



**POLITÉCNICO COLOMBIANO**  
**JAIME ISAZA CADAVID**

Patrón de diseño

**STATE**

Realizado por

**Sebastian Zapata Castaño**

Dirigido por

Hector Manuel Vanegas Solis

Universidad

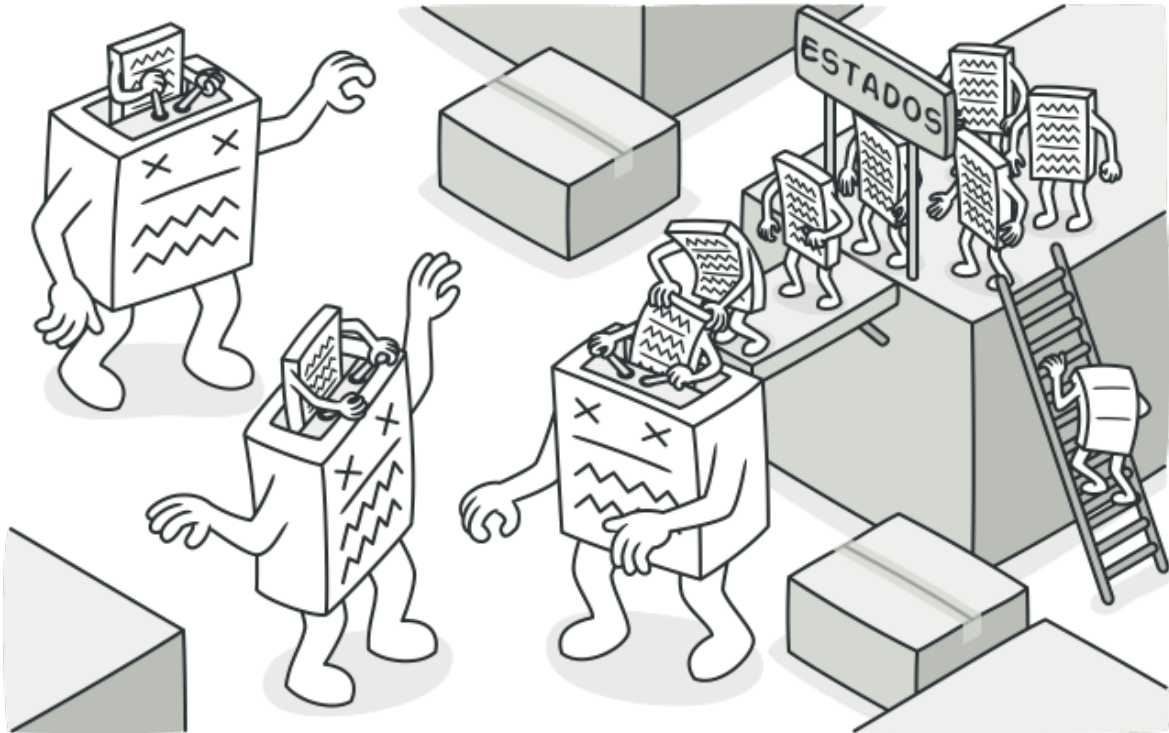
Politecnico Colombiano Jaime Isaza Cadavid

**Febrero, 2022**

# Descripcion general

---

State es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase. Iterator es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).



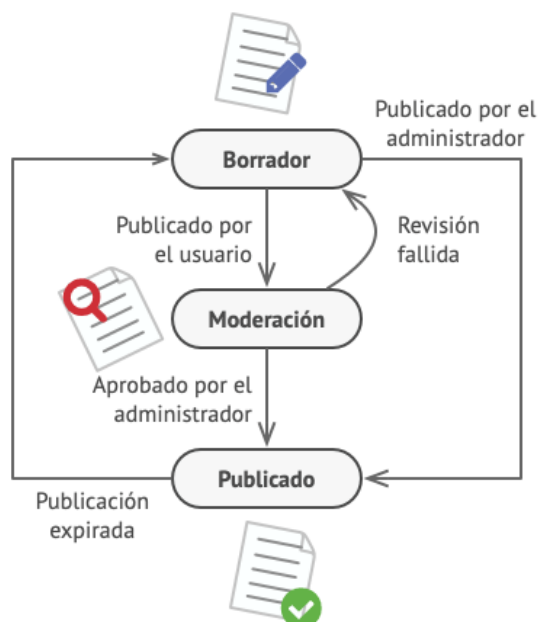
# ¿Que soluiona este patrón de diseño?

El patrón State está estrechamente relacionado con el concepto de la Máquina de estados finitos.

La idea principal es que, en cualquier momento dado, un programa puede encontrarse en un número finito de estados. Dentro de cada estado único, el programa se comporta de forma diferente y puede cambiar de un estado a otro instantáneamente. Sin embargo, dependiendo de un estado actual, el programa puede cambiar o no a otros estados. Estas normas de cambio llamadas transiciones también son finitas y predeterminadas.

También puedes aplicar esta solución a los objetos. Imagina que tienes una clase Documento. Un documento puede encontrarse en uno de estos tres estados: Borrador, Moderación y Publicado. El método publicar del documento funciona de forma ligeramente distinta en cada estado:

- En `Borrador`, mueve el documento a moderación.
- En `Moderación`, hace público el documento, pero sólo si el usuario actual es un administrador.
- En `Publicado`, no hace nada en absoluto.

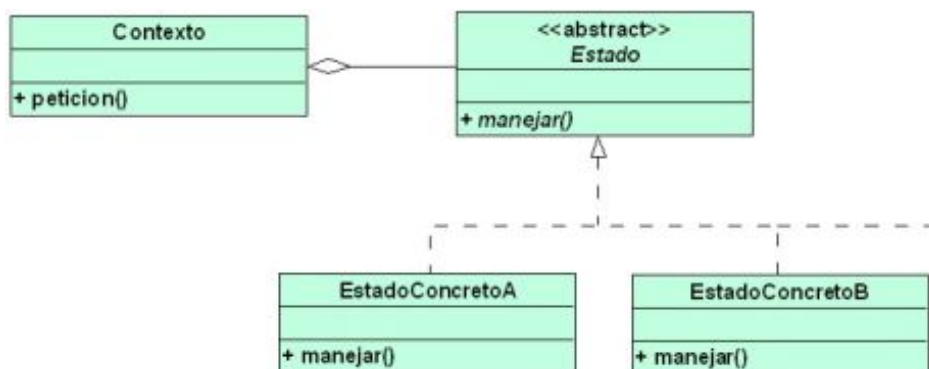


El patrón State sugiere que crees nuevas clases para todos los estados posibles de un objeto y extraigas todos los comportamientos específicos del estado para colocarlos dentro de esas clases.

# State como diagrama UML

---

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 tipos de estado en que se puede encontrar.



## Ventajas y desventajas de Estrategia

### 0.0.1. Ventajas del patrón de diseño state

Principio de responsabilidad única. Organiza el código relacionado con estados particulares en clases separadas.

Principio de abierto/cerrado. Introduce nuevos estados sin cambiar clases de estado existentes o la clase contexto.

Simplifica el código del contexto eliminando voluminosos condicionales de máquina de estados.

### 0.0.2. Desventajas del patrón de diseño state


Aplicar el patrón puede resultar excesivo si una máquina de estados sólo tiene unos pocos estados o raramente cambia.

# Ejemplo en código del patrón de diseño State

---



```
01. package State;
02.
03. public class Main
04. {
05.     public static void main(String[] args)
06.     {
07.         Semaforo objSemaforo = new Semaforo();
08.
09.         // Muestra el aviso por defecto (verde, no hay alerta)
10.         objSemaforo.mostrar();
11.
12.         objSemaforo.setEstado( new EstadoNaranja() );
13.         objSemaforo.mostrar();
14.
15.         objSemaforo.setEstado( new EstadoRojo() );
16.         objSemaforo.mostrar();
17.     }
18. }
```


 [text pop-up](#)

```

01. package State;
02.
03. public class Semaforo
04. {
05.     private EstadoSemaforo objEstadoSemaforo;
06.
07.     // -----
08.
09.     public Semaforo() {
10.         this.objEstadoSemaforo = new EstadoVerde();
11.     }
12.
13.     // -----
14.
15.     public void setEstado( EstadoSemaforo objEstadoSemaforo ) {
16.         this.objEstadoSemaforo = objEstadoSemaforo;
17.     }
18.
19.     // -----
20.
21.     @Override
22.     public void mostrar() {
23.         this.objEstadoSemaforo.mostrar();
24.     }
25. }

```

[Collapse Code](#)

 [text pop-up](#)

```

01. Package State;
02.
03. public abstract class EstadoSemaforo
04. {
05.     // Método que deberán crear las clases que hereden de ésta
06.     public abstract void mostrar();
07. }

```

[Collapse Code](#)

01. `package State;`  
 02.  
 03. `public class EstadoVerde extends EstadoSemaforo`  
 04. `{`  
 05.  `public EstadoVerde() {`  
 06.  `}`  
 07.  
 08. `// -----`  
 09.  
 10. `@Override`  
 11. `public void mostrar() {`  
 12.  `System.out.println("Luz verde");`  
 13. `}`  
 14. `}`

✖ [text](#) [pop-up](#)

[Collapse Code](#)

01. `package State;`  
 02.  
 03. `public class EstadoVerde extends EstadoSemaforo`  
 04. `{`  
 05.  `public EstadoVerde() {`  
 06.  `}`  
 07.  
 08. `// -----`  
 09.  
 10. `@Override`  
 11. `public void mostrar() {`  
 12.  `System.out.println("Luz naranja");`  
 13. `}`  
 14. `}`

✖ [text](#) [pop-up](#)

[Collapse Code](#)

01. `package State;`  
 02.  
 03. `public class EstadoVerde extends EstadoSemaforo`  
 04. `{`  
 05.  `public EstadoVerde() {`  
 06.  `}`  
 07.  
 08. `// -----`  
 09.  
 10. `@Override`  
 11. `public void mostrar() {`  
 12.  `System.out.println("Luz roja");`  
 13. `}`  
 14. `}`

[copy](#) [text](#) [pop-up](#)

[Collapse Code](#)

## Referencias

- Patrón de diseño State (comportamiento). (n.d.). Informatcapc.com. Retrieved February 9, 2022, from <https://informatcapc.com/patrones-de-diseno/state.php>
- State. (n.d.). Refactoring.guru. Retrieved February 9, 2022, from <https://refactoring.guru/es/design-patterns/state>