**Universidad de Las Américas**

Facultad de Ingenierías y Ciencias Agropecuarias

*Ingeniería de Softwate*

**Informe de laboratorio**

1. **DATOS DEL ALUMNO: Camila Cabrera**
2. **TEMA DE LA PRÁCTICA:** Implementación Patrón Estructural Abstract Factory y Proxy
3. **OBJETIVO DE LA PRÁCITCA**
4. Realizar el ejercicio propuesto de implementación de los Patrónes Abstract Factory y Proxy para mejorar la comprensión de su aplicación.
5. **OBJETIVOS ESPECÍFICOS**
   * Leer y comprender los principios y características de los patrones Abstract Factory y Proxy

* Inicializar Visual Studio Code para proyectos de java
* Crear un proyecto de java en visual Studio code para Abstract Factory
* Crear un proyecto de java en visual Studio code para Proxy
* Implementar la documentación para el proyecto ejemplo del patrón Abstract Factory
* Implementar la documentación para el proyecto ejemplo del patrón Proxy

1. **MATERIALES Y MÉTODOS**

**Materiales:**

* Visual studio code
* JDK 17
* Guía práctica de Patrón Proxy
* Guía práctica de Patrón Abstract Factory

**Métodos:**

1. Validación del JDK 17.
2. Configuración de Visual Studio Code para proyectos de Java.
3. Creación del proyecto Java sin herramientas de construcción del proyecto.
4. Implementación de la guía práctica del patrón Adapter Factory
5. Registro de hallazgos durante el proceso de implementación de la documentación.
6. Implementación de la guía práctica del patrón Proxy
7. Registro de hallazgos durante el proceso de implementación de la documentación.
8. **DESARROLLO DE LA PRÁCTICA Y RESULTADOS**

**Marco Teórico**

El Patrón Adapter es un patrón de diseño estructural que permite que interfaces incompatibles trabajen juntas. Esto es útil cuando se tienen dos clases con interfaces incompatibles pero se necesita que interactúen entre sí. El patrón Adapter actúa como un intermediario entre estas dos clases, convirtiendo la interfaz de una clase en otra interfaz que el cliente espera.

Adapter proporciona una manera para que dos clases que no pueden trabajar juntas debido a sus interfaces incompatibles puedan colaborar. Esto se logra mediante la creación de una clase Adapter que envuelve a una de las clases, permitiendo que esta clase sea utilizada a través de una interfaz compatible con la otra clase.

El patrón Proxy es un patrón de diseño estructural que proporciona un sustituto o representante de otro objeto para controlar el acceso a este. Se utiliza cuando es necesario que un objeto actúe como intermediario para otro objeto, proporcionando un nivel de indirección entre el cliente y el objeto real.

El propósito principal del patrón Proxy es controlar y gestionar el acceso a un objeto de manera eficiente y segura. Al utilizar un Proxy, se puede realizar operaciones adicionales antes o después de acceder al objeto real, como la verificación de permisos, la carga perezosa (lazy loading), el almacenamiento en caché, el registro de llamadas, etc.

**Desarrollo de la práctica patrón Proxy**

* **Validación del JDK 17:**
  + Para verificar la versión del JDK se comprueba con el comando “java -version” en la terminal del computador.

*Imagen #1: Validación versión JDK*

|  |
| --- |
|  |

*Fuente: Elaboración propia*

* **2. Configuración de Visual Studio Code para proyectos de Java.**
  + En el IDE visual studio code se debe instalar o validar la dependencia para Java (Imagen 2) .

*Imagen #2: Dependencia de Java*

|  |
| --- |
|  |

*Fuente: Elaboración propia*

* **Creación del proyecto Java sin herramientas de construcción del proyecto.**
  + Se inicializo el proyecto de Java escogiendo la opción sin herramientas de construcción.

*Imagen #3: Dependencia de Java*

|  |
| --- |
|  |

**Implementación de la guía práctica del patrón Proxy.**

* Se siguió la guía práctica de Patron Singleton obtenida en el siguiente link: [Adapter (refactoring.guru)](https://refactoring.guru/es/design-patterns/adapter) para la elaboración del código.

**Carpeta Round**

Se crea la carpeta Round dentro del proyecto. En esta carpeta se crean las siguientes clases:

|  |
| --- |
| **RoundHole.** |
|  |
| La clase RoundHole representa un agujero redondo, y su funcionalidad principal es verificar si un objeto de tipo RoundPeg puede encajar dentro de este agujero.  En esta clase se declara el atributo de radio y su método get. Se diseña un método que determina si un objeto Roundpeg puede entrar en un RoundHole. |

|  |
| --- |
| **RoundPeg** |
|  |
| La clase RoundPeg representa un objeto redondo que tiene radio y un método get para obtener el atributo. En el constructor se debe enviar al diámetro del objeto. |

**Carpeta Square**

|  |
| --- |
| SquarePeg |
|  |
| La clase SquarePeg representa un objeto cuadrado que tiene un ancho y su método get. Y un método ‘getSquare()’ calcula y devuelve el área del objeto cuadrado. |

**Adapters**

|  |
| --- |
| SquarePegAdapter |
|  |
| La clase SquarePegAdapter es una clase adaptador que permite que un objeto de tipo SquarePeg (objeto cuadrado) se utilice en lugar de un objeto de tipo RoundPeg (objeto redondo), adaptando sus interfaces incompatibles.  El método que interactúa con ambas interfaces es el Método getRadius, el cual sobrescribe el método getRadius() de la clase padre RoundPeg. Calcula y devuelve el radio que sería equivalente al objeto cuadrado y que encaje en un agujero redondo. Utiliza una fórmula que calcula el radio mínimo de un círculo que puede contener completamente el objeto cuadrado. El cálculo implica obtener la mitad del ancho del pasador cuadrado (peg.getWidth() / 2). |

**Main**

|  |
| --- |
| Main |
|  |
| En el main se usa el patrón adaptar verificando que los objetos calcen (fit) dentro de un agujero redondo.  Primero , se crea un objeto RoundHole que representa un agujero redondo con un radio de 5 unidades.  Se crea un objeto RoundPeg que representa un pasador redondo con un radio de 5 unidades.  Se verifica si el pasador redondo encaja en el agujero redondo utilizando el método fits() de la clase RoundHole. Dado que ambos tienen el mismo radio, el pasador redondo encaja en el agujero redondo.  Se crean dos objetos SquarePeg, uno con un ancho de 2 unidades y otro con un ancho de 20 unidades.  Se intenta utilizar directamente el método fits() de la clase RoundHole con el objeto smallSqPeg, que representa un pasador cuadrado. Sin embargo, esto no compila porque fits() espera un objeto RoundPeg, no un SquarePeg.  Se resuelve el problema utilizando el patrón Adapter. Se crean dos objetos SquarePegAdapter, cada uno envolviendo un objeto SquarePeg respectivamente.  Se verifica si el pasador cuadrado adaptado encaja en el agujero redondo utilizando el método fits() de la clase RoundHole. Dado que el adaptador calcula un radio que permitiría que el pasador cuadrado encaje en el agujero redondo, el pasador cuadrado con un ancho de 2 unidades encaja. |

|  |
| --- |
| **Resultado ejecución** |
|  |

**Implementación de la guía práctica del patrón Proxy.**

**Carpeta: some\_cool\_media\_library**

|  |
| --- |
| ThirdPartyYouTubeLib |
|  |
| Esta interfaz ThirdPartyYouTubeLib define un método para acceder a videos populares y obtener videos individuales a través de su ID. |

|  |
| --- |
| ThirdPartyYouTubeClass |
|  |
| ThirdPartyYouTubeClass implementa la interfaz ThirdPartyYouTubeLib, que proporciona funcionalidades para acceder a videos de YouTube. Primero, se simula una conexión al servidor de YouTube utilizando el método connectToServer(). Luego, se llama al método getRandomVideos() para obtener una colección de videos populares aleatorios y se devuelve esta colección.  El método getVideo(String videoId), implementa la funcionalidad para obtener la simulación de un video específico de YouTube dado su ID. Primero, se construye la URL del video y se simula una conexión al servidor de YouTube utilizando el método connectToServer(). Luego, se llama al método getSomeVideo(videoId) para obtener el video correspondiente y se retorna este video. |

|  |
| --- |
| Video |
|  |
| Esta clase define los atributos de Video, en particular quema la inicialización del atributo data con “Random videos” en el constructor. |

**Carpeta: proxy**

|  |
| --- |
| YouTubeCacheProxy |
|  |
| La clase YouTubeCacheProxy es una clase que implementa el patrón proxy que actúa como intermediario entre el cliente y el servicio de YouTube (ThirdPartyYouTubeLib). Su objetivo principal es proporcionar un caché para almacenar y recuperar videos populares y videos individuales para mejorar la eficiencia y reducir el tiempo de acceso a los videos de YouTube. |

**Carpeta: downloder**

|  |
| --- |
| YouTubeDownloader |
|  |
| La clase YouTubeDownloader proporciona funcionalidades para descargar y renderizar páginas de video desde YouTube utilizando la interfaz realizada. Utiliza los métodos getVideo(String videoId) y popularVideos() de la API para obtener los videos individuales y los videos populares respectivamente, y luego los renderiza en la consola como si fueran páginas HTML. |

|  |
| --- |
| main |
|  |
| La clase YouTubeDownloader proporciona funcionalidades para descargar y renderizar páginas de video desde YouTube utilizando la interfaz realizada. Utiliza los métodos getVideo(String videoId) y popularVideos() de la API para obtener los videos individuales y los videos populares respectivamente, y luego los renderiza en la consola como si fueran páginas HTML. |

1. **OPINIÓN PERSONAL**

* Me pareció interesante la aplicación del patrón proxy ya que se puede trabajar con varios enfoques, pero el ejemplo me indico como usar este patrón en relación al cache y como puede mejorar significativamente la eficiencia y el rendimiento de las aplicaciones que requieren acceder a servicios externos. El ejemplo fue la descarga de videos de YouTube, que en este caso fue una simulación, pero en un entorno real puede resultar muy eficiente. Esto evita múltiples solicitudes redundantes al servidor y almacenar localmente los datos previamente obtenidos, se reduce la carga en la red y se acelera el tiempo de respuesta para los usuarios.
* Considero interesante ver como el patrón proxy actúa como un intermediario que gestiona el acceso a los recursos remotos, esto permite optimizar el uso de recursos tanto locales como remotos. En el ejemplo, al utilizar el caché para almacenar temporalmente los datos descargados, se minimiza la necesidad de acceder repetidamente al servidor para obtener la misma información, lo que reduce el consumo de ancho de banda y la carga en el servidor.
* De los dos patrones explorando el patrón que más me llamo la atención fue el patrón Adapter ya que si me he encontrado con ocasiones en las que posiblemente hubiera necesitado implementar este patrón, pero no sabía que existía o como implementarlo. Creo que es importante porque permite reutilizar componentes existentes que no se diseñaron originalmente para trabajar juntos. Al envolver el componente incompatible con un Adapter que cumple con la interfaz esperada, podemos aprovechar el código existente sin necesidad de modificarlo.
* Algo que pude identificar de estos patrones fue que buscan la modularidad. Al utilizar el patrón Proxy y el patrón Adapter, entendí cómo se puede estructurar el código de manera que los componentes puedan interactuar entre sí de manera transparente, sin necesidad de conocer los detalles de implementación de cada uno. Esto promueve la modularidad, ya que cada componente puede ser desarrollado, probado y mantenido de manera independiente.

1. **ANEXOS**

Link Github del proyecto: [CamilaACT/LaboratorioSingelton (github.com)](https://github.com/CamilaACT/LaboratorioSingelton)

1. **BIBLIOGRAFÍA**

Refactoring guru. (s.f.). *Refactoring guru*. Obtenido de Singleton: https://refactoring.guru/es/design-patterns/singleton