PATRONES DE DISEÑO

ESTRATEGIA

MÉTODO PLANTILLA

MVC

GRUPO 4:
Judith Andrés
Sergio Añón,
Silvia Cañas
Roberto Aibar

ESTRATEGIA

Strategy es un patrón de diseño de comportamiento que convierte un grupo de comportamientos en objetos y los hace intercambiables dentro del objeto de contexto original.

El objeto original, llamado contexto, contiene una referencia a un objeto de estrategia y le delega la ejecución del comportamiento. Para cambiar la forma en que el contexto realiza su trabajo, otros objetos pueden sustituir el objeto de estrategia actualmente vinculado, por otro.

😞**Problema**
Un día decidiste crear una aplicación de navegación para viajeros ocasionales. La aplicación giraba alrededor de un bonito mapa que ayudaba a los usuarios a orientarse rápidamente en cualquier ciudad.

Una de las funciones más solicitadas para la aplicación era la planificación automática de rutas. Un usuario debía poder introducir una dirección y ver la ruta más rápida a ese destino mostrado en el mapa.

La primera versión de la aplicación sólo podía generar las rutas sobre carreteras. Las personas que viajaban en coche estaban locas de alegría. Pero, aparentemente, no a todo el mundo le gusta conducir durante sus vacaciones. De modo que, en la siguiente actualización, añadiste una opción para crear rutas a pie. Después, añadiste otra opción para permitir a las personas utilizar el transporte público en sus rutas.

Sin embargo, esto era sólo el principio. Más tarde planeaste añadir la generación de rutas para ciclistas, y más tarde, otra opción para trazar rutas por todas las atracciones turísticas de una ciudad.

Aunque desde una perspectiva comercial la aplicación era un éxito, la parte técnica provocaba muchos dolores de cabeza. Cada vez que añadías un nuevo algoritmo de enrutamiento, la clase principal del navegador doblaba su tamaño. En cierto momento, se volvió demasiado difícil de mantener.

Cualquier cambio en alguno de los algoritmos, ya fuera un sencillo arreglo de un error o un ligero ajuste de la representación de la calle, afectaba a toda la clase, aumentando las probabilidades de crear un error en un código ya funcional.

Además, el trabajo en equipo se volvió ineficiente. Tus compañeros, contratados tras el exitoso lanzamiento, se quejaban de que dedicaban demasiado tiempo a resolver

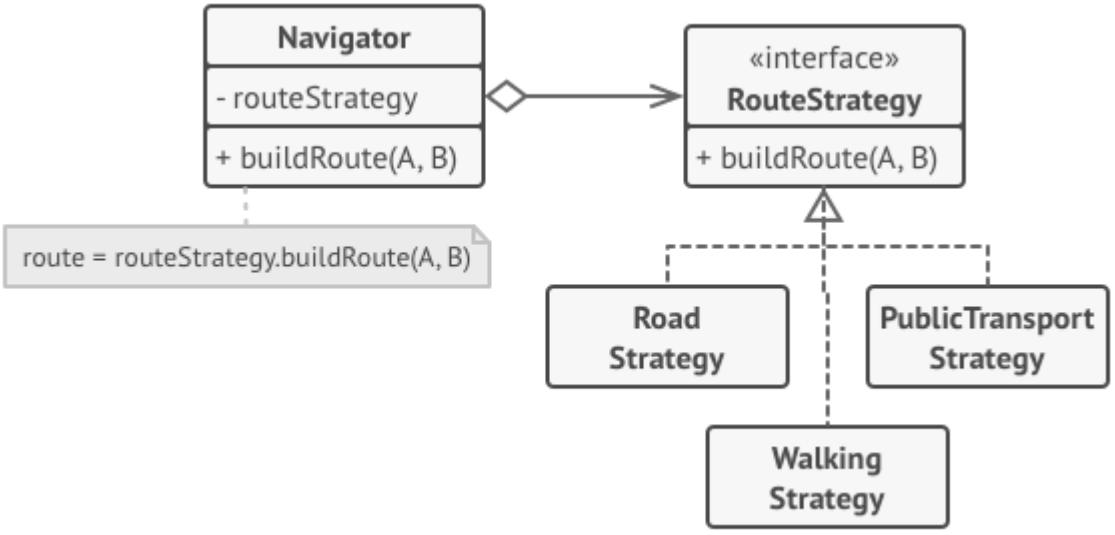
conflictos de integración. Implementar una nueva función te exige cambiar la misma clase enorme, entrando en conflicto con el código producido por otras personas.

😍Solución
El patrón Strategy sugiere que tomes esa clase que hace algo específico de muchas formas diferentes y extraigas todos esos algoritmos para colocarlos en clases separadas llamadas *estrategias*.

La clase original, llamada *contexto*, debe tener un campo para almacenar una referencia a una de las estrategias. El contexto delega el trabajo a un objeto de estrategia vinculado en lugar de ejecutarlo por su cuenta.

La clase contexto no es responsable de seleccionar un algoritmo adecuado para la tarea. En lugar de eso, el cliente pasa la estrategia deseada a la clase contexto. De hecho, la clase contexto no sabe mucho acerca de las estrategias. Funciona con todas las estrategias a través de la misma interfaz genérica, que sólo expone un único método para disparar el algoritmo encapsulado dentro de la estrategia seleccionada.

De esta forma, el contexto se vuelve independiente de las estrategias concretas, así que puedes añadir nuevos algoritmos o modificar los existentes sin cambiar el código de la clase contexto o de otras estrategias.



En nuestra aplicación de navegación, cada algoritmo de enrutamiento puede extraerse y ponerse en su propia clase con un único método `crearRuta`. El método acepta un origen y un destino y devuelve una colección de puntos de control de la ruta.

Incluso contando con los mismos argumentos, cada clase de enrutamiento puede crear una ruta diferente. A la clase navegadora principal no le importa qué algoritmo se selecciona ya que su labor principal es representar un grupo de puntos de control en el mapa. La clase tiene un método para cambiar la estrategia activa de enrutamiento, de modo que sus clientes, como los botones en la interfaz de usuario, pueden sustituir el comportamiento seleccionado de enrutamiento por otro.

Aplicabilidad

1) Utiliza el patrón Strategy cuando quieras utiliza 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 subtareas específicas de distintas maneras.

2) 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.

3) 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.

4) Utiliza el patrón cuando tu clase tenga un enorme operador condicional que cambie entre distintas variantes del mismo algoritmo.

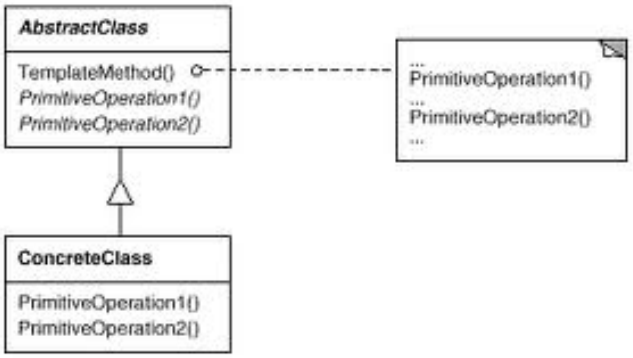
El patrón Strategy te permite suprimir dicho condicional extrayendo todos los algoritmos para ponerlos en clases separadas, las cuales implementan la misma interfaz. El objeto original delega la ejecución a uno de esos objetos, en lugar de implementar todas las variantes del algoritmo

MÉTODO PLANTILLA

Template Method es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.

El patrón Template Method se puede reconocer por los métodos de comportamiento que ya tienen un comportamiento “por defecto” definido por la clase base.

La solución que propone el patrón Template Method es abstraer todo el comportamiento que comparten las entidades en una clase (abstracta) de la que, posteriormente, extenderán dichas entidades. Esta superclase definirá un método que contendrá el esqueleto de ese algoritmo común (método plantilla o template method) y delegará determinada responsabilidad en las clases hijas, mediante uno o varios métodos abstractos que deberán implementar.



😞Problema

Imagina que estás creando una aplicación de minería de datos que analiza documentos corporativos. Los usuarios suben a la aplicación documentos en varios formatos (PDF, DOC, CSV) y ésta intenta extraer la información relevante de estos documentos en un formato uniforme.

La primera versión de la aplicación sólo funcionaba con archivos DOC. La siguiente versión podía soportar archivos CSV. Un mes después, le “enseñaste” a extraer datos de archivos PDF.

😍Solución

El patrón Template Method sugiere que dividas un algoritmo en una serie de pasos, conviertas estos pasos en métodos y coloques una serie de llamadas a esos métodos dentro de un único *método plantilla*. Los pasos pueden ser abstractos, o contar con una implementación por defecto. Para utilizar el algoritmo, el cliente debe aportar su

propia subclase, implementar todos los pasos abstractos y sobrescribir algunos de los opcionales si es necesario (pero no el propio método plantilla).

Utiliza el patrón **Template Method** cuando quieras permitir a tus clientes que extiendan únicamente pasos particulares de un algoritmo, pero no todo el algoritmo o su estructura.

El patrón **Template Method** te permite convertir un algoritmo monolítico en una serie de pasos individuales que se pueden extender fácilmente con subclases, manteniendo intacta la estructura definida en una superclase.

Utiliza el patrón cuando tengas muchas clases que contengan **algoritmos** casi idénticos, pero con algunas diferencias mínimas. Como resultado, puede que tengas que modificar todas las clases cuando el algoritmo cambie.

Cuando conviertes un algoritmo así en un método plantilla, también puedes elevar los pasos con implementaciones similares a una superclase, eliminando la duplicación del código. El código que varía entre subclases puede permanecer en las subclases.

Cómo implementarlo

Analiza el algoritmo objetivo para ver si puedes dividirlo en pasos. Considera qué pasos son comunes a todas las subclases y cuáles siempre serán únicos.

Crea la clase base abstracta y declara el método plantilla y un grupo de métodos abstractos que representen los pasos del algoritmo. Perfila la estructura del algoritmo en el método plantilla ejecutando los pasos correspondientes. Considera declarar el método plantilla como `final` para evitar que las subclases lo sobrescriban.

No hay problema en que todos los pasos acaben siendo abstractos. Sin embargo, a algunos pasos les vendría bien tener una implementación por defecto. Las subclases no tienen que implementar esos métodos.
Piensa en añadir ganchos entre los pasos cruciales del algoritmo.

Para cada variación del algoritmo, crea una nueva subclase concreta. Ésta *debe* implementar todos los pasos abstractos, pero también *puede* sobrescribir algunos de los opcionales.

MVC Modelo vista controlador (MVC)

Modelo Vista Controlador (MVC) es un estilo de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

- El **Modelo** que contiene una representación de los datos que maneja el sistema, su lógica de negocio, y sus mecanismos de persistencia.
- La **Vista**, o interfaz de usuario, que compone la información que se envía al cliente y los mecanismos de interacción con éste.
- El **Controlador**, que actúa como intermediario entre el Modelo y la Vista, gestionando el flujo de información entre ellos y las transformaciones para adaptar los datos a las necesidades de cada uno.

El modelo es el responsable de:

- Acceder a la capa de almacenamiento de datos. Lo ideal es que el modelo sea independiente del sistema de almacenamiento.

  Eso se consigue dividiendo el modelo en dos paquetes:

  modelo.beans→donde se declaran las clases java. Cada clase corresponde a una tabla de la base de datos.

  modelo.DAO→ahí se definen los métodos para acceder a los datos. (se puede utilizar JPA, JDBC…)

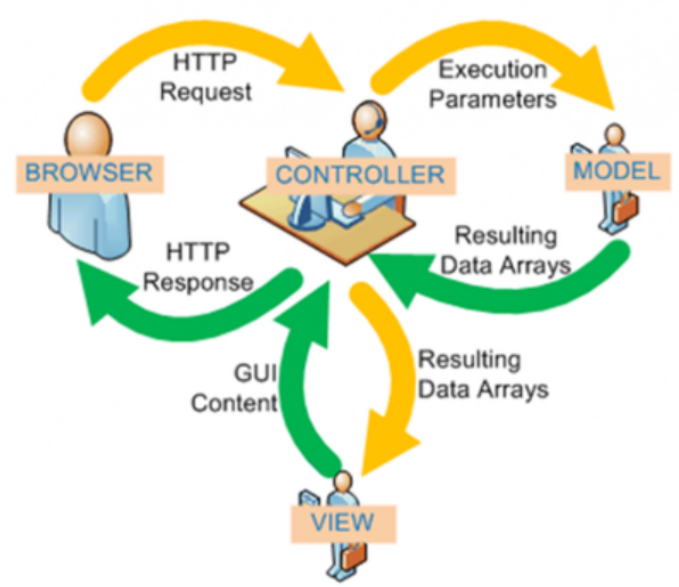
- Define las reglas de negocio (la funcionalidad del sistema). Lleva un registro de las vistas y controladores del sistema.
- Si estamos ante un modelo activo, notificará a las vistas los cambios que en los datos pueda producir un agente externo (por ejemplo, un fichero por lotes que actualiza los datos, un temporizador que desencadena una inserción, etc.).

El controlador es responsable de:

- Recibe los eventos de entrada (un clic, un cambio en un campo de texto, los datos de un formulario de acceso para hacer logIn,etc.).
- Contiene reglas de gestión de eventos, del tipo "SI Evento Z, entonces Acción W". Estas acciones pueden suponer peticiones al modelo o a las vistas. Una de estas peticiones a las vistas puede ser una llamada al método "Actualizar()". Una petición al modelo puede ser "Obtener_tiempo_de_entrega (nueva_orden_de_venta)".

Las vistas son responsables de:

- Recibir datos del modelo y los muestra al usuario.
- Tienen un registro de su controlador asociado (normalmente porque además lo instancia).
- Pueden dar el servicio de "Actualización()", para que sea invocado por el controlador o por el modelo (cuando es un modelo activo que informa de los cambios en los datos producidos por otros agentes).



EJEMPLOS CON CÓDIGO

ESTRATEGIA:

```
// La interfaz estrategia declara operaciones comunes a todas
// las versiones soportadas de algún algoritmo. El contexto
// utiliza esta interfaz para invocar el algoritmo definido por
// las estrategias concretas.
interface Strategy is
    method execute(a, b)

// Las estrategias concretas implementan el algoritmo mientras
// siguen la interfaz estrategia base. La interfaz las hace
// intercambiables en el contexto.
class ConcreteStrategyAdd implements Strategy is
    method execute(a, b) is
        return a + b

class ConcreteStrategySubtract implements Strategy is
    method execute(a, b) is
        return a - b

class ConcreteStrategyMultiply implements Strategy is
    method execute(a, b) is
        return a * b

// El contexto define la interfaz de interés para los clientes.
class Context is
    // El contexto mantiene una referencia a uno de los objetos
    // de estrategia. El contexto no conoce la clase concreta de
    // una estrategia. Debe trabajar con todas las estrategias a
    // través de la interfaz estrategia.
    private strategy: Strategy

    // Normalmente, el contexto acepta una estrategia a través
    // del constructor y también proporciona un setter
    // (modificador) para poder cambiar la estrategia durante el
    // tiempo de ejecución.
    method setStrategy(Strategy strategy) is
        this.strategy = strategy

    // El contexto delega parte del trabajo al objeto de
    // estrategia en lugar de implementar varias versiones del
    // algoritmo por su cuenta.
    method executeStrategy(int a, int b) is
        return strategy.execute(a, b)


// El código cliente elige una estrategia concreta y la pasa al
// contexto. El cliente debe conocer las diferencias entre
```

```
// estrategias para elegir la mejor opción.
class ExampleApplication is
    method main() is
        Create context object.

        Read first number.
        Read last number.
        Read the desired action from user input.

        if (action == addition) then
            context.setStrategy(new ConcreteStrategyAdd())

        if (action == subtraction) then
            context.setStrategy(new ConcreteStrategySubtract())

        if (action == multiplication) then
            context.setStrategy(new ConcreteStrategyMultiply())

        result = context.executeStrategy(First number, Second number)

        Print result.
```

MÉTODO PLANTILLA:

```
// La clase abstracta define un método plantilla que contiene un
// esqueleto de algún algoritmo compuesto por llamadas,
// normalmente a operaciones primitivas abstractas. Las
// subclases concretas implementan estas operaciones, pero dejan
// el propio método plantilla intacto.
class GameAI is
    //El método plantilla define el esqueleto de un algoritmo.
    method turn() is
        collectResources()
        buildStructures()
        buildUnits()
        attack()

    // Algunos de los pasos se pueden implementar directamente
    // en una clase base.
    method collectResources() is
        foreach (s in this.builtStructures) do
            s.collect()

    // Y algunos de ellos pueden definirse como abstractos.
    abstract method buildStructures()
    abstract method buildUnits()

    // Una clase puede tener varios métodos plantilla.
    method attack() is
        enemy = closestEnemy()
        if (enemy == null)
```

```
            sendScouts(map.center)
        else
            sendWarriors(enemy.position)

    abstract method sendScouts(position)
    abstract method sendWarriors(position)

// Las clases concretas tienen que implementar todas las
// operaciones abstractas de la clase base, pero no deben
// sobrescribir el propio método plantilla.
class OrcsAI extends GameAI is
    method buildStructures() is
        if (there are some resources) then
            // Construye granjas, después cuarteles y después
            // fortaleza.

    method buildUnits() is
        if (there are plenty of resources) then
            if (there are no scouts)
                // Crea peón y añádelo al grupo de exploradores.
            else
                // Crea soldado, añádelo al grupo de guerreros.

    // ...

    method sendScouts(position) is
        if (scouts.length > 0) then
            // Envía exploradores a posición.

    method sendWarriors(position) is
        if (warriors.length > 5) then
            // Envía guerreros a posición.

// Las subclases también pueden sobrescribir algunas operaciones
// con una implementación por defecto.
class MonstersAI extends GameAI is
    method collectResources() is
        // Los monstruos no recopilan recursos.

    method buildStructures() is
        // Los monstruos no construyen estructuras.

    method buildUnits() is
        // Los monstruos no construyen unidades.
```

Template Method

```
]public class Delantero extends Jugador {

    private final int golesMarcados;

    public Delantero(String nombre, int minutosJugados, int salario,
            int golesMarcados) {
        super(nombre, minutosJugados, salario);
        this.golesMarcados = golesMarcados;
    }

    @Override
    public float calculaPuntosPorObjetivos() {
        return 30 * (golesMarcados / super.getPartidosJugados());
    }

    @Override
    public float getPuntosPenalizacionPorSalarioAlto() {
        return (float) (salario * 0.1);
    }

}
```

Subclase Jugador

```
]public class Portero extends Jugador {

    private final int golesEncajados;

    public Portero(String nombre, int minutosJugados, int salario,
            int golesEncajados) {
        super(nombre, minutosJugados, salario);
        this.golesEncajados = golesEncajados;
    }

    @Override
    public float calculaPuntosPorObjetivos() {
        return 50 - (30 * golesEncajadosPorPartido());
    }

    private float golesEncajadosPorPartido() {
        return golesEncajados / super.getPartidosJugados();
    }

    @Override
    public float getPuntosPenalizacionPorSalarioAlto() {
        return (float) (salario * 0.08);
    }

}
```

```java
public class Prueba {

    public static void main(String args[]) {

        final int TOTAL_MINUTOS_JUGADOS_EQUIPO = 360;

        final Jugador jugador1 = new Delantero("Cristiano Ronaldo", 235, 14, 4);
        escribeValoracionJugador(jugador1, TOTAL_MINUTOS_JUGADOS_EQUIPO);

        final Jugador jugador2 = new Portero("Iker Casillas", 360, 10, 2);
        escribeValoracionJugador(jugador2, TOTAL_MINUTOS_JUGADOS_EQUIPO);

        final Jugador jugador3 = new Delantero("Messi", 280, 13, 0);
        escribeValoracionJugador(jugador3, TOTAL_MINUTOS_JUGADOS_EQUIPO);

        final Jugador jugador4 = new Portero("Victor Valdés", 360, 11, 6);
        escribeValoracionJugador(jugador4, TOTAL_MINUTOS_JUGADOS_EQUIPO);

    }

    private static void escribeValoracionJugador(Jugador jugador,
            int totalMinutosJugadosEquipo) {
        System.out.println("El jugador " + jugador.getNombre()
                + " obtuvo una valoración de "
                + jugador.calculaValoracion(totalMinutosJugadosEquipo));
    }

}
```

```
El jugador Cristiano Ronaldo obtuvo una valoración de GALACTICO
El jugador Iker Casillas obtuvo una valoración de BUENO
El jugador Messi obtuvo una valoración de MALISIMO
El jugador Victor Valdés obtuvo una valoración de MALO
```

MCV:

Crearemos un modelo llamado Biblioteca que incluya un listado de libros y que al crearse, añada unos libros por defecto (por tener unos datos por efecto)

```
public class Biblioteca
{
    public List<Libro> Libros { get; set; }

    public Biblioteca()
    {
        Libros = new List<Libro>
                            {
                                new Libro { Isbn = "11122", Titulo = "Los Piratas del Caribe", TipoLibro  = "Novela"},
                                new Libro { Isbn = "22211", Titulo = "Los Pilares de la tierra", TipoLibro  = "Novela"},
                                new Libro { Isbn = "33311", Titulo = "Steve Jobs", TipoLibro  = "Biografía"}
                            };
    }
}
```

```
public class BibliotecaController : Controller
{
    Biblioteca miBiblioteca = new Biblioteca();
```

Se genera un código que permite la visualización, el alta, la edición y el borrado de libros.

```
@model IEnumerable<CSI_Biblioteca.Models.Libro>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            Isbn
        </th>
        <th>
            Titulo
```

El resultado es

El resto de métodos se implementan de una forma similar.

Vamos a incorporar dos métodos adicionales al modelo Biblioteca que usaremos desde el controlador.

```
        public int NumeroLibros()
        {
            return Libros.Count();
        }

        public Libro ObtenerPorIsbn(string isbn)
        {
            foreach (var libroBuscar in Libros)
            {
                if (libroBuscar.Isbn == isbn)
                {
                    return libroBuscar;
                }
            }

            return null;
        }
```

En el controlador vamos a cambiar la declaración del objeto porque sino los cambios que hagamos alta o baja no se refejarán (en cada llamada se redeclara el objeto). Añadomos static para que los datos se mantengan entre llamadas.

```
        static Biblioteca miBiblioteca = new Biblioteca();
```

Al segundo método le añadimos el código para dar de alta un libro a la biblioteca

```
[HttpPost]
public ActionResult Create(FormCollection collection)
{
    try
    {
        miBiblioteca.Libros.Add(new Libro
                                    {
                                       Isbn =  (miBiblioteca.Libros.Count() + 1).ToString(),
                                       Titulo = collection["Titulo"],
                                       TipoLibro = collection["Categoria"]
                                    });

        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```