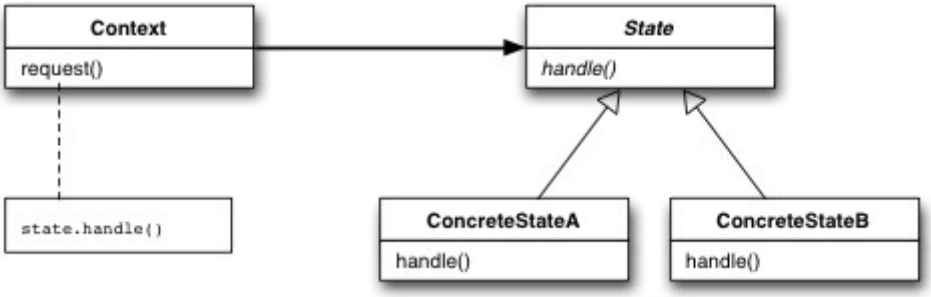
State

Diagrama de clases e interfaces



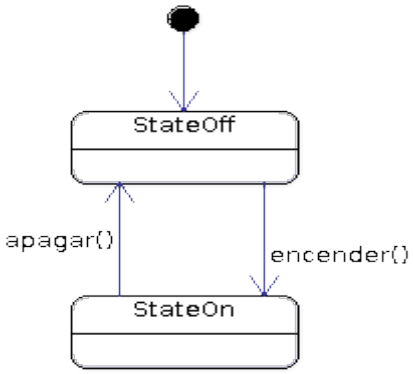
Intención

- El patrón State hace que un objeto cambie su comportamiento al cambiar su estado interno.
- El cambio de comportamiento se lleva a cabo en tiempo de ejecución, dependiendo del estado actual del objeto.

Motivación

Los diagramas de estado utilizan para describir un determinado comportamiento dinámico. La manera “tradicional” de implementar las transiciones entre estados a menudo consiste en complejos bloques condicionales, frecuentemente anidados.

Supongamos que para definir el comportamiento de un interruptor utilizamos el siguiente diagrama de estados:



Un comportamiento como el anterior es fácil de implementar mediante sentencias if-else. Por ejemplo, podríamos tener una clase Interruptor como la siguiente:

Interruptor.java

```
package comportamiento.state.interruptor_sin_patron;

// Objeto de contexto
public class Interruptor {
    private boolean encendido;
    public void conmutar() {
        if (!encendido) {
            encendido = true;
            System.out.println("encendido");
        } else {
            encendido = false;
            System.out.println("apagado");
        }
    }
}
```

Y una clase cliente que cree un interruptor y lo conmute unas cuantas veces:

MainClient.java

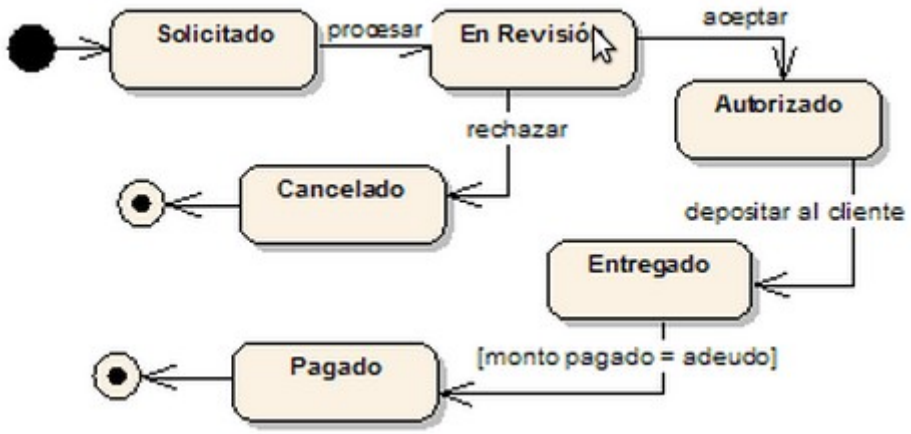
```
package comportamiento.state.interruptor_sin_patron;

public class MainClient {
      public static void main(String args[]) {
            Interruptor interruptor =
                        new Interruptor();
            interruptor.conmutar();
            interruptor.conmutar();
```

```
            interruptor.conmutar();
      }
}
```

Salida:

```
encendido!
apagado!
encendido!
```

Ahora bien, imaginemos algo más complejo, como sería el caso de un diagrama de estados que describa el flujo de aprobación de un préstamos bancario:

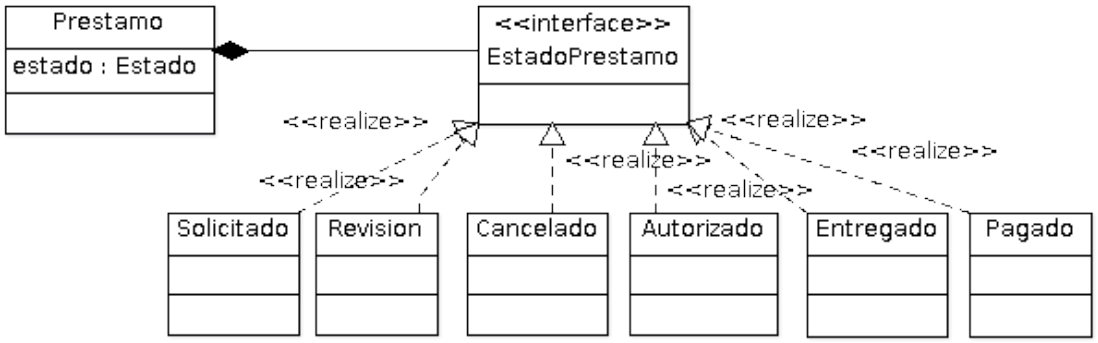


Está claro que una implementación “tradicional” de este diagrama de estados no resultará tan sencilla como la del interruptor. Si diseñamos una clase *Prestamo* en la que sus métodos contengan la lógica de casos necesaria para reflejar las acciones condicionales dependientes del estado, acabaremos por tener diversas sentencias *if-else* o *switch-case*, así como bastantes propiedades booleanas que permitan seguir la pista de lo que toca hacer después de cada paso. Este tipo de implementación produce un código confuso, difícil de entender y de mantener.

El patrón State propone una manera elegante de resolver este problema: crear clases "estado" que implementen una interfaz común, donde cada una implemente la lógica requerida para cubrir cada caso. De esta manera, en lugar de definir en el objeto de contexto *(Interruptor/Prestamo)* las operaciones dependientes del estado, el objeto de

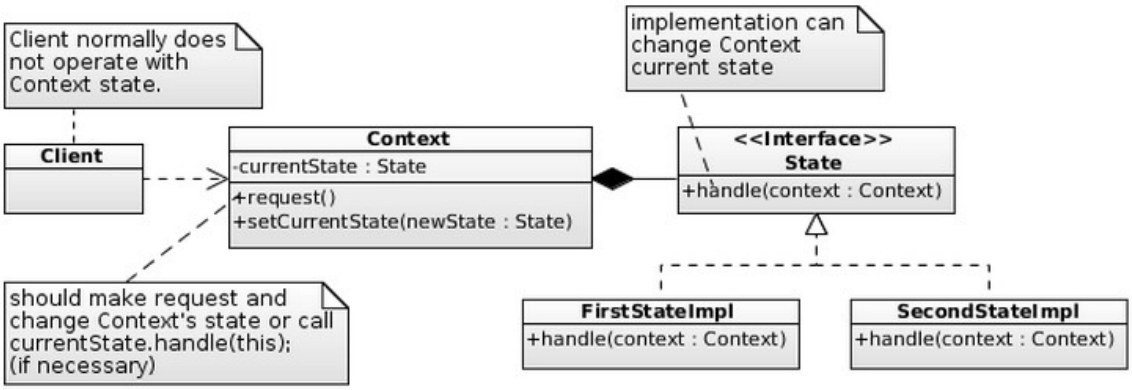
contexto delegará en el objeto de "estado" actual. Lógicamente, para que esto funcione, el objeto de contexto necesitará una referencia a un objeto "estado", la cual siempre tiene que hacer referencia al estado actual.

El siguiente diagrama de clases muestra el diseño para el ejemplo del préstamo bancario:



Implementación

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



Las clases participantes en el patrón son las siguientes:

- **Context:** Define la interfaz con la que interactúan los clientes y mantiene una referencia de State, cuya instancia pertenece a una de las subclases de State, con lo que queda definido el estado actual de Context.
- **State:** Define una interfaz para encapsular el comportamiento asociado con un estado particular de Context.

- **ConcreteState subclasses** (FirstStateImpl, SecondStateImpl): Cada una de estas subclases implementa un comportamiento asociado con el estado de Context.

Aplicabilidad y Ejemplos

El uso del patrón State es adecuado:

- Como se ha comentado, este patrón es adecuado cuando el comportamiento de un objeto depende de su estado. Al localizar cada comportamiento específico en su respectiva subclase, se reduce la complejidad total del sistema. Además, si detectamos nuevo comportamiento, sencillamente tenemos que crear una nueva subclase para encapsularlo.

- A pesar de que el diseño resultante puede comportar un gran número de subclases, éstas suelen ser sencillas de entender y de mantener. Además, siempre es mejor alternativa que tener en un solo objeto con métodos largos y con bloques condiciones dependientes del estado del objeto.

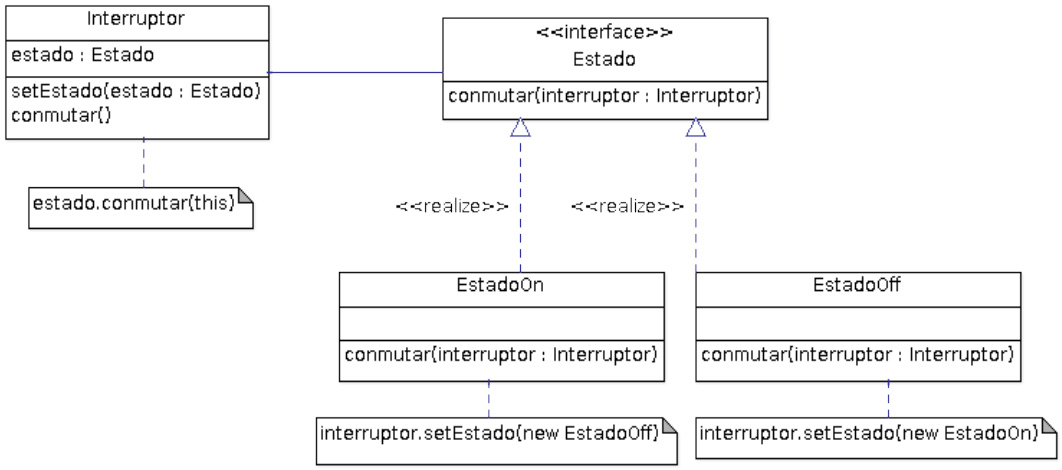
- Por otro lado, al diseñar una jerarquía de clases estado, las transiciones entre estados quedan bien explicitadas, al contrario de lo que sucede cuando un solo objeto define su estado actual en función de sus datos internos (asignando valores a ciertas variables).

- Usualmente las clases estado no tienen estado propio, por lo que pueden ser singletons y compartirse.

Ejemplo 1: Conmutación de un interruptor

Este ejemplo trata de un programa que modela de una manera simplista el comportamiento de un interruptor.

El siguiente diagrama de clases muestra cada componente del ejemplo:



Un interruptor debe presentar un interfaz que permita a las clases cliente conmutar entre sus dos posibles estados (encendido y apagado). Veamos tal interfaz:

Estado.java

```
package comportamiento.state.interruptor;


public interface Estado {
	void conmutar(Interruptor interruptor);
}
```

La interfaz anterior tiene que ser implementada por las clases que representan los posibles estados (encendido y apagado). Irónicamente, estas subclases carecen de estado, por lo que lo más óptimo en términos de rendimiento es que sean singletons:

EstadoOn.java

```
package comportamiento.state.interruptor;


/*
```

```
 * Singleton
 */
public class EstadoOn implements Estado {

	private static EstadoOn estadoOn;

	private EstadoOn() {
	}

	public static Estado getInstance() {
		if (estadoOn == null) {
			estadoOn = new EstadoOn();
		}
		System.out.println("On");
		return estadoOn;
	}

    @Override
    public void conmutar(Interruptor interruptor) {
        // Transicion al siguiente estado.
        // Actualizar el estado del objeto de contexto
        interruptor.setEstado(EstadoOff.getInstance());
    }
}
```

Notad del código anterior, que en el método conmutar() lo que se hace es ocuparse de la transición hacia el siguiente estado posible del interruptor. En este caso de On a Off. Para ello, se utiliza la propia referencia de Interruptor, es decir, que la instancia de Interruptor siempre tiene el estado adecuado.

Tanto la clase Interruptor como la clase EstadoOn y EstadoOff pueden encargarse del orden que debe seguir la secuencia de estados y bajo que circunstancias debe realizar

la transición de estado. En este caso, se ha decido que se haga en las subclases de Estado.

EstadoOff.java

```
package comportamiento.state.interruptor;


/*
 * Singleton
 */
public class EstadoOff implements Estado {

	private static EstadoOff estadoOff;

	private EstadoOff() {
	}

	public static Estado getInstance() {
		if (estadoOff == null) {
			estadoOff = new EstadoOff();
		}
		System.out.println("Off");
		return estadoOff;
	}

    @Override
    public void conmutar(Interruptor interruptor) {
      // Transicion al siguiente estado.
        // Actualizar el estado del objeto de contexto
        interruptor.setEstado(EstadoOn.getInstance());
    }
}
```

Veamos ahora la clase Interruptor.

Interruptor.java

```
package comportamiento.state.interruptor;


/*
 * Objeto de contexto.
 * El comportamiento del interruptor al encenderse y al apagarse
 * se delega en la correspondiente subclase "estado", por lo que
 * aqui queda un código limpio y sencillo
 */
public class Interruptor {

    private Estado estado;

    public Interruptor(Estado estado) {
      this.estado = estado;
    }

    public void setEstado(Estado estado) {
        this.estado = estado;
    }

    public void conmutar() {
        estado.conmutar(this);
    }
}
```

La clase Interruptor sostiene una referencia al estado actual. Cuando se crea el interruptor, es necesario que la clase cliente que lo invoca pase un estado como argumento. Este será el estado inicial.

A destacar de la clase Interruptor:

- Implementa el código que actualiza su atributo “estado”.
- Invoca al método conmutar de la subclase correspondiente de Estado, es decir, delega las peticiones que le llegan de las clases cliente en las subclases de Estado. El interruptor se pasa él mismo como argumento a las subclases de Estado, con lo que tales subclases pueden acceder a los componentes públicos de Interruptor (y actualizar así su estado).

Partiendo de la premisa que un interruptor se crea en estado apagado, la primera vez que se llame a conmutar(), se invocará al método conmutar() de EstadoOff, que como hemos visto, lo único que hace es actualizar el estado del Interruptor con la instancia de EstadoOn.

El ejemplo anterior puede parecer una sobreingeniería. Sin embargo, si lo extrapolamos para un caso en el que necesitemos varias subclases de Estado veremos cuán importante es mantener simple la clase de contexto (Interruptor en el ejemplo mostrado).

Finalmente, veamos una clase cliente:

MainClient.java

```
package comportamiento.state.interruptor;


public class MainClient {
	public static void main(String args[]) {
		Interruptor interruptor =
				new Interruptor(EstadoOff.getInstance());
		interruptor.conmutar();
		interruptor.conmutar();
		interruptor.conmutar();
	}
}
```

Comentarios sobre la clase cliente:

Interruptor es la interfaz primaria para la clase cliente, la cual configura el interruptor con un objeto estado inicial. No obstante, una vez que el interruptor está configurado ya no tiene que tratar de nuevo con el objeto estado.

Salida:

Off

On

Off

On

Ejemplo 2: Flujo de estados de un pedido

Supongamos que una empresa que fabrica y distribuye sus productos nos solicita una aplicación que gestione el ciclo de vida de los pedidos de venta. Una vez que nos reunimos con el responsable de ventas, obtenemos las siguientes reglas de negocio:

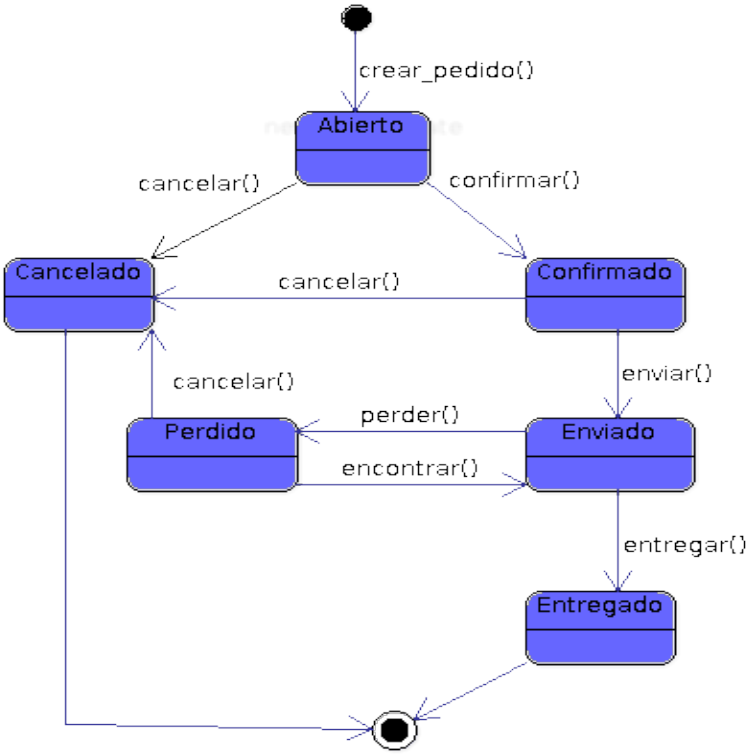
- Cuando se crea un pedido este obtiene el estado de “Abierto”. En este estado, al pedido se añaden tantas líneas de artículo como sean necesarias y se le establece una dirección de envío. De “Abierto” puede pasar a “Confirmado” o a “Cancelado”. Para pasar a “Cancelado” no es necesario ningún requisito, en cambio, para pasar a “Confirmado”, el pedido debe cumplir lo siguiente:
    - Ha contener al menos una línea de artículo, es decir, no puede ser un pedido vacío.
    - Debe tener una dirección de envío o varias.
- Un pedido en estado “Confirmado”, puede pasar a “Cancelado”, sin ningún requisito especial o puede pasar a “Enviado”. “Enviado” significa que el pedido es enviado al cliente mediante un transportista. Para pasar a “Enviado”, el pedido tiene que estar pagado.
- Un pedido en estado “Enviado” puede pasar a estado “Entregado” o a estado “Perdido”. “Entregado” significa que el cliente ha recibido correctamente el

pedido. “Perdido” significa que por algún motivo se ha extraviado el pedido durante la fase de envío. Podemos mantener el pedido en este estado durante un tiempo prudencial, ya que a veces la pérdida es temporal (cosas de las agencias de transporte) y el pedido suele aparecer de nuevo.

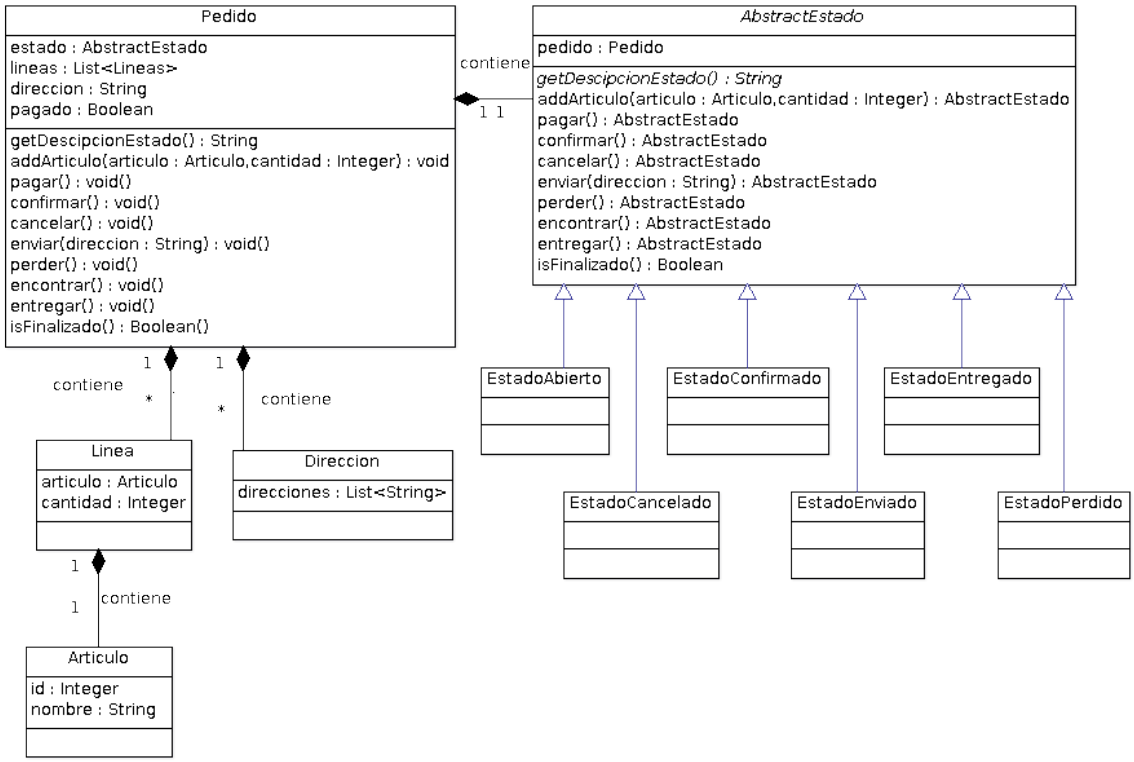
- Un pedido en estado “Perdido” puede pasar a estado “Cancelado”, lo que significa que no se ha podido encontrar, o puede pasar de nuevo a estado “Enviado”, lo que significa que el pedido ha sido encontrado. Hay que notar que no existe ningún estado “Encontrado”, pues el hecho de encontrar un pedido significa que éste pasa de “Perdido” a “Enviado”.

- Cuando un pedido está en estado “Entregado” o en estado “Cancelado”, decimos que el pedido está finalizado, aunque no se trata de ningún estado.

El siguiente diagrama de estados ilustra las reglas de negocio anteriores:



El siguiente diagrama de clases muestras los principales componentes que forman la aplicación de ejemplo:



Del gráfico vemos que:

Un objeto Pedido contiene:

- Un objeto Estado, el cual a su vez requiere un objeto Pedido.
- Una lista de objetos Linea. Un objeto Linea contiene un objeto Articulo.
- Un objeto Direccion

Veamos cada una de estas clases necesarias por Pedido:

Articulo.java

```
package comportamiento.state.pedidos;


public class Articulo {
	private String id;
	private String nombre;
```

```
	public String getId() { return id; }
	public String getNombre() {   return nombre;    }

	public void setId(String id) {
		this.id = id;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
}
```

Ahora veamos la clase Linea, que no es más que una pareja 'articulo-cantidad':

Linea.java

```
package comportamiento.state.pedidos;

public class Linea {
	private Articulo articulo;
	private int cantidad;

	public Linea(Articulo articulo, int cantidad) {
		this.articulo = articulo;
		this.cantidad = cantidad;
	}

	public Articulo getArticulo() {
		return articulo;
	}

	public int getCantidad() {
```

```
            return cantidad;
      }
}
```

Por otro lado, tenemos la clase Direccion:

Direccion.java

```
package comportamiento.state.pedidos;

import java.util.List;

public class Direccion {

      List<String> direcciones;

      public List<String> getDirecciones() {
            return direcciones;
      }

      public void setDirecciones(List<String> direcciones) {
            this.direcciones = direcciones;
      }
}
```

También tenemos una enum que contiene los posible estados de un pedido. De esta manera evitamos usar Strings, que siempre pueden dar lugar a errores tipográficos:

Estados.java

```
package comportamiento.state.pedidos;

public enum Estados {
```

```
        Abierto,
        Cancelado,
        Confirmado,
        Entregado,
        Enviado,
        Perdido
}
```

Veamos ahora la jerarquía de clases de AbstractEstado, comenzando por ella misma.

Cosas a tener en cuenta de su diseño:

- Contiene la referencia al Pedido como atributo privado, por lo que las subclases deberán proporcionar tal referencia en su construcción y accederán a ella vía método getter de AbtractPedido.
- Tiene una implementación para cada uno de los métodos que conforman la interfaz completa de estado, aunque hay que tener en cuenta que la ejecución de cualquiera de estos métodos en esta propia clase significa que no se ha ejecutado el método correspondiente de una subclase, luego se está ejecutando una transición incorrecta y la finalidad de la implementación de estos métodos aquí es lanzar una excepción.
- Disponer en AbtractPedido de los métodos descritos en el párrafo anterior es muy cómodo, ya que si una subclase no requiere alguno de ellos porque no aplica, no debe preocuparse de nada, pues la clase AbtractPedido controlará ese caso.

AbstractEstado.java

```
package comportamiento.state.pedidos;

public abstract class AbstractEstado {

        private static final String
                PRE = "No se puede ",
```

```
		SUF = "un pedido cuyo estado es ",
		ERR_AGREGAR = PRE + "agregar lineas a " + SUF,
		ERR_PAGAR = PRE + "pagar " + SUF,
		ERR_CONFIRMAR = PRE + "confirmar " + SUF,
		ERR_CANCELAR = PRE + "cancelar " + SUF,
		ERR_ENVIAR = PRE + "enviar " + SUF,
		ERR_PERDER = PRE + "pasar a estado PERDIDO " + SUF,
		ERR_ENCONTRAR = PRE + "encontrar " + SUF,
		ERR_ENTREGAR = PRE + "entregar " + SUF;

	private Pedido pedido;

	public AbstractEstado(Pedido pedido) {
		this.pedido = pedido;
	}

	protected abstract Estados getDescipcionEstado();

	public Pedido getPedido() {
		return pedido;
	}

	public AbstractEstado addArticulo(Articulo articulo, int
cantidad) {
		throw new IllegalStateException(ERR_AGREGAR +
getDescipcionEstado());
	}

	public AbstractEstado pagar() {
		throw new IllegalStateException(ERR_PAGAR +
getDescipcionEstado());
	}
```

```
	public AbstractEstado confirmar() {

		throw new IllegalStateException(ERR_CONFIRMAR +
getDescipcionEstado());

	}


	public AbstractEstado cancelar() {

		throw new IllegalStateException(ERR_CANCELAR +
getDescipcionEstado());

	}


	public AbstractEstado enviar(String direccion) {

		throw new IllegalStateException(ERR_ENVIAR +
getDescipcionEstado());

	}


	public AbstractEstado perder() {

		throw new IllegalStateException(ERR_PERDER +
getDescipcionEstado());

	}


	public AbstractEstado encontrar() {

		throw new IllegalStateException(ERR_ENCONTRAR +
getDescipcionEstado());

	}


	public AbstractEstado entregar() {

		throw new IllegalStateException(ERR_ENTREGAR +
getDescipcionEstado());

	}


	public boolean isFinalizado() {

		return false;

	}
```

```
}
```

Veamos ahora cada una de las subclases de estado.

EstadoAbierto.java

```
package comportamiento.state.pedidos;


public class EstadoAbierto extends AbstractEstado {


	public EstadoAbierto(Pedido pedido) {
		super(pedido);
	}


	@Override
	protected Estados getDescipcionEstado() {
		return Estados.Abierto;
	}


	@Override
	public AbstractEstado addArticulo(Articulo item, int cantidad) {
		getPedido().getLineas().add(new Linea(item, cantidad));
		return this;
	}


	@Override
	public AbstractEstado confirmar() {
		if (getPedido().getLineas().isEmpty()) {
			throw new IllegalStateException("Error: Se requiere
que exista al menos una linea para poder confirmar el pedido.");
		} else if (getPedido().getDireccion() == null) {
			throw new IllegalStateException("Error: Se requiere
una direccion para poder confirmar el pedido.");
```

```
            }
            return new EstadoConfirmado(getPedido());
        }

        @Override
        public AbstractEstado cancelar() {
            return new EstadoCancelado(getPedido());
        }
}
```

Sobre la clase EstadoAbierto notad que:

- Sólo se ocupa de redefinir los métodos de AbstractEstado que le son aplicables, ya que el resto son ilegales para ella, por lo que si invoca a alguno de esos métodos, se ejecutará la implementación de clase base, la cual lanza una excepción de estado ilegal.
- El método addArticulo() no requieren ningún tipo de comprobación, ya que según las reglas de negocio, si un pedido se encuentra en estado “Abierto” debe admitir siempre la adición de nuevas líneas.
- El método cancelar() hace que el pedido pase a estado “Cancelado” casi por arte de magia (bien, no hay ninguna magia, sencillamente, como veremos a continuación, EstadoCancelado siempre responde *true* cuando se le pregunta si el pedido está finalizado).
- El método confirmar() requiere comprobar que se cumplen las reglas de negocio establecidas para pasar de estado “Abierto” a “Confirmado”.

EstadoCancelado.java

```
package comportamiento.state.pedidos;

public class EstadoCancelado extends AbstractEstado {
```

```
	public EstadoCancelado(Pedido pedido) {
		super(pedido);
	}

	@Override
	protected Estados getDescipcionEstado() {
		return Estados.Cancelado;
	}

	public boolean isFinalizado() {
		return true;
	}
}
```

No hay mucho que decir sobre la clase EstadoCancelado. Lo único interesante es observar que:

- No redefine ningún método de AbstractEstado, ya que un pedido cancelado no debe permitir nada.
- Siempre responde *true* cuando se le pregunta si el pedido está finalizado.

EstadoConfirmado.java

```
package comportamiento.state.pedidos;

public class EstadoConfirmado extends AbstractEstado {

	public EstadoConfirmado(Pedido pedido) {
		super(pedido);
	}
```

```
	@Override
	protected Estados getDescipcionEstado() {
		return Estados.Confirmado;
	}

	@Override
	public AbstractEstado cancelar() {
		return new EstadoCancelado(getPedido());
	}

	@Override
	public AbstractEstado enviar(String direccion) {
		if (!getPedido().isPagado()) {
			throw new IllegalStateException("Error: No se puede
enviar un pedido que aun no se ha pagado.");
		}
		return new EstadoEnviado(getPedido());
	}

}
```

Comentarios sobre EstadoConfirmado:

- Permite pasar a estado “Cancelado”.
- Permite pasar a estado “Enviado”, siempre que el pedido esté pagado.

EstadoEnviado.java

```
package comportamiento.state.pedidos;
```

```java
public class EstadoEnviado extends AbstractEstado {

	public EstadoEnviado(Pedido pedido) {
		super(pedido);
	}

	@Override
	protected Estados getDescipcionEstado() {
		return Estados.Enviado;
	}

	@Override
	public AbstractEstado perder() {
		return new EstadoPerdido(getPedido());
	}

	@Override
	public AbstractEstado entregar() {
		return new EstadoEntregado(getPedido());
	}
}
```

Poco que comentar de EstadoEnviado, pues ya vemos que permite pasar a estado “Perdido” o a estado “Entregado” sin necesidad de ningún tipo de comprobación.

EstadoPerdido.java

```java
package comportamiento.state.pedidos;

public class EstadoPerdido extends AbstractEstado {
```

```
	public EstadoPerdido(Pedido pedido) {
		super(pedido);
	}

	@Override
	protected Estados getDescipcionEstado() {
		return Estados.Perdido;
	}

	@Override
	public AbstractEstado cancelar() {
		return new EstadoCancelado(getPedido());
	}

	@Override
	public AbstractEstado encontrar() {
		return new EstadoEnviado(getPedido());
	}
}
```

En EstadoPerdido, vemos que:

Permite pasar a “Cancelado”, lo cual es útil cuando veamos que la agencia de transportes no localiza el pedido por ningún sitio.

Permite pasar a “Enviado”, lo cual es más útil aún, cuando la agencia de transportes nos notifica que ha aparecido el pedido y que está de camino al cliente.

EstadoEntregado.java

```
package comportamiento.state.pedidos;
```

```
public class EstadoEntregado extends AbstractEstado {

	public EstadoEntregado(Pedido pedido) {
		super(pedido);
	}

	@Override
	protected Estados getDescipcionEstado() {
		return Estados.Entregado;
	}

	public boolean isFinalizado() {
		return true;
	}
}
```

Sobre EstadoEntregado: igual que en EstadoCancelado, lo único que se le puede hacer a un pedido entregado es preguntarse si se encuentra finalizado, cuya respuesta es afirmativa.

Una vez vistas las clases de la jerarquía de AbstracEstado, pasemos a ver la clase Pedido:

Pedido.java

```
package comportamiento.state.pedidos;

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

/**
 * Context
 */
```

```
public class Pedido {
	private AbstractEstado estadoPedido;
	private List<Linea> lineas;
	private Direccion direccion;
	private boolean pagado;

	public Pedido() {
		estadoPedido = new EstadoAbierto(this);
		lineas = new ArrayList<Linea>();
	}

	public Estados getDescripcionEstado() {
		return estadoPedido.getDescipcionEstado();
	}

	/*
	 * Operaciones delegadas sobre la subclase de estado en curso
	 */
	public void addArticulo(Articulo articulo, int cantidad) {
		estadoPedido = estadoPedido.addArticulo(articulo,
cantidad);
	}

	public void cancelar() {
		estadoPedido = estadoPedido.cancelar();
	}

	public void confirmar() {
		estadoPedido = estadoPedido.confirmar();
	}
```

```
public void perder() {
	estadoPedido = estadoPedido.perder();
}

public void entregar() {
	estadoPedido = estadoPedido.entregar();
}

public void encontrar() {
	estadoPedido = estadoPedido.encontrar();
}

public void pagar() {
	estadoPedido = estadoPedido.pagar();
}

public void enviar(String direccion) {
	estadoPedido = estadoPedido.enviar(direccion);
}

public boolean isFinalizado() {
	return estadoPedido.isFinalizado();
}

/*
 * Getters y setters
 */
public List<Linea> getLineas() {
	return lineas;
}
```

```
	public Direccion getDireccion() {
		return direccion;
	}

	public boolean isPagado() {
		return pagado;
	}

	public void setLineas(List<Linea> lineas) {
		this.lineas = lineas;
	}

	public void setDireccion(Direccion direccion) {
		this.direccion = direccion;
	}

	public void setPagado(boolean pagado) {
		this.pagado = pagado;
	}
}
```

A destacar sobre la clase Pedido:

Tiene como atributos:

Una referencia a AbstractPedido llamada 'estadoPedido', gracias a la cual puede variar su comportamiento dinámicamente en tiempo de ejecución.

Una lista de líneas de pedido.

Una objeto Direccion

Un booleano para saber si el pedido está pagado.

Respecto a los métodos:

Implementa toda una serie de métodos relacionados con las acciones que se pueden realizar sobre un pedido, de hecho son todas las transiciones posibles por las que puede atravesar un pedido. No obstante, la implementación de tales métodos es una simple delegación en el método de la subclase de estado que se encuentre activa en tiempo de ejecución. Lo interesante es ver cómo se logra actualizar automáticamente el estado del pedido (la referencia 'estadoPedido') simplemente habiendo hecho que los métodos en AbstractEstado retornen una referencia de estado, en lugar de *void* o cualquier otro tipo.

Finalmente, veamos una clase cliente:

MainClient.java

```
package comportamiento.state.pedidos;


import java.util.Arrays;


public class MainClient {
	private Pedido pedido;
	private Direccion direccion;
	private Articulo cable_USB;
	private Articulo papel_DIN_A_4;


	public static void main(String args[]) {
		MainClient main = new MainClient();
		main.preparar();
		System.out.println("Prueba de flujo de ejecucion
normal...");
		main.probarFlujoNormal();
		main.reset();
		System.out.println("Prueba de flujo de ejecucion
anomalo...");
		main.probarFlujoAnomalo();
	}
```

```
	public void reset() {
		pedido = new Pedido();
	}

	public void preparar() {
		pedido = new Pedido();
		direccion = new Direccion();
		direccion.setDirecciones(Arrays.asList("Mr S. Claus",
"Northpole 1A", "Arctica"));

		cable_USB = crearArticulo("1230010", "USB Cable (5.0m)");
		papel_DIN_A_4 = crearArticulo("1230030", "A4 paper (500
p)");
	}

	public void probarFlujoNormal() {
		pedido.setDireccion(direccion);
		pedido.addArticulo(cable_USB, 4);
		pedido.addArticulo(papel_DIN_A_4, 20);
		verificarEstado(Estados.Abierto,
pedido.getDescripcionEstado());

		pedido.confirmar();
		verificarEstado(Estados.Confirmado,
pedido.getDescripcionEstado());

		pedido.setPagado(true);

		pedido.enviar("555555X");
		verificarEstado(Estados.Enviado,
pedido.getDescripcionEstado());

		pedido.perder(); // El transportista lo ha perdido
```

```
            verificarEstado(Estados.Perdido,
pedido.getDescripcionEstado());


            // ...


            pedido.encontrar(); // Vuelve al estado de Enviado

            verificarEstado(Estados.Enviado,
pedido.getDescripcionEstado());


            pedido.entregar();

            verificarEstado(Estados.Entregado,
pedido.getDescripcionEstado());


            confirmarCierto(pedido.isFinalizado());

      }


      public void probarFlujoAnomalo() {

            pedido.setDireccion(direccion);

            pedido.setPagado(true);

            pedido.addArticulo(cable_USB, 4);


            pedido.confirmar();

            pedido.enviar("555555X");

            try {

                  pedido.cancelar();

                  throw new RuntimeException("Error! Un pedido enviado
no puede cancelarse.");

            } catch (RuntimeException e) {

                  System.out.println("Excepcion controlada: " + e);

            }

      }


      private Articulo crearArticulo(String id, String nombre) {
```

```
            Articulo item = new Articulo();
            item.setNombre(nombre);
            item.setId(id);
            return item;
      }

      private void confirmarCierto(boolean condicion) {
            if (!condicion) {
                  throw new RuntimeException("Error! La condicion no
se ha cumplido");
            }
      }

      private void verificarEstado(Estados estadoEsperado, Estados
estadoReal) {
            if (!estadoEsperado.equals(estadoReal)) {
                  throw new RuntimeException("Error! Estado esperado: "
+
                              estadoEsperado + " | estado real: " +
estadoReal);
            } else {
                  System.out.println("Ok! Estado esperado: " +
                              estadoEsperado + " | estado real: " +
estadoReal);
            }
      }
}
```

Salida:

```
Prueba de flujo de ejecucion normal...
Ok! Estado esperado: Abierto | estado real: Abierto
Ok! Estado esperado: Confirmado | estado real: Confirmado
Ok! Estado esperado: Enviado | estado real: Enviado
Ok! Estado esperado: Perdido | estado real: Perdido
Ok! Estado esperado: Enviado | estado real: Enviado
Ok! Estado esperado: Entregado | estado real: Entregado
Prueba de flujo de ejecucion anomalo...
Excepcion controlada: java.lang.IllegalStateException: No se puede cancelar un pedido cuyo estado es Enviado
```

Problemas específicos e implementación

Una de las cuestiones interesantes sobre la implementación del patrón State es quién define las transiciones entre estados, ¿ la clase de contexto o las subclases de estado?

Ambas opciones son válidas. Si el criterio para las transiciones puede ser fijado de antemano (es constante), entonces las transiciones pueden implementarse en la clase de contexto.

No obstante, por lo general, es más apropiado y flexible permitir que sean las propias subclases las que especifiquen su sucesor y que lleven a cabo las condiciones que disparan las transiciones hacia otros estados. Descentralizar la lógica del comportamiento transicional hace que resulte sencillo modificar o extender tal comportamiento definiendo nuevas subclases.

El único inconveniente de esta aproximación es que existirá siempre una dependencia entre subclases estado (existirá conocimiento de la existencia de otras clases hermanas), pero esto normalmente no es un problema.

Patrones relacionados

Strategy: El patrón Strategy tiene una construcción muy similar a la del patrón State, sin embargo, la intención de uno y otro es muy diferente. Cada subclase de State representa el comportamiento para un estado determinado de la clase de contexto. En cambio, cada subclase de Strategy representa un algoritmo diferente que puede utilizar la clase de contexto. Se podría decir que State es un caso particular de Strategy.

Flyweight: El patrón Flyweight determina cuándo y cómo pueden compartirse los objetos State.

Singleton: Los objetos State a menudo son Singleton's.