El concepto de Command Pattern o patrón comando es uno de los más conocidos en el mundo de la programación. ¿Para qué sirve el patrón comando y que situaciones resuelve?. En programación nos podemos encontrar en muchas situaciones en las que tenemos que gestionar tareas que reciben algún tipo de objeto como parámetro.



Una vez recibido este objeto deberemos procesarle. En principio es una tarea que parece muy sencilla y nos bastaría con tener una clase que implemente las diferentes tareas para el objeto .Vamos a ver un ejemplo concreto que nos ayude a clarificar. Supongamos que disponemos de una clase Producto.

```
package com.arquitecturajava;

public class Producto {

	private int id;
	private String nombre;
	private double precio;
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getNombre() {
		return nombre;
	}
```

```
        public void setNombre(String nombre) {
                this.nombre = nombre;
        }
        public double getPrecio() {
                return precio;
        }
        public void setPrecio(double precio) {
                this.precio = precio;
        }
}
```

Tareas a realizar con el producto

Deberemos realizar varias tareas :

- ValidarProducto
- EnviarPorCorreo
- Imprimir

Estamos ante una situación muy sencilla, nos es suficiente con crear una clase que disponga de tres métodos que que encarguen cada uno de una operación:

```
package com.arquitecturajava;

public class GestorProductos {

    public void validarProducto(Producto producto) {

        if (producto.getPrecio() < 100) {
            System.out.println("Producto válido");
        } else {
            System.out.println("Producto inválido");
        }
```

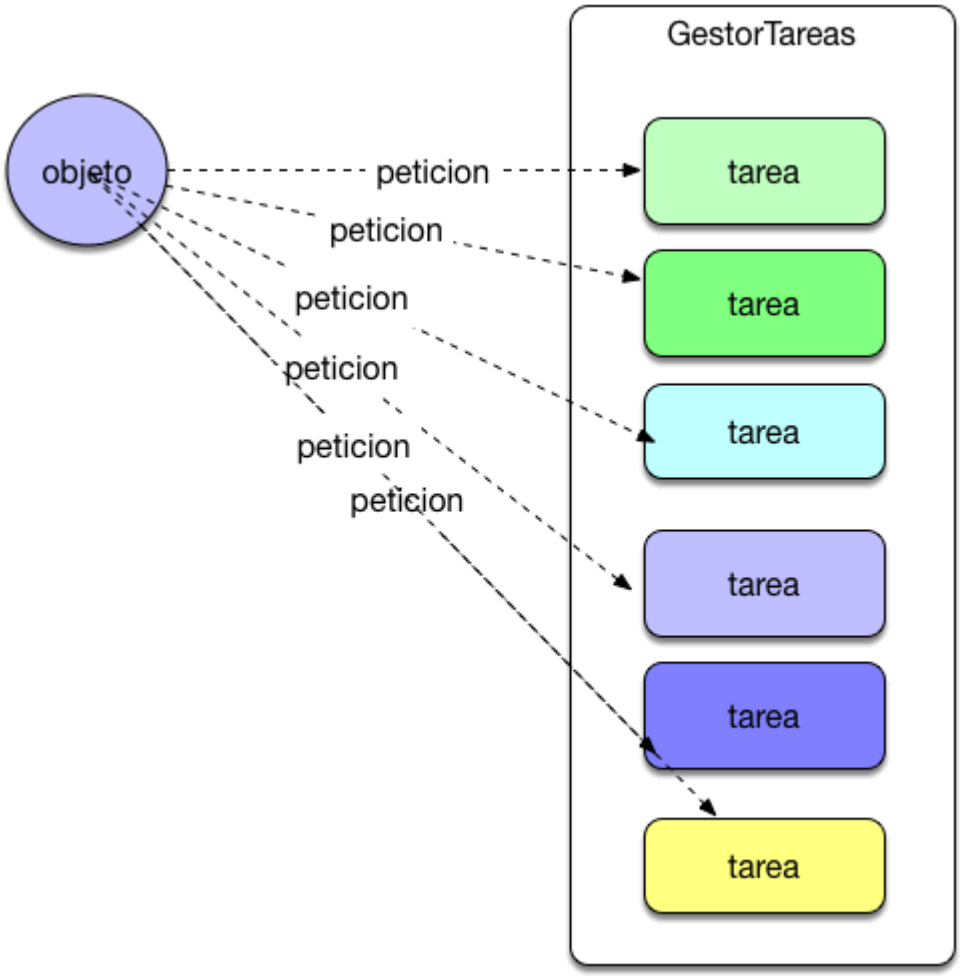
```
    }

    public void imprimirProducto(Producto producto) {
        System.out.println(producto.getNombre());
        System.out.println(producto.getId());
        System.out.println(producto.getPrecio());
    }

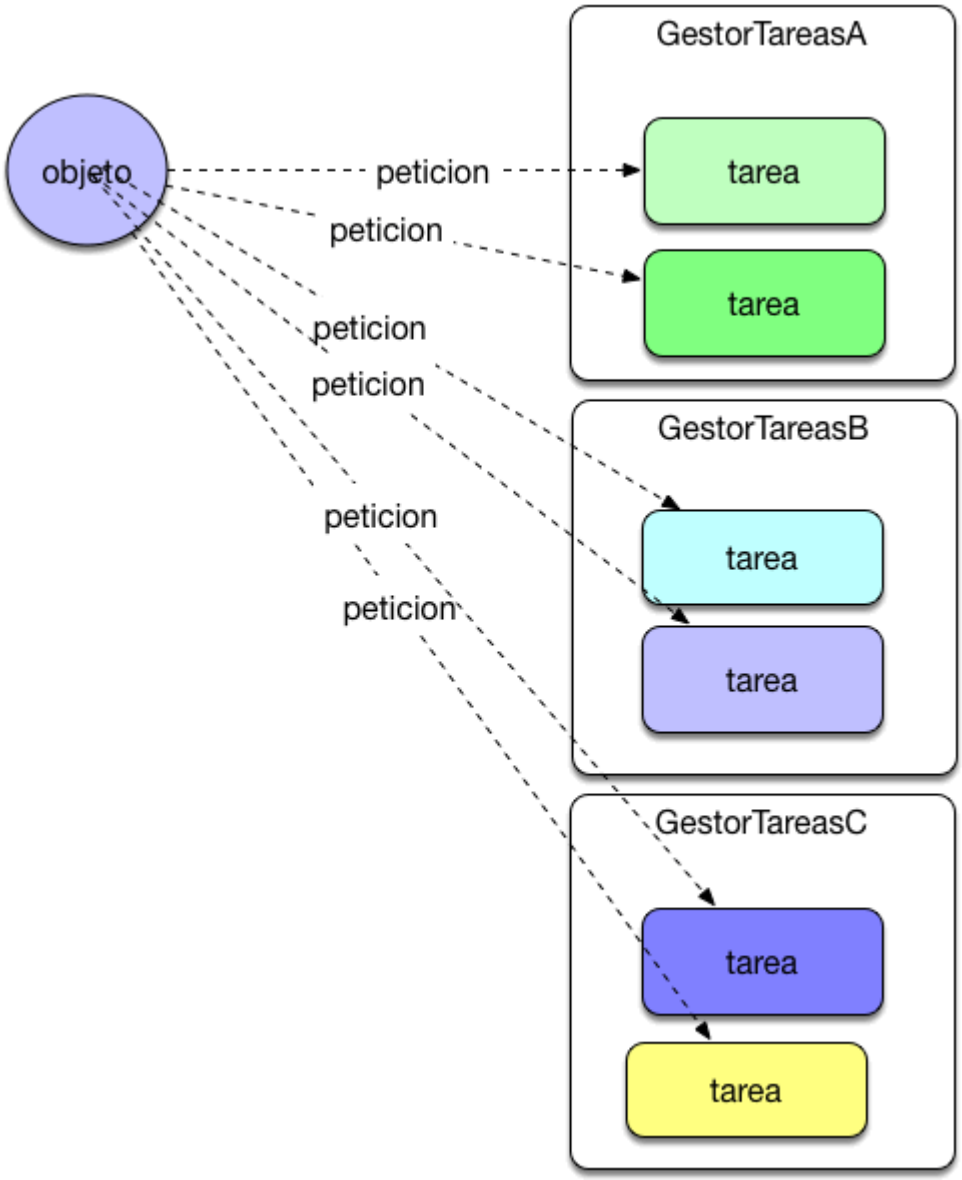
    public void enviarPorCorreo(Producto producto) {
        System.out.println(producto.getNombre() + " enviado por
correo");
    }
}
```

Sin Java Command Pattern

Los problemas surgen cuando no hay solo tres tareas que ejecutar , sino que el número de tareas crece de forma exponencial .



Esto en principio nos puede resultar raro , pero no hay nada mas que mirar nuestro programa y darnos cuentas que podemos ejecutar muchas tareas y muy diferentes sobre el concepto de producto . Una solución sencilla pasaría por construir más clases que almacenen los diferentes métodos.

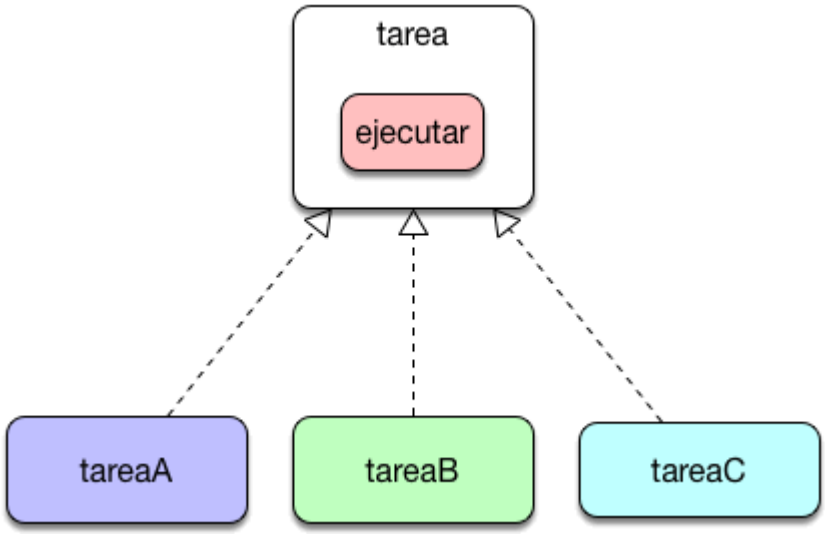


Sin embargo no siempre es la mejor solución ya que genera un fuerte acoplamiento entre el cliente y los diferentes componentes, el cliente necesita conocer muchas clases Por otro lado no siempre es sencillo decidir que tareas van en cada clase ya que con el paso del tiempo las tareas y la relación entre ellas varia en un negocio.

Java Command Pattern una solución elegante

Para solventar este problema vamos a construir un ejemplo usando Java Command Pattern .

Este patrón se encarga de definir el concepto abstracto de Tarea y de construir varias clases que lo implementen




```
package com.arquitecturajava;

public interface TareaProducto {

    // Método abstracto para ejecutar una tarea con un producto
    void ejecutar(Producto producto);
}
```

```
package com.arquitecturajava;

public class TareaEnvioCorreo implements TareaProducto {

    @Override
    public void ejecutar(Producto producto) {
        System.out.println(producto.getNombre() + " enviado por
correo");
    }
```

```
}
```

```
package com.arquitecturajava;

public class TareaImprimirProducto implements TareaProducto {

    @Override
    public void ejecutar(Producto producto) {
        System.out.println(producto.getNombre());
        System.out.println(producto.getId());
        System.out.println(producto.getPrecio());
    }

}
```

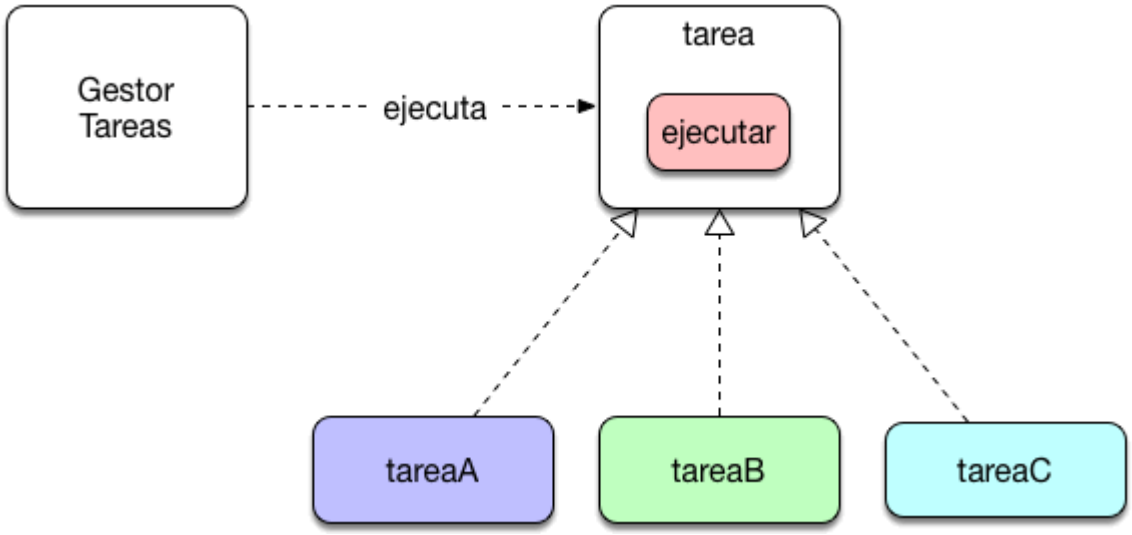
```
package com.arquitecturajava;

public class ValidarProducto implements TareaProducto {

    @Override
    public void ejecutar(Producto producto) {
        if (producto.getPrecio() < 100) {
            System.out.println("Producto válido");
        } else {
            System.out.println("Producto inválido");
        }
    }

}
```

Una vez hemos construido la jerarquía de clases nos queda definir el GestorTareas que se encarga de ejecutar cada una de las tareas.




```
package com.arquitecturajava;

public class GestorTareas {

    public void ejecutar(TareaProducto tarea, Producto p) {
        tarea.ejecutar(p);
    }
}
```

Este diseño nos permite tener una mayor flexibilidad ya que cada tarea es independiente. El añadir nuevas tareas no afecta al resto de tareas. Por otro lado es muy sencillo generar nuevas clases que por ejemplo agrupen tareas. Como es el claso de SuperTarea que es una clase que aglutina una lista de tareas que deseamos ejecutar.

```
package com.arquitecturajava;

import java.util.ArrayList;
import java.util.List;

public class SuperTarea implements TareaProducto {
```

```
    private List<TareaProducto> lista = new
ArrayList<TareaProducto>();

    public void addTarea(TareaProducto tarea) {
        lista.add(tarea);
    }

    @Override
    public void ejecutar(Producto producto) {
        lista.forEach(t -> t.ejecutar(producto));
    }
}
```

Hemos usado programación funcional de Java 8 para recorrer el array de tareas y ejecutarlas todas. Es momento de generar una SuperTarea que contenga varias tareas (comandos) y que el gestor de tareas las ejecute:

```
package com.arquitecturajava;

public class Principal {

    public static void main(String[] args) {

        SuperTarea st = new SuperTarea();

        st.addTarea(new ValidarProducto());
        st.addTarea(new TareaEnvioCorreo());

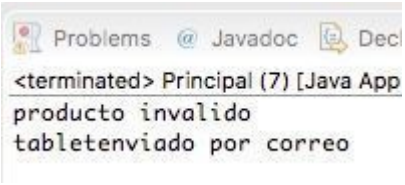
        GestorTareas gt = new GestorTareas();
        Producto p = new Producto(1, "tablet", 100);

        gt.ejecutar(st, p);
```

```
    }

}
```

Ejecutamos el programa y veremos el resultado en la consola en donde se ejecutan todas las tareas que hemos solicitado .



El uso de Command Pattern o patrón comando es una de las soluciones que más flexibilidad aporta al código cuando tenemos diferentes tareas que realizar:

- Command Query Separation (CQS) e Iteradores
- Curso Experto Arquitectura y Productividad
- Java Thread concurrencia y Runnables
- Java Pattern Predicate y filtrados