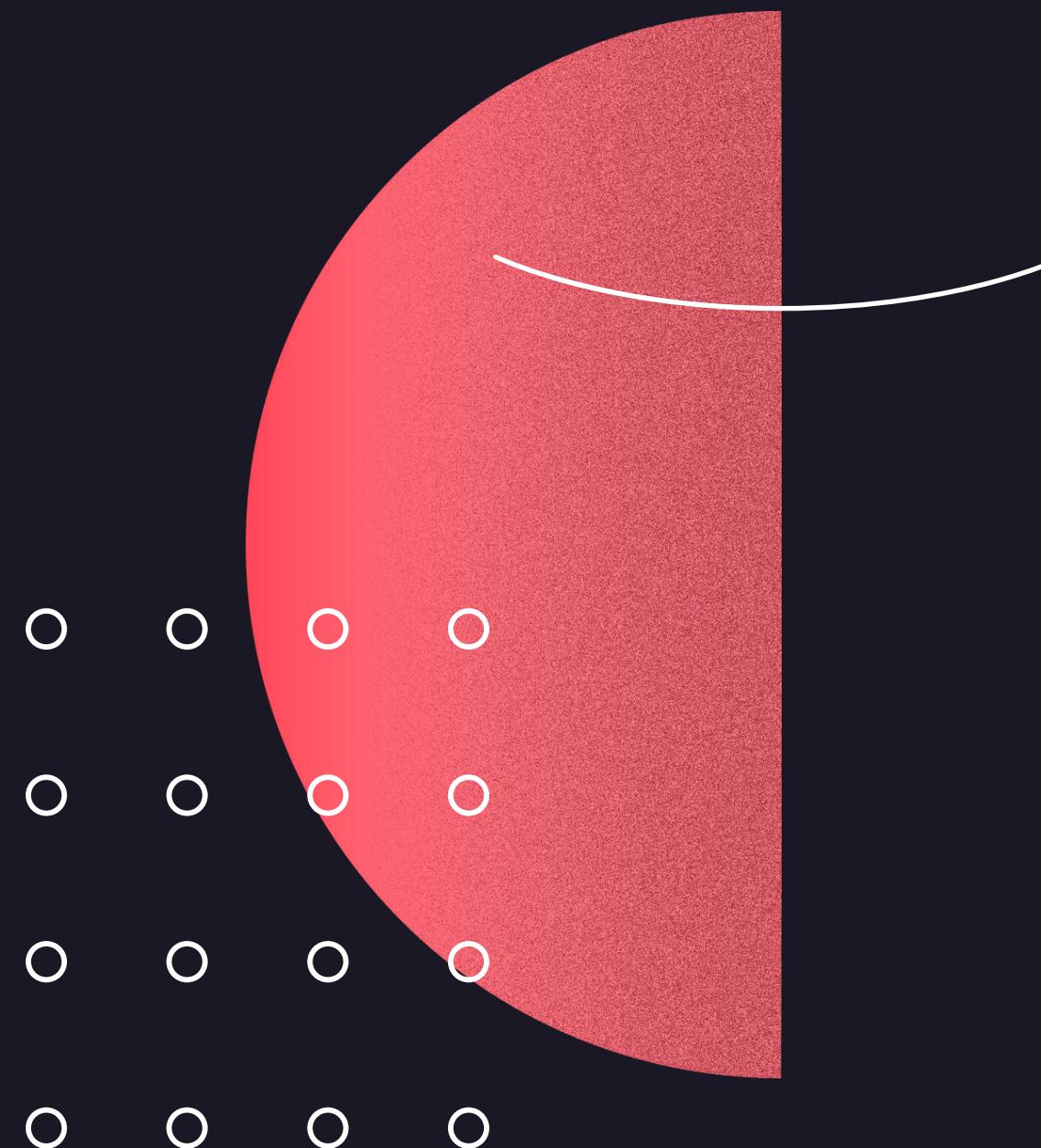




# ADAPTER/WRAPPER

Esteban Castañeda Blanco C01795  
Daniel Lizano Morales C04285



# Patrón de diseño estructural

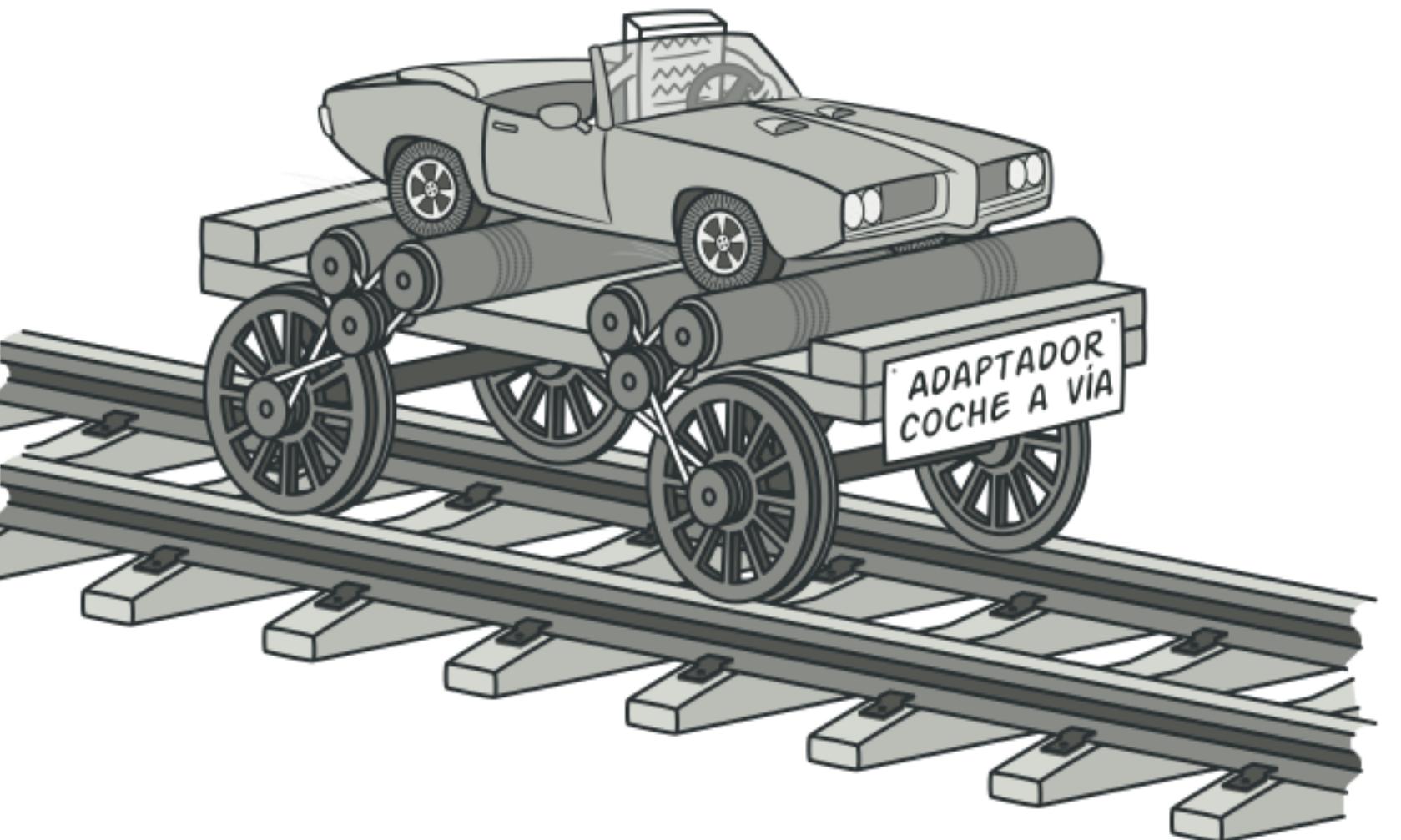
Los patrones estructurales explican el cómo ensamblar objetos y clases en estructuras más grandes, manteniendo la flexibilidad y eficiencia. Buscan resolver los problemas que requieren la composición de objetos

# CONCEPTO Y PROPÓSITO

Patrón que convierte la interfaz de una clase a otra interfaz que compatible con la estructura del proyecto

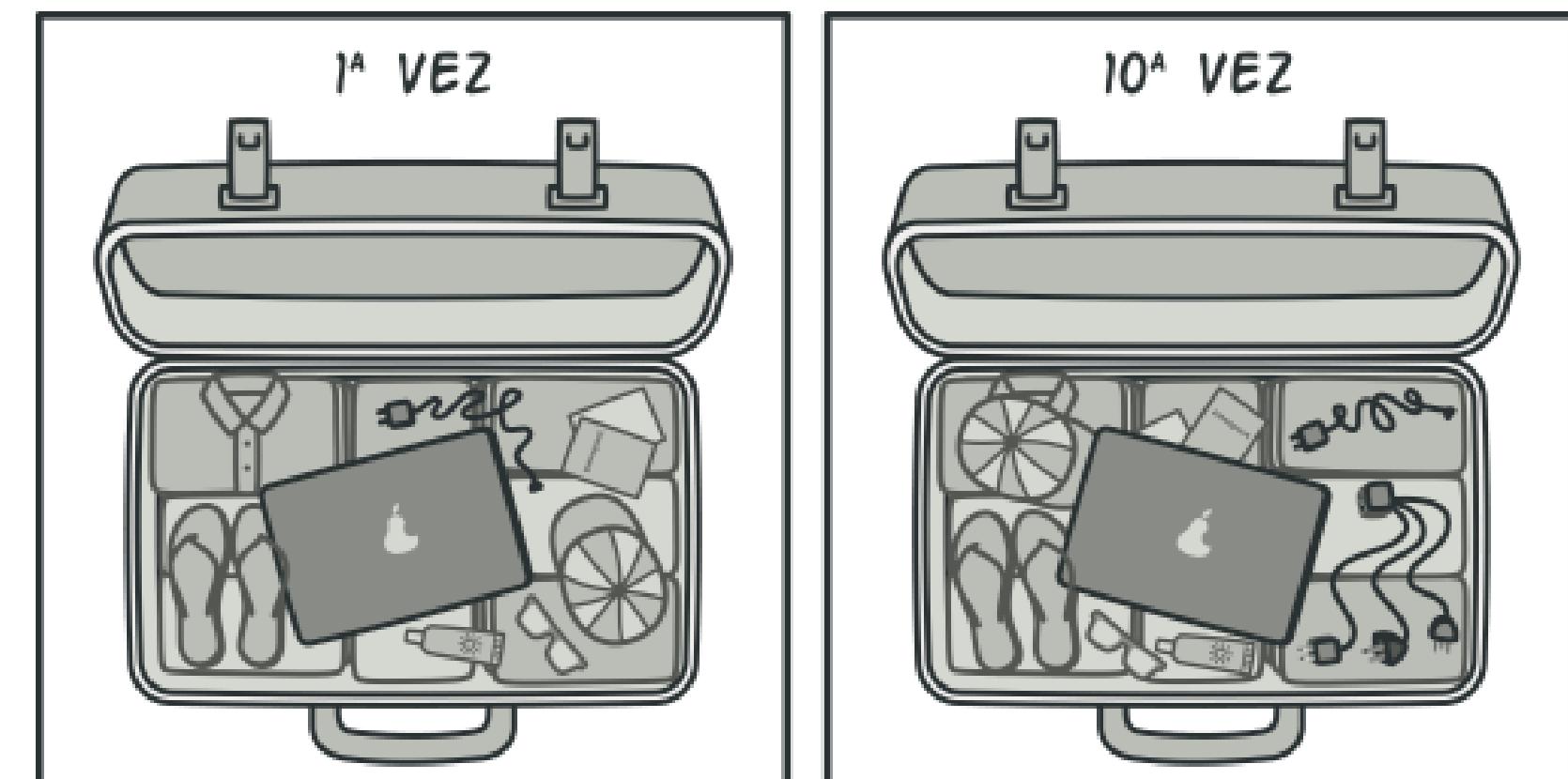
Involucra una sola clase que es responsable de unir funcionalidades de interfaces independientes o incompatibles, combiendo la capacidad de dos interfaces independientes

Tiene como principal propósito permitir la colaboración entre objetos que poseen interfaces incompatibles



# Analogía del viaje al extranjero

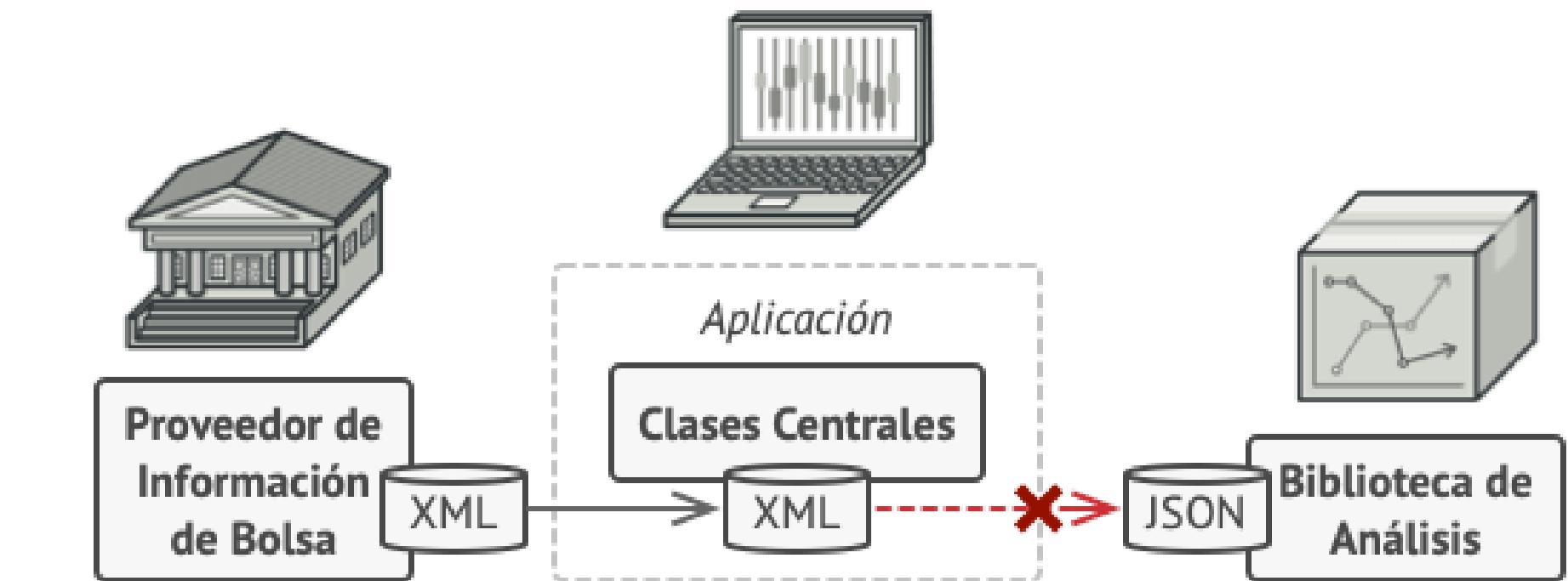
VIAJAR AL EXTRANJERO



*Una maleta antes y después de un viaje al extranjero.*

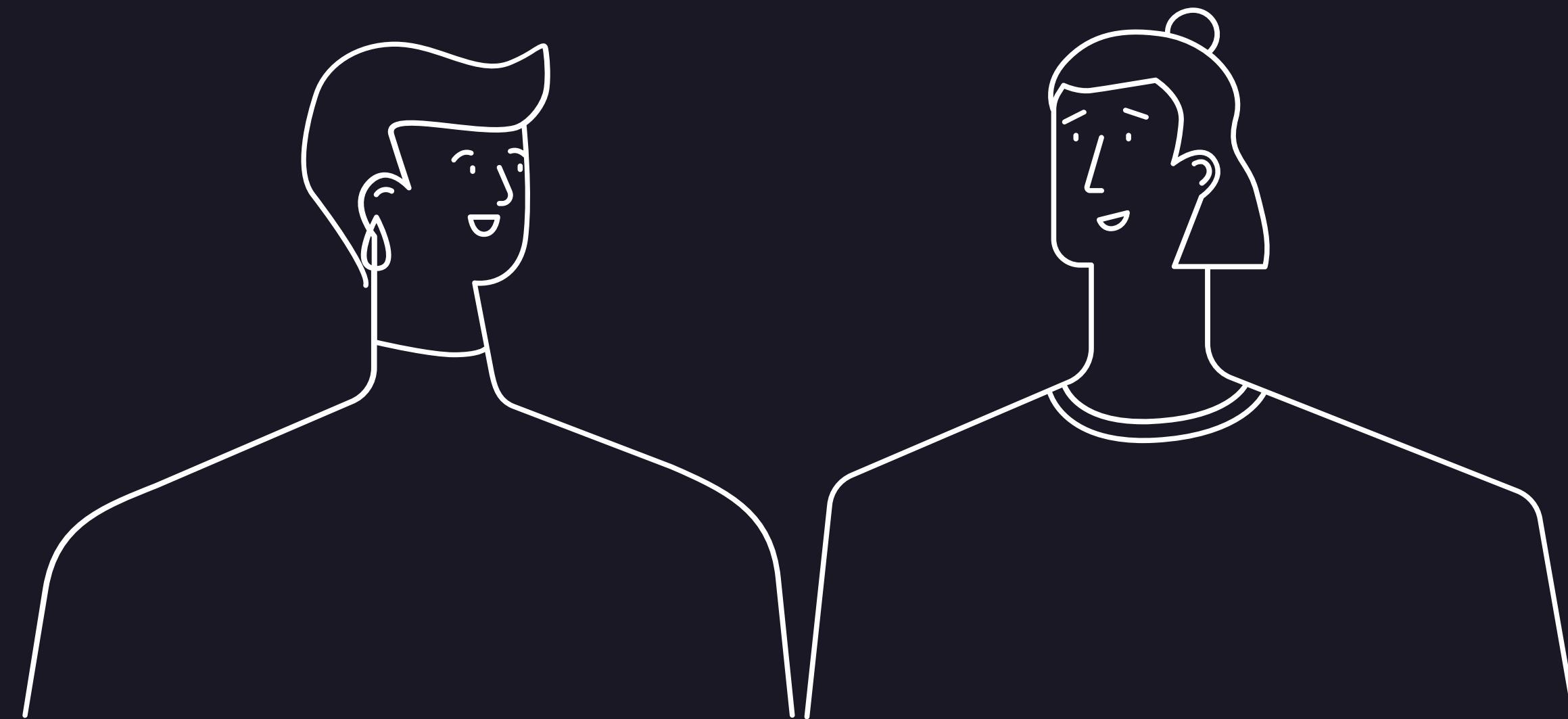
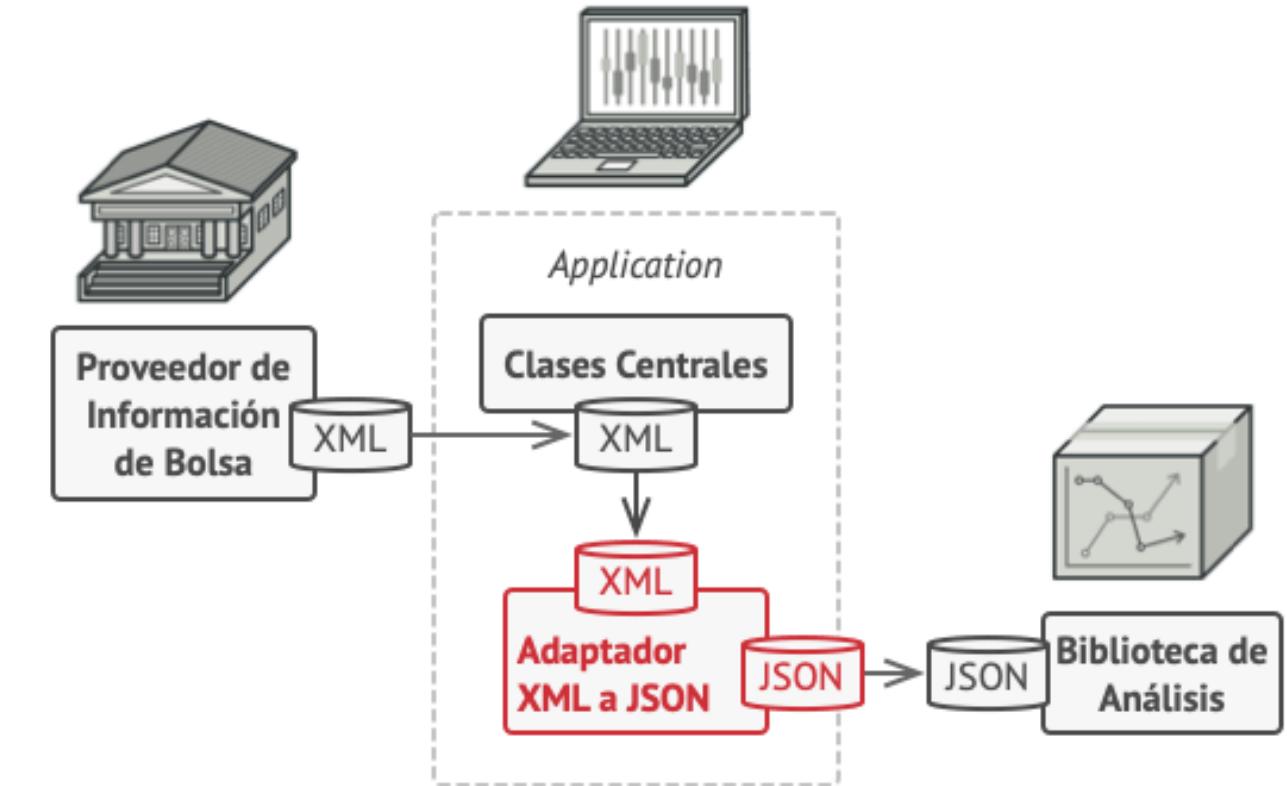
# PROBLEMA

Implementar una biblioteca que utiliza el formato JSON en un sistema que utiliza XML



# SOLUCIÓN

Crear adaptadores de XML a JSON para cada clase de la biblioteca de análisis con la que trabaje tu código directamente. Después ajustas tu código para que se comunique con la biblioteca únicamente a través de estos adaptadores



# VENTAJAS

## Principio de responsabilidad única

Se puede separar la interfaz o el código de conversión de datos de la lógica de negocio primaria del programa.

## Principio de abierto/cerrado

Se puede introducir nuevos tipos de adaptadores al programa sin descomponer el código cliente existente, siempre y cuando trabajen con los adaptadores a través de la interfaz con el cliente



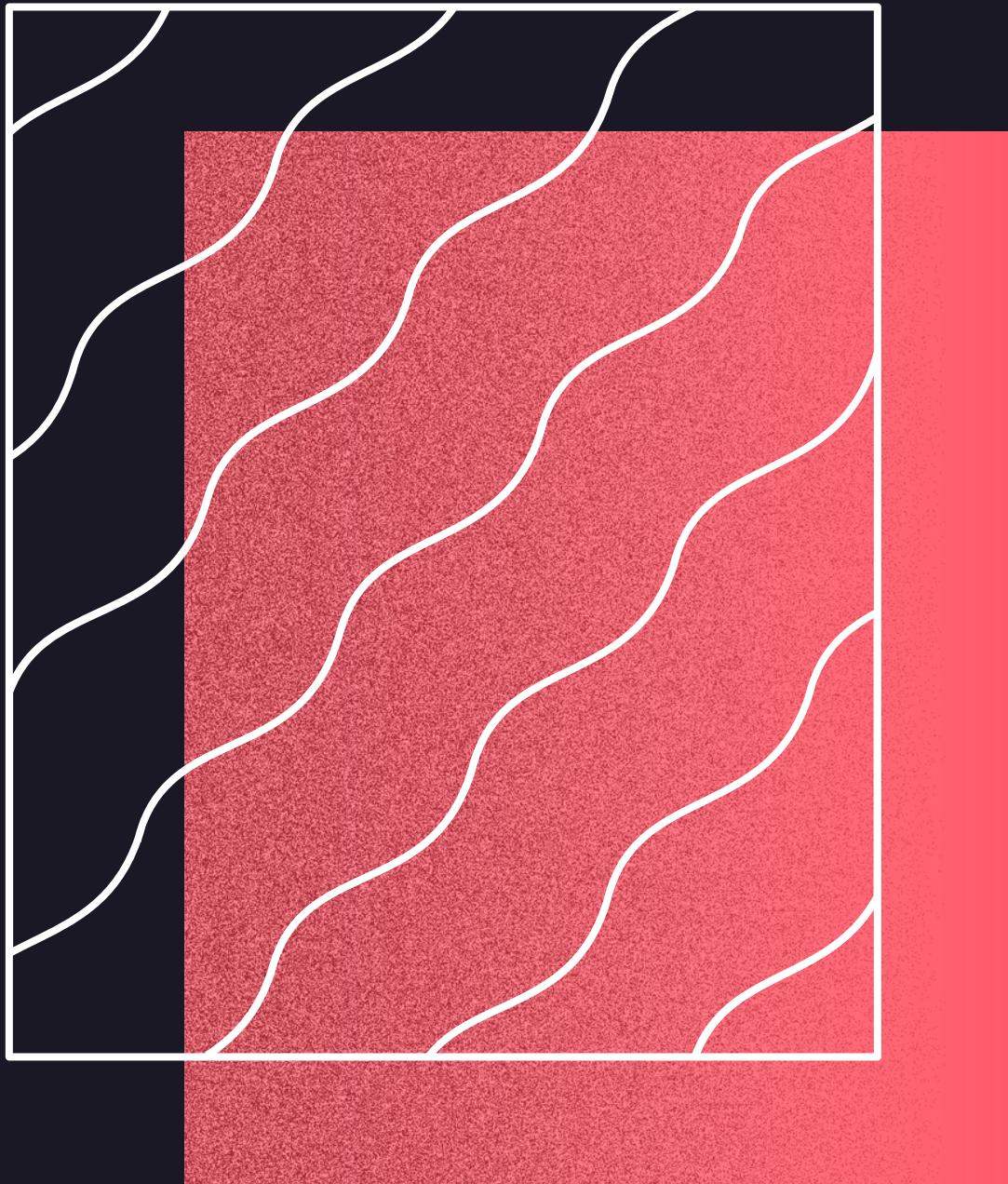
# DESVENTAJAS

## Overhead saturado

Al ser todas las request redirigidas se causa un crecimiento en el overhead del programa.

## Varias adaptaciones

Dependiendo de los requerimientos del sistema, varias adaptaciones pueden ser requeridas por lo que se debe de tener una cadena de adaptación larga.



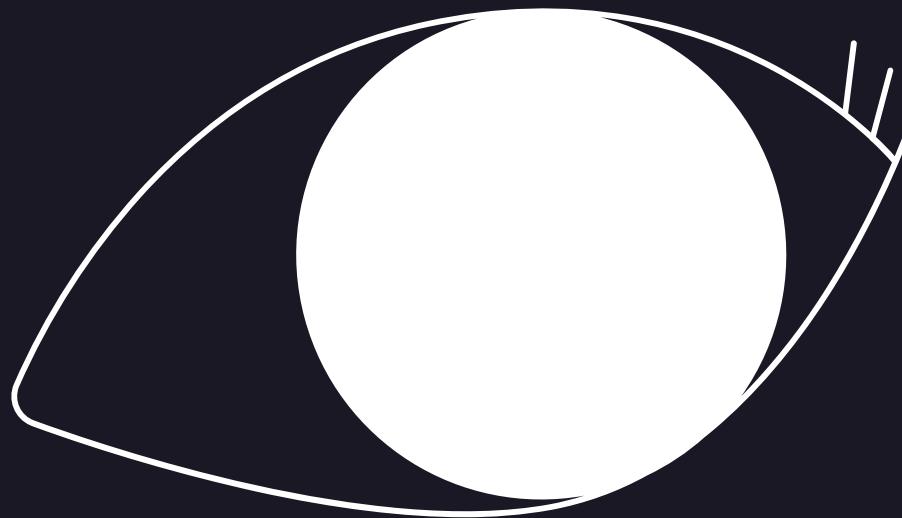
# ¿CUANDO APLICARLO?

- Cuando se requiera utilizar una clase existente pero su interfaz no es compatible con el resto del código.
- Cuando se desee reutilizar varias subclases existentes que carecen de alguna funcionalidad común que no se puede agregar a la superclase.

# IMPLEMENTACIÓN



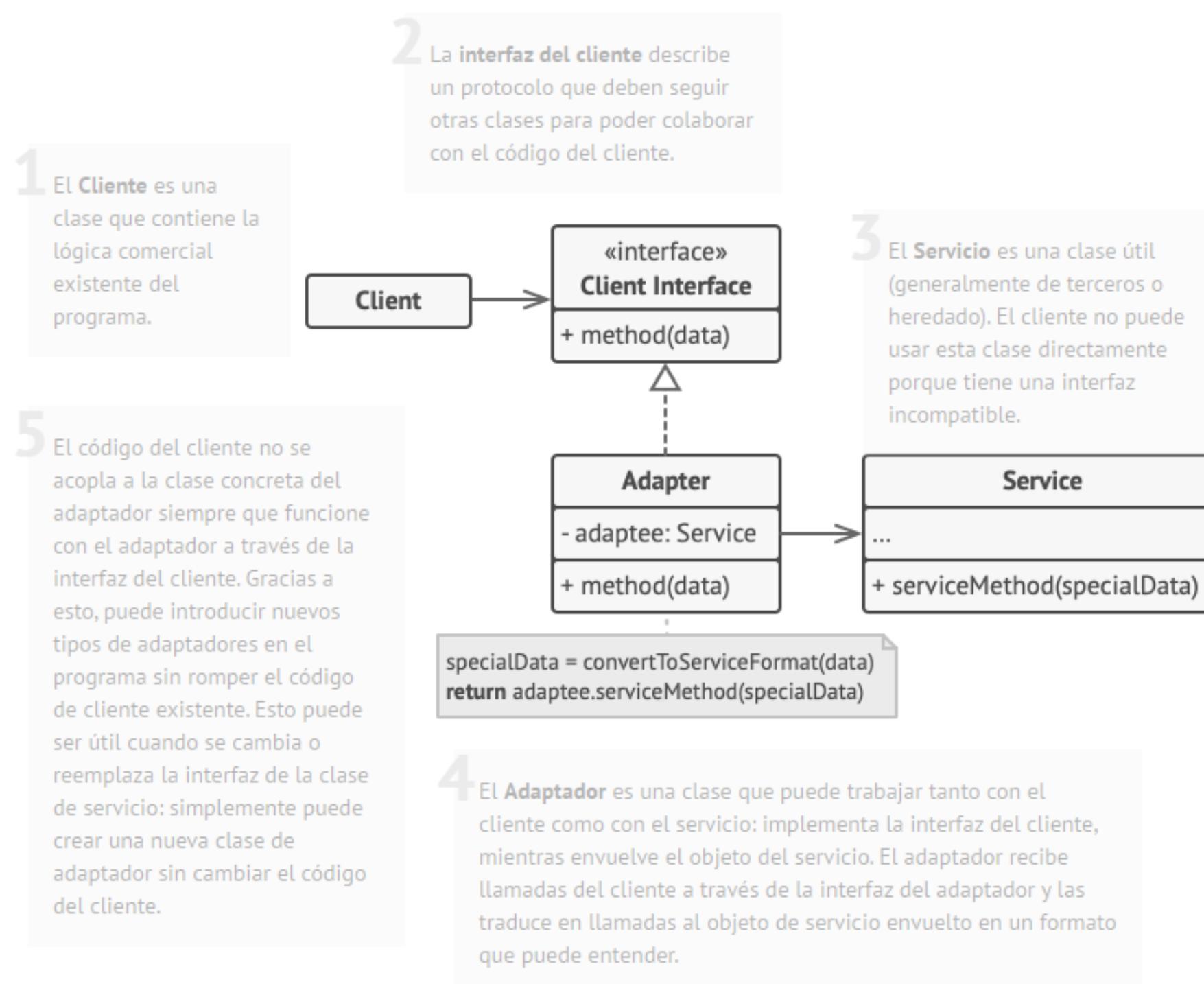
1. Asegurarse de tener al menos dos clases con interfaces incompatibles.
2. Declarar la interfaz del cliente y describir cómo los clientes se comunican con el servicio.
3. Crear la clase de adaptador y hacer que siga la interfaz del cliente.
4. Agregar un campo a la clase de adaptador para almacenar una referencia al objeto de servicio.
5. Implementar los métodos de la interfaz del cliente en la clase de adaptador. El adaptador debe delegar la mayor parte del trabajo real al objeto de servicio, manejando solo la interfaz o la conversión de formato de datos.
6. Los clientes deben usar el adaptador a través de la interfaz del cliente. Esto le permitirá cambiar o extender los adaptadores sin afectar el código del cliente.



# TIPOS DE IMPLEMENTACIONES

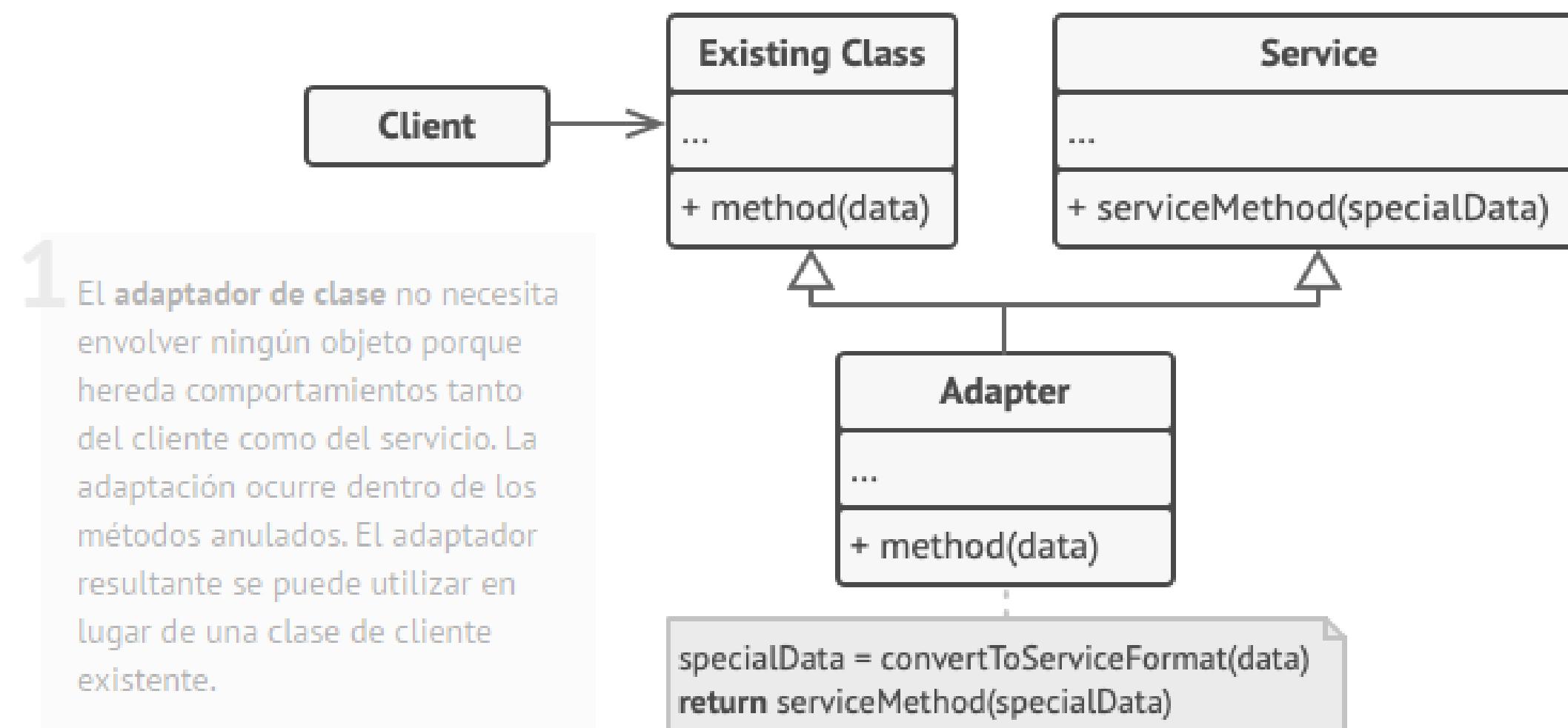
# ADAPTADOR DE OBJETO

Esta implementación utiliza el principio de composición de objetos: el adaptador implementa la interfaz de un objeto y envuelve el otro. Se puede implementar en todos los lenguajes de programación populares.



# ADAPTADOR DE CLASE

Esta implementación usa la herencia: el adaptador hereda las interfaces de ambos objetos al mismo tiempo. Tenga en cuenta que este enfoque solo se puede implementar en lenguajes de programación que admitan la herencia múltiple, como C++.



# EJEMPLO DE CÓDIGO

Clases principales

```
class Bird {
public:
    virtual void fly() = 0;
    virtual void makeSound() = 0;
};

class Sparrow : public Bird {
public:
    void fly() override {
        std::cout << "Flying" << std::endl;
    }
    void makeSound() override {
        std::cout << "Chirp Chirp" << std::endl;
    }
};

class ToyDuck {
public:
    virtual void squeak() = 0;
};

class PlasticToyDuck : public ToyDuck {
public:
    void squeak() override {
        std::cout << "Squeak" << std::endl;
    }
};
```

```
class BirdAdapter : public ToyDuck {
public:
    BirdAdapter(Bird &bird) : bird_(bird) { }
    void squeak() override {
        bird_.makeSound();
    }

private:
    Bird &bird_;
};

int main() {
    Sparrow sparrow;
    PlasticToyDuck toyDuck;
    ToyDuck &birdAdapter = *(new BirdAdapter(sparrow));

    std::cout << "Sparrow..." << std::endl;
    sparrow.fly();
    sparrow.makeSound();
    std::cout << "ToyDuck..." << std::endl;
    toyDuck.squeak();
    std::cout << "BirdAdapter..." << std::endl;
    birdAdapter.squeak();
    return 0;
}
```

# CONSECUENCIAS

Se logra la interoperabilidad entre sistemas o componentes cuyas interfaces no son compatibles

Se logra hacer que el código sea flexible y reutilizable ya los objetos existentes se pueden aplicar a contextos nuevos sin necesidad de modificarlos

Puede disminuir el rendimiento del sistema porque el uso de adaptadores puede generar una sobrecarga en el código

Vuelve el código más complejo lo que dificulta su mantenibilidad



# PATRONES RELACIONADOS

- Bridge
- Decorator
- Composite
- Facade

# Referencias

Adapter. (2022, enero 1). Refactoring.Guru. <https://refactoring.guru/design-patterns/adapter>

Design Patterns - Adapter Pattern. (s/f). Tutorialspoint.com. Recuperado de [https://www.tutorialspoint.com/design\\_pattern/adapter\\_pattern.html](https://www.tutorialspoint.com/design_pattern/adapter_pattern.html)

Kean, K. (2023, enero 29). What is the adapter design pattern and how can you use it? MUO. <https://www.makeuseof.com/adapter-design-pattern-what-how-use/>

<https://medium.com/swlh/adapter-pattern-what-it-is-and-how-to-use-it-83e35a02e7f9>

# ACTIVIDAD