# Patrones de Diseño

Juan Pablo Sandoval

# *Patrones de que veremos*

- *Adapter*
- *Proxy*
- *Singleton*

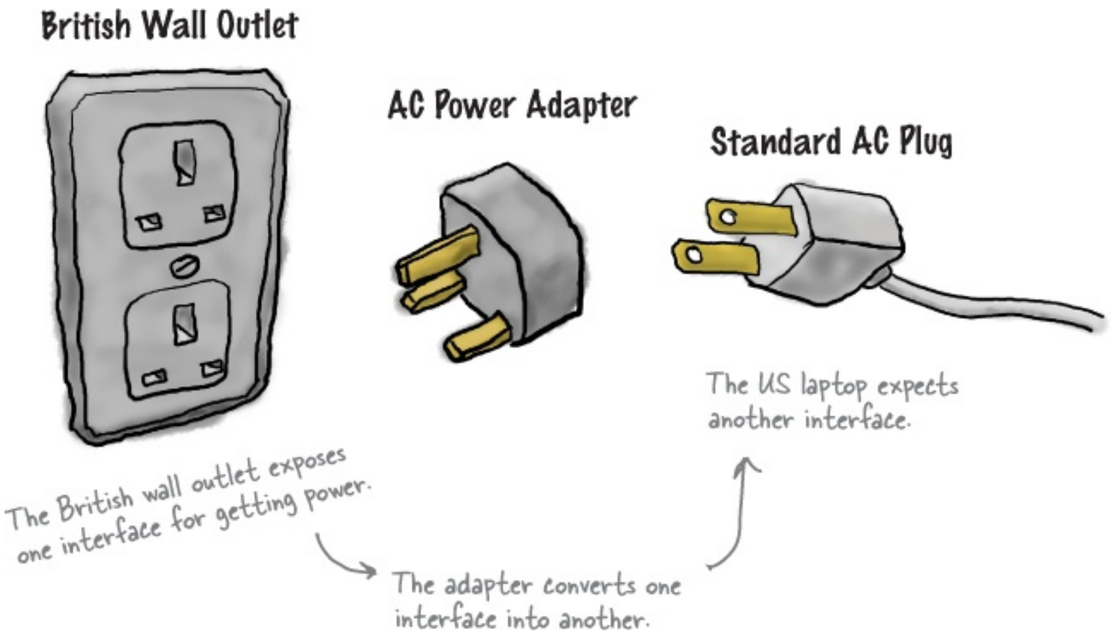
# *Adapter*

*Adapter es un patrón de diseño structural que permite que dos objetos con interfaces (métodos) incompatibles colaboren.*

*La idea general es que existen dos clases, A y B, la clase A espera que la clase B tenga un método (o mas) muy particular. Ya que la clase B no tiene ese método (aunque tenga uno muy parecido) se dice que A y B son incompatibles.*

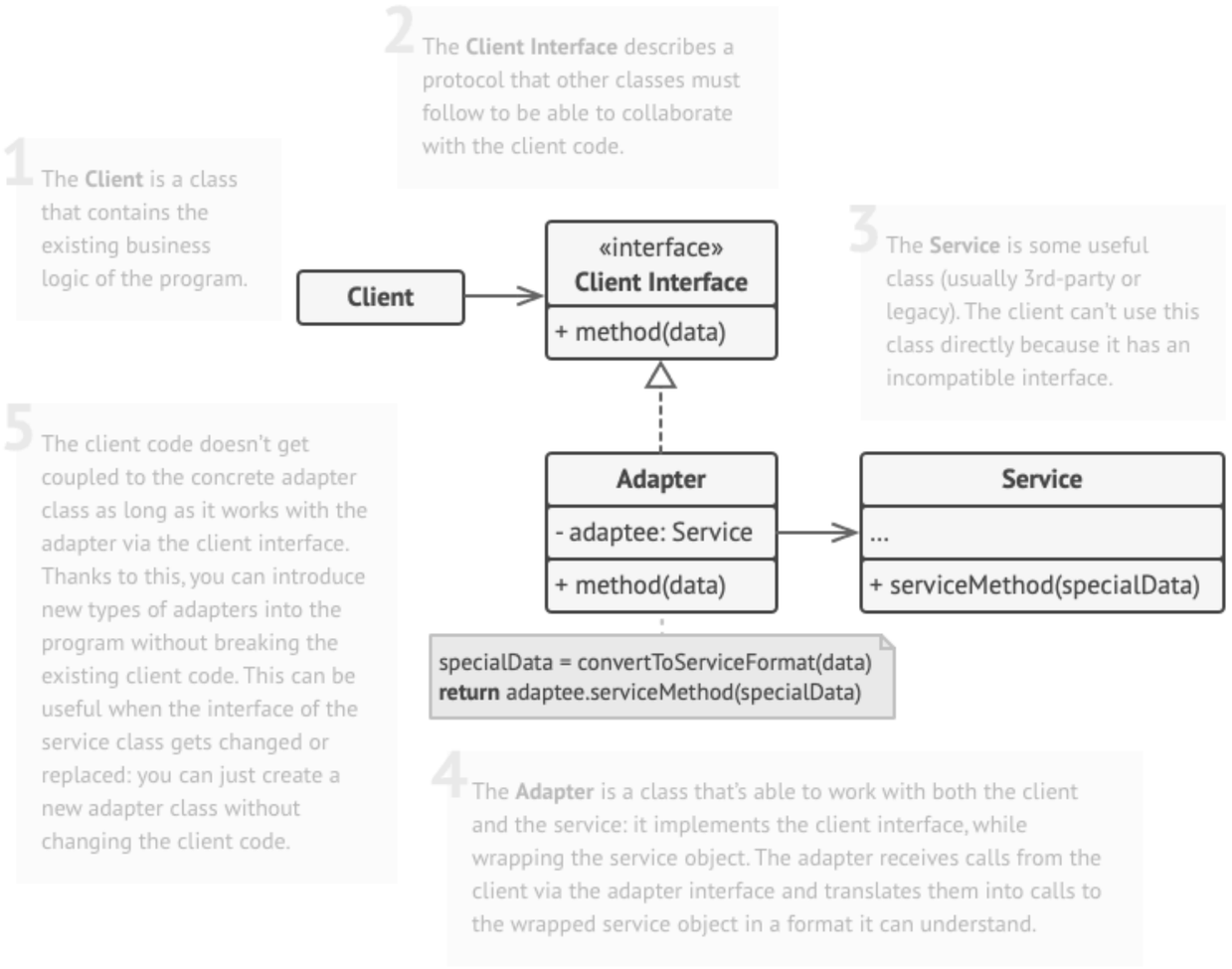
*Otra forma de incompatibilidad es que la clase A quiera mandar un dato y la clase B espere que mande otro dato.*

*La idea es una tercera clase C, el adaptador, la clase C tendrá el método que la clase A espera, y C interactuara con la clase B con los métodos que tenga. De esta forma no sera necesario modificar la clase A ni la clase B. El adaptador tendrá los métodos necesarios para que puedan enviarse mensajes a través de el.*




https://refactoring.guru/design-patterns/adapter

# *Adapter - meta estructura*




https://refactoring.guru/design-patterns/adapter

```ruby
class ConsoleClient
    def initialize(service)
        @service = service
    end
    def run
        rut = "22.456.456-2"
        clave = "secreta"
        if @service.login(rut,clave) then
            puts "autenticacion exitosa"
        else
            puts "autenticacion fallida"
        end
    end
end
```

```ruby
class LoginService
    def initialize()
        @users = { "22456456-2" => "secreta",
                   "11789789-3" => "password",
                   "33456123-4" => "clave" }
    end
    def login_user(rut, pass)
        return @users[rut.strip] == pass.strip
    end
end
```

```ruby
service = LoginService.new
client = ConsoleClient.new(service)
client.run
```

*En este ejemplo las interfaces son incompatibles, el servicio espera que se mande el RUT sin puntos, por lo que las interfaces son incompatibles. La pregunta es,* ***como podemos hacer que el código anterior funcione sin modificar el código de ninguna clase?***

```ruby
class ConsoleClient
    def initialize(service)
        @service = service
    end
    def run
        rut = "22.456.456-2"
        clave = "secreta"
        if @service.login(rut,clave) then
            puts "autenticacion exitosa"
        else
            puts "autenticacion fallida"
        end
    end
end

class LoginServiceAdapter
    def initialize(originalService)
        @originalService = originalService
    end
    def login(rut, pass)
        adaptedrut = rut.gsub(".", "")
        return   @originalService.login_user(adaptedrut, pass)
    end
end
```

```ruby
service = LoginService.new
adaptador = LoginServiceAdapter.new(service)
client = ConsoleClient.new(adaptador)
client.run
```

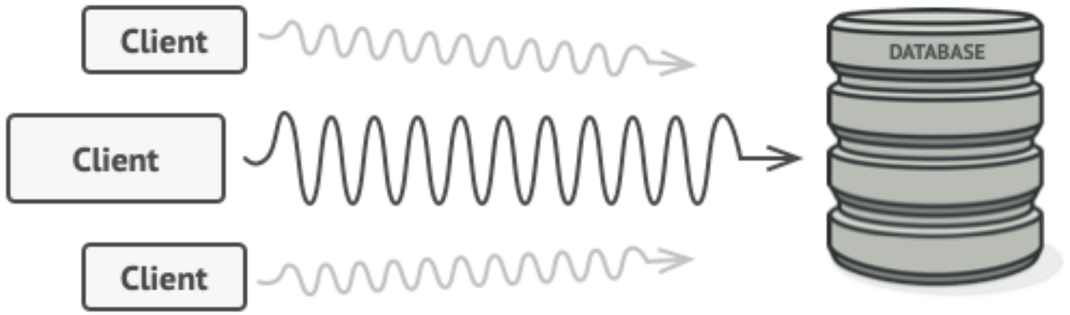
*Al cliente le mandamos el adaptador, como el adaptador tiene los métodos tal cual necesita el cliente, el cliente no logra diferenciar si esta ejecutando el adaptador o el servicio original*

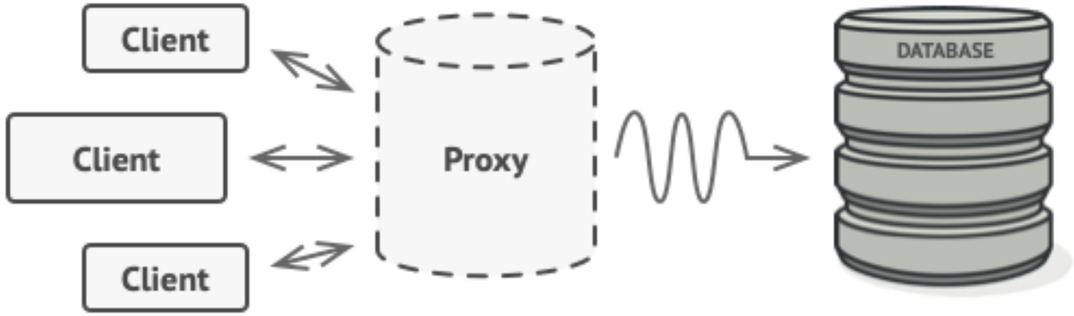
# *Proxy*

*Proxy es un patron de diseño estructural que permite proporcionar un substituto al objeto original. El objeto proxy tiene la referencia al objeto original.*

*La idea general es que los clientes puedan utilizar al objeto proxy o al original de forma indistinta. El objeto proxy para responder a las llamadas que le hacen siempre consulta al objeto original.*

***Para que sirve entonces?** La ventaja es que el objeto proxy puede realizar varias operaciones antes y después de hacer la llamada al original. Asimismo, es posible filtrar las llamadas al objeto original entre otras operaciones utiles.*

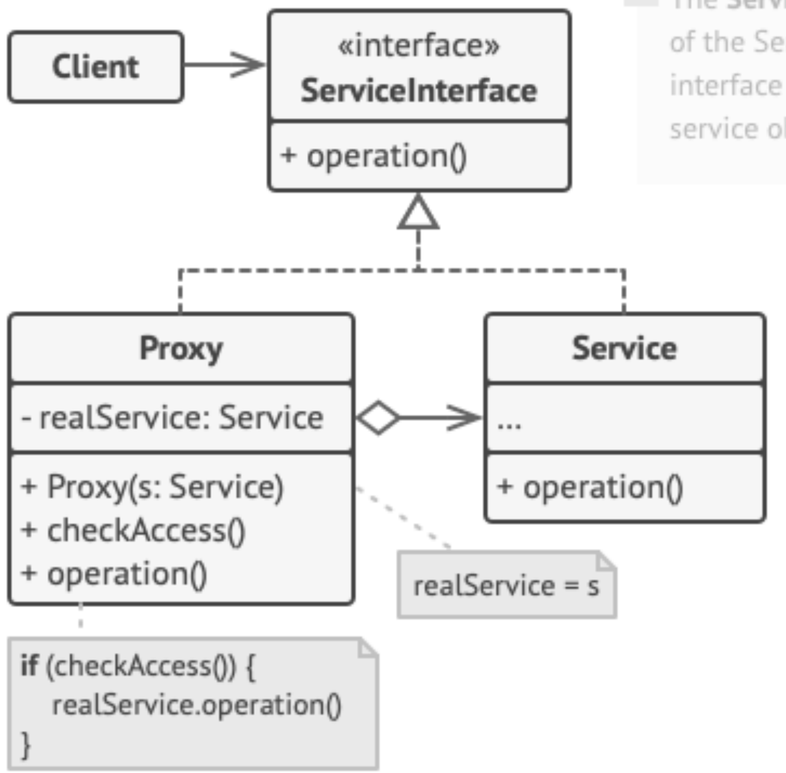



https://refactoring.guru/design-patterns/proxy

# Proxy

4 The **Client** should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.




1 The **Service Interface** declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.

2 The **Service** is a class that provides some useful business logic.

3 The **Proxy** class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object.

Usually, proxies manage the full lifecycle of their service objects.

https://refactoring.guru/design-patterns/proxy

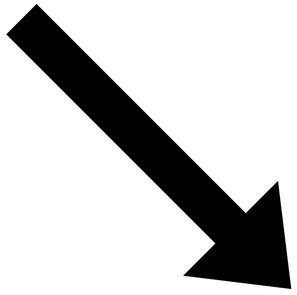
```ruby
class ConsoleClient
    def initialize(service)
        @service = service
    end
    def run
        rut = "22456456-2"
        clave = "secreta"
        if @service.login(rut,clave) then
            puts "autenticacion exitosa"
        else
            puts "autenticacion fallida"
        end
    end
end
```

```ruby
class LoginService
    def initialize()
        @users = { "22456456-2" => "secreta",
                   "11789789-3" => "password",
                   "33456123-4" => "clave" }
    end
    def login_user(rut, pass)
        return @users[rut.strip] == pass.strip
    end
end
```

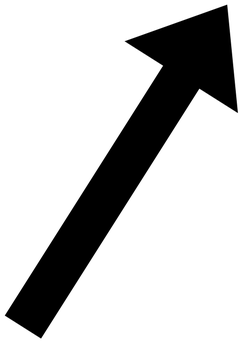
```ruby
service = LoginService.new
client = ConsoleClient.new(service)
client.run
```

*Consideré el ejemplo anterior, supongamos que ahora los métodos del cliente y del servicio de login son compatibles. El cliente envía los datos como corresponde y el servicio tiene los métodos tal cual el cliente espera.*
*Sin embargo, ahora queremos imprimir un mensaje en con sola (Log) cada ves que un cliente hace un login fallido.* ***Como podemos agregar esta funcionalidad sin tocar ninguna de las dos clases?***

```ruby
class ConsoleClient
    def initialize(service)
        @service = service
    end
    def run
        rut = "22456456-2"
        clave = "secreta"
        if @service.login(rut,clave) then
            puts "autenticacion exitosa"
        else
            puts "autenticacion fallida"
        end
    end
end
```



```ruby
class LoginProxy
    def initialize(objetoOriginal)
        @objetoOriginal = objetoOriginal
    end
    def login(rut,pass)
        result = @objetoOriginal.login(rut,pass)
        if result == false then
            puts "failed login attempt #{rut}"
        end
        return result
    end
end
```

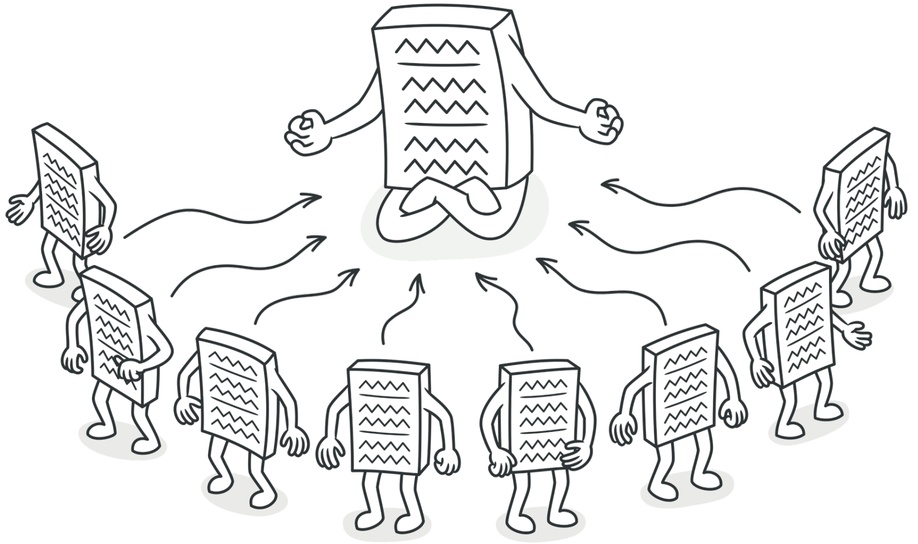


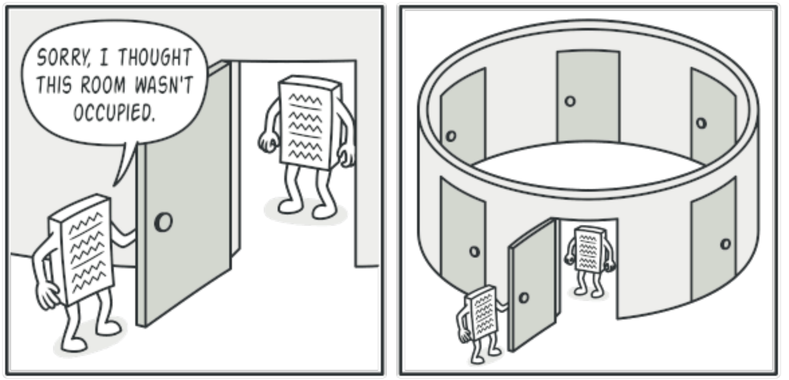
```ruby
class LoginService
    def initialize()
        @users = { "22456456-2" => "secreta2",
            "11789789-3" => "password",
            "33456123-4" => "clave" }
    end
    def login(rut, pass)
        return @users[rut.strip] == pass.strip
    end
end
```

# *Singleton*

*La motivación detrás del patrón singleton, es tratar de controlar el numero de instancias que se crean de una clase. El patrón singleton limita el numero de instancias a crear a solo una.*

*A veces es necesario tener un solo objeto de una clase el cual pueda ser consultado en diferentes partes del programa. Es decir el patron singleton permite crear un objeto de acceso global en el sistema, lo cual es conveniente en ciertas situaciones.*





# *Singleton - Ejemplo*

*El siguiente código modela a un folder principal en un sistema operativo, the root folder.*

*Note que la segunda linea hace el método de clase "new" sea privado, lo que significa, que el "new" no puede ser llamado de otro lugar que no sea la misma clase. Esto permite que no se puedan crear objetos de esta clase con el operador "new".*

*Para que otra clases accedan al único objeto de esta clase creamos el método de clase "instance". Que es publico y controla que nose cree mas de una instancia de la clase.*

*Note que el "new" se llama dentro de este método esto solo es posible pq el método en cuestión esta dentro de la misma clase.*

```ruby
class MainFolder
  private_class_method :new
  def initialize
    @name = "root"
  end
  def self.instance
    if @instance == nil
      @instance = new
    end
    return @instance
  end
end
folder = MainFolder.instance
```