TEXT CLASS REVIEW

TEMAS A TRATAR EN LA CUE

- Maven y su manejo de dependencias.
- La librería Lombok.
- El patrón de diseño de Inyección de dependencias, su utilidad y uso.

MAVEN

Maven es una herramienta potente para la gestión y construcción de proyectos Java, ya integrada en Eclipse. Gestiona el ciclo de vida completo del proyecto: creación, construcción, validación, compilación y tiempo de ejecución. Nos facilita, a través de su repositorio, la descarga automática de las dependencias necesarias para el proyecto, la validación de la funcionalidad de los servicios, ejecutar pruebas unitarias y la configuración del entorno de desarrollo del proyecto, entre otros aspectos.

Maven introduce el concepto de "artefacto" [*artifact*], una abstracción del concepto de código fuente reutilizable, que considera también su arquitectura, es decir, un artefacto contempla también la estructura del código (paquetes y clases), sus dependencias necesarias, sus pruebas unitarias, configuraciones, documentación, etc. Al crear un proyecto Maven, se está creando un artefacto y al agregar dependencias a este, se están agregando otros artefactos al proyecto. Por tanto, también podemos utilizar un artefacto existente como punto de partida de un nuevo artefacto. A estos artefactos que se utilizan como *plantillas* de nuevos artefactos se les llama "arquetipos" [*archetypes*].

Un arquetipo es básicamente un artefacto simple que contiene el *prototipo* de proyecto que se desea crear y poseerá distintas estructuras, dependencias y configuraciones mínimas para facilitar la implementación de su funcionalidad objetivo. Esto permite mayor rapidez al momento de crear nuevos artefactos, que, por su función a concretar, pueden poseer estructuras similares a otros ya creados. En los ejercicios proporcionados para esta sección, nos remitiremos al uso de la arquitectura (arquetipo) más simple que Maven proporciona al crear un proyecto.

MANEJO DE DEPENDENCIAS CON MAVEN

Cuando hablamos de dependencias con respecto a Maven, nos referimos a los distintos artefactos, librerías o *frameworks* de los que hará uso nuestro proyecto, estos son agregados al POM para declarar su uso. El "Project Object Model" o POM es una representación en **XML** del proyecto Maven contenido en el archivo *pom.xml* encontrado en la carpeta raíz del proyecto. Para aprovechar las funcionalidades de instalación automática de dependencias de Maven, debemos agregar las dependencias a utilizar especificando su versión, id de artefacto, id de grupo, scope y otras configuraciones circunstanciales envueltas en la etiqueta `<dependency> </dependency>`. La información sobre el id artefacto, grupo y versión se puede encontrar en el repositorio de Maven. Si no se encuentra en el archivo, se debe hacer una entrada `<dependencies> </dependencies>` dentro de la cual se agregarán las dependencias. En los ejercicios proporcionados para esta sección crearemos un proyecto Maven y configuraremos el POM para agregar las dependencias al proyecto.

LOMBOK

Lombok es una librería para la creación automática de herramientas que permite ahorrar tiempo y código. A través de anotaciones, se instruye para que lombok implemente automáticamente los constructores, mutadores, accesores o el patrón de diseño *builder* de una clase, entre otras funcionalidades. Se puede acceder a una lista completa de las anotaciones aquí. No ahondaremos mucho en sus funcionalidades más nos ayudaremos de algunas de ellas para ahorrar tiempo y código. Veremos su instalación y usos más cotidianos en los ejercicios proporcionados para esta sección.

INYECCIÓN DE DEPENDENCIAS

La inyección de dependencias (u objetos), en términos simples, es un patrón de diseño, en la programación orientada a objetos, en el que se proporcionan los objetos a las clases en vez de que las clases mismas creen los objetos que necesitan. Esto permite evitar el alto acoplamiento entre clases que, a la larga, produce inconvenientes en el mantenimiento de un proyecto, siguiendo así uno de los principios *SOLID* de *inversión de dependencias*. Por tanto, las clases no crean los objetos que necesitan para concretar su funcionalidad, sino que estos son creados y proporcionados por otra clase *contenedora*.

Las inyecciones de dependencias, aunque puedan realizarse a través de clases, se recomienda realizarlas a través de *interfaces*. Una interfaz, conceptualmente, es un *contrato* en el que se

establecen cuáles serán los comportamientos de una clase, es decir, es un archivo en el que se especifican los nombres de métodos, tipos de datos que retornan esos métodos y partir de que parámetros lo harán, pero, no se establece como lograrán estos comportamientos; no se implementan los métodos en la interfaz. Luego, una clase que haga uso de dicho contrato debe implementar estos métodos, siguiendo las reglas establecidas por la interfaz que se está implementando; los métodos de la clase tendrán el mismo nombre, parámetros y tipos de datos de retorno que los establecidos en la interfaz que implementa, pero, la implementación en sí, el código que logra este comportamiento, puede ser distinto para cada clase que implementa una misma interfaz.

La inyección por medio de interfaces, versus por medio de clases, facilita la inyección de distintas dependencias a una misma clase que conoce, a priori, que será inyectada con una dependencia con cierto comportamiento preestablecido por una interfaz.

REPOSITORIO LOCAL DE MAVEN

El repositorio local de Maven es el conjunto donde Maven almacena todos los archivos jar del proyecto, bibliotecas o dependencias. De forma predeterminada, el nombre de la carpeta se establece en '.m2' y, de forma predeterminada, la ubicación es 'Bibliotecas \ Documentos \ .m2'.

REPOSITORIO CENTRAL DE MAVEN

El repositorio central de Maven es la ubicación predeterminada para que Maven descargue todas las bibliotecas de dependencia del proyecto para su uso. Para cualquier biblioteca involucrada en el proyecto, Maven primero busca en la carpeta .m2 del Repositorio local, y si no encuentra la biblioteca necesaria, busca en el Repositorio central y carga la biblioteca en el repositorio local.

EL CICLO DE VIDA Y FASES

El otro concepto fundamental de Maven para la gestión de la construcción de un proyecto es el ciclo de vida. Un ciclo de vida está compuesto por un conjunto de fases. En Maven se definen tres ciclos de vida por defecto y normalmente nunca te vas a ver en la necesidad de definir ninguno adicional. Los tres ciclos de vida de Maven son:

- **El ciclo de vida default**, que gestiona la construcción y despliegue del proyecto.

- **El ciclo de vida clean**, que gestiona la limpieza del directorio del proyecto. Es decir, se encarga de eliminar todos los archivos generados en el proceso de construcción y despliegue.
- **El ciclo de vida site**, que gestiona la creación de la documentación del proyecto.

Las fases que componen cada ciclo de vida también están predefinidas. Por ejemplo, el ciclo de vida default está compuesto por las siguientes fases:

Validate, initialize, generate-sources, process-sources, generate-resources, process-resources, **compile**, process-classes, generate-test-sources, process-test-sources, generate-test-resources, process-test-resources, test-compile, process-test-classes, **test**, prepare-package, **package**, pre-integration-test, integration-test, post-integration-test, verify, **install**, **deploy**.

De entre todas ellas las más relevantes son:

Validate: Que valida que el proyecto es correcto y que está disponible toda la información necesaria.

Compile: Que compila el código fuente del proyecto.

Test: Que prueba el código compilado utilizando algún framework de pruebas unitarias. Estas pruebas no deben requerir que el código esté empaquetado o desplegado.

Package: Coge el código compilado y lo empaqueta en un formato distribuible. Por ejemplo, como un JAR.

Verify: Ejecuta comprobaciones de pruebas de integración para asegurarse que se cumplen los criterios de calidad.

Install: Instala el paquete en el repositorio local (el directorio ${user.home}/.m2) para que se pueda usar como dependencia en otros proyectos localmente.

Deploy: Copia el paquete final a un repositorio remoto para compartirlo con otros desarrolladores. Hay que tener en cuenta que este despliegue no se refiere en general al despliegue de la aplicación en un servidor web, sino a dejarlo disponible en un repositorio para que lo usen otros desarrolladores.

Aunque cada fase es responsable de un paso específico en el ciclo de vida al que corresponda, la manera en la que se realizan esas responsabilidades puede variar en función del tipo de proyecto (ej. si es un proyecto web, una aplicación de consola, una librería, una aplicación de móviles...), del tipo de empaquetado utilizado (ej. si es WAR, JAR, EAR...), etc. Para configurar qué se hace en cada fase, cada fase se enlaza con uno o más objetivos de plugins. De este

modo, ejecutar una fase del ciclo de vida equivale a ejecutar los objetivos de plugins vinculados a esa fase. Algunas fases tienen ya vinculadas objetivos de plugins por defecto. Esta vinculación varía dependiendo del tipo de packaging del proyecto.

PARA ILUSTRAR ESTO, CONTEMPLE EL SIGUIENTE EJEMPLO:

Se tiene una clase "carpintero" que tiene por función el cortar una tabla por la mitad. Si declaramos en esta clase que la herramienta a utilizar será un serrucho manual y le inyectamos el objeto serrucho desde otra clase contenedora, entonces la clase carpintero logrará la función, pero, solo podrá hacerlo con esa herramienta específica. Luego, si quisiéramos que el carpintero utilice también una sierra eléctrica, tendríamos que, además de crear la clase contenedora para la sierra eléctrica, refactorizar (reescribir) el código de la clase carpintero para que pueda "reconocer" como utilizar este nuevo objeto para lograr la misma función.

Para evitar esto, en vez de declarar, en la clase carpintero, derechamente qué herramienta utilizará para cortar, se le puede informar que será inyectado con una herramienta que se comporta como cierta interfaz establece. En este caso, se le declara que se le inyectará una *dependencia que implementa la interfaz* "HerramientaParaCortarMadera" que declara que las clases que la implementen deben proporcionar un método cortar() que recibe por parámetro una tabla y retorna dos mitades de la tabla, entonces, independiente de cuál sea el objeto inyectado, mientras este implemente la interfaz "HerramientaParaCortarMadera", la clase carpintero sabrá que con ese objeto puede cortar una tabla y obtendrá dos mitades de ella, logrando completar su función, con una nueva herramienta, sin la necesidad de refactorizar su código.

Resumiendo, la utilización de interfaces durante la inyección facilita la implementación de nuevas dependencias sin la necesidad de volver a reestructurar el código de las clases que hacen uso de esas dependencias. En los ejercicios proporcionados para esta sección haremos una demostración práctica del concepto de inyección de dependencias con interfaces.

LINKS:

1- LINKMAVENREPO https://mvnrepository.com/

2-LINKSOLID: https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures b34ce2f1e898

3- LINKPOM: http://maven.apache.org/pom.html

4- LINKMAVE: http://maven.apache.org/guides/index.html