SPRINT 1

Nombre: Cristian Camilo Paez Rodriguez

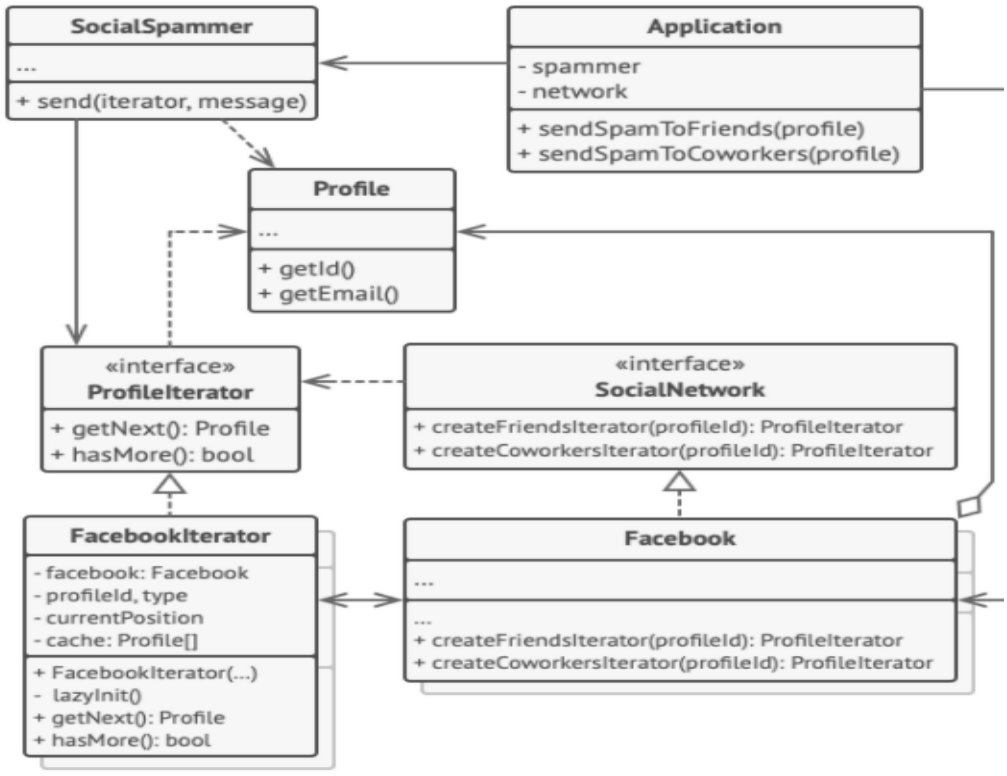
PATRONES DE DISEÑO

PATRON DE COMPORTAMIENTO: ITERATOR

Iterator es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).

Ejemplo 1:

En este ejemplo, el patrón **Iterator** se utiliza para recorrer un tipo especial de colección que encapsula el acceso al grafo social de Facebook. La colección proporciona varios iteradores que recorren perfiles de distintas formas.



Ejemplo de iteración de perfiles sociales.

El iterador ‘amigos’ puede utilizarse para recorrer los amigos de un perfil dado. El iterador ‘colegas’ hace lo mismo, excepto que omite amigos que no trabajen en la misma empresa que la persona objetivo. Ambos iteradores implementan una interfaz común que permite a los clientes extraer perfiles sin profundizar en los detalles de la implementación, como la autenticación y el envío de solicitudes REST.

El código cliente no está acoplado a clases concretas porque sólo trabaja con colecciones e iteradores a través de interfaces. Si decides conectar tu aplicación a una nueva red social, sólo necesitas proporcionar nuevas clases de colección e iteradoras, sin cambiar el código existente.

```
// La interfaz de colección debe declarar un método fábrica para
// producir iteradores. Puedes declarar varios métodos si hay
// distintos tipos de iteración disponibles en tu programa.
interface SocialNetwork is
    method createFriendsIterator(profileId):ProfileIterator
    method createCoworkersIterator(profileId):ProfileIterator


// Cada colección concreta está acoplada a un grupo de clases
// iteradoras concretas que devuelve, pero el cliente no lo
// está, ya que la firma de estos métodos devuelve interfaces
// iteradoras.
class Facebook implements SocialNetwork is
    // ... El grueso del código de la colección debe ir aquí ...
    // Código de creación del iterador.
    method createFriendsIterator(profileId) is
        return new FacebookIterator(this, profileId, "friends")
    method createCoworkersIterator(profileId) is
        return new FacebookIterator(this, profileId, "coworkers")


// La interfaz común a todos los iteradores.
interface ProfileIterator is
    method getNext():Profile
    method hasMore():bool


// La clase iteradora concreta.
class FacebookIterator implements ProfileIterator is
    // El iterador necesita una referencia a la colección que
    // recorre.
    private field facebook: Facebook
    private field profileId, type: string
```

```
    // Un objeto iterador recorre la colección
    // independientemente de otro iterador, por eso debe
    // almacenar el estado de iteración.
    private field currentPosition
    private field cache: array of Profile

    constructor FacebookIterator(facebook, profileId, type) is
        this.facebook = facebook
        this.profileId = profileId
        this.type = type

    private method lazyInit() is
        if (cache == null)
            cache = facebook.socialGraphRequest(profileId, type)

    // Cada clase iteradora concreta tiene su propia
    // implementación de la interfaz iteradora común.
    method getNext() is
        if (hasMore())
            currentPosition++
            return cache[currentPosition]

    method hasMore() is
        lazyInit()
        return currentPosition < cache.length


// Aquí tienes otro truco útil: puedes pasar un iterador a una
// clase cliente en lugar de darle acceso a una colección
// completa. De esta forma, no expones la colección al cliente.
//
// Y hay otra ventaja: puedes cambiar la forma en la que el
// cliente trabaja con la colección durante el tiempo de
// ejecución pasándole un iterador diferente. Esto es posible
// porque el código cliente no está acoplado a clases iteradoras
// concretas.
class SocialSpammer is
    method send(iterator: ProfileIterator, message: string) is
        while (iterator.hasMore())
            profile = iterator.getNext()
            System.sendEmail(profile.getEmail(), message)


// La clase Aplicación configura colecciones e iteradores y
// después los pasa al código cliente.
class Application is
    field network: SocialNetwork
    field spammer: SocialSpammer

    method config() is
        if working with Facebook
            this.network = new Facebook()
        if working with LinkedIn
            this.network = new LinkedIn()
```

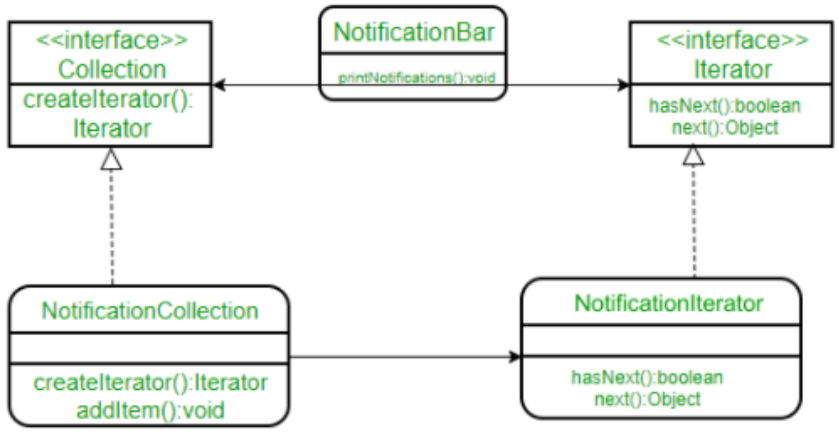
```
        this.spammer = new SocialSpammer()

    method sendSpamToFriends(profile) is
        iterator = network.createFriendsIterator(profile.getId())
        spammer.send(iterator, "Very important message")

    method sendSpamToCoworkers(profile) is
        iterator = network.createCoworkersIterator(profile.getId())
        spammer.send(iterator, "Very important message")
```

Ejemplo 2:

Supongamos que estamos creando una barra de notificaciones en nuestra aplicación que muestra todas las notificaciones que se encuentran en una colección de notificaciones. NotificationCollection proporciona un iterador para iterar sobre sus elementos sin exponer cómo ha implementado la colección (matriz en este caso) al Cliente (NotificationBar).



```
// Un programa Java para demostrar la implementación
// de patrón de iterador con el ejemplo de
// notificaciones

// Una clase de notificación simple
class Notification
{
	// Para almacenar el mensaje de notificación
	String notification;

	public Notification(String notification)
	{
		this.notification = notification;
	}
	public String getNotification()
	{
		return notification;
	}
}

// Interfaz de colección
interface Collection
{
```

```
        public Iterator createIterator();
}

// Colección de notificaciones
class NotificationCollection implements Collection
{
        static final int MAX_ITEMS = 6;
        int numberOfItems = 0;
        Notification[] notificationList;

        public NotificationCollection()
        {
                notificationList = new Notification[MAX_ITEMS];

                // Agreguemos algunas notificaciones ficticias
                addItem("Notification 1");
                addItem("Notification 2");
                addItem("Notification 3");
        }

        public void addItem(String str)
        {
                Notification notification = new Notification(str);
```

```
            if (numberOfItems >= MAX_ITEMS)
                System.err.println("Full");
            else
            {
                notificationList[numberOfItems] = notification;
                numberOfItems = numberOfItems + 1;
            }
        }

        public Iterator createIterator()
        {
            return new NotificationIterator(notificationList);
        }
}

// También podríamos usar Java.Util.Iterator
interface Iterator
{
        // indica si hay más elementos para
        // iterar sobre
        boolean hasNext();

        // devuelve el siguiente elemento
```

```
        Object next();
}

// Iterador de notificación
class NotificationIterator implements Iterator
{
        Notification[] notificationList;

        // mantiene la posición actual del iterador sobre el array
        int pos = 0;

        // El constructor toma la matriz de notifiactionList
        // va a iterar.
        public NotificationIterator (Notification[] notificationList)
        {
                this.notificationList = notificationList;
        }

        public Object next()
        {
// devuelve el siguiente elemento en la matriz e incrementa
        pos
        Notification notification = notificationList[pos];
```

```
            pos += 1;
            return notification;
        }

        public boolean hasNext()
        {
            if (pos >= notificationList.length ||
                notificationList[pos] == null)
                return false;
            else
                return true;
        }
}

// Contiene una colección de notificaciones como objeto de
// NotificationCollection
class NotificationBar
{
        NotificationCollection notifications;

        public NotificationBar(NotificationCollection notifications)
        {
            this.notifications = notifications;
```

```
	}

	public void printNotifications()
	{
		Iterator iterator = notifications.createIterator();
		System.out.println("-------NOTIFICATION BAR--------");
		while (iterator.hasNext())
		{
			Notification n = (Notification)iterator.next();
			System.out.println(n.getNotification());
		}
	}
}

// Clase de controlador
class Main
{
	public static void main(String args[])
	{
		NotificationCollection nc = new NotificationCollection();
		NotificationBar nb = new NotificationBar(nc);
		nb.printNotifications();
```

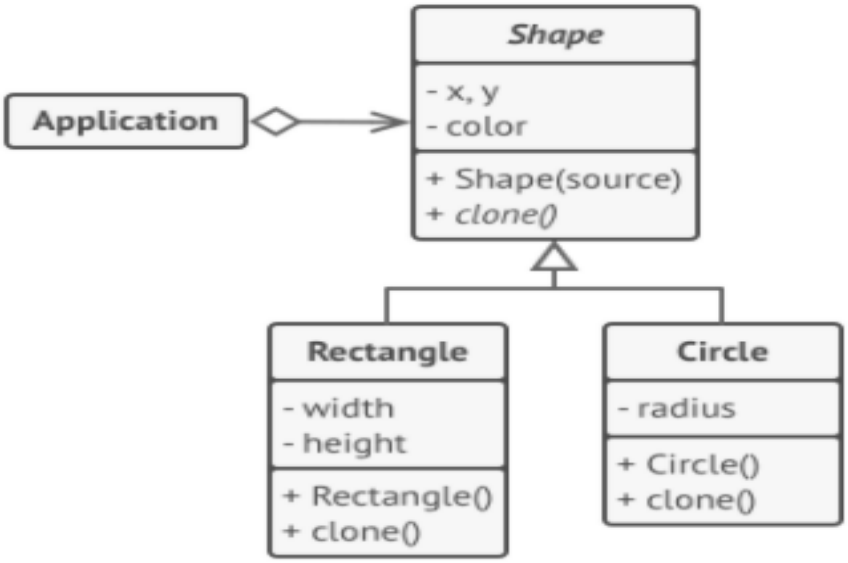
```
    }
}
```

PATRON CREACIONAL: PROTOTYPE

Prototype es un patrón de diseño creacional que nos permite copiar objetos existentes sin que el código dependa de sus clases.

Ejemplo 1:

En este ejemplo, el patrón **Prototype** nos permite producir copias exactas de objetos geométricos sin acoplar el código a sus clases.



Clonación de un grupo de objetos que pertenece a una jerarquía de clase.

Todas las clases de forma siguen la misma interfaz, que proporciona un método de clonación. Una subclase puede invocar el método de clonación padre antes de copiar sus propios valores de campo al objeto resultante.

```
// Prototipo base.
abstract class Shape is
    field X: int
    field Y: int
    field color: string

    // Un constructor normal.
    constructor Shape() is
        // ...

    // El constructor prototipo. Un nuevo objeto se inicializa
    // con valores del objeto existente.
    constructor Shape(source: Shape) is
        this()
        this.X = source.X
        this.Y = source.Y
        this.color = source.color

    // La operación clonar devuelve una de las subclases de
    // Shape (Forma).
    abstract method clone():Shape


// Prototipo concreto. El método de clonación crea un nuevo
// objeto y lo pasa al constructor. Hasta que el constructor
// termina, tiene una referencia a un nuevo clon. De este modo
// nadie tiene acceso a un clon a medio terminar. Esto garantiza
// la consistencia del resultado de la clonación.
class Rectangle extends Shape is
    field width: int
    field height: int

    constructor Rectangle(source: Rectangle) is
        // Para copiar campos privados definidos en la clase
        // padre es necesaria una llamada a un constructor
        // padre.
        super(source)
        this.width = source.width
        this.height = source.height

    method clone():Shape is
        return new Rectangle(this)


class Circle extends Shape is
    field radius: int

    constructor Circle(source: Circle) is
        super(source)
        this.radius = source.radius

    method clone():Shape is
        return new Circle(this)
```

```
// En alguna parte del código cliente.
class Application is
    field shapes: array of Shape

    constructor Application() is
        Circle circle = new Circle()
        circle.X = 10
        circle.Y = 10
        circle.radius = 20
        shapes.add(circle)

        Circle anotherCircle = circle.clone()
        shapes.add(anotherCircle)
        // La variable `anotherCircle` (otroCírculo) contiene
        // una copia exacta del objeto `circle`.

        Rectangle rectangle = new Rectangle()
        rectangle.width = 10
        rectangle.height = 20
        shapes.add(rectangle)

    method businessLogic() is
        // Prototype es genial porque te permite producir una
        // copia de un objeto sin conocer nada de su tipo.
        Array shapesCopy = new Array of Shapes.

        // Por ejemplo, no conocemos los elementos exactos de la
        // matriz de formas. Lo único que sabemos es que son
        // todas formas. Pero, gracias al polimorfismo, cuando
        // invocamos el método `clonar` en una forma, el
        // programa comprueba su clase real y ejecuta el método
        // de clonación adecuado definido en dicha clase. Por
        // eso obtenemos los clones adecuados en lugar de un
        // grupo de simples objetos Shape.
        foreach (s in shapes) do
            shapesCopy.add(s.clone())

        // La matriz `shapesCopy` contiene copias exactas del
        // hijo de la matriz `shape`.
```

Ejemplo 2:

Tenemos una fábrica de camisetas con estampados, típicas de las ferias y mercadillos. Para crear nuevas camisetas, cogeremos una similar y modificaremos únicamente el color, la talla y el estampado. Empezamos con el prototipo:

```
public abstract class Camiseta {
        private String nombre;
        private Integer talla;
        private String color;
```

```
        private String manga;
        private String estampado;
        private Object material;

        public Camiseta (String nombre,Integer talla, String color, String
manga, String estampado, Object material){
                this.nombre = nombre;
                this.talla = talla;
                this.color = color;
                this.manga = manga;
                this.estampado = estampado;
                this.material = material;
        }
        public abstract Camiseta clone();
        /*
         * Todos los getter y los setter.
        */
}
```

Ahora construiremos los prototipos concretos para camisetas de manga larga y manga corta:

```
public class CamisetaMCorta extends Camiseta{
        public CamisetaMCorta(Integer talla, String color, String estampado){
                this.nombre = “Prototipo”;
                this.talla = talla;
                this.color = color;
                this.manga = “Corta”;
                this.estampado = estampado;
                this.material = new Lana();
        }
        public Camiseta clone(){
                return new CamisetaMCorta(this.talla, this.color,
this.estampado);
        }
}


public class CamisetaMLarga extends Camiseta{
        public CamisetaMLarga(Integer talla, String color, String estampado){
                this.nombre = “Prototipo”;
                this.talla = talla;
                this.color = color;
                this.manga = “Larga”;
                this.estampado = estampado;
                this.material = new Lana();
        }
        public Camiseta clone(){
                return new CamisetaMLarga(this.talla, this.color,
this.estampado);
```

```
        }
}
```

Por último, el método main hará de cliente y creará distintas camisetas tanto de manga larga como de manga corta a partir de prototipos.

```
public static void main(String[] args){
// Recibiremos en los argumentos los estampados de las camisetas

// Creamos los prototipos
Camiseta prototipoMCorta = new CamisetaMCorta(40, “blanco”, “Logotipo”);
Camiseta prototipoMLarga = new prototipoMLarga(40, “blanco”, “Logotipo”);

// Almacenamos las camisetas disponibles
ArrayList camisetas = new ArrayList();

for(int i = 0; i<args.length;i++){
                Camiseta cc =  prototipoMCorta.clone();
                cc.setEstampado(args[i]);

                for(int j = 35; j<60; j++){
                        Camiseta cc_talla = cc.clone();
                        cc_talla.setTalla(j);
                        camisetas.add(cc_talla);
                }


                Camiseta cl =  prototipoMLarga.clone();
                cl.setEstampado(args[i]);

                for(int j = 35; j<60; j++){
                        Camiseta cl_talla = cl.clone();
                        cl_talla.setTalla(j);
                        camisetas.add(cl_talla);
                }
        }
}
```