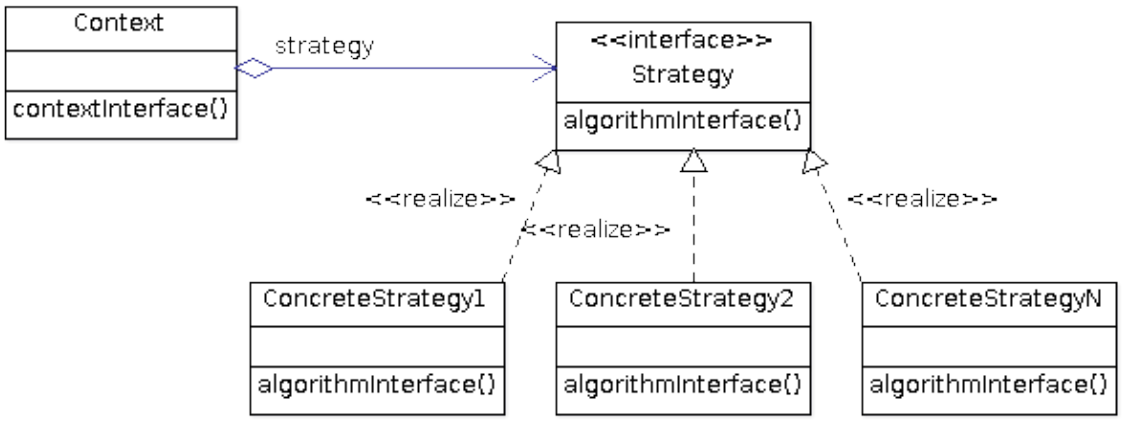
Strategy

Diagrama de clases e interfaces



Intención

- Define un grupo de clases que representan un conjunto de comportamientos intercambiables en tiempo de ejecución.

Motivación

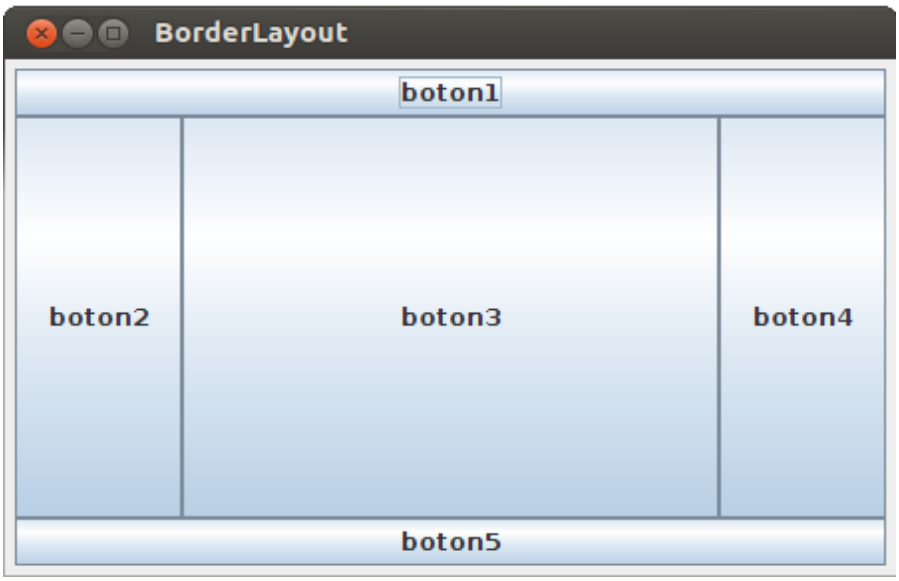
En un escenario de programación gráfica basado en AWT/Swing, la clase LayoutManager ofrece diferentes estrategias a la hora de disponer los controles en un contenedor. Veamos algunas de estas estrategias:

BoxLayout: Cada control ocupa una de las siguientes posiciones: Norte, Sur, Este, Oeste o Centro.

El siguiente código configura un pequeño ejemplo sobre este organizador de presentación (layout):

```
...

contentPane = new JPanel();

contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
```

```
contentPane.setLayout(new BorderLayout(0, 0));
setContentPane(contentPane);
JButton btnNewButton1 = new JButton("boton1");
contentPane.add(btnNewButton1, BorderLayout.NORTH);
JButton btnNewButton2 = new JButton("boton2");
contentPane.add(btnNewButton2, BorderLayout.WEST);
JButton btnNewButton3 = new JButton("boton3");
contentPane.add(btnNewButton3, BorderLayout.CENTER);
JButton btnNewButton4 = new JButton("boton4");
contentPane.add(btnNewButton4, BorderLayout.EAST);
JButton btnNewButton5 = new JButton("boton5");
contentPane.add(btnNewButton5, BorderLayout.SOUTH);
...
```



BoxLayout: Los controles se disponen uno al lado del otro formando una fila, o bien uno debajo del otro formando una columna. La primera disposición se da cuando en el constructor indicamos el parámetro BoxLayout.X_AXIS y la segunda con BoxLayout.Y_AXIS (realmente hay dos opciones más, de uso menos frecuente:

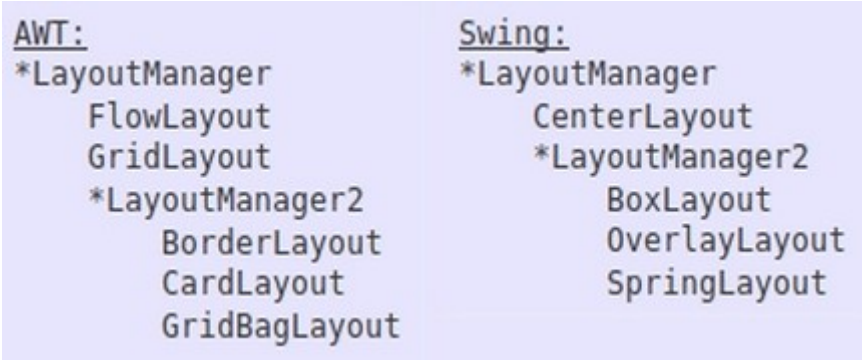
LINE_AXIS y PAGE_AXIS, que dependen de la propiedad ComponentOrientation del contenedor).

Por ejemplo para la disposición horizontal:

```
...
contentPane = new JPanel();
contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
setContentPane(contentPane);
contentPane.setLayout(new BoxLayout(contentPane, BoxLayout.X_AXIS));
JButton btnNewButton1 = new JButton("boton1");
contentPane.add(btnNewButton1);
JButton btnNewButton2 = new JButton("boton2");
contentPane.add(btnNewButton2);
JButton btnNewButton3 = new JButton("boton3");
contentPane.add(btnNewButton3);
JButton btnNewButton4 = new JButton("boton4");
contentPane.add(btnNewButton4);
JButton btnNewButton5 = new JButton("boton5");
contentPane.add(btnNewButton5);
```

Cambiando a Y_AXIS:

Existen varios layouts más: FlowLayout, GridLayout, GridBagLayout, etc. La siguiente imagen muestra la jerarquía de layouts propios de AWT y los proporcionados por Swing:



En la jerarquía de AWT:

- FlowLayout y GridLayout son clases que implementan la interfaz LayoutManager.
- LayoutManager2 es una interfaz que extiende a la interfaz LayoutManager.
- BorderLayout, CardLayout y GridBagLayout son clases que implementan la interfaz LayoutManager2.

En la jerarquía de Swing:

- CenterLayout es una clase que implementa la interfaz LayoutManager.
- BoxLayout, OverlayLayout y SpringLayout son clases que implementan la interfaz LayoutManager2.

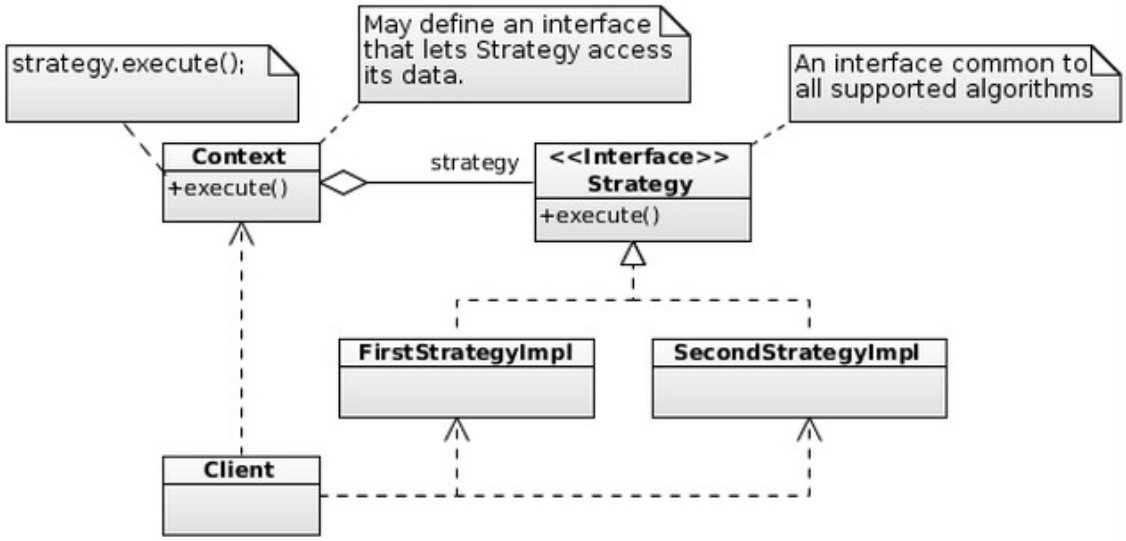
Del ejemplo visto, lo importante es apreciar como mediante el método setLayout() del contenedor podemos especificar la estrategia o política deseada para la disposición de los controles. Esta es la esencia del patrón Strategy: los algoritmos (layouts en el ejemplo) son intercambiables sin afectar al resto de la aplicación. Así, hemos visto que podíamos pasar al método setLayout() una subclase u otra de LayoutManager/LayoutManager2 para obtener una diferente organización de los controles.

El hecho de obtener resultados diferentes o iguales en función de una u otra estrategia no es un tema que incumba al patrón Strategy, sino a la lógica del programa. Por ejemplo, en un programa de ordenación de números, podríamos tener diferentes

algoritmos de ordenación. Tales algoritmos representarían las diferentes estrategias y se diferenciarían por ser más rápidos unos que otros, pero está claro que todos deberían obtener el mismo resultado: los números ordenados.

Implementación

El funcionamiento del patrón es el mostrado en el diagrama de clases siguiente:



Las clases participantes en el patrón son las siguientes:

- **Strategy:** Declara una interfaz común para todas las estrategias. La clase Context usa esta interfaz para así poder llamar a la estrategia definida por cualquier subclase (FirstStategyImpl, SecondStategyImpl, ...)
- **ConcreteStrategies (FirstStrategyImpl, SecondStrategyImpl):** Cada una implementa el algoritmo (estrategia, política) usando la interfaz Strategy.
- **Context:** Mantiene una referencia de tipo Strategy, la cual se configura con un objeto de ConcreteStrategies. Por otro lado, Context puede definir una interfaz para permitir que Strategy acceda a sus datos.

Aplicabilidad y Ejemplos

El uso del patrón Strategy es adecuado:

- Se tienen diversas clases relacionadas que difieren sólo en su comportamiento.
- Se requieren múltiples variaciones de un mismo algoritmo, es decir, varias implementaciones que proporcionan el mismo comportamiento.
- El comportamiento de una clase debe ser definido en tiempo de ejecución.
- Una clase presenta muchos comportamientos que se tratan con sentencias *if-elseif* o *switch-case*.

Ejemplo 1: Algoritmos de ordenación

Este ejemplo ordena una lista de 100 números reales comprendidos entre el 1.00 y el 100.00.

El funcionamiento es el siguiente:

Una clase cliente configura un objeto de contexto para ordenar la lista, primero mediante el método de la burbuja y después con el algoritmo QuickSort. El resultado de ambas ordenaciones, que lógicamente es el mismo, se muestra por pantalla, junto con el tiempo transcurrido (en nanosegundos) para cada método (estrategia) de ordenación. Por tanto, estamos ante un caso de diferente algoritmo y mismo comportamiento.

Comenzamos con una interfaz que define la única operación que necesitamos: ordenar()

Estrategia.java

```
package comportamiento.strategy.ordenacion;

public interface Estrategia {
	public void ordenar(double[] lista);
}
```

Veamos ahora las subclases (estrategias):

EstrategiaBurbuja.java

```
package comportamiento.strategy.ordenacion;


public class EstrategiaBurbuja implements Estrategia {
      public void ordenar(double[] lista) {
            int newLowest = 0;            // index of first comparison
          int newHighest = lista.length-1;  // index of last
comparison


          while (newLowest < newHighest) {
              int highest = newHighest;
              int lowest  = newLowest;
              newLowest = lista.length;    // start higher than any
legal index
              for (int i=lowest; i<highest; i++) {
                  if (lista[i] > lista[i+1]) {
                     // exchange elements
                     double temp = lista[i];
                     lista[i] = lista[i+1];
                     lista[i+1] = temp;
                     if (i<newLowest) {
                         newLowest = i-1;
                         if (newLowest < 0) {
                             newLowest = 0;
                         }
                     } else if (i>newHighest) {
                         newHighest = i+1;
                     }
                  }
              }
          }
      }
```

```
}
```

EstrategiaQuickSort.java

```
package comportamiento.strategy.ordenacion;

public class EstrategiaQuickSort implements Estrategia {
	public void ordenar(double[] lista) {
		quicksort(lista, 0, lista.length - 1);
	}

	private void quicksort(double[] lista, int left, int right) {
		if (right <= left)
			return;
		int i = partition(lista, left, right);
		quicksort(lista, left, i - 1);
		quicksort(lista, i + 1, right);
	}

	private int partition(double[] lista, int left, int right) {
		int i = left;
		int j = right;
		while (true) {
			while (lista[i] < lista[right])
				i++;
			while (less(lista[right], lista[--j]))
				if (j == left)
					break;
			if (i >= j)
				break;
			exch(lista, i, j);
		}
```

```
            exch(lista, i, right);
            return i;
      }

      private boolean less(double x, double y) {
            return (x < y);
      }

      private void exch(double[] lista, int i, int j) {
            double swap = lista[i];
            lista[i] = lista[j];
            lista[j] = swap;
      }
}
```

Ahora, la clase de contexto. Esta clase es la que las clases clientes instancían y configuran con una u otra estrategia de ordenación:

Context.java

```
package comportamiento.strategy.ordenacion;

public class Contexto {
      private Estrategia estrategia;

      public void ordenarLista(double[] lista) {
            estrategia.ordenar(lista);
      }

      public Estrategia getEstrategia() {
            return estrategia;
      }
```

```
	public void setEstrategia(Estrategia estrategia) {
		this.estrategia = estrategia;
	}
}
```

Finalmente, la clase cliente:

MainClient.java

```
package comportamiento.strategy.ordenacion;

public class MainClient {

   private static double[] list = {
		71.8,9.2,33.4,88.6,4.9,51.32,70.11,72.72,21.29,67.01,
		42.22,55.44,94.00,57.43,54.18,45.77,89.39,37.32,25.39,
		12.02,7.19,56.23,53.66,81.91,92.03,36.34,62.32,40.99,
		22.12,77.33,24.45,34.00,50.43,91.92,97.08,10.11,3.01,
		69.61,5.00,68.10,6.09,73.55,31.43,100.00,82.44,59.33,
		41.91,76.48,61.47,32.34,16.02,44.38,29.55,90.06,8.66,
		35.34,75.06,87.87,27.88,18.12,15.45,20.20,95.00,96.01,
		85.99,79.56,13.77,47.05,11.02,58.55,43.66,78.78,26.00,
		39.32,86.05,14.99,74.45,52.55,98.01,1.11,64.11,84.32,
		99.09,80.58,28.59,66.44,2.88,19.01,60.99,38.98,46.77,
		30.54,65.43,23.20,83.66,93.03,48.43,49.47,17.87,63.49
	};

	private Contexto contexto;

	public MainClient() {
		contexto = new Contexto();
```

```
            contexto.setEstrategia(new EstrategiaBurbuja());
            executeAndShow();
            contexto.setEstrategia(new EstrategiaQuickSort());
            executeAndShow();
      }

      public static void main(String[] args) {
            new MainClient();
      }

      private void executeAndShow() {
            long tStart = System.nanoTime();
            contexto.ordenarLista(list);
            long tEnd = System.nanoTime();
            long dif = tEnd-tStart;

      System.out.println("---------------------------------------");
            for (int i = 0; i < list.length; i++) {
                  System.out.print(list[i] + " ");
            }
            System.out.println("\nTiempo de ordenacion: " + dif);
      }
}
```

Salida (parcial, por falta de espacio):

```
---------------------------------------
1.11 2.88 3.01 4.9 5.0 6.09 7.19 8.66 9.2 10.11 11.02 12.02 13.77 14.99 15.45 16.02 17.87
Tiempo de ordenacion: 1892988
---------------------------------------
1.11 2.88 3.01 4.9 5.0 6.09 7.19 8.66 9.2 10.11 11.02 12.02 13.77 14.99 15.45 16.02 17.87
Tiempo de ordenacion: 949229
```

Ejemplo 2: Cálculo de importes en el alquiler de películas

En el siguiente ejemplo, una empresa de alquiler de películas (videoclub), nos pide una aplicación que permita conocer el importe que se le tiene que cobrar a cada cliente. La persona interesada en el programa nos explica varias reglas de negocio:

Tipos de películas

Una película es del tipo Normal, Antigua o Estreno. Debe ser posible cambiar el tipo de cualquier película, por ejemplo, una película se puede dar de alta como Estreno y después de 2 meses debe poder cambiarse a Normal.

Precios

Cada una de los tipos de películas comentados anteriormente tiene una política de precios:

Antigua

A pagar 1,5€ si la película se devuelve antes de 3 días desde que se alquiló.

Si se devuelve después del tercer día, se pagará 1,5€ de suplemento por cada uno de estos días “extras”.

Normal

A pagar 2€ si la película se devuelve antes de 2 días desde que se alquiló.

Si se devuelve después del segundo día, se pagará 1,5€ de suplemento por cada uno de estos días “extras”.

Estreno

A pagar 3€ por día de préstamo, independientemente del día que se devuelva.

Puntos

El alquiler de películas proporciona puntos a los clientes, los cuales se pueden canjear por nuevos alquileres. Aún no están pensando la política de canjeo, por lo que en una primera fase de la aplicación, es suficiente con hacer que cada vez que se muestre lo que debe pagar un cliente, se muestren también los puntos generados.

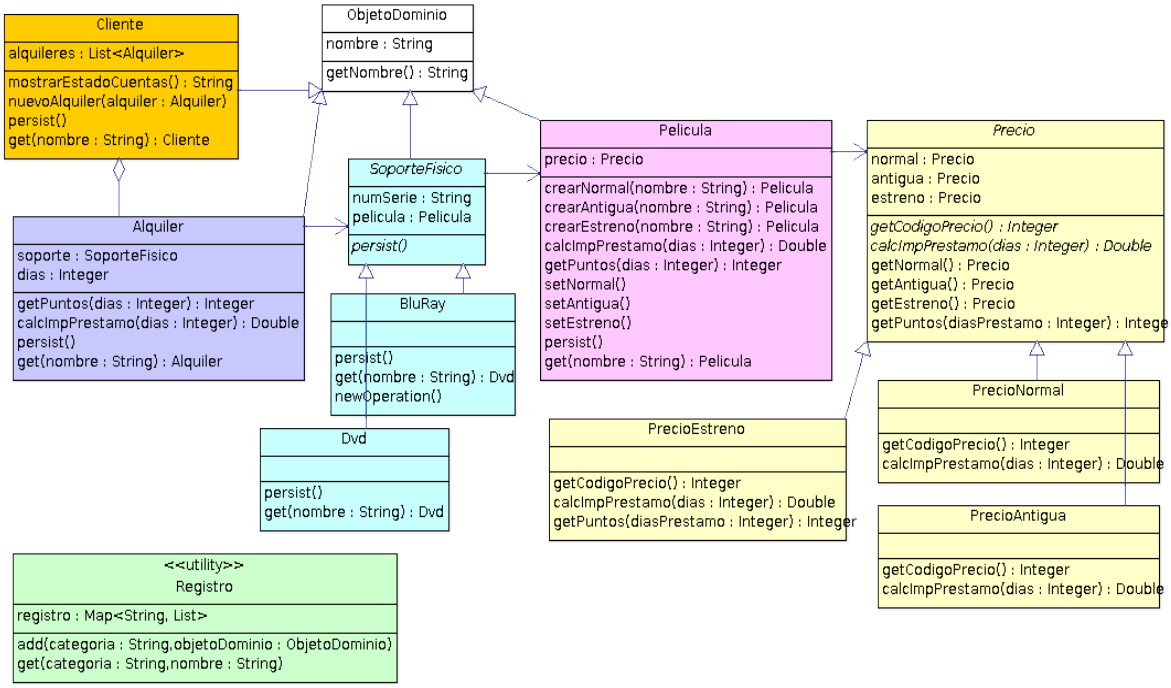
Lo que sí nos dicen, es que las películas de tipo Normal o Antigua generan 1 punto por alquiler (es indiferente cuando se devuelvan), mientras que las pertenecientes al tipo Estreno generan:

- 2 puntos por alquiler si el préstamo sólo ha durado 1 día.
- 1 puntos por alquiler si el préstamo ha durado más de 1 día.

Soportes físicos

Las películas se prestan en formato Dvd o BluRay, a petición del cliente. Esto no afecta al precio de los préstamos. Es posible que en un futuro próximo se añadan nuevos tipos de soportes.

Con toda la información anterior, diseñamos el siguiente diagrama de clases para la aplicación:



Nota: Para no complicar el ejemplo se ha hecho una simplificación burda a la gestión de alquileres y devoluciones, ya que en lugar de ofrecer un medio para alquilar películas y otro para devolver, todo se realiza de una vez, es decir, mediante la clase Alquiler se alquila tal película por tal periodo de tiempo, lo cual no es un modelo de gestión muy apropiado.

Veamos el código:

Comenzamos por la jerarquía de Precio, esto es, el corazón del ejemplo en tanto que ilustra el uso del patrón Estrategy.

Precio.java

```
package comportamiento.strategy.video;


public abstract class Precio {
      private static Precio normal = new PrecioNormal();
      private static Precio antigua = new PrecioAntigua();
      private static Precio estreno = new PrecioEstreno();

    public abstract double calcularImportePrestamo(int diasPrestamo);

    public int getPuntos(int diasPrestamo){
        return 1;
    }

    public static Precio getNormal() {
        return normal;
    }

    public static Precio getAntigua() {
        return antigua;
    }

    public static Precio getEstreno() {
        return estreno;
    }

}
```

Comentarios sobre la clase Precio:

- Contiene tres referencias estáticas, una para guardar un objeto de cada tipo de película. Para devolver tales objetos, dispone de los respectivos métodos estáticos. Por tanto, siempre que le piden una película de alguno de los tres tipos existentes, siempre devuelve la misma referencia (lo que ya va bien, pues sólo se quiere calcular el importe o los puntos, pues son conceptos que van ligados al tipo de película).
- Define 1 método abstracto que sus subclases deberán implementar:
  - *double calcularImportePrestamo(int diasPrestamo)*

  Este método es la estrategia que cada subclase implementa.
- Otro método digno de mención es el que sirve para calcular los puntos por alquiler que acumula cada cliente: *int getPuntos(int diasPrestamo)*

  Dado que sólo la subclase PrecioEstreno tiene una lógica distintas a las otras dos subclases, este método no se declara como abstracto, sino que se proporciona una implementación base que aprovecharan estas dos subclases. PrecioEstreno redefine este método.

Ahora veamos las subclases:

PrecioNormal.java

```
package comportamiento.strategy.video;


class PrecioNormal extends Precio {
      @Override
      public double calcularImportePrestamo(int diasPrestamo) {
        double resultado = 2;
        if (diasPrestamo > 2) {
            resultado += (diasPrestamo - 2) * 1.5;
        }
        return resultado;
    }
```

```
}
```

PrecioAntigua.java

```
package comportamiento.strategy.video;

class PrecioAntigua extends Precio {
      @Override
      public double calcularImportePrestamo(int diasPrestamo){
        double resultado = 1.5;
        if (diasPrestamo > 3) {
            resultado += (diasPrestamo - 3) * 1.5;
        }
        return resultado;
    }
}
```

PrecioEstreno.java

```
package comportamiento.strategy.video;

class PrecioEstreno extends Precio {
      @Override
      public double calcularImportePrestamo(int diasPrestamo) {
        return diasPrestamo * 3;
    }
      @Override
      public int getPuntos(int diasPrestamo) {
        return (diasPrestamo > 1) ? 1 : 2;
    }
}
```

Veamos ahora las entidades, comenzando por la clase ObjetoDominio. Todas las clases de entidad extienden a ObjetoDominio, ya que esta clase sirve para tener un ancestro común para todas las entidades. ObjetoDominio se encarga de almacenar el nombre de cualquier objeto, lo cual es muy útil cuando queremos guardar y recuperar objetos de entidad mediante un mapa, tal y como hacemos utilizando la clase Registro (lo vemos en breve).

ObjetoDominio.java

```
package comportamiento.strategy.video;


public class ObjetoDominio {


	protected String nombre = "sin nombre";


	public ObjetoDominio(String nombre){
		this.nombre = nombre;
	}


	public String getNombre()	{
		return nombre;
	}


	@Override
	public String toString() {
		return nombre;
	}
}
```

Veamos ahora la clase Registro. No se trata de una clase de entidad, sino de una clase de utilidad que permite guardar y recuperar cualquier subclase de ObjetoDominio mediante un mapa.

A destacar que:

Para guardar un ObjetoDominio se tiene que invocar al método add() proporcionado dos parámetros: un String con la categoría (Dvd, BluRay, Pelicula, Cliente, etc) a la que pertenece el objeto entidad a guardar y la propia referencia del objeto a guardar.

Para recuperar un ObjetoDominio hay que llamar al método get() proporcionado dos parámetros: un String con la categoría y otro String con el nombre del objeto a recuperar.

Registro.java

```
package comportamiento.strategy.video;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

/**
 * Clase de utilidad que guarda entidades en un mapa
 * organizandolas por categorias. Cada clave del mapa
 * es una categoria, y las entidades se almancenan dentro
 * de una lista para una misma categoria.
 */
public class Registro {

	private static Map<String, ArrayList<ObjetoDominio>> registro;

	static {
		registro =
				new HashMap<String,
ArrayList<ObjetoDominio>>();
	}

	public static void add(String categoria, ObjetoDominio objeto) {
```

```
            if ("".equals(objeto.getNombre().trim())) {
                  throw new RuntimeException("Error: Se ha intentado
guardar " +
                              "en el Registro un objeto nulo o sin
nombre");
            }


            ArrayList<ObjetoDominio> listaObjetos =
registro.get(categoria);
            if (listaObjetos == null) { // La categoria no existe, la
creamos.
                  ArrayList<ObjetoDominio> lista = new
ArrayList<ObjetoDominio>();
                  lista.add(objeto);
                  registro.put(categoria, lista);
            } else {
                  listaObjetos.add(objeto);
            }
      }


      /**
       * Devuelve un ObjetoDominio, según la categoria especificada y
       * el nombre del objeto. Esto implica, que el objeto debe tener
       * nombre antes de haberse guardado en el mapa.
       */
      public static ObjetoDominio get(String categoria, String nombre)
{
            ArrayList<ObjetoDominio> listaObjetos =
registro.get(categoria);
            if (listaObjetos == null) { // La categoria no existe
                  throw new RuntimeException("Error: La categoria '" +
categoria + "' no existe.");
            }
```

```
            for (ObjetoDominio objeto : listaObjetos) {
                  if (objeto.nombre.equals(nombre)) {
                        return objeto;
                  }
            }

            return null;
      }
}
```

Veamos ahora la clase Pelicula.

- Pelicula es la clase de contexto, es decir, la clase que utiliza una u otra estrategia para cambiar su comportamiento dinámicamente.
- La idea de esta clase es que una clase cliente solicita mediante uno de sus tres métodos de factoría estáticos, la creación de un objeto película. En el momento de esta creación, se obtiene una referencia a uno de los tres tipos de precios (Normal, Antigua o Estreno) y tal referencia se almacena en la propia clase Pelicula. Con esto se tiene correctamente configurado el objeto Pelicula.
- Cuando una clase cliente solicite un calculo de importe o de puntos, la clase Pelicula invocará a tales métodos sobre el objeto Precio que se configuró en la creación del objeto Pelicula.

Otras cosas a tener en cuenta:

- Define un atributo String llamado CATEGORIA con el valor “peliculas”, mediante el cual indicará su categoría al registrarse (método persist()) en la clase Registro o materializarse (método get()).
- No permite la instanciación exterior mediante constructor.
- Tiene métodos públicos que permiten cambiar el tipo de una película (cambiar el tipo es en el fondo cambiar la referencia a la que apunta su atributo precio para que apunte a otra de las subclases de Precio).

Pelicula.java

```
package comportamiento.strategy.video;

public class Pelicula extends ObjetoDominio {

	private static final String CATEGORIA = "peliculas";

	private Precio precio;

	// Evitar instanciacion exterior
	private Pelicula(String nombre) {
		super(nombre);
	}

	/*
	 * Metodos de factoria
	 */
	public static Pelicula crearEstreno(String nombre) {
		Pelicula pel = new Pelicula(nombre);
		pel.setEstreno();
		return pel;
	}

	public static Pelicula crearNormal(String nombre) {
		Pelicula pel = new Pelicula(nombre);
		pel.setNormal();
		return pel;
	}

	public static Pelicula crearDesfasada(String nombre) {
```

```
        Pelicula pel = new Pelicula(nombre);
        pel.setAntigua();
        return pel;
    }

    // ********************************************************

    double calcularImportePrestamo(int diasPrestamo) {
        return precio.calcularImportePrestamo(diasPrestamo);
    }

    int getPuntos(int diasPrestamo) {
        return precio.getPuntos(diasPrestamo);
    }

    public void setNormal() {
        precio = Precio.getNormal();
    }

    public void setEstreno() {
        precio = Precio.getEstreno();
    }

    public void setAntigua() {
        precio = Precio.getAntigua();
    }

    public void persist() {
        Registro.add(CATEGORIA, this);
    }
```

```
	public static Pelicula get(String nombre) {
		return (Pelicula) Registro.get(CATEGORIA, nombre);
	}

	@Override
	public String toString() {
		return "Pelicula [nombre=" + nombre + ", precio=" + precio
+ "]";
	}
}
```

Pasemos a ver ahora la jerarquía de entidades formada por SoporteFisico y sus subclases:

SoporteFisico.java

```
package comportamiento.strategy.video;


public abstract class SoporteFisico extends ObjetoDominio {


	protected String numSerie;
	protected Pelicula pelicula;


	public SoporteFisico(String nombre, String numSerie, Pelicula
pelicula) {
		super(nombre);
		this.numSerie = numSerie;
		this.pelicula = pelicula;
	}


	public String getNumSerie() {
		return numSerie;
```

```
	}

	public Pelicula getPelicula() {
		return pelicula;
	}

	public abstract void persist();

}
```

Algunos comentarios:

El parámetro 'nombre', recibido en el constructor, se lo pasa a su clases padre: ObjetoDominio.

En tanto que clase abstracta:

- Contiene atributos comunes para sus subclases (numSerie y Pelicula)
- Define una interfaz común para sus subclases (ambas tienen que implementar el método persist()):

Dvd.java

```
package comportamiento.strategy.video;

public class Dvd extends SoporteFisico {

	private static final String CATEGORIA = "dvds";

	public Dvd(String nombre, String numSerie, Pelicula pelicula) {
		super(nombre, numSerie, pelicula);
	}
```

```
	@Override
	public void persist() {
		Registro.add(CATEGORIA, this);
	}

	public static Dvd get(String nombre) {
		return (Dvd) Registro.get(CATEGORIA, nombre);
	}

	@Override
	public String toString() {
		return "Dvd [nombre=" + nombre + ", numSerie=" + numSerie
+ ", pelicula=" + pelicula
				+ "]";
	}

}
```

BluRay.java

```
package comportamiento.strategy.video;

public class BluRay extends SoporteFisico {

	private static final String CATEGORIA = "blu-rays";

	public BluRay(String nombre, String numSerie, Pelicula pelicula)
{
		super(nombre, numSerie, pelicula);
	}
```

```
	@Override
	public void persist() {
		Registro.add(CATEGORIA, this);
	}

	public static BluRay get(String nombre) {
		return (BluRay) Registro.get(CATEGORIA, nombre);
	}

	@Override
	public String toString() {
		return "BluRay [nombre=" + nombre + ", numSerie=" +
numSerie + ", pelicula=" + pelicula
				+ "]";
	}

}
```

Poco que comentar sobre las subclases anteriores, ya que se limitan a implementar el método abstracto definido en su clase base y a definir su atributo CATEGORIA.

Veamos ahora la clase Alquiler. Esta clase se construye con una referencia a un objeto SoporteFisico y un entero representando los días que se quiere alquilar la película (al principio del ejemplo ya se ha advertido sobre esta simplificación).

Hay que tener en cuenta que mediante la referencia a SoporteFisico, la clase Alquiler tiene acceso a Pelicula, y por tanto, a los métodos públicos de ésta, como getPuntos() y calcularImportePrestamo().

Alquiler.java

```
package comportamiento.strategy.video;
```

```
public class Alquiler extends ObjetoDominio {


	private static final String CATEGORIA = "alquileres";


	private SoporteFisico soporteFisico;

	private int diasPrestamo;


	public Alquiler(String nombre, SoporteFisico soporteFisico, int
diasPrestamo) {

		super(nombre);

		this.soporteFisico = soporteFisico;

		this.diasPrestamo = diasPrestamo;

	}


	public int getPuntos() {

	  return soporteFisico.getPelicula().getPuntos(diasPrestamo);

    }


	public double calcularImportePrestamo() {

	  return
soporteFisico.getPelicula().calcularImportePrestamo(diasPrestamo);

    }


	public int getDiasPrestamo() {

		return diasPrestamo;

	}


	public SoporteFisico getSoporteFisico() {

		return soporteFisico;

	}


	public void persist() {
```

```
            Registro.add(CATEGORIA, this);
      }

      public static Alquiler get(String nombre) {
            return (Alquiler) Registro.get(CATEGORIA, nombre);
      }

      @Override
      public String toString() {
            return "Alquiler [nombre=" + nombre + ", diasPrestamo=" +
diasPrestamo + ", soporteFisico="
                        + soporteFisico + "]";
      }

}
```

Veamos ahora la clase que modela a un cliente.

Esta clase:

- Define una lista de alquileres.
- Dispone de un método público para añadir una nuevo alquiler a la lista de alquileres del cliente.
- Dispone de un método público mediante el que presenta las películas alquiladas por el cliente, lo que debe pagar por cada una y por el total, así como los puntos obtenidos.
- Define varios métodos privados para calcular información agregada a partir de cada objeto Alquiler almacenado en la lista de alquileres.

Cliente.java

```
package comportamiento.strategy.video;
```

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

public class Cliente extends ObjetoDominio {

	private static final String CATEGORIA = "clientes";

	private List<Alquiler> alquileres = new ArrayList<Alquiler>();

	public Cliente(String nombre) {
		super(nombre);
	}

	public String mostrarEstadoCuentas() {
		String head = "Alquiler del cliente " + getNombre() + ":
\n";
		String resultado = "";
		for (Alquiler alquiler : alquileres) {
		resultado += "\t" +
alquiler.getSoporteFisico().getPelicula().getNombre() +
				"\t" +
String.valueOf(alquiler.calcularImportePrestamo()) + "€\n";
		}
	    // Informacion pie factura
		resultado += "A cobrar: " +
String.valueOf(calcularPagoTodosAlquileres()) + "€\n";
		resultado += "Puntos: " +
String.valueOf(getPuntosTodosAlquileres()) + " puntos";
		return head + resultado;
	}

	private double calcularPagoTodosAlquileres() {
		double resultado = 0;
```

```
            for (Alquiler alquiler : alquileres) {
                  resultado += alquiler.calcularImportePrestamo();
            }
            return resultado;
      }

      private int getPuntosTodosAlquileres() {
        int resultado = 0;
        for (Alquiler alquiler : alquileres) {
                  resultado += alquiler.getPuntos();
            }
        return resultado;
    }

      public void nuevoAlquiler(Alquiler alquiler) {
            alquileres.add(alquiler);
      }

      public void persist() {
            Registro.add(CATEGORIA, this);
      }

      public static Cliente get(String nombre) {
            return (Cliente) Registro.get(CATEGORIA, nombre);
      }

      @Override
      public String toString() {
            return "Cliente [nombre=" + nombre + ", alquileres=" +
printAlquileres() + "]";
      }
```

```
	private String printAlquileres() {
		String resultado = "";
		for (Alquiler alquiler : alquileres) {
			resultado += alquiler.toString() + "\n";
		}
		return resultado;
	}
}
```

Por último, veamos una clase cliente que ejercite la aplicación:

MainClient.java

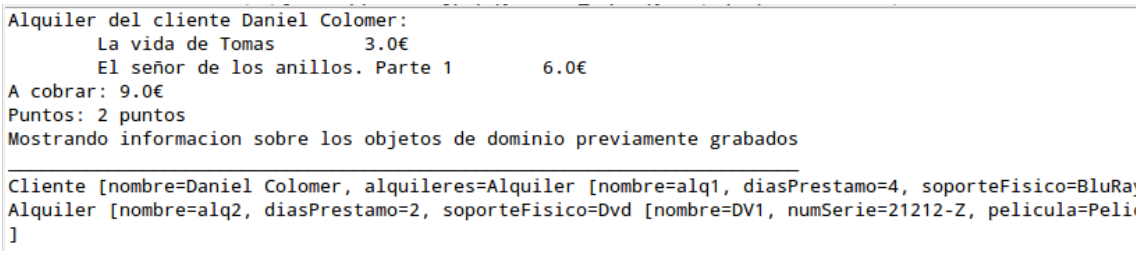
```
package comportamiento.strategy.video;

public class MainClient {

	private static final int[] diasPrestamo = {
		1, 2, 3, 4, 5, 6, 7
	};

	public static void main(String[] args) {

		// Creamos un cliente
		Cliente cliente = new Cliente("Daniel Colomer");
		cliente.persist();

		// Creamos una pelicula de tipo normal
		Pelicula peli = Pelicula.crearNormal("La vida de Tomas");
		/*
```

```
             * Ahora cambiamos su estado para indicar que está
desfasada.
             * Esto tiene implicaciones serias, ya que cambia la
manera
             * en que se calcula su precio de alquiler.
             */
            peli.setAntigua();
            // La grabamos
            peli.persist();

            // Creamos un soporte fisico para esa pelicula
            BluRay bluRay = new BluRay("BR1","11111-X", peli);
            bluRay.persist();

            // Creamos un alquiler de 4 dias para ese soporte fisico
            Alquiler alquiler = new Alquiler("alq1", bluRay,
diasPrestamo[3]);
            alquiler.persist();

            // Creamos un enlace entre el cliente y el alquiler
            cliente.nuevoAlquiler(alquiler);

            // Creamos una pelicula
            Pelicula peli2 = Pelicula.crearEstreno("El señor de los
anillos. Parte 1");
            peli2.persist();

            // Creamos un soporte fisico para esa pelicula
            Dvd dvd = new Dvd("DV1","21212-Z", peli2);
            dvd.persist();

            // Creamos un alquiler de 2 dias para ese soporte fisico
```

```
            Alquiler alquiler2 = new Alquiler("alq2", dvd,
diasPrestamo[1]);
            alquiler2.persist();

            // Creamos un enlace entre el cliente y el alquiler
            cliente.nuevoAlquiler(alquiler2);

            // Mostrar por pantalla lo que se le debe al video club
            String estadoCuentas = cliente.mostrarEstadoCuentas();
            System.out.println(estadoCuentas);

            mostrarObjetosPersistidos();
      }

      private static void mostrarObjetosPersistidos() {
            say("Mostrando informacion sobre los objetos de dominio
previamente grabados");

say("_____
_____");

            /*say(Pelicula.get("La vida de Tomas"));
            say(Pelicula.get("El señor de los anillos. Parte 1"));

            say(BluRay.get("BR1"));
            say(Dvd.get("DV1"));

            say(Alquiler.get("alq1"));
            say(Alquiler.get("alq2"));*/

            say(Cliente.get("Daniel Colomer"));
      }
```

```
	private static void say(Object target) {
		System.out.println(target);
	}
}
```

Salida (parcial horizontalmente):



Problemas específicos e implementación

Colaboraciones

Paso de datos

Strategy y Context interactúan para implementar el algoritmo (estrategia) escogido. Esta interacción puede hacer de varias formas:

La clase de contexto puede pasar todos los datos requeridos por el algoritmo cuando éste es invocado.

La clase de contexto puede pasarse ella misma como un argumento en los servicios de Strategy, lo cual significa que Strategy tiene que acceder de nuevo a la clase de contexto para tomar de ella lo que requiera.

Relación [cliente-contexto] [contexto-strategy]

El contexto recibe peticiones de las clases clientes y las reenvía hacia sus estrategias. Normalmente, las clases cliente crean y pasan estrategias concretas al contexto. A partir de ese momento, las clases clientes interactúan exclusivamente con el contexto.

Subclasificar el contexto

En el último ejemplo visto lo que queríamos eran variaciones en el comportamiento del contexto (que se calculara el precio o los puntos de una u otra manera). Por tanto ¿no podríamos directamente hacer subclases del contexto y ahorrarnos la jerarquía de Strategy?

No es una buena idea subclasificar el contexto, ya que para cada una de esas subclases se mezclaría cada estrategia con el código propio del contexto (romperíamos el principio de responsabilidad única), haciéndolo difícil de entender y de mantener. Por tanto, lo mejor es la propuesta que hace el patrón: encapsular cada algoritmo en una subclase permitiendo variar el algoritmo de manera independiente al contexto.

Eliminación de sentencias condicionales

Las estrategias o algoritmos eliminan las problemáticas sentencias condicionales. Cuando en una clase se observen tales bloques condicionales, seguramente tal clase es firme candidata para refactorizar y aplicar el patrón Strategy, State o Template Method.

Algoritmos con ventajas/inconvenientes

Cuando es requisito el poder elegir entre diferentes implementaciones, hay que pensar que las subclases de Strategy pueden ofrecer el mismo comportamiento con diferente imlementación. De esta manera, el cliente puede elegir entre las ventajas de cada algoritmo, ya que uno puede ser más rápido que otro pero consumir más recursos, etc.

Clase de contexto con estrategia por defecto

Cuando aplica, resulta conveniente que la clase de contexto se configure automáticamente con una estrategia predeterminada, ya que esto evita que las clases cliente del contexto tengan que interactuar con objetos estrategia.

Patrones relacionados

Flyweight: El patrón Strategy a menudo sirve para crear objetos flyweights.

State: El patrón Strategy tiene una arquitectura idéntica al patrón State, sin embargo el propósito de ambos patrones es muy diferente.

Template Method: El patrón Strategy guarda cierta relación con el patrón Template Method. De hecho, Strategy, al ser una forma de polimorfismo y por tanto, uno de los pilares de la Orientación a Objetos, suele guardar relación con muchos patrones.