



Ingeniería de software

Iterator & Template Method

Patrones de **diseño**

Equipo 5

Introducción

Los patrones de diseño son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software. En esta exposición hablaremos sobre dos patrones de diseño muy útiles: iterator y template method.

```
3 require File.expand_path("../test_helper", __FILE__)
4 # Prevent database truncation if the environment is production
5 abort("The Rails environment is running in production mode!")
6 require 'spec_helper'
7 require 'rspec/rails'
8
9 require 'capybara/rspec'
10 require 'capybara/rails'
11
12 Capybara.javascript_driver = :webkit
13 Category.delete_all; Category.create
14 Shoulda::Matchers.configure do |config|
15   config.integrate do |int|
16     with.test_framework :rspec
17     with.library :rails
18   end
19 end
20
21 # Add additional requires below this line. You can require
22 # spec/support/ and its subdirectories. These files will
23 # run as spec files by default. You can also use
24 # in _spec.rb will both be required when you run
25 # run twice. It is recommended that you do not
26 # end with _spec.rb. You can configure the
27 # behavior on the command line using --no-color
28 # or --color.
29
30 No results found for 'mongoid'
```

Iterator


Alias : **Cursor**

Intensión

Proporcionar un modo eficiente de acceder secuencialmente a los elementos de una colección sin exponer su representación interna del objeto.

Problema

Las colecciones se usan ampliamente en los sistemas de software, y es común necesitar recorrerlas y acceder a sus elementos de forma secuencial. Sin embargo, acceder a los elementos de una colección a través de su representación subyacente puede ser difícil y poco práctico. [1]

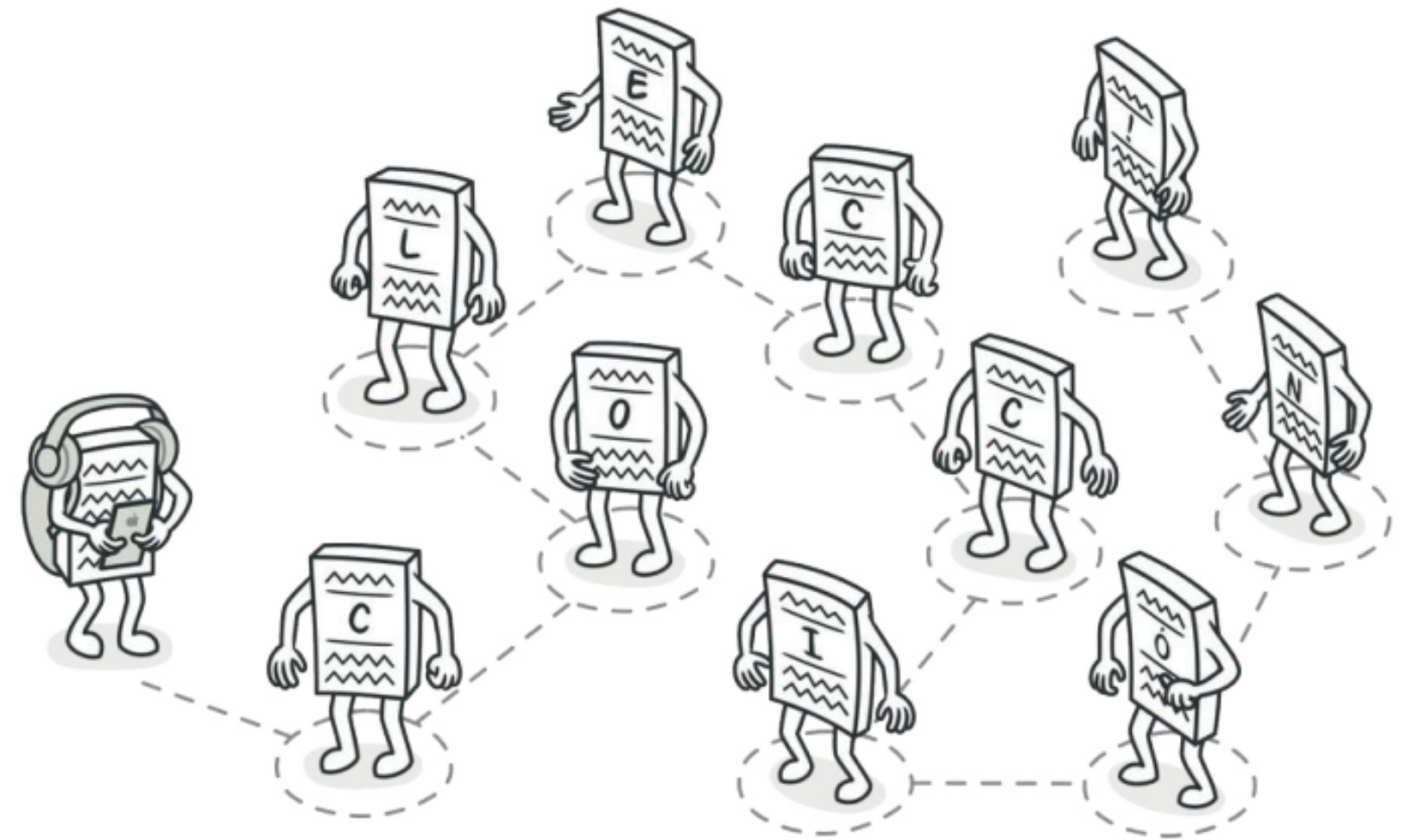


[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. Páginas 289-304, 360-365.

Solución

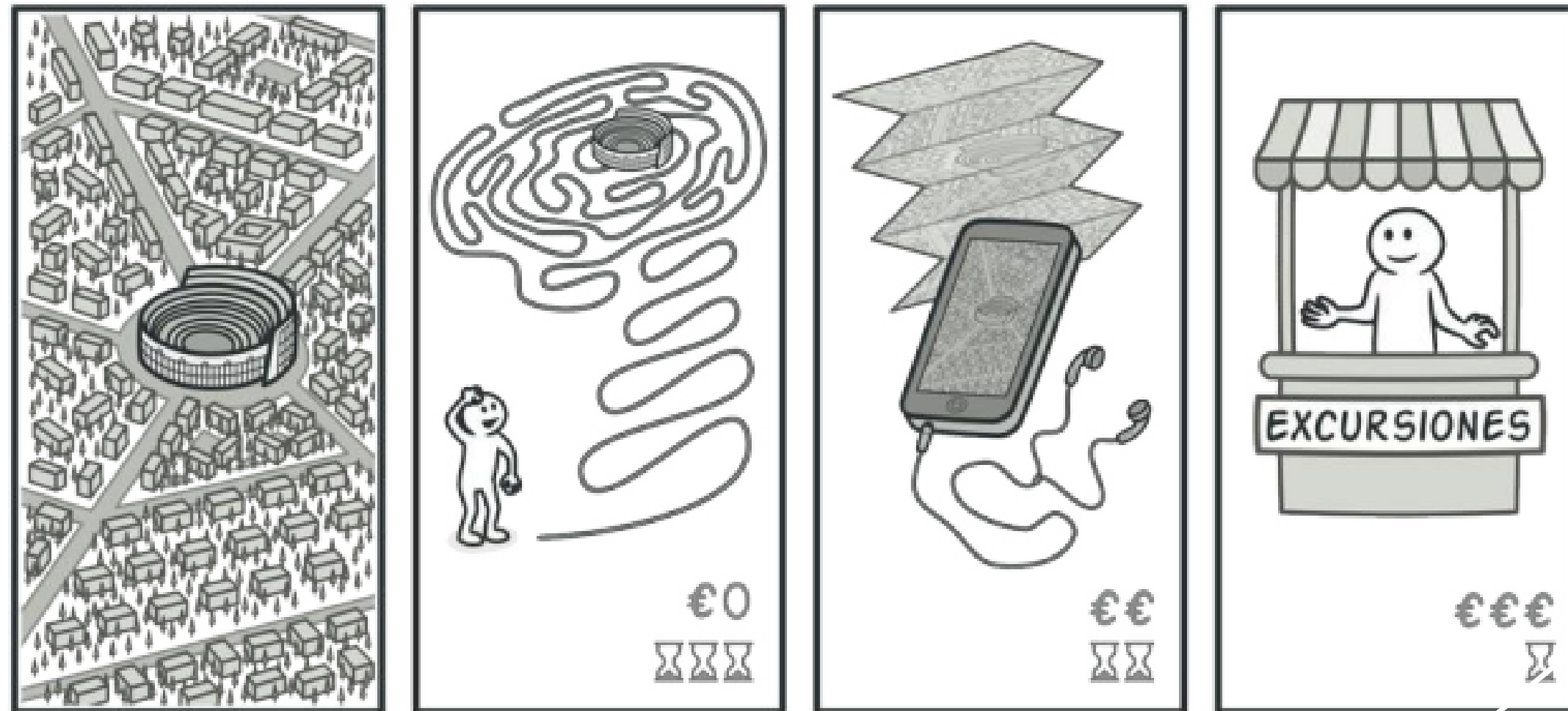
El patrón Iterator define una **interfaz** común para los iteradores y una clase concreta que implementa esta interfaz para cada tipo de colección.

Los iteradores mantienen un seguimiento de su posición actual en la colección y proporcionan métodos para acceder a los elementos de la colección de forma secuencial. [1]



[1] Freeman, E., & Freeman, E. (2004). Head First Design Patterns: A Brain-Friendly Guide. O'Reilly Media. Páginas 313-422.

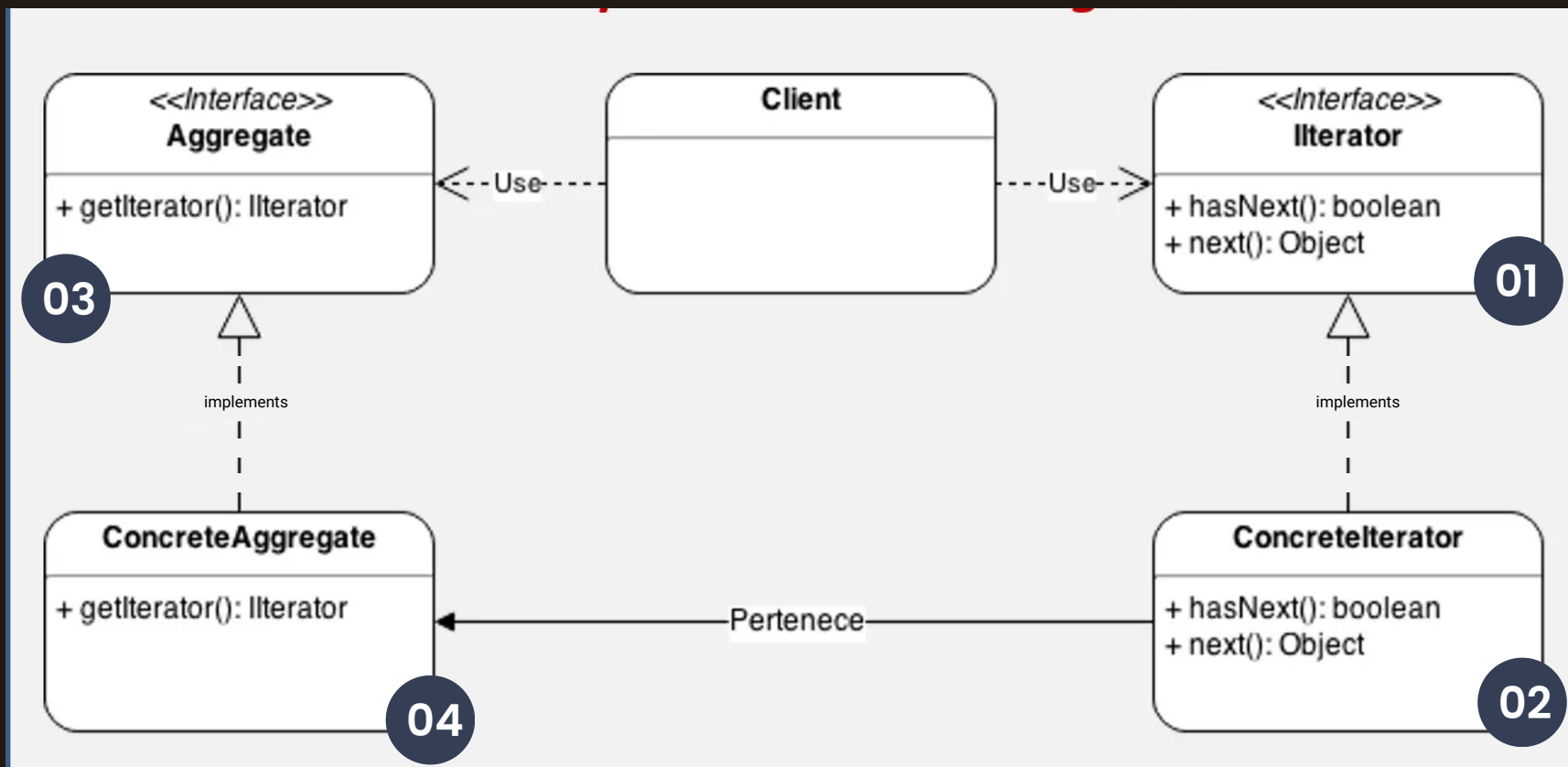
Analogía



Varias formas de pasear por Roma.

Estructura estática

Iterator



01

Define una interfaz para acceder secuencialmente a los elementos de una colección.

ConcreteIterator

02

Implementa la interfaz del Iterator y mantiene un seguimiento de la posición actual en la colección.

Estructura estática

Aggregate

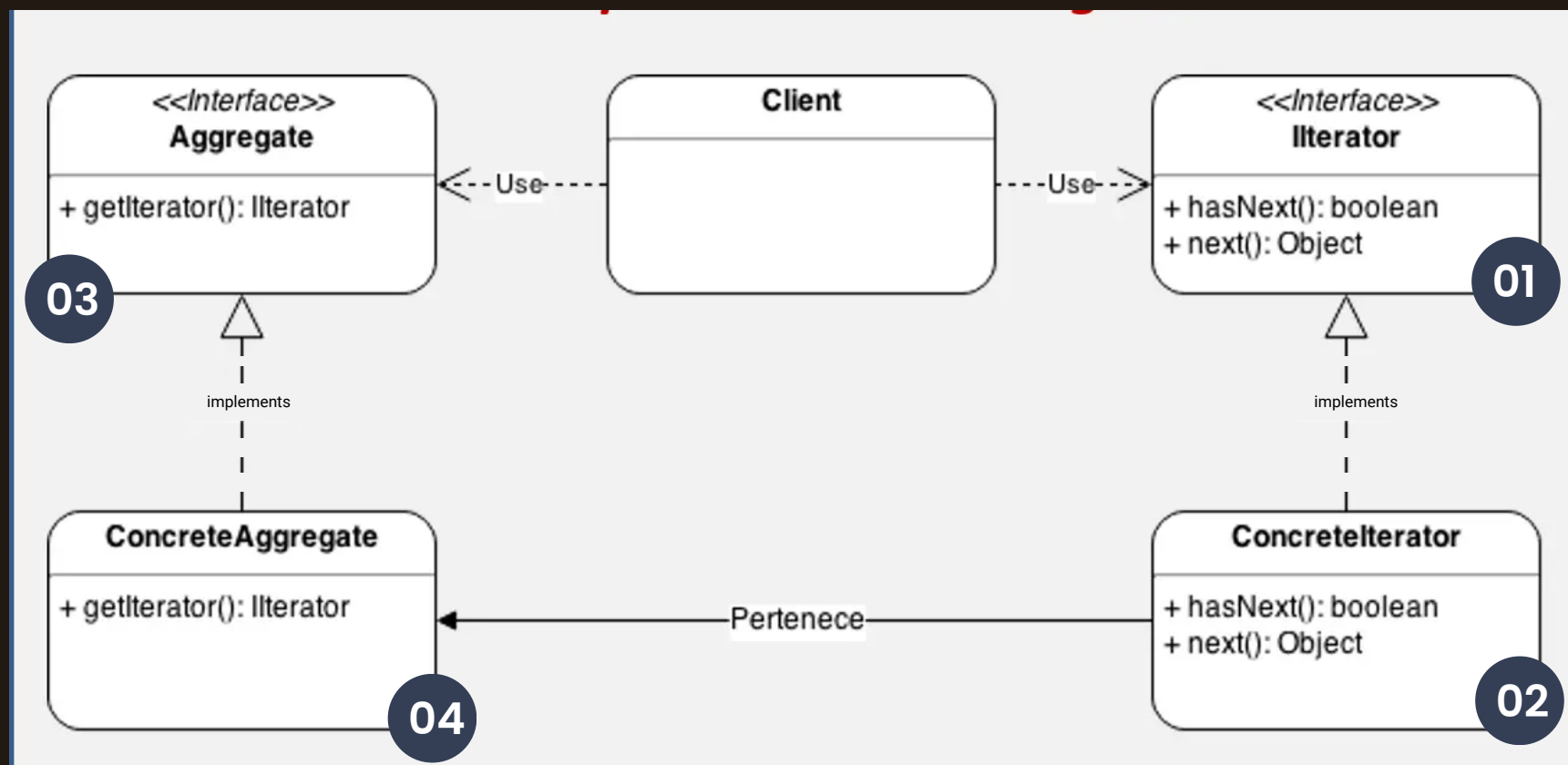
03

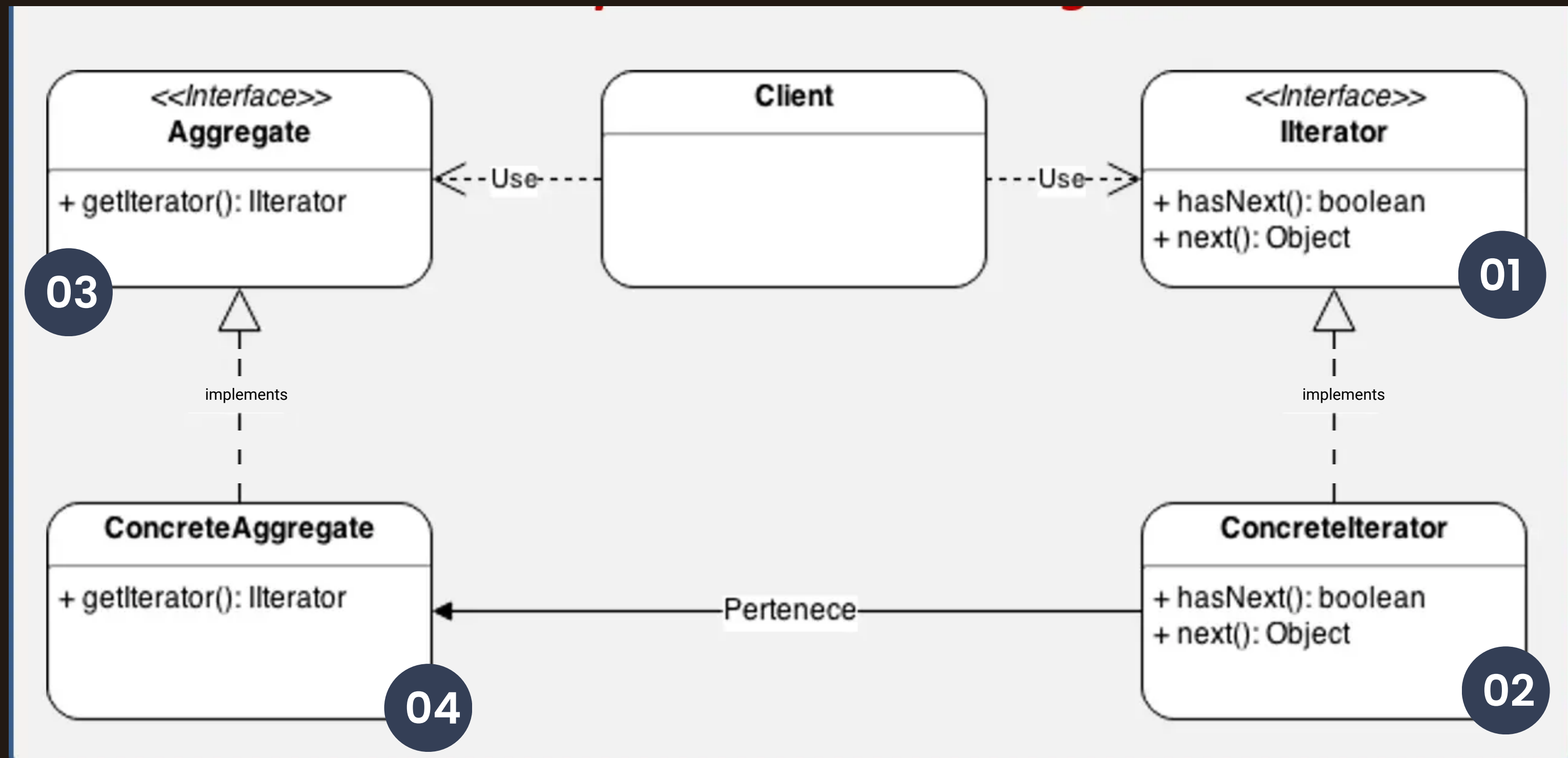
Define una interfaz para crear un objeto iterador.

ConcreteAggregate

04

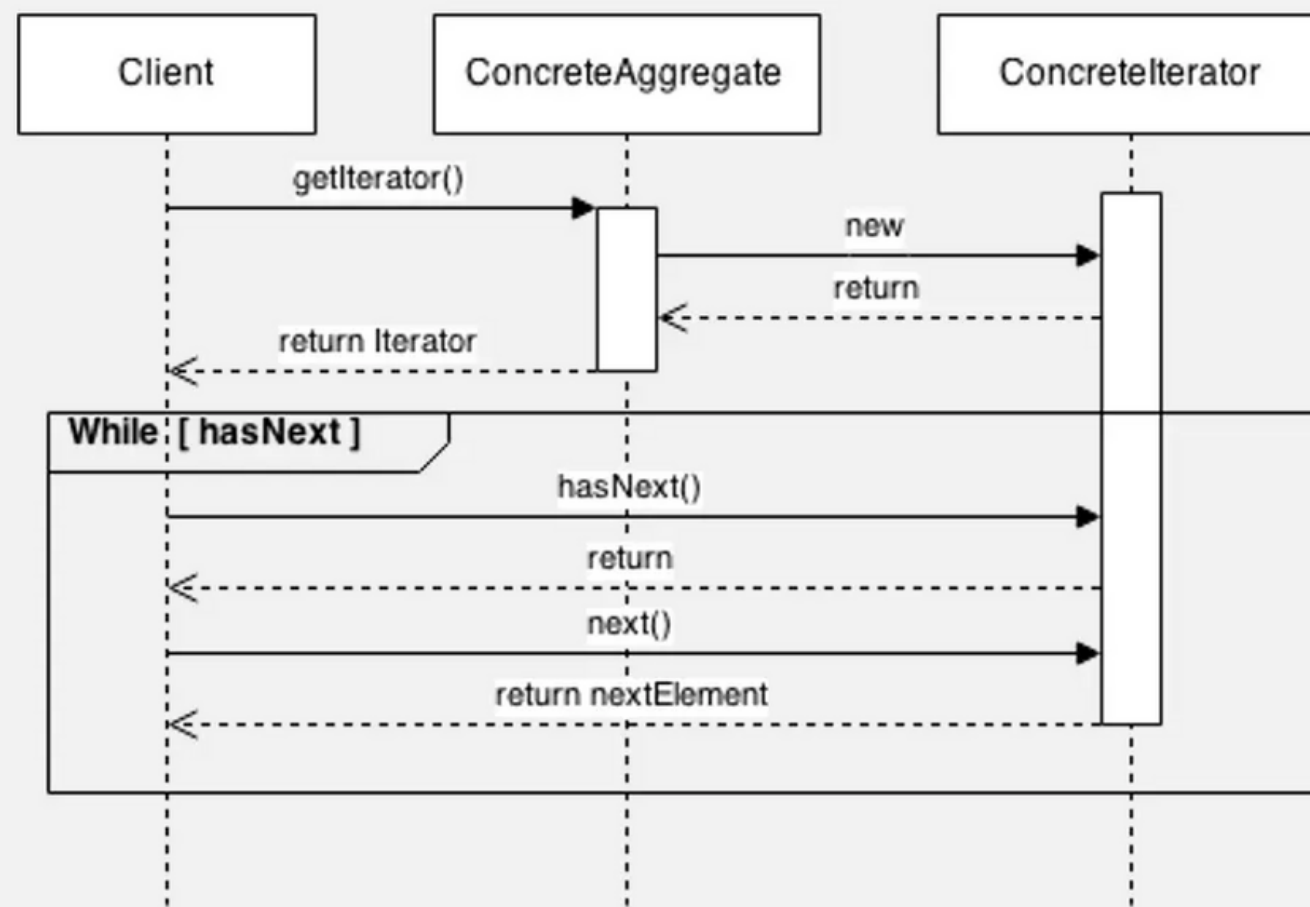
Implementa la interfaz del Aggregate y devuelve una instancia del ConcreteIterator que puede recorrer la colección.







Estructura dinámica



01

El **cliente** envía una solicitud para un iterador a ConcreteAggregate .

02

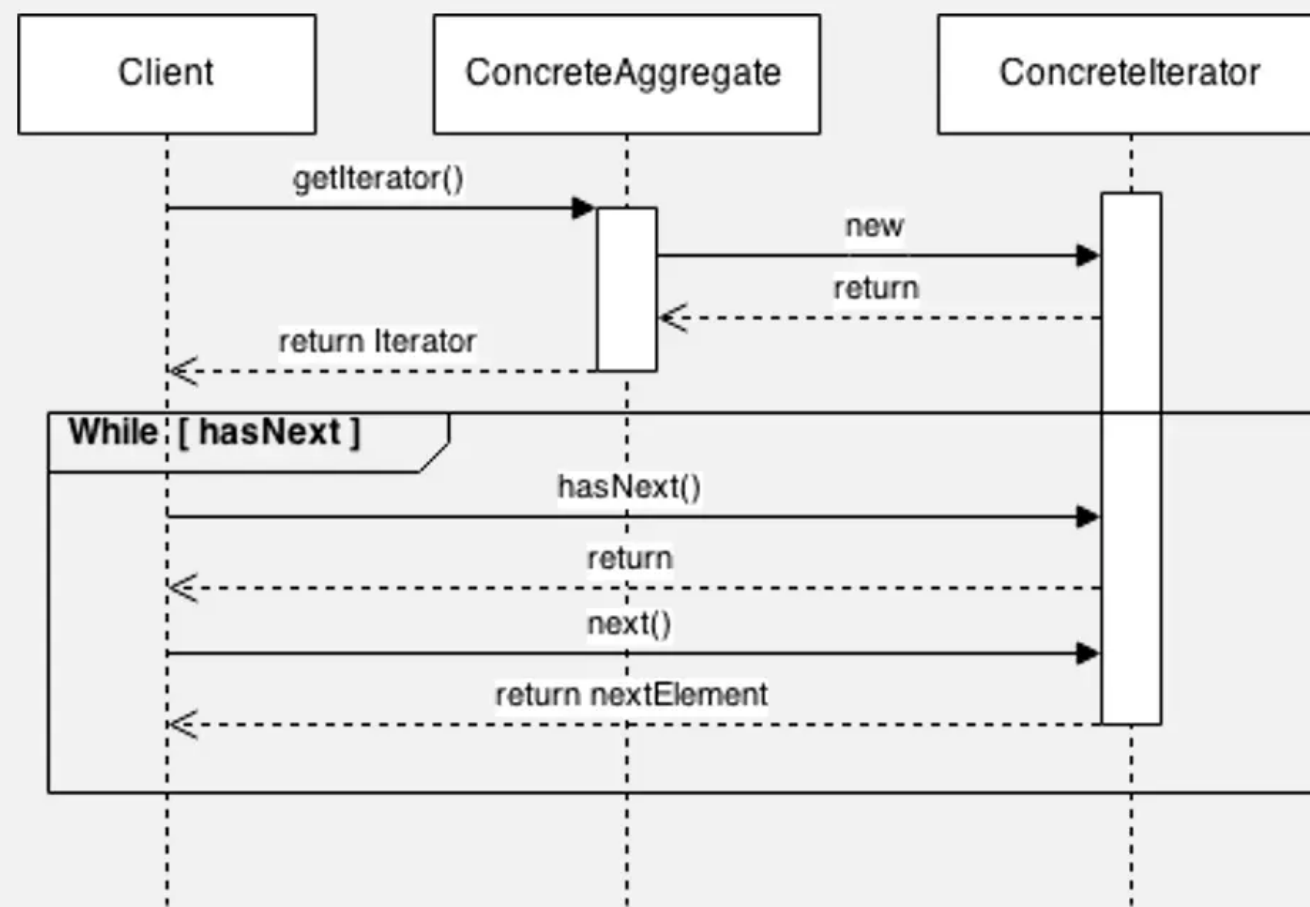
ConcreteAggregate crea un nuevo iterador.

03

El **cliente** revisa todos los elementos de la estructura, el bucle termina cuando no hay elementos para revisar, que va a ser señalado por el método **hasNext**.



Estructura dinámica



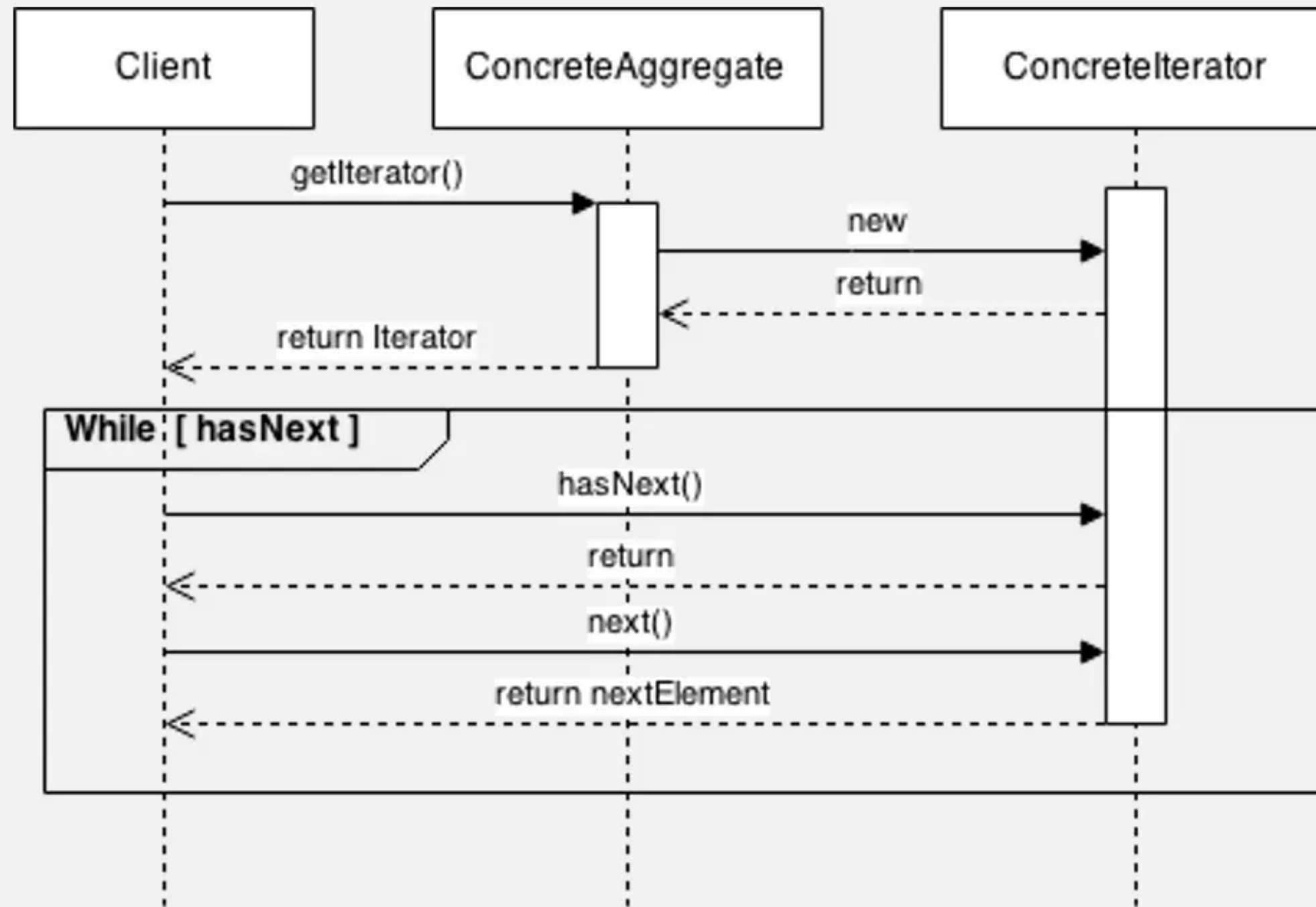
er

04

El **cliente** envía la solicitud de un nuevo elemento al iterador usando el siguiente método.

05

Si quedan elementos por revisar, volvemos al paso 3, algo que se repetirá hasta que se hayan revisado todos los elementos.



Colaboraciones

El iterador accede secuencialmente a los elementos de la colección sin conocer su representación interna.

Usos conocidos

En cualquier lugar donde se necesite recorrer los elementos de una colección de objetos sin exponer su representación interna. Por ejemplo, se puede utilizar para recorrer los elementos de una **lista enlazada** o una **tabla hash**.

Ven- ta- jas

Puede mejorar la eficiencia en el acceso a los elementos.

Proporciona una forma de recorrer los elementos de una colección sin conocer su representación interna.

Aumenta la flexibilidad y la reutilización del código.

Facilita el acceso a los elementos de la colección.

Desventaja

Puede haber un costo en términos de rendimiento debido al uso de un objeto iterador adicional. [1]

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. Páginas 289-304, 360-365.

Patrones relacionados



Ya que proporciona una forma de recorrer una estructura jerárquica de objetos, se relaciona con los patrones:

Visitor

Composite



Implementación

```
public class ListaEnlazada implements Iterable<Integer> {
    private Nodo inicio;

    public void agregar(int val) {
        if (inicio == null) {
            inicio = new Nodo(val);
        } else {
            Nodo actual = inicio;
            while (actual.siguiente != null) {
                actual = actual.siguiente;
            }
            actual.siguiente = new Nodo(val);
        }
    }

    @Override
    public Iterator<Integer> iterator() {
        return new Iterador(inicio);
    }
}
```

```
private static class Iterador implements Iterator<Integer> {
    private Nodo actual;

    public Iterador(Nodo nodo) {
        actual = nodo;
    }

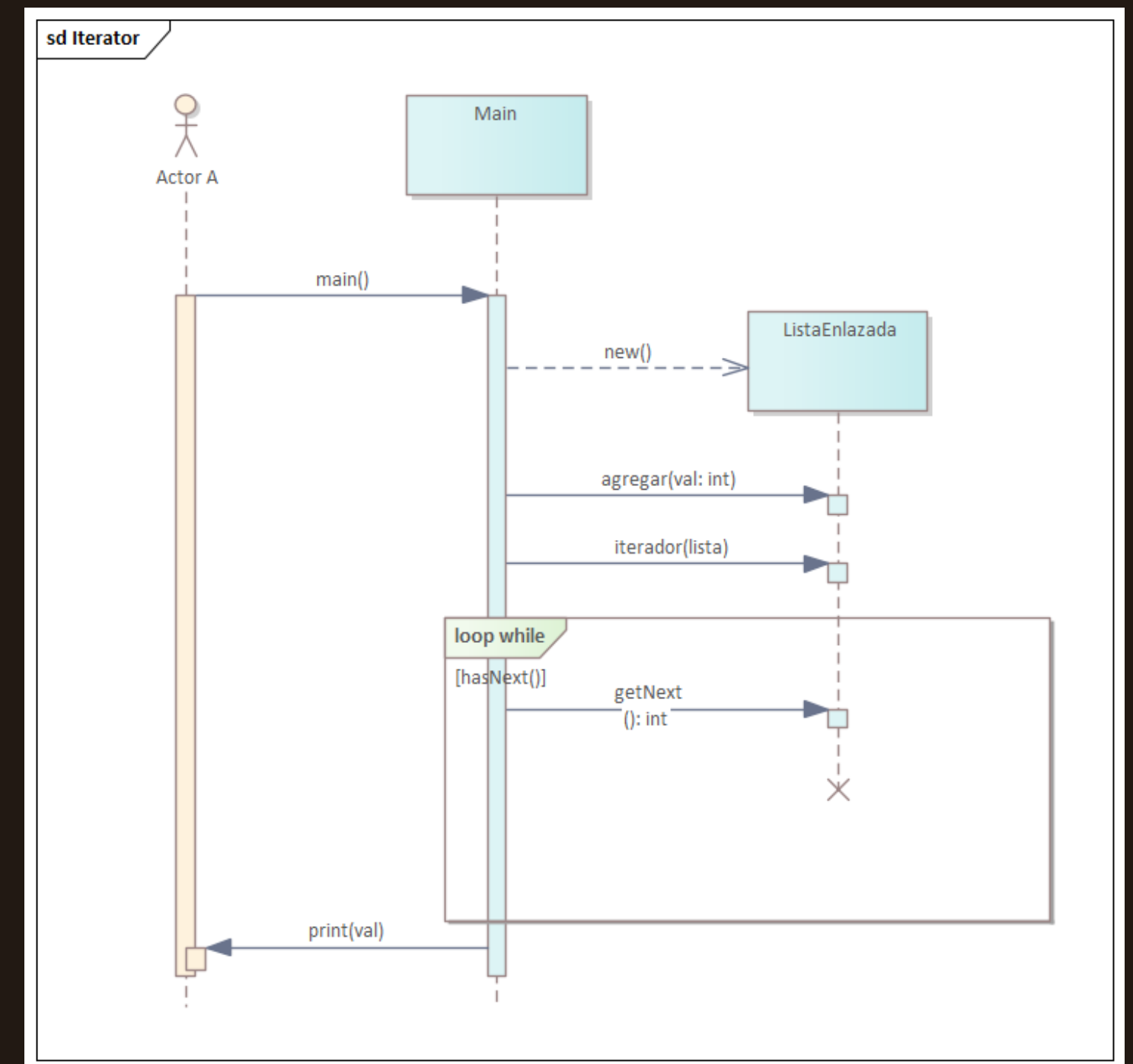
    @Override
    public boolean hasNext() {
        return actual != null;
    }

    @Override
    public Integer getNext() {
        if (!hasNext()) {
            return null;
        }
        int valor = actual.valor;
        actual = actual.siguiente;
        return valor;
    }
}
```

Implementación

```
public static void main(String[] args) {
    ListaEnlazada lista = new ListaEnlazada();
    lista.agregar(1);
    lista.agregar(2);
    lista.agregar(3);

    Iterator<Integer> iterador = lista.iterator();
    while (iterador.hasNext()) {
        int valor = iterador.getNext();
        System.out.println(valor);
    }
}
```





Template Method




Intensión

Definir la estructura básica de un algoritmo en una superclase y dejar que las subclases implementen los pasos específicos.

Problema

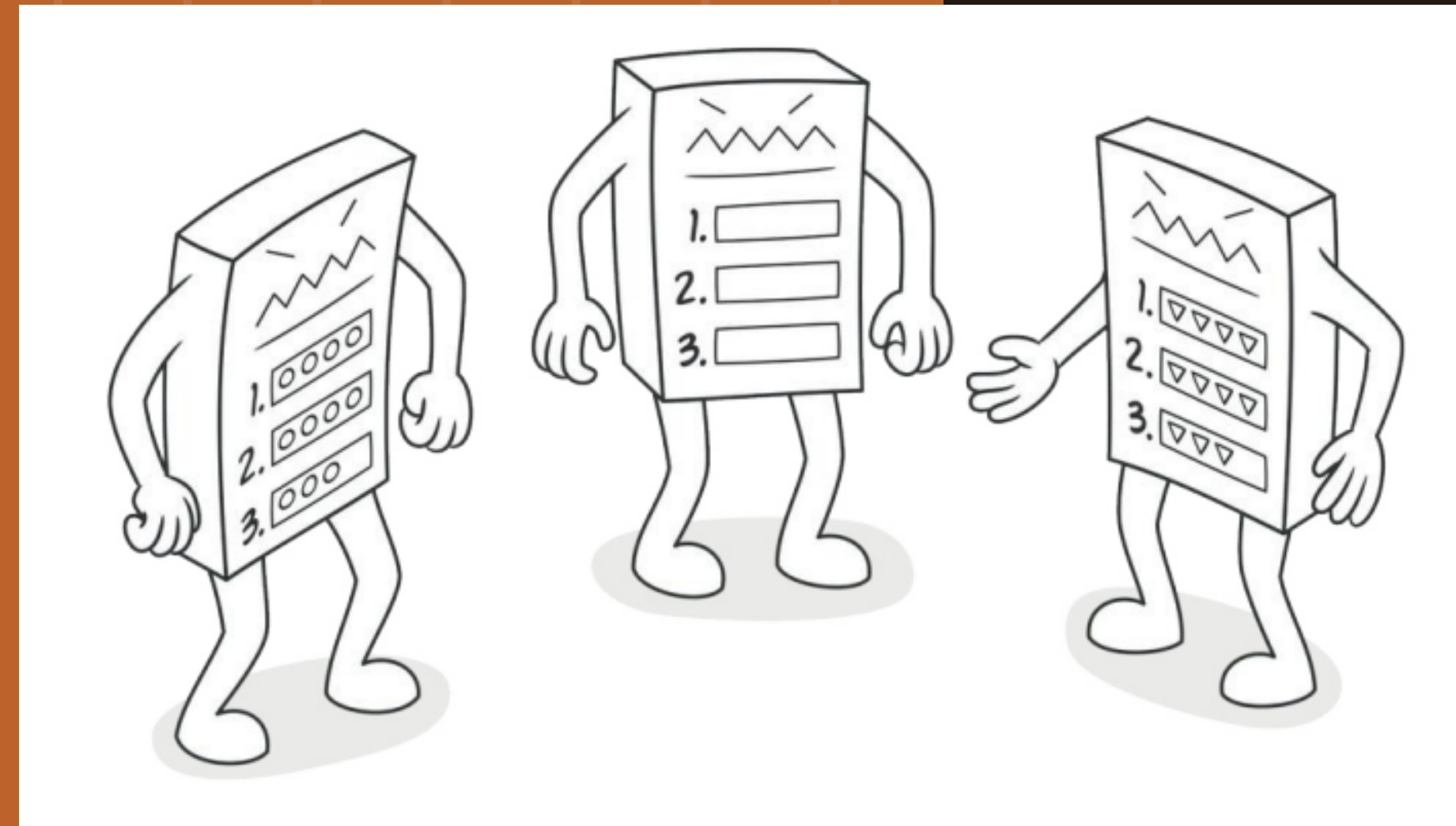
Cuando se tiene un algoritmo que sigue un proceso común, pero con variaciones en algunos de sus pasos, se pueden encontrar situaciones donde las subclases deben duplicar código para mantener la consistencia en el algoritmo, lo cual aumenta la complejidad y el riesgo de errores. [1]



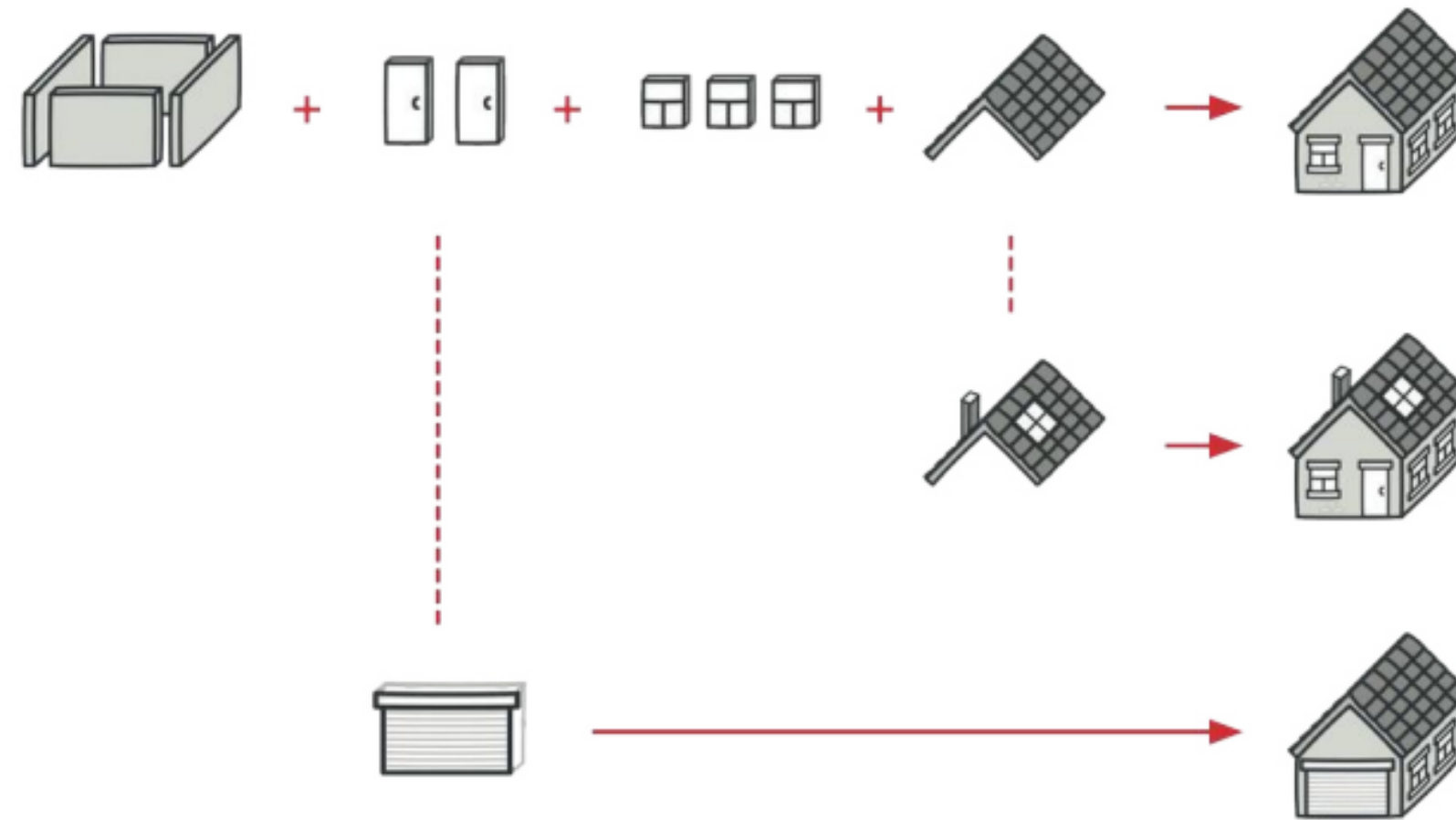
[1] Fowler, M. (2018). Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional. Páginas 205-209, 280-287.

Solución

Definir una operación de plantilla que defina la estructura general de un algoritmo y que contenga métodos abstractos que las subclases puedan implementar para proporcionar la implementación específica de algunos pasos. [1]

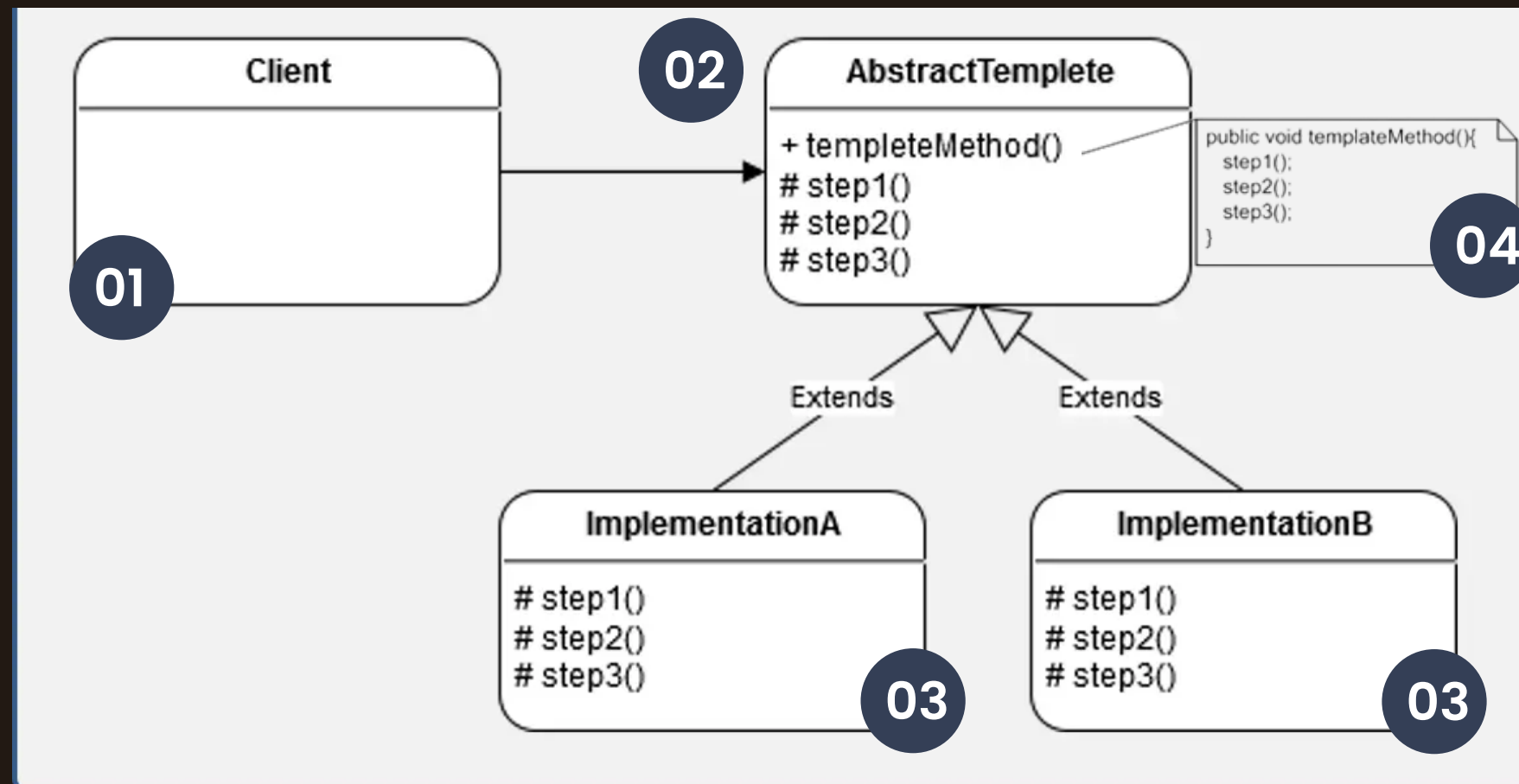


Analogía



Un plan arquitectónico típico puede alterarse ligeramente para que encaje mejor con las necesidades del cliente.

Estructura estática



Client

01

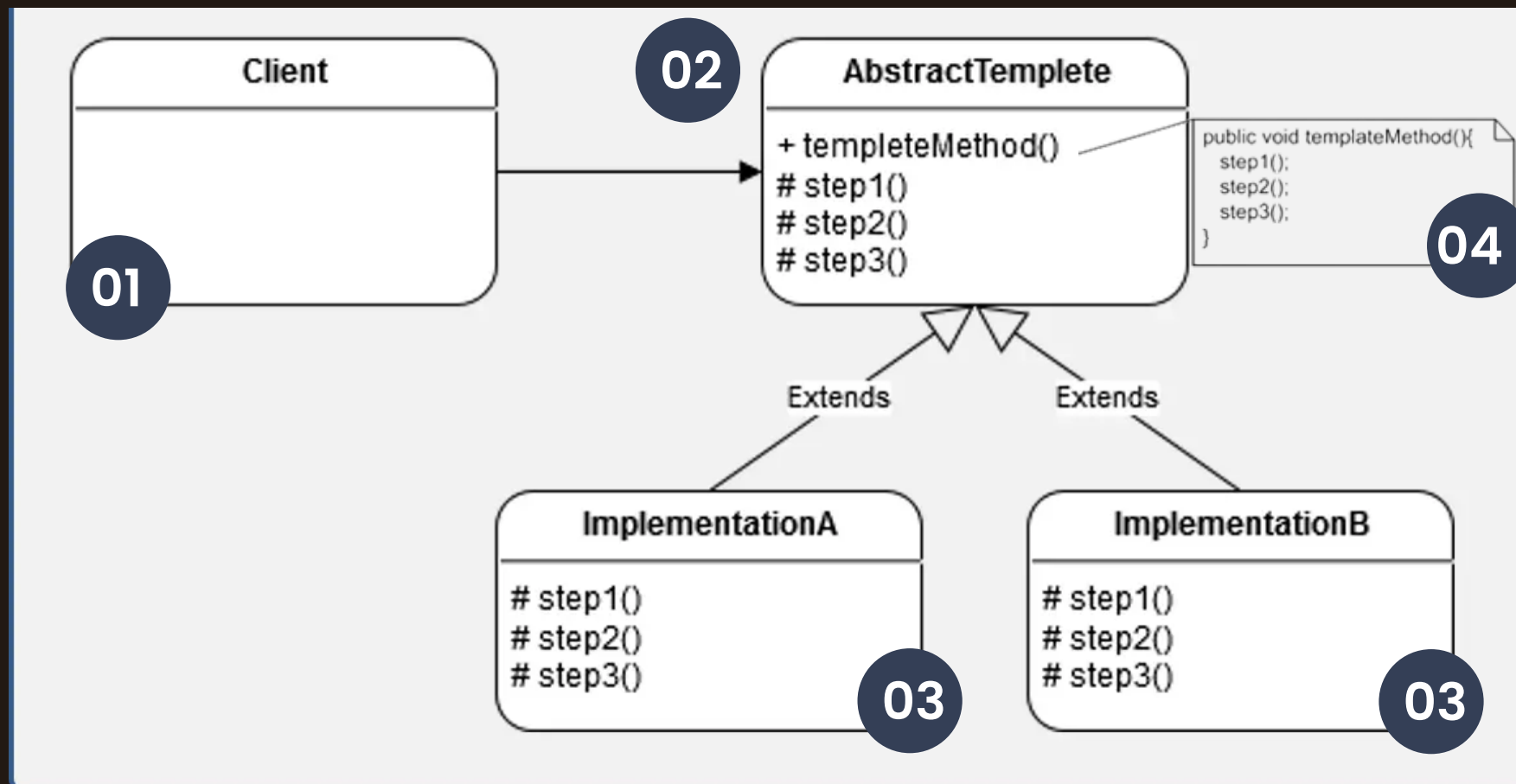
Es el componente que desencadena la ejecución de la plantilla.

AbstractTemplate

02

Clase abstracta que incluye operaciones que definen los pasos necesarios para llevar a cabo la ejecución del algoritmo. El método **templateMethod** ejecuta `step1`, `step2` y `step3` en orden.

Estructura estática



Implementación

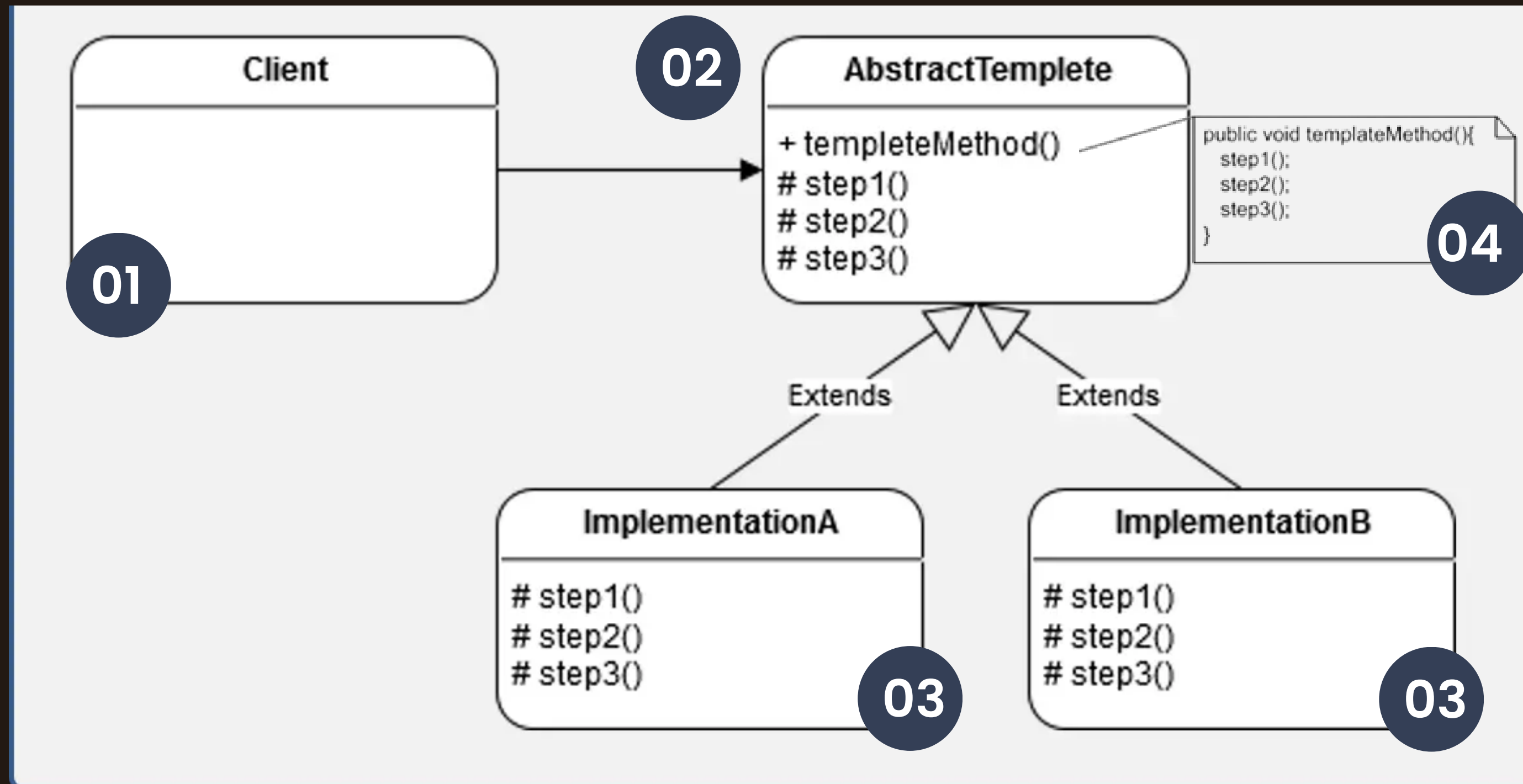
03

Representa una plantilla concreta que hereda de **AbstractTemplate** e implementa sus métodos.

templateMethod

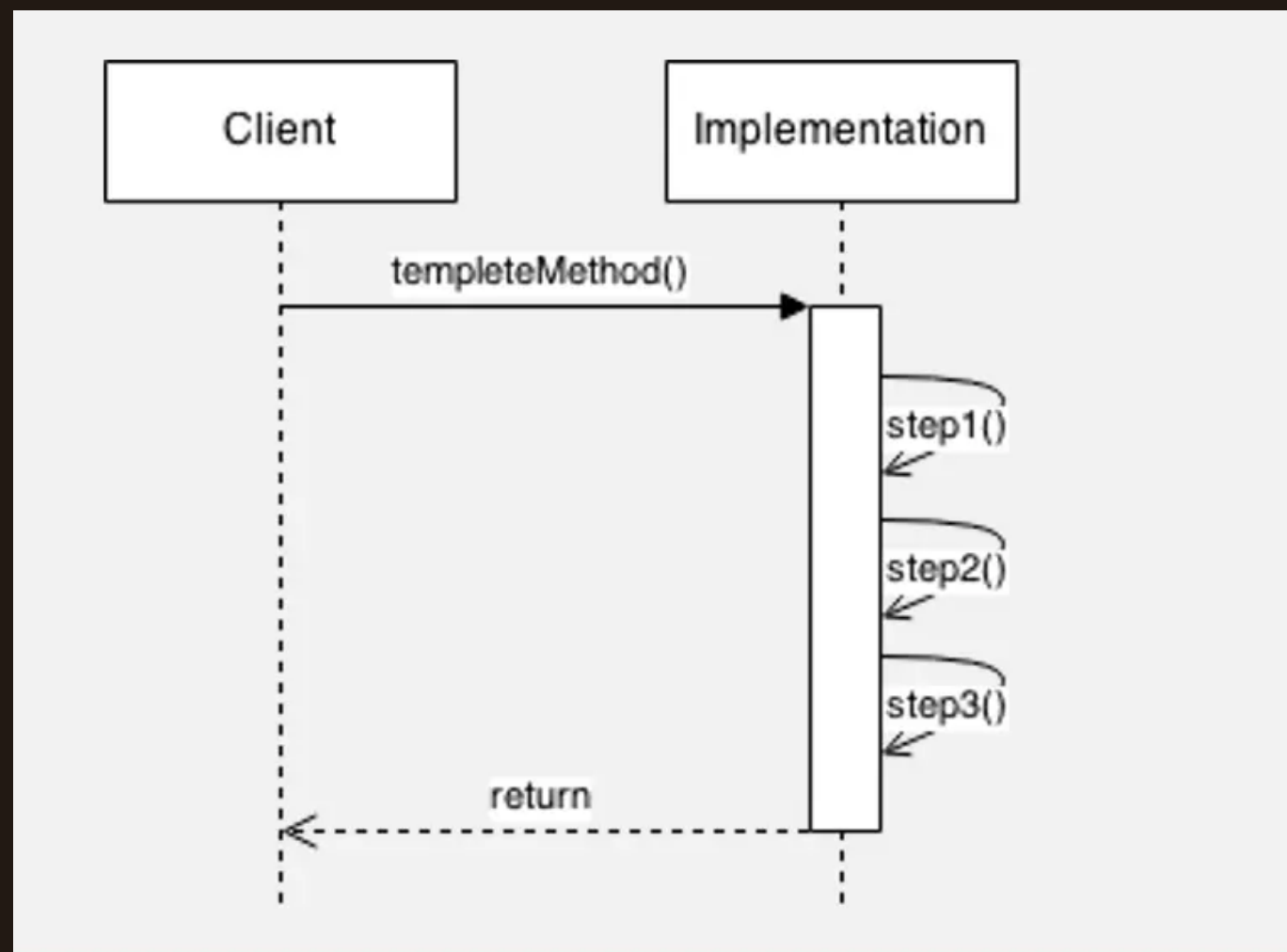
04

Define el algoritmo en términos de los métodos definidos en la clase abstracta y los métodos implementados por las subclases





Estructura dinámica



01

Client crea y obtiene una instancia de **Implementation**.

02

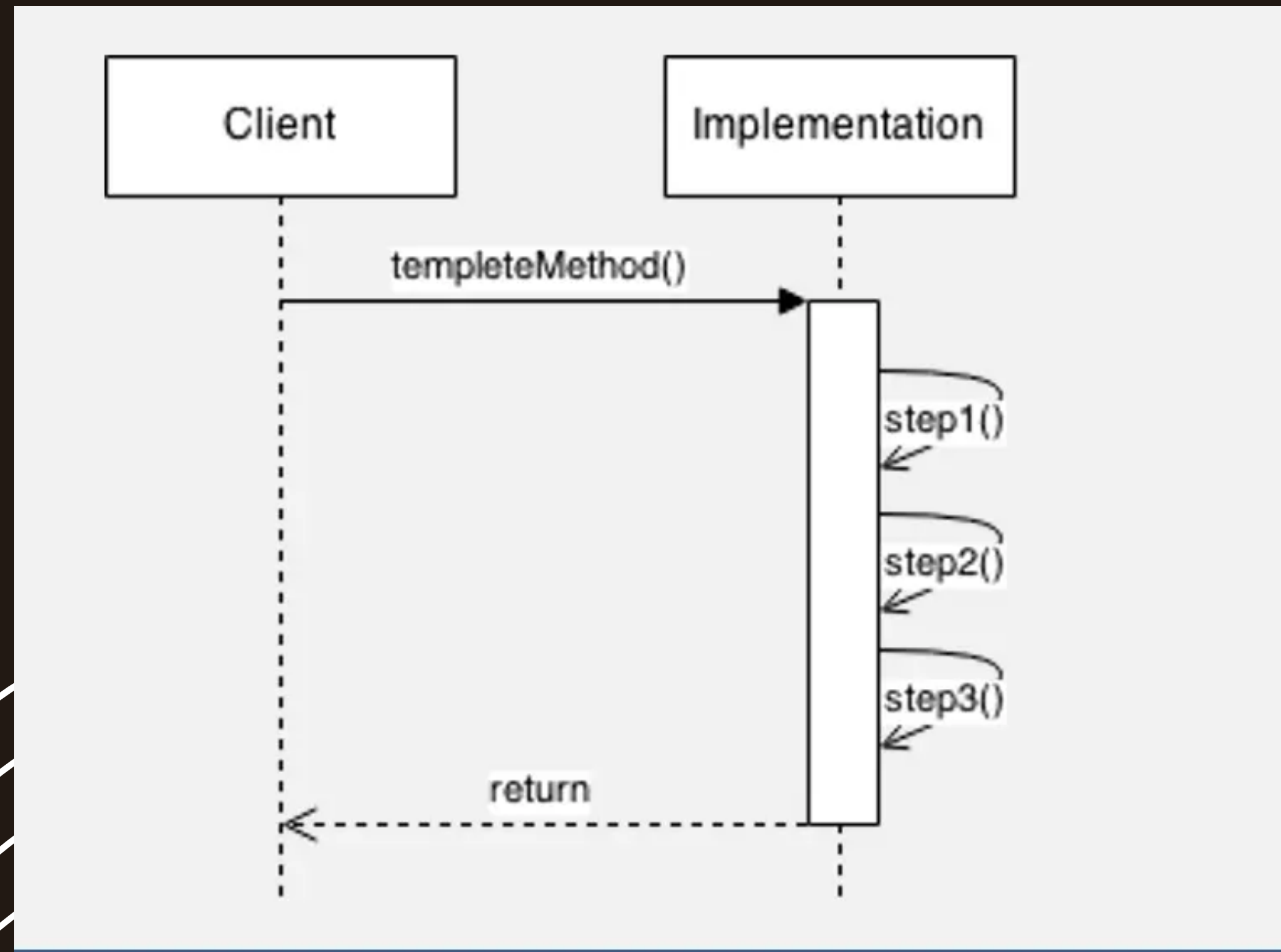
Client ejecuta el **templateMethod**.

03

La implementación predeterminada de **templateMethod** ejecuta los métodos **step1**, **step2** y **step3** en orden.

04

Implementation devuelve un resultado.



Colaboraciones

La clase TemplateMethod llama a los métodos definidos en la clase abstracta, así como a los métodos implementados por las subclases.

Usos conocidos

En el diseño de **marcos de aplicación**, donde la estructura general de la aplicación se define en una clase base, mientras que la implementación específica de algunas partes se delega a las subclases.

Ven- ta- jas

Permite el reúso de código

Controla el esqueleto del algoritmo mientras permite que las subclases cambien la implementación de ciertos pasos.

✦ Puede hacer que el código sea más complejo y difícil de seguir

Puede aumentar la complejidad al tener múltiples niveles de herencia. ✦

Desventajas

Patrones relacionados

Factory Method

Strategy

Comparten la idea de definir un esqueleto de algoritmo y permitir que las subclases o implementaciones concretas proporcionen detalles específicos.



Implementación

```
public abstract class SistemaOperativo {  
  
    protected abstract void cargarSistemaOperativo(String distro);  
    protected abstract void mostrarEscritorio(String escritorio);  
  
    private void encenderComputadora(){  
        System.out.println("Encendiendo Computadora...");  
    }  
  
    public final void iniciarSistemaOperativo(String distro) {  
        encenderComputadora();  
        cargarSistemaOperativo(distro);  
        mostrarEscritorio(distro);  
    }  
}
```


Implementación

```
public class Windows extends SistemaOperativo {  
    @Override  
    protected void cargarSistemaOperativo(String distro) {  
        System.out.println("Iniciar firmware UEFI");  
        System.out.println("Cargar núcleo Windows NT");  
        System.out.println("Arranca Windows Boot Manager");  
    }  
  
    @Override  
    protected void mostrarEscritorio(String escritorio) {  
        System.out.println("Muestra escritorio Windows");  
    }  
}
```



Implementación

```
public class MacOS extends SistemaOperativo {  
  
    @Override  
    protected void cargarSistemaOperativo(String distro) {  
        System.out.println("Iniciar firmware EFI");  
        System.out.println("Cargar núcleo XNU");  
        System.out.println("Arranca Apple bootloader");  
    }  
  
    @Override  
    protected void mostrarEscritorio(String escritorio) {  
        System.out.println("Muestra escritorio MacOS");  
    }  
}
```

Implementación

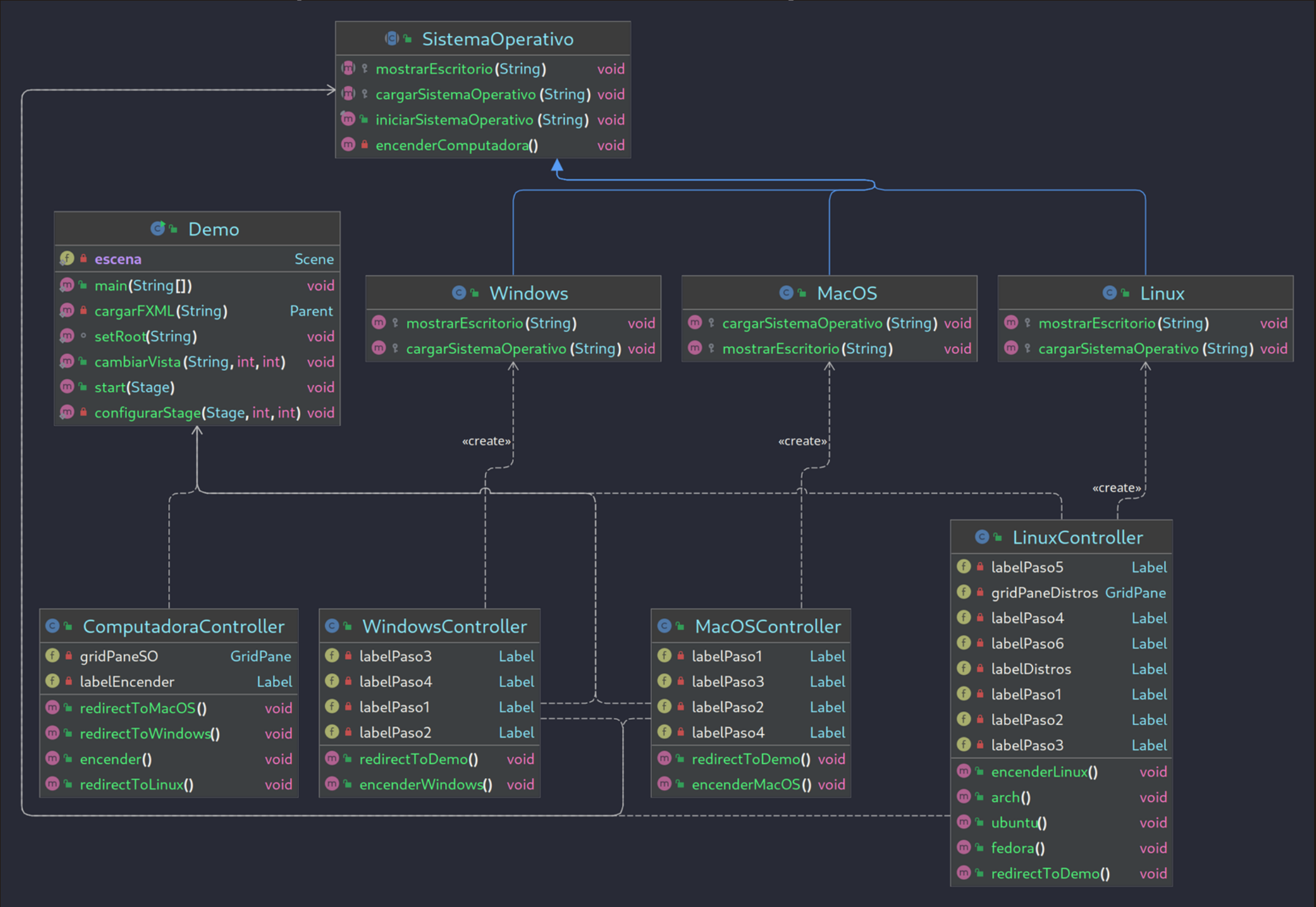
```
public class Linux extends SistemaOperativo {

    @Override
    protected void cargarSistemaOperativo(String distro) {
        System.out.println("Inicia POST...");
        System.out.println("Cargar UEFI...");
        System.out.println("Arranca bootloader GRUB...");
        switch (distro) {
            case "Arch":
                System.out.println("Cargar kernels de Arch");
                break;
            case "Ubuntu":
                System.out.println("Cargar kernels de Ubuntu");
                break;
            case "Fedora":
                System.out.println("Cargar kernels de Fedora");
                break;
        }
        System.out.println("Se ejecuta Init...");
    }
}
```

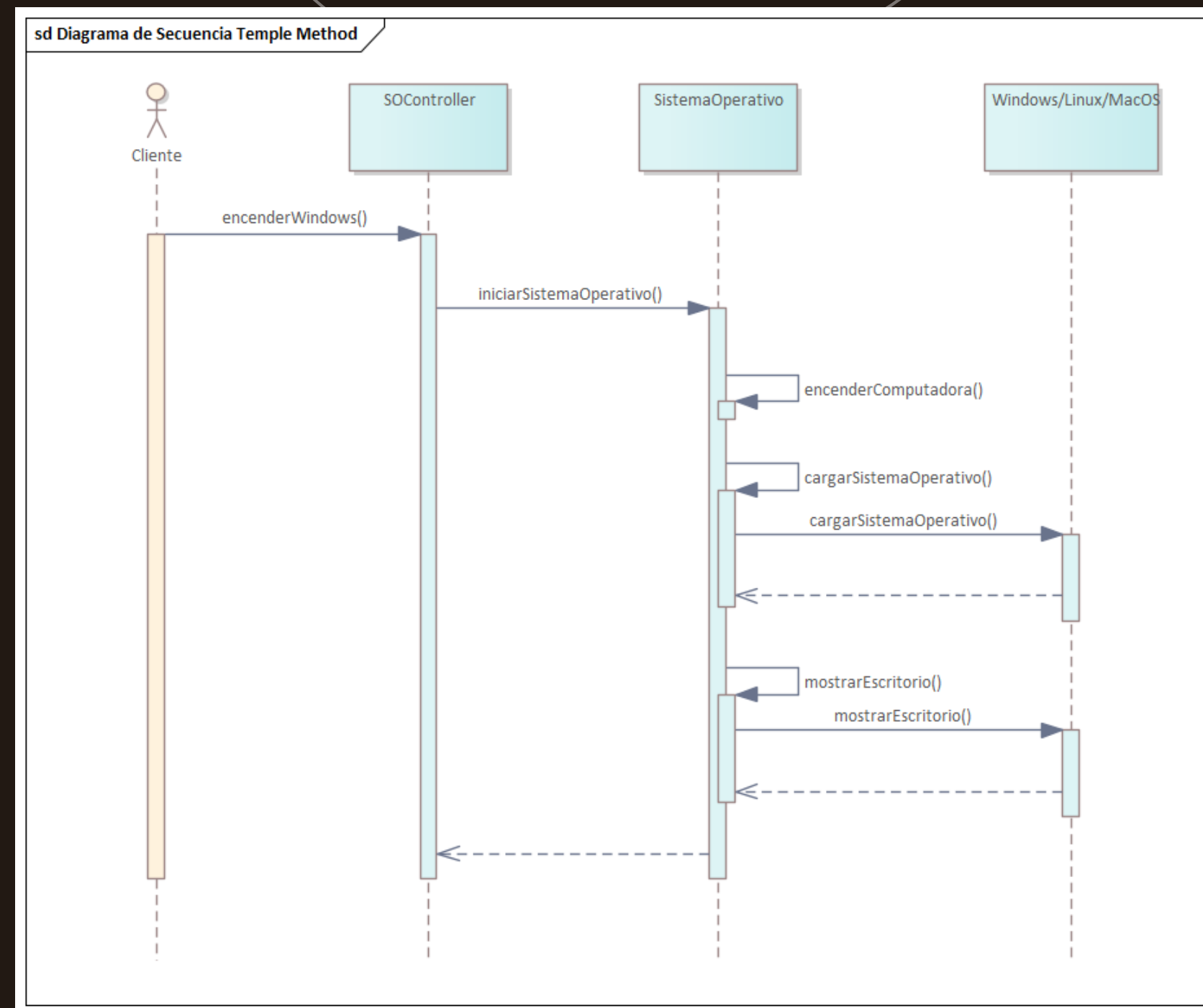
```
    @Override
    protected void mostrarEscritorio(String escritorio) {
        switch (escritorio) {
            case "Arch":
                System.out.println("Muestra escritorio GNOME");
                break;
            case "Ubuntu":
                System.out.println("Muestra escritorio KDE");
                break;
            case "Fedora":
                System.out.println("Muestra escritorio XCFE");
                break;
        }
    }
}
```



Implementación



Implementación



Conclusiones

Los patrones Iterator y Template Method son importantes en el desarrollo de software. Iterator permite recorrer colecciones de objetos de manera eficiente, mientras que Template Method define la estructura general de un algoritmo.

Ambos ayudan a reducir la duplicación de código y mejorar la mantenibilidad del programa.





Ingeniería de software

¡Gracias!

Equipo 5