

Práctica 4 - Bases de Datos 2

Hayk Kocharyan
757715@unizar.es

Juan José Tambo Tambo
755742@unizar.es

Pedro Tamargo Allué
758267@unizar.es

Jesús Villacampa Sagaste
755739@unizar.es

26 de mayo de 2020

Índice

1. Esfuerzos invertidos	1
2. Modelo conceptual	1
3. Entorno de trabajo y ejecución	2
4. Generación de esquema lógico con JPA	5
5. Adopción de un esquema lógico preexistente con JPA	6
6. Consultas con JPA	7
7. Conclusiones y opiniones	9
8. Apéndice 1: Figuras	10

Índice de figuras

1. Diagrama ER de la Base de Datos bancaria de <i>Banquito</i>	11
2. Diagrama de clases UML de la Base de Datos bancaria de <i>Banquito</i>	12
3. Estructura de un proyecto con la herramienta <i>Apache Maven</i>	12
4. Resultado de la búsqueda del Driver <i>JDBC</i> de <i>Oracle</i> en <i>Maven Central</i>	13

1. Esfuerzos invertidos

- Hayk:
 - Formación en *JPA*: 3 horas
 - Memoria: 1 hora
- Juan José:
 - Formación en *JPA*: 3 horas
 - Código Java del esquema lógico preexistente: 2 horas
 - Consultas de ambos esquemas: 4 horas
 - Memoria: 1 hora
- Pedro:
 - Formación en *JPA*: 3 horas
 - Código del esquema lógico desde 0: 1 hora
 - Código Java del esquema lógico preexistente: 2 horas
 - Memoria: 4 horas
- Jesús:
 - Formación en *JPA*: 3 horas
 - Consultas de ambos esquemas: 4 horas
 - Memoria: 2 horas

2. Modelo conceptual

Durante el desarrollo de la práctica 1 se diseñó una base de datos para un banco que quería gestionar cuentas con múltiples propietarios, diferentes tipos de cuentas (cuentas ahorro y cuentas corrientes), operaciones (transacciones entre cuentas, o movimientos de dinero en efectivo) y las sucursales de la entidad.

En la Figura 1 se puede observar el esquema entidad relación sobre el problema planteado. Se ha planteado las relaciones entre los distintos tipos de cuentas como una generalización, ya que todas comparten ciertos atributos como el número de cuenta, el *IBAN*, la fecha de apertura y el saldo restante. Esta generalización es exclusiva ya que no se considera posible la capacidad de que una cuenta pertenezca a los dos tipos de entidades al mismo tiempo. También, se trata de una generalización total ya que no se considera el caso de que una cuenta no pertenezca a algunos de las entidades derivadas de *Cuenta*.

Con transacción ocurre lo mismo, una transferencia entre cuentas y las operaciones de retirada o ingreso de efectivo se pueden generalizar en una nueva entidad *Transacción* con los atributos comunes a ambas entidades, tales como: el número de la transacción, la fecha y hora de la misma, su importe y una descripción a modo de concepto. Se trata de una generalización exclusiva ya que una transferencia no puede pertenecer a los dos subtipos a la vez. También, se trata de una generalización total ya que en el contexto del problema no tiene sentido que exista una transferencia que no pertenezca a *Operación* (operaciones en efectivo) o a *Transferencia* (transferencia de saldo entre cuentas).

Se ha decidido que *Transacción* debía ser débil respecto a *cuenta* ya que depende en existencia e identificación de *cuenta*, por lo tanto la relación *Realizar* es una relación *1:N* entre *Cuenta* y *Transacción*. No obstante, existe una transacción que no indica la debilidad de la entidad *Transacción* respecto a *Cuenta*, la relación *Recibir*, una relación *1:N* que relaciona las entidades *Cuenta* y *Transferencia*, y cuyo significado es relacionar una transferencia con la cuenta beneficiaria.

Los clientes titulares de una cuenta se reflejan en la entidad *Cliente*, que almacena el DNI, el nombre,

los apellidos la dirección y el email. Se ha decidido que el *DNI* sea la clave primaria ya que es único para los ciudadanos.

La relación de *Cliente* con *Cuenta* se encuentra en *Poseer*, una relación *M:N* que posibilita que una cuenta tenga más de un titular.

Para el almacenamiento de las sucursales de la entidad bancaria, se ha diseñado una entidad *Sucursal*, que almacena el código de la entidad, su dirección postal y el teléfono de la oficina. Se ha decidido que el código de la sucursal sea la clave primaria que identifique a las sucursales ya que dentro de la misma entidad bancaria no existen dos sucursales con el mismo código.

Se ha considerado que las transacciones se tienen que realizar en una sucursal, por lo tanto existe una relación entre *Transacción* y *Sucursal*. También, una cuenta corriente debe ser abierta en una determinada sucursal de la entidad.

Como apoyo hacia el desarrollo orientado a objetos con *Java* se ha utilizado como referencia un diagrama de clases *UML* (Figura 2) con la misma semántica que el diagrama entidad relación explicado anteriormente.

3. Entorno de trabajo y ejecución

Para la realización de la práctica se ha utilizado la herramienta *Apache Maven*¹ para compilar, gestionar las dependencias y ejecutar el código.

Como entorno de desarrollo se va a utilizar *Visual Studio Code*², un editor de código abierto que tiene integraciones con diferentes herramientas (como *Maven*) y lenguajes de programación.

Para configurar la herramienta *Maven* debemos crear un proyecto con la forma especificada en la Figura 3. El fichero *pom.xml* (*Project Object Model*) se corresponde con el fichero que utiliza *Maven* para gestionar las dependencias y establecer distintas etapas del ciclo de vida del código, tales como *COMPILE*, *EXEC* o *CLEAN*.

Para no utilizar un fichero *pom.xml* vacío desde 0, se ha procedido a utilizar un *archetype*, el cual proporciona una estructura básica. El *archetype* utilizado ha sido: *archetype-quickstart-jdk8*.

El código se desarrollará en el directorio *src/main/java/package-name*, donde *package-name* se corresponde con el *groupId* especificado en el fichero *pom.xml*.

Para ejecutar el código generado primero debemos compilarlo, para ello, desde la línea de comandos del sistema operativo, en el mismo directorio donde se encuentre el fichero *pom.xml* escribiremos la siguiente orden:

```
# Limpiamos los resultados de una ejecución anterior
mvn clean
# Primero compilamos el código Java
mvn compile
# Donde package-name se corresponde con el groupId especificado.
mvn exec:java -Dexec.mainClass="package-name.App"
```

Para añadir las dependencias de los distintos *JAR* que necesite el proyecto, hay que añadirlas en el apartado *dependencies* del fichero *pom.xml* de la siguiente forma:

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
```

¹<https://maven.apache.org/>

²<https://code.visualstudio.com/>

```

</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>${junit.version}</version>
  <scope>test</scope>
</dependency>
<!-- Driver JDBC Oracle -->
<dependency>
  <groupId>com.oracle.database.jdbc</groupId>
  <artifactId>ojdbc8</artifactId>
  <version>19.6.0.0</version>
</dependency>
<!-- Jar de Hibernate Core -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.4.15.Final</version>
</dependency>
<!-- Jar de entity manager -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>5.4.15.Final</version>
</dependency>
<!-- javax.persistence -->
<dependency>
  <groupId>javax.persistence</groupId>
  <artifactId>persistence-api</artifactId>
  <version>1.0.2</version>
</dependency>
</dependencies>

```

Siendo cada elemento entre las etiquetas *dependency* una dependencia que *Maven* tiene que resolver. Para obtener la información acerca de la dependencia se puede consultar el repositorio de *Maven Central*³. En este repositorio se puede realizar una búsqueda de la dependencia y obtener información de la misma. En la Figura 4 se puede observar el resultado de la búsqueda del driver *JDBC* de *Oracle* utilizando este sistema. En la parte derecha de la misma se puede observar que se proporciona la etiqueta *dependency* que ha sido añadida al fichero *pom.xml*.

Tras la configuración de la herramienta es necesaria la configuración de *JPA*, a partir del fichero *persistence.xml*. Este fichero debe estar alojado en el *Classpath* del proyecto, dentro del directorio *META-INF*. Por lo tanto se ha creado un directorio *src/main/resources/META-INF/* en el cual se alojará este fichero. Para la configuración de la unidad de persistencia se utilizará la base de datos *Oracle* ubicada en *danae04.cps.unizar.es*. El fichero *persistence.xml* tendrá la siguiente forma:

```

<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">
  <persistence-unit name="UnidadPersistenciaAlumnos"
    transaction-type="RESOURCE_LOCAL">

```

³<https://search.maven.org/>

```

<provider>org.hibernate.ejb.HibernatePersistence</provider>
<properties>
  <property name="hibernate.dialect"
    value="org.hibernate.dialect.Oracle8iDialect" />
  <property name="hibernate.show_sql" value="true"/>
  <property name="hibernate.hbm2ddl.auto" value="create"/>

  <property name="javax.persistence.jdbc.driver"
    value="oracle.jdbc.driver.OracleDriver"/>
  <property name="javax.persistence.jdbc.url"
    value="jdbc:oracle:thin:@danae04.cps.unizar.es:1521:barret"/>
  <property name="javax.persistence.jdbc.user" value=""/>
  <property name="javax.persistence.jdbc.password" value=""/>

</properties>
</persistence-unit>
</persistence>

```

Tras la ejecución del código *JPA* puede aparecer este error:

```

[WARNING] thread Thread[Timer-0,5,bases2.App] was interrupted but is
  still alive after waiting at least 15000msecs
[WARNING] thread Thread[Timer-0,5,bases2.App] will linger despite
  being asked to die via interruption
[WARNING] thread Thread[InterruptTimer,5,bases2.App] will linger
  despite being asked to die via interruption
[WARNING] thread Thread[Hibernate Connection Pool Validation
  Thread,5,bases2.App] will linger despite being asked to die
  via interruption
[WARNING] NOTE: 3 thread(s) did not finish despite being asked to
  via interruption. This is not a problem with exec:java,
  it is a problem with the running code.
  Although not serious, it should be remedied.
[WARNING] Couldn't destroy threadgroup
  org.codehaus.mojo.exec.ExecJavaMojo$IsolatedThreadGroup
  [name=bases2.App,maxpri=10]
java.lang.IllegalThreadStateException
  at java.lang.ThreadGroup.destroy (ThreadGroup.java:776)
  at [...]

```

Este error es debido a que en el final del código un proceso *Daemon* se encarga de monitorizar que todos los hilos (*Threads*) terminen correctamente. Para solventar este error podemos utilizar este otro comando para ejecutar el código:

```

mvn exec:java
  -Dexec.mainClass="package-name.App"
  -Dexec.cleanupDaemonThreads=false

```

4. Generación de esquema lógico con JPA

Para la generación del esquema lógico utilizando *JPA* se ha utilizado el diagrama de clases de la Figura 2. El primer paso de la generación ha sido la confección de las clases *Java* utilizando las anotaciones necesarias. En este caso, se han utilizado las anotaciones *@Entity* para marcar las entidades y las etiquetas *@Column* para las columnas. Para las columnas se ha añadido un atributo *name* para especificar el nombre de esa columna en la base de datos y así conocer los nombres de las columnas. Para que *JPA* funcione correctamente se deben definir los métodos *hashCode()* y *equals()*. También, en el ámbito de las consultas es conveniente definir un método *toString()* para obtener una salida comprensible del contenido de cada uno de los objetos. También es necesario que las clases tengan una estructura tipo *Java Bean*⁴, esto implica que las clases deben disponer de un constructor por defecto (se llama sin parámetros) y métodos *getters* y *setters* para el acceso directo a cada uno de sus atributos, que deberán ser marcados como privados para impedir su acceso directo desde fuera de la propia clase.

Para resolver la herencia en este primer esquema lógico se ha decidido establecer las *superclases* *Cuenta* y *Transacción* como clases abstractas. De esta manera no podrán ser instanciadas y por lo tanto se obligará a que la herencia sea obligatoria. Además al utilizar dos subclases, la herencia será disjunta ya que un objeto no puede pertenecer a dos subclases al mismo tiempo en este modelo conceptual.

El mecanismo predeterminado para la resolución de estas herencias ha sido el de crear una sola tabla con el nombre del padre con todos los atributos del padre y los específicos de cada uno de los hijos. Para determinar si una tupla en la base de datos pertenece a un subtipo o a otro se dispone de una columna (por defecto, *DTYPE*) que contiene el nombre del subtipo cuyos valores se encuentran con valores permitidos para ese subtipo.

Sobre las relaciones, se ha utilizado una relación *ManyToMany* bidireccional entre *Cliente* y *Cuenta*, siendo cliente la parte poseedora. Esta acción genera una tabla nueva *Cliente_Cuenta*, que representará lo mismo que la tabla *Poseer* de la base de datos de *Banquito*.

No se han utilizado más relaciones bidireccionales pero es de destacar la relación *ManyToOne* unidireccional entre *Transacción* y *Cuenta*, ya que transacción es una entidad débil ante cuenta. La columna resultante de esta relación actúa también como parte de la clave primaria al utilizarse la anotación *@Id*, siendo la clave primaria (*NumTransaccion*, *realizante*).

⁴<https://es.wikipedia.org/wiki/JavaBean>

5. Adopción de un esquema lógico preexistente con JPA

Partiendo de las clases desarrolladas en el apartado anterior, se va a proceder a realizar las modificaciones sobre las mismas para adaptar al esquema lógico existente. Esta tarea no ha resultado especialmente difícil ya que *JPA* provee muchas anotaciones para modificar el comportamiento del esquema que se va a generar o que se va a adoptar. Para facilitar esta tarea se ha modificado un campo del fichero *persistence.xml*, el campo *hibernate.hbm2ddl.auto* adoptará el valor *validate*. Esto comprobará si el esquema generado por *JPA* coincide con el esquema existente en el remoto especificado en ese mismo fichero. No realiza ninguna modificación sobre la base de datos.

Para dejar claro el nombre de las tablas sobre las que actúan cada una de las entidades (*@Entity*) desarrolladas se va a utilizar la anotación (*@Table(name = "[...]"*)).

Las relaciones desarrolladas anteriormente se han mantenido, no obstante se ha utilizado la anotación *@JoinColumn* para establecer atributos tales como el nombre de la columna, restricciones de unicidad. En la relación *ManyToMany* entre *Cliente* y *Cuenta* se ha utilizado la anotación *@JoinTable* de la siguiente manera:

```
@ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
@JoinTable(
    name = "POSEER",
    joinColumns = @JoinColumn(
        referencedColumnName = "DNI", name = "DNI"
    ),
    inverseJoinColumns = @JoinColumn(
        referencedColumnName = "NUM_CUENTA", name = "NUM_CUENTA"
    )
)
private List<Cuenta> cuentas;
```

Con esta anotación se ha establecido que la relación entre cliente y cuenta se establecerá sobre una tabla *POSEER* con columnas *DNI* referenciando a *Cliente.DNI* y con la columna *NUM_CUENTA* referenciando a *Cuenta.NUM_CUENTA*.

Sobre la herencia, se han utilizado distintas estrategias de las propuestas por *JPA*, en este caso, para la tabla cuenta se ha utilizado la estrategia *JOINED*, la cual divide los campos específicos de la tabla padre en una tabla y los campos específicos de cada subtipo en sus respectivas tablas. En las tablas de los subtipos para establecer el nombre de la columna que actúa como clave primaria se utiliza la anotación *@PrimaryKeyJoinColumn* con el atributo *name*, en este caso se establecerá *name = "ID_CUENTA"* para que coincidan con el esquema lógico preexistente.

En el caso de la herencia entre *Transacción*, *Transferencia* y *Operación*, se ha establecido un mecanismo de herencia distinto. En este caso la superclase (transacción) no será una entidad, si no que se utilizará la anotación *@MappedSuperclass* para indicar que esta superclase no se persistirá como una tabla en la base de datos, pero sus campos si que serán persistidos en cada uno de sus subtipos (transferencia y operación).

6. Consultas con JPA

Aquí se expondrán las consultas, con su código SQL, JPQL, Criteria API.

JUANJO00000 PORFAVOR EXPLICA LO DE QUERY Y QUERYTRANSFORMER DICHIENDO QUE ES DE HIBERNATE Y NO DE JPA Y POR QUE LO HAS USADO.

Para la elaboración de las consultas en *JPA* se han propuesto 5 consultas diferentes y se han abordado tanto para el esquema lógico generado con *JPA* como para el esquema lógico preexistente adaptado. También se han utilizado los diferentes lenguajes de consultas propuestos (SQL, JPQL y Criteria). Las consultas propuestas son las siguientes:

- Consulta 1: Clientes que más dinero han extraído de sus cuentas
- Consulta 2: Sucursales en las que los clientes menores de 30 años tienen cuentas corrientes.
- Consulta 3: Última transferencia de todas cuentas de todos los clientes
- Consulta 4: Máximo movimiento (de ingreso) en una transacción en cada una de las cuentas de cada cliente.
- Consulta 5: Ingreso total de dinero de las cuentas del usuario mas mayor

Para la consulta 3, para las dos versiones, se ha abordado la solución en *JPQL*.

El código para la versión de esquema lógico adaptado es el siguiente:

```
EntityManager em = emf.createEntityManager();
String query_text =
    "SELECT cl.Nombre, real.numCuenta, tr.fecha " +
    "FROM Transferencia tr JOIN tr.realizante real
    JOIN real.propietarios cl " +
    "WHERE tr.fecha IN ( SELECT MAX(tr2.fecha) FROM Transferencia tr2
    WHERE tr2.realizante.numCuenta = tr.realizante.numCuenta) "+
    "GROUP BY cl.Nombre, real.numCuenta, tr.fecha ORDER BY cl.Nombre";

javax.persistence.Query query3 = em.createQuery(query_text);

@SuppressWarnings("unchecked")
List<Query3> results =
    query3.unwrap(org.hibernate.query.Query.class)
    .setResultTransformer(new Query3Transformer()).getResultList();

System.out.println("----- Mostrando Query3 en JPQL -----");

for (Query3 q : results) {
    System.out.println(q);
}
```

El código para la versión de esquema lógico creado es el siguiente:

```
EntityManager em = emf.createEntityManager();
String query_text =
    "SELECT cl.Nombre, real.numCuenta, tr.fecha " +
    "FROM Transaccion tr JOIN tr.realizante real JOIN
    real.propietarios cl " + "WHERE tr.fecha IN
    ( SELECT MAX(tr2.fecha) FROM Transaccion tr2 WHERE
    tr2.realizante.numCuenta = tr.realizante.numCuenta) "+
    "GROUP BY cl.Nombre, real.numCuenta, tr.fecha
    ORDER BY cl.Nombre";
```



```

    javax.persistence.Query query3 = em.createQuery(query_text);

    @SuppressWarnings("unchecked")
    List<Query3> results =
        query3.unwrap(org.hibernate.query.Query.class)
            .setResultTransformer(new Query3Transformer()).getResultList();

    System.out.println("----- Mostrando Query3 en JPQL -----");

    for (Query3 q : results) {
        System.out.println(q);
    }

```

Para la consulta 5, para las dos versiones, se ha abordado la solución en *JPQL*.
El código para la versión de esquema lógico adaptado es el siguiente:

```

EntityManager em = emf.createEntityManager();
String query_text =
    "SELECT cl.Nombre, realizantes.numCuenta, SUM(op.importe),
    cl.edad "+ "FROM Operacion op JOIN op.realizante realizantes "
    + "JOIN realizantes.propietarios cl " +
    "WHERE op.tipo = 'Ingreso' AND "+ "cl.edad =
    (SELECT MAX(cl2.edad) FROM Cliente cl2) "
    + "GROUP BY realizantes.numCuenta, cl.Nombre, cl.edad ORDER BY
    SUM(op.importe) DESC";

    javax.persistence.Query query5 = em.createQuery(query_text);
    @SuppressWarnings("unchecked")
    List<Query5> results =
        query5.unwrap(org.hibernate.query.Query.class)
            .setResultTransformer(new Query5Transformer()).getResultList();

    System.out.println("----- Mostrando Query5 en JPQL -----");

    for (Query5 q : results) {
        System.out.println(q);
    }

```

El código para la versión de esquema lógico creado es el siguiente:

```

EntityManager em = emf.createEntityManager();
String query_text =
    "SELECT cl.Nombre, realizantes.numCuenta, SUM(op.importe), cl.Edad "
    + "FROM Operacion op JOIN op.realizante realizantes "
    + "JOIN realizantes.propietarios cl " + "WHERE op.tipo = 'Retirada' AND "
    + "cl.Edad = (SELECT MAX(cl2.Edad) FROM Cliente cl2) "
    + "GROUP BY realizantes.numCuenta, cl.Nombre, cl.Edad ORDER BY
    SUM(op.importe) DESC";

    javax.persistence.Query query5 = em.createQuery(query_text);
    @SuppressWarnings("unchecked")
    List<Query5> results = query5.unwrap(org.hibernate.query.Query.class)
        .setResultTransformer(new Query5Transformer()).getResultList();

```

```
System.out.println("----- Mostrando Query5 en JPQL -----");

for (Query5 q : results) {
    System.out.println(q);
}
```

7. Conclusiones y opiniones

JPA es una herramienta muy potente para el acceso a los datos de una base de datos de manera uniforme desde el código *Java*. Nos permite abstraernos de la estructura de la base de datos teniendo solamente en cuenta las entidades definidas en el mismo código.

Uno de los aspectos a tener en cuenta de esta librería es su gran adaptabilidad a una base de datos ya existente, esto es gracias a que existen muchas anotaciones que permiten modificar el comportamiento de *JPA* drásticamente.

La utilización y formación en la herramienta no ha resultado complicada debido a que la sintaxis y semántica de las anotaciones no dificultó la tarea. Además se han consultado los capítulos del libro *Java Persistence with Hibernate* ubicados en la página de la asignatura.

PONER UNA CONCLUSIÓN DE LAS CONSULTAS.

Para finalizar, se ha utilizado anteriormente el framework *Django*⁵. Este Framework está orientado al desarrollo web utilizando *Python*. Además cuenta con un *ORM* que hace casi totalmente transparente el acceso a la base de datos de cara al programador.

La diferencia principal entre estos dos *ORMs* se basa en las anotaciones, siendo *JPA* más sencillo de implantar, tanto en el caso de la generación del esquema lógico desde 0 como en la adopción de un esquema preexistente.

⁵<https://www.djangoproject.com/>

8. Apéndice 1: Figuras

Diagrama ER BANCO

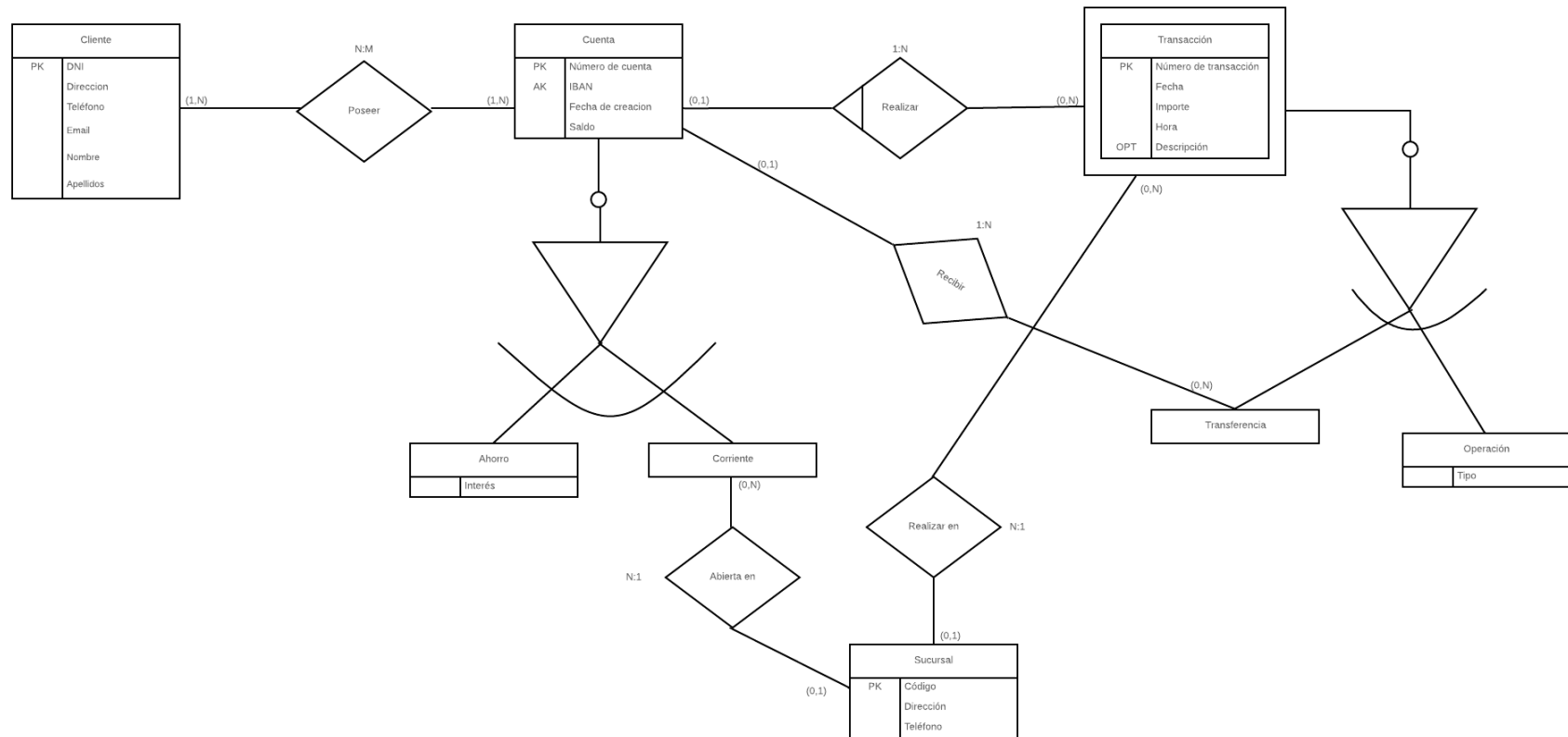


Figura 1: Diagrama ER de la Base de Datos bancaria de *Banquito*

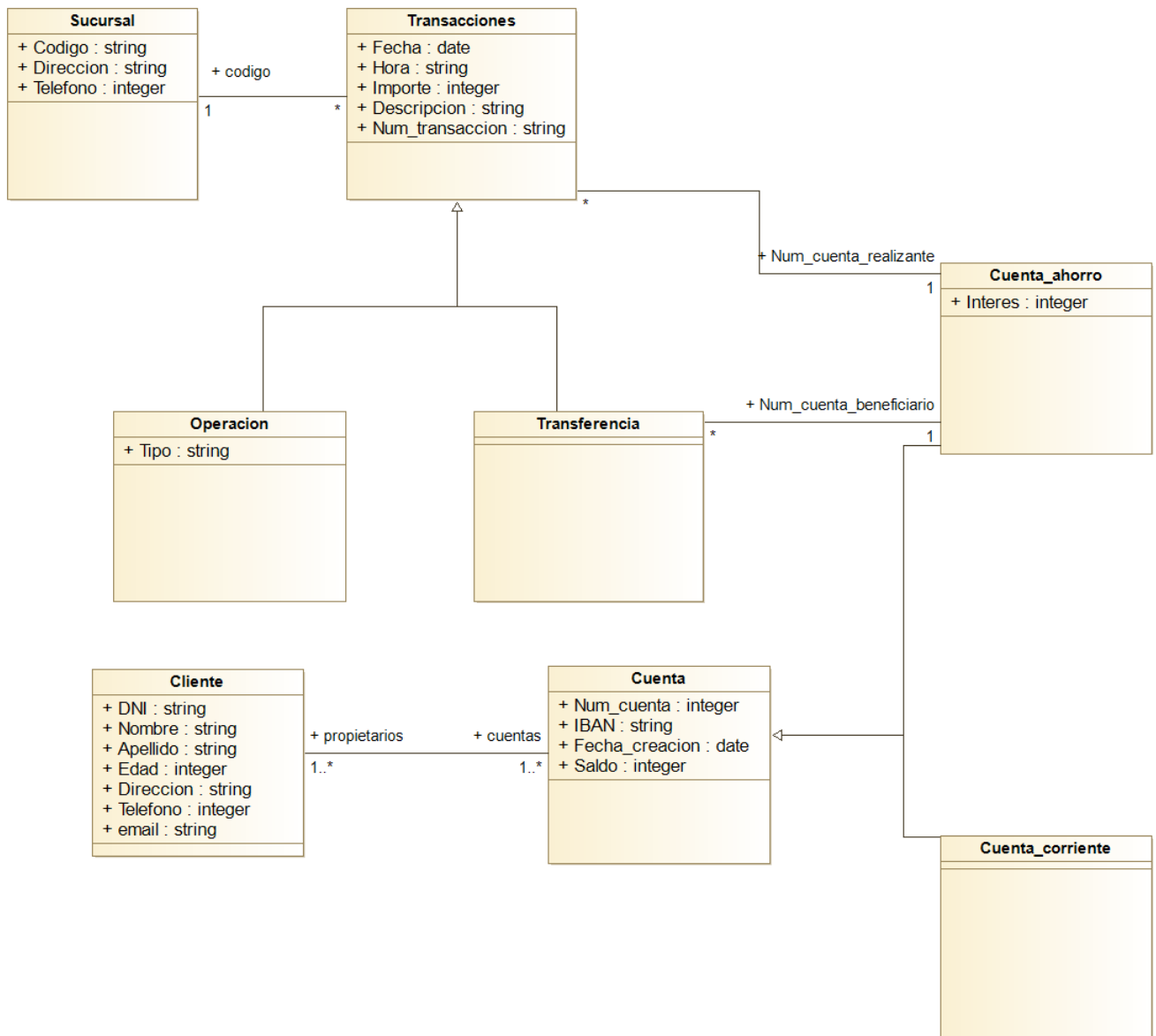


Figura 2: Diagrama de clases UML de la Base de Datos bancaria de *Banquito*

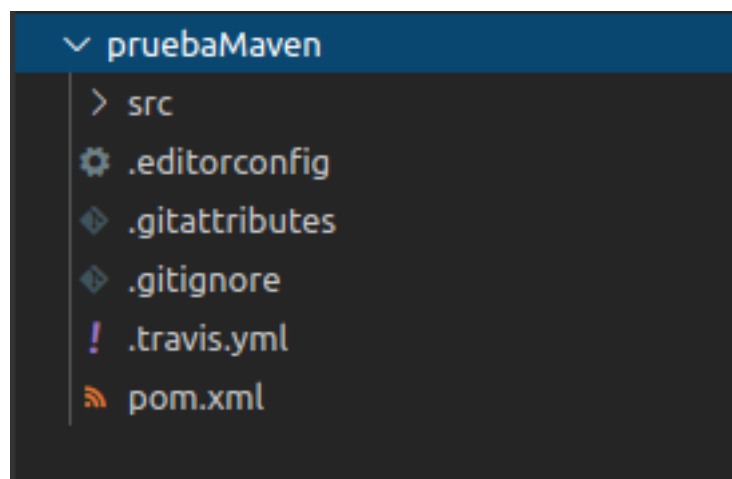


Figura 3: Estructura de un proyecto con la herramienta *Apache Maven*

sonatype | The Central Repository Quick Stats Who is Sonatype? Report A Vulnerability GitHub


com.oracle.database.jdbc:ojdbc8:19.6.0.0

com.oracle.database.jdbc:ojdbc8: 19.6.0.0

Browse Downloads

ojdbc8

Oracle JDBC Driver compatible with JDK8, JDK9, and JDK11

 **com.oracle.database.jdbc:ojdbc8** 19.6.0.0

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.oracle.database.jdbc</groupId>
  <artifactId>ojdbc8</artifactId>
  <version>19.6.0.0</version>
  <packaging>jar</packaging>

  <!-- Start elements that need to be inserted -->
  <!-- Start of fields requiring product specific values -->
  <name>ojdbc8</name>
  <description>Oracle JDBC Driver compatible with JDK8, JDK9, and JDK11</description>
```

Apache Maven
maven.apache.org

```
<dependency>
  <groupId>com.oracle.database.jdbc</groupId>
  <artifactId>ojdbc8</artifactId>
  <version>19.6.0.0</version>
</dependency>
```

Gradle Groovy DSL
gradle.org

```
implementation 'com.oracle.database.jdbc:ojdbc8:19.6.0.0'
```

Gradle Kotlin DSL

Figura 4: Resultado de la búsqueda del Driver *JDBC* de *Oracle* en *Maven Central*