

GOI ESKOLA
POLITEKNIKO
A
ESCUELA
POLITÉCNICA
SUPERIOR



PATRON DE DISEÑO STATE

GRADO EN INGENIERÍA INFORMÁTICA

AUTORA:

ANE SAJERAS

5 de junio de 2019

RESUMEN

Principalmente el objetivo de este trabajo es definir de manera clara y precisa las funcionalidades de State y ver como este afecta en el sistema. Para ello, en este documento se describe diferente información sobre el patrón de diseño State como es el caso del análisis, diseño y código.

ÍNDICE

1. INTRODUCCIÓN	1
2. PATRON DE DISEÑO STATE	1
3. APLICABILIDAD	1
4. ESTRUCTURA Y PARTICIPANTES.....	2
5. CONSECUENCIAS.....	2
6. IMPLEMENTACIÓN	2
7. CASOS DE USO Y ESCENARIOS	4
8. DIAGRAMA DE ACTIVIDAD.....	5
9. IMPLEMENTACIÓN DEL PATRON DE DISEÑO STATE	6
10. DIAGRAMA DE SECUENCIA	9
11. DIAGRAMA DE CLASES	10
12. VENTAJAS Y DESVENTAJAS	11
13. CONCLUSIÓN.....	11

1. INTRODUCCIÓN

El diseño de aplicaciones software es una de las actividades en las que aun predomina el arte sobre el método. La calidad de diseño de la interacción de los objetos y la asignación de responsabilidades presenta gran variación: las decisiones poco acertadas dan lugar a sistemas y componentes frágiles y difíciles de mantener, entender, reutilizar o extender. Por ello, los patrones de diseño indican los principios a seguir para un diseño hábil. Se pueden considerar como un amplio repertorio de principios generales.

Los patrones de diseño son soluciones para problemas típicos y recurrentes que nos podemos encontrar a la hora de desarrollar una aplicación. Para que una solución sea considerada un patrón debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Además, debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias. Dependiendo de su finalidad pueden ser:

- **Patrones creacionales:** solucionan problemas de creación de instancias y nos ayudan a encapsular y abstraer dicha creación.
- **Patrones estructurales:** solucionan problemas de composición (agregación) de clases y objetos.
- **Patrones de comportamiento:** ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos, así como los algoritmos que encapsulan.

En este caso hablaremos de los patrones de comportamiento, en concreto de State que permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Es decir, parecerá como si el objeto hubiese cambiado sus clases.

2. PATRON DE DISEÑO STATE

El patrón State no indica exactamente dónde definir las transiciones de un estado a otro. Existen dos formas de solucionar esto: Una es definiendo estas transiciones dentro de la clase *Contexto*, la otra es definiendo estas transiciones en las subclases de *Estado*. Es más conveniente utilizar la primera solución cuando el criterio a aplicar es fijo, es decir, no se modificará. En cambio, la segunda resulta conveniente cuando este criterio es dinámico, el inconveniente aquí se presenta en la dependencia de código entre las subclases.

La implementación de este patrón estará en la gestión de la tarjeta para la Asociación Afro situada en Vitoria-Gasteiz, donde los diferentes estados de la tarjeta estarán representados con diversos colores.

3. APLICABILIDAD

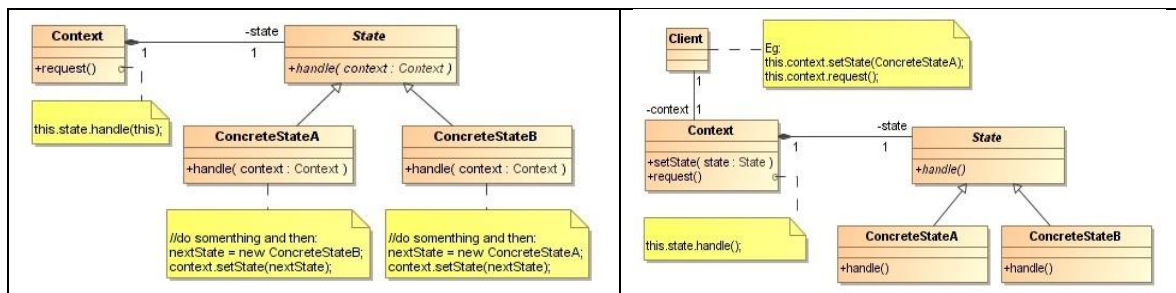
El patrón State se usa en cualquiera de estos casos:

- Cuando el comportamiento de un objeto depende de su estado y debe cambiar su comportamiento en tiempo de ejecución dependiendo de ese estado.

- Las operaciones tienen muchos condicionales que dependen del estado del objeto. Este estado está representado normalmente por uno o más enumerados. A menudo, varias operaciones contendrán la misma estructura condicional. El patrón State pone cada rama del condicional en una clase separada. Esto permite tratar el estado del objeto que puede variar independientemente de otros objetos.

4. ESTRUCTURA Y PARTICIPANTES

La estructura del patrón de diseño State se puede definir de dos formas diferentes, como puede apreciarse en las imágenes inferiores.



Dónde los participantes de dicha estructura son los siguientes: Contexto, State y ConcreteState. **Contexto** define la interfaz de interés para los clientes. La clase **State** por su parte define una interfaz para encapsular el comportamiento asociado con un estado particular del contexto. Por último, cada clase **ConcreteState** implementa un comportamiento asociado con un estado el contexto.




5. CONSECUENCIAS

- Se localizan fácilmente las responsabilidades de los estados específicos. Esto facilita la ampliación de estados.
- Hace los cambios de estado explícitos puesto que en otros tipos de implementación los estados se cambian modificando valores en variables
- Evita la utilización de estructuras condicionales.
- Se incrementa el número de subclases.
- Impone una estructura sobre el código y hace más clara su intención.
- Hace explícitas las transiciones entre cuándo se tiene que ejecutar un comportamiento u otro.
- Los objetos State pueden ser compartidos por varios contextos.
- Permite a un objeto cambiar de clase en tiempo de ejecución.

6. IMPLEMENTACIÓN

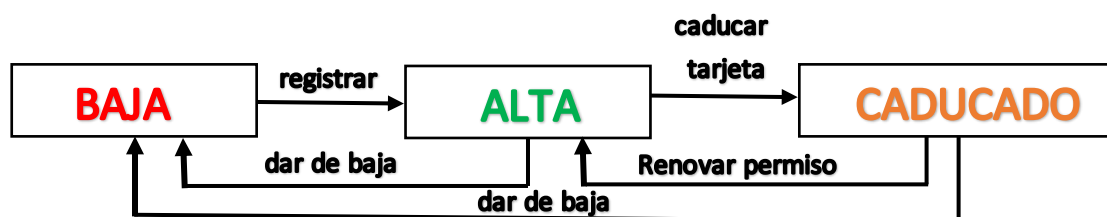
El patrón de diseño State nos puede ser de gran utilidad en los casos que por ejemplo una entidad tenga asociado un grafo de estados con transiciones permitidas y no permitidas entre algunos estados. En función del estado, sus datos y la transición la entidad puede comportarse de forma diferente.

En este caso cuando el trabajador o voluntario de la asociación busca información del estado de las tarjetas o quiere crear una nueva, en la pantalla de la aplicación aparecerá toda la información referente a las tarjetas. Para ello, cada estado tendrá un diferente color que nos dará la información de la situación de las tarjetas. Por ejemplo, supongamos que tenemos una entidad Tarjeta que a lo largo de su vida en la aplicación pasa por estos diferentes estados:

COLOR	SITUACIÓN	DESCRIPCIÓN
	Dar de alta	La familia a la que la ONG ofrece ayuda tiene asignada una tarjeta.
	Dar de baja	La tarjeta está disponible para asignársela a otra familia.
	Fecha de caducidad	Indica que la tarjeta está caducada, ha llegado a la fecha límite.

Y tiene diferentes transiciones como:

- **Registrar:** La familia se ha registrado en la ONG por lo que se le ha asignado una tarjeta. Debido a ello, la tarjeta pasa de estar del estado rojo al estado verde.
- **Caducar tarjeta:** La tarjeta ha llegado a la fecha límite/ máxima por lo que el estado de la tarjeta pasa de estar verde a estar naranja.
- **Dar de baja:** La tarjeta ha sido dada de baja, debido a que la familia ha dejado de pertenecer a la ONG, por lo que el estado pasa de estar verde o naranja a estar rojo.
- **Renovar permiso:** La tarjeta que tiene asignada la familia ha llegado fecha límite/ máxima por lo que tiene que renovar el permiso, por eso el estado pasa de estar naranja a estar verde.



En caso de diseñar este flujo de estados sin el patrón State probablemente acabaríamos con una clase con un montón de condiciones y métodos de bastantes líneas sin una organización clara a simple vista. Para evitar esta situación, aplicaremos el patrón State a este pequeño flujo de estados. En cuanto a código este patrón se basa en dos ideas:

- Cada una de estas clases contendrá un método por cada posible transición.
- Cada estado será representado en una clase.

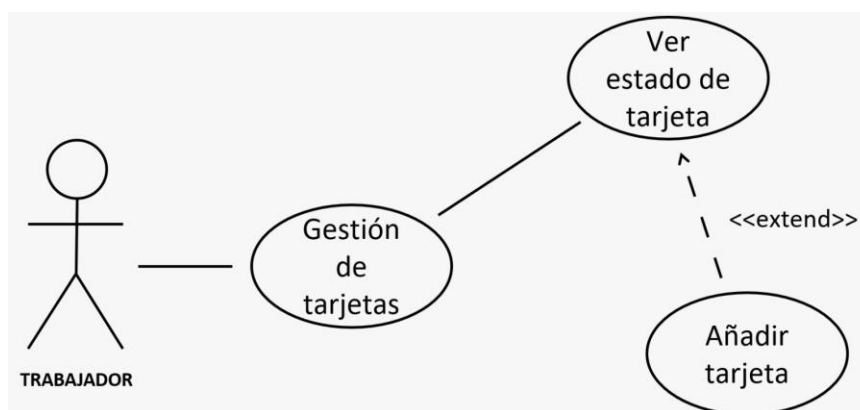
Teniendo en cuenta el diagrama superior un fácil ejemplo del funcionamiento de este sistema sería el siguiente:

SITUACIONES	ESTADOS DE LA TARJETA	
	INICIAL	FINAL
1. La familia se da de alta en la ONG, por lo que se le asigna una tarjeta.		
2. Ha pasado cierto tiempo y la tarjeta de la familia se ha caducado		
3. La familia renueva permisos		
4. La familia deja de pertenecer a la ONG		

De esta manera dependiendo del estado en el que se encuentre la tarjeta, el identificador de esta tendrá diferente color. Con este método se intenta facilitar el trabajo de los empleados de la ONG, debido a que resultará más fácil e intuitiva la aplicación.

7. CASOS DE USO Y ESCENARIOS

Los diagramas de casos de uso documentan el comportamiento de un sistema desde el punto de vista del usuario. Por lo tanto, los casos de uso determinan los requisitos funcionales del sistema, es decir, representan las funciones que un sistema puede ejecutar. Su ventaja principal es la facilidad para interpretarlos, lo que hace que sean especialmente útiles en la comunicación con el cliente. El caso de uso de nuestro sistema sería el siguiente:



En este caso de uso, el trabajador realiza la gestión de tarjetas. También ve el estado de la tarjeta y entonces también puede añadir diferentes tarjetas.

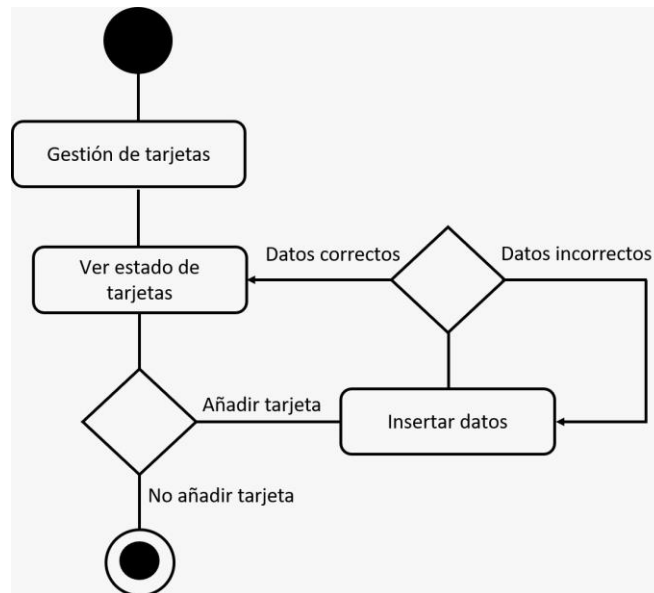
Por su parte el escenario es una secuencia específica de acciones que ilustra un posible comportamiento dentro del caso de uso y representa una instancia de in caso de uso. Un posible escenario de nuestro sistema sería el siguiente:

Código	CU 1.1	
Nombre	VER ESTADO DE TARJETA	
Descripción	Tanto el trabajador como el administrador ven el estado de la tarjeta	
Actores	Trabajador y administrador	
Precondición	Para ver el estado de la tarjeta esta tiene que existir	
Flujo normal		Flujo alternativo
1. Selecciona la tarjeta 2. Extiende caso de uso CU1.2- Añadir tarjeta		1.1. Tarjeta no valida. El sistema muestra mensaje.
Postcondiciones	Tanto el trabajador como el administrador han podido ver el estado de dicha tarjeta	

Código	CU 1.2	
Nombre	AÑADIR TARJETA	
Descripción	Tanto el trabajador como el administrador van a poder añadir nuevas/diferentes tarjetas	
Actores	Trabajador y administrador	
Precondición	Para poder añadir nuevas tarjetas, los datos de los usuarios tienen que estar registrados con anterioridad y tienen que existir los datos	
Flujo normal		Flujo alternativo
1. Introducir datos personales de la familia 2. Introducir número de tarjeta 3. Aceptar la operación		1.1. El número de tarjeta está en uso (está asignado a otra familia). El sistema muestra un mensaje. 3.1. El usuario ha cancelado la operación. Deberá volver a ingresar todos los datos de la familia.
Postcondiciones	Tanto el trabajador como el administrador han añadido nuevas tarjetas	

8. DIAGRAMA DE ACTIVIDAD

Los diagramas de actividad por su parte sirven para representar el comportamiento dinámico de un sistema. Para ello hacen hincapié en la secuencia de actividades que se llevan a cabo y las condiciones que disparan esas actividades. La implementación de dicho diagrama permite describir como un sistema implementa su funcionalidad como se puede apreciar en el siguiente diagrama de actividad:



9. IMPLEMENTACIÓN DEL PATRON DE DISEÑO STATE

Teniendo en cuenta los diferentes diagramas desarrollados en apartados anteriores, se ha implementado un código utilizando el patrón de diseño State, dónde dependiendo del estado en el que se encuentra la tarjeta esta aparecerá en diferentes colores. El código utilizado ha sido el siguiente:

1. Se ha implementado la clase **Tarjeta** con todos sus atributos que se utilizaran como es el id, descripción etc.

```

package modelos;

import state.EstadoRojo;

public class Tarjeta {

    int id;
    String tipo;
    String descripcion;
    EstadoTarjeta objEstadoTarjeta;
    public Tarjeta(int id, String tipo) {
        this.id=id;
        this.tipo=tipo;
        objEstadoTarjeta = new EstadoVerde();
    }

    public Tarjeta(int id, String tipo, String descripcion) {
        this.id=id;
        this.tipo=tipo;
        this.descripcion = descripcion;
        objEstadoTarjeta = new EstadoVerde();
    }

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getTipo() {
        return tipo;
    }
    public void setTipo(String tipo) {
        this.tipo = tipo;
    }
    public String getDescripcion() {
        return descripcion;
    }
    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }
    public Tarjeta() {
        this.objEstadoTarjeta = new EstadoRojo(this);
    }
    public void setEstado (EstadoTarjeta objEstadoTarjeta) {
        this.objEstadoTarjeta = objEstadoTarjeta;
    }

    public String mostrarAviso() {
        return this.objEstadoTarjeta.mostrar();
    }
}

```

2. Se ha creado la clase abstracta **EstadoTarjeta** que más adelante se utilizará en las clases **EstadoRojo**, **EstadoVerde** y **EstadoNaranja** que se explicaran más adelante.

```

package state;

import modelos.Tarjeta;

public abstract class EstadoTarjeta {

    protected Tarjeta objTarjeta;
    public abstract String mostrar();
}

```

3. Se han creado las clases **EstadoRojo**, **EstadoVerde** y **EstadoNaranja** donde se muestran los diferentes estados en los que la tarjeta se puede encontrar. Además, las siguientes clases extienden de la clase abstracta llamada **EstadoTarjeta**.

ESTADO VERDE	<pre> package state; import modelos.Tarjeta; public class EstadoVerde extends EstadoTarjeta { String estado = "VERDE"; public EstadoVerde() { } @Override public String mostrar() { return estado; } } </pre>	<ul style="list-style-type: none"> • Cuando se da de alta una nueva familia o una familia que haya pertenecido con anterioridad a la ONG.
ESTADO NARANJA	<pre> package state; import modelos.Tarjeta; public class EstadoNaranja extends EstadoTarjeta { String estado = "NARANJA"; public EstadoNaranja(Tarjeta objTarjeta) { this.objTarjeta = objTarjeta; } @Override public String mostrar() { return estado; } } </pre>	<ul style="list-style-type: none"> • Cuando la tarjeta de la familia se ha caducado.
ESTADO ROJO	<pre> package state; import modelos.Tarjeta; public class EstadoRojo extends EstadoTarjeta { String estado = "ROJO"; public EstadoRojo(Tarjeta tarjeta) { this.objTarjeta = tarjeta; } @Override public String mostrar() { return estado; } } </pre>	<ul style="list-style-type: none"> • La familia se ha dado de baja. • La familia ha dejado de pertenecer a la ONG

4. Por último, se ha implementado la clase **RendererTarjeta** con el fin de asignarle a cada estado un diferente color. Para todo ello se ha utilizado el patrón de diseño State. Esta clase extiende de JLabel e implementa ListCellRenderer<Tarjeta>.

```

package modelos;

import java.awt.Color;

public class RendererTarjeta extends JLabel implements ListCellRenderer<Tarjeta> {
    private final Color COLOR_VERDE = new Color(34, 177, 76);
    private final Color COLOR_ROJO = new Color(255, 0, 0);
    private final Color COLOR_NARANJA = new Color(255, 164, 32);
    private final int TAMANOLETRA = 16;
    private final int DELGADEZ = 1;
    Beneficiario beneficiario;

    @Override
    public Component getListCellRendererComponent(JList<? extends Tarjeta> list, Tarjeta t, int index,
        boolean isSelected, boolean cellHasFocus) {
        Border border = BorderFactory.createLineBorder(Color.BLUE, DELGADEZ);
        Border borderNormal = BorderFactory.createBevelBorder(BevelBorder.RAISED);
        setFont(new Font("Arial", Font.BOLD, TAMANOLETRA));

        if(t.mostrarAviso().equals("VERDE"))
            setForeground(COLOR_VERDE);

        if(t.mostrarAviso().equals("ROJO"))
            setForeground(COLOR_ROJO);

        if((t.mostrarAviso().equals("NARANJA")) && (beneficiario.getfCaducidad().equals(beneficiario.getFechaActual())))
            setForeground(COLOR_NARANJA);

        if (isSelected && cellHasFocus) {
            setBorder(border);
        } else {
            setBorder(borderNormal);
        }

        this.setText(String.valueOf(t.getId()));
        this.setOpaque(true);

        return this;
    }
}

```

EXPLICACIÓN GENERAL

- Al inicio del programa se crea un objeto llamado **Tarjeta** (que al ser instanciado quedará por defecto en **EstadoVerde**) y mostramos el aviso visual correspondiente. Posteriormente cambiaremos otras dos veces su estado.
- De este modo se mostrará un aviso u otro dependiendo del estado de **Tarjeta** en un momento dado. Además, el número de tarjeta estará de diferente color, dependiendo del estado en el que se encuentre. El estado rojo será que la tarjeta está disponible para asignársela a otra familia, el estado verde por su parte indica que la esa tarjeta ya está asignada a una familia, mientras que el estado naranja indicará que la tarjeta está caducada, por lo que se tendrá que renovar los permisos correspondientes.
- Observa que en la propiedad **objEstadoTarjeta** de **Tarjeta** queda registrado el estado de este en todo momento.
- A la hora de cambiar el estado de un objeto de tipo **Tarjeta**, a la instancia de estado que le pasamos como parámetro le agregamos una referencia a dicho objeto, quedando registrada en la propiedad **objEstadoTarjeta** de ésta, de forma que en caso necesario pueda acceder a su contenedor.

10. DIAGRAMA DE SECUENCIA

Los diagramas de secuencia son un tipo de diagrama utilizado para modelar interacción entre objetos en un sistema según UML. Además, destacan el orden temporal de los mensajes.

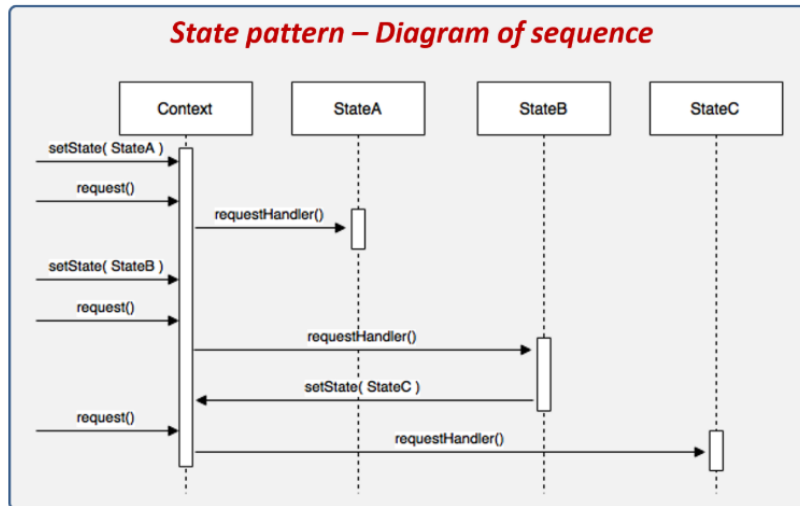
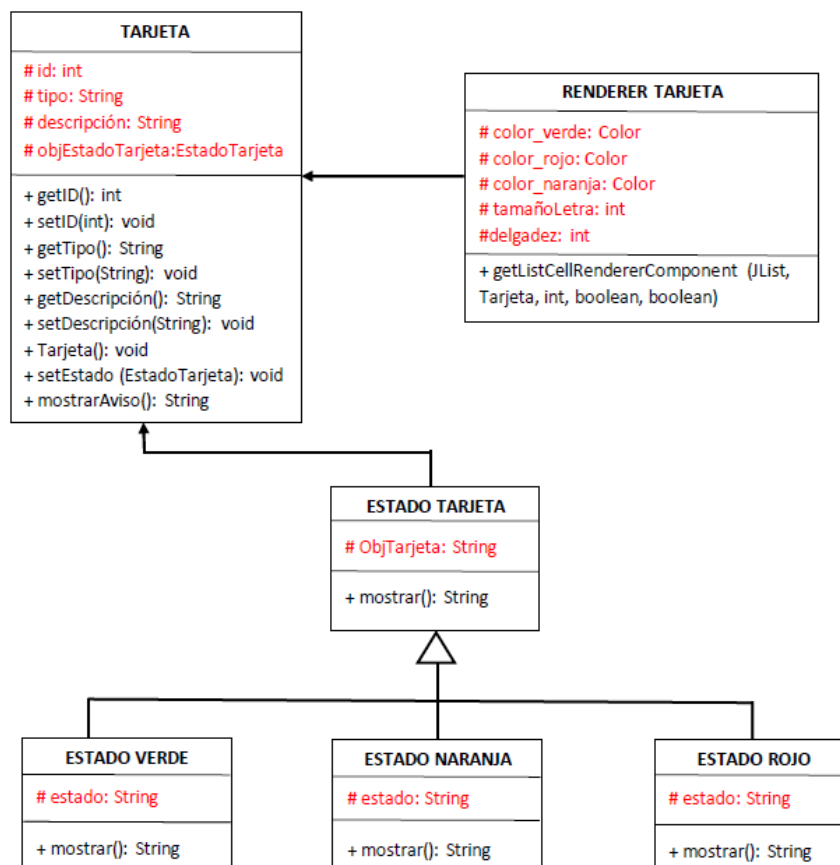


Diagrama de secuencia del patrón de diseño State.

11. DIAGRAMA DE CLASES

El diagrama de clases se utiliza para describir la vista estática de un sistema mostrando las clases del sistema, sus atributos, operaciones (o métodos) y las relaciones entre los objetos.



12. VENTAJAS Y DESVENTAJAS

Las ventajas del método State son las siguientes:

- Localiza el comportamiento dependiente del estado y divide dicho comportamiento en diferentes estados.
- Se localizan fácilmente las responsabilidades de los estados específicos, dado que se encuentran en las clases que corresponden a cada estado. Esto brinda una mayor claridad en el desarrollo y el mantenimiento posterior.
- Hace los cambios de estado explícitos puesto que en otros tipos de implementación los estados se cambian modificando valores en variables, mientras que aquí al estar representado cada estado.
- Los objetos de los estados pueden ser compartidos.

Por su parte las desventajas serían:

- Se incrementa el número de subclases.

13. CONCLUSIÓN

- El patrón State está pensado para que un programa sea escalable y fácil de manejar.
- El diseño se puede aplicar en diferentes situaciones.
- El patrón ha sido utilizado para la implementación y diseño de software de juegos.