PATRON DE DISEÑO

STATE

HÉCTOR DANIEL MEDRANO MEZA

¿QUÉ ES EL PATRON DE DISEÑO «STATE»?

Es un patrón de diseño **de comportamiento**, el cual permite a un objeto cambiar su comportamiento cuando un estado interno cambia. Hace parecer al objeto como si hubiese cambiado de clase.

FINITO», EN LAS QUE HAY UNA CANTIDAD DESIGNADA DE ESTADOS EN LAS QUE UN OBJETO SE PUEDE ENCONTRAR; DEPENDIENDO DEL ESTADO, EL PROGRAMA SE COMPORTA DE UNA FORMA U OTRA.

EJEMPLO ~ 1

A modo de ejemplo tenemos el siguiente caso, donde queremos representar los diferentes estados de un documento: Borrador, moderación y aprobación.

Cada estado tienen sus sets de reglas que lo llevan de uno a otro estado, según corresponda.

EJEMPLO ~ 2

DocumentDemo.java > ...

```java
public class DocumentDemo {
    private String state;

    public void publish()
    {
        switch(state)
        {
            case "draft":
                state = "moderation";
                break;

            case "moderation":
                state = "published";
                break;

            case "published":
                // No ocurre nada más
                break;
        }
    }
}
```

Si quisiésemos representarlo dentro del código, quedaría algo similar a esto.

En ciertos casos puede ser suficiente, pero lo cierto es que llega un punto en el que el código, de crecer demasiado, puede volverse insostenible.

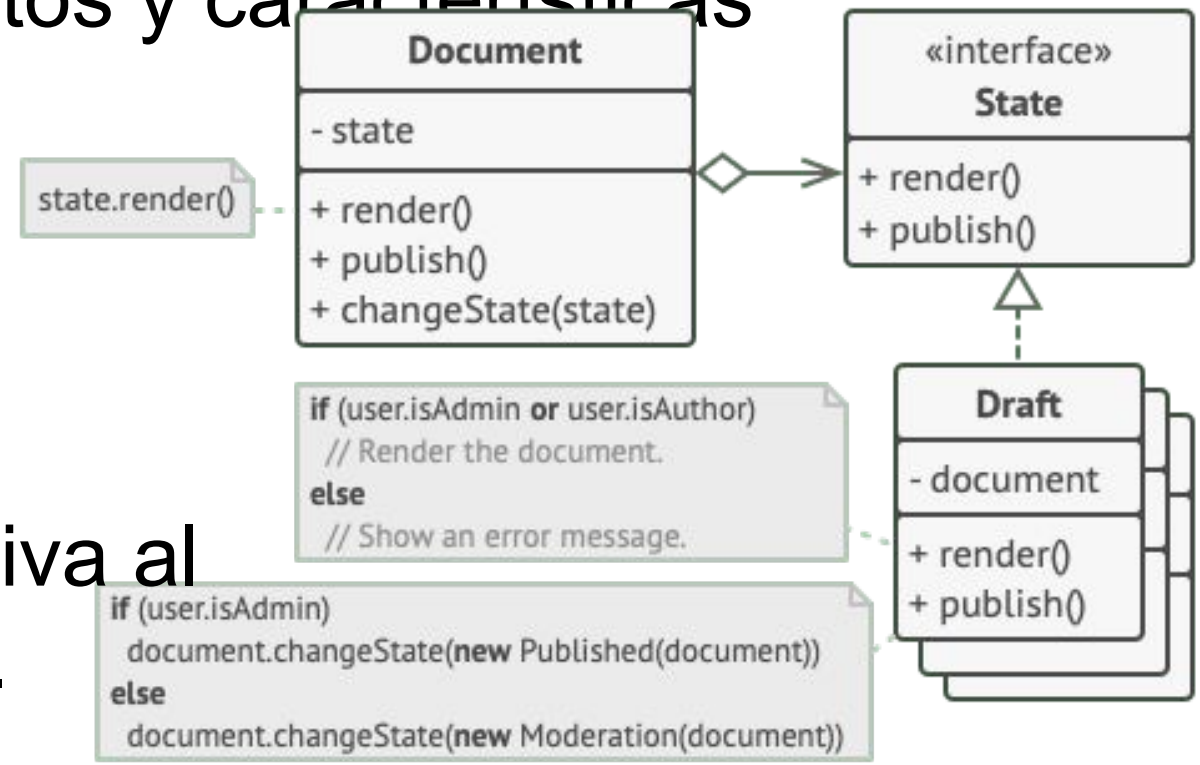
EJEMPLO ~ 3

Mientras más estados, o comportamientos dependientes de estados, el código va a volverse más y más grande y complejo. Cualquier cambio a la lógica entre transición de estados involucraría cambiar código en cada uno de los

SOLUCIÓN

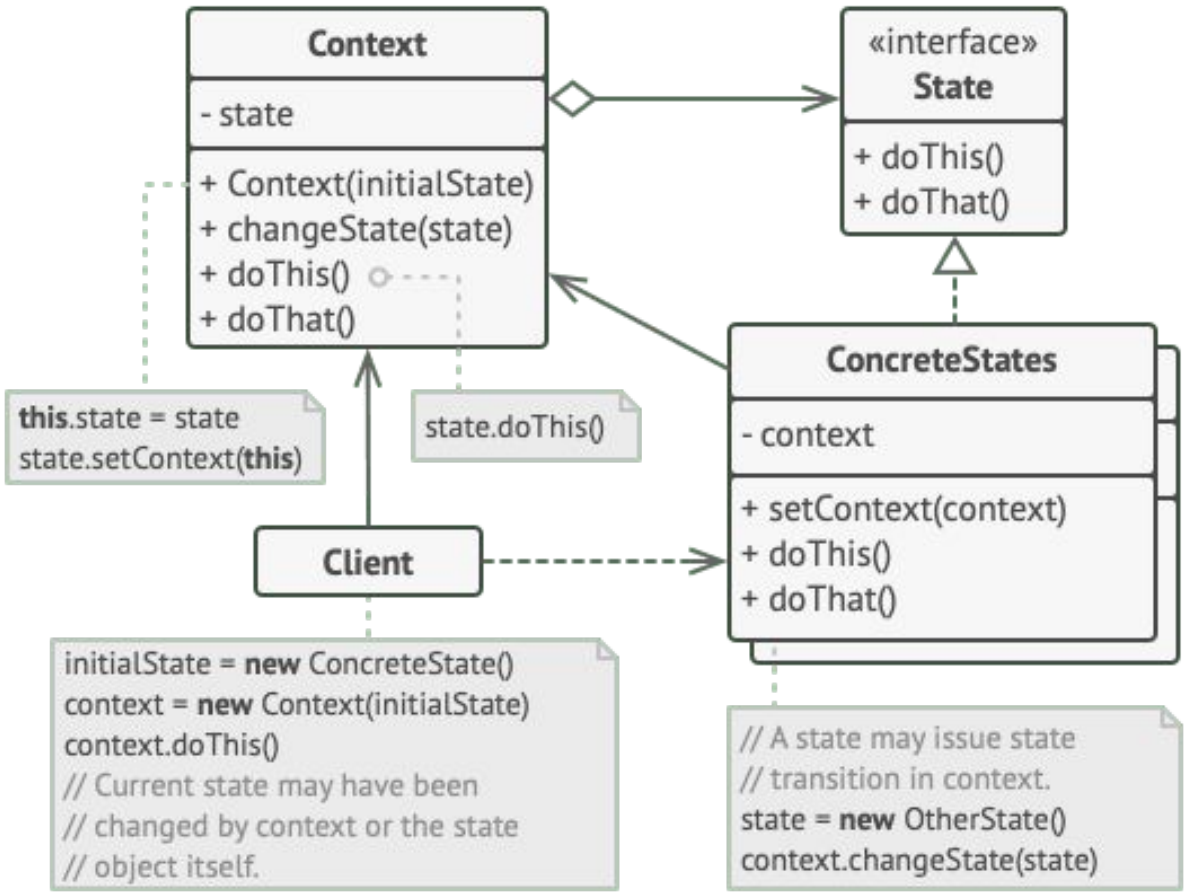
En lugar de utilizar métodos condicionales para cada estado, los representaremos mediante sus propias clases, las cuales guardarán sus comportamientos y características respectivas.

El objeto original, llamado «Contexto», mantiene una referencia a su estado, mientras que la clase respectiva al estado realiza todo el trabajo.



ESTRUCTURA

1. **«Contexto»:** Mantiene una referencia a su estado actual. Tiene un setter que le permite cambiar su estado.
2. **«Estado»:** Es una interfaz que declara los métodos que tienen en común todos los estados.
3. **«Casos Concretos»:** Cada clase creada en base a la interfaz representa un estado. Implementa los comportamientos



PROS

- Organiza el código en clases según el comportamiento del estado, dándole una mejor organización.
- Se pueden introducir nuevos estados sin cambiar contextos o clases ya existentes.
- Simplifica el código al eliminar la necesidad de Switches e If's.

CONTRAS

- Cuando no hay tantos posibles estados y tenemos **por seguro** que el código no va a complicarse demasiado, el patrón de diseño resulta innecesario, y es más preferible usar un Switch en su lugar.

EJEMPLO EN CÓDIGO ~ 1

Clase Abstracta «State»

document > State.java > ...

```java
public abstract class State
{
    Document document;

    State(Document document)
    {
        this.document = document;
    }

    public abstract void Publish();
    public abstract String CurrentStage();
}
```

Contexto «Document»

document > Document.java > ...

```java
public class Document
{
    private State state;

    public Document()
    {
        this.state = new Draft(this);
    }

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

    public void NextState()
    {
        this.state.Publish();
    }

    public void PrintStage()
    {
        System.out.println(this.state.CurrentStage());
    }
}
```

EJEMPLO EN CÓDIGO ~ 2

Estado «Draft»

document > Draft.java > ...

```java
public class Draft extends State
{
    public Draft(Document document)
    {
        super(document);
    }

    @Override
    public void Publish()
    {
        document.ChangeToNextState(new Moderation(document));
    }

    @Override
    public String CurrentStage()
    {
        return "Etapa de Borrador";
    }
}
```

Estado «Moderation»

document > Moderation.java > ...

```java
public class Moderation extends State
{
    public Moderation(Document document)
    {
        super(document);
    }

    @Override
    public void Publish()
    {
        document.ChangeToNextState(new Published(document));
    }

    @Override
    public String CurrentStage()
    {
        return "Etapa de Moderación";
    }
}
```

Estado «Published»

```java
public class Published extends State
{
    public Published(Document document)
    {
        super(document);
    }

    @Override
    public void Publish()
    {
        // Nada
    }

    @Override
    public String CurrentStage()
    {
        return "Etapa de Publicación";
    }
}
```

EJEMPLO EN CÓDIGO ~ 3

Main

document > Main.java > ...

```java
class Main
{
    Run | Debug
    public static void main(String[] args)
    {
        Document document = new Document();

        document.PrintStage();

        document.NextState();

        document.PrintStage();

        document.NextState();

        document.PrintStage();

        document.NextState();
    }
}
```

Impresión

```
Etapa de Borrador
Etapa de Moderación
Etapa de Publicación
```

EJERCICIO

Realizar un programa que represente a un semáforo. Hay tres estados: Luz verde, amarilla y roja. El semáforo iniciará por default en la luz verde.

A su vez, el semáforo tendrá 3 métodos:

- NextState(): Cambia el color del semáforo al siguiente, según la secuencia (ver el diagrama).

- GetCurrentState(): Indica cuál es el color actual del semáforo.

- Advance(): Indica si se puede avanzar o no. Para el color verde indicar que sí; para el amarillo indicar que debe de ir disminuyendo su velocidad; para el rojo indicar que no se puede avanzar.

No debe de utilizarse ni un solo if o switch en todo el programa.