Carrera: Licenciatura en Sistemas

Materia: Orientación a Objetos II

Equipo docente:
Titular: Prof. María Alejandra Vranić alejandravranic@gmail.com
Ayudantes: Prof. Leandro Ríos leandro.rios.unla@gmail.com
Prof. Gustavo Siciliano gussiciliano@gmail.com
Prof. Romina Mansilla romina.e.mansilla@gmail.com

Año: 2018

Patrones de diseño

Patrón State (Estado) - Comportamiento de Objetos

Este patrón permite que un objeto modifique su comportamiento cuando cambia su estado interno. Se tiene que definir una "máquina de estados" donde cada estado va a determinar el comportamiento del objeto. Con lo mencionado podemos concluir en que este patrón puede facilitar el manejo de un objeto a medida que va cambiando de estado durante su vida, y que al cambiar su estado tiene que cambiar su comportamiento, tal vez, incluso en tiempo de ejecución.

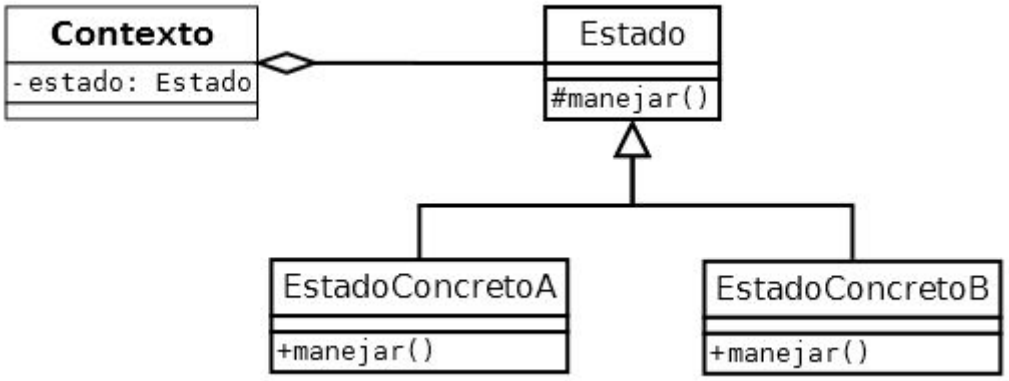
Este patrón tiene los siguientes componentes:

1. Contexto: Es la clase que define la interfaz con el cliente. Tiene una instancia de una subclase State concreta que define su estado actual.
2. State: Define una interfaz para el encapsulamiento de la responsabilidades que deberán ser implementadas en cada Subclase State concreta.
3. Subclase State concreta: Son las clases que tienen el comportamiento de los métodos de la clase State y que terminan definiendo la conducta que va a tener la clase Contexto que lo implemente.

Cuando aplicar este patrón:

1. Si el comportamiento del objeto depende de su estado y según este estado depende el comportamiento en tiempo de ejecución.
2. Los métodos según el estado del objeto ejecutan extensas sentencias condicionales, este patrón permite tratar al estado del objeto como un objeto pleno, independiente de otros objetos.

Estructura:



Conclusiones:

1. Dependiendo del estado del objeto define el comportamiento y lo divide en diferentes estados. Permite añadir nuevas subclases si es necesario definir un nuevo estado y realizar transiciones entre los estados. Utilizando este diseño State las transición del estado del objeto no queda delegada a if o switch, ya que son las subclases que encapsulan el comportamiento según el estado.
2. Especifica el comportamiento del objeto según el estado en forma independiente y marcando la transición del objeto por los estados.
3. El comportamiento de objeto Estado puede ser reutilizado en otro contexto.

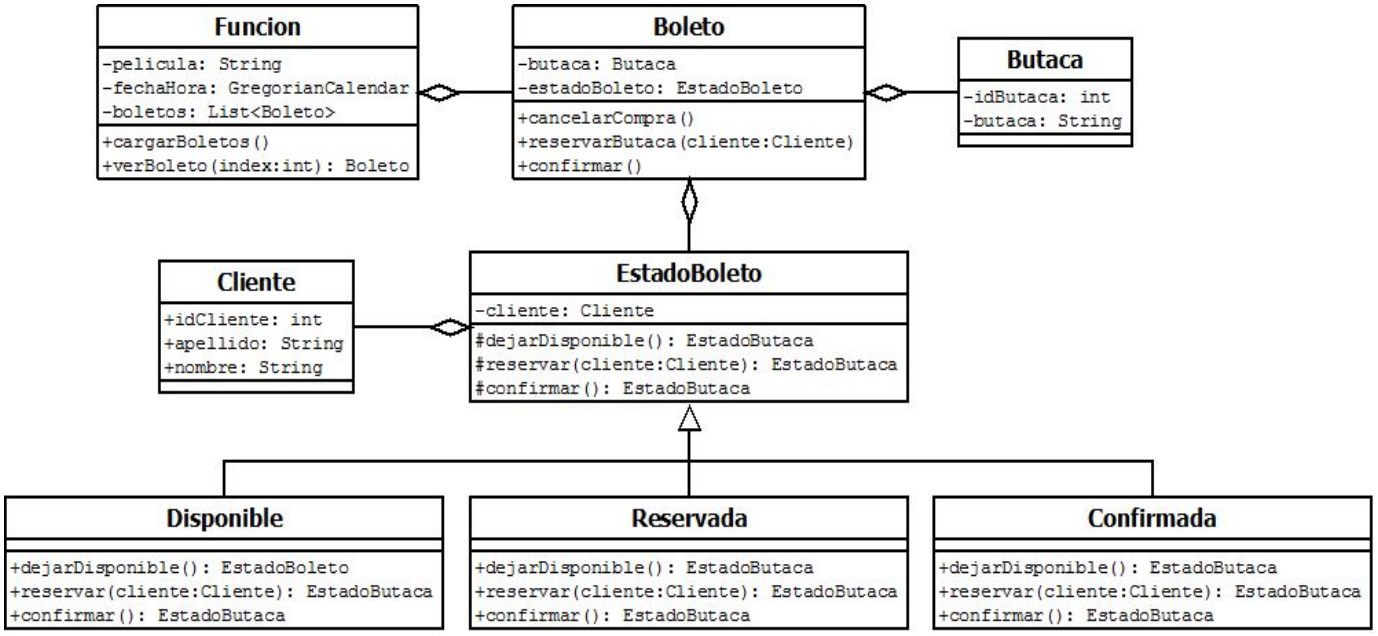
Ejemplo de Boletería de un Cine:

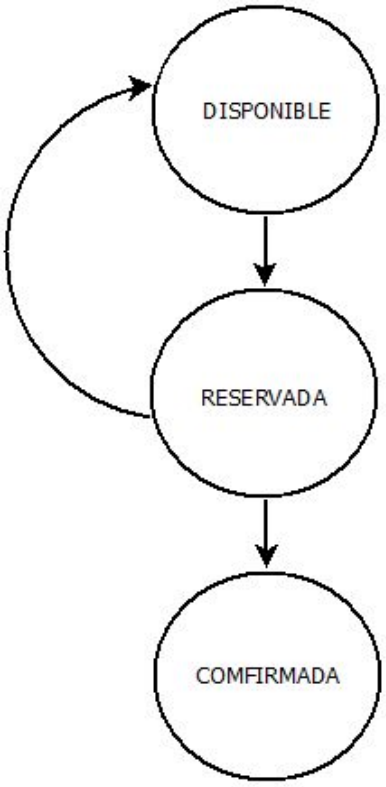
Tenemos que desarrollar un sistema para un cine, se tiene que poder cargar funciones y reservar boletos para ellas por internet, el boleto va a decir la función y la butaca correspondiente. Cada boleto podría pasar por tres momentos o estados:

1- Disponible: Está relacionado con una butaca que aún no se ocupó por ningún Cliente.
2- Reservada: Significa que un Cliente la seleccionó y tiene 3 minutos para cargar su información básica. En este estado aún no se finalizó la compra del boleto, pero al estar reservada otro Cliente no puede intentar reservarla.
3- Confirmada: En este estado el Cliente completó toda la información básica y confirmo la compra. El boleto pasa a ser suyo

Con lo cual un boleto al crearse se inicializa con el estado Disponible, luego sólo podría pasar a Reservada y de este podría pasar a Confirmada o volver a Disponible en caso de que el Cliente cancele la operación.

Estructura:


Funcion
-pelicula: String
-fechaHora: GregorianCalendar
-boletos: List<Boleto>
+cargarBoletos()
+verBoleto(index:int): Boleto
Boleto
-butaca: Butaca
-estadoBoleto: EstadoBoleto
+cancelarCompra()
+reservarButaca(cliente:Cliente)
+confirmar()
Butaca
-idButaca: int
-butaca: String
Cliente
+idCliente: int
+apellido: String
+nombre: String
EstadoBoleto
-cliente: Cliente
#dejarDisponible(): EstadoButaca
#reservar(cliente:Cliente): EstadoButaca
#confirmar(): EstadoButaca
Disponible
+dejarDisponible(): EstadoBoleto
+reservar(cliente:Cliente): EstadoButaca
+confirmar(): EstadoButaca
Reservada
+dejarDisponible(): EstadoButaca
+reservar(cliente:Cliente): EstadoButaca
+confirmar(): EstadoButaca
Confirmada
+dejarDisponible(): EstadoButaca
+reservar(cliente:Cliente): EstadoButaca
+confirmar(): EstadoButaca


DISPONIBLE
RESERVADA
COMFIRMADA

Código Java

Clase Butaca

```
package modelo;

public class Butaca {

	private int idButaca;
	private String butaca;

	public Butaca(int idButaca, String butaca) {
		this.idButaca = idButaca;
		this.butaca = butaca;
	}

	/*---------- GETs y SETs ----------*/
	public int getIdButaca() {
		return idButaca;
	}
	public void setIdButaca(int idButaca) {
		this.idButaca = idButaca;
	}
	public String getButaca() {
		return butaca;
	}
	public void setButaca(String butaca) {
		this.butaca = butaca;
	}
}
```

Clase Funcion

```
package modelo;

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

public class Funcion {

	private String pelicula;
	private GregorianCalendar fechaHora;
	private List<Boleto> boletos;

	public Funcion(String pelicula, GregorianCalendar fechaHora) {
		this.pelicula = pelicula;
		this.fechaHora = fechaHora;
		this.cargarBoletos();
	}

	/*---------- METODOS ----------*/
	protected void cargarBoletos() {
		List<Boleto> lstBoleto = new ArrayList<Boleto>();
```

```
			for(int i=0 ; i<50 ; i++){
				lstBoleto.add(new Boleto(new Butaca((i+1),"Asiento "+(i+1))));
			}
			this.setBoletos(lstBoleto);
		}

		public Boleto verBoleto(int index){
			return this.getBoletos().get(index);
		}

		/*---------- GETs y SETs ----------*/
		public String getPelicula() {
			return pelicula;
		}
		public void setPelicula(String pelicula) {
			this.pelicula = pelicula;
		}
		public GregorianCalendar getFechaHora() {
			return fechaHora;
		}
		public void setFechaHora(GregorianCalendar fechaHora) {
			this.fechaHora = fechaHora;
		}
		public List<Boleto> getBoletos() {
			return boletos;
		}
		public void setBoletos(List<Boleto> boletos) {
			this.boletos = boletos;
		}
}
```

Clase Cliente

```
package modelo;

public class Cliente {

		private int idCliente;
		private String apellido;
		private String nombre;

		public Cliente(String apellido, String nombre) {
			this.apellido = apellido;
			this.nombre = nombre;
		}

		/*---------- GETs Y SETs ----------*/
		public int getIdCliente() {
			return idCliente;
		}
		public String getApellido() {
			return apellido;
		}
		public void setApellido(String apellido) {
```

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

Clase Boleto

```
package modelo;

import state.Disponible;
import state.EstadoBoleto;

public class Boleto {

	private Butaca butaca;
	private EstadoBoleto estado;

	public Boleto(Butaca butaca) {
		this.butaca = butaca;
		this.estado = new Disponible();
	}

	/*---------- METODOS ----------*/
	public void cancelarCompra() {
		this.setEstado(this.getEstado().dejarDisponible());
	}

	public void reservarButaca(Cliente cliente) {
		this.setEstado(this.getEstado().reservar(cliente));
	}

	public void confirmar() {
		this.setEstado(this.getEstado().confirmar());
	}

	/*--------- GETs y SETs ---------*/
	public Butaca getButaca() {
		return butaca;
	}
	public void setButaca(Butaca butaca) {
		this.butaca = butaca;
	}
	public EstadoBoleto getEstado() {
		return estado;
	}
	public void setEstado(EstadoBoleto estado) {
		this.estado = estado;
	}
}
```

Clase EstadoBoleto

```
package state;
import modelo.Cliente;

public abstract class EstadoBoleto {

	private Cliente cliente;

	public abstract EstadoBoleto dejarDisponible();

	public abstract EstadoBoleto reservar(Cliente cliente);

	public abstract EstadoBoleto confirmar();

	public Cliente getCliente() {
		return cliente;
	}
	public void setCliente(Cliente cliente) {
		this.cliente = cliente;
	}
}
```

Clase Disponible

```
package state;

import modelo.Cliente;

public class Disponible extends EstadoBoleto{

	@Override
	public EstadoBoleto dejarDisponible() {
		try {
			throw new Exception("Error, esta butaca ya está disponible");
		} catch (Exception e) {
			e.printStackTrace();
		}
		return this;
	}
	@Override
	public EstadoBoleto reservar(Cliente cliente) {
		Reservada ocupada = new Reservada();
		ocupada.setCliente(cliente);
		return ocupada;
	}
	@Override
	public EstadoBoleto confirmar() {
		try {
			throw new Exception("Error, no tiene una butaca seleccioanda");
		} catch (Exception e) {
			e.printStackTrace();
		}
		return this;
	}
```

```
}
```

Clase Reserva

```
package state;

import modelo.Cliente;

public class Reservada extends EstadoBoleto{

	@Override
	public EstadoBoleto dejarDisponible() {
		return new Disponible();
	}

	@Override
	public EstadoBoleto reservar(Cliente cliente) {
		try {
			throw new Exception("Error, esta butaca ya está reservada");
		} catch (Exception e) {
			e.printStackTrace();
		}
		return this;
	}

	@Override
	public EstadoBoleto confirmar() {
		return new Confirmada();
	}
}
```

Clase Confirmada

```
package state;

import modelo.Cliente;

public class Confirmada extends EstadoBoleto{

	@Override
	public EstadoBoleto dejarDisponible() {
		try {
			throw new Exception("Error, esta butaca ya está confirmada");
		} catch (Exception e) {
			e.printStackTrace();
		}
		return this;
	}

	@Override
	public EstadoBoleto reservar(Cliente cliente) {
		try {
			throw new Exception("Error, esta butaca ya está confirmada");
```

```
			} catch (Exception e) {
				e.printStackTrace();
			}
			return this;
		}

		@Override
		public EstadoBoleto confirmar() {
			try {
				throw new Exception("Error, esta butaca ya está confirmada");
			} catch (Exception e) {
				e.printStackTrace();
			}
			return this;
		}
}
```

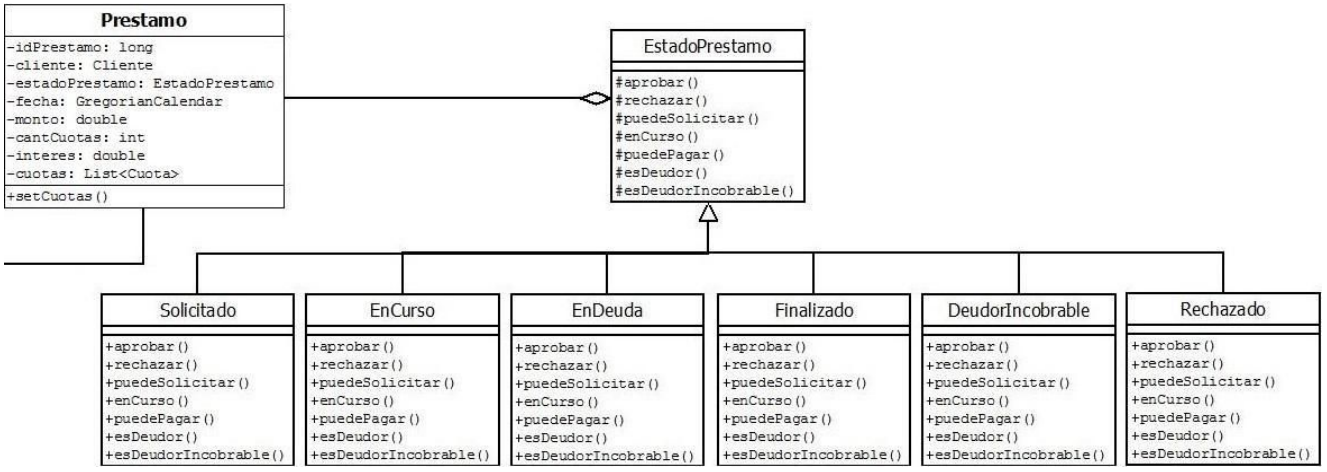
Clase Main

```
package test;

import java.util.GregorianCalendar;

import modelo.Boleto;
import modelo.Cliente;
import modelo.Funcion;

public class Main {

	public static void main(String[] args) {

			Cliente cliente = new Cliente("Siciliano","Gustavo Hernan");
			Funcion funcion = new Funcion("La teoría del todo", new
GregorianCalendar(5, 2, 2018));

			Boleto boleto10 = funcion.verBoleto(10);
			boleto10.reservarButaca(cliente);
			boleto10.confirmar();

			Boleto boleto11 = funcion.verBoleto(11);
			boleto11.reservarButaca(cliente);
			boleto11.cancelarCompra();

			Boleto boleto12 = funcion.verBoleto(12);
			boleto12.confirmar(); //Qué pasa?
			boleto12.cancelarCompra(); //Qué pasa?
			boleto12.reservarButaca(cliente); //Qué pasa?
			boleto12.reservarButaca(cliente); //Qué pasa?
			boleto12.cancelarCompra(); //Qué pasa?

	}

}
```

Ejemplo con el sistema francés de préstamos:

Supongamos para realizar acciones sobre un préstamo existen ciertas condiciones, por ejemplo, un cliente no puede solicitar un nuevo préstamo si tiene un préstamo como deudor incobrable, o uno como deudor y uno en curso, o dos en curso, o también si tiene tres rechazados (ya que se lleva a cabo un análisis en el área de préstamos bancarios). Además existe la validación en el ciclo de vida de préstamo que va de “Solicitado” a “En Curso” o “Rechazado”, de “En Curso” a “EnDeuda” o “Finalizado”, de “En Deuda” a “Finalizado” o “DeudorIncobrable” y de “Rechazado” a “Solicitado” en caso de pedir una revisión. Estos casos nos sugieren una especie de configuración, sobre el comportamiento del préstamo, que varía según su estado. Entonces en este caso es conveniente armar la configuración en una “máquina de estados”, la cual nos va permitir restringir el comportamiento del préstamo. Vamos a adaptar el diagrama visto en Composite para agregar esta “máquina”.

El diagrama quedaría así:



Preguntas:

1- ¿Si no conocieras este patrón como resolverías este problema?
2- ¿Porque el Patrón State es de comportamiento?
3- ¿Cuál es la diferencia entre las subclases del patrón State?
4- ¿El comportamiento del estado del objeto cambia en tiempo de implementación o de ejecución?

Referencias utilizadas:

Título: Patrones de Diseño

Autores: Erich Gamma - Richard Helm - Ralph Johnson -Johon Vlissides

Editorial: Pearson Addison Wesley

SourceMaking https://sourcemaking.com/design_patterns