

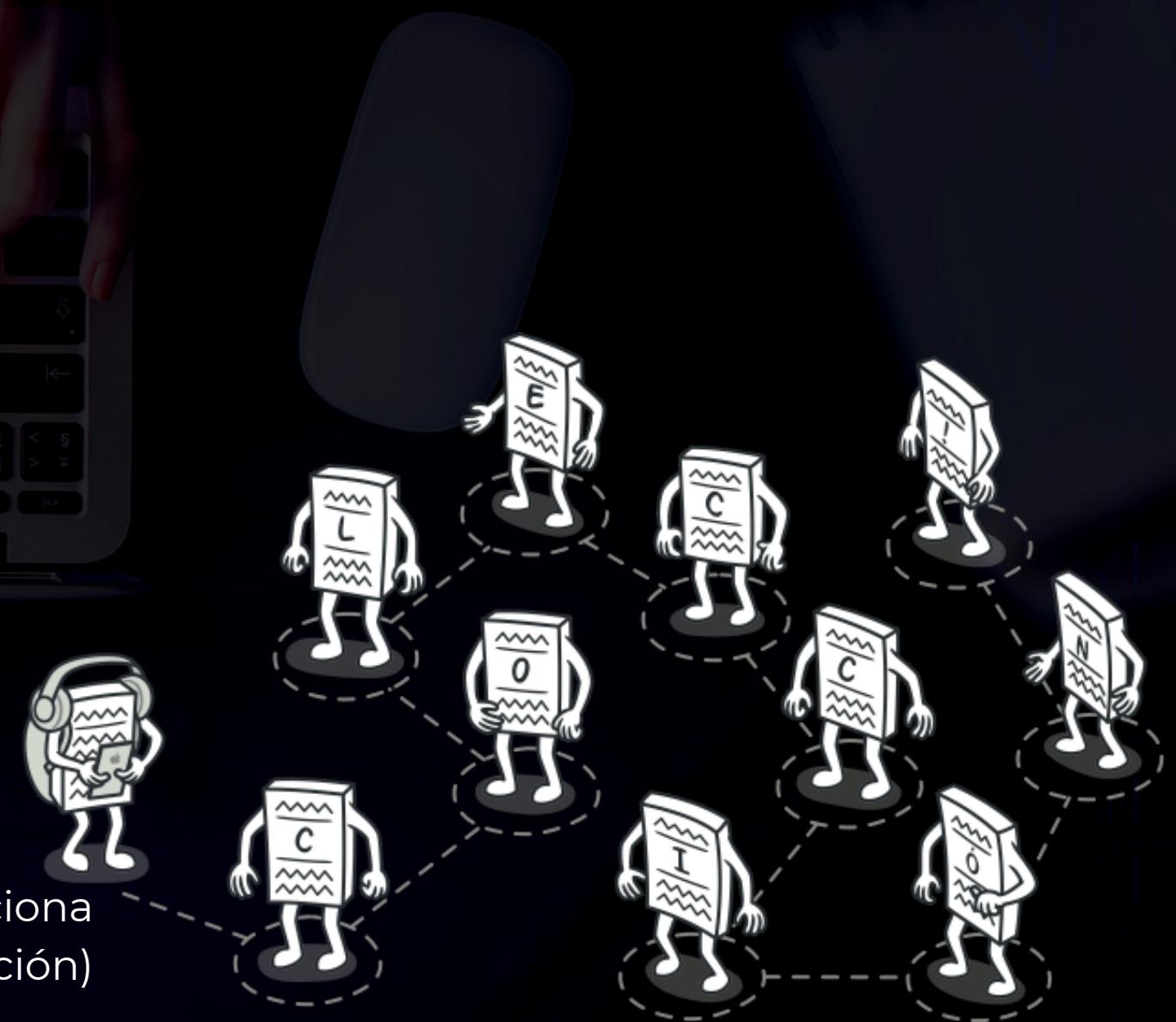


WILLIAM YARURO  
JOSE LOPEZ

# ITERATOR PATTERN

Es un patrón de diseño ampliamente utilizado en el desarrollo de software que proporciona una forma de acceder a los elementos de un objeto agregado (como una lista o colección) secuencialmente sin exponer su representación subyacente.

Este patrón de diseño permite recorrer una estructura de datos sin que sea necesario conocer la estructura interna de la misma. Es especialmente útil cuando trabajamos con estructuras de datos complejas, ya que nos permite recorrer sus elementos mediante un Iterador, el Iterador es una interface que proporciona los métodos necesarios para recorrer los elementos de la estructura de datos, los métodos más comunes son:



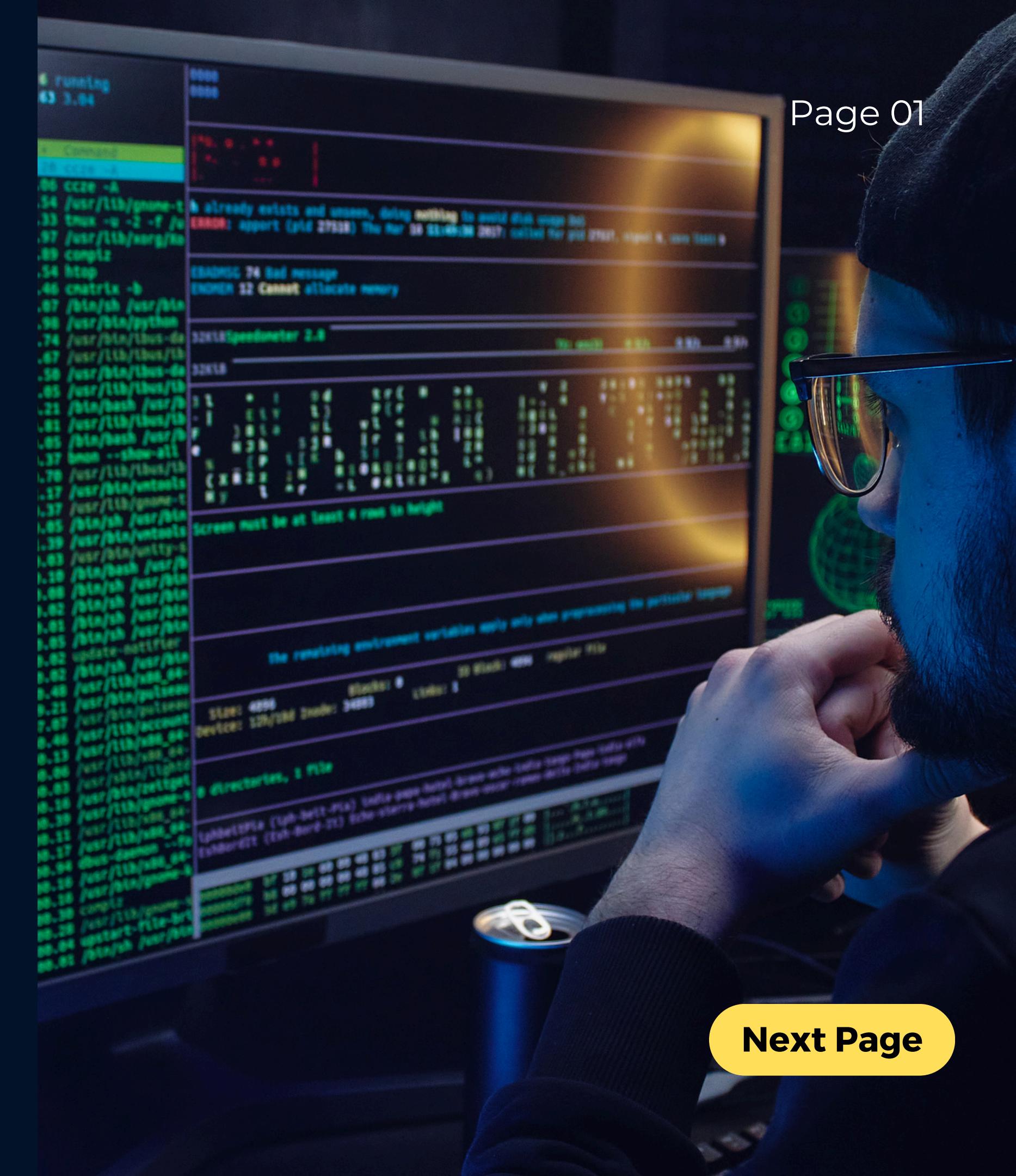
**Next Page**



# ¿QUÉ HACE EL PATRÓN DE DISEÑO ITERATOR?

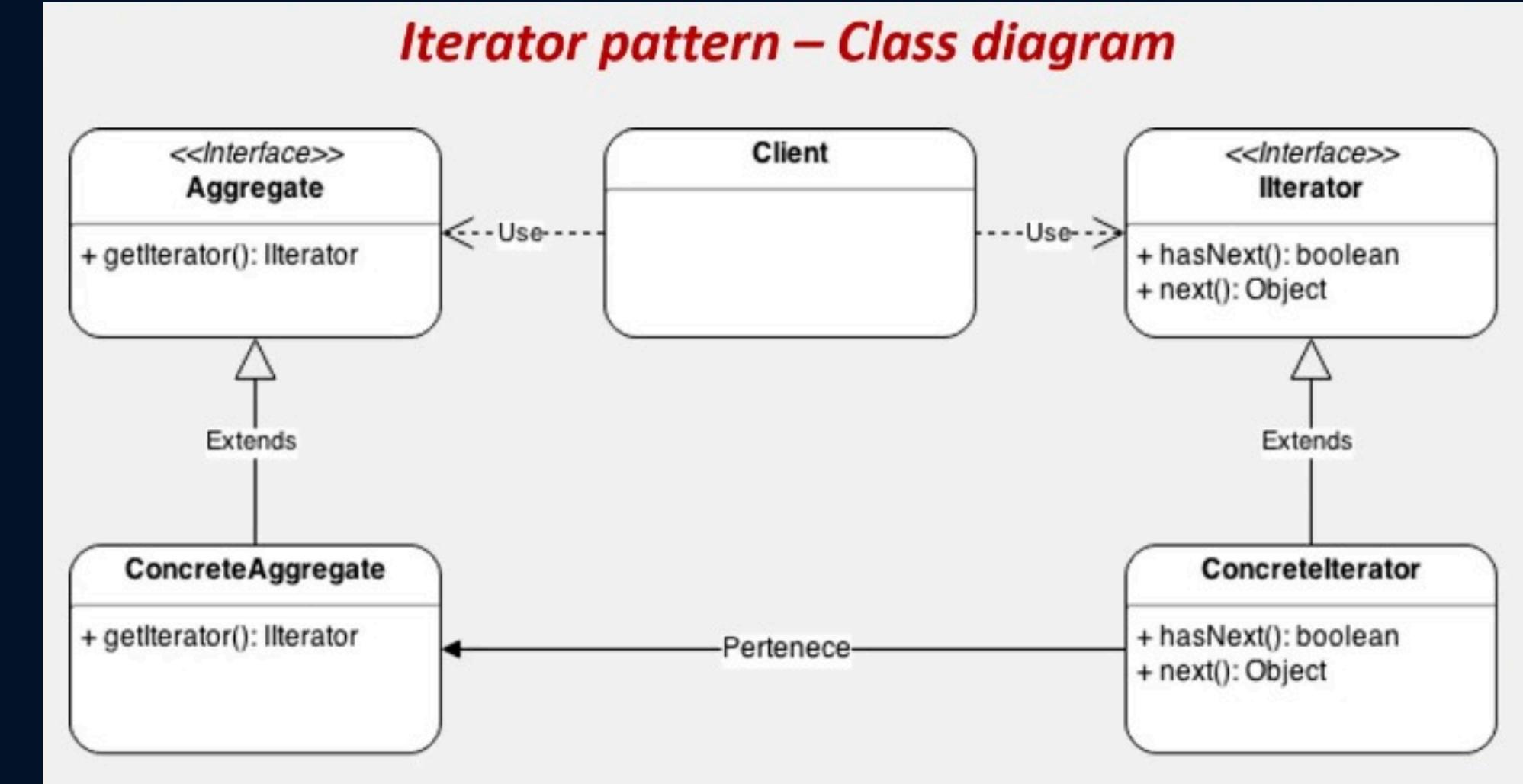
Define un objeto separado, llamado iterador, que encapsula los detalles de recorrer los elementos del agregado, permitiendo que el agregado cambie su estructura interna sin afectar a la forma en que se accede a sus elementos.

- El patrón iterador es un patrón de diseño relativamente sencillo y de uso frecuente. Hay muchas estructuras de datoscolecciones disponibles en cada lenguaje.
- Cada colección debe proporcionar un iterador que le permita iterar a través de sus objetos. Sin embargo, al hacerlo debe asegurarse de no exponer su implementación.



# PROPOSITO ITERATOR PATTERN

El propósito del patrón Iterator es proporcionar una manera de acceder a los elementos de una colección de forma secuencial sin revelar los detalles internos de la colección.



LOS ELEMENTOS DEL PATRÓN ITERATOR SE DESCRIBEN A CONTINUACIÓN:

- CLIENT: ACTOR QUE UTILIZA AL ITERATOR.
- AGGREGATE: INTERFACE QUE DEFINE LA ESTRUCTURA DE LAS CLASES QUE PUEDEN SER ITERADAS.
- CONCRETEAGGREGATE: CLASE QUE CONTIENE LA ESTRUCTURA DE DATOS QUE DESEAMOS ITERAR.
- IITERATOR: INTERFACE QUE DEFINE LA ESTRUCTURA DE LOS ITERADORES, LA CUAL DEFINE LOS MÉTODOS NECESARIOS PARA PODER REALIZAR LA ITERACIÓN SOBRE EL CONCRETEAGGREGATOR.
- CONCRETEITERATOR: IMPLEMENTACIÓN DE UN ITERADOR CONCRETO, EL CUAL HEREDA DE IITERATOR PARA IMPLEMENTAR DE FORMA CONCRETA CÓMO ITERAR UN CONCRETEAGGREGATE.

# ¿CUANDO IMPLEMENTARLO?

01

Cuando necesites recorrer una colección de objetos sin exponer su representación subyacente.

03

Cuando deseas encapsular el proceso de iteración de modo que el código que usa la colección no dependa de su estructura interna.

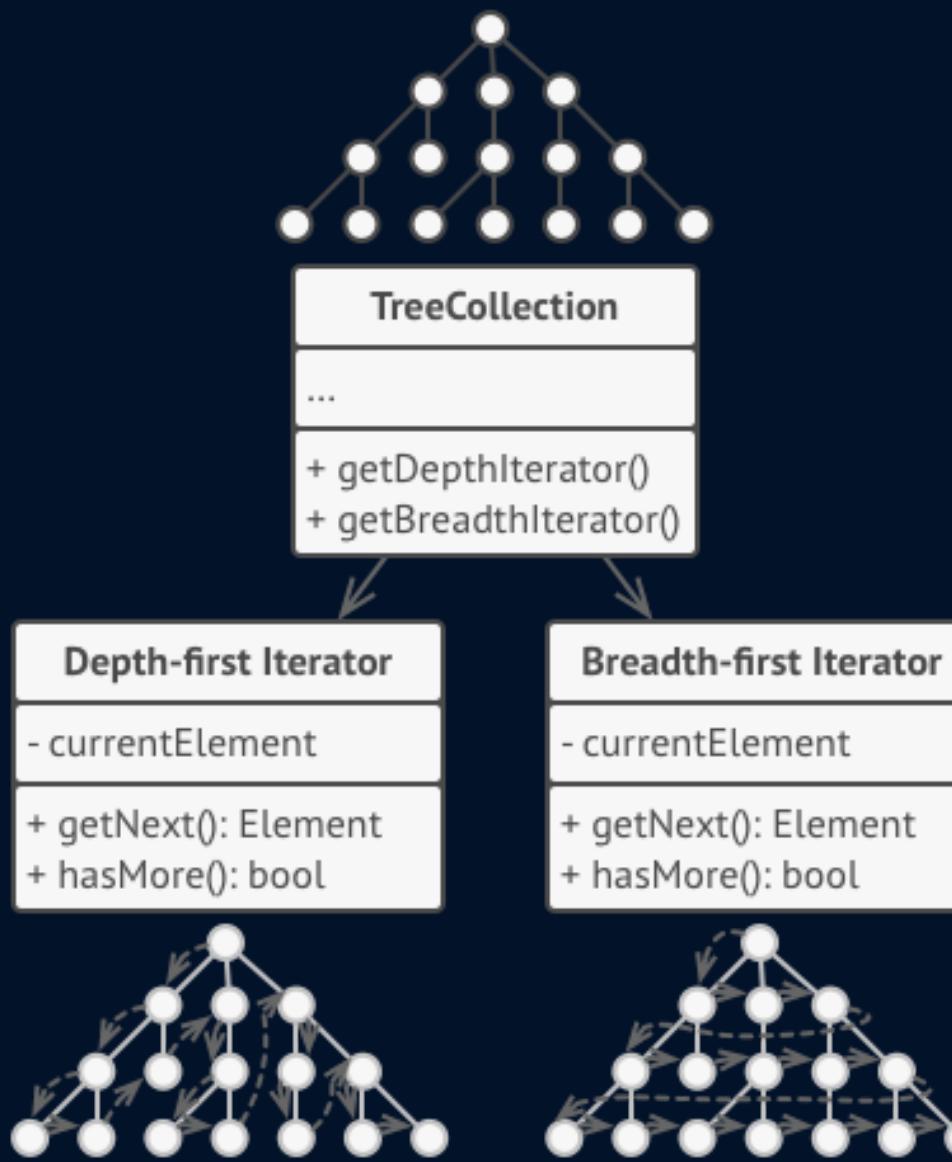
02

Cuando deseas proporcionar múltiples formas de recorrer una colección.

04

Cuando la estructura es compleja y recorrerla puede resultar difícil para quien no está familiarizado con la estructura.

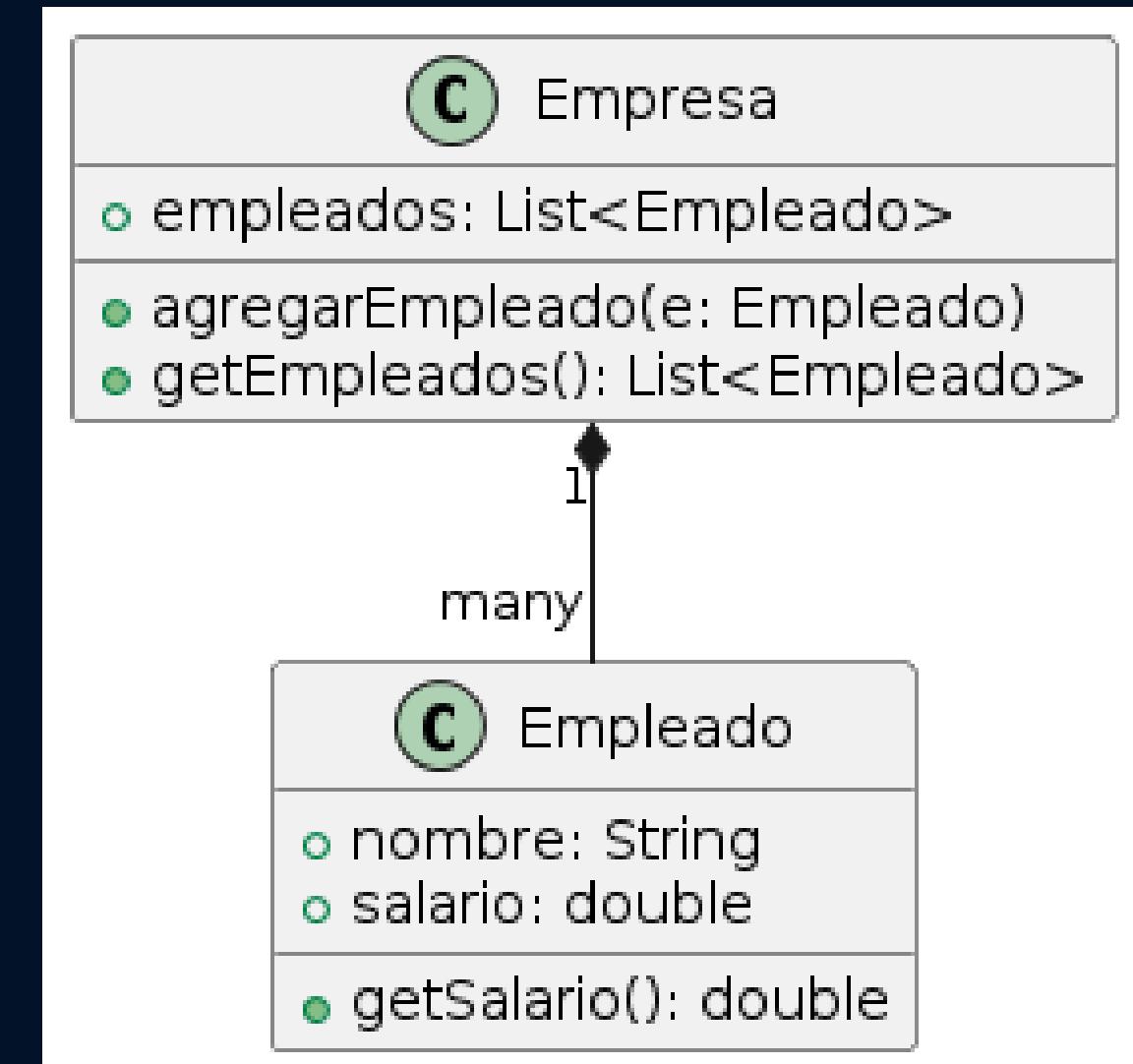
**Next Page**



# SITUACION

*Consideremos una aplicación que maneja una lista de empleados en una empresa, donde se requiere calcular el salario total de todos los empleados.*

- *La empresa tiene diferentes empleados con distintos salarios.*
- *A partir de una clase principal, se solicita el cálculo del salario total sumando los salarios de todos los empleados.*



# SITUACION

```
public class Empleado { 7 usages
    private String nombre; 1 usage
    private double salario; 2 usages

    public Empleado(String nombre, double salario) { 3 usages
        this.nombre = nombre;
        this.salario = salario;
    }

    public double getSalario() { 1 usage
        return salario;
    }
}
```

```
public class Empresa { 2 usages
    private List<Empleado> empleados; 3 usages

    public Empresa() { 1 usage
        empleados = new ArrayList<>();
    }

    public void agregarEmpleado(Empleado empleado) { 3 usages
        empleados.add(empleado);
    }

    public List<Empleado> getEmpleados() { 1 usage
        return empleados;
    }
}
```

# SITUACION

## PROBLEMAS SIN EL PATRÓN ITERATOR

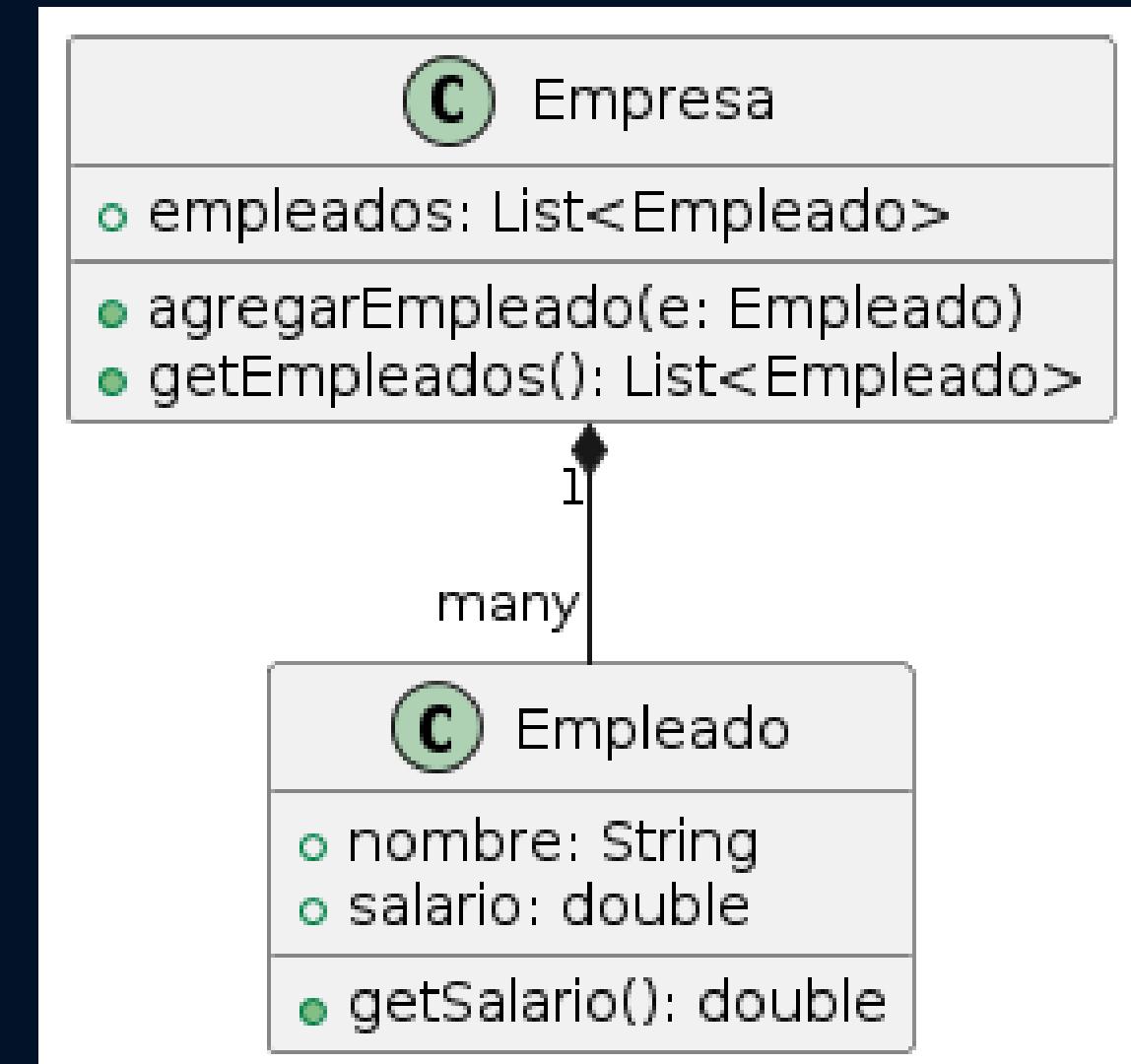
- **ACOPLAMIENTO:** EL CÓDIGO CLIENTE NECESITA CONOCER LA ESTRUCTURA INTERNA DE EMPRESA PARA ITERAR SOBRE LOS EMPLEADOS.
- **REPETICIÓN DE CÓDIGO:** CADA VEZ QUE SE NECESITE ITERAR SOBRE EMPRESA, SE TENDRÁ QUE ESCRIBIR UN BUCLE SIMILAR.
- **POCA FLEXIBILIDAD:** CAMBIAR LA ESTRUCTURA INTERNA DE EMPRESA REQUERIRÍA CAMBIOS EN TODO EL CÓDIGO QUE LA USE.

```
public class EjemploSinIterator {  
    public static void main(String[] args) {  
        Empresa empresa = new Empresa();  
        empresa.agregarEmpleado(new Empleado(nombre: "Juan", salario: 3000));  
        empresa.agregarEmpleado(new Empleado(nombre: "Maria", salario: 3500));  
        empresa.agregarEmpleado(new Empleado(nombre: "Pedro", salario: 4000));  
  
        double salarioTotal = 0;  
        for (Empleado empleado : empresa.getEmpleados()) {  
            salarioTotal += empleado.getSalario();  
        }  
        System.out.println("Salario total de la empresa: " + salarioTotal);  
    }  
}
```

# SITUACION

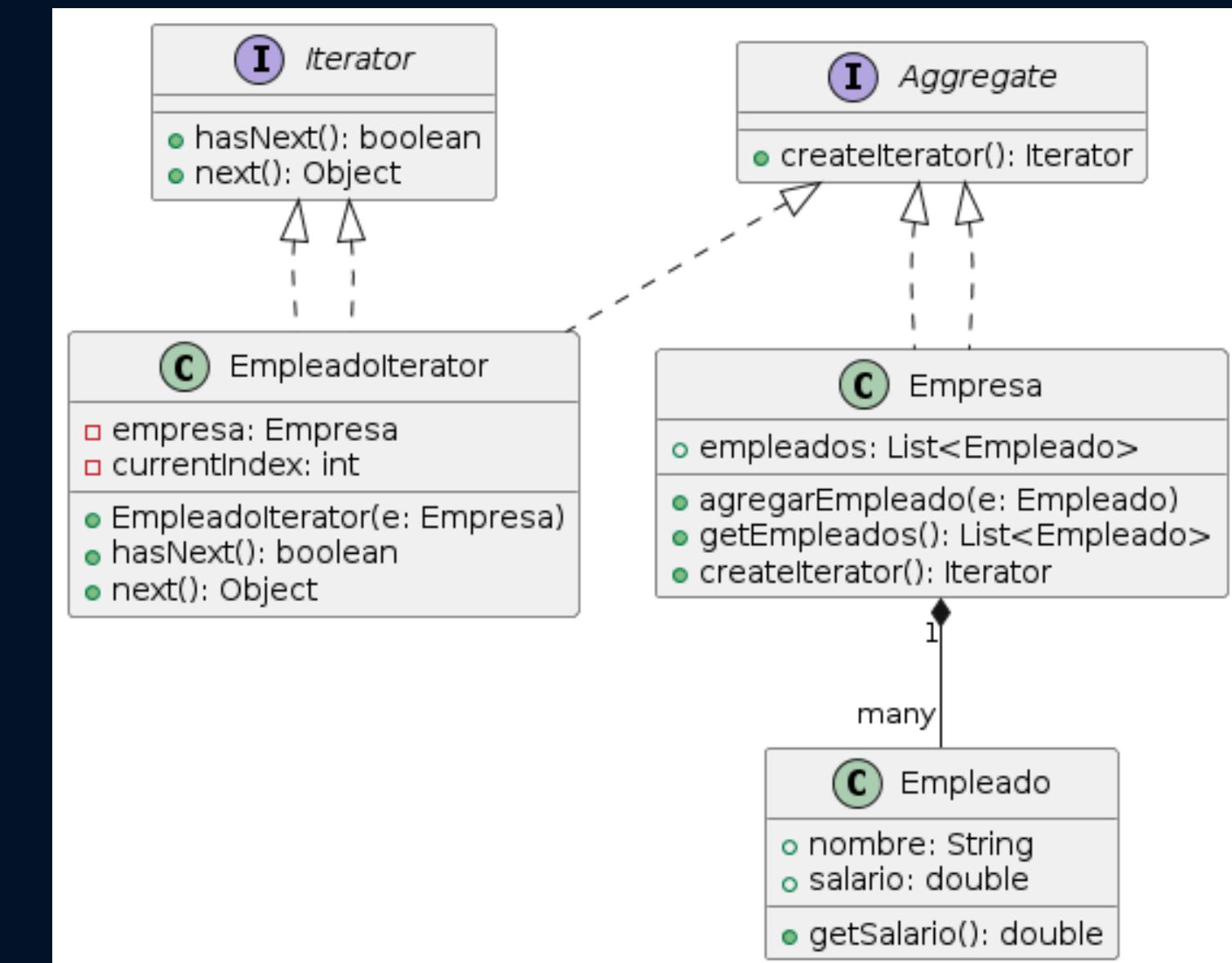
*Supongamos que la empresa necesita una mayor flexibilidad para iterar sobre los empleados. Se desea:*

- 1. Ocultar la estructura interna de la colección de empleados, de modo que el código cliente no dependa de la implementación específica (lista, arreglo, etc.).*
- 2. Proporcionar múltiples formas de iteración, como iterar por departamentos, por antigüedad, o por cualquier otro criterio sin cambiar la estructura interna de la colección.*
- 3. Facilitar la extensión futura, permitiendo agregar nuevos tipos de iteradores sin modificar el código existente.*



# SITUACION

Para resolver estos problemas, implementamos el patrón Iterator. Esto permite iterar sobre la lista de empleados de una manera desacoplada y flexible.



# SITUACION

```
public interface Iterator { 4 usages 1 implementation
    boolean hasNext(); 2 usages 1 implementation
    Object next(); 1 usage 1 implementation
}
```

```
public class Empleado { no usages
    private String nombre; 1 usage
    private double salario; 2 usages

    public Empleado(String nombre, double salario) { no usages
        this.nombre = nombre;
        this.salario = salario;
    }

    public double getSalario() { no usages
        return salario;
    }
}
```

```
public interface Aggregate { no usages 1 implementation
    Iterator createIterator(); no usages 1 implementation
}
```

```
public class EmpleadoIterator implements Iterator { 1 usage
    private Empresa empresa; 3 usages
    private int currentIndex = 0; 2 usages

    public EmpleadoIterator(Empresa empresa) { 1 usage
        this.empresa = empresa;
    }

    @Override 2 usages
    public boolean hasNext() {
        return currentIndex < empresa.getSize();
    }

    @Override 1 usage
    public Object next() {
        if (hasNext()) {
            return empresa.getEmpleadoAt(currentIndex++);
        }
        return null;
    }
}
```

# SITUACION

```
public class Empresa implements Aggregate { no usages
    private List<Empleado> empleados; 5 usages

    public Empresa() { no usages
        empleados = new ArrayList<>();
    }

    public void agregarEmpleado(Empleado empleado) { no usages
        empleados.add(empleado);
    }

    public Empleado getEmpleadoAt(int index) { no usages
        if (index >= 0 && index < empleados.size()) {
            return empleados.get(index);
        }
        return null;
    }

    public int getSize() { no usages
        return empleados.size();
    }

    @Override no usages
    public Iterator createIterator() {
        return new EmpleadoIterator(empresa: this);
    }
}
```

```
public class CasoConIterator {
    public static void main(String[] args) {
        Empresa empresa = new Empresa();
        empresa.agregarEmpleado(new Empleado( nombre: "Juan", salario: 3000));
        empresa.agregarEmpleado(new Empleado( nombre: "Maria", salario: 3500));
        empresa.agregarEmpleado(new Empleado( nombre: "Pedro", salario: 4000));

        Iterator iterator = empresa.createIterator();
        double salarioTotal = 0;
        while (iterator.hasNext()) {
            Empleado empleado = (Empleado) iterator.next();
            salarioTotal += empleado.getSalario();
        }
        System.out.println("Salario total de la empresa: " + salarioTotal);
    }
}
```

# PROS Y CONTRAS DEL PATRÓN ITERATOR

---

## PROS

- DESACOPLAMIENTO: EL CLIENTE NO NECESITA CONOCER LA ESTRUCTURA INTERNA DE LA COLECCIÓN.
- REUTILIZACIÓN DE CÓDIGO: LOS ITERADORES PUEDEN SER REUTILIZADOS EN DIFERENTES COLECCIONES.
- FLEXIBILIDAD: CAMBIAR LA ESTRUCTURA INTERNA DE LA COLECCIÓN NO AFECTA AL CÓDIGO CLIENTE.
- CLARIDAD: EL CÓDIGO DE ITERACIÓN SE MANTIENE LIMPIO Y SEPARADO DE LA LÓGICA DE LA COLECCIÓN.

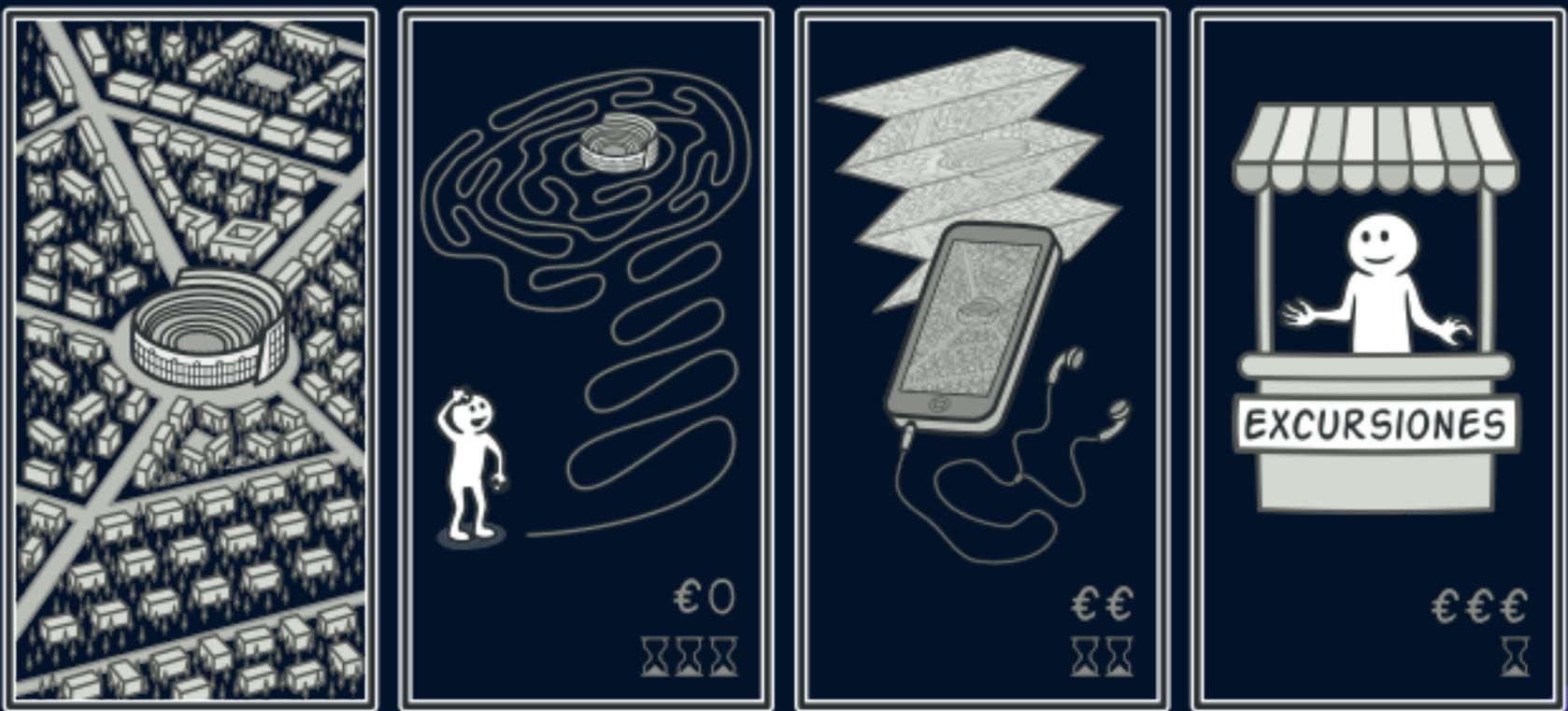
## CONTRAS

- SOBRECARGA ADICIONAL: IMPLEMENTAR EL PATRÓN ITERATOR PUEDE AÑADIR COMPLEJIDAD ADICIONAL AL CÓDIGO.
- RENDIMIENTO: EN ALGUNAS SITUACIONES, EL USO DE ITERADORES PUEDE TENER UN IMPACTO EN EL RENDIMIENTO DEBIDO A LA SOBRECARGA DE MÉTODOS ADICIONALES.

# MEJORA DE LA SITUACIÓN

EL USO DEL PATRÓN ITERATOR MEJORA LA SITUACIÓN INICIAL AL:

- REDUCIR EL ACOPLAMIENTO ENTRE EL CLIENTE Y LA ESTRUCTURA INTERNA DE LA COLECCIÓN.
- FACILITAR LA MODIFICACIÓN DE LA ESTRUCTURA INTERNA DE LA COLECCIÓN SIN AFECTAR AL CÓDIGO CLIENTE.
- PROMOVER LA REUTILIZACIÓN DE CÓDIGO MEDIANTE LA IMPLEMENTACIÓN DE ITERADORES REUTILIZABLES.
- MEJORAR LA CLARIDAD Y MANTENIBILIDAD DEL CÓDIGO, YA QUE LA LÓGICA DE ITERACIÓN SE MANEJA DE MANERA SEPARADA Y MODULAR.



ESTE ENFOQUE PERMITE UN DISEÑO MÁS ROBUSTO Y FLEXIBLE, LO QUE FACILITA EL MANTENIMIENTO Y LA EVOLUCIÓN DEL CÓDIGO A LO LARGO DEL TIEMPO.

# GRACIAS

No debí estudiar programación

