UNIVERSIDAD TECNOLÓGICA DE PANAMÁ

FACULTAD DE INGENIERÍA DE SISTEMAS COMPUTACIONALES

LICENCIATURA EN INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

INGENIERÍA DE SOFTWARE II

CHARLA: PATRONES DE COMPORTAMIENTO
STATE, STRATEGY, TEMPLATE METHOD & VISITOR

ALUMNO:

STEVEN CEDEÑO	**20-70-4684**
CHRISTIAN NAVARRO	**8-1019-116**
ARIEL FLORES	**8-970-2465**
ORIS OVALLES	**8-969-1733**
DAVID TORRES	**8-963-282**

PROFESORA:

VANESSA CASTILLO

GRUPO 1IL-143

SEMESTRE I 2023

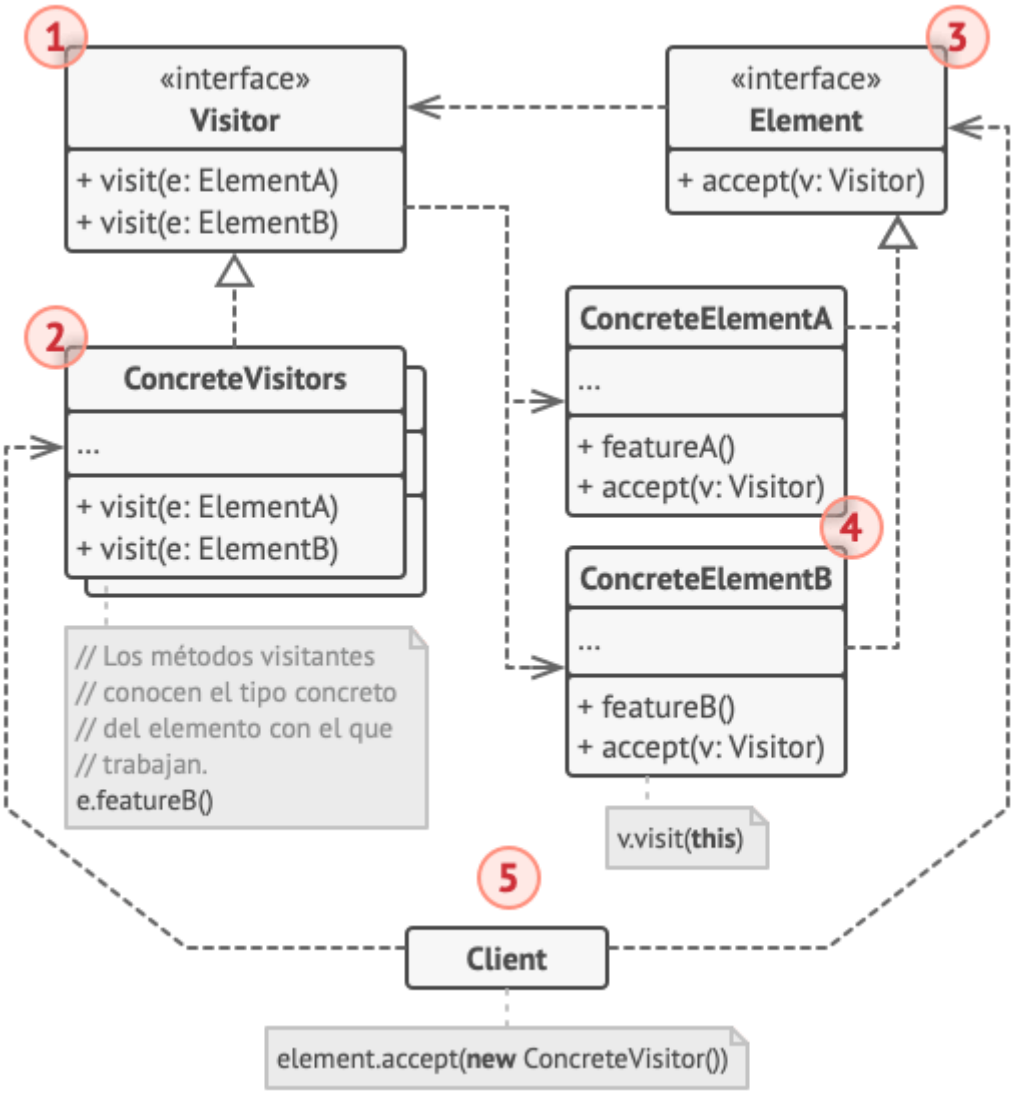
DESARROLLO

VISITOR (STEVEN)

El patrón Visitor es un patrón de diseño de software que se utiliza para separar la estructura de un objeto de sus operaciones o algoritmos. Proporciona una forma de agregar operaciones adicionales a una clase sin modificar su estructura.

El propósito principal del patrón Visitor es separar la lógica de la operación de los objetos sobre los que opera. Esto significa que se pueden agregar nuevas operaciones sin modificar las clases existentes. El patrón Visitor utiliza el polimorfismo para lograr esto.

ESTRUCTURA



1. La interfaz **Visitante** declara un grupo de métodos visitantes que pueden tomar elementos concretos de una estructura de objetos como argumentos. Estos métodos pueden tener los mismos nombres si el programa está escrito en un lenguaje que soporte la sobrecarga, pero los tipos de sus parámetros deben ser diferentes.

2. Cada **Visitante Concreto** implementa varias versiones de los mismos comportamientos, personalizadas para las distintas clases de elemento concreto.

3. La **interfaz Elemento** declara un método para "aceptar" visitantes. Este método deberá contar con un parámetro declarado con el tipo de la interfaz visitante.

4. Cada **Elemento Concreto** debe implementar el método de aceptación. El propósito de este método es redirigir la llamada al método adecuado del visitante correspondiente a la clase de elemento actual. Piensa que, aunque una clase base de elemento implemente este método, todas las subclases deben sobrescribir este método en sus propias clases e invocar el método adecuado en el objeto visitante.

5. El **Cliente** representa normalmente una colección o algún otro objeto complejo (por ejemplo, un árbol **Composite**). A menudo, los clientes no son conscientes de todas las clases de elemento concreto porque trabajan con objetos de esa colección a través de una interfaz abstracta.

APLICABILIDAD

- **Utilizar el patrón Visitor cuando se necesite realizar una operación sobre todos los elementos de una compleja estructura de objetos (por ejemplo, un árbol de objetos).**

El patrón Visitor te permite ejecutar una operación sobre un grupo de objetos con diferentes clases, haciendo que un objeto visitante implemente distintas variantes de la misma operación que correspondan a todas las clases objetivo.

- **Utilizar el patrón Visitor para limpiar la lógica de negocio de comportamientos auxiliares.**

El patrón Visitor nos permite hacer que las clases primarias de nuestra aplicación estén más centradas en sus trabajos principales extrayendo el resto de los comportamientos y poniéndolos dentro de un grupo de clases visitantes.

- **Utilizar el patrón cuando un comportamiento solo tenga sentido en algunas clases de una jerarquía de clases, pero no en otras.**

Se puede extraer este comportamiento y ponerlo en una clase visitante separada e implementar únicamente aquellos métodos visitantes que acepten objetos de clases relevantes, dejando el resto vacíos.

PROS

- **Principio de abierto/cerrado.** Puedes introducir un nuevo comportamiento que puede funcionar con objetos de clases diferentes sin cambiar esas clases.

- **Principio de responsabilidad única.** Puedes tomar varias versiones del mismo comportamiento y ponerlas en la misma clase.

- **Un objeto visitante puede acumular cierta información útil mientras trabaja con varios objetos.** Esto puede resultar útil cuando quieras atravesar una compleja estructura de objetos, como un árbol de objetos, y aplicar el visitante a cada objeto de esa estructura.

CONTRAS

- Debes actualizar todos los visitantes cada vez que una clase se añada o elimine de la jerarquía de elementos.

- Los visitantes pueden carecer del acceso necesario a los campos y métodos privados de los elementos con los que se supone que deben trabajar.

RELACIONES CON OTROS PATRONES

- Se puede tratar al patrón Visitor como una versión más potente del patrón **Command**. Sus objetos pueden ejecutar operaciones sobre varios objetos de distintas clases.

- Se puede utilizar el patrón Visitor para ejecutar una operación sobre un árbol **Composite** entero.

- Se puede utilizar Visitor en conjunto con **Iterator** para recorrer una estructura de datos compleja y ejecutar alguna operación sobre sus elementos, incluso aunque todos tengan clases distintas.

PASOS PARA LA IMPLEMENTACIÓN

1. Declarar la interfaz visitante con un grupo de métodos “visitantes”, uno por cada clase de elemento concreto existente en el programa.

2. Declarar la interfaz de elemento. Si estamos trabajando con una jerarquía de clases de elemento existente, añadir el método abstracto de “aceptación” a la clase base de la jerarquía. Este método debe aceptar un objeto visitante como argumento.

3. Implementar los métodos de aceptación en todas las clases de elemento concreto. Estos métodos simplemente deben redirigir la llamada a un método visitante en el objeto visitante entrante que coincida con la clase del elemento actual.

4. Las clases de elemento sólo deben funcionar con visitantes a través de la interfaz visitante. Los visitantes, sin embargo, deben conocer todas las clases de elemento concreto, referenciadas como tipos de parámetro de los métodos de visita.

5. Por cada comportamiento que no pueda implementarse dentro de la jerarquía de elementos, crea una nueva clase concreta visitante e implementa todos los métodos visitantes.

   Puede que nos encontremos en una situación en la que el visitante necesite acceso a algunos miembros privados de la clase elemento. En este caso, podemos hacer estos campos o métodos públicos, violando la encapsulación del elemento, o anidar la clase visitante en la clase elemento. Esto último sólo es posible si tenemos la suerte de trabajar con un lenguaje de programación que soporte clases anidadas.

6. El cliente debe crear objetos visitantes y pasarlos dentro de elementos a través de métodos de “aceptación”.

IMPLEMENTACIÓN

```
# Paso 1: Declarar la interfaz Visitante con métodos para cada clase de
elemento concreto existente en el programa.
class Visitante:
    def visitarElementoA(self, elementoA):
        pass

    def visitarElementoB(self, elementoB):
        pass

# Paso 2: Declarar la interfaz de Elemento.
class Elemento:
    def aceptar(self, visitante):
        pass
```

```python
# Paso 3: Implementa los métodos de aceptación en las clases de elemento
concreto.
class ElementoA(Elemento):
    def aceptar(self, visitante):
        visitante.visitarElementoA(self)

    def operacionA(self):
        # Lógica específica de ElementoA
        print("Operación A en ElementoA")

class ElementoB(Elemento):
    def aceptar(self, visitante):
        visitante.visitarElementoB(self)

    def operacionB(self):
        # Lógica específica de ElementoB
        print("Operación B en ElementoB")

# Paso 4: Implementa los métodos visitantes en una clase concreta visitante.
class VisitanteConcreto(Visitante):
    def visitarElementoA(self, elementoA):
        # Implementación del método visitante para ElementoA
        elementoA.operacionA()

    def visitarElementoB(self, elementoB):
        # Implementación del método visitante para ElementoB
        elementoB.operacionB()

# Paso 5: Cliente que crea objetos visitantes y los pasa a través de métodos
de aceptación.
if __name__ == "__main__":
    elementoA = ElementoA()
    elementoB = ElementoB()

    visitante = VisitanteConcreto()

    elementoA.aceptar(visitante)
    elementoB.aceptar(visitante)
```

En este código Python, se definen las clases **Visitante** y **Elemento** como interfaces utilizando clases base sin métodos implementados. Luego, se implementan las clases **ElementoA** y **ElementoB,** donde se define el método **aceptar** y las operaciones específicas de cada clase.

Después, se crea la clase **VisitanteConcreto** que hereda de **Visitante** e implementa los métodos visitantes correspondientes. Por último, en el bloque **if __name__ ==**

"__main__":, se crea una instancia de **ElementoA**, una instancia **de ElementoB**, una instancia de **VisitanteConcreto** y se llama al método **aceptar** en cada elemento pasando el visitante correspondiente.

STRATEGY (DAVID)

El patrón de diseño de software de strategy permite la selección dinámica de algoritmos en tiempo de ejecución. Según el entorno o los requisitos únicos, puede encapsular varios algoritmos o técnicas y hacerlos intercambiables. Este patrón garantiza que cada estrategia tenga una firma de método consistente al definir una interfaz común para todas las estrategias. Cada estrategia utiliza esta interfaz para implementar el algoritmo de una manera única.

La clase de contexto que hace uso de las estrategias realiza un seguimiento del enfoque elegido actualmente y le asigna la responsabilidad de ejecutar el algoritmo. Puede hacer que su código sea más flexible y fácil de mantener utilizando el patrón de estrategia. No es necesario cambiar la clase de contexto para realizar la transición entre diferentes estrategias, y se pueden agregar nuevas estrategias sin cambiar ningún código actual.

Este patrón es especialmente útil en situaciones en las que hay varios algoritmos que trabajan todos hacia el mismo objetivo, pero pueden tener varias compensaciones en términos de eficacia, complejidad o comportamiento. Desvincular la implementación del algoritmo del contexto le permite seleccionar el mejor curso de acción en función de las elecciones del usuario o las circunstancias del tiempo de ejecución.

PROS

Flexibilidad y capacidad de mantenimiento: el patrón de estrategia simplifica el cambio de estrategias existentes o la introducción de otras nuevas sin cambiar la clase de contexto que las utiliza. Con esta flexibilidad, puede modificar el comportamiento de su sistema en tiempo de ejecución o dejar espacio para nuevos requisitos sin realizar modificaciones significativas en el código. Fomenta un diseño flexible y modular.

Reutilización de código: puede aplicar los mismos algoritmos o métodos en una variedad de contextos o aplicaciones encapsulándolos en distintas clases. La capacidad de diseñar, probar y mantener individualmente cada táctica da como resultado un código más limpio y más reutilizable.

Simplifica la lógica compleja: el patrón de estrategia ayuda a desensamblar lógica o algoritmos complejos en partes más pequeñas y fáciles de administrar. Cada enfoque se concentra en un objetivo particular, lo que hace que el código sea más fácil de comprender, depurar y mantener. Al dividir las preocupaciones, apoya el principio de responsabilidad única.

Mejora la capacidad de prueba: el patrón de estrategia le permite probar cada estrategia por separado, reemplazando o burlándose de otras según sea necesario. Debido a esta separación, es más sencillo crear pruebas unitarias que validen la validez y solidez de cada estrategia individual.

Fomenta la coherencia: las estrategias brindan a los usuarios una forma coherente de interactuar con la clase de contexto siguiendo una interfaz estándar. Como resultado, hay menos posibilidades de errores o inconsistencias cuando se emplean varios algoritmos.

Configuración y selección en tiempo de ejecución: el patrón de estrategia permite la selección en tiempo de ejecución de varias estrategias, lo que da como resultado un comportamiento dinámico. Cambiar las tácticas en respuesta a las preferencias del usuario, la configuración del sistema o las circunstancias del tiempo de ejecución permite un comportamiento adaptativo y adaptable.

CONTRAS

Mayor complejidad: el uso del patrón de estrategia puede resultar en el desarrollo de nuevas clases, interfaces e indirección, lo que puede complicar y dificultar la comprensión del código base. Las aplicaciones pequeñas o sencillas que pueden no requerir la flexibilidad y el modularidad adicional que ofrece el patrón de estrategia pueden encontrar esta complejidad particularmente difícil. Si no se maneja adecuadamente, puede resultar en un aumento de la carga de trabajo mental, tiempos de desarrollo más prolongados e incluso la introducción de defectos.

Uso excesivo de clases de clases: el patrón de estrategia requiere con frecuencia la creación de clases distintas para cada estrategia, lo que puede hacer que su base de código contenga más clases de las necesarias. Esto puede hacer que sea más difícil explorar y mantener el código base, especialmente si usa muchas estrategias.

APLICABILIDAD

- Procesamiento de pagos: varios métodos de pago, como tarjetas de crédito, PayPal y transferencias bancarias, se pueden usar en un sistema de procesamiento de pagos como estrategias distintas. Puede elegir entre métodos de pago utilizando el patrón Estrategia según las preferencias del usuario o las necesidades de la organización.

- Desarrollo del juego: en los juegos se utilizan con frecuencia diferentes tácticas para el comportamiento del oponente, la elección de armas o la dificultad del juego. Puede incorporar estos métodos y cambiar entre ellos dinámicamente, ofreciendo varias experiencias de juego, utilizando el patrón de estrategia.

TEMPLATE METHOD (ARIEL)

El patrón de diseño Template Method es una técnica que permite definir la estructura de un algoritmo en una clase base, mientras permite que las subclases implementen pasos específicos sin alterar dicha estructura. En otras palabras, proporciona un esqueleto para un algoritmo, definiendo los pasos comunes y permitiendo que las subclases personalicen los pasos individuales según sea necesario.

Funcionamiento

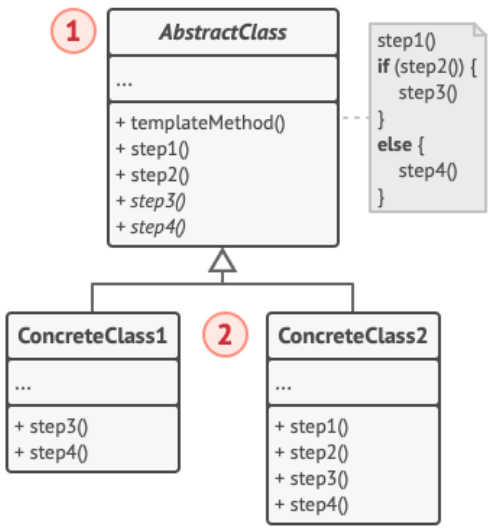
El funcionamiento del patrón Template Method es el siguiente: se crea una clase base abstracta que contiene un método plantilla, el cual define la secuencia de pasos del algoritmo. Algunos de estos pasos pueden ser abstractos, lo que obliga a las subclases a proporcionar su propia implementación, mientras que otros pasos pueden tener una implementación por defecto en la clase base. Las subclases heredan de la clase base y pueden sobrescribir los pasos abstractos o los opcionales según sus necesidades.

El objetivo del patrón Template Method es eliminar la duplicación de código en algoritmos similares, al tiempo que se permite la extensibilidad y la personalización de ciertos pasos. Al definir la estructura general en la clase base y permitir que las subclases implementen pasos individuales, se logra un código más modular y fácil de mantener.

Estructura

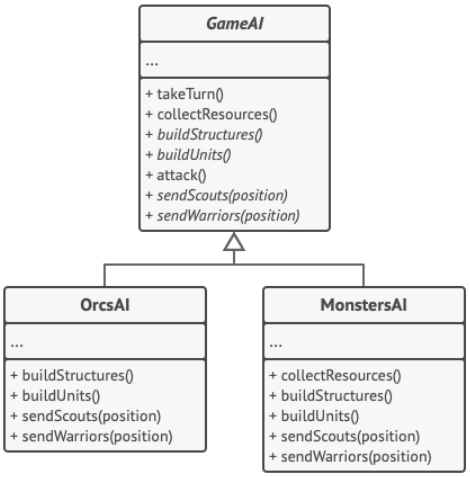
1. La **Clase Abstracta** declara métodos que actúan como pasos de un algoritmo, así como el propio método plantilla que invoca estos métodos en un orden específico. Los pasos pueden declararse abstractos o contar con una implementación por defecto.

2. Las **Clases Concretas** pueden sobrescribir todos los pasos, pero no el propio método plantilla.



Implementación

Un ejemplo de implementación del patrón Template Method es el siguiente: supongamos que tenemos un juego de estrategia en el que diferentes razas (orcos, humanos, monstruos) tienen su propia inteligencia artificial (IA) para tomar decisiones. Cada raza tiene una estructura de unidades y edificios similares, pero con comportamientos específicos.



En este caso, se puede crear una clase base abstracta llamada "GameAI" que contiene un método plantilla llamado "turn()" que define la secuencia de pasos para que la IA tome una decisión en su turno. Los pasos pueden incluir la recolección de recursos, la construcción de estructuras, la creación de unidades y el ataque.

Las subclases, como "OrcsAI" y "MonstersAI", heredan de la clase base "GameAI" y proporcionan su propia implementación de los pasos abstractos, como la construcción de estructuras y la creación de unidades. Cada subclase puede personalizar estos pasos según el comportamiento deseado para la raza correspondiente.

En resumen, el patrón de diseño Template Method permite definir la estructura de un algoritmo en una clase base, mientras permite que las subclases personalicen los pasos individuales. Esto promueve la reutilización de código, el modularidad y la extensibilidad. Un ejemplo práctico es el uso del patrón en un juego de estrategia para implementar la IA de diferentes razas.

STATE (ORIS)

El patrón de diseño State se caracteriza por cambiar el comportamiento de la aplicación según su estado. Para lograr esto, necesitas crear una serie de clases que representen los diferentes estados por los que puede pasar la aplicación; es decir, cada estado en el que puede navegar una aplicación requiere una clase.

La idea básica es que un programa solo puede tener una cantidad limitada de espacio en un momento dado. En cada estado único, el programa se comporta de manera diferente y puede cambiar instantáneamente de un estado a otro. Pero dependiendo del estado actual, el programa puede o no cambiar a otros estados. Estas reglas de cambio, llamadas transiciones, también son definitivas 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.

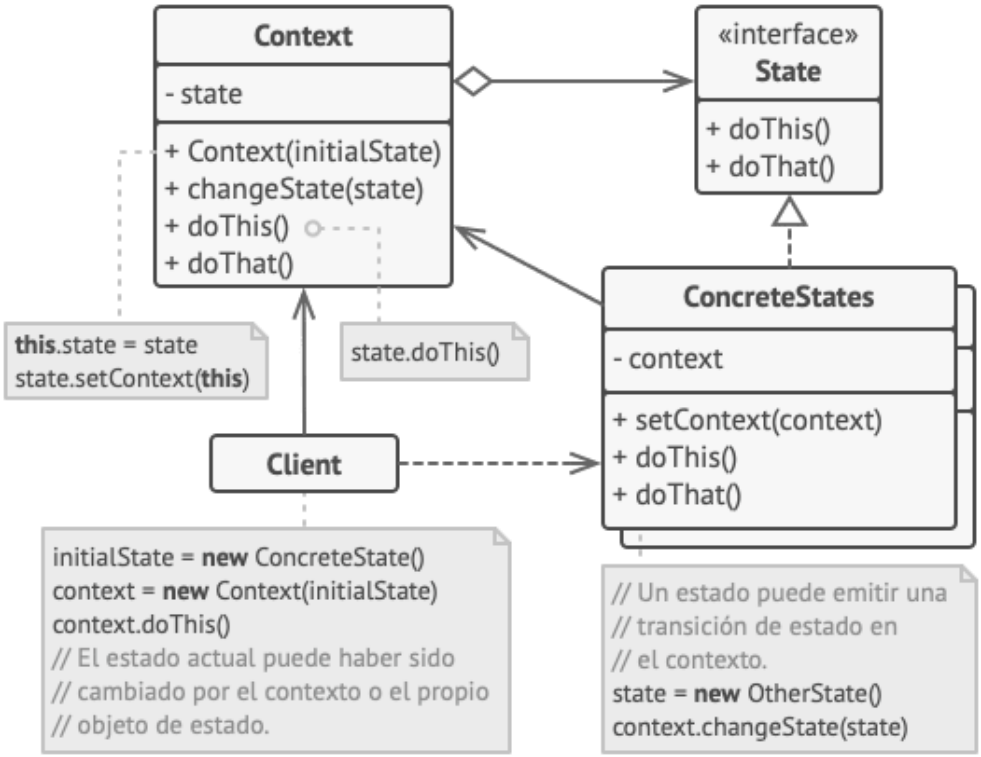
El método publicar del documento funciona de forma ligeramente distinta en cada estado:

- 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.

Aplicabilidad

- Utilice el patrón state cuando tenga un objeto que se comporte de manera diferente a su estado actual, la cantidad de estados sea enorme y el código específico del estado cambie con frecuencia.
- Este patrón puede ser utilizado cuando se tenga una clase contaminada con condicionales enormes que cambian el comportamiento de la clase en función de los valores actuales de los campos de clase.
- 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.

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 State declara métodos específicos del estado. Estos métodos deben tener sentido para cada estado por separado, porque no desea que uno de sus estados tenga métodos inútiles que nunca se llaman.

3. Los Estados concretos proporcionan implementaciones de sus propios métodos específicos del estado. Puede evitar la duplicación de código similar en varios modos agregando clases intermedias abstractas que encapsulen algún comportamiento común.

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.

REFERENCIAS

https://es.wikipedia.org/wiki/Visitor_(patrón_de_diseño)

https://refactoring.guru/es/design-patterns/visitor

State (refactoring.guru)

https://es.wikipedia.org/wiki/Patr%C3%B3n_de_m%C3%A9todo_de_la_plantilla

https://refactoring.guru/es/design-patterns/template-method