemplo, la clase `Memento` simplemente almacena el estado de un objeto `Editor` mediante la propiedad de texto.

La clase `Caretaker` se encarga de crear una instancia de `Memento` y almacenarla en una lista cada vez que se llama al método `save()` del objeto `Editor`.

Para restaurar el estado anterior, el método `restore()` de la clase `Caretaker` obtiene el último `Memento` de la lista y establece la propiedad de texto del objeto `Editor` en el valor almacenado en el `Memento`.

Con esta implementación, podemos utilizar el patrón Memento en nuestra aplicación de la siguiente manera:

```
val editor = Editor()

editor.write("Texto inicial")

editor.save()

editor.write("Cambio 1")

editor.save()

editor.write("Cambio 2")

println(editor.text) // "Cambio 2"

editor.restore()
```

```
println(editor.text) // "Cambio 1"
```

Ejemplo de Memento: Estado de un juego

Además de su uso en editores de texto, el patrón Memento también se puede utilizar en aplicaciones de juegos para guardar y restaurar el estado de un juego.

Por ejemplo, podemos utilizar el patrón Memento para guardar y restaurar el estado de un juego de ajedrez.

En Kotlin, podemos implementar el patrón Memento en un juego de ajedrez de la siguiente manera:

```
class Memento(val board: List<List<Piece>>)

class Caretaker {

    private val mementos = mutableListOf<Memento>()

    fun save(game: ChessGame) {

        mementos.add(Memento(game.board))

    }

    fun restore(game: ChessGame) {

        game.board = mementos.last().board

    }

}

class ChessGame {
```

```
    var board = List(8) { List(8) { EmptyPiece } }

    val caretaker = Caretaker()

    fun makeMove(from: Pair<Int, Int>, to: Pair<Int, Int>) {

        // Mover pieza en el tablero

    }

    fun save() {

        caretaker.save(this)

    }

    fun restore() {

        caretaker.restore(this)

    }

}
```

En este ejemplo, la clase `Memento` almacena el estado del tablero del juego de ajedrez mediante la propiedad `board`.

La clase `Caretaker` se encarga de crear una instancia de `Memento` y almacenarla en una lista cada vez que se llama al método `save()` del objeto `ChessGame`.

Para restaurar el estado anterior, el método `restore()` de la clase `Caretaker` obtiene el último `Memento` de la lista y

establece la propiedad `board` del objeto `ChessGame` en el valor almacenado en el `Memento`.

De esta manera, podemos utilizar el patrón Memento en nuestro juego de ajedrez para guardar y restaurar el estado del juego en cualquier momento.

Conclusión

En conclusión, el patrón de diseño Memento es un patrón de diseño de comportamiento que se utiliza para permitir a un objeto volver a un estado anterior sin violar el principio de encapsulamiento.

En Kotlin, podemos implementar este patrón mediante la creación de una clase `Memento` que almacena el estado de un objeto y una clase `Caretaker` que se encarga de crear y restaurar el estado del objeto a través de la clase `Memento`.

Este patrón es útil en aplicaciones donde es necesario revertir cambios o restaurar un objeto a un estado anterior.

Observer

El patrón de diseño Observer es uno de los patrones más populares y utilizados en la programación orientada a objetos.

Este patrón permite que un objeto, conocido como «sujeto», notifique a sus «observadores» cuando sucede un cambio en su estado.

Por ejemplo, imaginemos que tenemos una aplicación de mensajería que notifica a los usuarios cuando reciben un nuevo mensaje.

En este caso, el sujeto sería el mensaje y los observadores serían los usuarios que están suscritos a ese mensaje.

Cuando se recibe un nuevo mensaje, el sujeto notifica a todos sus observadores y éstos pueden actualizar su interfaz de usuario para mostrar el nuevo mensaje.

En Kotlin, podemos implementar el patrón Observer utilizando interfaces y clases genéricas.

Por ejemplo, podríamos tener una interfaz llamada `Observable` que define los métodos necesarios para agregar y eliminar observadores, así como para notificar a los observadores cuando se produce un cambio en el estado del sujeto.

```
interface Observable<T> {

    fun addObserver(observer: Observer<T>)

    fun removeObserver(observer: Observer<T>)

    fun notifyObservers()

```

```
}
```

También tendríamos una interfaz llamada `Observer` que define un método para actualizar el estado del observador cuando se produce un cambio en el sujeto.

```
interface Observer<T> {

    fun update(data: T)

}
```

Luego, podríamos implementar nuestro sujeto, por ejemplo una clase `Message` que implementa la interfaz `Observable` y mantiene una lista de observadores.

```
class Message(var content: String) : Observable<Message> {

    private val observers = mutableListOf<Observer<Message>>()

    fun setContent(content: String) {

        this.content = content

        notifyObservers()

    }

    override fun addObserver(observer: Observer<Message>) {

        observers.add(observer)

    }

    override fun removeObserver(observer: Observer<Message>) {
```

```
        observers.remove(observer)

    }

    override fun notifyObservers() {

        observers.forEach { it.update(this) }

    }

}
```

Por último, podríamos implementar nuestros observadores, por ejemplo una clase `User` que implementa la interfaz `Observer` y muestra el nuevo mensaje cuando se produce un cambio en el sujeto.

```
class User(var name: String) : Observer<Message> {

    override fun update(data: Message) {

        println("Nuevo mensaje para $name: ${data.content}")

    }

}
```

Para utilizar estas clases, podríamos crear un mensaje y agregar usuarios como observadores.

Cuando se modifique el contenido del mensaje, se notificará a los observadores y éstos podrán mostrar el nuevo contenido en su interfaz de usuario.

```
val message = Message("Hola mundo")
```

```
val user1 = User("Juan")

val user2 = User("Ana")

message.addObserver(user1)

message.addObserver(user2)

message.setContent("¿Cómo estás?")
```

En este ejemplo, se imprimiría en la consola:

```
Nuevo mensaje para Juan: ¿Cómo estás?

Nuevo mensaje para Ana: ¿Cómo estás?
```

De esta manera, el patrón Observer nos permite notificar a múltiples observadores cuando se produce un cambio en el estado de un sujeto, permitiendo una comunicación eficiente entre objetos en nuestro código.

Conclusión

En resumen, el patrón Observer es una técnica muy útil en la programación orientada a objetos que permite que un objeto notifique a sus observadores cuando se produce un cambio en su estado.

Esto nos permite implementar una comunicación eficiente entre objetos en nuestro código, lo que nos permite diseñar sistemas más flexibles y adaptables.

En Kotlin, podemos implementar el patrón Observer utilizando interfaces y clases genéricas.

Por ejemplo, podemos tener una interfaz `Observable` que define los métodos necesarios para agregar y eliminar observadores, así como para notificar a los observadores cuando se produce un cambio en el estado del sujeto.

También podemos tener una interfaz `Observer` que define un método para actualizar el estado del observador cuando se produce un cambio en el sujeto.

Luego, podemos implementar nuestro sujeto y nuestros observadores utilizando estas interfaces.

Este patrón es muy útil en aplicaciones que requieren notificaciones en tiempo real, como por ejemplo aplicaciones de mensajería, redes sociales o sistemas de alertas. Además, es un patrón muy utilizado en la programación concurrente ya que permite notificar a los observadores de manera segura y eficiente.

En conclusión, el patrón Observer es una herramienta muy útil en la programación orientada a objetos y en Kotlin podemos implementarlo de manera sencilla y eficiente utilizando interfaces y clases genéricas.

State

El patrón de diseño State es un patrón de comportamiento que nos permite cambiar el comportamiento de un objeto en función del estado en el que se encuentre.

Este patrón nos permite desacoplar el comportamiento de un objeto de su implementación, lo que nos permite cambiar el comportamiento de un objeto en tiempo de ejecución de forma sencilla y mantenible.

Ejemplo de State: Ascensor

Para ilustrar el uso de este patrón en Kotlin, consideremos un ejemplo de una máquina de estados finitos que simula un ascensor.

En este caso, el ascensor puede encontrarse en uno de tres estados: *Subiendo*, *Bajando* y *Detenido*.

Cada uno de estos estados tiene un comportamiento diferente en cuanto a la acción que debe realizar cuando se presiona un botón de piso.

Para implementar el patrón de diseño State en Kotlin, podemos utilizar una interfaz que represente el estado del ascensor y una clase que represente la máquina de estados finitos del ascensor.

La interfaz del estado del ascensor tendría un método para manejar la acción de presionar un botón de piso, y cada una de las clases que representan los estados del ascensor implementaría este método de forma diferente.

A continuación se muestra un ejemplo de cómo se podría implementar el patrón de diseño State en Kotlin para el caso del ascensor.

La interfaz `ElevatorState` representa un estado del ascensor y tiene un método `handleFloorButtonPress()` que se encarga de manejar la acción de presionar un botón de piso:

```
interface ElevatorState {

    fun handleFloorButtonPress(floor: Int)

}
```

La clase `Elevator` es la máquina de estados finitos del ascensor y su estado inicial es `StoppedState`.

```
class Elevator {

    var state: ElevatorState = StoppedState()

    fun pressFloorButton(floor: Int) {

        state.handleFloorButtonPress(floor)

    }

}
```

Este sería un ejemplo de implementación de los estados:

```
class StoppedState: ElevatorState {

    override fun handleFloorButtonPress(floor: Int) {

        if (floor > Elevator.currentFloor) {

            Elevator.state = GoingUpState()

        } else if (floor < Elevator.currentFloor) {

            Elevator.state = GoingDownState()

        } else {

            // Do nothing, already on the selected floor

        }

    }

}

class GoingUpState: ElevatorState {

    override fun handleFloorButtonPress(floor: Int) {

        if (floor > Elevator.currentFloor) {

            // Add floor to list of floors to visit

        } else if (floor < Elevator.currentFloor) {

            // Change direction and add floor to list of floors to
visit
```

```
        } else {
            // Stop elevator on selected floor
            Elevator.state = StoppedState()
        }
    }
}
class GoingDownState: ElevatorState {
    override fun handleFloorButtonPress(floor: Int) {
        if (floor > Elevator.currentFloor) {
            // Change direction and add floor to list of floors to
visit
        } else if (floor < Elevator.currentFloor) {
            // Add floor to list of floors to visit
        } else {
            // Stop elevator on selected floor
            Elevator.state = StoppedState()
        }
    }
}
```

Cuando se presiona un botón de piso, se llama al método `pressFloorButton()` de la clase `Elevator`, que a su vez llama al método `handleFloorButtonPress()` del estado actual del ascensor.

Cada una de las clases que representan los estados del ascensor implementa este método de forma diferente, de modo que el comportamiento del ascensor cambia en función del estado en el que se encuentre.

Por ejemplo, si el ascensor se encuentra en el estado `StoppedState` y se presiona un botón de piso en un piso superior al actual, el ascensor cambiará su estado a `GoingUpState` y empezará a subir.

Si en cambio se presiona un botón de piso en un piso inferior al actual, el ascensor cambiará su estado a `GoingDownState` y empezará a bajar.
Si se presiona un botón de piso en el mismo piso en el que se encuentra el ascensor, no se producirá ningún cambio de estado.

Conclusión

El patrón de diseño State nos permite cambiar el comportamiento de un objeto en función de su estado.

En Kotlin, podemos implementar este patrón mediante el uso de una interfaz que represente los estados del objeto y una clase que represente la máquina de estados finitos del objeto.

Cada una de las clases que representan los estados del objeto implementan el método correspondiente al comportamiento que se desea en cada estado, lo que nos permite cambiar el comportamiento del objeto en tiempo de ejecución de forma sencilla y mantenible.

Strategy

El patrón de diseño Strategy es un patrón de diseño de software que permite a una clase cambiar su comportamiento en tiempo de ejecución.

Esto se logra mediante la creación de diferentes estrategias o algoritmos que pueden ser intercambiados fácilmente.
En Kotlin, podemos implementar el patrón Strategy mediante la creación de una interfaz que define el comportamiento deseado y luego crear diferentes clases que implementen esa interfaz de manera diferente.

Ejemplo de patrón Strategy

Por ejemplo, supongamos que queremos crear una aplicación de estrategia militar en la que se puedan utilizar diferentes estrategias de ataque.

Podríamos crear una interfaz llamada `AttackStrategy` que define el comportamiento de ataque, y luego crear diferentes clases que implementen esa interfaz de manera diferente, como `AirAttack` o `GroundAttack`.

Para hacer uso de estas diferentes estrategias de ataque, podríamos crear una clase llamada `MilitaryUnit` que tenga una propiedad `attackStrategy` que sea una instancia de la interfaz `AttackStrategy`.

Entonces, podríamos cambiar la estrategia de ataque de una unidad militar en tiempo de ejecución simplemente asignando una nueva estrategia a la propiedad `attackStrategy`.

A continuación, se muestra un ejemplo de cómo implementar el patrón Strategy en Kotlin:

```
// Interfaz que define el comportamiento de ataque

interface AttackStrategy {

    fun attack()

}

// Clase que implementa la estrategia de ataque aéreo

class AirAttack: AttackStrategy {

    override fun attack() {

        println("Atacando desde el aire")

    }
```

```
}

// Clase que implementa la estrategia de ataque terrestre
class GroundAttack: AttackStrategy {
    override fun attack() {
        println("Atacando desde el suelo")
    }
}

// Clase que representa una unidad militar
class MilitaryUnit(var attackStrategy: AttackStrategy) {
    fun attack() {
        attackStrategy.attack()
    }
}

// Creación de una unidad militar con una estrategia de ataque aéreo
val militaryUnit = MilitaryUnit(AirAttack())
militaryUnit.attack() // Imprime "Atacando desde el aire"

// Cambio de la estrategia de ataque a una estrategia terrestre
militaryUnit.attackStrategy = GroundAttack()
```

```
militaryUnit.attack() // Imprime "Atacando desde el suelo"
```

Conclusión

En resumen, el patrón de diseño Strategy nos permite cambiar el comportamiento de una clase en tiempo de ejecución mediante la creación de diferentes estrategias o algoritmos que pueden ser intercambiados fácilmente.

Esto nos permite tener un código más flexible y adaptable a diferentes situaciones.

En Kotlin, podemos implementar el patrón Strategy mediante la creación de una interfaz que define el comportamiento deseado y luego crear diferentes clases que implementen esa interfaz de manera diferente.

Template Method

El patrón de diseño Template Method es uno de los patrones de diseño más utilizados en la programación orientada a objetos.

Este patrón se basa en la idea de definir una plantilla de un algoritmo y dejar que las subclases implementen los pasos específicos del mismo.

Para entender mejor el patrón Template Method, consideremos un ejemplo en el que queremos implementar una clase que represente una receta de cocina.

En este caso, la clase base sería la clase `Recipe`, que tendría un método template llamado `cook()`, que contiene el algoritmo general de la receta.

En el método `cook()`, primero se pondría el agua a hervir, luego se añadirían los ingredientes y se cocinaría a fuego lento. Sin embargo, la elección de los ingredientes y la duración de la cocción dependen de la receta específica que se esté preparando.

Por lo tanto, en la clase base `Recipe` se dejaría un espacio para que las subclases implementen estos pasos específicos.

A continuación, se presenta un ejemplo de cómo se podría implementar esta clase en Kotlin:

```
abstract class Recipe {

    fun cook() {

        boilWater()

        addIngredients()

        cookSlowly()

    }

    abstract fun addIngredients()

    abstract fun cookSlowly()
```

```
    fun boilWater() {
        println("Poniendo agua a hervir...")
    }
}

class PastaRecipe : Recipe() {
    override fun addIngredients() {
        println("Añadiendo pasta y salsa al agua hirviendo...")
    }
    override fun cookSlowly() {
        println("Cocinando a fuego lento durante 12 minutos...")
    }
}

class SoupRecipe : Recipe() {
    override fun addIngredients() {
        println("Añadiendo verduras y carne al agua hirviendo...")
    }
    override fun cookSlowly() {
        println("Cocinando a fuego lento durante 30 minutos...")
    }
```

```
}
```

En este ejemplo, la clase `Recipe` es la clase base que contiene el método `cook()` como plantilla del algoritmo de la receta.

La clase `PastaRecipe` y la clase `SoupRecipe` son subclases que implementan los pasos específicos de cada receta.

Al utilizar el patrón Template Method, podemos asegurarnos de que todas las recetas sigan el mismo algoritmo general, pero que cada una pueda personalizarse según sus ingredientes y duración de la cocción.

Además, el patrón Template Method también nos permite reutilizar código y evitar la repetición de código en las subclases.

En el ejemplo anterior, la clase base Recipe tiene un método «boilWater» que es utilizado por todas las subclases.

De esta manera, no es necesario que cada subclase implemente el mismo método para hervir agua.

En resumen, el patrón Template Method es una herramienta útil en la programación orientada a objetos para definir un algoritmo general y permitir que las subclases implementen los pasos específicos.

Este patrón nos permite reutilizar código y evitar la repetición de código en las subclases.

En Kotlin, podemos implementar el patrón Template Method utilizando la palabra clave `abstract` y las funciones abstractas.

Visitor

El patrón de diseño Visitor es un patrón de diseño de software que permite agregar comportamientos nuevos a una estructura de objetos sin modificar las clases de los objetos individuales.

Esto se logra mediante la creación de una clase `Visitor` que contiene los nuevos comportamientos y la implementación de una interfaz en cada clase de la estructura de objetos que permita la visita del visitor.

En Kotlin, podemos implementar el patrón de diseño Visitor utilizando una clase sealed y las interfaces de visita.

Por ejemplo, supongamos que tenemos una estructura de objetos que representa una factura con diferentes elementos, como productos, impuestos y descuentos.

Podemos crear una clase sealed llamada `Element` que represente cada uno de estos elementos y una interfaz llamada

`Visitor` que contenga los métodos de visita para cada uno de ellos:

```
sealed class Element {

    abstract fun accept(visitor: Visitor)

}

interface Visitor {

    fun visit(product: Product)

    fun visit(tax: Tax)

    fun visit(discount: Discount)

}
```

Luego, podemos crear las clases concretas que implementen la clase sealed `Element` y la interfaz `Visitor` para cada uno de los elementos de la factura:

```
class Product(val name: String, val price: Double) : Element() {

    override fun accept(visitor: Visitor) {

        visitor.visit(this)

    }

}

class Tax(val rate: Double) : Element() {

    override fun accept(visitor: Visitor) {
```

```
        visitor.visit(this)

    }
}

class Discount(val amount: Double) : Element() {

    override fun accept(visitor: Visitor) {

        visitor.visit(this)

    }
}
```

Por último, podemos crear una clase visitor concreta que implemente la interfaz «Visitor» y contenga los comportamientos deseados para cada uno de los elementos de la factura:

```
class InvoiceVisitor : Visitor {

    var total = 0.0

    override fun visit(product: Product) {

        total += product.price

    }

    override fun visit(tax: Tax) {

        total *= (1 + tax.rate)

    }
```

```
    override fun visit(discount: Discount) {

        total -= discount.amount

    }
}
```

De esta manera, podemos utilizar la clase `InvoiceVisitor` para calcular el total de la factura sin tener que modificar las clases de los elementos individuales.

Por ejemplo:

```
val elements = listOf(
    Product("Laptop", 1000.0),
    Tax(0.15),
    Discount(100.0)
)

val visitor = InvoiceVisitor()

for (element in elements) {
    element.accept(visitor)
}
```

```
println("Total: ${visitor.total}") // Total: 915.0
```

En este ejemplo, la clase `InvoiceVisitor` recorre la lista de elementos y visita cada uno de ellos, aplicando los comportamientos deseados para calcular el total de la factura.

De esta manera, podemos agregar nuevos comportamientos a la estructura de objetos de la factura sin tener que modificar las clases de los elementos individuales.

Conclusión

En resumen, el patrón de diseño Visitor es una forma de agregar nuevos comportamientos a una estructura de objetos sin modificar las clases de los objetos individuales.

Esto se logra mediante la creación de una clase visitor que contiene los nuevos comportamientos y la implementación de una interfaz en cada clase de la estructura de objetos que permita la visita del visitor.

En Kotlin, podemos utilizar una clase sealed y las interfaces de visita para implementar el patrón de diseño Visitor de manera sencilla y eficiente.

¡Espero que te haya gustado!

Siempre puedes enviarme tu feedback a contacto@devexperto.com y estaré encantado de leerlo.

Recuerda que si estás buscando aplicar tus conocimientos, tengo las siguientes opciones:

- Aprender Kotlin y su ecosistema en Kotlin Expert
- Dominar Jetpack Compose para Android en Compose Expert
- El programa Architect Coders para llevar tu calidad de software al siguiente nivel (¡incluye un training gratuito de 2 semanas!)

Muchas gracias, y seguimos en contacto por:

- YouTube
- Twitter
- Instagram
- LinkedIn

Y si has llegado a este documento por otra vía, puedes suscribirte aquí y llevarte de regalo la guía para iniciarte en Kotlin.

¡Un abrazo!

Antonio Leiva