Universidad de los Andes
DPOO
Guillermo José Hernández Estupiñán | 202220866
Taller 5 - Patrones

Patrón escogido: Iterador

Link del repositorio: https://github.com/VanHakobyan/DesignPatterns

1. Información general del proyecto seleccionado

El proyecto que seleccioné para realizar este taller hace parte del repositorio "DesignPatterns" creado por Vanik Hakobyan. En este caso, este repositorio se enfocó en la recreación e implementación de varios tipos de estructuras y algoritmos de datos en el lenguaje C. El propósito de este repositorio es básicamente educativo, todas las implementaciones son código abierto y fue creado para enseñar como funciona cada tipo de patrón de diseño. El link del repositorio es: https://github.com/VanHakobyan/DesignPatterns. El repositorio recrea un total de 23 patrones de diseño, todos terminados e implementados en distintos ejemplos.

Como bien indica en la descripción del repositorio, el autor se basó en "Elements of Reusable Object-Oriented Software". Este es un libro de ingeniería de software que describe patrones de diseño de software. Los autores del libro son Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, con un prólogo de Grady Booch. El libro está dividido en dos partes, siendo los dos primeros capítulos una exploración de las capacidades y desafíos de la programación orientada a objetos, mientras que los capítulos restantes describen 23 patrones clásicos de diseño de software.

En cuanto al diseño, me parece que el principal reto radica en la extensa documentación asociada a cada clase, así como en la meticulosa planificación de la estructura de datos y algoritmos dentro de la librería. Se percibe claramente que el código ha pasado por rigurosos controles de calidad. Además, la distribución de cada clase en diversas carpetas facilita la localización eficiente de la información relevante, agilizando el proceso. En cuanto a la asignación de responsabilidades y las colaboraciones, no creo que hayan sido problemas significativos, ya que, como mencioné anteriormente, las clases tienden a operar de forma independiente, con pocas interrelaciones en general.

2. Explicación clara del patrón, sus elementos y la motivación para su uso en la resolución de problemas

Para darle una explicación a este patrón, imaginemos que tienes una colección de elementos, como una lista de números, y quieres recorrer cada elemento de esa lista. El patrón de diseño Iterador es como tener un "puntero" que te ayuda a recorrer los elementos de la colección sin necesidad de conocer los detalles internos de cómo se almacenan esos elementos.

Básicamente, el patrón iterador es como un guía que te permite ir uno por uno a través de los elementos de una colección, sin que necesites preocuparte por la estructura interna de esa colección. Esto hace que el código sea más flexible porque puedes cambiar la implementación de la colección sin afectar el código que usa el Iterador.

Para ponerlo en contexto con la Programación Orientada a Objetos (POO), podríamos tener una interfaz Iterador que define métodos como siguiente(), haySiguiente(), y obtenerElemento(). Luego, cada tipo de colección (como una lista o un conjunto) podría tener su propio Iterador que implementa estos métodos de acuerdo con la forma en que la colección almacena sus elementos.

Así, cuando quieras recorrer una colección, simplemente obtienes su Iterador y usas los métodos estándar sin preocuparte por los detalles internos de la colección. Este enfoque hace que el código sea más modular y fácil de mantener, ya que puedes cambiar la implementación de la colección sin afectar el código que utiliza el Iterador.

La motivación que tuve para usar este patrón fue debido a la flexibilidad y la simplificación del código. Imagina que en el futuro decides cambiar la manera en que se almacenan esos datos. Si estás utilizando el patrón Iterador, puedes hacer esa modificación sin tener que reescribir todo el código que interactúa con esos datos. Solo necesitas ajustar la implementación del Iterador según la nueva estructura de datos.

Además, el Iterador proporciona una interfaz común para recorrer datos, lo que hace que tu código sea más comprensible y mantenible. Otros desarrolladores que trabajen en el proyecto podrán

entender fácilmente cómo acceder a los elementos de la colección sin tener que sumergirse en detalles complicados.

3. Cómo se utiliza el patrón en el proyecto

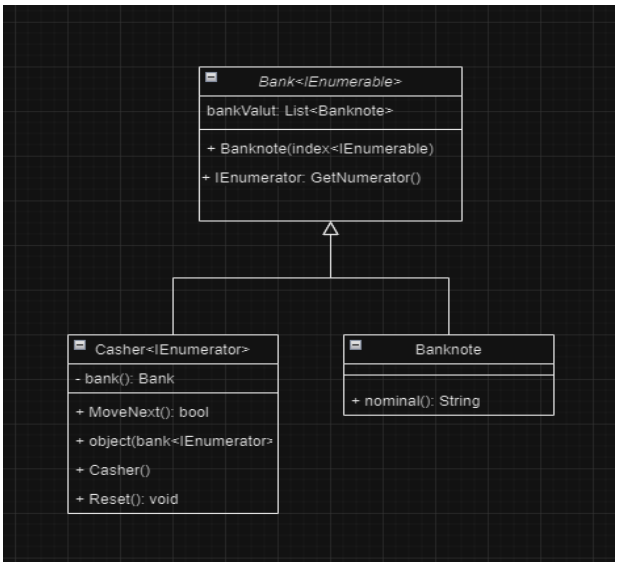
```
internal class Bank : IEnumerable
{
    List<Banknote> bankValut = new List<Banknote> {
        new Banknote(),
        new Banknote(),
        new Banknote(),
        new Banknote()
    };
    public Banknote this[int index]
    {
        get { return bankValut[index]; }
        set { bankValut.Insert(index, value); }
    }
    public IEnumerator GetEnumerator()
    {
        return new Casher(this);
    }
    public int count
    {
        get { return bankValut.Count; }
    }
}
```

```
internal class Casher : IEnumerator
{
    private Bank bank;
    int current = -1;
    public Casher(Bank bank)
    {
        this.bank = bank;
    }

    public object Current
    {
        get { return bank[current]; }
    }

    public bool MoveNext()
    {
        if (current < bank.count - 1)
        {
            current++;
            return true;
        }
        return false;
    }

    public void Reset()
    {
        current = -1;
    }
}
```

En este proyecto, se utiliza el patrón Iterador para recorrer los elementos de la clase Bank que contiene una colección de objetos Banknote (billetes). La idea es proporcionar una forma estandarizada de recorrer los billetes en el banco sin exponer los detalles internos de la implementación de la colección.

El patrón Iterador permite recorrer la colección de billetes en el banco de manera sencilla y sin tener que conocer los detalles internos de la implementación de la clase Bank. Esto hace que el código sea más flexible y fácil de entender, ya que se sigue un enfoque estándar para recorrer la colección.

4. Información y estructura del fragmento del proyecto donde aparece el patrón.



En la estructura del fragmento podemos identificar la presencia del patrón iterador, debido a que destaca la creación de una lista de Banknote dentro de la clase Bank y la definición de un indexador (this[int index]) que permite acceder a los elementos de la lista. Además, el método GetEnumerator() de Bank devuelve una instancia de la clase Casher, la cual implementa la interfaz IEnumerator.

Los elementos clave del patrón iterador son más evidentes en la clase Casher, que actúa como el iterador. Implementa la interfaz IEnumerator y contiene métodos esenciales como Current, MoveNext(), y Reset(). Estos métodos son fundamentales para el control del flujo durante la iteración sobre la colección de Banknote.

5. Ventajas de usar el patrón en ese punto del proyecto

En este proyecto, el uso del patrón iterador ofrece diversas ventajas que mejoran la organización y flexibilidad del código. En primer lugar, la abstracción del recorrido de la colección de instancias de Banknote en la clase Bank permite que el cliente, en este caso, el código dentro del método Main, no tenga que preocuparse por los detalles internos de almacenamiento o implementación del recorrido. Esto favorece la encapsulación y facilita futuras modificaciones en la estructura interna de la colección sin afectar al código del cliente.

La separación de responsabilidades es otro punto clave. Mientras la clase Casher se encarga del recorrido de la colección, la clase Bank se centra en la gestión y provisión de acceso a las instancias de Banknote. Esta división sigue el principio de "separación de preocupaciones", simplificando el mantenimiento y la comprensión del código.

La extensibilidad del código se ve favorecida por esta implementación. Si en el futuro se decide cambiar la estructura interna de la colección o proporcionar diferentes formas de recorrer los elementos, esto se puede lograr ajustando la implementación del iterador (Casher) sin afectar al código del cliente.

El patrón iterador ofrece una manera eficiente de recorrer colecciones mientras mantiene un bajo acoplamiento entre las clases, lo que facilita la extensibilidad y el mantenimiento del código.

6. Desventajas o limitaciones de usar el patrón en ese punto del proyecto

Aunque el uso del patrón iterador en este proyecto ofrece ventajas significativas, también hay algunas consideraciones importantes en términos de desventajas. En primer lugar, la implementación de clases como Casher, IEnumerable, e IEnumerator, puede introducir complejidad adicional en el código. En proyectos más pequeños o simples, esta complejidad podría parecer innecesaria y dificultar la comprensión del código.

Además, existe la preocupación de una posible sobrecarga de memoria y rendimiento. La introducción de clases e interfaces adicionales puede tener un impacto en los recursos del sistema, lo que podría ser una consideración crítica en proyectos donde la eficiencia es esencial.

También, podría existir el caso en el cual el uso del patrón iterador puede resultar menos eficiente en comparación con enfoques más directos de iteración, especialmente si la colección se recorre de manera predecible y repetitiva. La introducción de un iterador podría considerarse como un exceso en estas situaciones.

Otra consideración importante es la intuitividad del código, especialmente para desarrolladores nuevos en el proyecto. El patrón iterador puede no ser tan intuitivo para aquellos que no están familiarizados con él, lo que podría afectar la facilidad de mantenimiento del código, especialmente en equipos con cambios de personal.