GameX

TAREA AED UT3 G5: MANUAL DEL DESARROLLADOR

LingCheng Jiang
IES EL RINCON | 2ºDAMT AED

Contenido

0. Introducción

0.1. Enunciado

A partir del enunciado que se les ha facilitado a cada grupo, crear el diagrama E-R necesario para planificar la BD, a continuación, realizar la correspondiente BD en SQL y crear una pequeña aplicación usando Maven, para probar el acceso desde Java mediante el conector JDBC a la base de datos probando todas las funcionalidades básicas (CRUD).

Debes entregar un solo archivo comprimido con todos los archivos, solo hay que hacer una entrega por grupo, además debes especificar todos los miembros del grupo con nombre, apellido y curso en un archivo .PDF dónde explicas brevemente en que consiste tu aplicación y como la has creado, y un pantallazo del gestor de tareas. Además, debes entregar como anexo a lo anterior el manual de instalación y es de usuario.

Habrá una exposición-debate del E/R, Esta será de máximo 10 min y su objetivo es estar seguros de que el diseño es correcto. La aplicación debe mostrarse funcionando en clase a la profesora, antes del examen de esta unidad.

El nombre del fichero solo debe poner por ejemplo Grupo1.zip

0.2. Especificación de los requisitos

1.Encargo para el Diseño de un Modelo Entidad-Relación para la Gestión de un Servicio de Alquiler de Juegos de Consolas

La empresa "GameX" está desarrollando un nuevo proyecto de servicio de alquiler de juegos de consolas y requiere un modelo entidad-relación para su sistema de gestión. Este modelo debe permitir una eficiente administración de la información relacionada con los juegos, los alquileres y los clientes frecuentes.

Detalles importantes a considerar:

Gestión de Juegos de Consolas: Se debe registrar información relevante de cada juego, incluyendo título, plataforma, género y disponibilidad en inventario.

Alquileres y Devoluciones: Los clientes pueden alquilar juegos por un período estándar de 3 días. Se registrará la fecha de alquiler y se calculará la fecha límite de devolución. Se aplicará un precio por retraso en la devolución de los juegos.

Ventaja para Clientes Frecuentes: Los clientes que alquilen más de un juego semanalmente tendrán la opción de extender el período de alquiler a 10 días sin costo adicional.

El modelo entidad-relación debe representar de manera clara y precisa la relación entre las entidades mencionadas, así como los atributos y restricciones asociadas a cada una, proporcionando una base sólida para el desarrollo del sistema de gestión del servicio de alquiler de juegos de consolas de "GameX".

1. Base de Datos

1.1. Planificación

Según las especificaciones, se considera una aplicación de complejidad baja, por lo que se baraja usar gestores de base de datos pequeños, entre los que se encuentra usar MySQL/MariaDB, descartando gestores pesados como PostgreSQL.

Suponemos que la empresa en un futuro tiene posibilidad de expansión, por lo que se descarta bases de datos no SQL, además de que uno de los requisitos es usar base de datos relacionales (Entidad/relación).

El SQL que se escriba será de la manera más neutra posible, para que en un futuro la empresa crezca pueda migrar fácilmente a un gestor de BD más potente en consultas complejas.

Además, para mantener un orden, se seguirá la siguiente notación:

Entidades y Relaciones: Pascal Case

Atributos: Camel Case.

Crearemos directamente la base de datos en MySQL con todo lo necesario, por lo que en el programa solo accederemos a los procedimientos o funciones expuestas en la BD, así aumentaremos aún más la seguridad de los datos, y concentraremos la carga principal en el servidor.

1.2. Identificando entidades y relaciones

De la descripción de los requisitos, se puede identificar 3 entidades, siendo:

Juego (idJuego, titulo, plataforma, genero, disponibilidad, precio)

Cliente (idCliente, nombreCompleto, dni, telefono, email, direccion, esFrecuente)

Alquiler (idAlquiler, fechaAlquiler, fechaLimite, fechaDevolucion, hayRetraso, multaRetraso, idCliente*)

Cabe a destacar que se supone que los posibles enums/entidades debiles (genero, estado, plataforma) se simplifican a string/varchar.

Se da a entender que un cliente puede realizar varios pedidos(alquileres) pero puede existir clientes con alquiler (se da de alta pero no alquila nada de momento), y un

alquiler debe tener un cliente para crearse y este solo puede estar relacionado a un solo cliente.

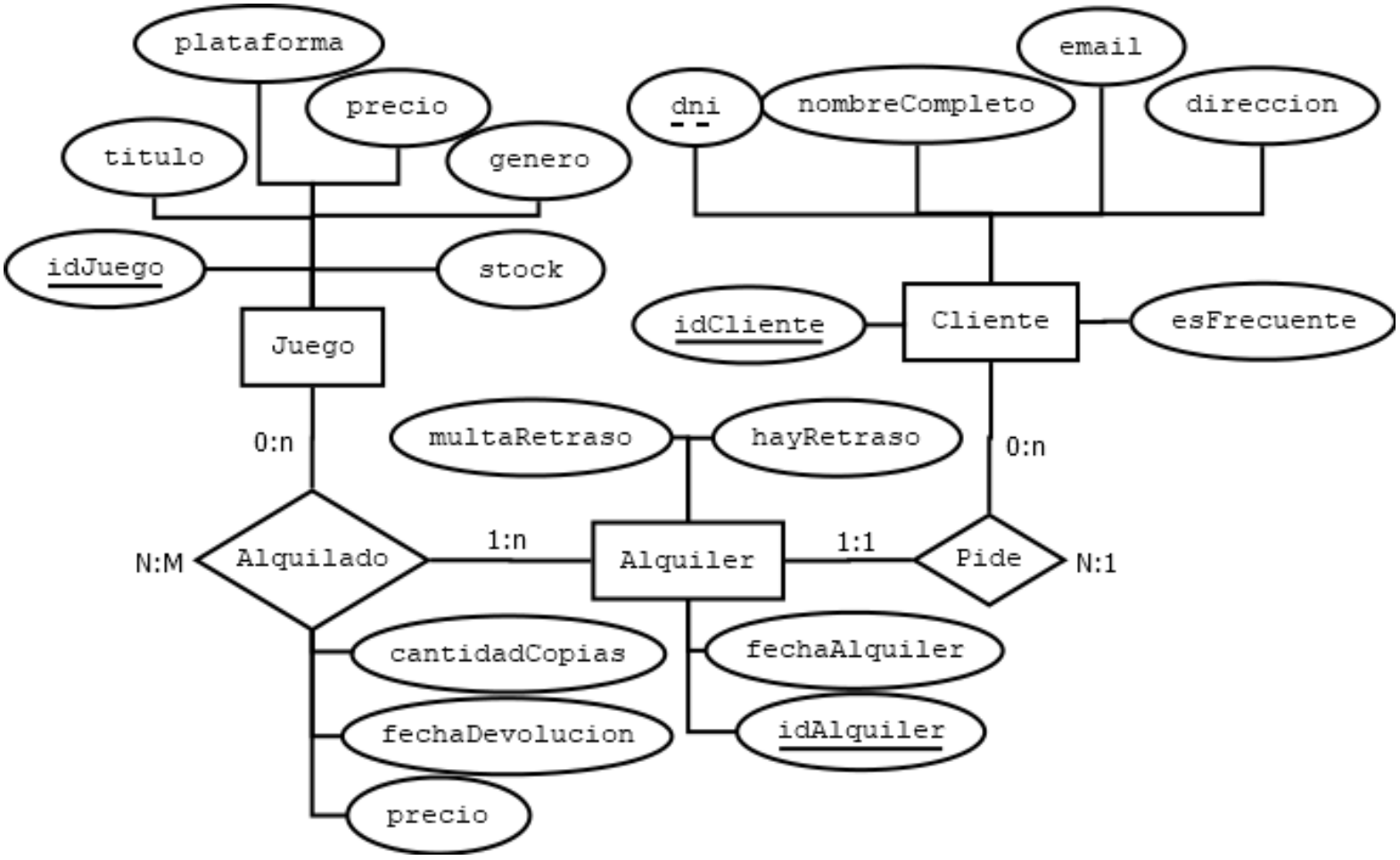
Además, un juego puede no estar en ningún alquiler (es nuevo y no se ha alquilado aun) y puede estar en varios alquileres en distintas fechas. Y un alquiler debe tener al menos un juego, pero al mismo tiempo en un alquiler se puede alquilar varios juegos.

No hay relación directa entre cliente y juego, todo se conecta por alquiler.

Por otra parte, al ser una relación N:M, se deberá crear otra tabla para que esta relación sea consistente, donde lo llamaremos como la relación: “Alquilado”, este guardará los datos del videojuego alquilado en relación al alquiler, su fecha de devolución, cantidad de copias prestada.

Es posible que hayan otros aspectos que no se ha tenido en cuenta, pero para prototipado, se considera suficiente los presuntos.

1.3. Modelo E/R



1.4. Diseño SQL

Con la entrega se adjuntará el archivo GameX.sql completo, en este documento solo se hablará de los aspectos más relevantes.

Este archivo estará diseñado para tener 3 subsecciones:

- Creación de BD y tablas (CREATE TABLES)
- Inserción de datos de prueba (INSERTs)
- Creación de procedimientos y/o funciones (CREATE PROCEDUREs)

Para cada tabla insertaremos unas 3 entidades de pruebas aprox. Crearemos procedimientos según las necesidades del programa ya que todos los cálculos se harán a nivel de base de datos por temas de seguridad y encapsulamiento.

para un primer planteamiento, crearemos los procedimientos:

- GetGamesByTitle()
- GetClientByNameOrDNI()
- GetClientByNameOrDNI()
- AddClient()
- AddRental()
- AddGameToRental()
- DeleteGameById()
- DeleteClientById()

2. Aplicación

La aplicación estará hecha usando la IDE IntelliJ con JavaFX y como requisito del enunciado Maven.

Como el trabajo es en JavaFX, se usará fxml, pero para trabajar de manera gráfica, usaremos Scene Builder, para poder arrastrar y diseñar la GUI de manera más fácil y hacer los retoques finales con fxml desde el proyecto.

La estructura del proyecto será con un MainView único y con varios controles personalizados.

Para centrarnos más en el backend, obviaremos todas las otras clases .java que están relacionadas a controlar el frontend, y nos centraremos en /Services/DBConnector.java, el conector de la base de datos, y los modelos en /Models/ Client.java, Rental.java y Game.java.

2.1. DBConnector

Esta será una clase donde se implementa un patrón singleton, es decir, que lo diseñaremos como una única instancia en el programa. Por lo que dentro de la clase tendrá una variable:

```
public static DBConnector instance;
```

Donde en el constructor al crear una nueva instancia dirá que esa será la única instancia valida.

Así desde cualquier lado del programa se podrá acceder a esta clase con DBConnector.instance.*.

Aparte, en esta clase estará unas variables constantes:

```
final String database = "GameX";
final String urlBase = "jdbc:mysql://" + "localhost" + ":" + 3306 + "/";
final String user = "GameX";
final String password = "!StrongPassword123";
```

Ahora, empezamos con el flujo de trabajo de la clase.

Al iniciar la app, se crea una instancia de la clase, este, intentará conectarse con las credenciales proporcionadas en las constantes, pero si falla, intentará crear la base de datos. Ya que se supone que el motor de base de datos está instalado y el usuario GameX también.

```
public void connect() {  2 usages  YarCrasy
    String dbUrl = urlBase + database;
    try {
        conn = DriverManager.getConnection(dbUrl, user, password);
    } catch (SQLException e) {
        if (!initDatabase()) System.err.println("Error connecting to database: " + e.getMessage());
    }
}

boolean initDatabase() {  1 usage  YarCrasy
    try {
        conn = DriverManager.getConnection(urlBase, user, password);
        String sql = "CREATE DATABASE IF NOT EXISTS " + database + ";";
        conn.createStatement().execute(sql);
        conn.close();
        conn = DriverManager.getConnection( url: urlBase + database, user, password);
    } catch (SQLException e) {
        System.err.println("Error initializing database: " + e.getMessage());
        return false;
    }

    try {
        setupDBFromFile();
    } catch (IOException | SQLException e) {
        System.err.println("Error reading SQL file: " + e.getMessage());
        return false;
    }
    return true;
}
```

Esto, intentará leer el archivo GameX.sql que ya está creado de antes, e irá ejecutando los querys uno a uno.

```
void setupDBFromFile() throws IOException, SQLException {  1 usage  YarCrasy
    File file = new File( pathname: "src/main/resources/com/yarcrasy/gamex/GameX.sql");
    StringBuilder sb = new StringBuilder();
    String delimiter = ";";

    FileReader fr = new FileReader(file);
    BufferedReader br = new BufferedReader(fr);
    String line;
    while ((line = br.readLine()) != null) {
        String trimmed = line.trim();
        if (trimmed.isEmpty() || trimmed.startsWith("--") || trimmed.startsWith("#")) {
            continue;
        }
        if (trimmed.toUpperCase().startsWith("DELIMITER")) {
            String[] parts = trimmed.split( regex: "\\s+",  limit: 2);
            delimiter = parts.length > 1 ? parts[1] : ";";
            continue;
        }
        sb.append(line).append("\n");
        String accumulated = sb.toString().trim();
        if (accumulated.endsWith(delimiter)) {
            String statement = accumulated.substring(0, accumulated.length() - delimiter.length()).trim();
            if (!statement.isEmpty()) {
                conn.createStatement().execute(statement);
            }
            sb.setLength(0);
        }
    }
    String leftover = sb.toString().trim();
    if (!leftover.isEmpty()) {
        conn.createStatement().execute(leftover);
    }
}
```

Además se crearán funciones con los modelos creados con setup…FromResultSet().

```
Game setupGameFromResultSet(ResultSet rs) throws SQLException {  2 usages  YarCrasy
    return new Game(
            rs.getInt( columnLabel: "idJuego"),
            rs.getString( columnLabel: "titulo"),
            rs.getString( columnLabel: "plataforma"),
            rs.getString( columnLabel: "genero"),
            rs.getInt( columnLabel: "stock")
    );
}
```

```
Client setupClientFromResultSet(ResultSet rs) throws SQLException {  2 usages  YarCrasy
    return new Client(
            rs.getString( columnLabel: "idCliente"),
            rs.getString( columnLabel: "dni"),
            rs.getString( columnLabel: "nombreCompleto"),
            rs.getString( columnLabel: "email"),
            rs.getString( columnLabel: "direccion"),
            rs.getBoolean( columnLabel: "esFrecuente")
    );
}
```

```java
Rental setupRentalFromResultSet(ResultSet rs) throws SQLException {  1 usage  YarCrasy
    Rental r = new Rental(
            rs.getInt( columnLabel: "idAlquiler"),
            rs.getInt( columnLabel: "idCliente"),
            rs.getDate( columnLabel: "fechaAlquiler").toLocalDate()
    );
    r.setDelayFee(rs.getFloat( columnLabel: "multaRetraso"));
    return r;
}
```

A partir de ahí, se irán creando las funciones a medida que se van requiriendo, llamando siempre a los procedimientos guardados. Ejemplo de uso:

```java
public List<Rental> getAllRentals() {  1 usage  YarCrasy
    List<Rental> rentals = new ArrayList<>();
    try {
        PreparedStatement ps = conn.prepareStatement( sql: "SELECT * FROM Alquiler;");
        ResultSet rs = ps.executeQuery();
        while (rs.next()) {
            rentals.add(setupRentalFromResultSet(rs));
        }
        return rentals;
    } catch (Exception e) {
        System.err.println(e.getMessage());
    }
    return rentals;
}

public boolean addGame(String title, String platform, float price, String genre, int stock) {
    try (PreparedStatement ps = conn.prepareStatement( sql: "CALL AddGame(?, ?, ?, ?, ?)")) {
        ps.setString( parameterIndex: 1, title);
        ps.setString( parameterIndex: 2, platform);
        ps.setFloat( parameterIndex: 3, price);
        ps.setString( parameterIndex: 4, genre);
        ps.setInt( parameterIndex: 5, stock);
        ps.executeQuery();
        ps.close();
        return true;
    } catch (Exception e) {
        System.err.println(e.getMessage());
    }
    return false;
}
```

Instalador

Como último paso, para dejar la instalación lo más fácil posible para el usuario final, configuraremos un instalador con Inno Setup. Donde lo programaremos para:

1. Detectar si **MySQL** está instalado.

    1.1. Si lo está, muestra un panel en el instalador diciendo pidiendo las credenciales del servidor MySQL

        1.1.1. al presionar el botón de continuar, hará un crearte user para crear el usuario de la app con el usuario proporcionado.

        1.1.2. Si hubo algún error, pedir al usuario que verifique si el usuario existe o si tiene permisos para crear.

    1.2. si no, descarga de la página oficial de MySQL community y configura automáticamente el server para que:

    - se ejecute al iniciar el PC;
    - el usuario “root” con contraseña "123"
    - otro usuario de la base de datos que usará desde la app.

2. Detectar si el **JRE** está instalado y configurado como predeterminado para abrir archivos .jar

    2.1. Si lo está, obviamos este paso.

    2.2. Si no, descarga el JRE, y lo configura como app predeterminada.

3. comprueba si en la ruta de instalación ya está instalado la app **GameX**.

    3.1. en caso afirmativo, preguntar si lo quiere sobre escribir.

    3.2. en caso contrario instalar la app en la ubicación predeterminada a menos de que el usuario elija otro directorio.