**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 .

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.

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.

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.

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](https://refactoring.guru/es/design-patterns/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.

**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.

**Pros**

* *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.

## Contras

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