SOLID: Principios de diseño (POO).

- SRP → SINGLE-RESPONSIBILITY PRINCIPLE: Nunca debe haber más de una razón para que una clase cambie. Agrupa las cosas que cambian por las mismas razones. Separa las cosas que cambian por razones diferentes (diferentes roles). La razón por la que es importante mantener una clase enfocada en una sola preocupación es que hace que la clase sea más robusta al haber menos acomplamiento.

- OCP → OPEN-CLOSED PRINCIPLE: Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para extensión, pero cerradas para modificación. Una entidad puede permitir que su comportamiento sea extendido sin modificar su código fuente (Generalización-Herencia). Esto asume que al módulo se le ha dado una descripción bien definida y estable. Las variables globales/públicas violane el OCP.
    - Polymorphic open–closed principle: (Clases abstractas-Interfaces) las implementaciones pueden cambiarse y se pueden crear múltiples implementaciones que podrían ser sustituidas polimórficamente unas por otras.

- LSP → LISKOV SUBSTITUTION PRINCIPLE: Una clase B es una subclase de la clase A, entonces debe ser posible usar B donde sea que A se espere sin cambiar el comportamiento del programa.

- ISP → INTERFACE SEGREGATION PRINCIPLE: Ningún código debe ser forzado a depender de métodos que no usa. ISP divide interfaces muy grandes en interfaces más pequeñas y específicas para que los clientes solo tengan que conocer los métodos que les interesan. El ISP está destinado a mantener un sistema desacoplado y, por lo tanto, más fácil de refactorizar, cambiar y rediseñar.

- DIP → DEPENDENCY INVERSION PRINCIPLE:
  Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones (por ejemplo, interfaces).
  Las abstracciones no deben depender de detalles. Los detalles (implementaciones concretas) deben depender de las abstracciones.
  En muchos casos, pensar en la interacción como un concepto abstracto permite reducir el acoplamiento de los componentes, facilitando la reutilización y las modificaciones en los módulos de bajo nivel..
  Supón que tienes una clase Aplicación (módulo de alto nivel) que depende de una clase BaseDeDatos (módulo de bajo nivel).
    - En lugar de que Aplicacion dependa directamente de BaseDeDatos, puedes introducir una interfaz abstracta IBaseDeDatos que separe los dos módulos. Aplicacion debería depender de IBaseDeDatos, y BaseDeDatos debería implementar IBaseDeDatos.
    - Así, si decides cambiar la implementación de la base de datos en el futuro (por ejemplo, pasar de MySQL a PostgreSQL), solo necesitarás crear una nueva clase que implemente IBaseDeDatos. No tendrás que hacer ningún cambio en la clase Aplicacion, ya que su dependencia es con la abstracción IBaseDeDatos, no con una implementación específica.

ANTIPATRONES

1) The Blob: Una sola clase se encarga de la mayoría de las responsabilidades mientras que otras clases son simples contenedores de datos. Tiene ausencia de diseño orientado a objetos (o de cualquier arquitectura en general). Ocurre cuando los programadores agregan funcionalidad en lugar de nuevas clases.
   Consecuencia: Dificulta la mantenibilidad y comprensión del código. Reduce la reusabilidad de las clases. Puede afectar el rendimiento debido a la carga excesiva en una sola clase.
   Solución: La solución incluye refactorizar el diseño para distribuir las responsabilidades de manera más uniforme y aislar el efecto de los cambios, eliminar todas las asociaciones indirectas "acopladas lejanamente" o redundantes.

2) Lava Flow: Ocurre cuando el código muerto y la información de diseño olvidada se acumulan en un diseño en constante cambio sin documentación y falta de entendimiento. Utilizar solamente en prototipos desechables.
   Consecuencia: Aumento de la complejidad del código. Reducción de la mantenibilidad. Inflación del tamaño del código, haciendo que el software sea más difícil de entender y modificar.
   Solución: documentar bien el código y cambios realizados, implementar revisiones de código periódicas, refactorizar código obsoleto y eliminar código innecesario luego de pruebas rigurosas.

3) Golden Hammer: Se refiere al uso obsesivo de una tecnología o concepto familiar en una amplia variedad de problemas de software., ya que se percibe como algo que se debe resolver mejor con el Golden Hammer. En muchos casos, no es adecuado para el problema, pero se dedica un esfuerzo mínimo a explorar soluciones alternativas.
   Consecuencia: soluciones ineficientes, mayor costo y tiempo de desarrollo y limitación en la innovación.
   Solución: educar al equipo, realizar evaluaciones objetivas antes de seleccionar herramienta o tecnología, fomentar un entorno innovativo.

4) Spaghetti Code: El código fuente de un programa tiene una estructura compleja y enredada, parecida a un plato de espaguetis, lo que lo hace difícil de leer, mantener y depurar. Las causas típicas incluyen la falta de experiencia en diseño orientado a objetos y revisiones de código, desarrollo apresurado y modificaciones y parches sucesivos.
   Consecuencia: Dificultad en mantenimiento y depuración. Incremento en el riesgo de introducir errores. Mayor costo y tiempo de desarrollo.
   Solución: dividir el código en funciones y clases más pequeñas y coherentes. Reemplazar saltos incondicionales con estructuras de control más legibles. Mejorar la documentación y comentarios.

5) Cut−and−Paste Programming: Se refiere a la práctica de copiar y pegar bloques de código dentro de una aplicación, en lugar de crear funciones o módulos reutilizables. A menudo ocurre cuando hay poca familiaridad con las nuevas tecnologías.
   Consecuencia: Conlleva altos costos de mantenimiento, duplicación de código, código menos legible.

Solución: Pensar en modularidad y reutilización de código desde el inicio. Crear funciones y clases para lógicas comunes. Identificar y eliminar duplicación. Refactorizar.

6) Tester Driven Development: Se refiere a un antipatrón en el que las pruebas de software o los informes de errores dirigen el desarrollo de software, en lugar de las necesidades del usuario o los requerimientos de las funcionalidades.
Consecuencia: baja calidad del código y retrasos en la entrega del software.
Solución: comenzar las pruebas de software en el momento adecuado y no demasiado pronto. Requerimientos completos y bien definidos. Adecuada capacitación del equipo. Mantener un enfoque en el valor del usuario y los requerimientos de las funcionalidades.

DESIGN PATTERNS

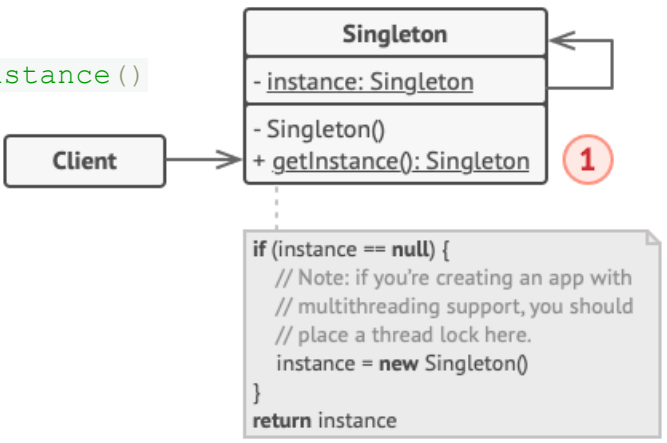
CREACIONALES

1. SINGLETON: Se utiliza para garantizar que una clase sólo tenga una única instancia en todo el programa, proporcionando un punto de acceso global a esa instancia. El objeto Singleton solo se inicializa cuando se requiere por primera vez.

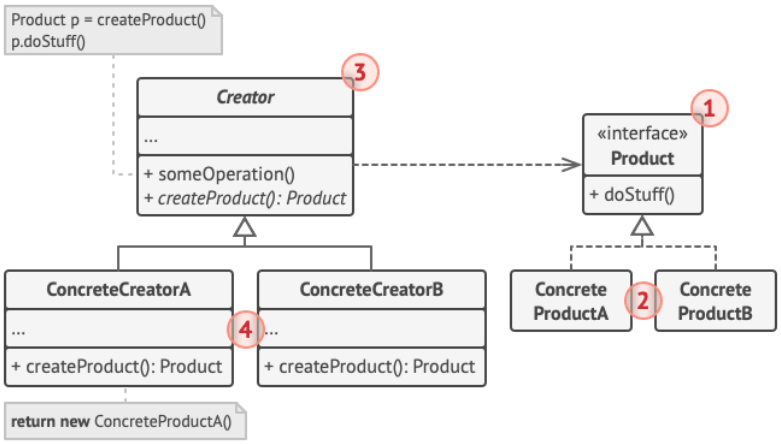
CONS: violates SRP, can mask bad design when the components know too much about each other. requires special treatment in multithread, difficult to do unit tests.

Convierte al constructor en una clase privada. Crea un static method que actúe como el constructor, y solo este tendrá acceso al constructor, siempre devolverá la misma instancia.

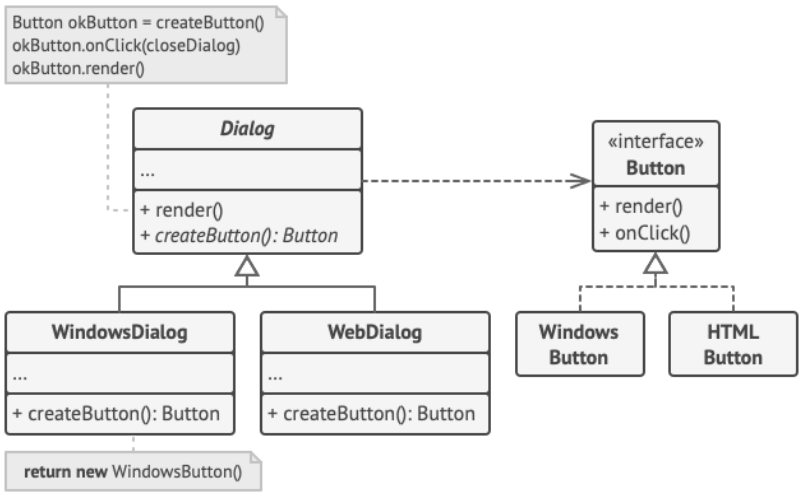
```
class Database is
    private static field instance: Database
    private constructor Database() is

    public static method getInstance() is
        // Return existing instance or create instance
    return Database.instance

    // Finally, any singleton should define some business logic
    // which can be executed on its instance.
    public method query(sql) is
        // …

class Application is
    method main() is
        Database foo = Database.getInstance()
```



2. FACTORY METHOD: Se utiliza para encapsular la creación de objetos en una clase separada, permitiendo delegar la responsabilidad de la creación de objetos a la clase Fábrica. Sugiere reemplazar las llamadas directas de construcción de objetos (usando el operador new) con llamadas a un método de Creator. Se puede sobrescribir el método de fábrica en una subclase y cambiar la clase de productos que se crean mediante el método. Proporciona una interfaz para crear objetos en una superclase y permite a las subclases alterar el tipo de objetos que se desean crear, facilitando la extensibilidad del código.
    - Abstract Factory: Puedes declarar el factory method como abstracto para obligar a todas las subclases a implementar sus propias versiones del método. Como alternativa, el método de fábrica base puede devolver algún tipo de producto predeterminado.



PROS: avoid tight coupling between creator and products, SRP, OCP.
CONS: complexity

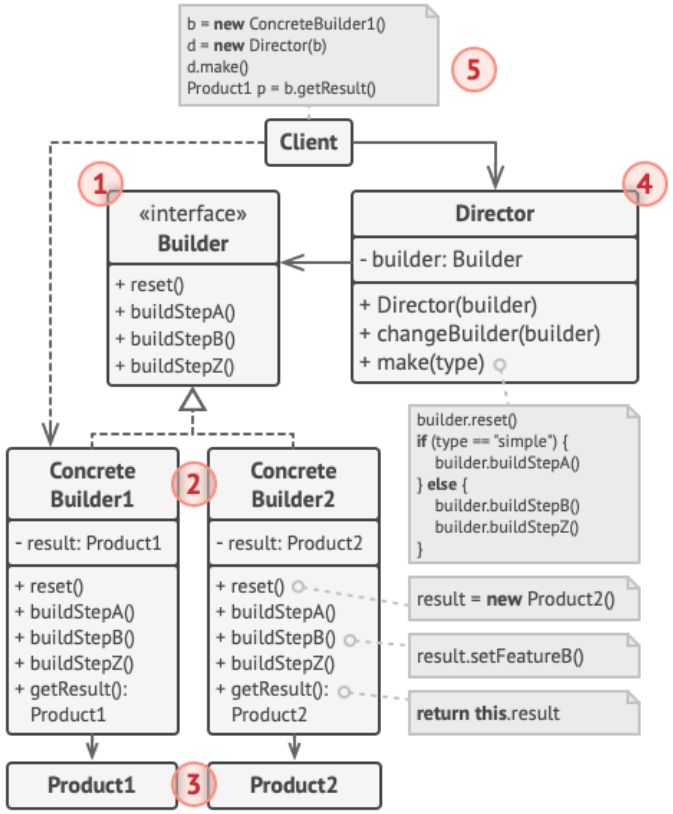


EJEMPLO: Los elementos UI al renderizar en diferentes sistemas operativos se pueden ver un poco diferentes, pero deben comportarse igual. Factory Method permite que no tengas que reescribir la lógica en cada caso. Mejor declaramos una clase base Dialog que produce botones, y subclases que devuelva Windows-styled buttons o HTML-styled buttons. En este caso, conviene que Dialog sea abstracta, para obligar a sobreescribir.

```
class Dialog is
    // The creator may also provide some default implementation
    // of the factory method.
    abstract method createButton():Button
    method render() is
        // Call the factory method to create a product object.
        Button okButton = createButton()
        // Now use the product.
        okButton.onClick(closeDialog)
        okButton.render()

class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()

// The product interface declares the operations that all
// concrete products must implement.
interface Button is
    method render()
    method onClick(f)

// Concrete products provide various implementations of the
// product interface.
class WindowsButton implements Button is
    method render(a, b) is // ...
    method onClick(f) is // ...

class HTMLButton implements Button is
    method render(a, b) is // ...
    method onClick(f) is // ...

class Application is
    field dialog: Dialog
    method initialize() is
        config = readApplicationConfigFile()
        if (config.OS == "Windows") then
            dialog = new WindowsDialog()
        else if (config.OS == "Web") then
            dialog = new WebDialog()
        else
            throw new Exception("Error! Unknown operating
system.")
    method main() is
        this.initialize()
```

```
        dialog.render()
```

3. BUILDER: Permite construir objetos complejos paso a paso, permitiendo también producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción. Builder abstrae el proceso de construcción del objeto final de su representación interna, permite la creación de diferentes variaciones o configuraciones del objeto final. Separa la lógica de construcción del objeto de la clase del objeto en sí. Especialmente útil cuando existe un objeto complejo que requiere una inicialización laboriosa paso a paso de muchos campos.
   PROS: SRP, step-by-step construction, reuse construction code.
   CONS: complexity.



La interfaz o clase abstracta “Builder“ define el comportamiento con los métodos llamados “steps”.
Las subclases implementan el comportamiento.
La clase “Director” define qué subclases y pasos se ejecutan.

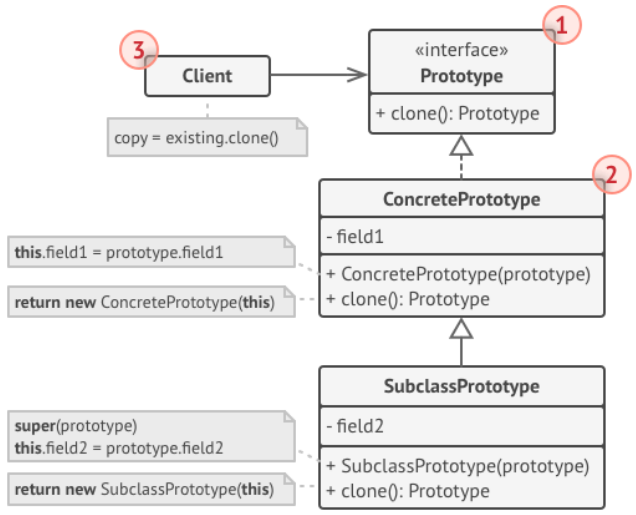
EJEMPLO: un auto puede ser construido de diferentes maneras, en lugar del constructor de Car quedar sobrecargado, extraemos el código a una clase Assembly. Si el cliente necesita armar un auto especial, puede comunicarse directamente con el Builder, en caso de querer un modelo de auto conocidos, delega la responsabilidad a Director, que sabe cómo usar un builder para construirlo.
Todos los autos tienen un manual, donde se describen las características y partes del auto, por ello es conveniente reusar el proceso de construcción del auto con el manual. Como manual no es lo mismo a armar un auto, utilizará los mismos métodos de construcción, pero en lugar de armar un auto con las piezas, las describe.

```
class Car is // ...
class Manual is // ...

// The builder interface specifies methods for creating the
```

```
// different parts of the product objects.
interface Builder is
    method reset()
    method setSeats(...)
    method setEngine(...)
    method setTripComputer(...)
    method setGPS(...)

class CarBuilder implements Builder is
    private field car:Car
    // A fresh builder instance should contain a blank product.
    constructor CarBuilder() is
        this.reset()
    method reset() is
        this.car = new Car()
    // All production steps work with the same product instance.
    method setSeats(...) is //...
    method setEngine(...) is //...
    method setTripComputer(...) is //...
    method setGPS(...) is //...
    method getProduct():Car is
        product = this.car
        this.reset()
        return product

// Unlike other creational patterns, builder lets you construct
// products that don't follow the common interface.
class CarManualBuilder implements Builder is
    private field manual:Manual
    constructor CarManualBuilder() is
        this.reset()
    method reset() is
        this.manual = new Manual()
    method setSeats(...) is //...
    method setEngine(...) is //...
    method setTripComputer(...) is //...
    method setGPS(...) is //…
    method getProduct():Manual is //...

// The director is helpful when producing
// products according to a specific order or configuration.
class Director is
    method constructSportsCar(builder: Builder) is
        builder.reset()
        builder.setSeats(2)
        builder.setEngine(new SportEngine())
        builder.setTripComputer(true)
        builder.setGPS(true)
```

```
    method constructSUV(builder: Builder) is
        // ...

// The client code creates a builder object, passes it to the
// director and then initiates the construction process.
class Application is
    method makeCar() is
        director = new Director()
        CarBuilder builder = new CarBuilder()
        director.constructSportsCar(builder)
        Car car = builder.getProduct()
        CarManualBuilder builder = new CarManualBuilder()
        director.constructSportsCar(builder)
        Manual manual = builder.getProduct()
```

4. PROTOTYPE: Se utiliza para crear nuevos objetos clonando un objeto existente, evitando la creación de objetos desde cero. El patrón declara una interfaz común para todos los objetos que admiten la clonación. Esta interfaz te permite clonar un objeto sin acoplar tu código a la clase de ese objeto. Usualmente, dicha interfaz contiene solo un método de clonación. ütil cuando se necesita gestionar diferentes estados o versiones de un “mismo” objeto, para evitar dependencias a clases concretas.



PROS: clone objects without coupling to their concrete classes, get rid of repeated initialization code in favor of cloning pre-built prototypes.
CONS: cloning complex objects with circular references might be very tricky.

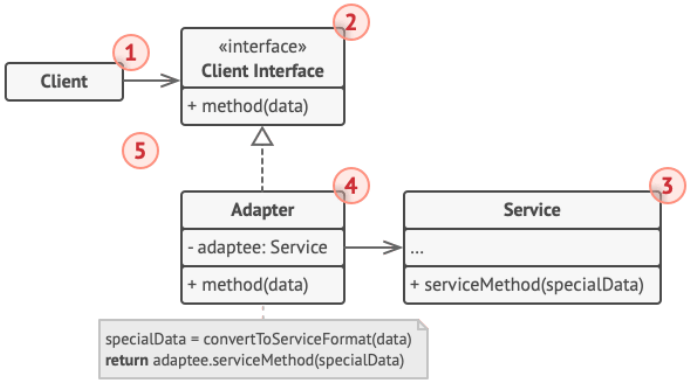
EJEMPLO: código que te permite realizar copias exactas de figuras geométricas. El Prototype es “Shape” y, “Rectangle” y “Circle” son las clases concretas.

```
abstract class Shape is
    field X: int
    field Y: int
    field color: string
    // Regular constructor.
    constructor Shape() is // ...
    // The prototype constructor. A fresh object is initialized
    // with values from the existing object.
    constructor Shape(source: Shape) is
```

```
        this()
        this.X = source.X
        this.Y = source.Y
        this.color = source.color
    // The clone operation returns one of the Shape subclasses.
    abstract method clone():Shape

class Rectangle extends Shape is
    field width: int
    field height: int
    constructor Rectangle(source: Rectangle) is
        // A parent constructor call is needed to copy private
        // fields defined in the parent class.
        super(source)
        this.width = source.width
        this.height = source.height
    method clone():Shape is
        return new Rectangle(this)

class Circle extends Shape is
    field radius: int
    constructor Circle(source: Circle) is
        super(source)
        this.radius = source.radius
    method clone():Shape is
        return new Circle(this)

class Application is
    field shapes: array of Shape
    constructor Application() is
        Circle circle = new Circle()
        circle.radius = 20
        shapes.add(circle)
        Circle anotherCircle = circle.clone()
```

ESTRUCTURALES

1. ADAPTER: Es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles. A grandes rasgos, se trata de un objeto especial que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla. Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en un formato y orden que el segundo objeto espera.



También se puede utilizar herencia para que el Adapter herede el comportamiento del cliente y del servicio.

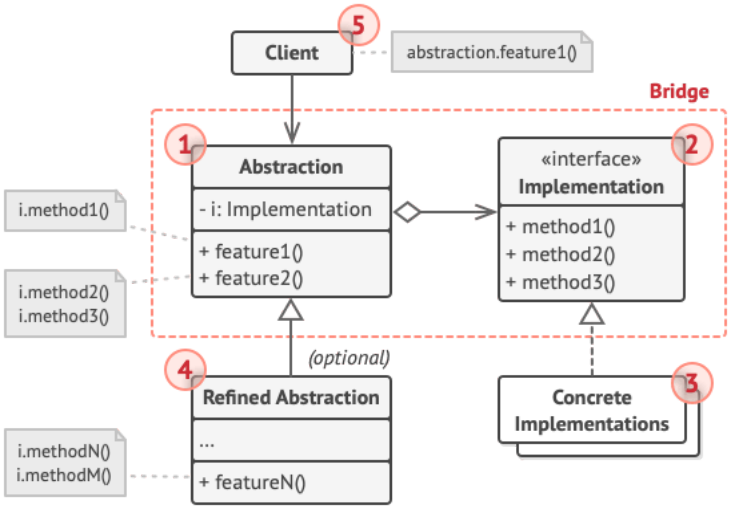
PROS: SRP permite separar la interfaz o el código de conversión, de la lógica de negocio. OCP, introducir nuevos tipos de adaptadores al programa sin descomponer el código cliente existente..
CONS: Complexity, sometimes it's simpler just to change the service class so that it matches the rest of your code.

EJEMPLO: enchufe cuadrado y tomacorriente redondo

```
class RoundHole is
    constructor RoundHole(radius) { ... }
    method getRadius() is
        // Return the radius of the hole.
    method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()

class SquarePeg is
    constructor SquarePeg(width) { ... }
    method getWidth() is
        // Return the square peg width.
class SquarePegAdapter extends RoundPeg is
    private field peg: SquarePeg
    constructor SquarePegAdapter(peg: SquarePeg) is
        this.peg = peg
    method getRadius() is
        // The adapter pretends that it's a round peg with a
        // radius that could fit the square peg that the adapter
        // actually wraps.
        return peg.getWidth() * Math.sqrt(2) / 2
```

```
// Somewhere in client code.
small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // this won't compile (incompatible types)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // true
hole.fits(large_sqpeg_adapter) // false
```

2. BRIDGE: Permite dividir una clase grande o un conjunto de clases relacionadas en dos jerarquías separadas: abstracción (interfaz, delega el trabajo) e implementación (realiza el trabajo), que pueden desarrollarse de manera independiente entre sí. El patrón Bridge pasa de la herencia a la composición del objeto. Esto quiere decir que se extrae una de las dimensiones a una jerarquía de clases separada. Utilizar cuando se quiere dividir y organizar una clase monolítica que tenga muchas variantes de una sola funcionalidad.
   PROS: create platform-independent classes and apps, high-level abstractions, OCP ya que puedo introducir nuevas abstracciones e implementaciones independientes entre sí, SRP ya que se centra por un lado en la lógica y por otro en la implementación..
   CONS: Complexity by applying to a highly cohesive class.



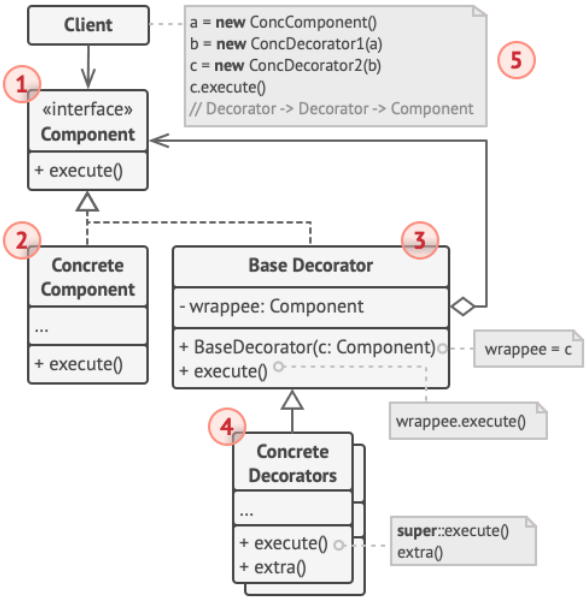
Usar cuando necesites extender una clase en varias dimensiones ortogonales (independientes).

EJEMPLO: aplicación que maneja dispositivos y controles remotos. Device es la interfaz Implementation, RemoteControl es la clase Abstraction

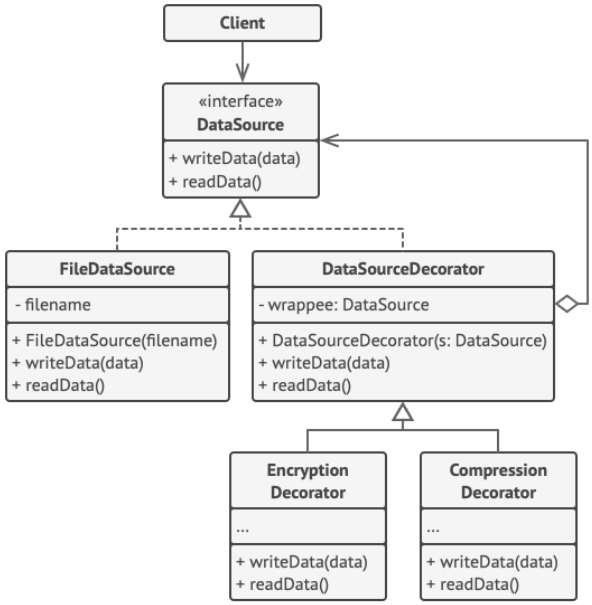
```
class RemoteControl is
    protected field device: Device
    constructor RemoteControl(device: Device) is
        this.device = device
    method togglePower() is
        if (device.isEnabled()) then
            device.disable()
```

```
        else
            device.enable()
    method volumeDown() is
        device.setVolume(device.getVolume() - 10)
    method volumeUp() is
        device.setVolume(device.getVolume() + 10)
    method channelDown() is
        device.setChannel(device.getChannel() - 1)
    method channelUp() is
        device.setChannel(device.getChannel() + 1)

class AdvancedRemoteControl extends RemoteControl is
    method mute() is
        device.setVolume(0)

interface Device is
    method isEnabled()
    method enable()
    method disable()
    method getVolume()
    method setVolume(percent)
    method getChannel()
    method setChannel(channel)

// All devices follow the same interface.
class Tv implements Device is
    // ...

class Radio implements Device is
    // ...

// Somewhere in client code.
tv = new Tv()
remote = new RemoteControl(tv)
remote.togglePower()

radio = new Radio()
remote = new AdvancedRemoteControl(radio)
```

3. DECORATOR: Permite añadir funcionalidades a objetos. Los decoradores proporcionan una alternativa flexible a la subclasificación para ampliar la funcionalidad. Consiste en encapsular el objeto original en una interfaz contenedora abstracta. Tanto los objetos decoradores como el objeto central heredan de esta interfaz abstracta. Usar cuando necesites poder asignar comportamientos adicionales a objetos en tiempo de ejecución sin romper el código que usa estos objetos.
   PROS: SRP, al dividir clase monolítica en varias clases pequeñas.
   CONS: resulta dificil eliminar un wrapper en específico de la pila de wrappers.



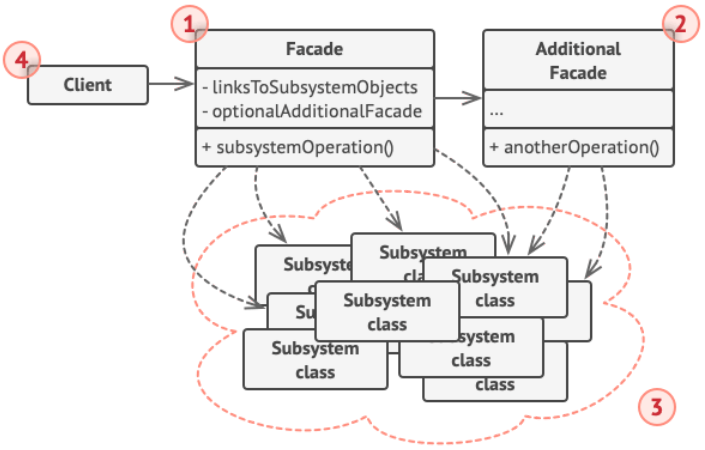
EJEMPLO: encriptar información sensible del código que usa esa información.




```
interface DataSource is
    method writeData(data)
    method readData():data

class FileDataSource implements DataSource is
    constructor FileDataSource(filename) { ... }
    method writeData(data) is
        // Write data to file.
    method readData():data is
        // Read data from file.

// The base decorator class follows the same interface as the
```

```
// other components. The primary purpose of this class is to
// define the wrapping interface for all concrete decorators.
class DataSourceDecorator implements DataSource is
    protected field wrappee: DataSource

    constructor DataSourceDecorator(source: DataSource) is
        wrappee = source

    // The base decorator simply delegates all work to the
    // wrapped component. Extra behaviors can be added in
    // concrete decorators.
    method writeData(data) is
        wrappee.writeData(data)

    method readData():data is
        return wrappee.readData()

class EncryptionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // ...

    method readData():data is
        // ...

class CompressionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // ...

    method readData():data is
        // ...

class Application is
    method dumbUsageExample() is
        source = new FileDataSource("somefile.dat")
        source.writeData(salaryRecords)
        // The target file has been written with plain data.
        source = new CompressionDecorator(source)
        source.writeData(salaryRecords)
        // The target file has been written with compressed data.
        source = new EncryptionDecorator(source)
        // Encryption > Compression > FileDataSource
        source.writeData(salaryRecords)
```

4. FACADE: Proporcionar una interfaz unificada a un conjunto de interfaces en un subsistema. Facade define una interfaz de nivel superior que facilita el uso del subsistema. Envuelve un subsistema complicado con una interfaz más simple. Podría proporcionar funcionalidad limitada en comparación con trabajar directamente con el subsistema. Sin embargo, incluye solo aquellas características que realmente le importan a los clientes.
   PROS: Puedes aislar tu código de la complejidad de un subsistema. Reducción del acoplamiento.
   CONS: Una fachada puede convertirse en un objeto todopoderoso acoplado a todas las clases de una aplicación.



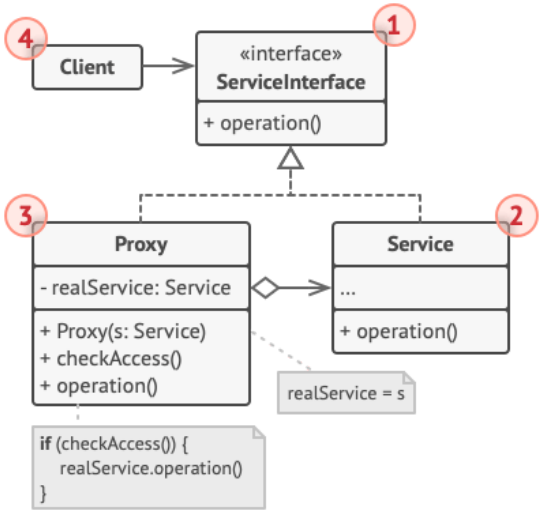
   Facade: sabe a dónde dirigir los pedidos de los clientes.
   Additional Facade: para evitar que Facade contenga features sin relación. Utilizado por clientes y otras facades.

EJEMPLO: interacción con frameworks complejos de conversión de videos.

```
// These are some of the classes of a complex 3rd-party video
// conversion framework.
class VideoFile
// ...
class OggCompressionCodec
// ...
class MPEG4CompressionCodec
// ...
class CodecFactory
// ...
class BitrateReader
// ...
class AudioMixer
// ...

class VideoConverter is
    method convert(filename, format):File is
        file = new VideoFile(filename)
        sourceCodec = (new CodecFactory).extract(file)
        if (format == "mp4")
```

```
            destinationCodec = new MPEG4CompressionCodec()
        else
            destinationCodec = new OggCompressionCodec()
        buffer = BitrateReader.read(filename, sourceCodec)
        result = BitrateReader.convert(buffer, destinationCodec)
        result = (new AudioMixer()).fix(result)
        return new File(result)

class Application is
    method main() is
        convertor = new VideoConverter()
        mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
        mp4.save()
```

5. PROXY: Permite proporcionar un sustituto o un placeholder para otro objeto. Un proxy controla el acceso al objeto original, lo que te permite realizar algo antes o después de que la solicitud llegue al objeto original. Es utilizado para optimizar recursos. El patrón Proxy sugiere que crees una nueva clase de proxy con la misma interfaz que un objeto de servicio original. Luego, actualiza tu aplicación para que pase el objeto de proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y delega todo el trabajo a él.
   PROS: OCP, puedes introducir nuevos proxies sin cambiar el servicio o los clientes. Puedes gestionar el ciclo de vida del objeto de servicio cuando a los clientes no les importa.
   CONS: La respuesta del servicio puede retrasarse.



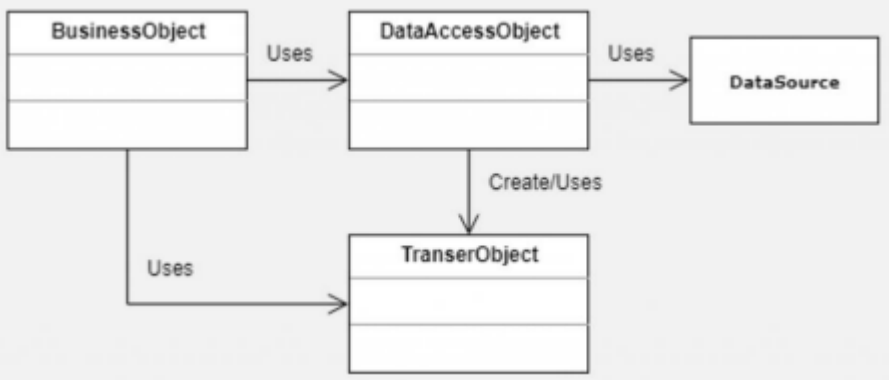
EJEMPLO: una librería de YouTube que nos ofrece el método descargar, sin embargo es muy eficiente. Si un cliente solicita varias veces su descarga, lo descarga una y otra vez en lugar de reusar la primera descarga. La clase Proxy implementa la misma interfaz que el descargador original y delega el trabajo, pero mantiene seguimiento de las descargas y devuelve los resultados en caché cuando la app solicita el mismo video múltiples veces.

```
interface ThirdPartyYouTubeLib is
    method listVideos()
```

```
    method getVideoInfo(id)
    method downloadVideo(id)

class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib is
    method listVideos() is
        // Send an API request to YouTube.
    method getVideoInfo(id) is
        // Get metadata about some video.
    method downloadVideo(id) is
        // Download a video file from YouTube.

class CachedYouTubeClass implements ThirdPartyYouTubeLib is
    private field service: ThirdPartyYouTubeLib
    private field listCache, videoCache
    field needReset
    constructor CachedYouTubeClass(service: ThirdPartyYouTubeLib)
is
        this.service = service
    method listVideos() is
        if (listCache == null || needReset)
            listCache = service.listVideos()
        return listCache
    method getVideoInfo(id) is
        if (videoCache == null || needReset)
            videoCache = service.getVideoInfo(id)
        return videoCache
    method downloadVideo(id) is
        if (!downloadExists(id) || needReset)
            service.downloadVideo(id)


// The GUI class, which used to work directly with a service
// object, we can safely pass a proxy object instead of a real
// service object since they both implement the same interface.
class YouTubeManager is
    protected field service: ThirdPartyYouTubeLib

    constructor YouTubeManager(service: ThirdPartyYouTubeLib) is
        this.service = service

    method renderVideoPage(id) is
        info = service.getVideoInfo(id)
        // Render the video page.

    method renderListPanel() is
        list = service.listVideos()
        // Render the list of video thumbnails.
```

```
    method reactOnUserInput() is
        renderVideoPage()
        renderListPanel()

// The application can configure proxies on the fly.
class Application is
    method init() is
        aYouTubeService = new ThirdPartyYouTubeClass()
        aYouTubeProxy = new CachedYouTubeClass(aYouTubeService)
        manager = new YouTubeManager(aYouTubeProxy)
        manager.reactOnUserInput()
```

6. DAO (Data Access Object): Es una abstracción de la persistencia de datos y se considera más cercano al repositorio de almacenamiento, que a menudo está centrado en tablas (base de datos). En muchos casos, nuestros DAO coinciden con las tablas de la base de datos, lo que permite una forma más sencilla de enviar/recuperar datos del almacenamiento, ocultando las consultas desagradables.



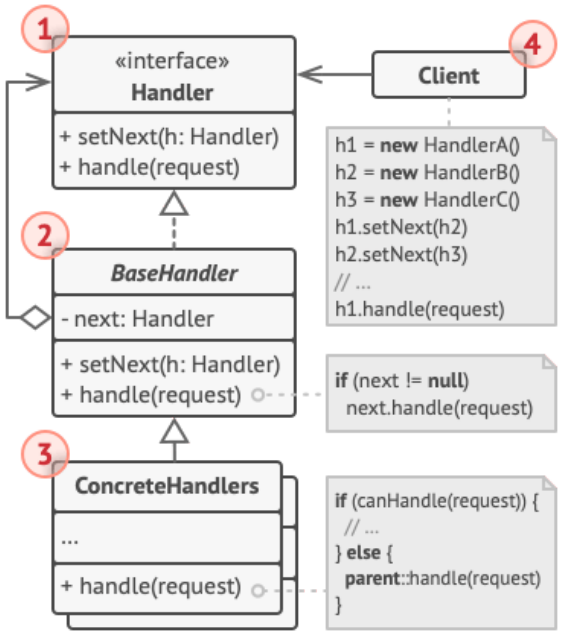
7. REPOSITORY: Es un mecanismo para encapsular el comportamiento de almacenamiento, recuperación y búsqueda, que emula una colección de objetos. • En otras palabras, un repositorio también trata con datos y oculta consultas similares a DAO. Sin embargo, se encuentra en un nivel superior, más cerca de la lógica de negocio de una aplicación. En consecuencia, Repository puede utilizar DAO para la obtención o persistencia de datos en la base de datos.

DAO vs. REPOSITORY

- DAO es una abstracción de la persistencia de datos.
+ Repository es una abstracción de una colección de objetos.
- DAO es un concepto de nivel inferior, más cercano a los sistemas de almacenamiento.
+ Repository es un concepto de nivel superior, más cercano a los objetos de Negocio.
- DAO funciona como una capa de acceso/mapeo de datos, ocultando consultas desagradables.
+ Repository es una capa entre Negocio y capas de acceso a datos, lo que oculta la complejidad de recopilar datos y preparar un objeto de negocio.
+ Repository puede usar un DAO para acceder al almacenamiento de datos

COMPORTAMIENTO

1. CHAIN OF RESPONSABILITY: Encapsula elementos de procesamiento dentro de una abstracción de “pipeline" y permite que los clientes envíen sus solicitudes a la entrada de esta ahí sin tener que gestionar cómo serán procesadas.
   En este patrón, los objetos que reciben la solicitud están encadenados entre sí. Una solicitud es pasada de un objeto a otro a lo largo de la cadena hasta que encuentra un objeto que es capaz de manejarla. No es necesario saber de antemano el número y tipo de objetos manejadores, ya que pueden configurarse dinámicamente.
   Simplifica las interconexiones entre objetos. En lugar de que los emisores y receptores mantengan referencias a todos los receptores candidatos, cada emisor mantiene una sola referencia a la cabeza de la cadena, y cada receptor mantiene una sola referencia a su sucesor inmediato en la cadena.
   Hay que asegurar tener una red de seguridad para solicitudes que no sean manejadas por ningún objeto.
   PROS: control the order of request handling, SRP, OCP.
   CONS: some requests may end up unhandled.



   EJEMPLO: estamos realizando el software para un banco y uno de los puntos más importantes es saber quién puede aprobar un crédito.

   //Main
   Banco banco = new Banco();
   banco.solicitudPrestamo(5600);

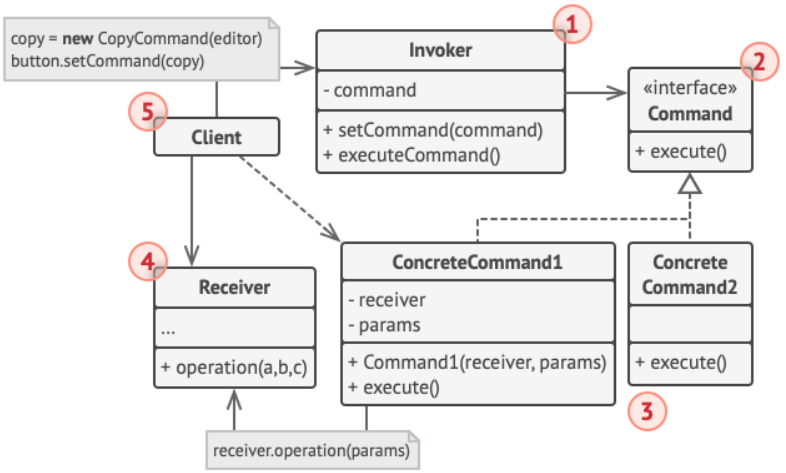
```
public interface IAprobador {

    public void setNext(IAprobador aprobador);
    public IAprobador getNext();
    public void solicitudPrestamo(int monto);

}
```

```java
public class EjecutivoDeCuenta implements IAprobador {
    private IAprobador next;

    public IAprobador getNext() {
        return next;
    }

    public void solicitudPrestamo(int monto) {
        if (monto <= 10000) {
            System.out.println("Lo manejo yo, el ejecutivo de cuentas");
        } else {
            next.solicitudPrestamo(monto);
        }
    }

    public void setNext(IAprobador aprobador) {
        next = aprobador;
    }

}

public class LiderTeamEjecutivo implements IAprobador {
    private IAprobador next;

    public IAprobador getNext() {
        return next;
    }

    public void solicitudPrestamo(int monto) {
        if (monto > 10000 && monto <= 50000) {
            System.out.println("Lo manejo yo, el lider");
        } else {
            next.solicitudPrestamo(monto);
        }
    }

    public void setNext(IAprobador aprobador) {
        next = aprobador;
    }
```

```java
public class Director implements IAprobador {
    private IAprobador next;

    public IAprobador getNext() {
        return next;
    }

    public void solicitudPrestamo(int monto) {
        if (monto >= 100000) {
            System.out.println("Lo manejo yo, el director");
        }
    }
```

```java
public class Banco implements IAprobador {
    private IAprobador next;

    public IAprobador getNext() {
        return next;
    }

    public void solicitudPrestamo(int monto) {
        EjecutivoDeCuenta ejecutivo = new EjecutivoDeCuenta();
        this.setNext(ejecutivo);

        LiderTeamEjecutivo lider = new LiderTeamEjecutivo();
        ejecutivo.setNext(lider);

        Gerente gerente = new Gerente();
        lider.setNext(gerente);

        Director director = new Director();
        gerente.setNext(director);

        next.solicitudPrestamo(monto);
    }

    public void setNext(IAprobador aprobador) {
        next = aprobador;
    }
}
```

2. COMMAND: Transforma una solicitud en un objeto independiente con toda la información sobre la solicitud. Esto permite parametrizar los objetos con operaciones, la idea central detrás del Patrón Command es la encapsulación de una solicitud de una acción a ser llevada a cabo en nombre de un objeto con los parámetros de esa acción.
   PROS: mayor desacoplamiento entre clases, la posibilidad de controlar diversas solicitudes fácilmente y una mayor flexibilidad en la ejecución de operaciones. Facilita la adición de nuevas operaciones sin modificar el código existente.
   CONS: puede llevar a un mayor número de clases y objetos, lo que aumenta la complejidad del código. El mantenimiento de un historial de solicitudes puede consumir bastante memoria, lo cual podría ser una desventaja en sistemas con recursos limitados.



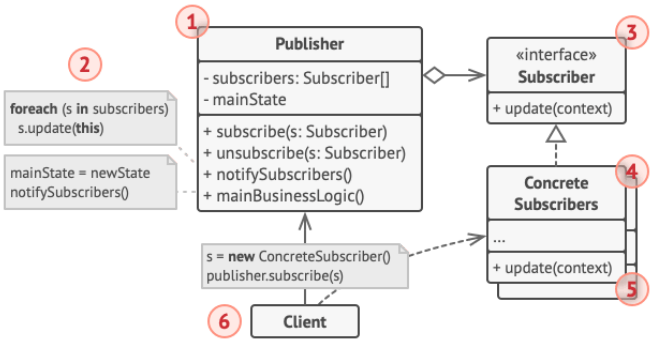
EJEMPLO: como mantiene el historial de operaciones, es utilizado en editores de texto.

```
abstract class Command is
    protected field app: Application
    protected field editor: Editor
    protected field backup: text
    constructor Command(app: Application, editor: Editor) is
        this.app = app
        this.editor = editor
    method saveBackup() is
        backup = editor.text
    method undo() is
        editor.text = backup
    abstract method execute()

class CopyCommand extends Command is
    // The copy command isn't saved to the history since it
    // doesn't change the editor's state.
    method execute() is
        app.clipboard = editor.getSelection()
        return false

class CutCommand extends Command is
```

```
    // The cut command does change the editor's state, therefore
    // it must be saved to the history.
    method execute() is
        saveBackup()
        app.clipboard = editor.getSelection()
        editor.deleteSelection()
        return true

class PasteCommand extends Command is
    method execute() is
        saveBackup()
        editor.replaceSelection(app.clipboard)
        return true

class UndoCommand extends Command is
    method execute() is
        app.undo()
        return false

// The global command history is just a stack.
class CommandHistory is
    private field history: array of Command
    // Last in...
    method push(c: Command) is
        // Push the command to the end of the history array.
    // ...first out
    method pop():Command is
        // Get the most recent command from the history.

class Editor is
    field text: string
    method getSelection() is
        // Return selected text.
    method deleteSelection() is
        // Delete selected text.
    method replaceSelection(text) is
        // Insert the clipboard's contents at the current
        // position.


// The application class sets up object relations. It acts as a
// sender: when something needs to be done, it creates a command
// object and executes it.
class Application is
     private String clipboard;
     private CommandHistory history = new CommandHistory();
     private Editor editor = new Editor();
```

```
public String getClipboard() {
  return clipboard;
}
public void setClipboard(String clipboard) {
  this.clipboard = clipboard;
}
public void executeCommand(Command command) {
  if (command.execute()) {
    history.push(command);
  }
}
public void undo() {
  Command command = history.pop();
  if (command != null) {
    command.undo();
  }
}

//Main
app.executeCommand(new CutCommand(app, app.editor));
app.executeCommand(new PasteCommand(app, app.editor));
app.undo();
```

3. OBSERVER: Permite definir un mecanismo de suscripción para notificar a múltiples objetos sobre eventos que ocurren en el objeto que están observando. El objeto que tiene algún estado interesante a menudo se llama Subject o Publisher. Todos los demás objetos que desean rastrear cambios en el estado del Publisher se llaman Subscribers.
   Este mecanismo consta de 1) un campo de array para almacenar una lista de referencias a objetos subscribers y 2) varios métodos públicos que permiten agregar subscribers y eliminarlos de esa lista.



   PROS: OCP, establish relations between objects at runtime.
   CONS: subscribers are notified in random order.

EJEMPLO: Vamos a suponer un ejemplo de una Biblioteca, donde cada vez que un lector devuelve un libro se ejecuta el método devuelveLibro(Libro libro) de la clase Biblioteca.

Si el lector devolvió el libro dañado entonces la aplicación avisa a ciertas clases que están interesadas en conocer este evento:

```
public class Libro {
    private String tiulo;
    private String estado;

    // Un libro seguramente tendrá más atributos
    // como autor, editorial, etc pero para nuestro
    // ejemplo no son necesarios.

    public String getTiulo() {
        return tiulo;
    }

    public void setTiulo(String tiulo) {
        this.tiulo = tiulo;
    }
```

```
public interface ILibroMalEstado {

    public void update();
}
```

```
public class Administracion implements ILibroMalEstado {

    public void update() {
        System.out.println("Administracion: ");
        System.out.println("Envio una queja formal...");
    }

}
```

```
public class Stock implements ILibroMalEstado {

public void update() {
        System.out.println("Stock: ");
        System.out.println("Le doy de baja...");
    }
}
```

```
public interface Subject {

    public void attach(ILibroMalEstado observador);
    public void dettach(ILibroMalEstado observador);
    public void notifyObservers();

}
```

```
public class AlarmaLibro implements Subject {
    private static ArrayList<ILibroMalEstado> observadores =
        new ArrayList<ILibroMalEstado>();

    public void attach(ILibroMalEstado observador) {
        observadores.add(observador);
    }

    public void dettach(ILibroMalEstado observador) {
        observadores.remove(observador);
    }

    public void notifyObservers() {
        for (int i = 0; i < observadores.size(); i++) {
            observadores.get(i).update();
        }
    }

}
```

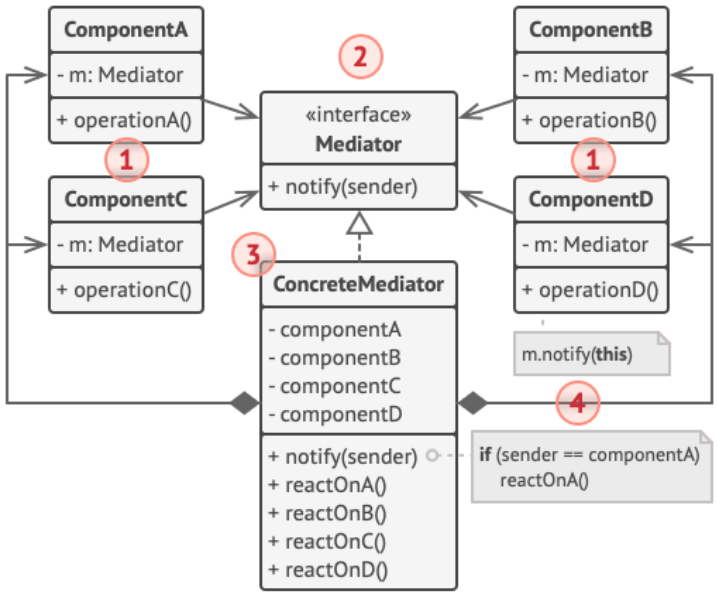
```
public class Biblioteca {

    public void devuelveLibro(Libro libro) {
        if (libro.getEstado().equals("MALO")) {
            AlarmaLibro a = new AlarmaLibro();
            a.notifyObservers();
        }
    }

}
```

```
    public static void main(String[] args) {
        AlarmaLibro a = new AlarmaLibro();
        a.attach(new Compras());
        a.attach(new Administracion());
        a.attach(new Stock());

        Libro libro = new Libro();
        libro.setEstado("MALO");

        Biblioteca b = new Biblioteca();
        b.devuelveLibro(libro);

    }
```

Problems | Javadoc | Declaration | Console | Searc

```
<terminated> Main (12) [Java Application] C:\Program Files\Java\jre6\bin
Solicito nueva cotizacion...
Administracion:
Envio una queja formal...
Stock:
Le doy de baja...
```

4. MEDIATOR: Se utiliza para reducir la comunicación compleja entre objetos estrechamente relacionados. En lugar de que los objetos se comuniquen directamente entre sí, estos objetos interactúan a través de un objeto mediador central.
   PROS: SRP, OCP, reduce coupling, reuse components.
   CONS: over time a mediator can evolve into a God Object.

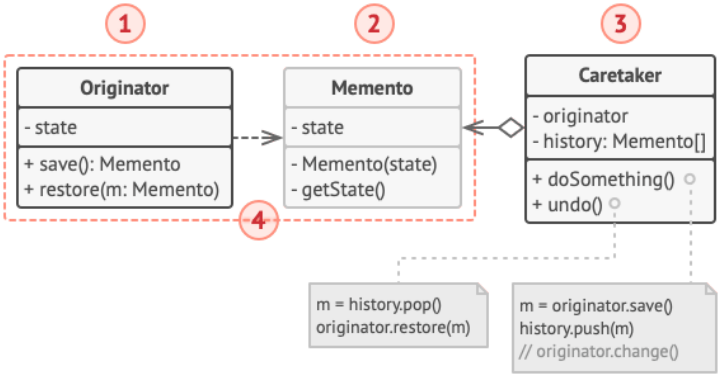


EJEMPLO: Nuestro ejemplo será un chat donde habrá usuarios que se comunicaran entre sí en un salón de chat. Para ellos se define una interfaz llamada IUsuarioChat que todos los objetos que quieran participar de un chat deberán implementar. La clase Usuario representa un usuario que quiera chatear.

```
public interface IUsuarioChat {

    public void recibe(String de, String msg);
    public void envia(String a, String msg);

}
public class Usuario implements IUsuarioChat {
    private String nombre;
    private SalonDeChat salon;

    public Usuario(SalonDeChat salonDeChat) {
        salon = salonDeChat;
    }

    public void recibe(String de, String msg) {
        String s = "el usuario " + de + " te dice: " + msg;
        System.out.println(nombre + ": " + s);
    }

    public void envia(String a, String msg) {
        salon.envia(nombre, a, msg);
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
```

```
public static void main(String[] args) {
    SalonDeChat s = new SalonDeChat();

    Usuario u = new Usuario(s);
    u.setNombre("Juan");
    s.registra(u);

    Usuario u1 = new Usuario(s);
    u1.setNombre("Pepe");
    s.registra(u1);

    Usuario u2 = new Usuario(s);
    u2.setNombre("Pedro");
    s.registra(u2);

    u.envia("Pepe", "Hola como andas?");
    u1.envia("Juan", "Todo ok, vos?");
    u2.envia("Martin", "Martin estas?");
}
```

5. MEMENTO: Se utiliza para capturar el estado interno de un objeto en un punto en el tiempo, de modo que el objeto pueda ser restaurado a ese estado más tarde. Esto es útil en características como los sistemas de deshacer/rehacer en editores de texto, o para tomar instantáneas del estado de un sistema. El Memento es un objeto que almacena el estado del Originator, Caretaker es el responsable de mantener registro de los diferentes estados del Originator mediante mementos. El Caretaker almacena y recupera mementos.
   PROS: produce snapshots without violating its encapsulation. simplify originators code by letting the caretaker maintain history of the originators state.
   CONS: consumes a lot of RAM, caretaker tracks the originators lifecycle, most dynamic programming languages can't guarantee that the state within the memento stays untouched.

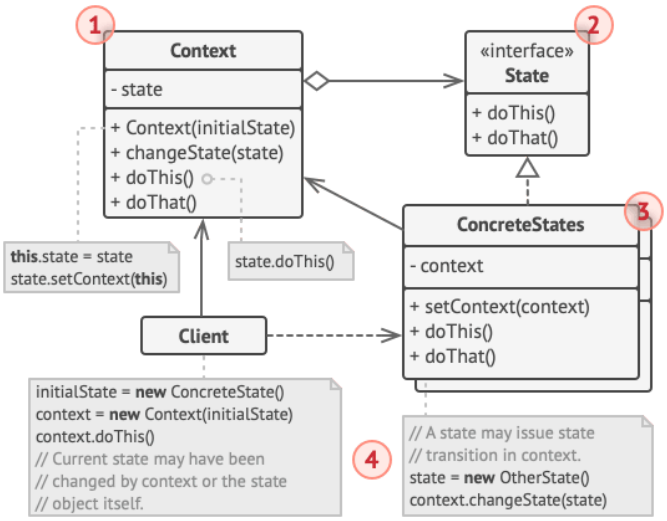


EJEMPLO: This example uses the Memento pattern alongside the Command (Caretaker) pattern for storing snapshots of the complex text editor's state and restoring an earlier state from these snapshots when needed.

```
// The originator holds some important data that may change over
// time. It also defines a method for saving its state inside a
// memento and another method for restoring the state from it.
class Editor is
    private field text, curX, curY, selectionWidth
    method setText(text) is
        this.text = text
    method setCursor(x, y) is
        this.curX = x
        this.curY = y
    method setSelectionWidth(width) is
        this.selectionWidth = width
    // Saves the current state inside a memento.
    method createSnapshot():Snapshot is
        return new Snapshot(this, text, curX, curY, selectionWidth)

// The memento class stores the past state of the editor.
class Snapshot is
    private field editor: Editor
    private field text, curX, curY, selectionWidth
```

```
    constructor Snapshot(editor, text, curX, curY,
selectionWidth)is
        this.editor = editor
        this.text = text
        this.curX = x
        this.curY = y
        this.selectionWidth = selectionWidth

    // At some point, a previous state of the editor can be
    // restored using a memento object.
    method restore() is
        editor.setText(text)
        editor.setCursor(curX, curY)
        editor.setSelectionWidth(selectionWidth)

// A command object can act as a caretaker. In that case, the
// command gets a memento just before it changes the
// originator's state. When undo is requested, it restores the
// originator's state from a memento.
class Command is
    private field backup: Snapshot

    method makeBackup() is
        backup = editor.createSnapshot()

    method undo() is
        if (backup != null)
            backup.restore()
    // ...
```

6. STATE: Permite a un objeto cambiar su comportamiento cuando su estado interno cambia. Esto puede ser útil en situaciones donde un objeto debe cambiar su comportamiento de manera dinámica en tiempo de ejecución, en función de ciertas condiciones. En otras palabras, el patrón State sugiere que se cree una nueva clase para cada estado posible de un objeto, y que se extraiga el comportamiento específico de ese estado a esa clase.
   En lugar de implementar todos los comportamientos por sí mismo, el objeto original, llamado Context, almacena una referencia a uno de los objetos de estado que representa su estado actual y delega todo el trabajo relacionado con el estado a ese objeto.
   Para cambiar el contexto a otro estado, se reemplaza el objeto de estado activo con otro objeto que representa ese nuevo estado. Esto es posible sólo si todas las clases de estado siguen la misma interfaz y el propio contexto trabaja con estos objetos a través de esa interfaz.
   PROS: SRP, OCP, simplify the code of the context by eliminating state machines.
   CONS: Applying the pattern can be overkill if a state machine has only a few states or rarely changes.

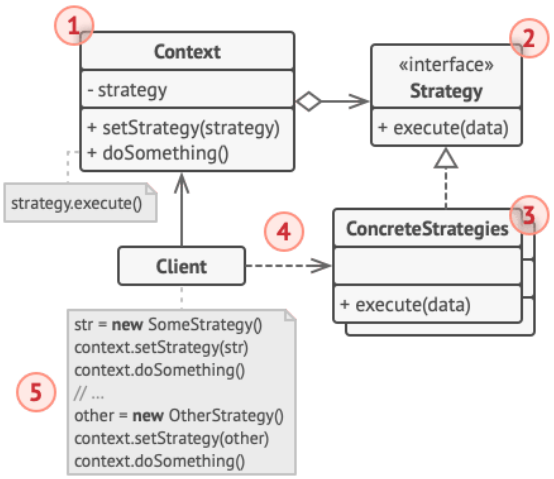


EJEMPLO:

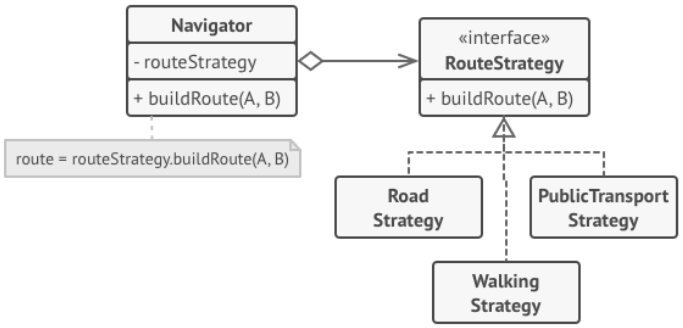
```
// The AudioPlayer class acts as a context. It also maintains a
// reference to an instance of one of the state classes that
// represents the current state of the audio player.
class AudioPlayer is
    field state: State
    field UI, volume, playlist, currentSong
    constructor AudioPlayer() is
        this.state = new ReadyState(this)
        UI = new UserInterface()
        UI.lockButton.onClick(this.clickLock)
        UI.playButton.onClick(this.clickPlay)
        UI.nextButton.onClick(this.clickNext)
        UI.prevButton.onClick(this.clickPrevious)
    method changeState(state: State) is
        this.state = state
    method clickLock() is
        state.clickLock()
    method clickPlay() is
        state.clickPlay()
    method clickNext() is
        state.clickNext()
    method clickPrevious() is
        state.clickPrevious()
    // A state may call some service methods on the context.

abstract class State is
    protected field player: AudioPlayer
    constructor State(player) is
        this.player = player
    abstract method clickLock()
    abstract method clickPlay()
    abstract method clickNext()
```

```
    abstract method clickPrevious()
// Concrete states implement various behaviors associated with a
// state of the context.
class LockedState extends State is
    method clickLock() is
        if (player.playing)
            player.changeState(new PlayingState(player))
        else
            player.changeState(new ReadyState(player))
    method clickPlay() is
        // Locked, so do nothing.
    method clickNext() is
        // Locked, so do nothing.
    method clickPrevious() is
        // Locked, so do nothing.

// They can also trigger state transitions in the context.
class ReadyState extends State is
    method clickLock() is
        player.changeState(new LockedState(player))
    method clickPlay() is
        player.startPlayback()
        player.changeState(new PlayingState(player))
    method clickNext() is
        player.nextSong()
    method clickPrevious() is
        player.previousSong()


class PlayingState extends State is
    method clickLock() is
        player.changeState(new LockedState(player))
    method clickPlay() is
        player.stopPlayback()
        player.changeState(new ReadyState(player))
    method clickNext() is
        if (event.doubleclick)
            player.nextSong()
        else
            player.fastForward(5)
    method clickPrevious() is
        if (event.doubleclick)
            player.previous()
        else
            player.rewind(5)
```

7. STRATEGY: Permite seleccionar un algoritmo o estrategia en tiempo de ejecución. En lugar de implementar un único algoritmo directamente dentro de una clase, el patrón Strategy utiliza interfaces para hacer que un conjunto de algoritmos sean intercambiables. Úsalo si tienes muchas clases similares que solo difieren en la forma en que ejecutan un comportamiento. Usar para separar la lógica del negocio de la implementación de los algoritmos. Usar cuando la clase tenga condicionales masivos que se diferencian en variantes del mismo algoritmo.
   El patrón Strategy sugiere que tomes una clase que hace algo específico de muchas maneras diferentes y extraigas todos estos algoritmos en clases separadas llamadas estrategias.
   PROS: swap algorithms used inside an object at runtime, isolate implementation details of an algorithm from the code, replace inheritance with composition, OCP.
   CONS: algorithms that rarely change it is not necessary to overcomplicate, clients must be aware of the differences.



EJEMPLO:

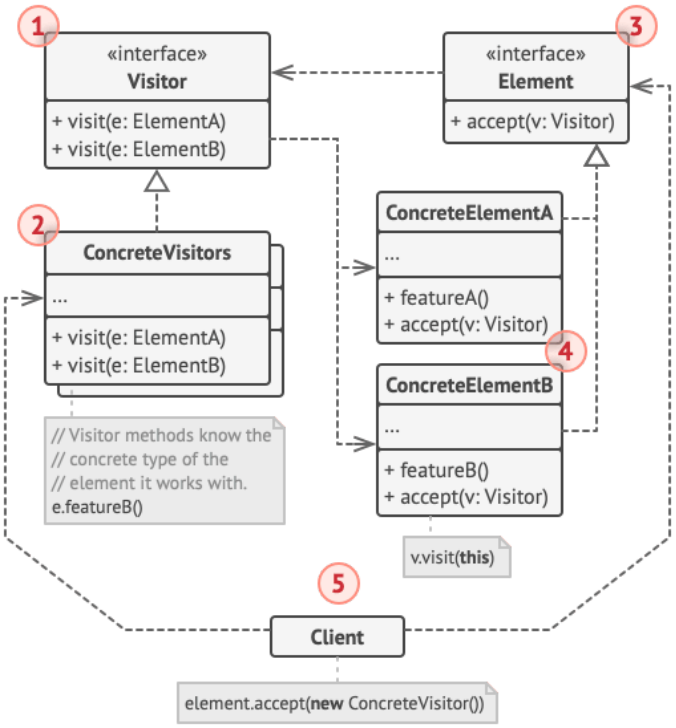



```
// The context uses this interface to call the algorithm defined
by // the concrete strategies.
interface Strategy is
    method execute(a, b)

class ConcreteStrategyAdd implements Strategy is
    method execute(a, b) is
        return a + b

class ConcreteStrategySubtract implements Strategy is
```

```
    method execute(a, b) is
        return a - b

class ConcreteStrategyMultiply implements Strategy is
    method execute(a, b) is
        return a * b

class Context is
    private strategy: Strategy
    method setStrategy(Strategy strategy) is
        this.strategy = strategy
    method executeStrategy(int a, int b) is
        return strategy.execute(a, b)

// The client should be aware of the differences
// between strategies in order to make the right choice.
class ExampleApplication is
    method main() is
        Create context object.
        Read first number.
        Read last number.
        Read the desired action from user input.
        if (action == addition) then
            context.setStrategy(new ConcreteStrategyAdd())
        if (action == subtraction) then
            context.setStrategy(new ConcreteStrategySubtract())
        if (action == multiplication) then
            context.setStrategy(new ConcreteStrategyMultiply())
        result = context.executeStrategy(First number, Second
number)
        Print result.
```

8. VISITOR: Permite separar algoritmos de los objetos sobre los que operan. Esto puede ser útil cuando necesitan realizar operaciones sobre estos objetos sin alterar sus clases. El patrón Visitor sugiere que coloques el nuevo comportamiento en una clase separada llamada Visitor, en lugar de intentar integrarlo en las clases existentes. El objeto original que debía realizar el comportamiento ahora se pasa a uno de los métodos del Visitor como argumento, proporcionando al método acceso a todos los datos necesarios contenidos en el objeto.
   En lugar de que el cliente seleccione el método adecuado, se propone que los propios objetos (sobre los que el Visitor operará) se encarguen de esta tarea. Como cada objeto conoce su propia clase, puede determinar de manera más directa qué método del Visitor debe ser ejecutado.
   Utiliza el patrón Visitor cuando tienes una estructura de objetos compleja, como un árbol de objetos, y necesitas realizar operaciones en todos sus elementos. El patrón permite que cada elemento "acepte" un visitante, que luego ejecuta la operación específica para ese tipo de elemento. Usa el patrón Visitor para separar

comportamientos adicionales o auxiliares de la lógica principal de negocio de tus clases. Evita la sobrecarga de clases con métodos innecesarios. Al extraer estos comportamientos en una clase visitante, puedes implementar solo los métodos que son relevantes para ciertas clases, manteniendo otras clases limpias y simples.
PROS: OCP, SRP, accumulate information in visitor objects while traverse complex object structure.
CONS: update every visitor each time a class gets added to or removed from the element hierarchy. Visitors might lack the necessary access to the private fields and methods of the elements that they're supposed to work with.



EJEMPLO: En Argentina todos los productos pagan IVA. Algunos productos poseen una tasa reducida. Utilizaremos el Visitor para solucionar este problema.

```
public interface Visitable {

    public double accept(Visitor visitor);

}
public class ProductoDescuento implements Visitable {
    private double precio;

    public double accept(Visitor visitor) {
        return visitor.visit(this);
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
```

```java
public class ProductoNormal implements Visitable {
    private double precio;

    public double accept(Visitor visitor) {
        return visitor.visit(this);
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }
}
public interface Visitor {
    public double visit(ProductoNormal normal);
    public double visit(ProductoDescuento reducido);

}
public class IVA implements Visitor {
    private final double impuestoNormal = 1.21;
    private final double impuestoReducido = 1.105;

    public double visit(ProductoNormal normal) {
        return normal.getPrecio() * impuestoNormal;
    }

    public double visit(ProductoDescuento reducido) {
        return reducido.getPrecio() * impuestoReducido;
    }

}
```

```java
public static void main(String[] args) {
    ProductoDescuento producto1 = new ProductoDescuento();
    producto1.setPrecio(100);
    ProductoNormal producto2 = new ProductoNormal();
    producto2.setPrecio(100);

    IVA iva = new IVA();
    double resultado1 = producto1.accept(iva);
    double resultado2 = producto2.accept(iva);

    System.out.println(resultado1);
    System.out.println(resultado2);

}
```

```
class Television {
        private int volume = 10;

        public void VolumeUp() {
                if (isOn) {
                        volume++;
                        Console.WriteLine($"Volumen: {volumen}");
                }
        }

        public void VolumeDown() {
                if (isOn) {
```

```
                        volume--;
                        Console.WriteLine($"Volumen: {volumen}");
                }
        }
}

abstract class Command {
        protected Television television;

        public Command(Television television) {
                this.television = television;
        }

        public abstract void Execute();
}

class VolumeUpCommand extends Command {
        public override void Execute() {
                television.VolumeUp();
        }
}

class VolumeDownCommandoextends Command {
        public override void Execute() {
                television.VolumeDown();
        }
}

class CommandHistory {
        private Stack<Command> history;

        public CommandHistory() {
                histry = new Stack<Command>();
        }

        public void Push(Command command) {
                history.push(command);
        }

        public void Pop() {
                history.pop();
        }
}
```

```
class Program {
	void ExecuteCommand(Command command, CommandHistory history) {
		command.Execute();
		history.Push(command);
	}

	static void Main() {
		Television television = new Television();
		Command turnOn = new TurnOnCommand(television);
		Command turnOff = new TurnOffCommand(television);
		Command volumeUp = new VolumeUpCommand(television);
		Command volumeDown = new VolumeDownCommand(television);
		CommandHistory history = new CommandHistory();

		string input = "";
		while (input != "exit") {
			Console.WriteLine("Escribe 'on' para encender, 'off' para apagar,
'volumeup' para subir volumen, 'volumedown' para bajar volumen, 'exit' para salir");
			input = Console.ReadLine();
			switch (input) {
				case "on":
					ExecuteCommand(turnOn, history);
					break;
				case "off":
					ExecuteCommand(turnOff, history);
					break;
				case "volumeup":
					ExecuteCommand(volumeUp, history);
					break;
				case "volumeDown":
					ExecuteCommand(volumeDown, history);

					break;
			}
		}
	}
}
```