Politécnico Colombiano Jaime Isaza Cadavid

Facultad de Ingeniería

State pattern

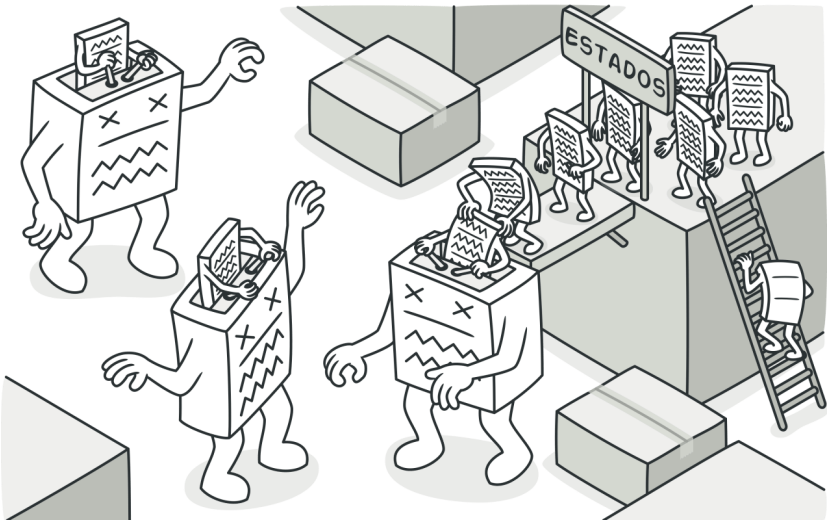
Diseño de software

Autor: Claudia
Apellidos: Gil Sánchez

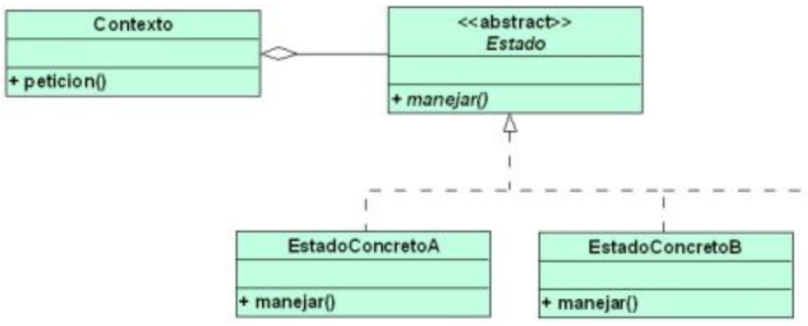
9 de febrero de 2022

Introducción

es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Para lograr esto es necesario crear una serie de clases que representarán los distintos estados por los que puede pasar la aplicación; es decir, se requiere de una clase por cada estado por el que la aplicación pueda pasar. El patrón de diseño State se caracteriza por modificar su comportamiento dependiendo del estado en el que se encuentra la aplicación y se utiliza cuando el comportamiento de un objeto cambia dependiendo del estado del mismo. En conclusiòn el objetivo de este patrón es permitirnos de forma sencilla que un objeto cambie su comportamiento en función del estado en el que se encuentra.



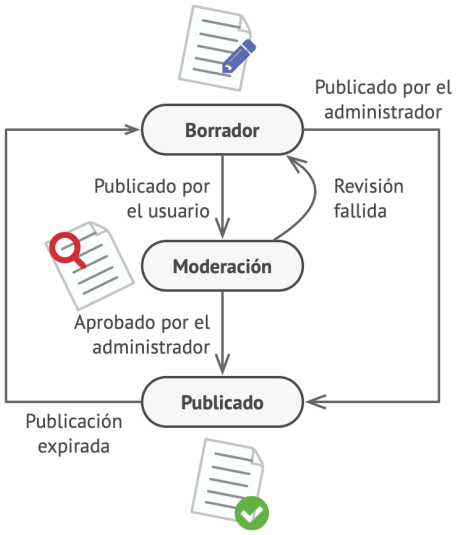
Según el siguiente diagrama en UML podemos apreciar que el objeto cuyo estado es susceptible de cambiar (Contexto) contendrá una referencia a otro objeto que define los distintos tip



State

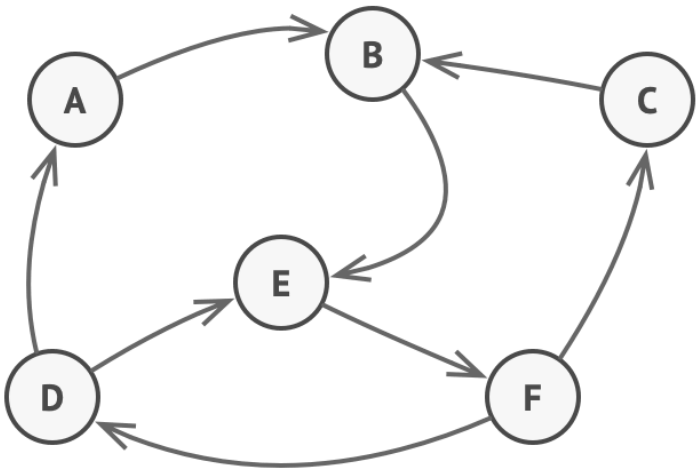
Si está cerrada y le damos la orden para conectarse pasará a conectando, una vez establecida la conexión pasará a abierta.

Mientras la conexión esté abierta podremos enviar paquetes, pero si intentamos enviar paquetes cuando la conexión está cerrada no mostrará un error. En determinadas ocasiones, cuando el contexto en el que se está desarrollando requiere que un objeto tenga diferentes comportamientos según el estado en que se encuentra, resulta complicado poder manejar el cambio de comportamientos y los estados de dicho objeto, todos dentro del mismo bloque de código. El patrón State propone una solución a esta complicación, creando básicamente, un objeto por cada estado posible del objeto que lo llama. Si la conexión está abierta y le decimos que se conecte simplemente puede ignorar esta orden.



Posibles estados y transiciones de un objeto de documento.

1. Los componentes del patrón son los siguientes:
   - Context: Representa el componente que puede cambiar de estado, el cual tiene entre sus propiedades el estado actual. En el ejemplo de la máquina de refrescos, éste sería la máquina como tal.
   - AbstractState: Clase base para la generación de los distintos estados. Se recomienda que sea una clase abstracta en lugar de una interface debido a que podemos definir comportamientos por default y así afectar el funcionamiento de todos los estados.
   - ConcreteState: Cada uno de estos componentes representa un posible estado por el cual la aplicación puede pasar, por lo que tendremos un ConcreteState por cada estado posible. Esta clase debe de heredar de AbstractState.

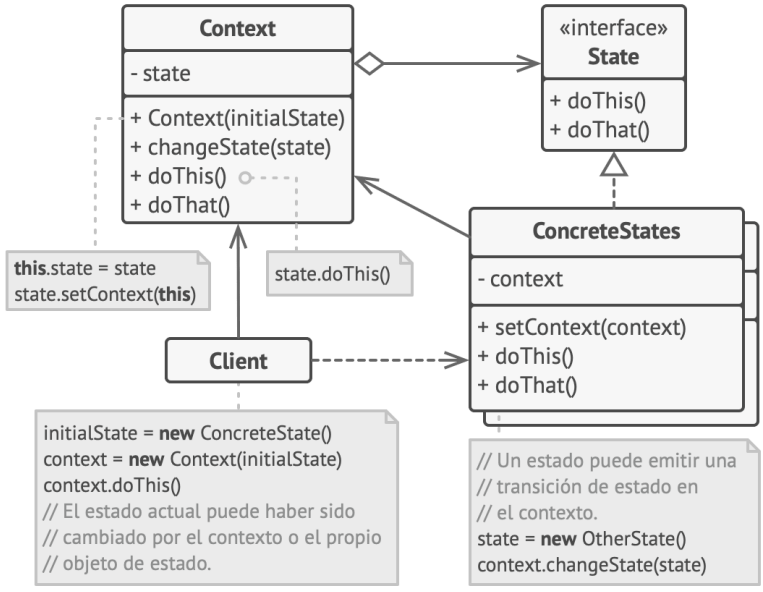


Máquina de estados finitos.

Estructura

1 La clase **Contexto** almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado. El contexto se comunica con el objeto de estado a través de la interfaz de estado. El contexto expone un modificador (*setter*) para pasarle un nuevo objeto de estado.

2 La interfaz **Estado** declara los métodos específicos del estado. Estos métodos deben tener sentido para todos los estados concretos, porque no querrás que uno de tus estados tenga métodos inútiles que nunca son invocados.



3 Los **Estados Concretos** proporcionan sus propias implementaciones para los métodos específicos del estado. Para evitar la duplicación de código similar a través de varios estados, puedes incluir clases abstractas intermedias que encapsulen algún comportamiento común.

Los objetos de estado pueden almacenar una referencia inversa al objeto de contexto. A través de esta referencia, el estado puede extraer cualquier información requerida del objeto de contexto, así como iniciar transiciones de estado.

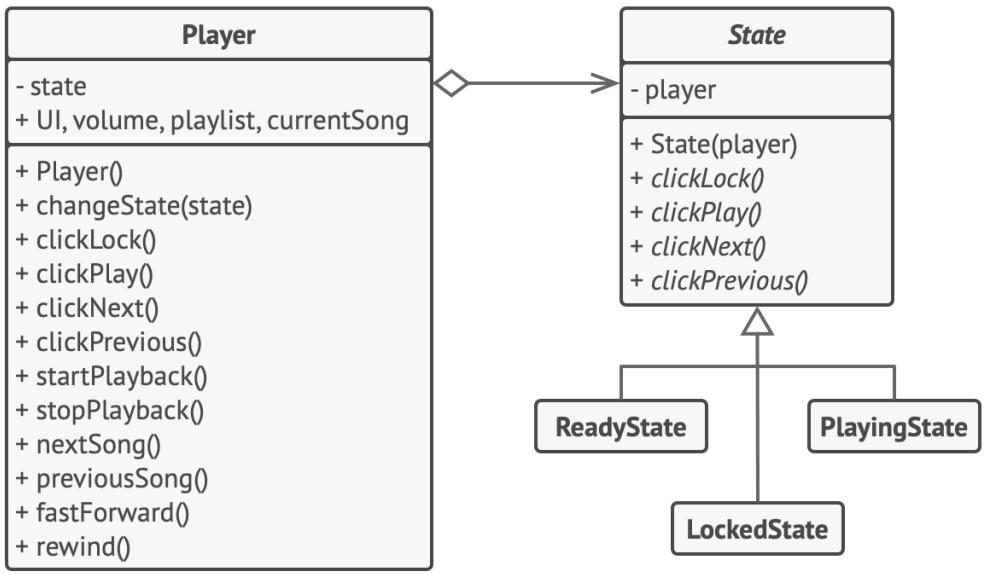
4 Tanto el estado de contexto como el concreto pueden establecer el nuevo estado del contexto y realizar la transición de estado sustituyendo el objeto de estado vinculado al contexto.

Consecuencias:

Se encuentran las siguientes ventajas:
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. Esta facilidad la brinda el hecho que los diferentes estados están representados por un único atributo (state) y no envueltos en diferentes variables y grandes condicionales.
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 State pueden ser compartidos si no contienen variables de instancia, esto se puede lograr si el estado que representan esta enteramente codificado en su tipo. Cuando se hace esto estos estados son Flyweights sin estado intrínseco.
Facilita la ampliación de estados
Permite a un objeto cambiar de clase en tiempo de ejecución dado que al cambiar sus responsabilidades por las de otro objeto de otra clase la herencia y responsabilidades del primero han cambiado por las del segundo.
Se encuentran la siguiente desventaja:
Se incrementa el número de subclases.

Similitud con otros patrones

Ambos patrones se basan en la composición: cambian el comportamiento del contexto delegando parte del trabajo a objetos ayudantes. De hecho, todos estos patrones se basan en la composición, que consiste en delegar trabajo a otros objetos. Sin embargo, State no restringe las dependencias entre estados concretos, permitiéndoles alterar el estado del contexto a voluntad. También puede comunicar a otros desarrolladores el problema que resuelve. Strategy hace que estos objetos sean completamente independientes y no se conozcan entre sí.



Ejemplo de cambio del comportamiento de un objeto con objetos de estado.

Cómo implementarlo

Para cambiar el estado del contexto, crea una instancia de una de las clases de estado y pásala a la clase contexto. Puede ser una clase existente que ya tiene el código dependiente del estado, o una nueva clase, si el código específico del estado está distribuido a lo largo de varias clases. En la clase contexto, añade un campo de referencia del tipo de interfaz de estado y un modificador (setter) público que permita sobrescribir el valor de ese campo.

Vuelve a repasar el método del contexto y sustituye los condicionales de estado vacíos por llamadas a métodos correspondientes del objeto de estado.

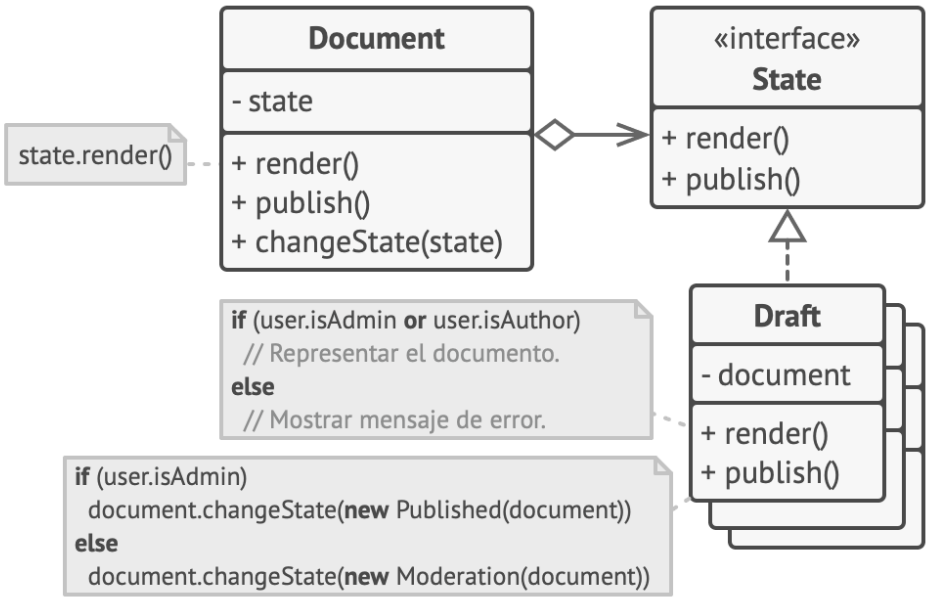
Al mover el código a la clase estado, puede que descubras que depende de miembros privados del contexto. Después repasa los métodos del contexto y extrae todo el código relacionado con ese estado para meterlo en tu clase recién creada. Anida las clases de estado en la clase contexto, pero sólo si tu lenguaje de programación soporta clases anidadas. Convierte el comportamiento que estás extrayendo para ponerlo en un método público en el contexto e invócalo desde la clase de estado.

Solucion del patron

En lugar de implementar todos los comportamientos por su cuenta, el objeto original, llamado contexto, almacena una referencia a uno de los objetos de estado que representa su estado actual y delega todo el trabajo relacionado con el estado a ese objeto.

Documento delega el trabajo a un objeto de estado Documento delega el trabajo a un objeto de estado.

Para la transición del contexto a otro estado, sustituye el objeto de estado activo por otro objeto que represente ese nuevo estado. El patrón State sugiere que crees nuevas clases para todos los estados posibles de un objeto y extraigas todos los comportamientos específicos del estado para colocarlos dentro de esas clases.



Documento delega el trabajo a un objeto de estado.

Conclusiones

En este post hemos visto como implementar una versión del patrón State, y es que este patrón tiene muchas formas de ser implementado. Este patrón es muy utilizados para temas de IA y nos ha permitido modelar los distintos comportamientos del enemigo sin tener que modificar para nada esta clase.

Ejemplo del patrón de diseño

```
package refactoring_guru.state.example.states;

import refactoring_guru.state.example.ui.Player;

/**
 * Common interface for all states.
 */
public abstract class State {
    Player player;

    /**
     * Context passes itself through the state constructor. This may help a
     * state to fetch some useful context data if needed.
     */
    State(Player player) {
        this.player = player;
    }

    public abstract String onLock();
    public abstract String onPlay();
    public abstract String onNext();
    public abstract String onPrevious();
}
```

states/LockedState.java

```
package refactoring_guru.state.example.states;

import refactoring_guru.state.example.ui.Player;

/**
 * Concrete states provide the special implementation for all interface methods.
 */
public class LockedState extends State {

    LockedState(Player player) {
        super(player);
        player.setPlaying(false);
    }

    @Override
    public String onLock() {
        if (player.isPlaying()) {
            player.changeState(new ReadyState(player));
            return "Stop playing";
        } else {
            return "Locked...";
        }
    }

    @Override
    public String onPlay() {
        player.changeState(new ReadyState(player));
        return "Ready";
    }
```

```
    @Override
    public String onNext() {
        return "Locked...";
    }

    @Override
    public String onPrevious() {
        return "Locked...";
    }
}
```

states/ReadyState.java

```
package refactoring_guru.state.example.states;

import refactoring_guru.state.example.ui.Player;

/**
 * They can also trigger state transitions in the context.
 */
public class ReadyState extends State {

    public ReadyState(Player player) {
        super(player);
    }

    @Override
    public String onLock() {
        player.changeState(new LockedState(player));
        return "Locked...";
    }

    @Override
    public String onPlay() {
        String action = player.startPlayback();
        player.changeState(new PlayingState(player));
        return action;
    }

    @Override
    public String onNext() {
        return "Locked...";
    }
    @Override
    public String onPrevious() {
        return "Locked...";
    }
}
```

```
package refactoring_guru.state.example.states;

import refactoring_guru.state.example.ui.Player;

public class PlayingState extends State {

    PlayingState(Player player) {
        super(player);
    }

    @Override
    public String onLock() {
        player.changeState(new LockedState(player));
        player.setCurrentTrackAfterStop();
        return "Stop playing";
    }

    @Override
    public String onPlay() {
        player.changeState(new ReadyState(player));
        return "Paused...";
    }

    @Override
    public String onNext() {
        return player.nextTrack();
    }

    @Override
    public String onPrevious() {
        return player.previousTrack();
    }
}
```

ui/Player.java: Código principal del reproductor

```
package refactoring_guru.state.example.ui;

import refactoring_guru.state.example.states.ReadyState;
import refactoring_guru.state.example.states.State;

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

public class Player {
    private State state;
    private boolean playing = false;
    private List<String> playlist = new ArrayList<>();
    private int currentTrack = 0;

    public Player() {
        this.state = new ReadyState(this);
        setPlaying(true);
        for (int i = 1; i <= 12; i++) {
            playlist.add("Track " + i);
        }
    }

    public void changeState(State state) {
        this.state = state;
    }

    public State getState() {
        return state;
    }
    public void setPlaying(boolean playing) {
        this.playing = playing;
    }

    public boolean isPlaying() {
        return playing;
    }

    public String startPlayback() {
        return "Playing " + playlist.get(currentTrack);
    }

    public String nextTrack() {
        currentTrack++;
        if (currentTrack > playlist.size() - 1) {
            currentTrack = 0;
        }
        return "Playing " + playlist.get(currentTrack);
    }

    public String previousTrack() {
        currentTrack--;
        if (currentTrack < 0) {
            currentTrack = playlist.size() - 1;
        }
        return "Playing " + playlist.get(currentTrack);
    }

    public void setCurrentTrackAfterStop() {
        this.currentTrack = 0;
    }
}
```

```java
package refactoring_guru.state.example.ui;

import javax.swing.*;
import java.awt.*;

public class UI {
    private Player player;
    private static JTextField textField = new JTextField();

    public UI(Player player) {
        this.player = player;
    }

    public void init() {
        JFrame frame = new JFrame("Test player");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel context = new JPanel();
        context.setLayout(new BoxLayout(context, BoxLayout.Y_AXIS));
        frame.getContentPane().add(context);
        JPanel buttons = new JPanel(new FlowLayout(FlowLayout.CENTER));
        context.add(textField);
        context.add(buttons);

        // Context delegates handling user's input to a state object. Naturally,
        // the outcome will depend on what state is currently active, since all
        // states can handle the input differently.
        JButton play = new JButton("Play");
        play.addActionListener(e -> textField.setText(player.getState().onPlay()));
        JButton stop = new JButton("Stop");
        stop.addActionListener(e -> textField.setText(player.getState().onLock()));
        JButton next = new JButton("Next");
        next.addActionListener(e -> textField.setText(player.getState().onNext()));
        JButton prev = new JButton("Prev");
        prev.addActionListener(e -> textField.setText(player.getState().onPrevious()));
        frame.setVisible(true);
        frame.setSize(300, 100);
        buttons.add(play);
        buttons.add(stop);
        buttons.add(next);
        buttons.add(prev);
    }
}
```

Demo.java: Código de inicialización

```java
package refactoring_guru.state.example;

import refactoring_guru.state.example.ui.Player;
import refactoring_guru.state.example.ui.UI;

/**
 * Demo class. Everything comes together here.
 */
public class Demo {
    public static void main(String[] args) {
        Player player = new Player();
        UI ui = new UI(player);
        ui.init();
    }
}
```

Referencias

Parra, D. (2021, October 11). Patrones de diseño - State pattern - The power ups . The power ups https://thepowerups-learning.com/patrones-de-diseno-state-pattern/.

Patrón de diseño State (comportamiento). (n.d.). Informaticapc.com. Retrieved February 9, 2022, from https://informaticapc.com/patrones-de-diseno/state.php.

State en Java. (n.d.). Refactoring.guru. Retrieved February 9, 2022, from https://refactoring.guru/es/design-patterns/state/java/example.

((N.d.). Wikipedia.Org. Retrieved February 9, 2022, from https://es.wikipedia.org/wiki/State_(patr\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{o\global\mathchardef\accent@spacefactor\spacefactor}\let\begingroup\let\typeout\protect\begingroup\def\MessageBreak{Ω(Font)}\let\protect\immediate\write\m@ne{LaTeXFontInfo:oninputline218.}\endgroup\endgroup\relax\let\ignorespaces\relax\accent19o\egroup\spacefactor\accent@spacefactor\futurelet\@let@token\protect\penalty\@M\hskip\z@skipn_de_dise~no).