

Universidad del Valle de Guatemala

Ingeniería de Software 1

Prof. Erick Marroquín

Tarea Investigativa 1

Juan Diego Solís Martínez - 23720

Nils Muralles Morales - 23727

Víctor Manuel Pérez Chávez - 23731

Diego Oswaldo Flores Rivas - 23714

Isabella Recinos Rodríguez- 23003

Guatemala, 5 de marzo de 2025

Resumen

Los patrones de diseño son técnicas y modelos organizacionales que buscan resolver problemas relacionados con el desarrollo de software. Existen múltiples tipos de patrones, creacionales, estructurales, de comportamiento y de acceso a datos. Entre los patrones explorados en esta tarea investigativa se encuentran los siguientes: Proxy, el cual provee un servicio intermediario entre un objeto y otro que desea interactuar con él; iterator, el cual permite recorrer distintos tipos de colecciones de datos sin exponer su estructura interna; y strategy, el cual busca establecer una familia de algoritmos intercambiables y encapsulables los unos en los otros.

Introducción

En el proceso de desarrollo de software, es importante contar con esquemas de código que sean escalables y mantenibles a lo largo del tiempo. Dichos esquemas son conocidos como patrones de diseño, los cuales ayudan a resolver problemas frecuentemente reportados durante el proceso de desarrollo. Por esta razón, en este trabajo se exploran los patrones proxy, iterator y strategy; con el fin de discutir su intención, motivación, aplicaciones, importancia y ventajas a la hora de llevar a cabo desarrollo de software moderno.

Patrones de diseño

1. Proxy

a. Intención:

El propósito del patrón proxy es proveer un servicio intermediario que controla el acceso al objeto en sí, con el objetivo de agregar lógica de programación al mismo en función de las necesidades funcionales con las que se quiera utilizar dicho objeto (Reactive Programming, 2021).

b. Conocido como:

El patrón también es conocido como surrogate (sustituto en español), wrapper cuando modifica el comportamiento de un objeto e intermediario cuando se utiliza para añadir seguridad (Reactive Programming, 2021).

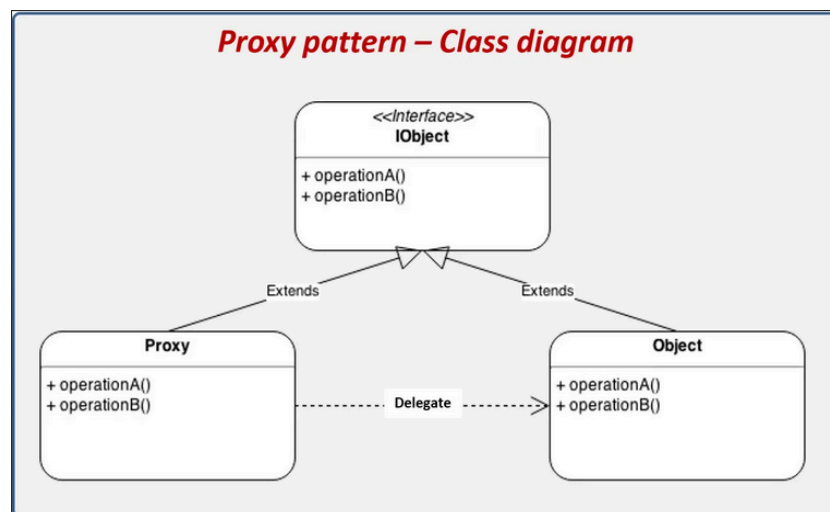
c. Motivo:

Se considera un patrón de diseño estructural debido a que proporciona una guía para la creación y utilización de objetos y clases que permiten controlar el acceso a los objetos. Este diseño es mantenible, escalable y cumple con el principio de responsabilidad única (Refactoring Guru, 2025).

d. Aplicaciones:

Entre las aplicaciones de este patrón se encuentran: Seguridad y autenticación, dado que un proxy puede ser utilizado para verificar datos del usuario antes de permitirle acceder al objeto sobre el que se realiza el proxy; Optimización de procesos, evitando el acceso a un objeto hasta que deba ser utilizado; y proxy remoto, donde otro servidor cuenta con una implementación del objeto (Refactoring Guru, 2025).

e. Estructura:



(Reactive Programming, 2021).

f. Participantes:

- i. IObject: Interfaz que define las funcionalidades que tanto el proxy como el objeto implementarán.
- ii. Object: Proporciona la implementación de la interfaz IObject y realiza las tareas de manera predeterminada.
- iii. Proxy: También implementa la interfaz IObject, pero a su vez delega las tareas y controla el acceso al objeto Object en sí. Simplemente, hace referencia al mismo y añade lógica por encima.

(Reactive Programming, 2021).

g. Colaboraciones:

- i. IObject (Interfaz Común):
 1. Define las funcionalidades que tanto el Proxy como el Object deben implementar.
 2. Garantiza que el cliente interactúe con ambos de manera transparente sin conocer la diferencia.
- ii. Object (Objeto Real):
 1. Proporciona la implementación concreta de la interfaz IObject.
 2. Ejecuta las tareas de manera predeterminada y sin restricciones adicionales.
- iii. Proxy:
 1. Implementa la interfaz IObject, actuando como intermediario entre el cliente y el objeto real.
 2. Puede realizar tareas adicionales antes y después de llamar al Object, como autenticación, registro de acceso, carga diferida o control de concurrencia.
 3. Mantiene una referencia interna al Object para delegar tareas cuando sea necesario.

(Reactive Programming, 2021).

h. Consecuencias:

- i. Beneficios
 1. Carga diferida: Permite retrasar la creación de objetos costosos en términos de recursos hasta que realmente sean necesarios. De esta forma, mejora la eficiencia al evitar asignaciones innecesarias de memoria y procesamiento.
 2. Control de acceso y seguridad: Permite restringir el acceso a objetos sensibles o que requieren permisos específicos.
 3. Caché y optimización de rendimiento: Puede almacenar en caché datos para evitar operaciones repetitivas sobre el objeto real.
 4. Registro y monitoreo de acciones: Intercepta las llamadas al objeto real para registrar eventos, métricas o depuración.

5. Manejo de objetos remotos en sistemas distribuidos: Facilita la comunicación con objetos ubicados en diferentes direcciones o servidores.

ii. Riesgos y posibles inconvenientes

1. Sobrecarga en operaciones simples: Para objetos livianos o de creación rápida, el uso de un proxy añade latencia sin aportar beneficios reales.
2. Complejidad innecesaria en el diseño: Si la aplicación no requiere control de acceso, caché o carga diferida, un proxy solo añade más abstracción.
3. Impacto negativo en el rendimiento: En algunos casos, la capa adicional de proxy puede hacer que las llamadas sean más lentas que acceder directamente al objeto real.
4. No es útil si no hay control de acceso: Si el código cliente ya puede interactuar con el objeto sin restricciones, un proxy solo introduce una barrera innecesaria.
5. Puede no ser necesario cuando la carga anticipada no es un problema: Si la creación anticipada de objetos no afecta el rendimiento del sistema, el proxy para carga diferida es innecesario.

(GeeksforGeeks, 2017)

i. Implementación:

- i. Crear la interfaz del objeto real: Definir una interfaz o clase abstracta que representa las operaciones que serán proporcionadas por el objeto real. Esta interfaz será implementada tanto por el objeto real como el proxy.
- ii. Crear el objeto real: Clase que implementa la interfaz y contiene la lógica o las operaciones reales que el cliente desea utilizar.
- iii. Crear la clase proxy: La clase proxy implementa la misma interfaz que el objeto real. Mantiene una referencia al objeto real y controla el acceso a él. También el proxy puede agregar lógica adicional, como registro de eventos (logging), almacenamiento en caché (caching) o verificaciones de seguridad antes de llamar a los métodos del objeto real.
- iv. El cliente usa el proxy: El cliente interactúa con el proxy en lugar de crear el objeto real directamente. El proxy decide cuándo y cómo reenviar la solicitud del cliente al objeto real.

(GeeksforGeeks, 2017).

j. Código de ejemplo:

Interfaz IObject (Subject)

```
1 // Interfaz IObject (Subject)
2 interface IObject {
3     void operationA();
4     void operationB();
5 }
6
```

Clase Object (Real Subject)

```
7 // Clase Object (Real Subject)
8 class Object implements IObject {
9     private String name;
10
11     public Object(String name) {
12         this.name = name;
13         loadFromDisk();
14     }
15
16     private void loadFromDisk() {
17         System.out.println("Cargando objeto: " + name);
18     }
19
20     public void operationA() {
21         System.out.println("Ejecutando operación A en: " + name);
22     }
23
24     public void operationB() {
25         System.out.println("Ejecutando operación B en: " + name);
26     }
27 }
28
```

Clase Proxy (Proxy)

```
29 // Clase Proxy (Proxy)
30 class Proxy implements IObject {
31     private Object realObject;
32     private String name;
33
34     public Proxy(String name) {
35         this.name = name;
36     }
37
38     public void operationA() {
39         if (realObject == null) {
40             System.out.println("Creando instancia de Object en Proxy...");
41             realObject = new Object(name);
42         } else {
43             System.out.println("Usando objeto en caché en Proxy...");
44         }
45         realObject.operationA();
46     }
47
48     public void operationB() {
49         if (realObject == null) {
50             System.out.println("Creando instancia de Object en Proxy...");
51             realObject = new Object(name);
52         } else {
53             System.out.println("Usando objeto en caché en Proxy...");
54         }
55         realObject.operationB();
56     }
57 }
58
```

Cliente

```
59 // Cliente
60 public class ProxyPatternExample {
61     public static void main(String[] args) {
62         IObject proxy = new Proxy("Ejemplo");
63
64         System.out.println("Primera ejecución de operationA():");
65         proxy.operationA(); // Crea y ejecuta
66
67         System.out.println("\nSegunda ejecución de operationA():");
68         proxy.operationA(); // Usa caché
69
70         System.out.println("\nEjecución de operationB():");
71         proxy.operationB(); // Usa caché
72     }
73 }
```

(GeeksforGeeks, 2017).

k. Usos conocidos:

- i. Sistemas de seguridad: Implementación de mecanismos que interceptan ejecuciones de procesos para validar si el usuario tiene los privilegios necesarios, evitando que usuarios no autorizados los ejecuten.
- ii. Optimización de recursos: Creación de objetos ligeros mediante la abstracción de partes reutilizables que pueden ser compartidas con otros objetos, reduciendo la capacidad de memoria requerida por la aplicación.
- iii. Control de acceso en redes: Los servidores proxy filtran tráfico o actúan como intermediarios entre el usuario e Internet, controlando y monitoreando el acceso a recursos.

(GeeksforGeeks, 2017).

l. Patrones relacionados:

- i. Adapter: Permite la conversión de una interfaz a otra, facilitando la integración de un objeto con un sistema diferente, similar al Proxy que controla el acceso a un objeto, pero en este caso adaptando las interfaces.
- ii. Bridge: Permite la separación de una abstracción y su implementación, algo similar al Proxy, ya que también puede funcionar como intermediario, pero sin la necesidad de acceder a un objeto de manera directa.
- iii. Composite: Estructura objetos en una jerarquía de partes y todo, lo que puede estar relacionado con el Proxy cuando el proxy actúa sobre un conjunto de objetos, gestionando su acceso de forma colectiva o individual.
- iv. Decorator: Modifica dinámicamente el comportamiento de un objeto, de manera parecida al Proxy, que también puede añadir funcionalidades adicionales (por ejemplo, control de acceso, validación, etc.) sin modificar el objeto original.
- v. Facade: Proporciona una interfaz simplificada para un sistema complejo, similar al Proxy, que actúa como una puerta de entrada controlada a un objeto o conjunto de objetos complejos.
- vi. Flyweight: Se enfoca en compartir objetos para economizar recursos, lo cual es un concepto relacionado con Proxy, ya que un proxy puede crear un objeto de forma perezosa (lazy instantiation), compartiendo su instancia cuando es necesario.

(Patrones Estructurales, 2025)

2. Iterator

a. Intención:

Es un patrón de diseño de comportamiento que permite recorrer los elementos de una colección de manera secuencial sin exponer su estructura interna (ya sea una lista, pila, árbol u otra). Facilita la navegación sobre distintos tipos de colecciones de forma uniforme, creando un método estándar para acceder a sus elementos sin necesidad de modificar su implementación interna (GeeksforGeeks, 2025).

b. Conocido como:

También es conocido como Iterador por su traducción al español (Spring Sale, 2025).

c. Motivo:

Las motivaciones que lo justifican como patrón son:

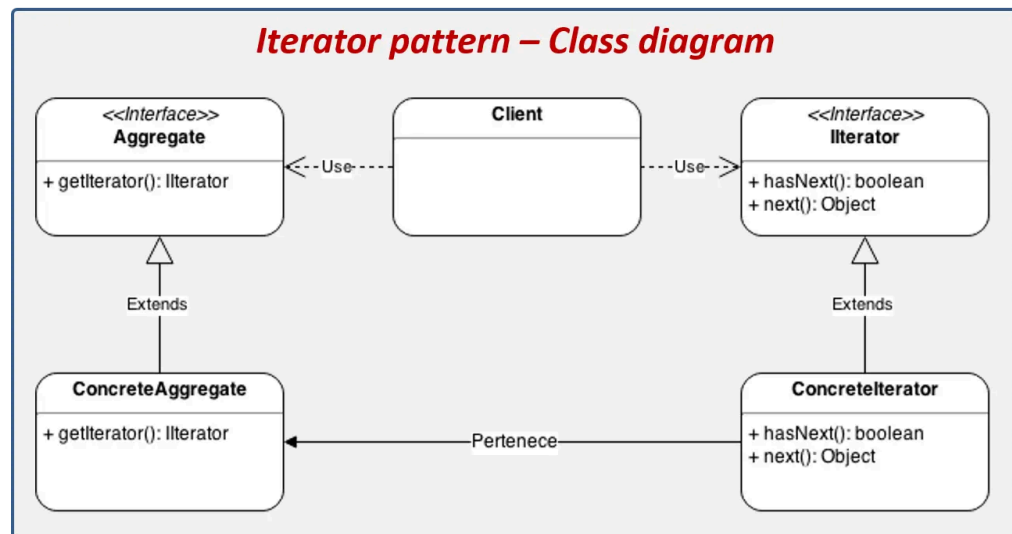
- i. Resuelve el recorrido de estructuras de datos sin exponer su implementación interna.
- ii. Encapsula la lógica de iteración, permitiendo un acceso uniforme a diferentes colecciones.
- iii. Ayuda a separar la estructura de los datos de la forma en que se recorren.
- iv. Mejora la reutilización y el mantenimiento del código.
(GeeksforGeeks, 2025)

d. Aplicaciones:

- i. En bibliotecas de colecciones, como Iterator en Java, foreach en C#, e iter() en Python.
- ii. En bases de datos para recorrer registros sin cargar todo en memoria.
- iii. En frameworks de UI para recorrer elementos de una interfaz (botones, inputs, etc.).
- iv. En procesamiento de archivos para leer líneas sin cargar el archivo completo.
- v. En inteligencia Artificial para recorrer árboles de decisión o grafos.

(Spring Sale, 2025).

e. Estructura:



(Blancarte, 2024).

f. Participantes:

- i. Client: Actor que utiliza al Iterator para recorrer una colección sin conocer su estructura interna.
- ii. Aggregate: Es la interfaz que define las estructuras que pueden ser recorridas con un iterador.
- iii. ConcreteAggregate: Es la clase que contiene la estructura de datos que se desea recorrer.
- iv. Iterator: Es la interfaz que define los métodos necesarios para recorrer una colección, como `hasNext()` para saber si hay más elementos y `next()` para obtener el siguiente elemento.
- v. ConcreteIterator: Es la implementación específica del iterador, que sabe exactamente cómo recorrer una **ConcreteAggregate**.

(Blancarte, 2024).

g. Colaboraciones:

- i. Las clases de **ConcreteAggregate** y **ConcreteIterator** son implementaciones de las interfaces, que contienen todos sus atributos y métodos específicos. **ConcreteAggregate** debe contener un método para retornar su iterador específico.
- ii. El Cliente interactúa con las clases concretas a través de sus interfaces, para poder intercambiar fácilmente las implementaciones.

(Refactoring Guru, 2025)

h. Consecuencias:

- i. SRP: Permite aplicar el principio de responsabilidad única, haciendo que cada clase agrupe funciones similares.
- ii. Open Closed Principle: Es posible agregar implementaciones de la estructura de datos y su iterador e intercambiarlas fácilmente, sin romper código existente.
- iii. Iterar en paralelo: Debido a que cada iterador es independiente, es posible iterar sobre una misma colección en paralelo. Por esta misma razón, también se puede pausar una iteración para continuar más adelante.
- iv. Complejidad: Aplicar el patrón puede añadir complejidad innecesaria a una aplicación si solo se está trabajando con estructuras simples.
- v. Rendimiento: En algunos casos, utilizar un iterador puede ser menos eficiente que acceder a los elementos de colecciones especializadas.

(Refactoring Guru, 2025).

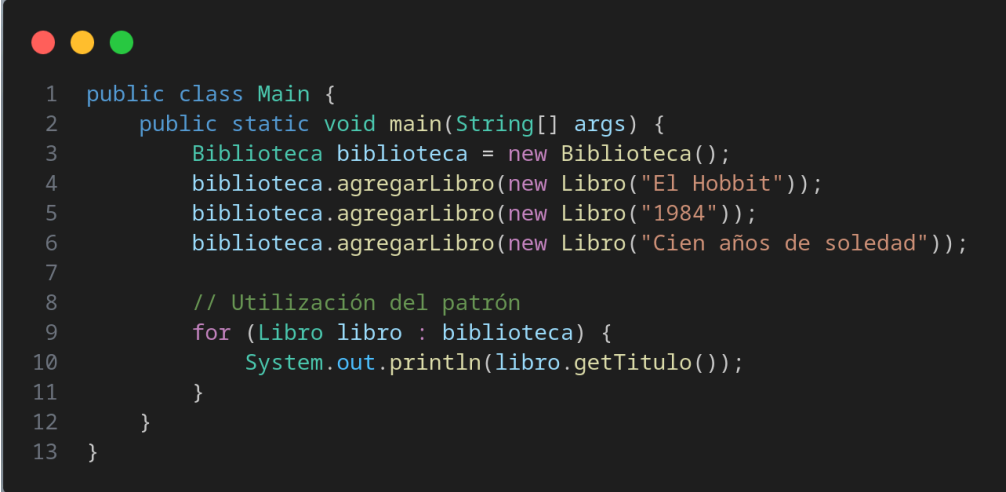
i. Implementación:

- i. Declarar la interfaz del iterador. Como mínimo debe tener un método para acceder al siguiente elemento, pero se pueden agregar más para otros usos.
- ii. Declarar la interfaz de la colección. Esta debe incluir un método para obtener su iterador a través de la interfaz.
- iii. Implementar la colección. Crear la clase con la estructura de datos específica que se va a utilizar con sus respectivos métodos. Para vincularla, esta debe pasarse a sí misma como parámetro en el constructor del iterador.
- iv. Implementar el iterador. Establecer la manera en que se recorre la estructura utilizada.
- v. Utilizarlos en el cliente. Reemplazar los recorridos de la colección con el uso de los iteradores.

j. Código de ejemplo:

```
1 // Colección concreta sobre la cual se itera
2 class Biblioteca implements Iterable<Libro> {
3     private List<Libro> libros = new ArrayList<>();
4
5     public void agregarLibro(Libro libro) {
6         libros.add(libro);
7     }
8
9     // Conexión con su iterador
10    @Override
11    public Iterator<Libro> iterator() {
12        return new LibroIterator(libros);
13    }
14 }
```

```
1 // Iterador personalizado
2 class LibroIterator implements Iterator<Libro> {
3     private List<Libro> libros;
4     private int indice = 0;
5
6     public LibroIterator(List<Libro> libros) {
7         this.libros = libros;
8     }
9
10    @Override
11    public boolean hasNext() {
12        return indice < libros.size();
13    }
14
15    @Override
16    public Libro next() {
17        return libros.get(indice++);
18    }
19 }
```



```

1  public class Main {
2      public static void main(String[] args) {
3          Biblioteca biblioteca = new Biblioteca();
4          biblioteca.agregarLibro(new Libro("El Hobbit"));
5          biblioteca.agregarLibro(new Libro("1984"));
6          biblioteca.agregarLibro(new Libro("Cien años de soledad"));
7
8          // Utilización del patrón
9          for (Libro libro : biblioteca) {
10             System.out.println(libro.getTitulo());
11         }
12     }
13 }

```

k. Usos conocidos:

- i. Recorrido de sistemas de archivos dentro de las aplicaciones.
- ii. Procesamiento del resultado de una query en base de datos.
- iii. Aplicaciones que utilizan grafos.
- iv. Redes sociales para feeds, publicaciones, redes de amigos, etc.

l. Patrones relacionados:

- i. Se puede utilizar junto con Factory para permitir que subclases de la colección puedan retornar diferentes tipos de iteradores compatibles.
- ii. El patrón Memento puede utilizarse para tomar “snapshots” del estado actual de una iteración o regresar a un estado previo si es necesario.
- iii. Para recorrer una estructura compleja y realizar operaciones sobre sus elementos, aunque estos sean de tipos diferentes, se puede utilizar junto con Visitor.

3. Strategy

a. Intención:

La intención de este patrón de diseño es establecer una familia de algoritmos, haciendo que sean intercambiables y encapsulables cada uno de ellos. Esto logra que el algoritmo sea independiente del cliente que lo usa. (Gamma *et al.*, 1994, p. 315).

b. Conocido como:

Este patrón de diseño también es conocido como "*Policy*" o patrón de política. (Gamma *et al.*, 1994, p. 315).

c. Motivo:

Este patrón de diseño surge por la necesidad de tener un código menos rígido, con la característica que sea altamente escalable y que principalmente pueda cambiar el comportamiento de un componente o módulo en tiempo real. (Gamma *et al.*, 1994, p. 315; Refactoring Guru, 2025).

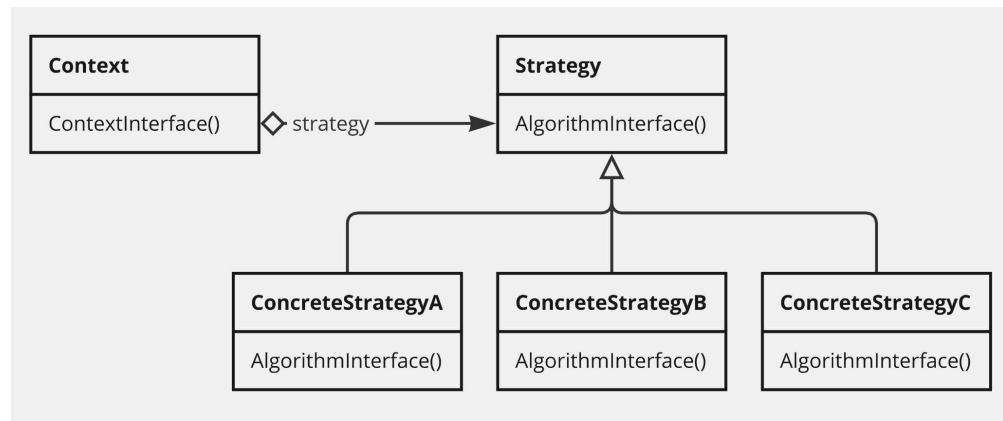
El punto clave de este patrón de diseño es el uso de clases separadas para la implementación de los algoritmos que se desean usar, ya que, no utiliza estructuras condicionales como "if" o "when" para seleccionar el comportamiento del código. Por tal motivo, a todo algoritmo que se encuentra encapsulado en una clase es llamado como una "strategy". (Gamma *et al.*, 1994, p. 315).

d. Aplicaciones:

- i. Cuando se quiere utilizar diferentes variantes de un algoritmo sin la necesidad de recurrir a estructuras condicionales y se desee cambiar su comportamiento en tiempo de ejecución.
- ii. Cuando se tienen muchas clases similares entre sí, que solo se diferencian en la forma que ejecutan algún comportamiento.
- iii. Cuando un algoritmo usa información o "*data*" que el cliente no debería saber.
- iv. Cuando en el código se tenga una cantidad masiva de estructuras condicionales, ya que la encapsulación de algoritmos en clases reduce su uso.

(Refactoring Guru, 2025).

e. Estructura:



(Gamma *et al.*, 1994, p. 316).

f. Participantes:

- i. Strategy: También conocido como compositor, se encarga de declarar una interfaz común para todos los algoritmos que se vayan a incluir. Context utiliza esta interfaz para llamar a un algoritmo definido en una ConcreteStrategy.
- ii. ConcreteStrategy: Se le conoce también como compositor simple, ya que se encarga de implementar el algoritmo usando la interfaz de Strategy.
- iii. Context: También conocido como la composición, es quien hace el llamado de las diferentes implementaciones de un algoritmo utilizando la interfaz Strategy, además de encargarse del cambio de algoritmo en tiempo de ejecución.

(Gamma *et al.*, 1994, p. 317).

g. Colaboraciones:

- i. Strategy y Context interactúan para implementar el algoritmo escogido. Un context puede pasar toda la “data” necesaria para el algoritmo cuando este es llamado y a su vez puede mandarse a sí mismo como un argumento para Strategy cuando este lo requiera.
- ii. El cliente solo interactúa con el Context, para el cual existe una familia de ConcretStrategy que implementan diferentes versiones del algoritmo definido en Strategy.

(Gamma *et al.*, 1994, p. 317).

h. Consecuencias:

- i. Familias de algoritmos relacionados: El uso de interfaces “Strategy” para algoritmos hace que todas las clases que las implementen tengan algoritmos en común, que únicamente se diferencian en cómo se comportan, lo que da como resultado un conjunto de clases que implementan los mismos algoritmos convirtiéndolos en familias.
- ii. Alternativa para la herencia: El uso de implementaciones e interfaces brinda una alternativa al uso de herencia, que además ofrece beneficios para la modularidad del código. Es posible heredar una clase Context directamente para manejar diferentes comportamientos de un algoritmo, pero a la larga esto hace la relación entre Context y Strategy más complicada y difícil de entender.
- iii. Una selección de implementaciones: Las Strategy pueden ofrecer diferentes implementaciones del mismo comportamiento, esto hace que el cliente tenga que elegir entre estrategias con diferentes compensaciones de tiempo y espacio.
- iv. Clientes deben conocer las diferentes strategy: Debido al potencial de este patrón de diseño, el cliente debe estar al tanto de cuáles son las Strategy de las cuales dispone para saber elegir la que más se adapta a su necesidad.
- v. Sobrecarga en la comunicación entre Context y Strategy: Debido a que Context tiene comunicación con todos los ConcreteStrategy que se han implementado para todas las variaciones del algoritmo en ocasiones puede solicitar data o parámetros que no son importantes para cierta implementación del algoritmo, entonces dichos parámetros nunca son usados y pueden ocasionar que Context se sobrecargue.
- vi. Aumento de la cantidad de objetos: El uso de Strategy como interfaces y ConcreteStrategy como clases que implementan dichas interfaces hace que se incremente el número de objetos a nivel general del código.

(Gamma *et al.*, 1994, p. 317-318).

i. Implementación:

- i. Identificar un algoritmo que se preste a cambios constantemente.
- ii. Declarar la interfaz Strategy en donde se hará la declaración del algoritmo anteriormente identificado.
- iii. Implementa el algoritmo creado en las clases ConcreteStrategy en donde para cada variación del algoritmo será utilizada una ConcreteStrategy diferente.
- iv. En la clase Context agregar un objeto de tipo Strategy para aplicar el polimorfismo.

- v. Los clientes asocian el Context como una única Strategy que les da el comportamiento del algoritmo dependiendo de lo que necesiten.

(Refactoring Guru, 2025).

j. Código de ejemplo:

i. Strategy:

```
1 public interface DiscountStrategy {
2     double applyDiscount(double price); //Algoritmo con comportamiento variable
3 }
```

ii. ConcreteStrategy:

```
1 // ConcreteStrategyA: Sin descuento (clientes normales)
2 public class ConcreteStrategyNoDiscount implements DiscountStrategy {
3     @Override
4     public double applyDiscount(double price) {
5         return price;
6     }
7 }
```

```
1 // ConcreteStrategyB: Descuento del 30% para clientes VIP
2 public class ConcreteStrategyVipDiscount implements DiscountStrategy{
3     @Override
4     public double applyDiscount(double price) {
5         return price * 0.7;
6     }
7 }
```

```
1 // ConcreteStrategyC: Descuento del 20% por temporada
2 public class ConcreteStrategySeasonalDiscount implements DiscountStrategy {
3     @Override
4     public double applyDiscount(double price) {
5         return price * 0.8;
6     }
7 }
```

iii. Context:

```
1 public class ContextShoppingCart {
2     private DiscountStrategy discountStrategy;
3
4     public ContextShoppingCart(DiscountStrategy discountStrategy) {
5         this.discountStrategy = discountStrategy;
6     }
7
8     public void setDiscountStrategy(DiscountStrategy discountStrategy) {
9         this.discountStrategy = discountStrategy;
10    }
11
12    public double calculateTotal(double price) {
13        return discountStrategy.applyDiscount(price);
14    }
15 }
16
```

iv. Main:

```
1 public class Main {
2     public static void main(String[] args) {
3         ContextShoppingCart cart = new ContextShoppingCart(new ConcreteStrategyNoDiscount()); // Inicialmente sin descuento
4
5         double price = 100.0;
6
7         System.out.println("Precio sin descuento: " + cart.calculateTotal(price));
8
9         cart.setDiscountStrategy(new ConcreteStrategyVipDiscount()); // Se aplica descuento VIP
10        System.out.println("Precio con descuento VIP: " + cart.calculateTotal(price));
11
12        cart.setDiscountStrategy(new ConcreteStrategySeasonalDiscount()); // Se aplica descuento por temporada
13        System.out.println("Precio con descuento por temporada: " + cart.calculateTotal(price));
14    }
15 }
```

v. Resultado:

```
Precio sin descuento: 100.0
Precio con descuento VIP: 70.0
Precio con descuento por temporada: 80.0
```

k. Usos conocidos:

- i. Es utilizado en algoritmos de ordenación.
- ii. En selección de opciones como métodos de pago.
- iii. Validación de formularios.

(Gamma *et al.*, 1994, p. 322-323).

I. Patrones relacionados:

- i. Template: Mientras que Strategy permite cambiar dinámicamente entre diferentes algoritmos, Template utiliza herencia para permitir la personalización de partes específicas del algoritmo dentro de una estructura fija.
- ii. State: Ambos patrones encapsulan comportamientos en clases separadas, pero State se centra en cambiar el comportamiento de un objeto en función de su estado interno, mientras que Strategy permite seleccionar entre diferentes comportamientos de manera externa y dinámica.
- iii. Command: Mientras que Strategy encapsula algoritmos que pueden intercambiarse para variar el comportamiento, Command encapsula acciones, permitiendo operaciones como comandos.
- iv. Decorator: Ambos utilizan la composición para extender o modificar comportamientos. Decorator añade responsabilidades adicionales a un objeto, mientras que Strategy cambia el comportamiento completo al seleccionar diferentes algoritmos.

(Refactoring Guru, 2025).

Conclusiones

- Los patrones de diseño permiten escribir código más modular, flexible y fácil de modificar sin afectar otras partes del sistema.
- Algunos patrones ayudan a mejorar el rendimiento del software, ya sea optimizando el acceso a recursos, estructurando mejor la lógica o reduciendo el costo computacional.
- El patrón de diseño Strategy resulta sumamente útil cuando se quiere cambiar el comportamiento del código en tiempo de ejecución, además fomenta la flexibilidad y el desacoplamiento del código.
- El patrón Iterator facilita el recorrido de colecciones sin acoplar la lógica de iteración, mejorando la flexibilidad y mantenibilidad del software.
- El patrón de diseño Proxy ofrece maneras de optimizar, organizar y asegurar el software de manera escalable y mantenible a lo largo del tiempo.

Bibliografía

Blancarte, O. (2024). Iterator. Oscar Blancarte. Software Architect.

<https://reactiveprogramming.io/blog/es/patrones-de-diseno/iterator>

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley

GeeksforGeeks. (2025). Iterator Design Pattern.

<https://www.geeksforgeeks.org/iterator-pattern/>

GeeksforGeeks. (2017, July 8). Proxy Design Pattern. GeeksforGeeks.

<https://www.geeksforgeeks.org/proxy-design-pattern/#how-to-implement-proxy-design-pattern>

Patrones estructurales. (2025). Refactoring.guru.

<https://refactoring.guru/es/design-patterns/structural-patterns>

Reactive Programming. (2021). Proxy - Patrón de diseño. Reactive Programming.

<https://reactiveprogramming.io/blog/es/patrones-de-diseno/proxy>

Refactoring Guru. (2025). Proxy - Patrón de diseño con ejemplo en Java. Refactoring.Guru.

<https://refactoring.guru/es/design-patterns/proxy/java/example>

Refactoring Guru. (2025). *Strategy*. Design Patterns.

<https://refactoring.guru/design-patterns/strategy>

Spring Sale. (2025). Iterator. <https://refactoring.guru/es/design-patterns/iterator>

Anexos

Repositorio de GitHub:

https://github.com/Isabella334/Proyecto_DAPA.git

Enlace al Google Docs:

https://docs.google.com/document/d/19Bs0GHcsgl36yhoOjqKACp0KdoK_dfiBGOkelShtEMY/edit?tab=t.0

Gestión del tiempo:

Nro.	Tarea	Encargado	Fecha inicio	Fecha entrega
1	Redacción de la intención del patrón Iterator	Juan Solís	04/05/2025	04/05/2025
2	Redacción del conocimiento del patrón Iterator	Juan Solís	04/05/2025	04/05/2025
3	Redacción del motivo del patrón Iterator	Juan Solís	04/05/2025	04/05/2025
4	Redacción de las aplicaciones del patrón Iterator	Juan Solís	04/05/2025	04/05/2025
5	Redacción de la estructura del patrón Iterator	Juan Solís	04/05/2025	04/05/2025
6	Redacción de los participantes del patrón Iterator	Juan Solís	04/05/2025	04/05/2025
7	Redacción de las colaboraciones del patrón Iterator	Victor Pérez	04/05/2025	04/05/2025
8	Redacción de las consecuencias del patrón Iterator	Victor Pérez	04/05/2025	04/05/2025
9	Redacción de la implementación del patrón Iterator	Victor Pérez	04/05/2025	04/05/2025
10	Elaboración del código de ejemplo del patrón Iterator	Victor Pérez	04/05/2025	04/05/2025

11	Redacción de los usos conocidos del patrón Iterator	Victor Pérez	04/05/2025	04/05/2025
12	Redacción de los patrones relacionados con el patrón Iterator	Victor Pérez	04/05/2025	04/05/2025
13	Redacción de la intención del patrón Proxy	Nils Muralles	04/05/2025	04/05/2025
14	Redacción del conocimiento del patrón Proxy	Nils Muralles	04/05/2025	04/05/2025
15	Redacción del motivo del patrón Proxy	Nils Muralles	04/05/2025	04/05/2025
16	Redacción de las aplicaciones del patrón Proxy	Nils Muralles	04/05/2025	04/05/2025
17	Redacción de la estructura del patrón Proxy	Nils Muralles	04/05/2025	04/05/2025
18	Redacción de los participantes del patrón Proxy	Nils Muralles	04/05/2025	04/05/2025
19	Redacción de las colaboraciones del patrón Proxy	Isabella Recinos	04/05/2025	04/05/2025
20	Redacción de las consecuencias del patrón Proxy	Isabella Recinos	04/05/2025	04/05/2025
21	Redacción de la implementación del patrón Proxy	Isabella Recinos	04/05/2025	04/05/2025
22	Elaboración del código de ejemplo del patrón Proxy	Isabella Recinos	04/05/2025	04/05/2025
23	Redacción de los usos conocidos del patrón Proxy	Isabella Recinos	04/05/2025	04/05/2025
24	Redacción de los patrones relacionados con el patrón Proxy	Isabella Recinos	04/05/2025	04/05/2025
25	Redacción de la intención del patrón Strategy	Diego Flores	04/05/2025	04/05/2025
26	Redacción del	Diego Flores	04/05/2025	04/05/2025

	conocimiento del patrón Strategy			
27	Redacción del motivo del patrón Strategy	Diego Flores	04/05/2025	04/05/2025
28	Redacción de las aplicaciones del patrón Strategy	Diego Flores	04/05/2025	04/05/2025
29	Redacción de la estructura del patrón Strategy	Diego Flores	04/05/2025	04/05/2025
30	Redacción de los participantes del patrón Strategy	Diego Flores	04/05/2025	04/05/2025
31	Redacción de las colaboraciones del patrón Strategy	Diego Flores	04/05/2025	04/05/2025
32	Redacción de las consecuencias del patrón Strategy	Diego Flores	04/05/2025	04/05/2025
33	Redacción de la implementación del patrón Strategy	Diego Flores	04/05/2025	04/05/2025
34	Elaboración del código de ejemplo del patrón Strategy	Diego Flores	04/05/2025	04/05/2025
35	Redacción de los usos conocidos del patrón Strategy	Diego Flores	04/05/2025	04/05/2025
36	Redacción de los patrones relacionados con el patrón Strategy	Diego Flores	04/05/2025	04/05/2025
37	Redacción de las conclusiones	Todos	05/03/2025	05/03/2025
38	Redacción de la introducción	Nils	05/03/2025	05/03/2025
39	Redacción del resumen	Nils	05/03/2025	05/03/2025
40	Elaboración de la presentación	Todos	05/03/2025	05/03/2025