PATRONES DE DISEÑO EN KOTLIN

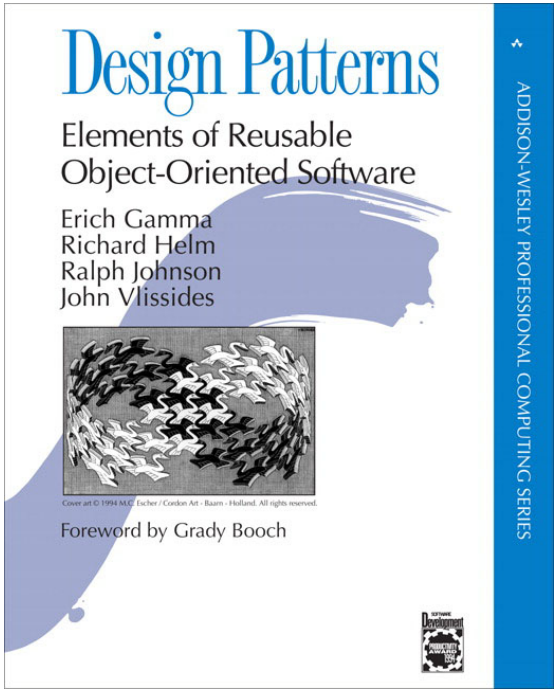
JULIO SALDAÑA
ANDROID DEVELOPER, AVANTICA

PATRONES DE DISEÑO

Son soluciones a problemas comunes en un determinado contexto.

Podemos pensar en un patrón como una plantilla para construir nuestras propias soluciones.

LA FUENTE


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

PRINCIPIOS

Programar interfaces y no implementaciones

Uso de composición en lugar de herencia.

PATRONES CREACIONALES

SINGLETON

Asegura que una clase es instanciada solo una vez

Provee un punto de acceso global a esta instancia.

```
object AnalyticsManager {

    fun send(screen:String, data:Data) {
            …
            //enviar datos a un servidor
    }
}

// No necesitamos crear un nuevo objeto
AnalyticsManager.send(MAIN_SCREEN, data)
```

```
object AnalyticsManager {

    fun send(screen:String, data:Data) {
        …
        //enviar datos a un servidor
    }
}

// No necesitamos crear un nuevo objeto
AnalyticsManager.send(MAIN_SCREEN, data)
```

```
object AnalyticsManager {

    fun send(screen:String, data:Data) {
        …
        //enviar datos a un servidor
    }
}

// No necesitamos crear un nuevo objeto
AnalyticsManager.send(MAIN_SCREEN, data)
```

```
object AnalyticsManager {

    fun send(screen:String, data:Data) {
        ...
        //enviar datos a un servidor
    }
}

// No necesitamos crear un nuevo objeto
AnalyticsManager.send(MAIN_SCREEN, data)
```

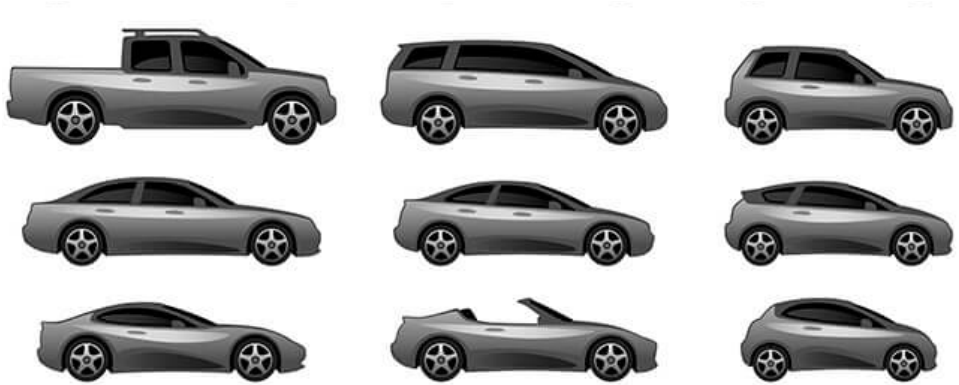
Cuando solo tienes un martillo …

BUILDER

Ayuda a construir objetos complejos de una manera sencilla.

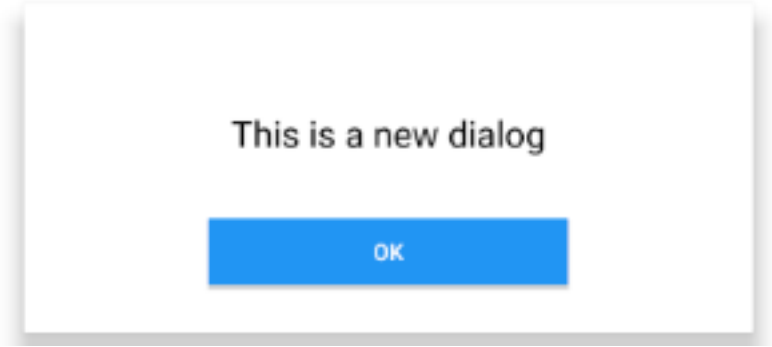
Permite producir diferentes tipos de objetos usando el mismo código.

Evita crear multiples constructores



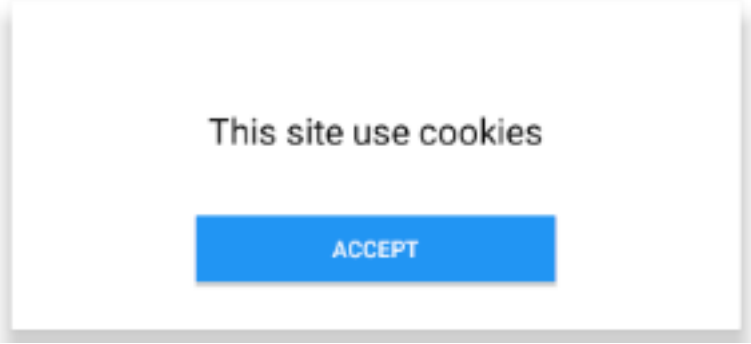
```
data class Dialog(
    val message: String,
    val positiveButtonText: String = "OK",
    val negativeButtonText: String? = null)
```

```
Dialog("This is a new dialog")
```



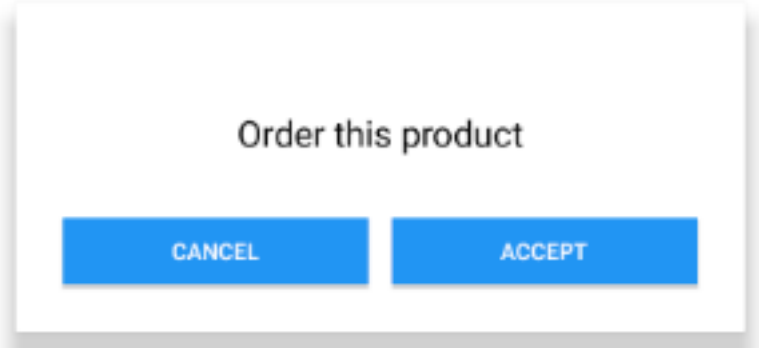
```
data class Dialog(
    val message: String,
    val positiveButtonText: String = "OK",
    val negativeButtonText: String? = null)
```

```
Dialog(positiveButtonText = "Accept", message = "This site use cookies")
```



```
data class Dialog(
   val message: String,
   val positiveButtonText: String = "OK",
   val negativeButtonText: String? = null)
```

```
Dialog(positiveButtonText = “Accept", negativeButtonText = "Cancel",
    message = "Order this product")
```

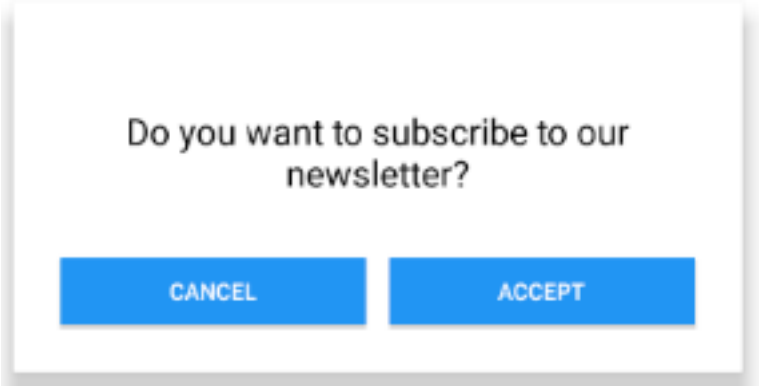


PROTOTYPE

Permite crear copias de objetos existentes.

Evita que llamemos procesos costosos multiples veces.

```
val orderDialog = Dialog( message = "Order this product”,
                          positiveButtonText = "Accept",
                          negativeButtonText = "Cancel" )

val subscriptionDialog  =  orderDialog.copy(“message = Do you want to subscribe to our newsletter?")
```



PATRONES ESTRUCTURALES

ADAPTER

Permite que objetos con interfaces incompatibles puedan colaborar.

```
class EuropeanSocket( plug: EuropeanPlug )
class EuropeanPlug
class UsPlug


class Adapter( plug: UsPlug ) {
   fun toEuropeanPlug( ): EuropeanPlug {
      // Lógica para convertir tipo
   }
}


val europeanPlug = Adapter( UsPlug( ) ).toEuropeanPlug( )
EuropeanSocket( europeanPlug )
```

```
class EuropeanSocket( plug: EuropeanPlug )
class EuropeanPlug
class UsPlug
```

```
class Adapter( plug: UsPlug ) {
   fun toEuropeanPlug( ): EuropeanPlug {
      // Lógica para convertir tipo
   }
}
```

```
val europeanPlug = Adapter( UsPlug( ) ).toEuropeanPlug( )
EuropeanSocket( europeanPlug )
```

```
class EuropeanSocket( plug: EuropeanPlug )
class EuropeanPlug
class UsPlug
```

```
class Adapter( plug: UsPlug ) {
  fun toEuropeanPlug( ): EuropeanPlug {
    // Lógica para convertir tipo
  }
}
```

```
val europeanPlug = Adapter( UsPlug( ) ).toEuropeanPlug( )
EuropeanSocket( europeanPlug )
```

```
class EuropeanSocket( plug: EuropeanPlug )
class EuropeanPlug
class UsPlug


class Adapter( plug: UsPlug ) {
   fun toEuropeanPlug( ): EuropeanPlug {
      // Lógica para convertir tipo
   }
}


val europeanPlug = Adapter( UsPlug( ) ).toEuropeanPlug( )
EuropeanSocket( europeanPlug )
```

```
class EuropeanSocket( plug : EuropeanPlug )
class EuropeanPlug
class UsPlug

fun UsPlug.toEuropeanPlug( ) : EuropeanPlug {
  // Lógica para convertir tipo
}


EuropeanSocket( UsPlug( ).toEuropeanPlug( ) )
```

```
"1.01".toBigDecimal()

 "1".toInt()

 100.toString()

 BigDecimal.ZERO.toString()


 val abc = listOf(“A", “B", "C")
 abc.toMutableList().add("D")
```

DECORATOR

Permite adicionar nuevos comportamientos a objetos.

Evita la creación de multiples subclases.

```
abstract class TextProcessor( protected open val textProcessor: TextProcessor? = null ) {
   abstract fun process( source:String ) : String
}

//input " hello   world "  output "hello_world"
class DashProcessor( override val textProcessor: TextProcessor? = null ) : TextProcessor( ) {
   override fun process( source: String ) : String {
     return ( textProcessor?.process(source) ?: source).trim( ).replace(Regex(" +"), "_" )
   }
}

class LowerCaseProcessor( override val textProcessor: TextProcessor? = null): TextProcessor( ) {
   override fun process( source: String ) : String {
      return (textProcessor?.process(source) ?: source).toLowerCase( )
   }
}
```

```
abstract class TextProcessor( protected open val textProcessor: TextProcessor? = null ) {
   abstract fun process( source:String ) : String
}

//input " hello   world "  output "hello_world"
class DashProcessor( override val textProcessor: TextProcessor? = null ) : TextProcessor( ) {
   override fun process( source: String ) : String {
     return ( textProcessor?.process(source) ?: source).trim( ).replace(Regex(" +"), "_" )
   }
}

class LowerCaseProcessor( override val textProcessor: TextProcessor? = null): TextProcessor( ) {
   override fun process( source: String ) : String {
      return (textProcessor?.process(source) ?: source).toLowerCase( )
   }
}
```

```
abstract class TextProcessor( protected open val textProcessor: TextProcessor? = null ) {
   abstract fun process( source:String ) : String
}

// input " hello   world "  output "hello_world"
class DashProcessor( override val textProcessor: TextProcessor? = null ) : TextProcessor( ) {
   override fun process( source: String ) : String {
     return ( textProcessor?.process(source) ?: source).trim( ).replace(Regex(" +"), "_" )
   }
}


class LowerCaseProcessor( override val textProcessor: TextProcessor? = null): TextProcessor( ) {
   override fun process( source: String ) : String {
      return (textProcessor?.process(source) ?: source).toLowerCase( )
   }
}
```

```
abstract class TextProcessor( protected open val textProcessor: TextProcessor? = null ) {
  abstract fun process( source:String ) : String
}

// input " hello   world "  output "hello_world"
class DashProcessor( override val textProcessor: TextProcessor? = null ) : TextProcessor( ) {
  override fun process( source: String ) : String {
    return ( textProcessor?.process(source) ?: source).trim( ).replace(Regex(" +"), "_" )
  }
}


class LowerCaseProcessor( override val textProcessor: TextProcessor? = null): TextProcessor( ) {
  override fun process( source: String ) : String {
    return (textProcessor?.process(source) ?: source).toLowerCase( )
  }
}
```

```
val snakeCaseProcessor = DashProcessor( LowerCaseProcessor( ) )


// returns hello_world
snakeCaseProcessor.process("Hello   World")
```

PATRONES DE COMPORTAMIENTO

CHAIN OF RESPONSIBILITY

Permite pasar una solicitud a través de una cadena de handlers.

Cada handler decide si procesa la respuesta o la pasa al siguiente handler.

```
typealias Handler = (password: String) -> Boolean

val validateLength = fun(next: Handler) : Handler {
   return fun(password:String): Boolean {
      return password.length in 6..12 && next(password)
   }
}

val validateFormat = fun(next: Handler) : Handler {
   return fun(password:String): Boolean{
      return PASSWORD_REGEX.matches(password) && next(password)
   }
}

val approve = fun() = fun( password:String ) = true
```

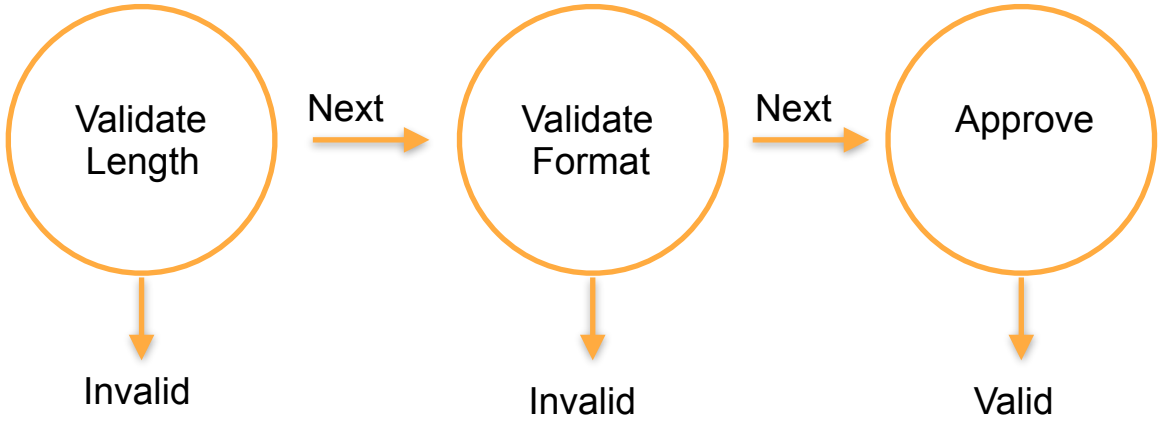
```
typealias Handler = (password: String) -> Boolean

val validateLength = fun(next: Handler) : Handler {
   return fun(password:String): Boolean {
      return password.length in 6..12 && next(password)
   }
}

val validateFormat = fun(next: Handler) : Handler {
   return fun(password:String): Boolean{
      return PASSWORD_REGEX.matches(password) && next(password)
   }
}

val approve = fun() = fun( password:String ) = true
```

```
typealias Handler = (password: String) -> Boolean

val validateLength = fun(next: Handler) : Handler {
   return fun(password:String): Boolean {
      return password.length in 6..12 && next(password)
   }
}

val validateFormat = fun(next: Handler) : Handler {
   return fun(password:String): Boolean{
      return PASSWORD_REGEX.matches(password) && next(password)
   }
}

val approve = fun() = fun( password:String ) = true
```

```
typealias Handler = (password: String) -> Boolean

val validateLength = fun(next: Handler) : Handler {
    return fun(password:String): Boolean {
        return password.length in 6..12 && next(password)
    }
}

val validateFormat = fun(next: Handler) : Handler {
    return fun(password:String): Boolean{
        return PASSWORD_REGEX.matches(password) && next(password)
    }
}

val approve = fun() = fun( password:String ) = true
```

```
typealias Handler = (password: String) -> Boolean

val validateLength = fun(next: Handler) : Handler {
   return fun(password:String): Boolean {
       return password.length in 6..12 && next(password)
   }
}

val validateFormat = fun(next: Handler) : Handler {
   return fun(password:String): Boolean{
       return  PASSWORD_REGEX.matches(password) && next(password)
   }
}

val approve = fun() = fun( password:String ) = true
```

```
typealias Handler = (password: String) -> Boolean

val validateLength = fun(next: Handler) : Handler {
   return fun(password:String): Boolean {
      return password.length in 6..12 && next(password)
   }
}

val validateFormat = fun(next: Handler) : Handler {
   return fun(password:String): Boolean{
      return PASSWORD_REGEX.matches(password) && next(password)
   }
}

val approve = fun() = fun( password:String ) = true
```



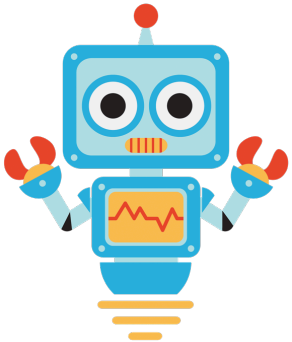

```
val validationChain =
validateLength(validateFormat( approve( ) ))

val isValid = validationChain("P@ssw0rd")
```

COMMAND

Un objeto puede contener toda la información de un request.

Permite retrasar o encolar una solicitud.

Puede soportar el deshacer operaciones.



```
class Robot {

  private val commands = mutableListOf<Command>()

  fun rotateLeft( ) = apply {
    commands.add { ... }
  }

  fun rotateRight( ) ...

  fun moveForward( ) ...

  fun moveBackwards( ) ...

  fun execute() {
    while (!commands.isEmpty( )) {
      val command = commands.removeAt(0)
      command.execute( )
    }
  }
}
```

```
val robot = Robot( )

robot.rotateRight( )
  .moveForward( )
  .moveForward()
  .rotateLeft( )
  .moveForward( )
  .moveForward( )
  .rotateLeft( )
  .moveBackwards( )
  .execute( )
```

```
class Robot {

  private val commands = mutableListOf<Command>()

  fun rotateLeft( ) = apply {
    commands.add { ... }
  }

  fun rotateRight( ) ...

  fun moveForward( ) ...

  fun moveBackwards( ) ...

  fun execute() {
    while (!commands.isEmpty( )) {
      val command = commands.removeAt(0)
      command.execute( )
    }
  }
}
```

```
val robot = Robot( )

robot.rotateRight( )
  .moveForward( )
  .moveForward()
  .rotateLeft( )
  .moveForward( )
  .moveForward( )
  .rotateLeft( )
  .moveBackwards( )
  .execute( )
```

```
class Robot {

    private val commands = mutableListOf<Command>()

    fun rotateLeft( ) = apply {
        commands.add { ... }
    }

    fun rotateRight( ) ...

    fun moveForward( ) ...

    fun moveBackwards( ) ...

    fun execute() {
        while (!commands.isEmpty( )) {
            val command = commands.removeAt(0)
            command.execute( )
        }
    }
}
```

```
val robot = Robot( )

robot.rotateRight( )
    .moveForward( )
    .moveForward()
    .rotateLeft( )
    .moveForward( )
    .moveForward( )
    .rotateLeft( )
    .moveBackwards( )
    .execute( )
```

```
class Robot {

  private val commands = mutableListOf<Command>()

  fun rotateLeft( ) = apply {
     commands.add { ... }
  }

  fun rotateRight( ) ...

  fun moveForward( ) ...

  fun moveBackwards( ) ...

  fun execute() {
     while (!commands.isEmpty( )) {
        val command = commands.removeAt(0)
        command.execute( )
     }
  }
}
```

```
val robot = Robot( )

robot.rotateRight( )
   .moveForward( )
   .moveForward()
   .rotateLeft( )
   .moveForward( )
   .moveForward( )
   .rotateLeft( )
   .moveBackwards( )
   .execute( )
```

```
class Robot {

    private val commands = mutableListOf<Command>()

    fun rotateLeft( ) = apply {
        commands.add { ... }
    }

    fun rotateRight( ) ...

    fun moveForward( ) ...

    fun moveBackwards( ) ...

    fun execute() {
        while (!commands.isEmpty( )) {
            val command = commands.removeAt(0)
            command.execute( )
        }
    }
}
```

```
val robot = Robot( )

robot.rotateRight( )
    .moveForward( )
    .moveForward()
    .rotateLeft( )
    .moveForward( )
    .moveForward( )
    .rotateLeft( )
    .moveBackwards( )
    .execute( )
```

```
val manager = supportFragmentManager
val transaction = manager.beginTransaction()

transaction
    .add(mainContainer, fragment)
    .replace(sideContainer, fragment2)
    .commit()
```

TEMPLATE

Define los pasos de un algoritmo

Permite cambiar que hace cada paso sin cambiar la estructura del algoritmo.

Base

Proteina

Mix-ins

Toppings

Salsas

```kotlin
abstract class Poke() {

    open fun addBase() {
        addSushiRice
    }

    abstract fun addProtein()

    abstract fun addMixins()

    abstract fun addToppings()

    open fun addSauces() {}

    fun cookPoke() {
        addBase()
        addMixins()
        addToppings()
        addProtein()
        addSauces()
    }
}
```

```kotlin
class Tartar() : Poke() {
    override fun addProtein() {
        addTuna()
    }

    override fun addMixins() {
        addAvocado()
    }

    override fun addToppings() {
        addCrispyOnion()
        addSesame()
    }

    override fun addSauces() {
        addSricachaMayo()
    }
}

// LLamado del metodo que ejecuta el algoritmo
Tartar().cookPoke()
```

```
fun poke(
    addBase: () -> Unit = { addSushiRice() },
    addProtein: () -> Unit,
    addMixins: () -> Unit,
    addToppings: () -> Unit,
    addSauces: (() -> Unit)? = null
) {
    addBase()
    addMixins()
    addToppings()
    addProtein()
    addSauces?.let { it() }
}
```

```
fun tartar() {
    poke(addProtein = addTuna,
        addMixins = addAvocado,
        addToppings = {
            addCrispyOnion()
            addSesame()
        })
}
```

GRACIAS