

# PATRÓN STRATEGY

**Strategy** es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

## APLICABILIDAD

- Utiliza el patrón Strategy cuando quieras utilizar distintas variantes de un algoritmo dentro de un objeto y poder cambiar de un algoritmo a otro durante el tiempo de ejecución.
- El patrón Strategy te permite alterar indirectamente el comportamiento del objeto durante el tiempo de ejecución asociándolo con distintos subobjetos que pueden realizar subtarefas específicas de distintas maneras.
- Utiliza el patrón Strategy cuando tengas muchas clases similares que sólo se diferencien en la forma en que ejecutan cierto comportamiento.
- El patrón Strategy te permite extraer el comportamiento variante para ponerlo en una jerarquía de clases separada y combinar las clases originales en una, reduciendo con ello el código duplicado.
- Utiliza el patrón para aislar la lógica de negocio de una clase, de los detalles de implementación de algoritmos que pueden no ser tan importantes en el contexto de esa lógica.
- El patrón Strategy te permite aislar el código, los datos internos y las dependencias de varios algoritmos, del resto del código. Los diversos clientes obtienen una interfaz simple para ejecutar los algoritmos y cambiarlos durante el tiempo de ejecución.
- Utiliza el patrón cuando tu clase tenga un enorme operador condicional que cambie entre distintas variantes del mismo algoritmo.
- El patrón Strategy te permite suprimir dicho condicional extrayendo todos los algoritmos para ponerlos en clases separadas, las cuales implementan la misma interfaz. El objeto original delega la ejecución a uno de esos objetos, en lugar de implementar todas las variantes del algoritmo.

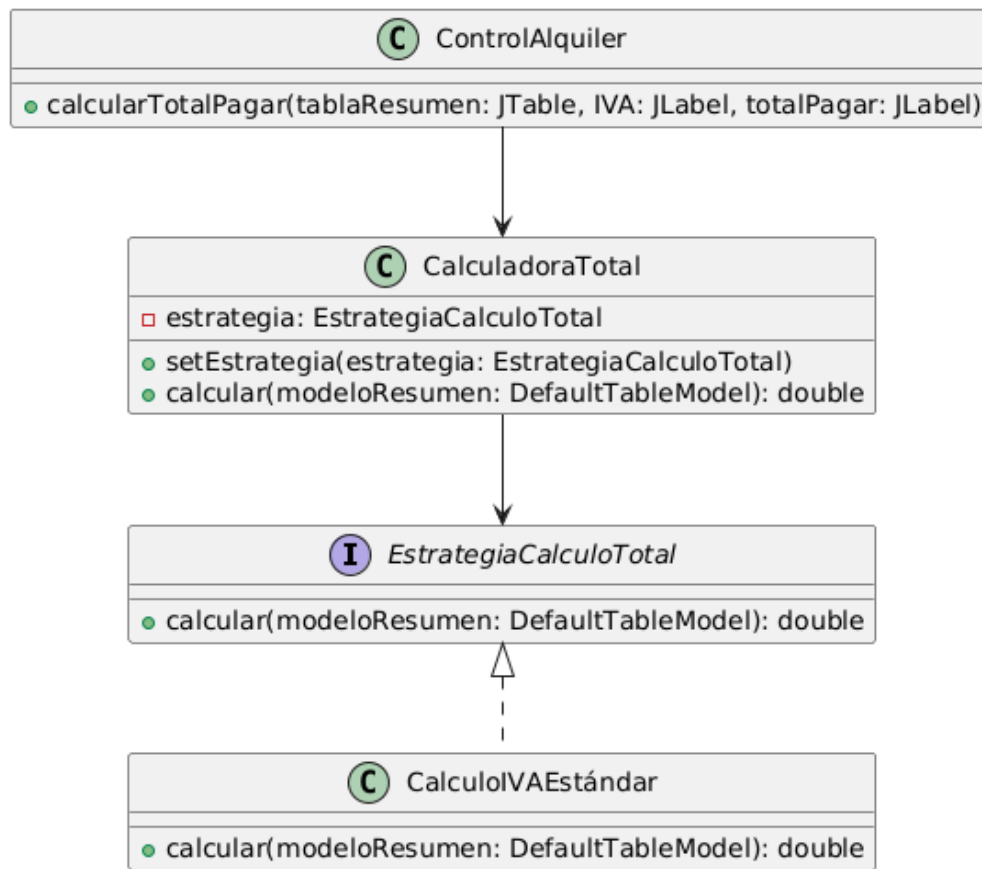
## CÓMO IMPLEMENTARLO

1. En la clase contexto, identifica un algoritmo que tienda a sufrir cambios frecuentes. También puede ser un enorme condicional que seleccione y ejecute una variante del mismo algoritmo durante el tiempo de ejecución.
2. Declara la interfaz estrategia común a todas las variantes del algoritmo.
3. Uno a uno, extrae todos los algoritmos y ponlos en sus propias clases. Todas deben implementar la misma interfaz estrategia.
4. En la clase contexto, añade un campo para almacenar una referencia a un objeto de estrategia. Proporciona un modificador *set* para sustituir valores de ese campo. La clase contexto debe trabajar con el objeto de estrategia únicamente a través de la interfaz estrategia. La clase contexto puede definir una interfaz que permita a la estrategia acceder a sus datos.
5. Los clientes de la clase contexto deben asociarla con una estrategia adecuada que coincida con la forma en la que esperan que la clase contexto realice su trabajo principal.

## PROS Y CONTRAS

- Puedes intercambiar algoritmos usados dentro de un objeto durante el tiempo de ejecución.
- Puedes aislar los detalles de implementación de un algoritmo del código que lo utiliza.
- Puedes sustituir la herencia por composición.
- *Principio de abierto/cerrado.* Puedes introducir nuevas estrategias sin tener que cambiar el contexto.
- Si sólo tienes un par de algoritmos que raramente cambian, no hay una razón real para complicar el programa en exceso con nuevas clases e interfaces que vengan con el patrón.
- Los clientes deben conocer las diferencias entre estrategias para poder seleccionar la adecuada.
- Muchos lenguajes de programación modernos tienen un soporte de tipo funcional que te permite implementar distintas versiones de un algoritmo dentro de un grupo de funciones anónimas. Entonces puedes utilizar estas funciones exactamente como habrías utilizado los objetos de estrategia, pero sin saturar tu código con clases e interfaces adicionales.

## DIAGRAMA UML



## IMPLEMENTACION EN EL PROYECTO

### ¿Por qué se aplicó el patrón Strategy?

En tu sistema de alquiler se calcula el total a pagar con un IVA fijo del 18%. Sin embargo, este cálculo podría variar en el futuro:

- Clientes con IVA reducido (10%)
- Ventas sin IVA
- Descuentos especiales

Modificar directamente el método `calcularTotalPagar()` cada vez que cambie el cálculo **viola el principio de abierto/cerrado (OCP)** del SOLID: deberíamos poder **agregar nuevas estrategias sin modificar código existente**.

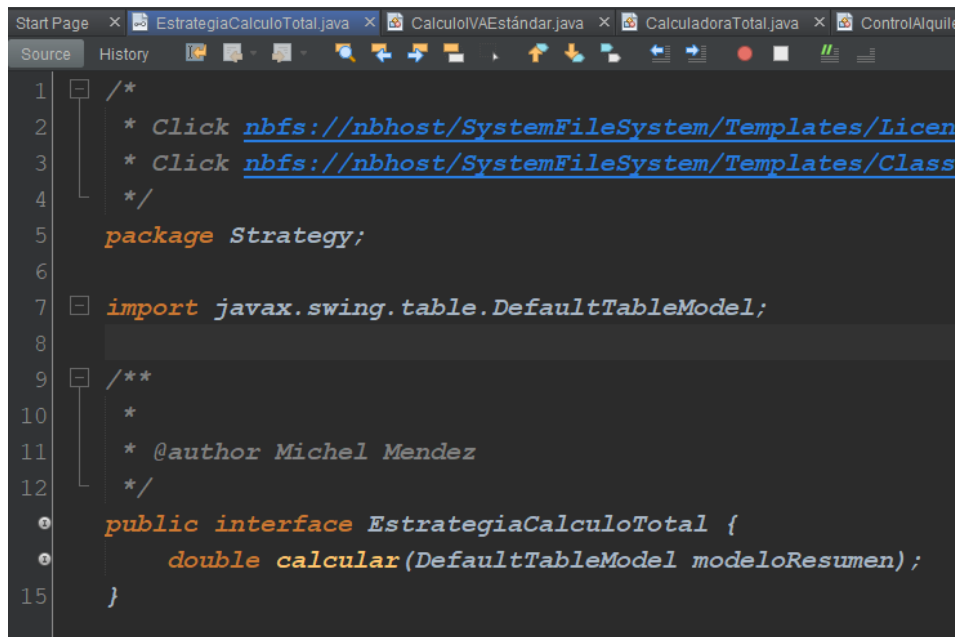
### Solución con Strategy:

Se encapsuló el algoritmo de cálculo del IVA dentro de una interfaz llamada `EstrategiaCalculoTotal`, y se definieron distintas clases con su propia lógica (por ejemplo: 18%, 10%, etc.). Luego, en el código que hace el cálculo (`ControlAlquiler`), se usa la **estrategia seleccionada dinámicamente**.

## EXPLICACION DEL CODIGO

### EstrategiaCalculoTotal.java

Esta es la **interfaz base**, que define el método:



```
1  /*
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Class
4   */
5   package Strategy;
6
7   import javax.swing.table.DefaultTableModel;
8
9   /**
10    *
11    * @author Michel Mendez
12    */
13   public interface EstrategiaCalculoTotal {
14       double calcular(DefaultTableModel modeloResumen);
15   }
```

**Propósito:** Permite que cualquier clase que implemente esta interfaz defina su propia forma de calcular el total (IVA, descuento, etc.).

```

1  /*
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to view and
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit
4   */
5   package Strategy;
6
7   import javax.swing.table.DefaultTableModel;
8
9   /**
10    *
11    * @author Michel Mendez
12    */
13   public class CalculoIVAEstándar implements EstrategiaCalculoTotal {
14       @Override
15       public double calcular(DefaultTableModel modeloResumen) {
16           double subtotal = 0;
17           for (int i = 0; i < modeloResumen.getRowCount(); i++) {
18               subtotal += (double) modeloResumen.getValueAt(i, 4);
19           }
20           return subtotal * 0.18;
21       }
22   }

```

**Propósito:** Implementa el cálculo actual estándar con un 18% de IVA.

```

1  /*
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to view and
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit
4   */
5   package Strategy;
6
7   import javax.swing.table.DefaultTableModel;
8
9   /**
10    *
11    * @author Michel Mendez
12    */
13   public class CalculadoraTotal {
14       private EstrategiaCalculoTotal estrategia;
15
16       public void setEstrategia(EstrategiaCalculoTotal estrategia) {
17           this.estrategia = estrategia;
18       }
19
20       public double calcular(DefaultTableModel modeloResumen) {
21           return estrategia.calcular(modeloResumen);
22       }
23   }

```

**Propósito:** Es el **contexto** del patrón Strategy. Usa la estrategia seleccionada para hacer el cálculo.

```

223     }
224
225     public void calcularTotalPagar(JTable tablaResumen, JLabel IVA, JLabel totalPagar) {
226
227         DefaultTableModel modelo = (DefaultTableModel) tablaResumen.getModel();
228
229         double totalSubtotal = 0;
230
231         for (int i = 0; i < modelo.getRowCount(); i++) {
232             totalSubtotal += (double) modelo.getValueAt(row: i, column: 4);
233         }
234
235         // Aplicacion de Strategy
236         Strategy.CalculadoraTotal calculadora = new Strategy.CalculadoraTotal();
237         calculadora.setEstrategia(new Strategy.CalculoIVAEstándar());
238
239         double totalIva = calculadora.calcular(modeloResumen: modelo);
240         double totalFinal = totalSubtotal;
241
242         totalPagar.setText(text: String.format(format: "%.2f", args: totalFinal));
243         IVA.setText(text: String.format(format: "%.2f", args: totalIva));
244     }
245 }

```

**Propósito:** Selecciona y aplica la estrategia concreta para calcular el IVA.

## CONCLUSION

La incorporación del patrón Strategy en el sistema de alquiler ha demostrado ser una decisión de diseño acertada, ya que permite desacoplar y encapsular el algoritmo de cálculo del total a pagar con IVA. Gracias a esta implementación, el sistema gana en flexibilidad, escalabilidad y mantenibilidad. Además, se prepara para futuros escenarios donde el cálculo del monto a pagar pueda variar según el tipo de cliente, promociones u otras reglas de negocio. Esta mejora no solo optimiza el código existente, sino que también sienta las bases para un crecimiento ordenado y alineado con los principios de diseño SOLID, en especial el principio de abierto/cerrado. En conjunto, esta solución contribuye a una arquitectura más robusta, limpia y preparada para el cambio.

## REFERENCIAS

Strategy. (s. f.). <https://refactoring.guru/es/design-patterns/strategy>