**LABORATORIO 3 – TDD – CVDS**

Samuel Felipe Díaz Mamanche y Santiago Silva

Escuela Colombiana de Ingeniería: Julio Garavito

Ciclos de Vida y Desarrollo de Software – Grupo #1

Mtro. Oscar David Ospina Rodríguez

4 de septiembre de 2024

**OBJETIVOS**

* Como hacer pruebas unitarias.
* Utilizar anotaciones @Test del framework JUnit
* Aplicar TDD.

**DESCRIPCIÓN PROYECTO**

El proyecto consiste en un sistema de gestión de bibliotecas, donde hay clases que representan Libro, Usuario, Prestamo, y Biblioteca. Los usuarios pueden tomar prestados libros de la biblioteca, y la Biblioteca se encarga de gestionar los préstamos, asegurarse de que los libros estén disponibles, y mantener un registro de los libros prestados.

**BITÁCORA DE SOLUCIÓN**

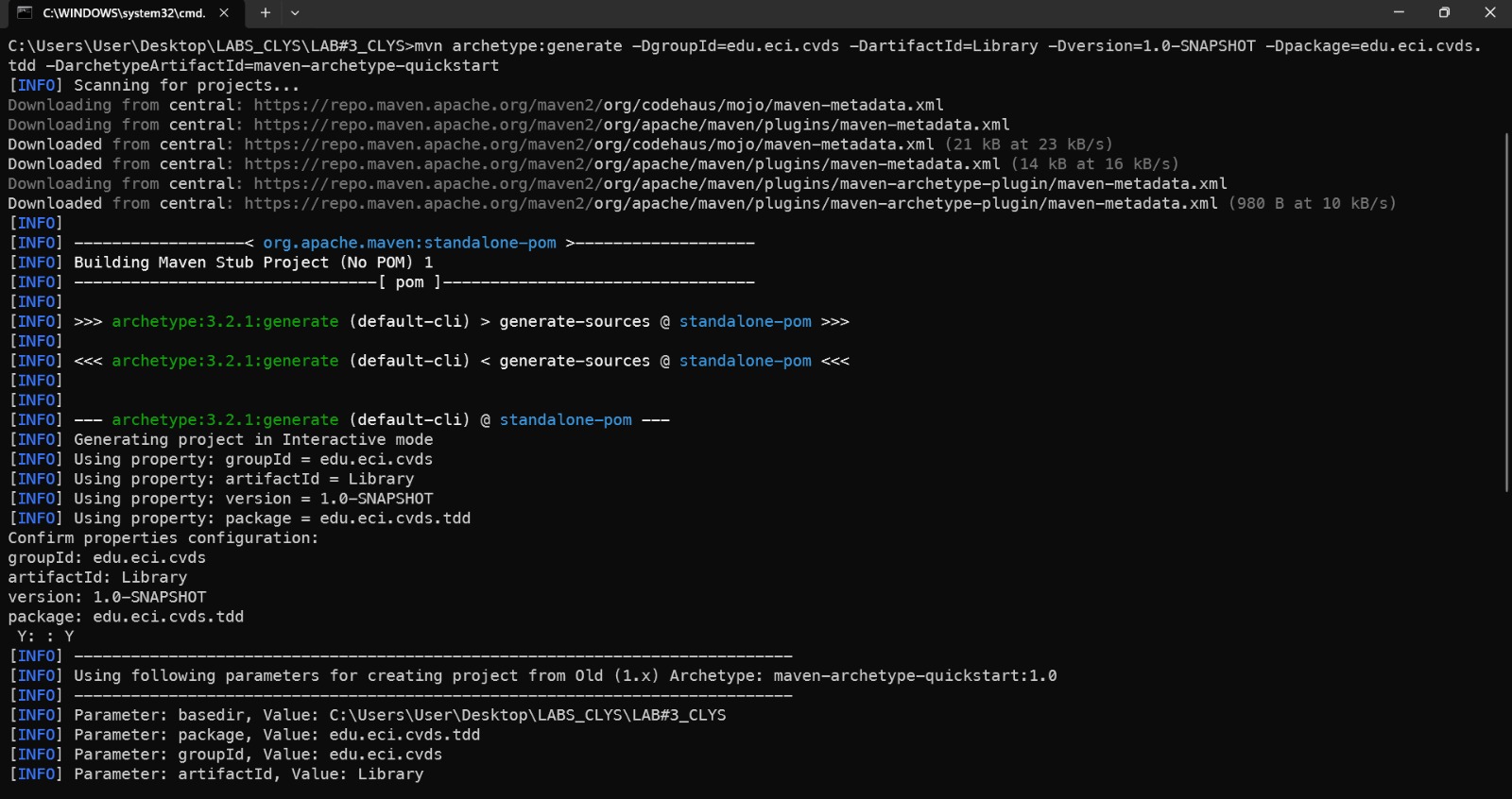
1. **Crear proyecto con Maven:**

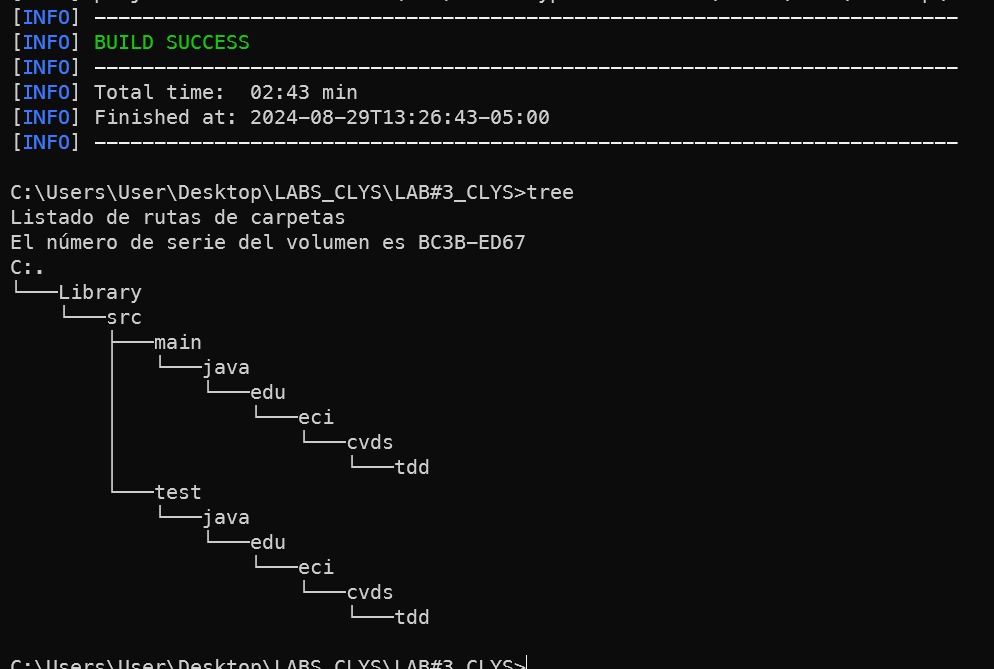
* Creamos un proyecto Maven con los siguientes parámetros:

mvn archetype:generate -DgroupId=edu.eci.cvds -DartifactId=Library -Dversion=1.0-SNAPSHOT -Dpackage=edu.eci.cvds.tdd -DarchetypeArtifactId=maven-archetype-quickstart

**Explicación de los parámetros:**

* **groupId:** Identifica de forma única al grupo.
* **artifactId:** Es el nombre del proyecto.
* **version:** La versión inicial del proyecto.
* **package:** El paquete base donde se ubicarán las clases Java.
* **archetypeArtifactId:** Especifica el arquetipo Maven que se utilizará para crear el proyecto. En este caso, estamos usando maven-archetype-quickstart, que es un proyecto Maven básico de arranque rápido.

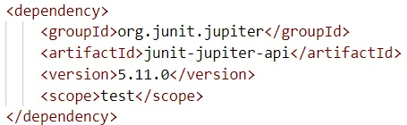
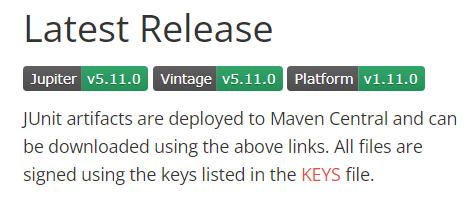
****

****

* Como podemos ver, según el comando tree, el proyecto se construyó con la estructura adecuada y nos generó el archivo .pom.

1. **Agregar dependencia junit5:**

* **Buscar en Maven central la dependencia de JUnit5 en su versión más reciente.**
* **Edite el archivo pom.xml del proyecto para agregar la dependencia.**

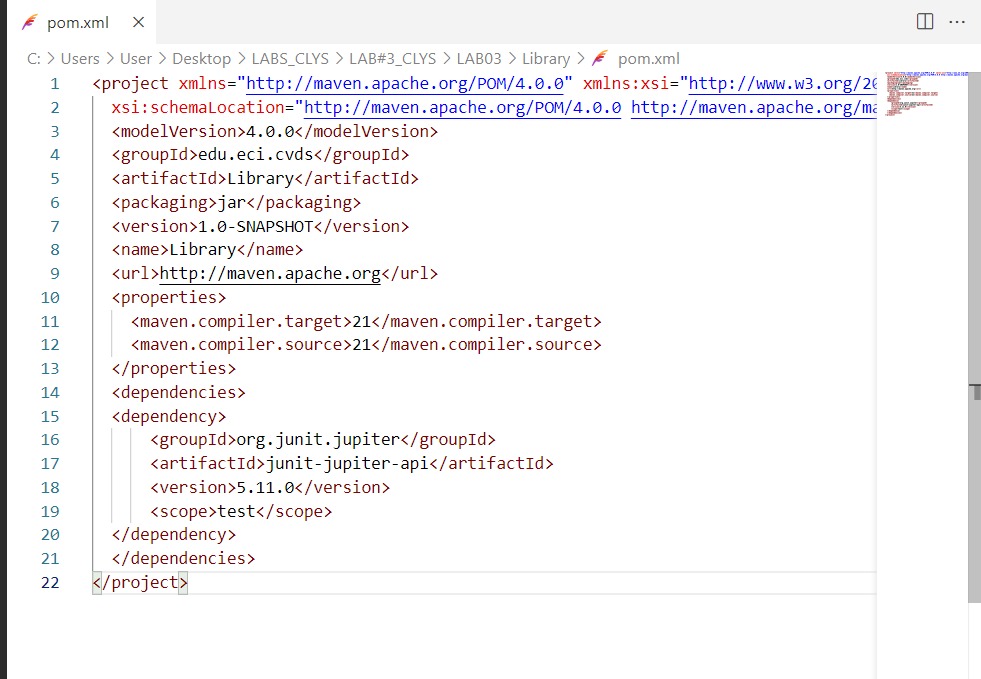
****

* La ultima versión de Junit disponible es la 5.11.0, por lo cual la dependencia que debemos agregar en el archivo .pom es la de la derecha.
* **Verificamos la versión de java que tenemos y agregamos las propiedades en el archivo pom:**

**Texto

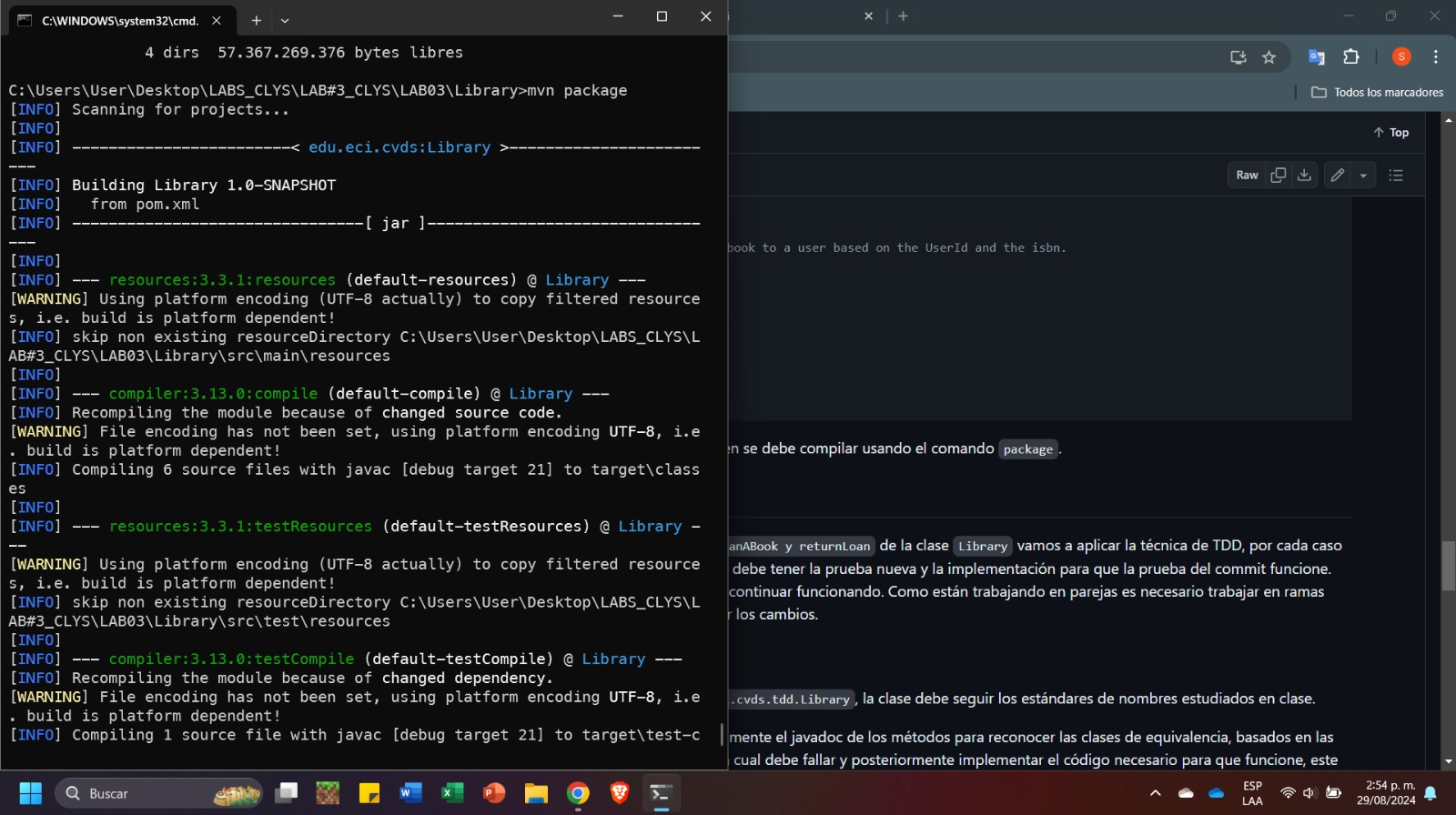
Descripción generada automáticamente**

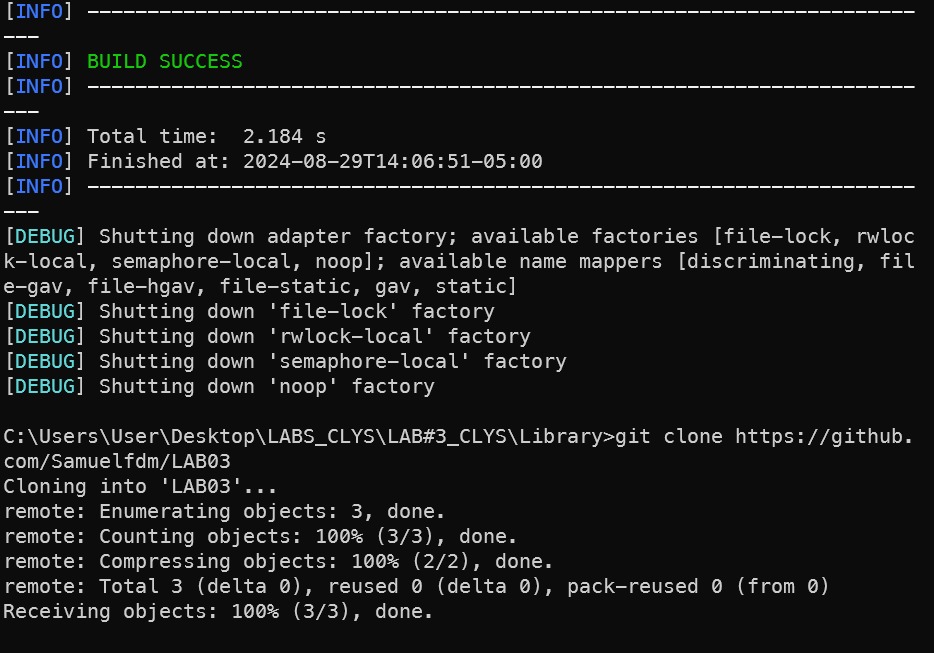
* **El archivo .pom quedó de la siguiente manera:**

****

* **Compile el proyecto para validar que todo esté bien.**

Ejecutamos el comando: mvn package

****

****

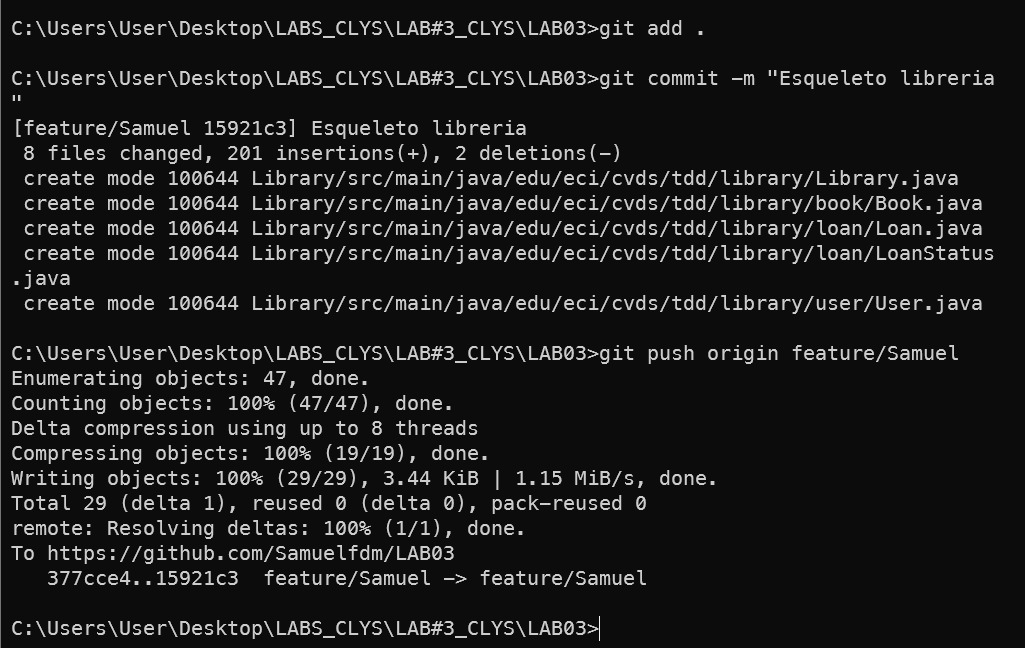
1. **Agregar esqueleto del proyecto:**

* Cree los siguientes paquetes dentro de edu.eci.cvds.tdd (también dentro de test):
* library
  + book
  + loan
  + user
* Y agregamos las clases de una vez, dentro de sus respectivos paquetes.

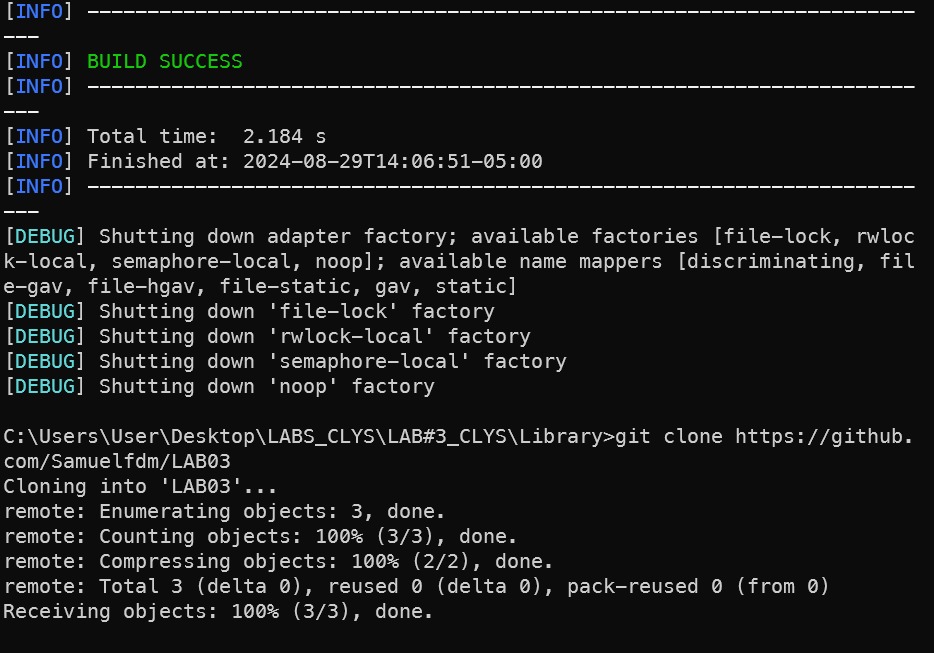
Imagen que contiene Texto

Descripción generada automáticamente

* Subimos los cambios a nuestro repositorio remoto con el esqueleto principal.

****

* Para validar que la estructura del proyecto está bien se debe compilar usando el comando package.

****

1. **Pruebas unitarias y TDD:**

Para poder implementar los métodos addBook, loanABook y returnLoan de la clase Library vamos a aplicar la técnica de TDD, por cada caso de prueba se debe hacer un commit, cada commit debe tener la prueba nueva y la implementación para que la prueba del commit funcione. Las pruebas anteriormente implementadas deben continuar funcionando. Como están trabajando en parejas es necesario trabajar en ramas independientes y utilizar Pull Request para mezclar los cambios.

Crear clase de prueba: Es necesario crear la clase de prueba para edu.eci.cvds.tdd.Library, la clase debe seguir los estándares de nombres estudiados en clase.

Para pensar en los casos de pruebas lean detenidamente el javadoc de los métodos para reconocer las clases de equivalencia, basados en las clases de equivalencia se debe crear una prueba la cual debe fallar y posteriormente implementar el código necesario para que funcione, este proceso se debe repetir hasta cumplir con la especificación definida en el javadoc.

**Solución**

* **Método addBook:**
* **Identificación de clases de equivalencia:**
* **Agregar un libro nuevo:** El libro debe ser agregado al mapa de libros y el tamaño del mapa debe incrementarse en 1.
* **Agregar un libro existente:** La cantidad del libro en el mapa debe incrementarse en 1.
* **Agregar un libro nulo:** Debe retornar false.

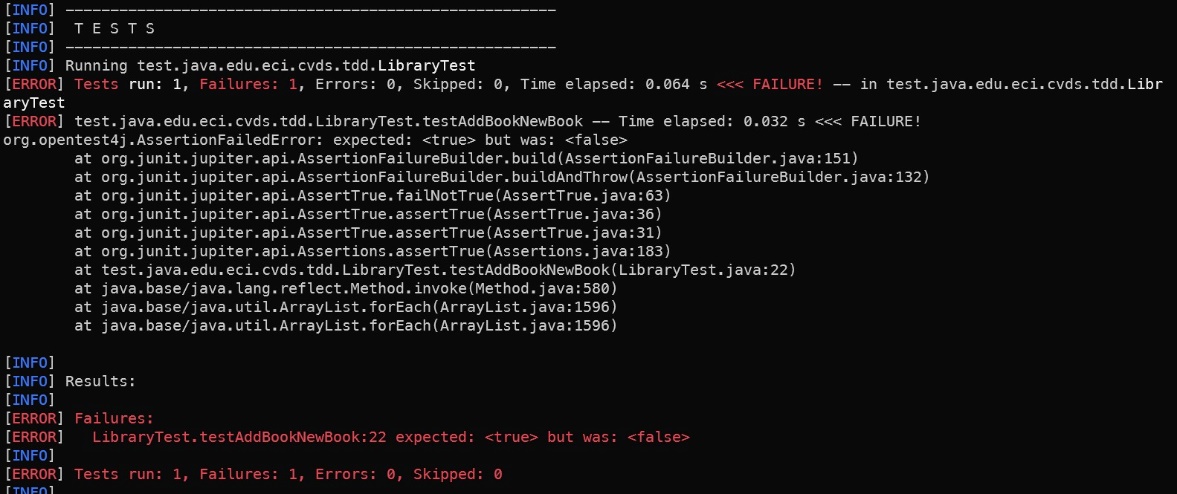
**A continuación, vamos a detallar una muestra del proceso TDD que seguimos para la implementación de las pruebas de cada método y su evolución, solo vamos a enfocarnos en el proceso de solucionar addBook para que no sea muy extenso el documento, puesto que fueron muchas pruebas. Las demás pruebas estarán validadas en GitHub con los commits.**

* **Prueba#1 – testAddBookNewBook():**

**Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente**

* **Ejecutamos en Maven y mostramos el resultado (Debe fallar):**

****

* **Implementación mínima para que la prueba funcione:**

**Texto

Descripción generada automáticamente**

* **Ejecutamos en Maven y mostramos el resultado (Debe funcionar) y hacemos commit:**

**Texto

Descripción generada automáticamente**

* **Prueba#2 – testAddBookExistingBook():**

**Texto

Descripción generada automáticamente**

* **Ejecutamos en Maven y mostramos el resultado (Debe fallar):**

**Texto

Descripción generada automáticamente**

* **Implementación mínima para que la prueba funcione:**

**Interfaz de usuario gráfica, Texto

Descripción generada automáticamente**

* **Ejecutamos en Maven y mostramos el resultado (Debe funcionar) y hacemos commit:**

**Texto

Descripción generada automáticamente**

* **Prueba#3 – testAddBookNullBook():**

**Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente**

* **Ejecutamos en Maven y mostramos el resultado (Debe fallar):**

**Texto

Descripción generada automáticamente**

* **Implementación mínima para que la prueba funcione:**

**Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente**

* **Ejecutamos en Maven y mostramos el resultado (Debe funcionar) y hacemos commit:**

**Texto

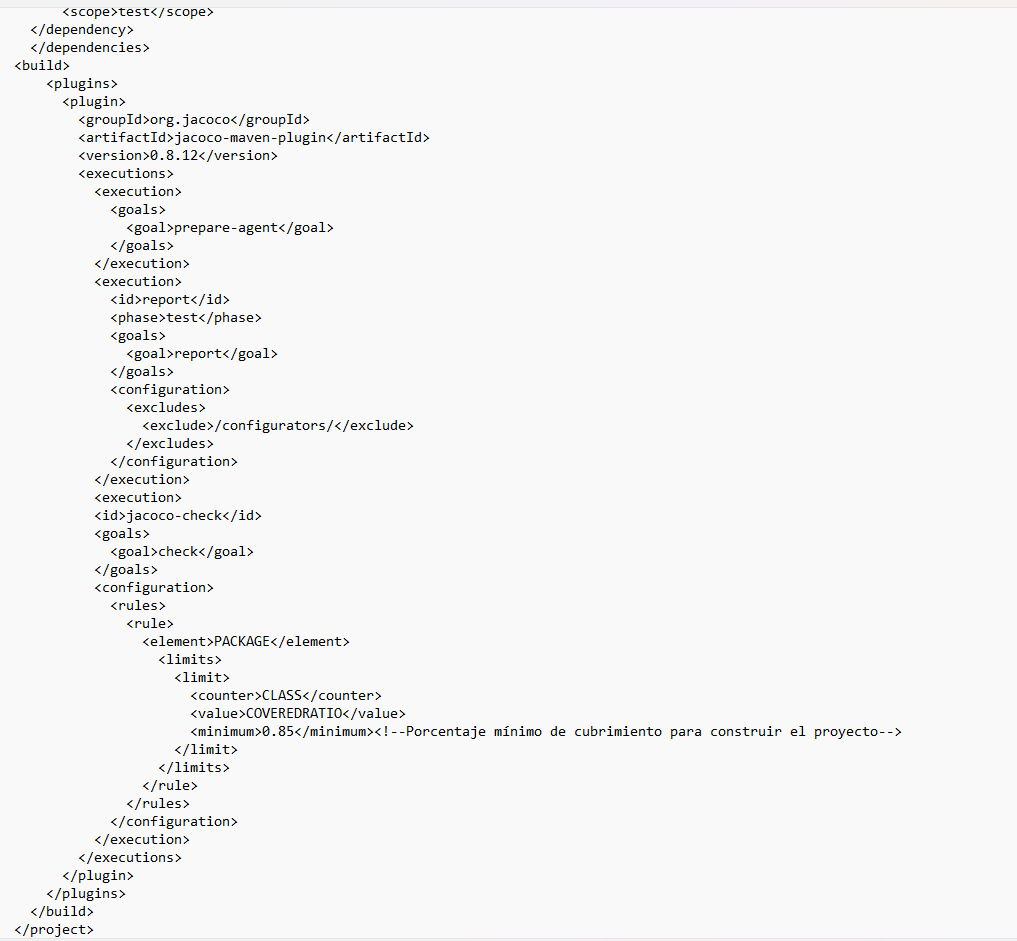
Descripción generada automáticamente**

* **Método loanABook:**
* **Identificación de clases de equivalencia:**
* **Libro Disponible:** El libro existe en la biblioteca y tiene unidades disponibles.
* **Libro No Disponible:** El libro existe en la biblioteca, pero no tiene unidades disponibles.
* **Libro Inexistente:** El libro no se encuentra registrado en la biblioteca.
* **Usuario Existente:** El usuario está registrado en la biblioteca.
* **Usuario Inexistente:** El usuario no está registrado en la biblioteca.
* **Préstamo Existente:** El usuario ya tiene un préstamo activo para el mismo libro.
* **Parámetros Válidos:** Ambos parámetros (userId y isbn) son cadenas de texto no nulas y con formato válido.
* **Parámetros Inválidos:** Al menos uno de los parámetros (userId o isbn) es nulo, vacío o contiene caracteres no válidos.
* **Método returnLoan:**
* **Identificación de clases de equivalencia:**
* **Préstamo Válido:** El préstamo existe en la lista de préstamos de la biblioteca.
* **Préstamo Inexistente:** El préstamo no se encuentra porque es null.

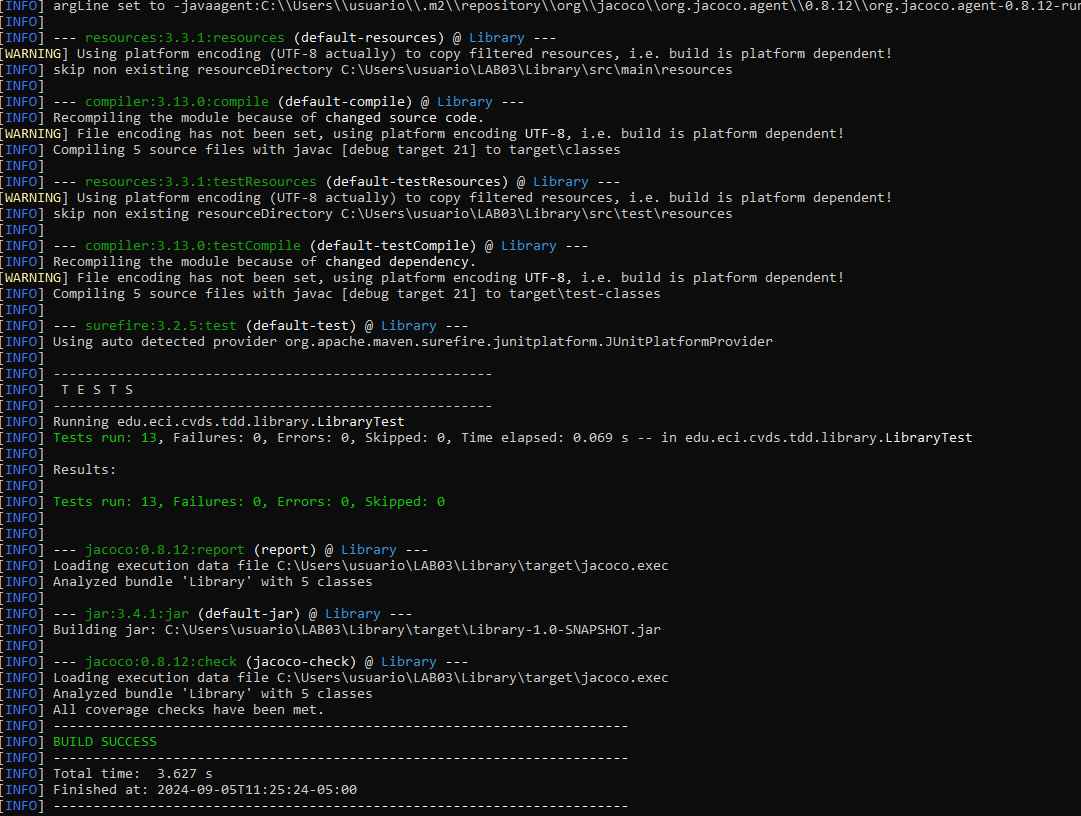
1. **COBERTURA:**

* **Agregar la dependencia de Jacoco, utilizar la última versión disponible en Maven central.**
* **Ahora al compilar el proyecto en la carpeta target se debe crear una carpeta con el nombre site la cual tiene un index.html, al abrir dicho archivo se debe ver la cobertura total y de cada una de las clases, el objetivo es tener la cobertura superior al 80%.**

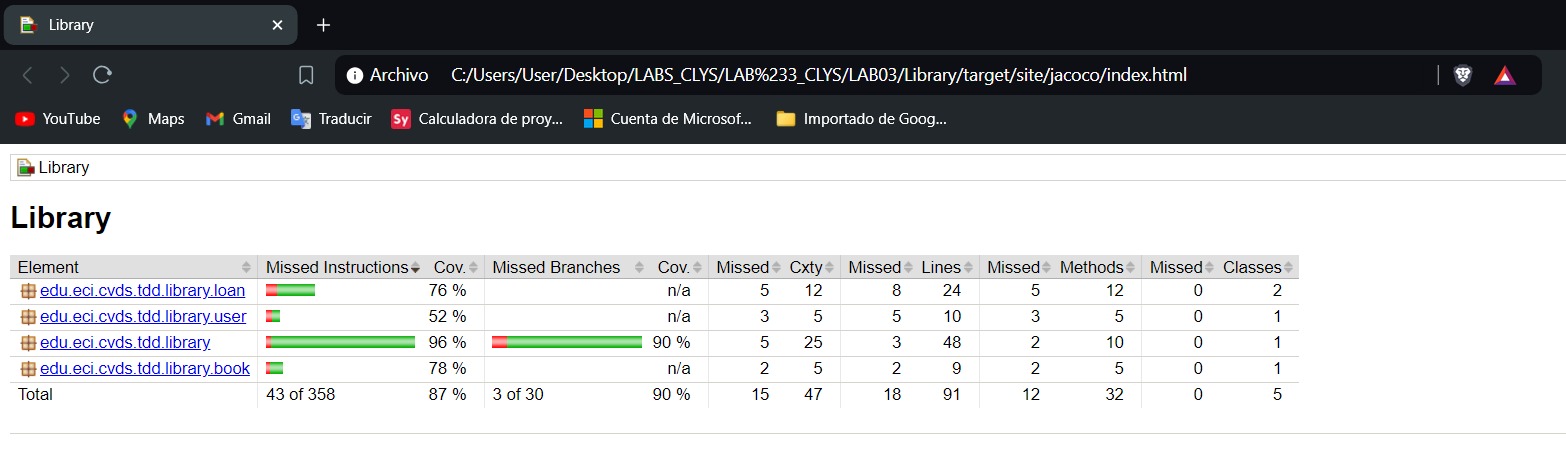
Para poder observar la revisión que hace Jacoco a nuestro código, debemos en primera instancia cambiar nuestro archivo pom.xml, debería verse asi:



Cuando hagamos esto, usaremos el comando “mvn compile” o “mvn clean verify” para que se reconozca y se compile las nuevas dependencias. Y nos aparecerá el mismo mensaje de pruebas exitosas, pero con la nueva dependencia.



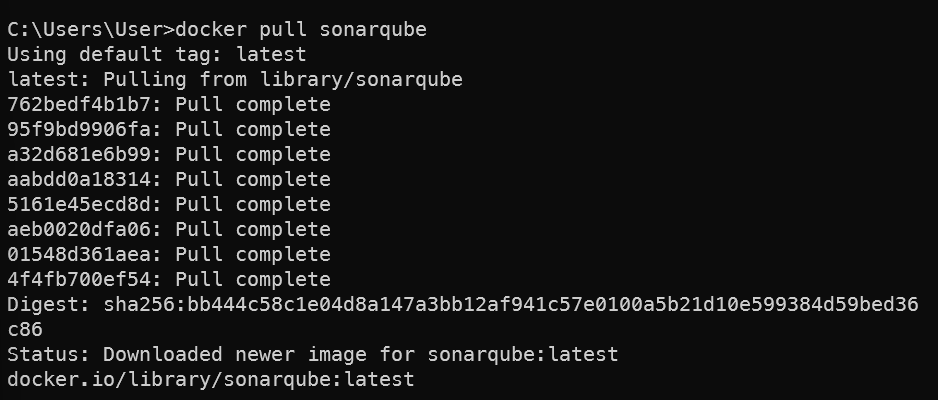
Después, para ver mas detallado y verificar los porcentajes de creación y utilidad de las pruebas iremos a la ruta target > site > index. Cuando abramos esto, nos aparecerá el detalle de las pruebas de nuestro proyecto.

****

* **Explore los links del reporte en el cual le muestra que partes del código tienen prueba y cuáles no.**
* **RTA:** Al analizar los links, nos damos cuenta de que hacen falta pruebas para la mayoría de los métodos getter y setter, pero como tal la implementación de la clase principal Library esta completa y el porcentaje de pruebas es de 96% para ella. En total tenemos un 87% de cobertura, lo que indica que cumplimos el objetivo.

1. **SONARQUBE**

* **Ahora es necesario hacer el análisis estático del código usando SonarQube, para lo cual necesitamos tener Docker. Para lo cual se debe descargar la imagen de docker con el siguiente comando docker pull sonarqube:**

****

* **Ahora se debe arrancar el servicio de SonarQube con el siguiente comando docker run -d --name sonarqube -e SONAR\_ES\_BOOTSTRAP\_CHECKS\_DISABLE=true -p 9000:9000 sonarqube:latest**

**Texto

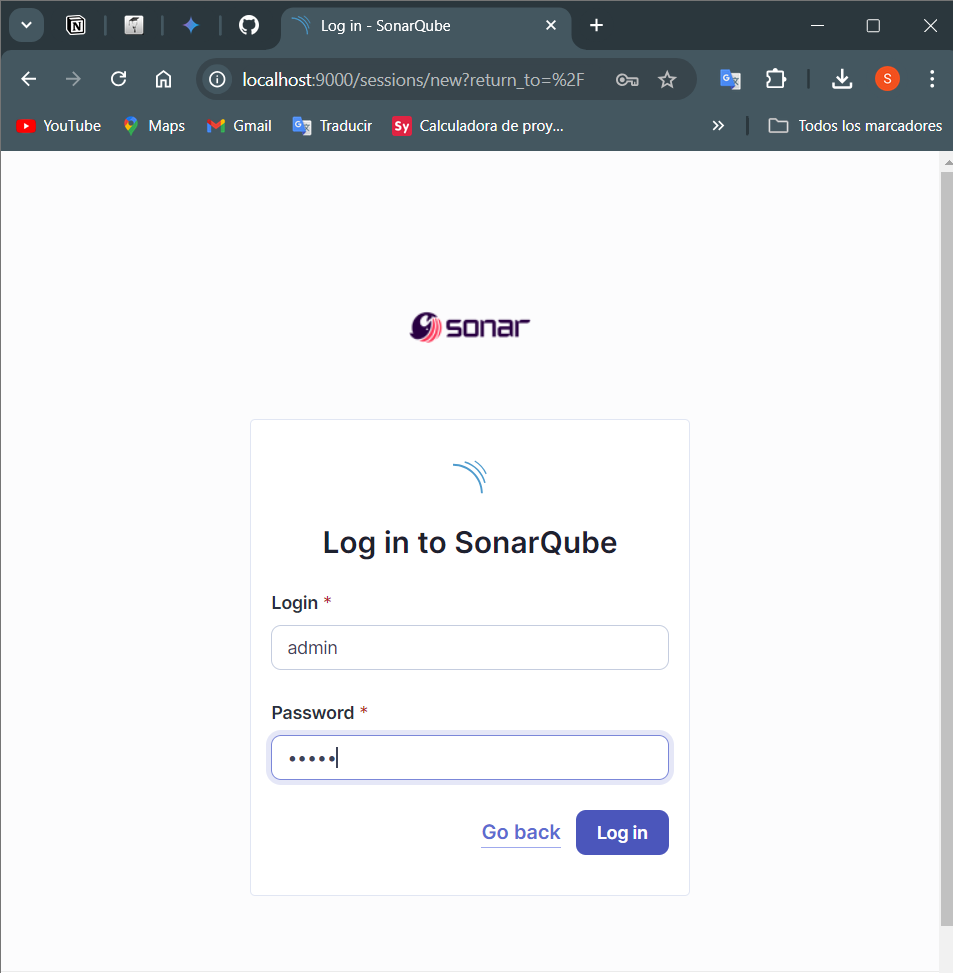
Descripción generada automáticamente**

* **Validar funcionamiento docker ps -a**

**Texto

Descripción generada automáticamente**

* **Iniciar sesión en sonar localhost:9000 cambiar la clave por defecto usuario y contraseña es admin.**

****

**Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente**

* **Cambiamos la contraseña por defecto admin por DiazSilva y entramos al banner principal:**

**Interfaz de usuario gráfica, Texto, Aplicación

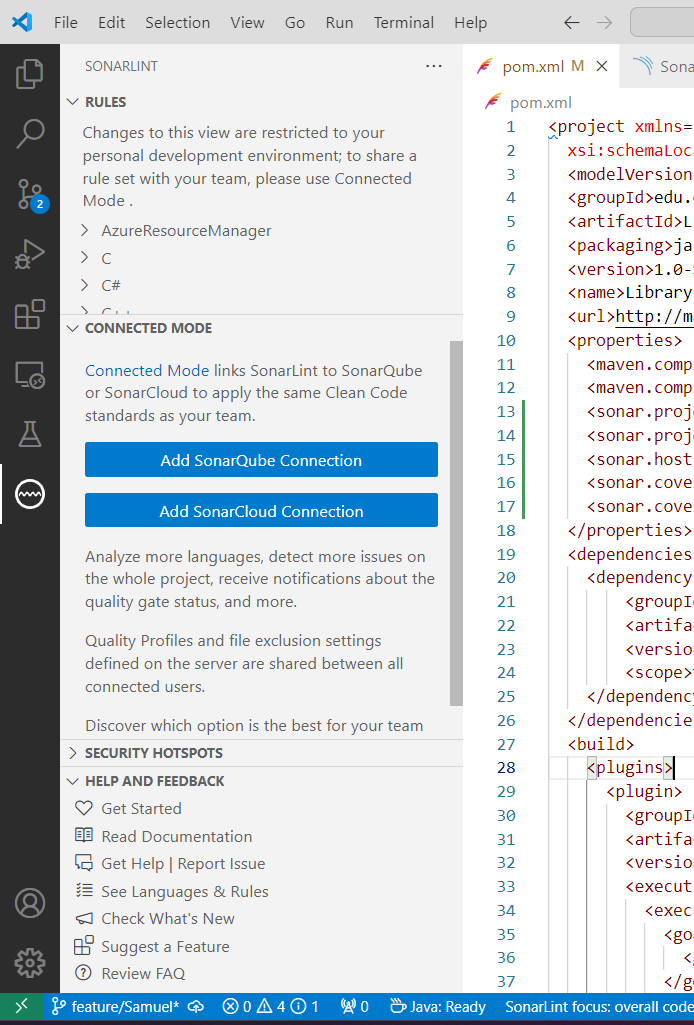
Descripción generada automáticamente**

* **Entrar a las opciones de la cuenta.**
  + **Account -> settings -> generate token.**
* **Una vez sonar este corriendo deben generar un token.**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente**

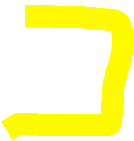
* **Instale sonarLint en el IDE que este manejando.**

****

* **Añada el plugin de Sonar en el archivo pom del proyecto.**
* **Añada las propiedades de SonarQube y Jacoco.**

**Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente**



**Escala de tiempo

Descripción generada automáticamente**



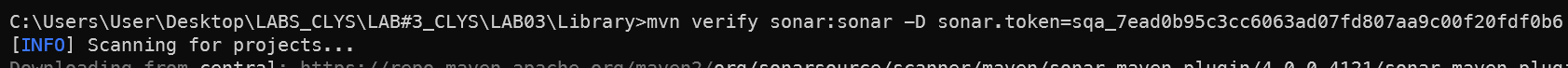
* **Construya el proyecto, genere el reporte de JACOCO y corrija el cubrimiento de las pruebas de unidad para que su proyecto se construya adecuadamente.**
* **genere la integración con sonar mvn verify sonar:sonar -D sonar.token=[TOKEN\_GENERADO]**

**Texto

Descripción generada automáticamente**

**Texto

Descripción generada automáticamente**

****

****

**…**

**Texto

Descripción generada automáticamente**

* **La integración con Sonar fue exitosa.**

**WEBGRAFÍA**

* https://www.geeksforgeeks.org/collections-unmodifiablelist-method-in-java-with-examples/
* https://www.zaptest.com/es/particionamiento-de-equivalencia-en-pruebas-de-software-que-es-tipos-proceso-enfoques-herramientas-y-mas
* https://programandoenjava.com/equals-y-hashcode-en-java/
* https://programandoenjava.com/hashmap-en-java/#:~:text=Un%20HashMap%20o%20(Map%20en,valor%20correspondiente%20de%20un%20HashMap