

METODOLOGÍAS DE PROGRAMACIÓN I

Patrón de comportamiento *Iterator*

Situación de ejemplo

Se posee una clase *Impresora* que recibe documentos a imprimir, la clase *Impresora* recorre las páginas de cada documento e imprime una por una.

```
class Impresora
    imprimir(documento)
        foreach(pagina in documento.paginas)
            imprimirPagina(pagina)

imprimirPagina(pagina)
    print("Imprimiendo página")
```

Situación de ejemplo

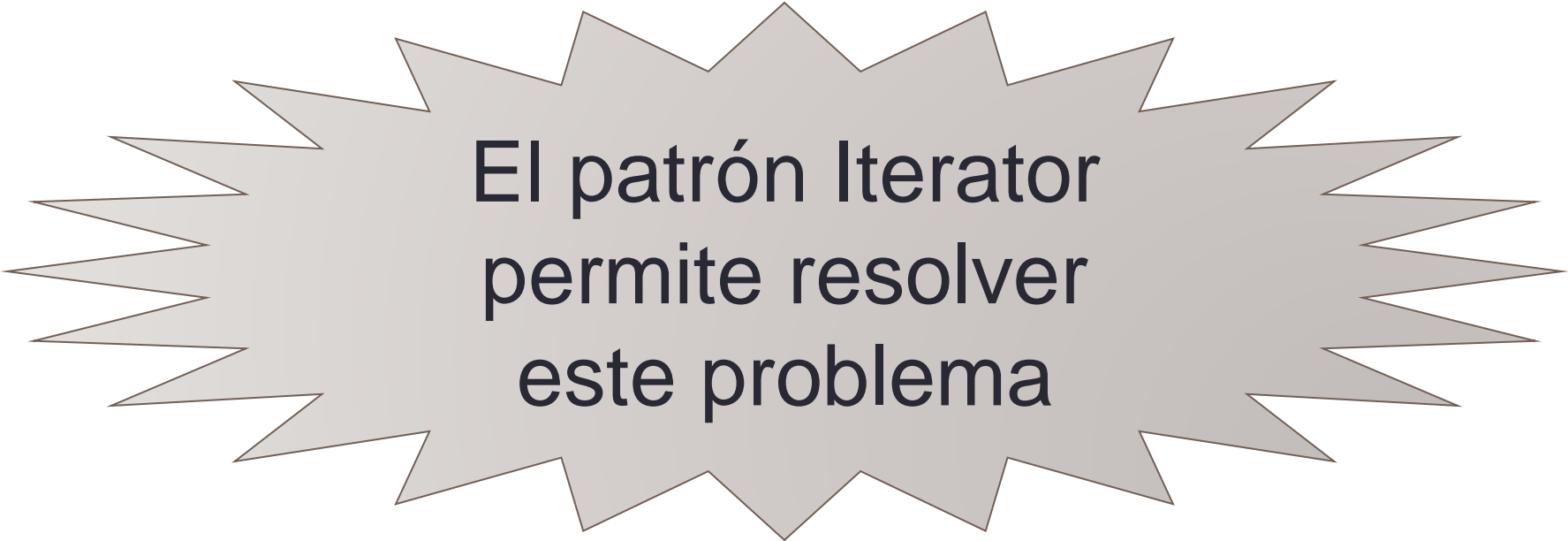
Se posee una clase *Impresora* que recibe documentos a imprimir, la clase *Impresora* recorre las páginas de cada documento e imprime una por una.

```
class Impresora
    imprimir(documento)
        foreach(pagina in documentos.paginas)
            imprimirPagina(pagina)
            imprimirPagina(pagina)
```

Esto funciona bien siempre que *paginas* sea un *Array*, *ArrayList* o *List* ¿Pero que pasa si alguien implementa un documento donde sus páginas son almacenadas en una Cola, en una Pila, en una lista enlazada o alguna otra estructura de datos?

Motivación

- Buscamos un diseño que permita que la impresora pueda "recorrer" todas las páginas de un documento independientemente de como estén almacenadas.



El patrón Iterator
permite resolver
este problema

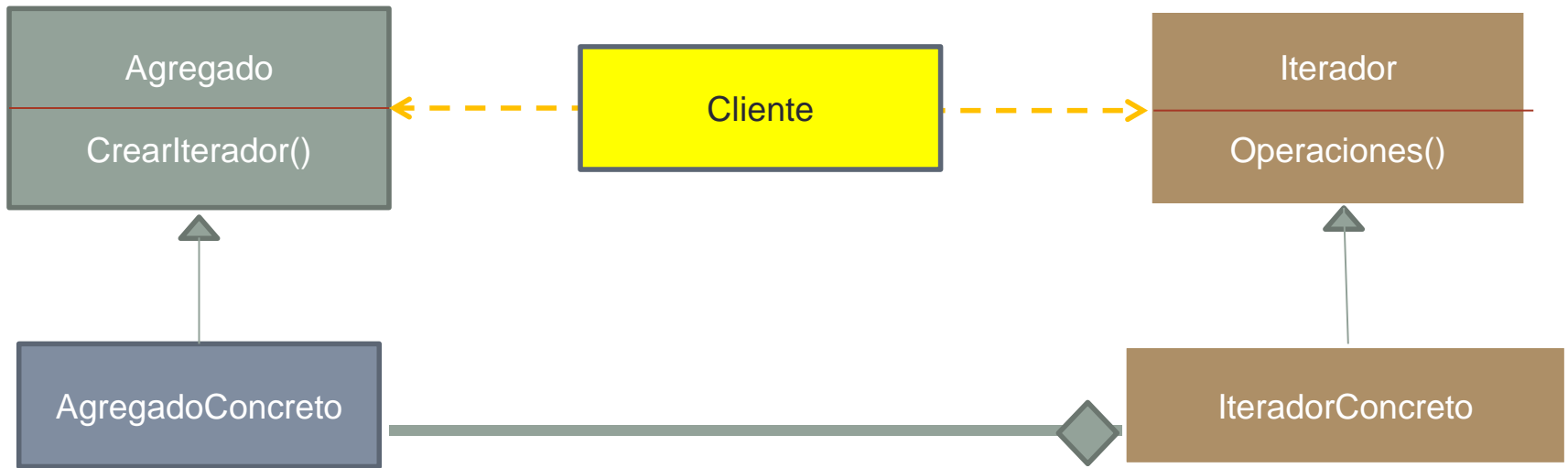
Iterator

Propósito: Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

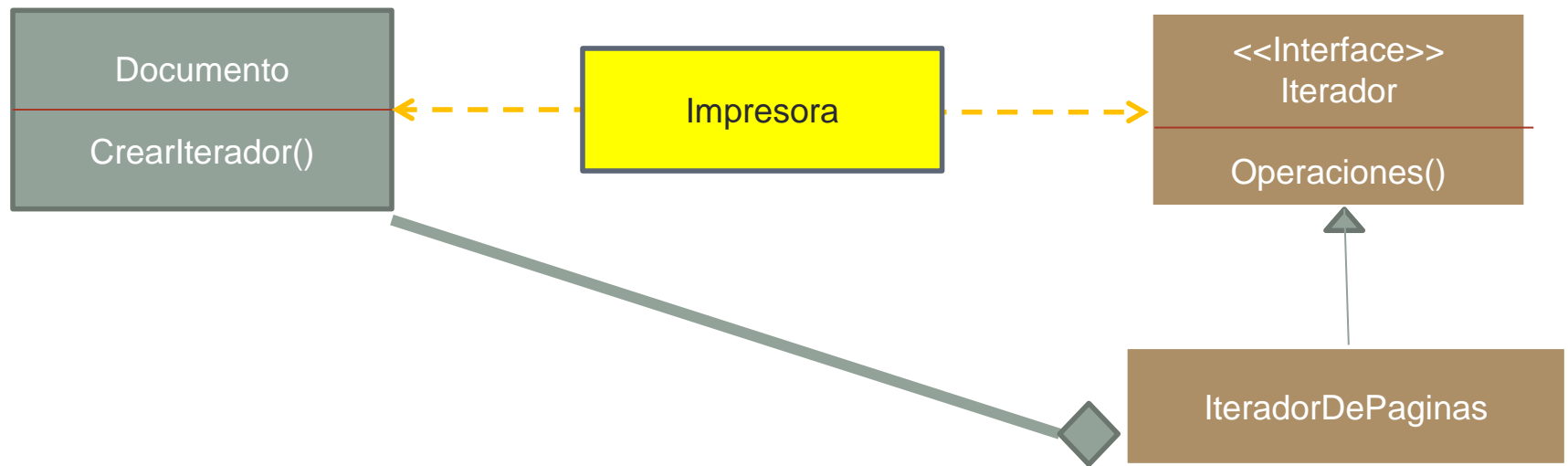
Aplicabilidad: usarlo cuando

- Necesite acceder al contenido de un objeto agregado sin exponer su representación interna.
- Necesite establecer varios recorridos sobre objetos agregados.
- Necesite proporcionar una interface uniforme para recorrer diferentes estructuras agregadas.

Iterator - Estructura



Iterator - Estructura



Iterator - Implementación

```
interface Iterador  
    void primero()  
    void siguiente()  
    boolean fin()  
    Iterable actual()
```

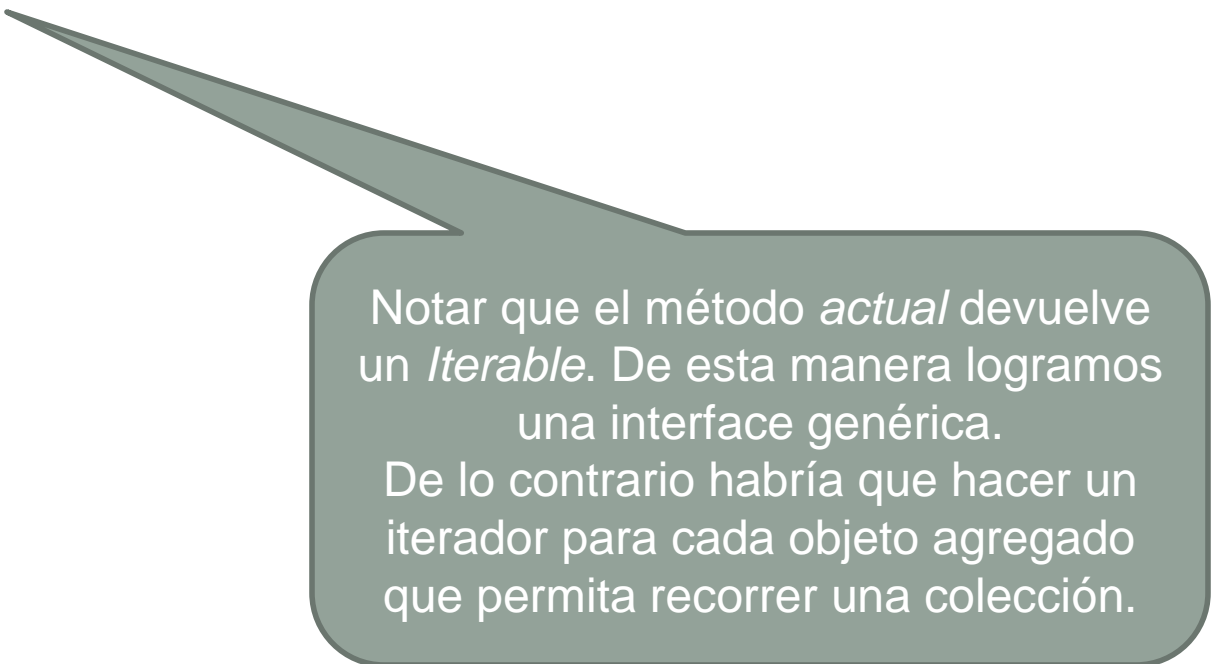

Iterator - Implementación

```
interface Iterador
    void primero()
    void siguiente()
    boolean fin()
    Iterable actual()
```

Definimos una interface *Iterador* para recorrer todos los elementos de una colección. Esta interface define métodos para posicionarse al principio de la colección, para ir al siguiente elemento, para saber si llegamos al final de la colección y para obtener el elemento actual.

Iterator - Implementación

```
interface Iterador  
    void primero()  
    void siguiente()  
    boolean fin()  
    Iterable actual()
```



Notar que el método *actual* devuelve un *Iterable*. De esta manera logramos una interface genérica.

De lo contrario habría que hacer un iterador para cada objeto agregado que permita recorrer una colección.

Iterator - Implementación

```
class IteradorDePaginas : Iterador
    int paginaActual;
    List paginas;
    constructor(documento)
        paginas = documento.paginas()
        primero()
    void primero()
        paginaActual = 0
    void siguiente()
        paginaActual = paginaActual + 1
    boolean fin()
        return paginaActual >= paginas.Count
    Iterable actual()
        return paginas[paginaActual]
```

Iterator - Implementación

```
class IteradorDePaginas : Iterador
    int paginaActual;
    List paginas;
    constructor(documento)
        paginas = documento.paginas
        primero()
    void primero()
        paginaActual = 0
    void siguiente()
        paginaActual = paginaActual + 1
    boolean fin()
        return paginaActual >= paginas.Count
    Iterable actual()
        return paginas[paginaActual]
```

Las páginas deberían
implementar la interfaz
Iterable

Iterator - Implementación

```
class IteradorDePaginas : Iterador
    int paginaActual;
    List paginas;
    constructor(documento)
        paginas = documento.paginas()
        primero()
    void primero()
        paginaActual = 0
    void siguiente()
        paginaActual = paginaActual + 1
    boolean fin()
        return paginaActual >= paginas.size()
    Iterable actual()
        return paginas[paginaActual]
```

El Iterador concreto *IteradorDePaginas* es el único objeto que conoce el estado interno de un *Documento*.

Dependiendo de la implementación puede guardar el *Documento* o solo las páginas

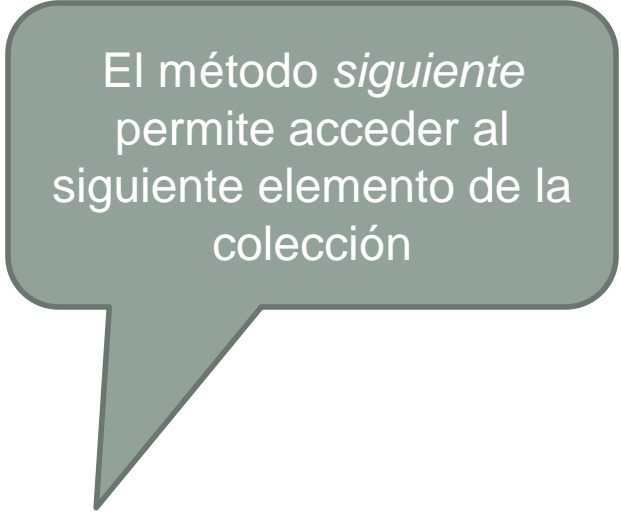
Iterator - Implementación

```
class IteradorDePaginas : Iterador
    int paginaActual;
    List paginas;
    constructor(documento)
        paginas = documento.paginas()
        primero()
    void primero()
        paginaActual = 0
    void siguiente()
        paginaActual = paginaActual + 1
    boolean fin()
        return paginaActual >= paginas.size()
    Iterable actual()
        return paginas[paginaActual]
```

El método *primero* setea como actual a la primer página de la colección

Iterator - Implementación

```
class IteradorDePaginas : Iterador
    int paginaActual;
    List paginas;
    constructor(documento)
        paginas = documento.paginas
        primero()
    void primero()
        paginaActual = 0
    void siguiente()
        paginaActual = paginaActual + 1
    boolean fin()
        return paginaActual >= paginas.Count
    Iterable actual()
        return paginas[paginaActual]
```



El método *siguiente* permite acceder al siguiente elemento de la colección

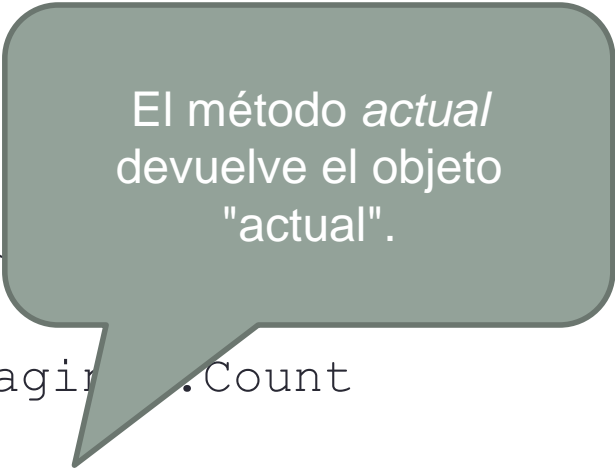
Iterator - Implementación

```
class IteradorDePaginas : Iterador
    int paginaActual;
    List paginas;
    constructor(documento)
        paginas = documento.paginas
        primero()
    void primero()
        paginaActual = 0
    void siguiente()
        paginaActual = paginaActual + 1
    boolean fin()
        return paginaActual >= paginas.Count
    Iterable actual()
        return paginas[paginaActual]
```

El método *fin* indica si se llegó al final de la colección, es decir, no hay más elementos para recorrer. Devuelve true si se alcanzó el fin de la colección, false en caso contrario

Iterator - Implementación

```
class IteradorDePaginas : Iterador
    int paginaActual;
    List paginas;
    constructor(documento)
        paginas = documento.paginas()
        primero()
    void primero()
        paginaActual = 0
    void siguiente()
        paginaActual = paginaActual + 1
    boolean fin()
        return paginaActual >= paginas.Count
    Iterable actual()
        return paginas[paginaActual]
```



El método *actual*
devuelve el objeto
"actual".

Iterator - Implementación

```
interface Iterable
```

```
    Iterador crearIterador ()
```

Debemos contar con una interface *Iterable*. Esta interface la deben implementar todos los objetos que posean un iterador para iterar sobre sus elementos

Iterator - Implementación

```
class Documento : Iterable
```

```
    Iterador crearIterador()
```

```
        return new IteradorDePaginas(self)
```

El *Documento* es el
responsable de crear el
iterador adecuado

Iterator - Implementación

```
class Impresora : Iterable
    void imprimir (documento)
        iterador = documento.crearIterador()
        while(not iterador.fin() )
            imprimirPagina( iterador.actual() )
            iterador.siguiente()
```

Iterator - Implementación

```
class Impresora : Iterable
    void imprimir (documento)
        iterador = documento.crearIterador()
        while(not iterador.fin() )
            imprimirPagina( iterador.actual() )
            iterador.siguiente()
```

Si la representación interna de *Documento* cambia,
¿es necesario modificar este algoritmo?

Iterator – Ventajas

- Permite variaciones en el recorrido de un agregado. Los agregados complejos puede recorrerse de diferentes maneras.
- Los iteradores simplifican la interfaz del agregado. La interfaz de recorrido elimina la necesidad de tener una interfaz similar en el agregado.
- Es posible hacer más de un recorrido a la vez. Como el estado del recorrido lo tiene el iterador, es posible tener más de un iterador sobre la misma colección al mismo tiempo.