UNIVERSIDAD TECNOLÓGICA DE PANAMÁ
FACULTAD DE INGENIERÍA DE SISTEMAS COMPUTACIONALES
LICENCIATURA EN DESARROLLO DE SOFTWARE

ASIGNATURA:

Desarrollo de Software IX (Comercio Electrónico)

ACTIVIDAD:

Presentación N°1

Temas:

Patrón state, Patrón observador

FACILITADOR:

Prof. Erick Agrazal

INTEGRANTES:

Rafael Chung 8-939-1136
Richard Zhang 8-988-1571
Uzziel Aparicio 8-978-1512

GRUPO:

1SL242

Índice

Introducción

En este trabajo de investigación nos enfocaremos en dos patrones fundamentales de la programación orientada a objetos: el Patrón State y el Patrón Observador. A lo largo del desarrollo de esta investigación, se analizarán los componentes principales, las ventajas y desventajas, aplicaciones de uso y ejemplos comparativos de ambos patrones en Javascript. El objetivo principal es comprender cómo su correcta implementación mejora la calidad y organización del código, y por qué su conocimiento es esencial para todo programador que aspire a seguir buenas prácticas de desarrollo.

Patrón State

¿Qué es el Patrón State?

El patrón state es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Es como si éste cambiase de clase. Para lograr esto es necesario crear una serie de clases que representarán los distintos estados por los que puede pasar la aplicación; es decir, se requiere de una clase por cada estado por el que la aplicación pueda pasar.

Analogía en el mundo real

Los botones e interruptores de nuestro smartphone se comportan de manera distinta dependiendo del estado actual del dispositivo.

- Cuando nuestro teléfono está desbloqueado, al pulsar botones se ejecutan varias funciones.
- Cuando nuestro teléfono está bloqueado, pulsar un botón desbloquear nuestra pantalla.

¿Cuándo usarlo?

- Se recomienda utilizar 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.
- Utiliza el patrón cuando tengas una clase contaminada con enormes condiciones que alteran el modo en que se comporta la clase de acuerdo con los valores actuales de los campos de la clase.
- Utiliza el patrón State cuando tengas mucho código duplicado por estados similares y transiciones de una máquina de estados basada en condiciones.

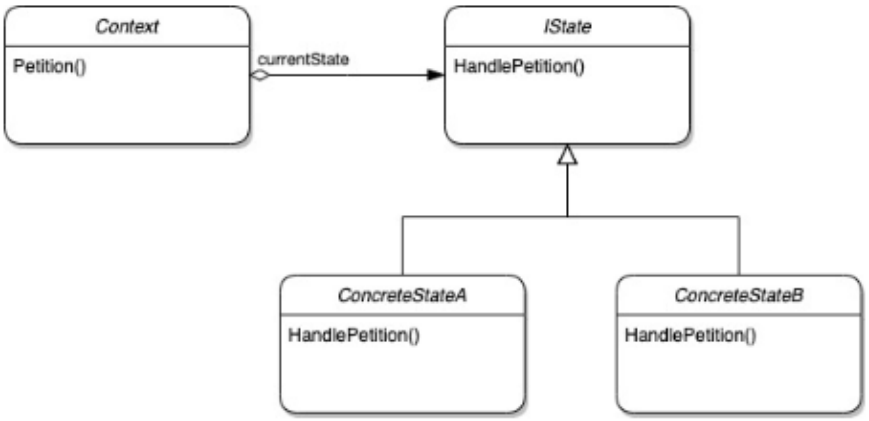
Componentes

Algunos de los componentes que conforman el Patrón State son:

- Contexto (Context)
    - Es la clase principal que mantiene una referencia al objeto de estado actual. También proporciona una interfaz para que los clientes

interactúen y por último, delega el comportamiento específico del estado al objeto de estado actual.

- Estado (State)
    - Es el que define una interfaz común para todos los estados concretos. Esta interfaz declara los métodos que deben implementar los estados específicos.
- Estado Concreto (Concrete State)
    - Implementa el comportamiento específico asociado con un estado particular del contexto. Cada clase de estado concreto encapsula el comportamiento correspondiente a un estado del contexto.
- Transición del Estado (State Transition)
    - Es el proceso mediante el cual el contexto cambia de un estado a otro. Esta transición puede ser iniciada por el contexto o por el propio estado concreto, dependiendo de la implementación.



Ventajas vs Desventajas

Ventajas	**Desventajas**
Cada estado se encapsula en su propia clase, lo que organiza el código relacionado con estados particulares en clases separadas.	Cada estado requiere una nueva clase, lo que puede llevar a una gran cantidad de clases en sistemas complejos.
Es posible introducir nuevos estados sin modificar las clases existentes del estado o la clase contexto.	Requiere un esfuerzo adicional y puede resultar más complejo en comparación con enfoques lineales, ya que implica definir estados, transiciones y acciones,

	lo que aumenta la complejidad del código.
Elimina condicionales voluminosos en la clase contexto, lo que facilita la lectura y mantenimiento del código.	Si hay muchos objetos, cada uno con su propio estado, el uso de memoria puede ser elevado. Esto se puede mitigar utilizando patrones como Singleton para los estados.
Permite cambiar el comportamiento de un objeto en tiempo de ejecución al cambiar su estado interno, facilitando la implementación de comportamientos polimórficos.	Puede existir una dependencia entre el contexto y los estados concretos, lo que puede dificultar la reutilización y el mantenimiento del código.

Ejemplos y Comparación

```
class EstadoEncendido {
    manejar() {
        console.log("La lámpara ya está encendida");
    }
}

class EstadoApagado {
    manejar() {
        console.log("La lámpara está apagada. Encendiendo...");
    }
}

class Lampara {
    constructor() {
        this.estado = new EstadoApagado();
    }

    cambiarEstado(estado) {
        this.estado = estado;
    }

    presionarInterruptor() {
        this.estado.manejar();
    }
}

const lampara = new Lampara();
lampara.presionarInterruptor(); // La lámpara está apagada. Encendiendo...
lampara.cambiarEstado(new EstadoEncendido());
lampara.presionarInterruptor(); // La lámpara ya está encendida
```

Ejemplo de código en JavaScript en el que se aplica el Patrón State

```
class Lampara {
    constructor() {
        this.encendida = false;
    }

    presionarInterruptor() {
        if (this.encendida) {
            console.log("La lámpara ya está encendida");
        } else {
            console.log("La lámpara está apagada. Encendiendo...");
            this.encendida = true;
        }
    }
}

const lampara = new Lampara();
lampara.presionarInterruptor();
lampara.presionarInterruptor();
```

Ejemplo de código en JavaScript sin aplicar el Patrón State

Aquí se crean dos clases: una para cuando la lámpara está encendida y otra para cuando está apagada. La idea es que el comportamiento de la lámpara cambia según su estado actual. Por ejemplo, si está apagada y se presiona el interruptor, se enciende. Pero si ya está encendida y se vuelve a presionar, solo muestra un mensaje diciendo que ya está encendida. Este enfoque es útil cuando hay varios estados y cada uno debe comportarse de forma distinta.

Luego hay otro ejemplo de lámpara, pero más simple, que no usa ese patrón. Solo tiene una variable **encendida** y con un **if** decide qué hacer. Este método está bien si solo hay dos estados y el comportamiento es muy sencillo, pero si luego se quiere agregar más funciones o modos a la lámpara, el código se puede volver un poco enredado.

Patrón observador

¿Qué es el patrón observador?

El patrón Observador es una solución común en desarrollo de software cuando se necesita establecer una relación de dependencia entre múltiples objetos. Su funcionamiento se basa en que un objeto central, denominado "sujeto", mantiene un registro de otros objetos llamados "observadores", los cuales dependen de él. Cuando el estado del sujeto cambia, automáticamente se notifica a todos los observadores para que puedan reaccionar y actualizarse en consecuencia. Esta dinámica resulta particularmente útil para la programación orientada a eventos, donde los cambios deben propagarse de forma inmediata a múltiples componentes sin necesidad de una intervención directa. Además, el patrón Observador es comúnmente empleado en interfaces gráficas de usuario (GUI), así como en sistemas de mensajería distribuidos como Kafka, donde se requiere que varios elementos respondan ante un mismo evento.
El patrón Observador facilita una separación clara entre el objeto principal (sujeto) y aquellos que dependen de él (observadores), permitiendo delimitar con precisión los distintos módulos del software. Esto contribuye a mejorar tanto la mantenibilidad como la posibilidad de reutilizar componentes en diferentes contextos. Gracias a este enfoque, es posible informar automáticamente a los objetos interesados cuando el estado del sujeto cambia, fomentando al mismo tiempo una colaboración eficiente, un uso sencillo y un acoplamiento más flexible entre los elementos del sistema.

Entre los beneficios clave de este patrón se encuentra el *desacoplamiento*, ya que el sujeto no necesita conocer ni la cantidad ni la identidad de los observadores que lo siguen. Esto genera una relación *flexible*, en la que ambos pueden evolucionar o modificarse de forma independiente. Además, el patrón facilita el manejo de eventos, ya que cada vez que el sujeto experimenta un cambio, todos los observadores registrados reciben una notificación de forma automática. También

permite establecer relaciones dinámicas, pues los observadores pueden añadirse o eliminarse en tiempo de ejecución sin interferir con el funcionamiento del sujeto.

Analogía en el mundo real

Imagina que trabajas en una aplicación meteorológica. La estación meteorológica (Sujeto) recopila datos como la temperatura y la humedad. Tienes varias pantallas (Observadores) que deberían actualizarse automáticamente cuando cambien los datos meteorológicos. En lugar de que las pantallas revisen constantemente si hay nuevos datos, la estación meteorológica puede simplemente notificarles cuando haya un cambio.

Representación del diagrama UML:

- Subject = Estación Meteorológica
- ConcreteSubject = Estación concreta con temperatura y humedad
- Observer = Interfaz que define actualizar()
- ConcreteObserver = Pantalla que muestra los datos.

Cuando aplicar el patrón observador en .Net

El patrón de diseño observador es especialmente útil en escenarios donde se utilizan notificaciones distribuidas mediante el enfoque push, ya que permite mantener una clara separación entre componentes distintos de una aplicación,

como por ejemplo entre la capa que maneja la lógica del negocio (fuente de datos) y la capa encargada de mostrar la información al usuario (interfaz gráfica). Este patrón es aplicable cuando un proveedor desea enviar actualizaciones a sus clientes mediante devoluciones de llamada.

Para llevar a cabo este patrón, se deben considerar los siguientes elementos:

- El proveedor o sujeto: Es el objeto responsable de emitir notificaciones a los observadores. Esta entidad debe implementar la interfaz IObservable<T>, lo que implica definir el método Subscribe, utilizado por los observadores interesados en recibir actualizaciones.
- El observador: Es el objeto que se suscribe al proveedor para recibir notificaciones. Debe implementar la interfaz IObserver<T>, la cual requiere definir tres métodos:
    - ☐ OnNext: recibe nueva información o actualizaciones.
    - ☐ OnError: se ejecuta cuando ocurre un error en el proveedor.
    - ☐ OnCompleted: se llama cuando el proveedor ha finalizado el envío de notificaciones.
- Un sistema para gestionar los observadores registrados: Normalmente, el proveedor utiliza una colección como List<T> para almacenar las referencias a todos los observadores que se han suscrito. Esto permite administrar múltiples suscriptores de manera eficiente. No se garantiza un orden específico en la entrega de notificaciones, por lo que el proveedor puede determinar libremente dicha orden.
- Una implementación de IDisposable: Esta sirve para permitir que los observadores cancelen su suscripción cuando lo deseen. Al suscribirse, los observadores reciben un objeto que implementa IDisposable, lo cual les permite invocar Dispose para anular su suscripción antes de que el proveedor finalice el proceso de notificación.
- El objeto de datos que se transmite desde el proveedor a los observadores. El tipo de este objeto corresponde con el parámetro genérico T en las interfaces IObservable<T> e IObserver<T>. Aunque en algunos casos puede coincidir con el proveedor, lo más común es que se trate de una clase o estructura distinta encargada de representar la información que se va a compartir.

Tipos de observadores y su diseño

1. Observadores Lineales:

Estos observadores se basan en la suposición de que el sistema tiene un comportamiento lineal o que puede aproximarse a uno alrededor de un punto de operación. Un ejemplo clásico es el filtro de Kalman, que combina una predicción del modelo del sistema con una corrección basada en mediciones. También están el observador de Luenberger, que utiliza una ganancia constante, y el observador de punto muerto, que logra una estimación precisa en un número finito de pasos. Son útiles cuando se conoce el modelo del sistema y las características del ruido.

2. Observadores No Lineales

Están diseñados para sistemas que presentan dinámicas no lineales o donde la aproximación lineal no es suficiente. Algunos, como el filtro de Kalman extendido o el filtro sin perfume, adaptan los métodos lineales a situaciones no lineales. Otros, como el observador de modo deslizante y el de alta ganancia, usan técnicas específicas del análisis no lineal para mejorar la precisión, especialmente cuando hay incertidumbre o perturbaciones.

3. Observadores Adaptativos

Se aplican cuando hay parámetros del sistema que no se conocen con exactitud o que cambian con el tiempo. Estos observadores ajustan sus cálculos mediante leyes adaptativas, utilizando métodos como la teoría de Lyapunov, descenso de gradiente o mínimos cuadrados. Permiten mejorar la estimación del estado y contribuyen a la identificación del sistema, aunque requieren ciertas condiciones para ser efectivos.

4. Observadores Distribuidos

Ideales para sistemas compuestos por varios subsistemas interconectados. Cada observador local realiza estimaciones con su información y se comunica con otros para lograr una estimación global. Son escalables y resistentes a fallos, pero enfrentan retos como la sincronización de datos, retrasos en la comunicación y la estructura de la red. Ejemplos incluyen el filtro de Kalman distribuido y el filtro de partículas distribuidas.

5. Métodos de Diseño de Observadores

Estos métodos permiten definir parámetros que permiten adaptativas según ciertos criterios. Se dividen en dos tipos: los analíticos, que usan herramientas matemáticas como la ecuación de Riccati o la matriz de observabilidad, y los numéricos, que emplean algoritmos de optimización, simulaciones o modelos de aprendizaje automático. La elección depende del sistema y los datos disponibles.

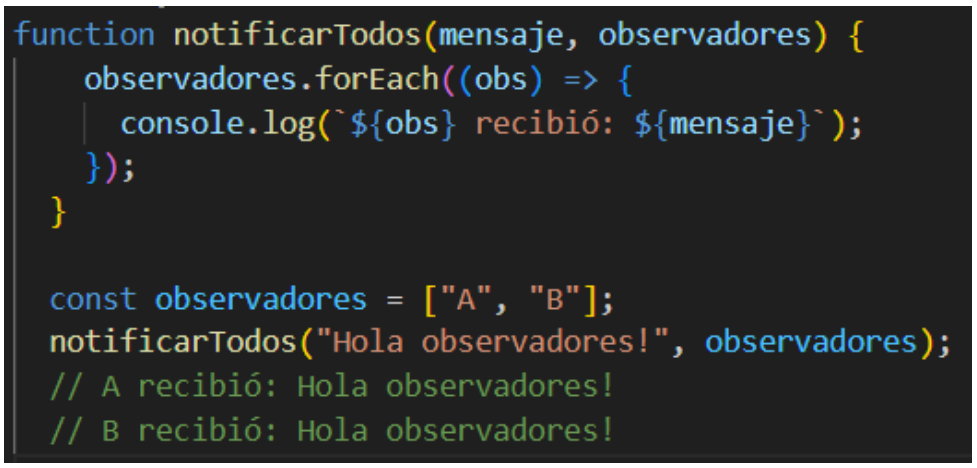
6. Consejos para la Implementación de Observadores

Antes de aplicar un observador, es fundamental validar el modelo del sistema y elegir el tipo de observador adecuado. También es importante ajustar sus parámetros y probarlo en diferentes condiciones para asegurarse de que funcione correctamente. Evaluar su rendimiento con métricas como el error de estimación o la velocidad de convergencia permite hacer mejoras según sea necesario.

Ejemplos y Comparación

```
class Sujeto {
    constructor() {
      this.observadores = [];
    }

    suscribir(obs) {
      this.observadores.push(obs);
    }

    notificar(mensaje) {
      this.observadores.forEach((obs) => obs.actualizar(mensaje));
    }
  }

  class Observador {
    constructor(nombre) {
      this.nombre = nombre;
    }

    actualizar(mensaje) {
      console.log(`${this.nombre} recibió: ${mensaje}`);
    }
  }

  const sujeto = new Sujeto();
  const obs1 = new Observador("A");
  const obs2 = new Observador("B");

  sujeto.suscribir(obs1);
  sujeto.suscribir(obs2);

  sujeto.notificar("Hola observadores!");
  // A recibió: Hola observadores!
  // B recibió: Hola observadores!
```

Ejemplo de un código en JavaScript en el que se aplica el Patrón Observador



```
function notificarTodos(mensaje, observadores) {
    observadores.forEach((obs) => {
      console.log(`${obs} recibió: ${mensaje}`);
    });
  }

  const observadores = ["A", "B"];
  notificarTodos("Hola observadores!", observadores);
  // A recibió: Hola observadores!
  // B recibió: Hola observadores!
```

Ejemplo de código en JavaScript que sin aplicar el Patrón Observador

Aquí hay un objeto principal llamado **Sujeto** que guarda una lista de observadores. Cuando el sujeto quiere avisar algo, les manda un mensaje a todos los que están suscritos. En este caso, los observadores son como oyentes que reciben el mensaje y reaccionan. Es muy útil cuando quieres que varios objetos se enteren automáticamente de que algo ha pasado, como una notificación o un cambio.

Después, hay un ejemplo más básico que trata de hacer lo mismo, pero sin clases. Solo hay una lista de nombres y una función que recorre esa lista mostrando un mensaje. Aunque parece similar, no es un verdadero patrón observador porque no hay conexión entre los elementos. Es solo una función que imprime mensajes. No se puede personalizar el comportamiento de cada observador, y no tienen su propia lógica.

Conclusión

Al finalizar esta investigación acerca de ambos patrones, pude ver la importancia de utilizar patrones de diseños a la hora de programar y desarrollar software. Ambos patrones son útiles porque nos permiten estructurar nuestro código de manera ordenada y mantenible. Dominar no sólo estos dos patrones, si no incluyendo otros, nos es esencial a nosotros como programadores porque nos ayuda a mantener nuestro código limpio, reutilizable y es una buena práctica.

- ***Richard Zhang***

La información presentada en este trabajo investigativo sobre el patrón de state y patrón observador, son herramientas teóricas, son de igual herramientas prácticas que nos ayudan en problemas que se presentan en los desarrollos de software, comprenden una estructura, ventajas y desafíos. Nos permiten tomar mejores decisiones de manera acertada para que las aplicaciones sean más escalables, mantenibles y organizadas. En el desarrollo del trabajo nos permitió conocer y la manera correcta de aplicar los patrones de diseño y esto puede marcar una gran diferencia en la calidad del software que se desarrolla.

- ***Uzziel Aparicio***

Al concluir esta investigación sobre el Patrón State y el Patrón Observador, se comprendió cómo ambos patrones contribuyen significativamente a mejorar la estructura del código, facilitando su mantenimiento y escalabilidad a largo plazo. Aunque existen muchos otros patrones de diseño, estos dos representan herramientas muy útiles que, cuando se aplican correctamente, pueden elevar la calidad del software, haciéndolo más organizado, limpio y profesional.

- ***Rafael Chung***

Bibliografía

1. Cookies, D. (2024, 16 noviembre). Observer Design Pattern: A Complete Guide with Examples. Medium. https://devcookies.medium.com/observer-design-pattern-a-complete-guide-with-examples-ec40648749ff
2. Chaudhary, R. (2024, 19 noviembre). The Observer Design Pattern: From Basics to Advanced. Medium. https://medium.com/@chaudharyritesh947/the-observer-design-pattern-from-basics-to-advanced-9ec9ef737fa1
3. Sarah. (2023, 16 octubre). What is Observer Design Pattern: A Brief Guide with a Case Study. eLuminous Technologies. https://eluminoustechnologies.com/blog/observer-design-pattern-case-study/
4. IEvangelist. (n.d.). Modelo de diseño de observador - .NET. Microsoft Learn. https://learn.microsoft.com/es-es/dotnet/standard/events/observer-design-pattern
5. ¿Cuáles son las técnicas más efectivas para el diseño del observador y la estimación del estado en los sistemas de control? (n.d.). https://www.linkedin.com/advice/1/what-most-effective-techniques-observer-design?lang=es&originalSubdomain=es
6. Refactoring Guru. (s.f.). *Patrón de diseño State*. Refactoring Guru. https://refactoring.guru/es/design-patterns/state
7. Reactive Programming. (s.f.). *Patrón de diseño: State*. ReactiveProgramming.io. https://reactiveprogramming.io/blog/es/patrones-de-diseno/state
8. The PowerUps Learning. (s.f.). *Patrón de diseño: State Pattern*. The PowerUps Learning. https://thepowerups-learning.com/patrones-de-diseno-state-pattern/
9. J. Landa. (s.f.). *Patrón de diseño State*. FcoJlanda.me. https://fcojlanda.me/es/sin-categoria-es/patron-de-diseno-state/
10. Pharokeepers. (2019, 2 agosto). *State Design Pattern*. PharoKeepers. https://pharokeepers.github.io/pharo/2019/08/02/State-Design-Pattern.html
11. NullPointer Excelsior. (s.f.). *State Machine Design Pattern*. NullPointer Excelsior. https://nullpointer-excelsior.github.io/posts/state-machine-design-pattern/
12. Kundupoglu, A. T. (s.f.). *Mastering the State Design Pattern in Java: A Detailed Guide with Examples*. Medium. https://medium.com/@ahmettemelkundupoglu/mastering-the-state-design-pattern-in-java-a-detailed-guide-with-examples-4544a975a914
13. Codeando Simple. (s.f.). *Patrón de diseño: State*. CodeandoSimple. https://codeandosimple.com/design-patterns-state.html