Nicolás Ruiz Pérez

202123608

Taller 5: Patrones

Patrón: Singleton

Caso de estudio: Library Management System -Java [1]

Descripción del patrón:

Singleton es un patrón de diseño en programación que se utiliza para asegurar que una clase tenga una única instancia y proporcionar un punto de acceso global a ella.

El objetivo principal del patrón Singleton es restringir la creación de objetos de una clase a una sola instancia y garantizar que haya un único punto de acceso para acceder a esa instancia en todo el programa. Esto puede ser útil en situaciones donde necesitas tener exactamente una instancia de una clase para coordinar acciones en todo el sistema [2].

La implementación típica de Singleton implica lo siguiente:

1. Un constructor privado: El constructor de la clase Singleton debe ser privado para evitar que se pueda crear una instancia directamente desde fuera de la clase.
2. Una instancia estática privada: La clase Singleton debe tener una variable estática privada que almacene la única instancia de la clase. Esta variable se inicializa dentro de la propia clase.
3. Un método estático público de acceso: La clase Singleton debe proporcionar un método estático público que devuelva la instancia única de la clase. Este método actúa como el punto de acceso global para obtener la instancia.

Es importante tener en cuenta que esta implementación no es thread-safe, lo que significa que, si múltiples hilos intentan acceder a la instancia al mismo tiempo, podrían crearse múltiples instancias. Para garantizar la concurrencia segura en entornos multi-hilo, es necesario agregar sincronización al método ‘getInstance()’ o utilizar otras técnicas como el uso de inicialización estática o implementaciones basadas en enum.

El patrón Singleton puede ser útil en diversas situaciones, como el acceso a recursos compartidos, la gestión de conexiones a bases de datos o servicios externos, el registro centralizado de eventos o configuraciones, entre otros casos en los que se requiera una única instancia de una clase en todo el programa. Sin embargo, también es importante tener en cuenta que el uso excesivo de Singletons puede dificultar la modularidad y el testeo unitario, por lo que es recomendable evaluar cuidadosamente su implementación en cada caso específico.

La inicialización típica del patrón en Java es la siguiente:

```java
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    // El constructor privado no permite que se genere un constructor por defecto.
    // (con mismo modificador de acceso que la definición de la clase)
    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Imagen 1 Implementación típica de Singleton en Java

Descripción del proyecto:

El repositorio de GitHub "Library Management System - Java" es un sistema de gestión de bibliotecas implementado utilizando los conceptos de Análisis y Diseño Orientado a Objetos. El código está escrito de forma minimalista en la interfaz gráfica de usuario (GUI), y las entidades están desacopladas. La interfaz es basada en consola. [1]

En el repositorio se proporciona el Diagrama de Clases del proyecto (imagen 2), así como el archivo del esquema de la base de datos.

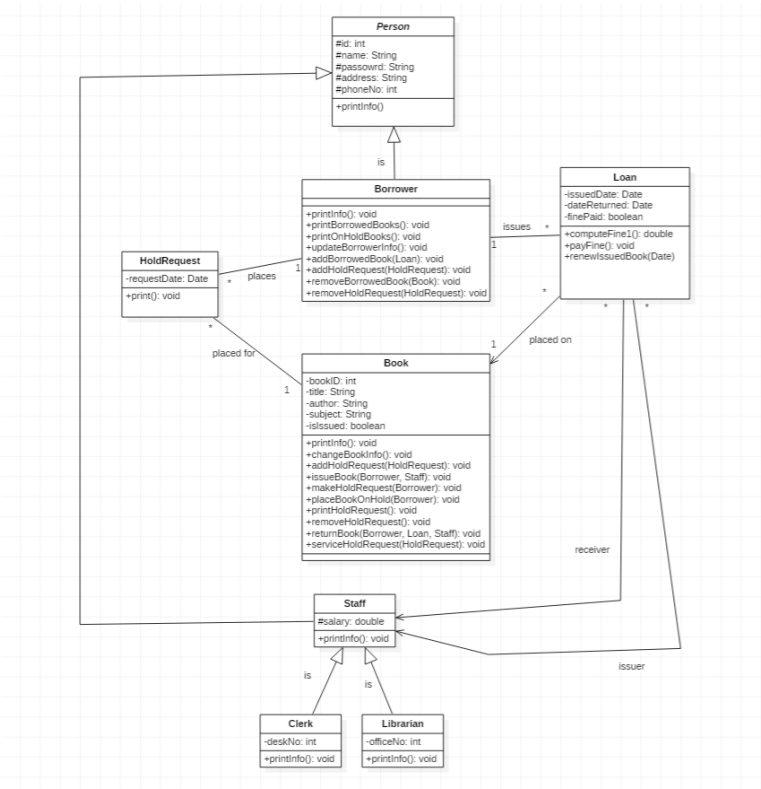

Imagen 2 UML del repositorio [1]

Como se puede observar en la imagen 2, la implementación solo consta de ocho clases de bajo nivel, sin embargo, es claro que hay clases que poseen muchas responsabilidades con respecto al resto, esto es una consecuencia del uso de Singleton, peso esto se discutirá más tarde.

Los actores involucrados en el sistema son los siguientes:

1. Bibliotecario (Librarian)

2. Empleado de Préstamo (Checkout Clerk)

3. Prestatario (Borrower)

4. Administrador (Administrator)

Casos de Uso:

Después de determinar los actores, se identifican las tareas que cada actor debe realizar en el sistema. Cada tarea se denomina caso de uso, ya que representa una forma específica en que el sistema se utilizará.

A continuación, se enumeran algunos de los casos de uso para cada actor:

Prestatario (Borrower):

- Buscar elementos por título.

- Buscar elementos por autor.

- Buscar elementos por tema.

- Reservar un libro si está en préstamo a otra persona.

- Ver la información personal del prestatario y la lista de libros actualmente prestados.

Empleado de Préstamo (Checkout Clerk):

- Todos los casos de uso del prestatario, además de:

- Registrar el préstamo de un elemento para un prestatario.

- Registrar la devolución de un elemento.

- Renovar un elemento.

- Registrar el pago de una multa.

- Agregar un nuevo prestatario.

- Actualizar la información personal de un prestatario (dirección, número de teléfono, etc.).

Bibliotecario (Librarian):

- Todos los casos de uso del prestatario y el empleado de préstamo, además de:
- Agregar un nuevo elemento a la colección.
- Eliminar un elemento de la colección.
- Modificar la información registrada sobre un elemento.

Administrador (Administrator):

- Agregar un empleado de préstamo.
- Agregar un bibliotecario.
- Ver el historial de libros prestados.
- Ver todos los libros en la biblioteca.

Este tipo de proyectos implican algunos retos de implementación, como su escalabilidad, pues la cantidad de libro es cada vez mayor, y puede implicar la creación de nuevas funcionalidades, además de que pueda manejar eficientemente grandes volúmenes de datos. Otro reto podría relacionarse con mantener una base de datos segura, ya que los textos que puede haber en una biblioteca pueden ser sensibles o incluso privados, además de tener una base de datos segura.

Uso del patrón en el código:

El patrón Singleton se utiliza en una de las clases de alto nivel de la aplicación.

Por ejemplo, en la clase *Library* se puede encontrar este código:

```
37          /*----Following Singleton Design Pattern (Lazy Instantiation)------------*/
38          private static Library obj;
39
40          public static Library getInstance()
41          {
42              if(obj==null)
43              {
44                  obj = new Library();
45              }
46
47              return obj;
48          }
49          /*----------------------------------------------------------------------*/
50
51          private Library()   // default cons.
52          {
53              name = null;
54              librarian = null;
55              persons = new ArrayList();
56
57              booksInLibrary = new ArrayList();
58              loans = new ArrayList();
59          }
60
```

Imagen 3 Código de la clase Library del proyecto

En la clase Library, se utiliza el patrón Singleton para garantizar que solo haya una instancia de la clase Library en todo el programa. Esto se logra mediante el uso de un método *getInstance()* y una variable estática *obj* que almacena la única instancia de la clase.

El método *getInstance()* verifica si la variable *obj* es nula. Si es nula, crea una nueva instancia de la clase Library y la asigna a *obj*. Si no es nula, simplemente devuelve la instancia existente almacenada en *obj*. De esta manera, se garantiza que solo se cree una instancia de la clase Library y se proporciona un punto de acceso global a esa instancia a través del método *getInstance()*.

En este caso particular, el patrón Singleton se utiliza para garantizar que solo exista una instancia de la clase Library y proporcionar un punto de acceso global a esa instancia. Sin embargo, el código proporcionado no implementa correctamente el patrón Singleton. Para solucionar los problemas y mejorar el diseño, se pueden considerar las siguientes alternativas:

Inicialización estática: En lugar de usar el enfoque de "Lazy Instantiation" actual, se puede utilizar la inicialización estática para garantizar la creación única y temprana de la instancia de Library. Esto se podría lograr definiendo el objeto *obj* como estático y asignándole una instancia de Library directamente en su declaración. Al hacerlo, se asegura que la instancia única se cree en el momento de la carga de la clase, antes de que se acceda a ella por primera vez. Un código de la estructura mencionada es el siguiente:

private static Library obj = new Library();

public static Library getInstance() {return obj;}

private Library() {...}

Separación de responsabilidades: El código de la clase Library parece tener demasiadas responsabilidades, como la gestión de personas, libros, préstamos, búsquedas, etc. Sería posible separar estas responsabilidades en diferentes clases y seguir los principios de diseño sólidos, como el Principio de Responsabilidad Única (SRP) y el Principio de Abierto/Cerrado (OCP). Por ejemplo, se pueden crear clases separadas para la gestión de personas, gestión de libros, gestión de préstamos, etc.

Inyección de dependencias: En lugar de utilizar el patrón Singleton para tener acceso global a la instancia de Library, se puede considerar utilizar la inyección de dependencias. Esto implica que las clases que necesiten acceder a la instancia de Library la reciban como parámetro en su constructor, en lugar de obtenerla directamente desde un método estático. Esto permite una mejor separación de responsabilidades y facilita la prueba unitaria de las clases.

Aún así, el haber usado este patrón en esta parte del proyecto implica las siguientes ventajas:

Acceso controlado a una única instancia: El patrón Singleton garantiza que solo exista una única instancia de la clase Library en todo el sistema. Esto puede ser útil en situaciones donde se requiere tener acceso a una única instancia globalmente disponible, como en el caso de una biblioteca o un recurso compartidos.

Fácil acceso desde múltiples puntos del código: Al utilizar el patrón Singleton, la instancia de *Library* se puede acceder fácilmente desde cualquier parte del código sin tener que pasar explícitamente una referencia o crear múltiples instancias. Esto simplifica la forma en que se accede y utiliza la instancia en diferentes partes del proyecto.

Fácil extensibilidad y reutilización: El uso del patrón Singleton facilita la extensibilidad y reutilización de la instancia de Library. Como solo hay una única instancia, cualquier modificación o mejora realizada en la implementación de Library se reflejará automáticamente en todas las partes del código que la utilizan.

Sin embargo, hay algunas desventajas de implementar este patrón en el código:

Acoplamiento fuerte: El patrón Singleton introduce un acoplamiento fuerte entre las clases que utilizan la instancia de Library y la propia clase Library. Esto puede dificultar la modificación o reemplazo de la implementación de Library en el futuro, ya que todas las clases dependen directamente de ella.

Dificultad en pruebas unitarias: Debido al acoplamiento mencionado anteriormente, las pruebas unitarias de las clases que utilizan la instancia de Library pueden volverse más complicadas. No es posible aislar fácilmente esas clases para probar su funcionalidad individualmente, lo que puede dificultar la detección y corrección de errores.

Dificultad en la escalabilidad: El patrón Singleton limita la escalabilidad del sistema, ya que solo permite una única instancia de Library. Si en el futuro se requiere utilizar múltiples instancias de Library (por ejemplo, para manejar diferentes bases de datos o configuraciones), el patrón Singleton no es una opción adecuada.

Referencias:

[1] Muneer, H & Amjad, M. Library Management System -Java. Disponible en: https://github.com/OSSpk/Library-Management-System-JAVA

[2] «Singleton design pattern». *HowToDoInJava*. 22 de octubre de 2012. Consultado el 10 de julio de 2020. Disponible en: https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/