UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Tópicos Especiales de Programación

Monadas y Category Theory

Elementos de la POO

Interfaces vs. Clases Abstractas

Una interfaz es un como un "acuerdo escrito" de qué mensajes (métodos) entiende un objeto, sin dictar cómo procesarlos. No tiene estado (datos).

Desacoplamos el "qué" del "cómo".

Piensa en un control remoto universal. La interfaz define botones como "encender", "cambiar canal", "subir volumen", pero no importa si controlas un TV Samsung, LG o Sony. Cada dispositivo implementa estas operaciones de manera diferente internamente, pero todos respetan el mismo contrato.

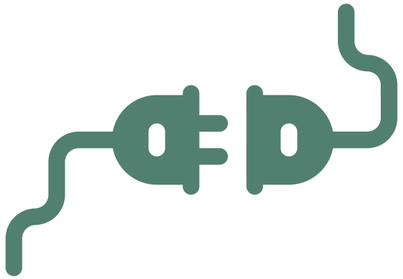
Imagina que estás diseñando una aplicación que necesita guardar datos. Puedes guardar en:

Base de datos SQL
Base de datos NoSQL
Sistema de archivos
Nube (AWS, Azure, etc.)

Sin Interfaz, cada vez que cambies el medio de almacenamiento, debes modificar todo el código que guarda datos.

Con Interfaz

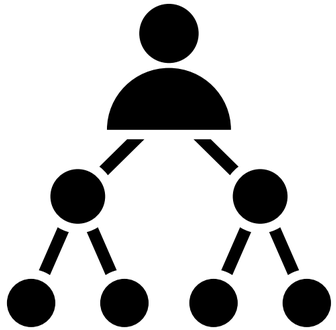
- El código que usa el repositorio no cambia si cambias de SQL a MongoDB
- Puedes probar tu aplicación con un RepositorioEnMemoria sin tocar bases de datos reales
- Diferentes partes del sistema pueden usar diferentes implementaciones sin conflictos
- Principio de Inversión de Dependencias: El código depende de abstracciones, no de implementaciones concretas



La **herencia** es un mecanismo que permite crear nuevas clases basadas en clases existentes, heredando sus atributos y métodos. La clase hija extiende a la clase padre.

Piensa en biología, los mamíferos heredan características de los vertebrados (columna vertebral, sistema nervioso central), pero agregan características propias (glándulas mamarias, pelo). Los humanos, a su vez, heredan de los mamíferos pero agregan características específicas.

Un Perro **ES UN** Animal
Un Estudiante **ES UNA** Persona



Mientras que...

Un Auto **TIENE UN** Motor → Esto es composición
Una Casa **TIENE** Habitaciones → Esto es composición

Composición vs Herencia

La composición es construir objetos complejos conteniendo instancias de otros objetos más simples. En lugar de heredar, el objeto delega responsabilidades a sus componentes.

Piensa en una computadora, esta no "hereda" de un procesador. Una computadora tiene un procesador, tiene memoria RAM, tiene un disco duro. Cada componente es independiente y puede ser reemplazado.

Un Auto **TIENE UN** Motor
Una Casa **TIENE** Habitaciones
Un Estudiante **TIENE UNA** Dirección

Ustedes que están viendo Bases de datos, pensar de esta manera puede ser util.

No todo puede resolverse con herencia

```
Vehiculo
├── VehiculoElectrico
│   ├── AutoElectrico
│   └── MotoElectrica
└── VehiculoCombustion
    ├── AutoGasolina
    └── MotoGasolina
```

¿Qué pasa con un auto híbrido? ¿Hereda de VehiculoElectrico o VehiculoCombustion?

Este es un caso para la Composición

```
Automovil
├── tiene → Motor (puede ser: Electrico, Gasolina, Diesel, Hibrido)
├── tiene → Transmision (puede ser: Manual, Automatica, CVT)
├── tiene → SistemaAudio
└── tiene → SistemaSeguiridad
```

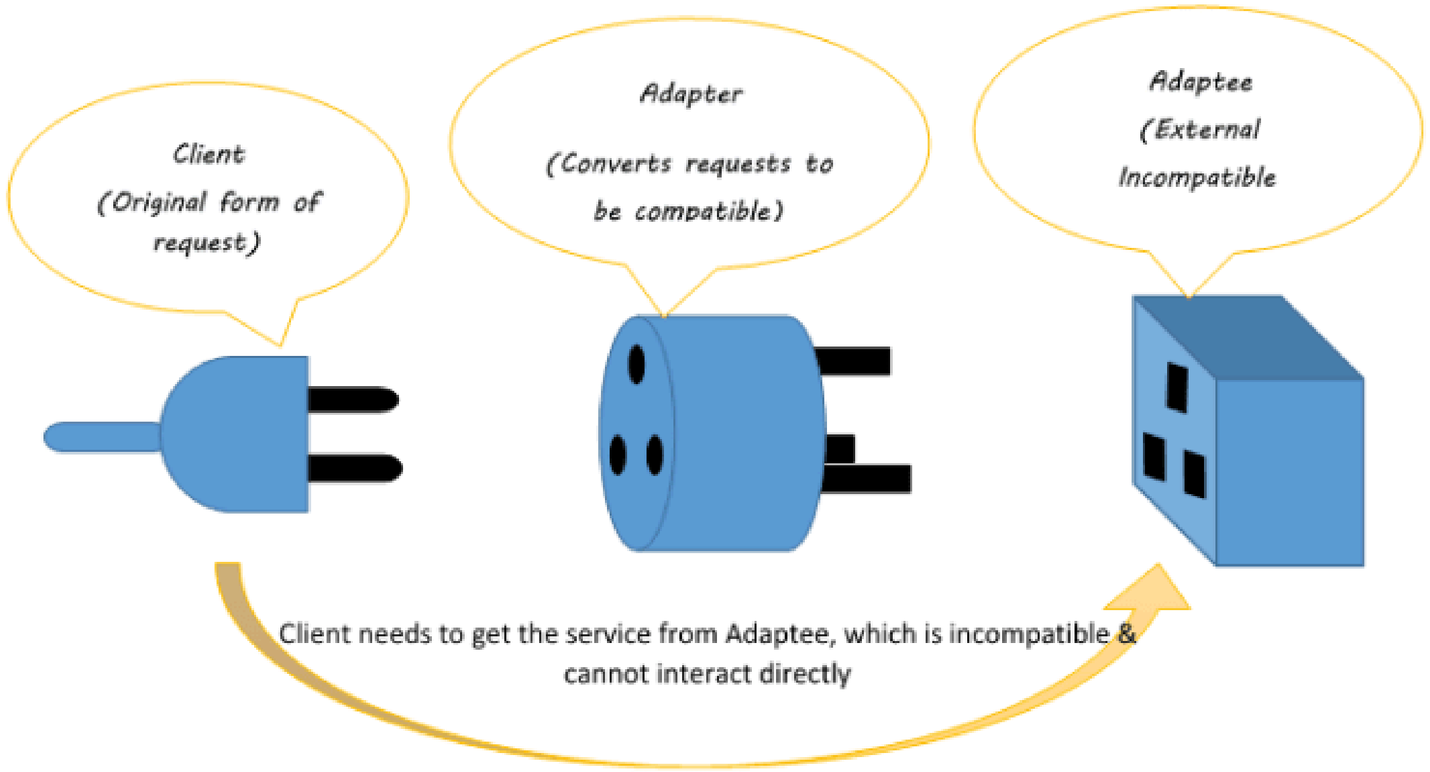
Donde:

- Puedes combinar cualquier tipo de motor con cualquier transmisión
- El mismo Motor puede usarse en Automovil, Motocicleta, Camion
- Agregar un nuevo tipo de motor no requiere cambiar la jerarquía
- Automovil.arrancar() delega a Motor.encender()

Patron Adapter

El patrón Adapter (adaptador) convierte la interfaz de una clase en otra interfaz que los clientes esperan. Permite que clases con interfaces incompatibles trabajen juntas.

Piensa en un adaptador de corriente cuando viajas. Si tu cargador tiene enchufe americano (2 pines planos) pero estás en Europa (2 pines redondos), necesitas un adaptador que:



- Target: La interfaz que el cliente espera
- Adaptee: La clase existente con interfaz incompatible
- Adapter: Implementa Target y contiene una instancia de Adaptee
- Cliente: Usa Target sin saber que hay un Adapter

Un escenario es cuando estamos integrando una **API Legacy**

Tu empresa tiene un sistema antiguo de pagos con esta interfaz:

SistemaPagosLegacy
- **procesarPagoTarjeta(numeroTarjeta, monto, comercio)**
- **verificarSaldo(numeroTarjeta)**
- **generarReciboTXT(transaccionId)**

Pero tu nueva aplicación espera esta interfaz moderna:

ProcesadorPagos
- **procesarPago(datosPago: PaymentData)**
- **consultarEstado(idTransaccion: string)**
- **generarRecibo(idTransaccion: string, formato: Format)**

Solución con Adapter:

AdapterPagosLegacy implements ProcesadorPagos
- **contiene → sistemaPagosLegacy: SistemaPagosLegacy**
- **procesarPago() → traduce y llama a procesarPagoTarjeta()**
- **consultarEstado() → traduce y llama a verificarSaldo()**
- **generarRecibo() → traduce y llama a generarReciboTXT()**

Mix-ins

Los mix-ins son una técnica para agregar funcionalidad a clases sin usar herencia múltiple (que muchos lenguajes no soportan). Permiten "mezclar" comportamiento de múltiples fuentes.

Piensa en los ingredientes de una pizza. La base (clase) es la masa. Los mix-ins son los toppings que agregas: queso, pepperoni, champiñones. Puedes mezclar los que quieras para crear tu pizza personalizada.

¿Cómo hacer que Usuario tenga capacidades de:

- Logging
- Serialización
- Validación
- Cache

... sin crear una jerarquía compleja?

Con herencia seria algo como:

Usuario → UsuarioConLogging → UsuarioConSerializacion → ...

Con Mix-ins algo como:

Usuario + Logging + Serialization + Validation = UsuarioCompleto

```
function withLogging(BaseClass) {
    return class extends BaseClass {
        log(mensaje) { /* implementación */ }
    }
}

// Aplicación
class Usuario { /* ... */ }
const UsuarioConLogging = withLogging(Usuario);
```

Lo que sucede:

- withLogging recibe una clase
- Retorna una nueva clase que extiende la original
- La nueva clase tiene los métodos originales + log()

El buen diseño orientado a objetos no se trata de usar todas las técnicas, sino de elegir las correctas para cada problema. **Simplicidad primero, sofisticación solo cuando sea necesaria.**

¿Que problemas tenemos?

```
class Animal {
    nombre: string
    volar(): void
    nadar(): void
    caminar(): void
}

class Perro extends Animal {
    // Problema: Los perros no vuelan
}

class Pez extends Animal {
    // Problema: Los peces no caminan ni vuelan
}

class Ave extends Animal {
    // Algunos pueden nadar, otros no
}
```

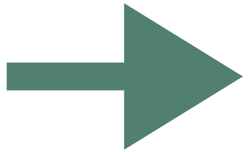
Un mejor diseño

```
interface Volador {
    volar(): void
}
interface Nadador {
    nadar(): void
}
interface Caminador {
    caminar(): void
}

class Animal {
    nombre: string
}

class Perro extends Animal implements Caminador, Nadador {
    caminar() { /* ... */ }
    nadar() { /* ... */ }
}
class Pez extends Animal implements Nadador {
    nadar() { /* ... */ }
}
class Pato extends Animal implements Volador, Nadador, Caminador {
    volar() { /* ... */ }
    nadar() { /* ... */ }
    caminar() { /* ... */ }
}
```

Estructuras de Datos Genéricas

Ya las conocemos

```
class ListaNumeros {
    private items: number[] = [];
    agregar(item: number) { this.items.push(item); }
    obtener(index: number): number { return this.items[index]; }
}

class ListaStrings {
    private items: string[] = [];
    agregar(item: string) { this.items.push(item); }
    obtener(index: number): string { return this.items[index]; }
}
```



```
class Lista<T> {
    private items: T[] = [];

    agregar(item: T): void {
        this.items.push(item);
    }

    obtener(index: number): T {
        return this.items[index];
    }

    obtenerTodos(): T[] {
        return this.items;
    }
}
```

Iteradores

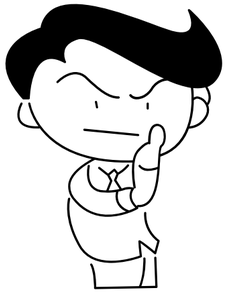
¿Por qué console.log() puede imprimir arrays, objetos, Maps, Sets, etc.?

Todas estas estructuras son iterables.

En TS/JS, un iterable es cualquier objeto que implementa el método [Symbol.iterator]().

Y se ve mas o menos así:

```
class Rango {
    constructor(
        private inicio: number,
        private fin: number
    ) {}

    // Hace que la clase sea iterable
    *[Symbol.iterator]() {
        for (let i = this.inicio; i <= this.fin; i++) {
            yield i;
        }
    }
}
// Uso
const rango = new Rango(1, 5);
for (const num of rango) {
    console.log(num); // 1, 2, 3, 4, 5
}
```



¿Qué es yield?

yield es como un botón de "Pausa y Entrega"

- Pausa la ejecución: La función se congela en esa línea
- Entrega un valor: Le da el valor actual al llamador
- Mantiene el estado: Recuerda dónde quedó para continuar después
- Espera: No continúa hasta que se pida el siguiente valor

```
// Generador de números infinitos
function* numerosPares(): IterableIterator<number> {
    let n = 0;
    while (true) {
        yield n;
        n += 2;
    }
}

const gen = numerosPares();
console.log(gen.next().value); // 0
console.log(gen.next().value); // 2
console.log(gen.next().value); // 4
```

Restricciones de Tipos (Type Constraints)

A veces necesitamos que T no sea cualquier tipo, sino que cumpla ciertos requisitos.

```
// No sabemos si T tiene el operador <
function minimo<T>(a: T, b: T): T {
    return a < b ? a : b; // Error de compilación
}
```

```
interface Comparable {
    compareTo(other: this): number;
}

function minimo<T extends Comparable>(a: T, b: T): T {
    return a.compareTo(b) < 0 ? a : b;
}
```

Restricción con multiples interfaces

```typescript
interface Identificable {
    id: number;
}
interface Nombrable {
    nombre: string;
}

// T debe implementar AMBAS interfaces
function registrar<T extends Identificable & Nombrable>(obj: T): void {
    console.log(`Registrando: ${obj.id} - ${obj.nombre}`);
}
```

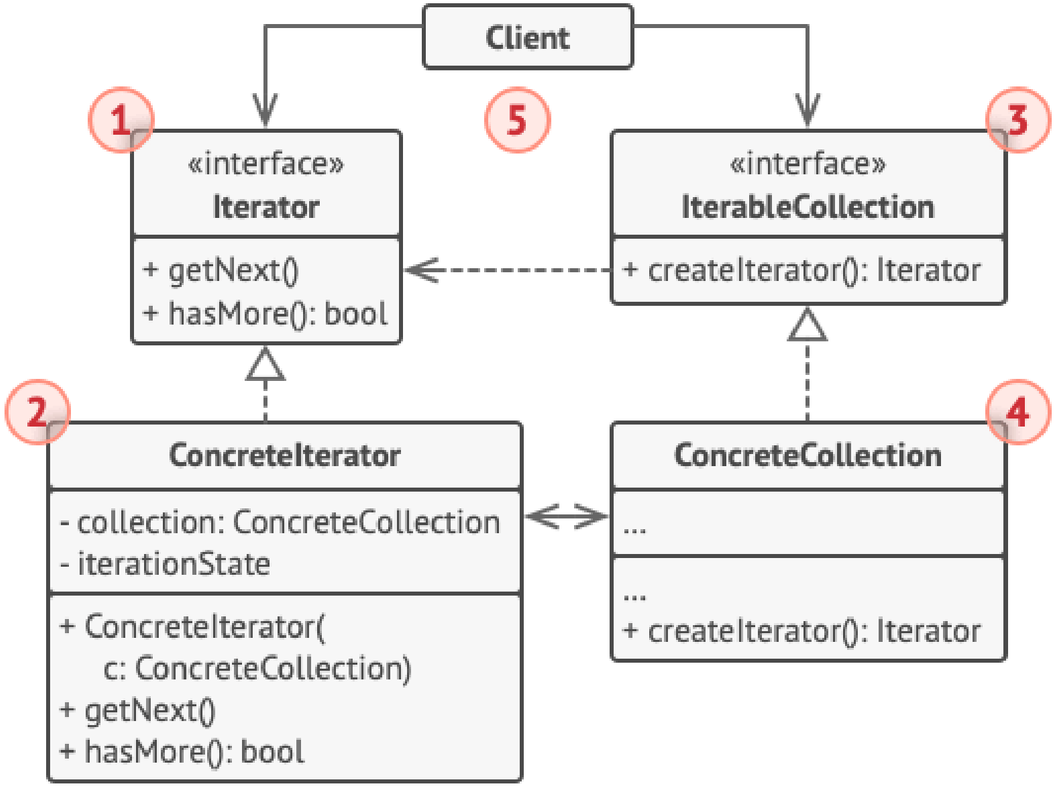
Restricción con Tipos Primitivos

```ts
// Solo acepta number o string
function procesar<T extends number | string>(valor: T): void {
    console.log(`Procesando: ${valor}`);
}

procesar(42);         // OK
procesar("hola");     // OK
procesar(true);       // Error: boolean no es number ni string
```

Patrón de Diseno Iterator

El patrón Iterator es un patrón de diseño de comportamiento que proporciona una forma de acceder secuencialmente a los elementos de una colección sin exponer su representación interna.

Te permite recorrer una colección (lista, árbol, grafo, etc.) sin necesitar saber cómo está implementada internamente.



- Iterator: Interfaz que define operaciones para recorrer elementos
- ConcreteIterator: Implementación específica del iterador
- IterableCollection: Interfaz de la colección que crea iteradores
- ConcreteCollection: Colección concreta que retorna su iterador

Sistema de Playlists

Imagina una aplicación de música como Spotify que maneja diferentes tipos de playlists:

- Playlist Lineal: Canciones en orden secuencial (1, 2, 3, 4...)
- Playlist Aleatoria: Canciones en orden aleatorio shuffle
- Playlist por Género: Agrupa canciones por categoría, luego reproduce

Sin Iterator, el reproductor necesitaría tener código específico para cada tipo:

Si es PlaylistLineal → reproducir en orden
Si es PlaylistAleatoria → generar orden aleatorio, luego reproducir
Si es PlaylistPorGenero → buscar canciones del género, luego reproducir

El reproductor está acoplado a cada implementación de playlist

Con Iterator, el reproductor simplemente pide "siguiente canción" sin importar el tipo:

Mientras haya canciones → Dame la siguiente canción → Reproducir

El reproductor está desacoplado - no necesita saber cómo está organizada la playlist

Cómo Funciona

Paso 1: Cada playlist crea su propio iterador especializado:

- PlaylistLineal → crea IteradorLineal (recorre posición 0, 1, 2...)
- PlaylistAleatoria → crea IteradorAleatorio (recorre índices mezclados)
- PlaylistPorGenero → crea IteradorPorGenero (recorre por categoría)

Paso 2: El reproductor recibe el iterador y usa la misma interfaz para todos:

iterator.hasNext() → ¿Hay más canciones?
iterator.next() → Dame la siguiente

Paso 3: Cada iterador sabe internamente cómo obtener "la siguiente":

- Lineal: Incrementa posición (pos++)
- Aleatorio: Toma siguiente de lista pre-mezclada
- Por Género: Salta al siguiente género cuando termina uno

Cómo Funciona

Paso 1: Cada playlist crea su propio iterador especializado:

- PlaylistLineal → crea IteradorLineal (recorre posición 0, 1, 2...)
- PlaylistAleatoria → crea IteradorAleatorio (recorre índices mezclados)
- PlaylistPorGenero → crea IteradorPorGenero (recorre por categoría)

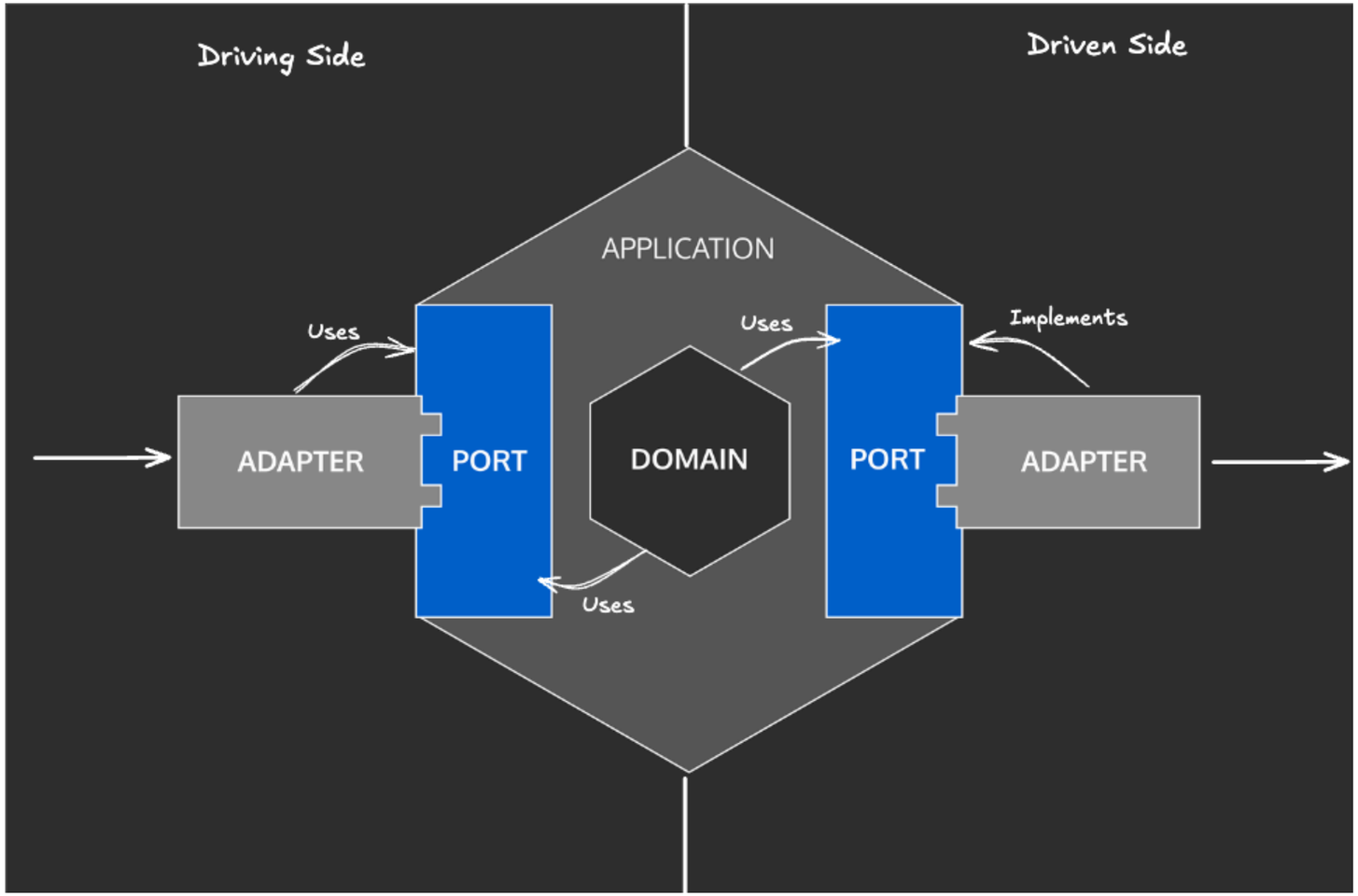
Paso 2: El reproductor recibe el iterador y usa la misma interfaz para todos:

iterator.hasNext() → ¿Hay más canciones?
iterator.next() → Dame la siguiente

Paso 3: Cada iterador sabe internamente cómo obtener "la siguiente":

- Lineal: Incrementa posición (pos++)
- Aleatorio: Toma siguiente de lista pre-mezclada
- Por Género: Salta al siguiente género cuando termina uno

Patrón de Puertos y Adaptadores



Continuará...