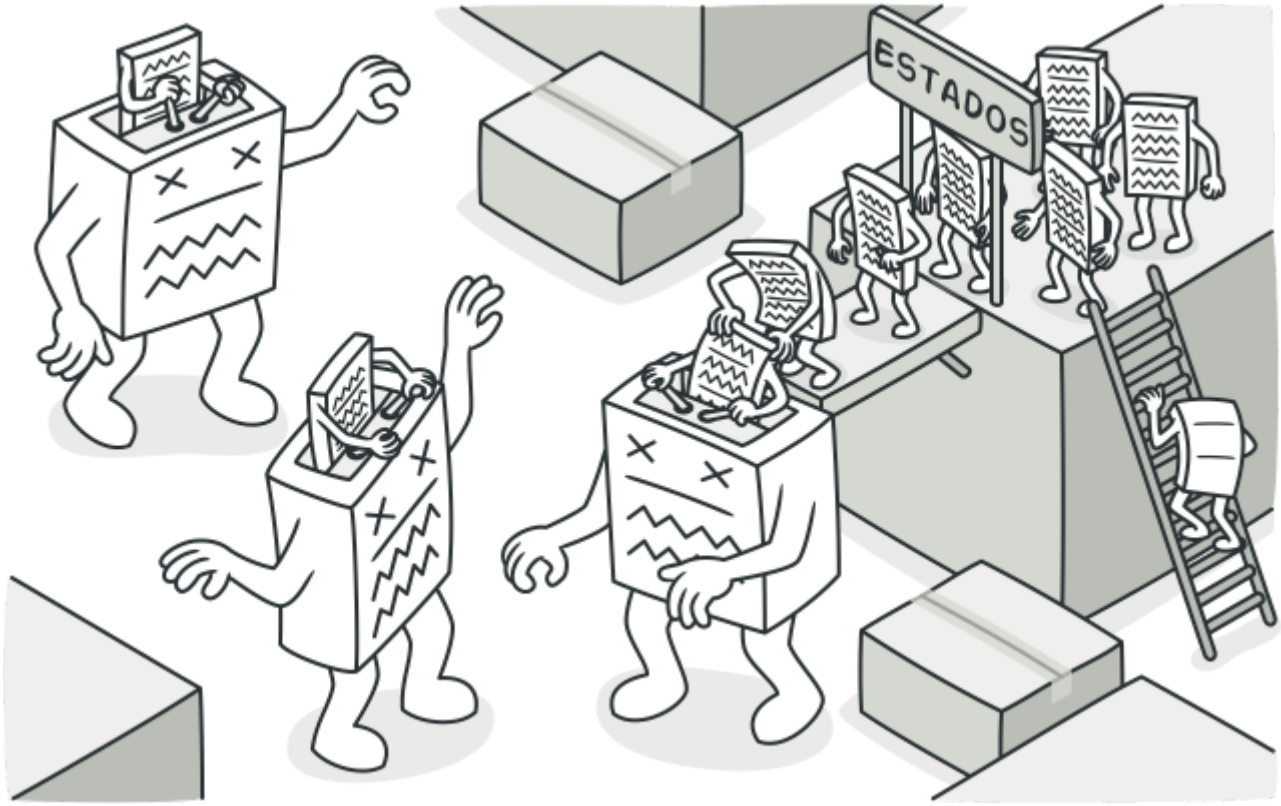
State

También llamado: Estado

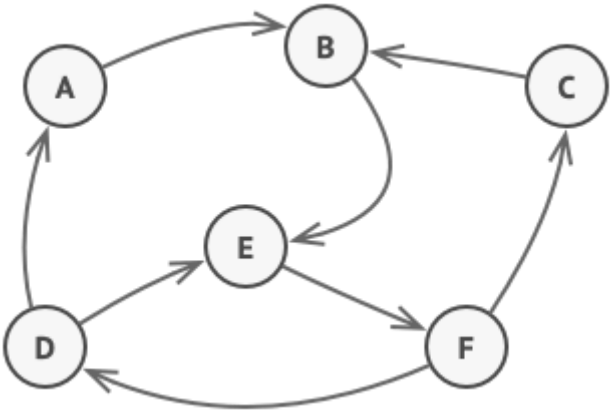
Propósito

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.



Problema

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

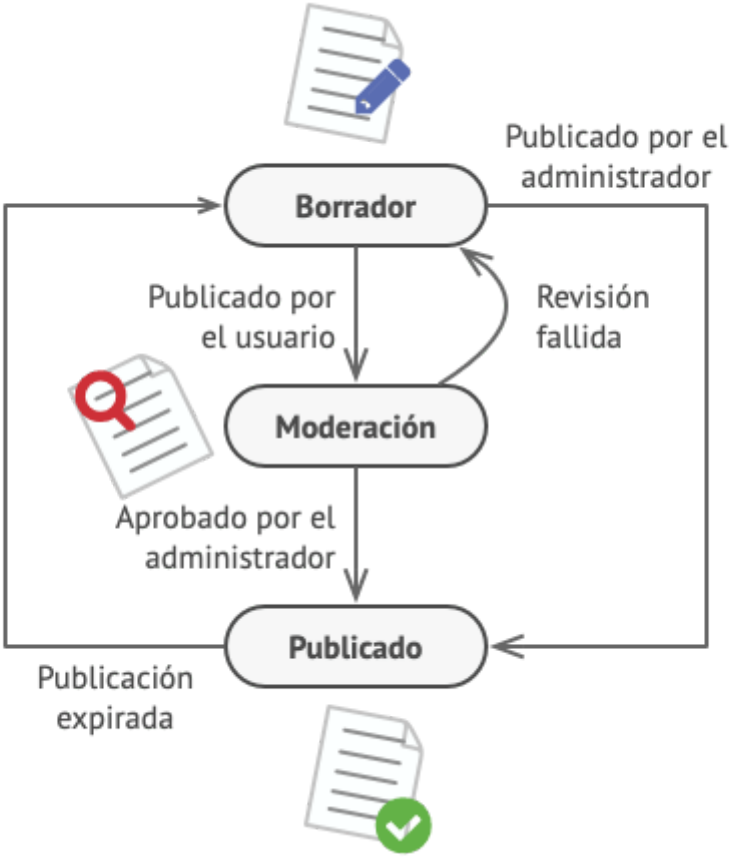


Máquina de estados finitos.

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.



Posibles estados y transiciones de un objeto de documento.

Las máquinas de estado se implementan normalmente con muchos operadores condicionales (`if` o `switch`) que seleccionan el comportamiento adecuado dependiendo del estado actual del objeto. Normalmente, este “estado” es tan solo un grupo de valores de los campos del objeto. Aunque nunca hayas oído hablar de máquinas de estados finitos, probablemente hayas implementado un estado al menos alguna vez. ¿Te suena esta estructura de código?

```
class Document is
    field state: string
    // ...
    method publish() is
        switch (state)
            "draft":
                state = "moderation"
                break
            "moderation":
                if (currentUser.role == "admin")
                    state = "published"
                break
            "published":
                // No hacer nada.
                break
    // ...
```

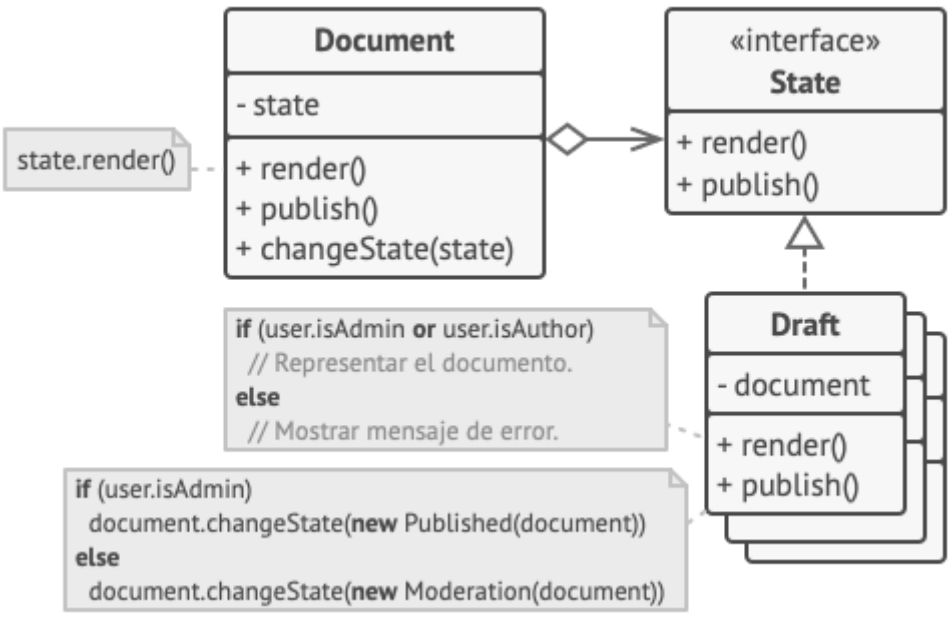
La mayor debilidad de una máquina de estado basada en condicionales se revela una vez que empezamos a añadir más y más estados y comportamientos dependientes de estados a la clase `Documento`. La mayoría de los métodos contendrán condicionales monstruosos que eligen el comportamiento adecuado de un método de acuerdo con el estado actual. Un código así es muy difícil de mantener, porque cualquier cambio en la lógica de transición puede requerir cambiar los condicionales de estado de cada método.

El problema tiende a empeorar con la evolución del proyecto. Es bastante difícil predecir todos los estados y transiciones posibles en la etapa de diseño. Por ello, una máquina de estados esbelta, creada con un grupo limitado de condicionales, puede crecer hasta convertirse en un abotargado desastre con el tiempo.

😊 Solución

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.

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.

Para la transición del contexto a otro estado, sustituye el objeto de estado activo por otro objeto que represente ese nuevo estado. Esto sólo es posible si todas las clases de estado

siguen la misma interfaz y el propio contexto funciona con esos objetos a través de esa interfaz.

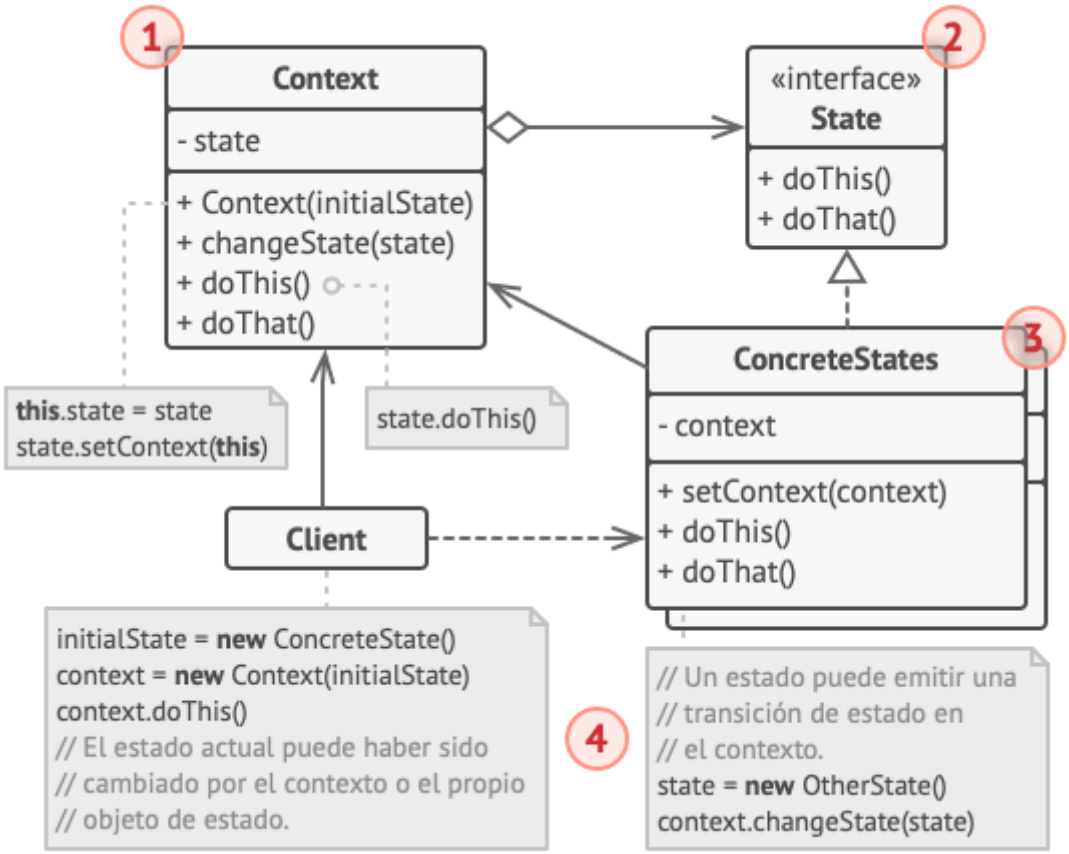
Esta estructura puede resultar similar al patrón **Strategy**, pero hay una diferencia clave. En el patrón State, los estados particulares pueden conocerse entre sí e iniciar transiciones de un estado a otro, mientras que las estrategias casi nunca se conocen.

Analogía en el mundo real

Los botones e interruptores de tu smartphone se comportan de forma diferente dependiendo del estado actual del dispositivo:

- Cuando el teléfono está desbloqueado, al pulsar botones se ejecutan varias funciones.
- Cuando el teléfono está bloqueado, pulsar un botón desbloquea la pantalla.
- Cuando la batería del teléfono está baja, pulsar un botón muestra la pantalla de carga.

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.

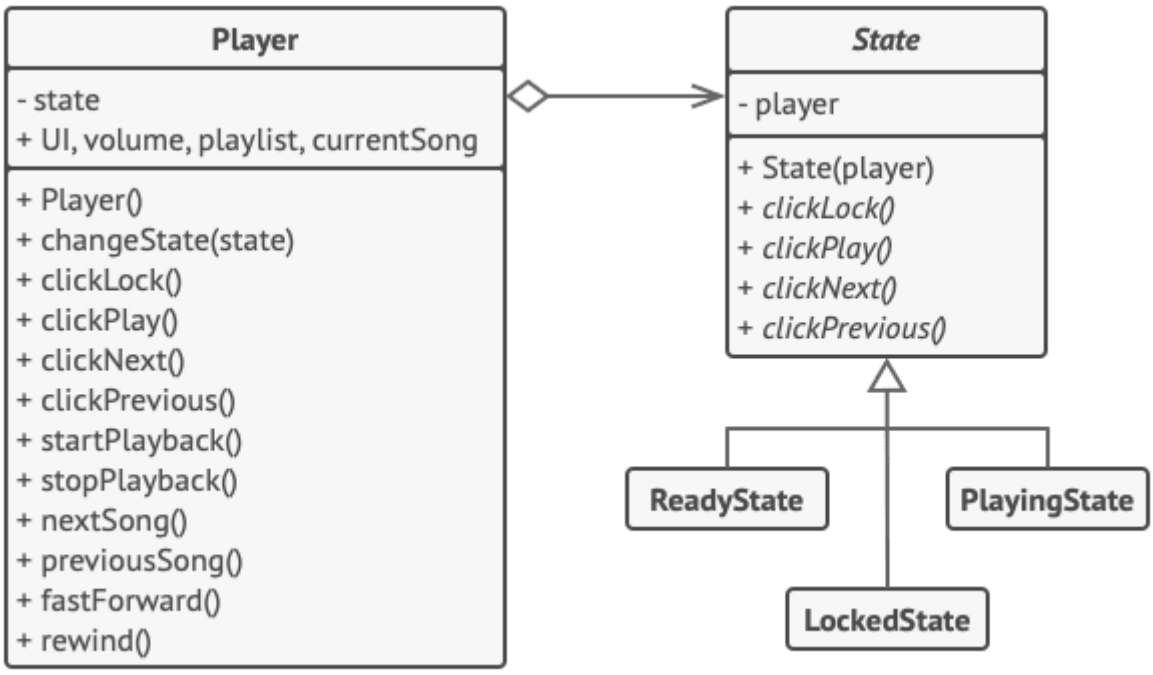
3. Los **Estados Concretos** proporcionan sus propias implementaciones para los métodos específicos del estado. Para evitar la duplicación de código similar a través de varios estados, puedes incluir clases abstractas intermedias que encapsulen algún comportamiento común.

   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.

Pseudocódigo

En este ejemplo, el patrón **State** permite a los mismos controles del reproductor de medios comportarse de forma diferente, dependiendo del estado actual de reproducción.



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

El objeto principal del reproductor siempre está vinculado a un objeto de estado que realiza la mayor parte del trabajo del reproductor. Algunas acciones sustituyen el objeto de estado actual del reproductor por otro, lo cual cambia la forma en la que el reproductor reacciona a las interacciones del usuario.

```
// La clase ReproductordeAudio actúa como un contexto. También
// mantiene una referencia a una instancia de una de las clases
// estado que representa el estado actual del reproductor de
// audio.
class AudioPlayer is
    field state: State
    field UI, volume, playlist, currentSong

    constructor AudioPlayer() is
        this.state = new ReadyState(this)

        // El contexto delega la gestión de entradas del usuario
        // a un objeto de estado. Naturalmente, el resultado
        // depende del estado que esté activo ahora, ya que cada
        // estado puede gestionar las entradas de manera
        // diferente.
        UI = new UserInterface()
        UI.lockButton.onClick(this.clickLock)
        UI.playButton.onClick(this.clickPlay)
        UI.nextButton.onClick(this.clickNext)
        UI.prevButton.onClick(this.clickPrevious)

    // Otros objetos deben ser capaces de cambiar el estado
```

```
        // activo del reproductor.
        method changeState(state: State) is
            this.state = state

        // Los métodos UI delegan la ejecución al estado activo.
        method clickLock() is
            state.clickLock()
        method clickPlay() is
            state.clickPlay()
        method clickNext() is
            state.clickNext()
        method clickPrevious() is
            state.clickPrevious()

        // Un estado puede invocar algunos métodos del servicio en
        // el contexto.
        method startPlayback() is
            // ...
        method stopPlayback() is
            // ...
        method nextSong() is
            // ...
        method previousSong() is
            // ...
        method fastForward(time) is
            // ...
        method rewind(time) is
            // ...


    // La clase estado base declara métodos que todos los estados
    // concretos deben implementar, y también proporciona una
    // referencia inversa al objeto de contexto asociado con el
    // estado. Los estados pueden utilizar la referencia inversa
    // para dirigir el contexto a otro estado.
    abstract class State is
        protected field player: AudioPlayer

        // El contexto se pasa a sí mismo a través del constructor
        // del estado. Esto puede ayudar al estado a extraer
        // información de contexto útil si la necesita.
        constructor State(player) is
            this.player = player

        abstract method clickLock()
        abstract method clickPlay()
        abstract method clickNext()
        abstract method clickPrevious()


    // Los estados concretos implementan varios comportamientos
    // asociados a un estado del contexto.
```

```
class LockedState extends State is

    // Cuando desbloqueas a un jugador bloqueado, puede asumir
    // uno de dos estados.
    method clickLock() is
        if (player.playing)
            player.changeState(new PlayingState(player))
        else
            player.changeState(new ReadyState(player))

    method clickPlay() is
        // Bloqueado, no hace nada.

    method clickNext() is
        // Bloqueado, no hace nada.

    method clickPrevious() is
        // Bloqueado, no hace nada.

// También pueden disparar transiciones de estado en el
// contexto.
class ReadyState extends State is
    method clickLock() is
        player.changeState(new LockedState(player))

    method clickPlay() is
        player.startPlayback()
        player.changeState(new PlayingState(player))

    method clickNext() is
        player.nextSong()

    method clickPrevious() is
        player.previousSong()


class PlayingState extends State is
    method clickLock() is
        player.changeState(new LockedState(player))

    method clickPlay() is
        player.stopPlayback()
        player.changeState(new ReadyState(player))

    method clickNext() is
        if (event.doubleclick)
            player.nextSong()
        else
            player.fastForward(5)

    method clickPrevious() is
        if (event.doubleclick)
```

```
            player.previous()
        else
            player.rewind(5)
```

Aplicabilidad

Utiliza el patrón State cuando tengas un objeto que se comporta de forma diferente dependiendo de su estado actual, el número de estados sea enorme y el código específico del estado cambie con frecuencia.

El patrón sugiere que extraigas todo el código específico del estado y lo metas dentro de un grupo de clases específicas. Como resultado, puedes añadir nuevos estados o cambiar los existentes independientemente entre sí, reduciendo el costo de mantenimiento.

Utiliza el patrón cuando tengas una clase contaminada con enormes condicionales que alteran el modo en que se comporta la clase de acuerdo con los valores actuales de los campos de la clase.

El patrón State te permite extraer ramas de esos condicionales a métodos de las clases estado correspondientes. Al hacerlo, también puedes limpiar campos temporales y métodos de ayuda implicados en código específico del estado de fuera de tu clase principal.

Utiliza el patrón State cuando tengas mucho código duplicado por estados similares y transiciones de una máquina de estados basada en condiciones.

El patrón State te permite componer jerarquías de clases de estado y reducir la duplicación, extrayendo el código común y metiéndolo en clases abstractas base.

Cómo implementarlo

1. Decide qué clase actuará como 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.

2. Declara la interfaz de estado. Aunque puede replicar todos los métodos declarados en el contexto, céntrate en los que pueden contener comportamientos específicos del estado.

3. Para cada estado actual, crea una clase derivada de la interfaz de estado. Después repasa los métodos del contexto y extrae todo el código relacionado con ese estado para meterlo en tu clase recién creada.

   Al mover el código a la clase estado, puede que descubras que depende de miembros privados del contexto. Hay varias soluciones alternativas:

   - Haz públicos esos campos o métodos.
   - Convierte el comportamiento que estás extrayendo para ponerlo en un método público en el contexto e invócalo desde la clase de estado. Esta forma es desagradable pero rápida y siempre podrás arreglarlo más adelante.
   - Anida las clases de estado en la clase contexto, pero sólo si tu lenguaje de programación soporta clases anidadas.

4. 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.

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

6. Para cambiar el estado del contexto, crea una instancia de una de las clases de estado y pásala a la clase contexto. Puedes hacer esto dentro de la propia clase contexto, en distintos estados, o en el cliente. Se haga de una forma u otra, la clase se vuelve dependiente de la clase de estado concreto que instancia.

⚖ Pros y contras

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

⇄ Relaciones con otros patrones

- **Bridge**, **State**, **Strategy** (y, hasta cierto punto, **Adapter**) tienen estructuras muy similares. De hecho, todos estos patrones se basan en la composición, que consiste en delegar trabajo a otros objetos. Sin embargo, todos ellos solucionan problemas diferentes. Un patrón no es simplemente una receta para estructurar tu código de una forma específica. También puede comunicar a otros desarrolladores el problema que resuelve.

- **State** puede considerarse una extensión de **Strategy**. Ambos patrones se basan en la composición: cambian el comportamiento del contexto delegando parte del trabajo a objetos ayudantes. *Strategy* hace que estos objetos sean completamente independientes y no se conozcan entre sí. Sin embargo, *State* no restringe las dependencias entre estados concretos, permitiéndoles alterar el estado del contexto a voluntad.

</> Ejemplos de código