Contenido

1. Diagrama de clases del dominio 3
2. Diagrama de secuencia 3
3. Arquitectura aplicación 5
4. Explicación patrones 6
5. Explicación componentes 8
6. Test unitarios 9
7. Manual de usuario 13
8. Contenido extra 19
9. Observaciones finales 20

1. Diagrama de clases del dominio

Respecto al diagrama de clases, aunque al principio teníamos nuestro diagrama, al final decidimos optar por la versión del profesor, puesto que nos parecía la versión más adecuada, por lo que la implementación se basa en el diagrama de clases proporcionado por el profesorado. Aunque hay que decir que hemos cambiado ciertos detalles respecto al diseño original.

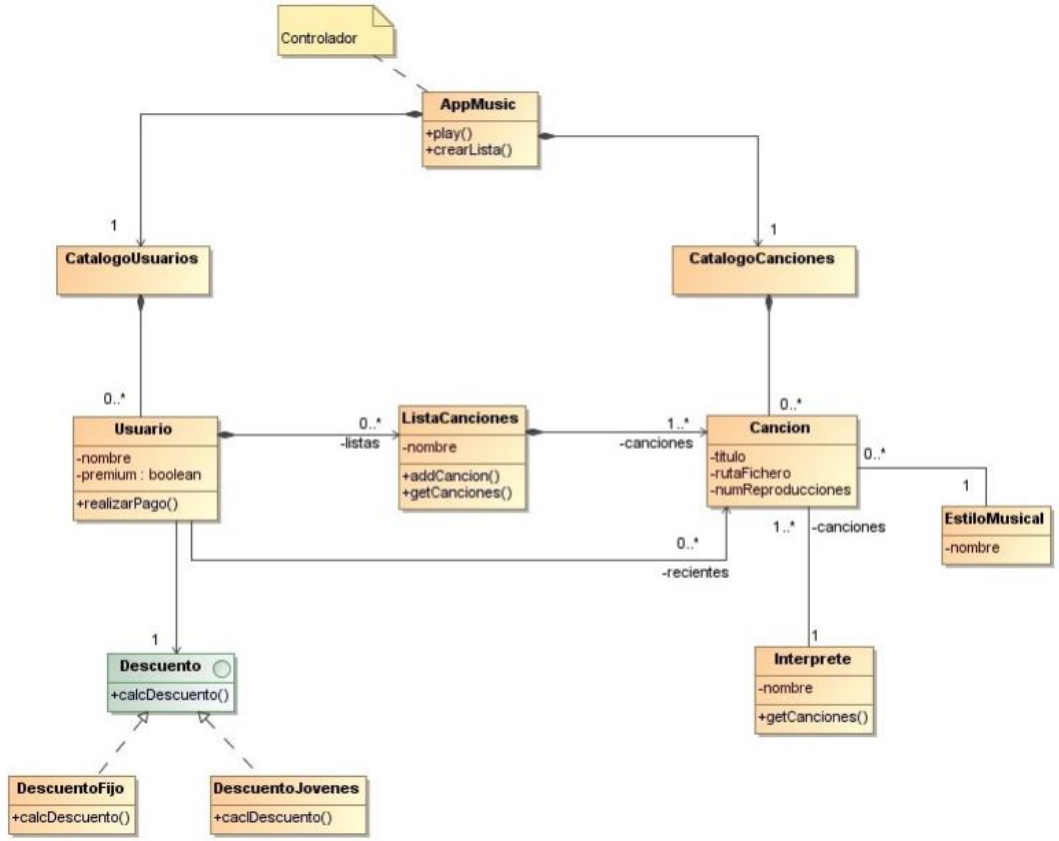

Figura 1. Diagrama de clases

Estos detalles que difieren respecto al diagrama de clases original son los siguientes:

- Canciones recientes: Hemos decidido que las canciones recientes reproducidas, sean una lista por defecto que obtiene el usuario nada más registrarse, que al principio estará vacía y se ira rellenando conforme el usuario reproduzca canciones, esta lista ni se podrá editar ni borrar.

En cambio, los detalles que hemos mantenido igual son los que se van a mostrar a continuación:

- Estilo musical. En el diseño de clases del dominio de una aplicación es común tener que decidir si una información se representa como un atributo de tipo primitivo o una clase o un enumerado. En nuestro caso hemos decidido implementarlo como un atributo de la clase canción, que será un string.
- Las clases persistentes serán Usuario, Canción, y ListaCanciones, es decir serán las clases que se mantienen en la base de datos, que en nuestro caso es la H2.

2. Diagrama de secuencia

Hemos realizado el diagrama de secuencia en un fichero *diagramaSecuencia.astah* y en formato *.pdf* (adjuntos ambos archivos en el fichero .zip de la entrega) ya que si lo ponemos como capturas de pantalla se vería muy pequeño y no permite su correcta visualización.

De todas formas, vamos a explicar paso por paso qué pasa en nuestro programa a la hora de la creación de una nueva Playlist:

- Cuando el usuario pulsa el botón crear en la VentanaNuevaLista, se llama al controlador para que compruebe si el nombre de la Playlist ya existe en el catálogo recuperado de la base de datos, el controlador llama al usuario y es él quien comprueba si, efectivamente, ya existe una Playlist creada con dicho nombre (devolviendo un booleano).

- Dependiendo del resultado del booleano, se procederá a hacer una cosa u otra. En primer lugar, si la variable es true, es decir, si el nombre insertado corresponde con una lista ya existente, nuestro programa lo notificará y procederá a recuperarla del catálogo para poder ofrecer al usuario la funcionalidad de modificar dicha Playlist. Para esto, se llama otra vez al controlador que llamará a su vez al usuario para que devuelva la Playlist con el nombre que nos indicó el usuario. Finalmente, para cada canción de la Playlist que nos devuelve el usuario, llamamos al controlador para que, dada una lista de intérpretes, nos imprima y nos rellene la información de cada canción que vamos a mostrar al usuario en la tabla de canciones ya añadidas en la Playlist.

- Si por otro lado el nombre que nos facilita el usuario no coincide con una Playlist ya creada (el resultado del booleano sería falso) entonces se abrirá un panel al usuario para poder buscar y filtrar canciones por título, intérprete y estilo. Antes de filtrar las canciones, VentanaNuevaLista se comunica con el controlador para comprobar tanto el intérprete como el título, este método comprueba que los Strings título e intérprete no estén rellenos con "TÍTULO" e "INTÉRPRETE" respectivamente (esos Strings son los que vienen por defecto en la ventana gráfica). Si coinciden con los String por defecto, es decir, el usuario no quiere buscar por título ni por intérprete, se llamará al filtrar canciones con esos argumentos igualados a "null" para que no se filtre por ellos.

- Una vez hecha la dicha comprobación del título y del intérprete. VentanaNuevaLista llama de vuelta al controlador para que filtre las canciones por título, intérprete y por estilo. Este, se encargará de llamar nuevamente al Catálogo de Canciones que será el encargado de filtrar utilizando streams, devolviendo una Lista de Canciones con las canciones que cumplan con los criterios indicados por el usuario.

- Utilizando esta lista, VentanaNuevaLista llama al controlador para que, para cada canción, imprima la lista de autores y el título y de esta manera, rellenar la tabla en la interfaz gráfica que muestra la información de las canciones que el usuario ha buscado.

- Una vez el usuario haya seleccionado las canciones que quiere añadir a su nueva Playlist, VentanaNuevaLista llama al controlador que llamará respectivamente al Catálogo de Canciones

pasando el conjunto de canciones seleccionadas por el usuario (utilizamos un conjunto para evitar la repetición de una misma canción en una Playlist) y este conjunto, el controlador se encargará de transformarlo a lista para poder registrar la Playlist en la base de datos.

- Por último, una vez tenemos el nombre de la Playlist y la lista de canciones que el usuario ha seleccionado, VentanaNuevaLista llama al controlador que llamará al usuario que llamará a su vez al AdaptadorListaCancionesTDS, encargado de la persistencia. Esta clase creará un objeto de tipo ListaCanciones con el nombre y con la lista de canciones facilitadas por el usuario. Será este objeto de tipo ListaCanciones (clase que hace referencia a las Playlist) la que será registrada en base de datos.

- Cada vez que se modifica alguna clase persistente, se modifica en el catálogo y en la base de datos respectivamente, pero cuando se recupera un objeto se recupera del catalogo que se inicializó al encender la aplicación.

3. Arquitectura aplicación

Nuestra aplicación se basa en una arquitectura conocida como "Arquitectura MVC", que consta de tres capas (o 4, si incluimos la base datos):

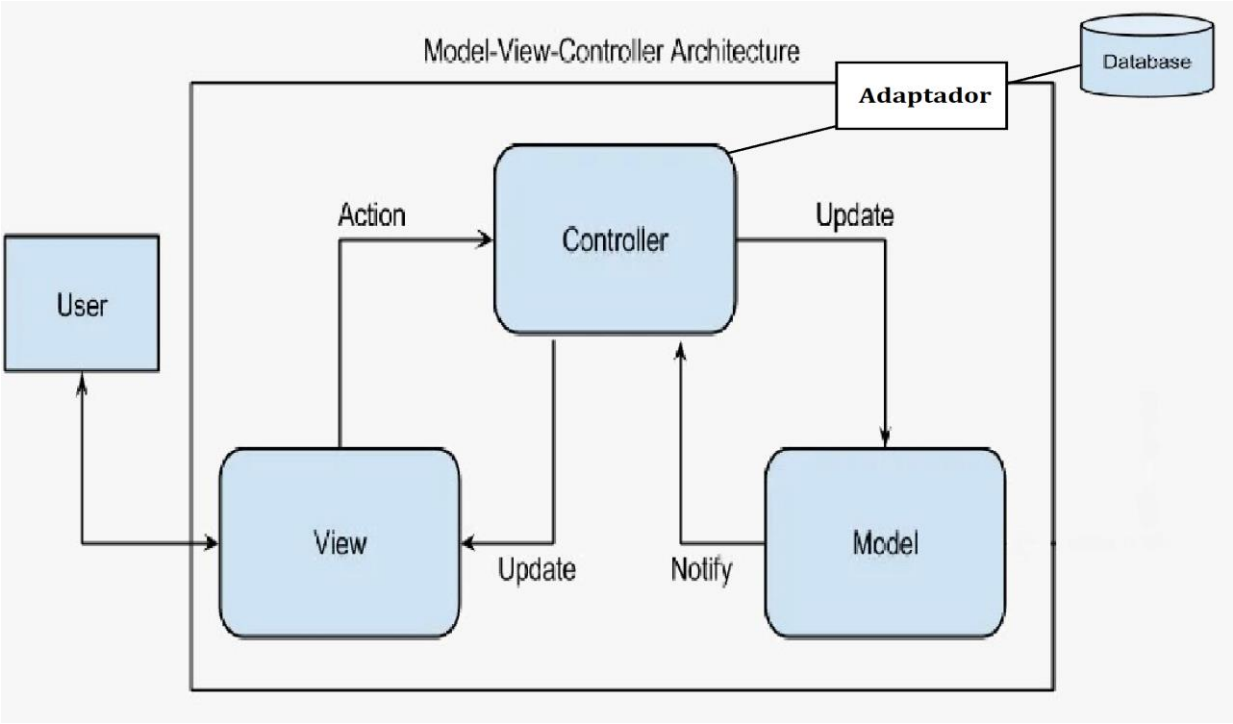

Figura 2: Arquitectura MVC

Esta arquitectura se basa en un modelo que interactúa con el usuario final gracias a las vistas, en nuestro caso práctico, todas las ventanas creadas gracias a JSwing. Estas vistas, ofrecen al usuario las funcionalidades de la aplicación mediante el uso de botones. Al pulsar alguno de estos botones, la vista

se comunica con el Controlador, clase que interactúa como intermediario y que es la clase que, realmente tiene los métodos que incluyen la funcionalidad de nuestra aplicación.

Además, si la vista quiere mostrar información al usuario, también llama al controlador para que este le devuelva la información requerida y la vista pueda mostrarla en la interfaz gráfica. Dicho controlador, se comunica y se abastece de las clases que le ofrece nuestro modelo, clase como "Usuario", "Canción" o "ListaCanciones". Estas clases representan los objetos al almacenar en la base de datos y con los que estamos trabajando en nuestra aplicación. Por último, el controlador, a través de los diversos adaptadores, es el encargado de llamar cuando sea necesario a la persistencia para poder almacenar o modificar los datos pertinentes.

4. Explicación patrones

En la realización de esta práctica, hemos aplicado estos patrones:

- Patrón singleton: este patrón de creación de objetos se utiliza a la hora de crear una instancia de algunas clases, en nuestro caso de las clases adaptador y del controlador de la aplicación. Se basa en restringir la creación de objetos de dichas clases a una única instancia haciendo que, si queremos acceder a cualquier funcionalidad de estas clases, solo podamos acceder a través de un punto de acceso global.

```
public static AppMusicControlador getInstancia() {
    if (unicaInstancia == null) {
        unicaInstancia = new AppMusicControlador();
    }

    return unicaInstancia;
}
```

Figura 3: Ejemplo de utilización del patrón singleton en controlador

- Factoría abstracta: este patrón de creación nos proporcionará una interfaz para crear familias de objetos relacionados o dependientes sin especificar la clase concreta. En nuestro caso práctico, sirve para poder agrupar los adaptadores.

- Adaptador: este patrón de diseño nos ayuda a convertir una interfaz de una clase a otra con la que nuestra clase cliente pueda trabajar. En nuestro caso práctico, este patrón nos va a ayudar a la hora de manejar la persistencia de la aplicación. Gracias a las clases adaptador tanto de usuario, de canción como de lista de canciones y a sus respectivas interfaces, podemos manejar de una forma más sencilla el servicio de persistencia

```
public void registrarCancion(Cancion cancion) {
    Entidad eCancion = null;
    boolean existe = true;

    // Si la entidad está registrada no la registra de nuevo
    try {
        eCancion = servPersistencia.recuperarEntidad(cancion.getCodigo());
    } catch (NullPointerException e) {
        existe = false;
    }

    if (eCancion == null) {
        existe = false;
    }
    if (existe) {
        return;
    }
    // crear entidad Cancion
    //Pasar la lista de interpretes a un solo string para meterlo en el servidor de persistencia"
    String listaInterpretes = "";
    for (String s : cancion.getListaInterpretes())
    {
        s.replace(" ", "_");
        listaInterpretes += s + ",";
    }
    eCancion = new Entidad();
    eCancion.setNombre("cancion");
    eCancion.setPropiedades(new ArrayList<Propiedad>(
            Arrays.asList(new Propiedad("titulo", cancion.getTitulo()),
                    new Propiedad("rutafichero", cancion.getRutaFichero()),
                    new Propiedad("numreproducciones", String.valueOf(cancion.getNumReproducciones())),
                    new Propiedad("estiloMusical", cancion.getEstiloMusical()),
                    new Propiedad("listaInterpretes", listaInterpretes))));

    // registrar entidad cancion
    eCancion = servPersistencia.registrarEntidad(eCancion);
    // asignar identificador unico
    // Se aprovecha el que genera el servicio de persistencia
    cancion.setCodigo(eCancion.getId());
}
```

Figura 4: Ejemplo del método registrarCancion en el adaptador para la persistencia

- Composite: este patrón es usado por nuestra interfaz gráfica, implementada gracias al uso de JSwing. Funciona de manera que nuestras ventanas gráficas están formadas por componentes que pueden contener más componentes dentro. Por ejemplo, un JPanel puede tener un JPanel dentro y así sucesivamente.

```
private void initialize() {
    frame = new JFrame();
    frame.setBounds(100, 100, 450, 300);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JLabel lblNewLabel = new JLabel("Bienvenidos a AppMusic");
    lblNewLabel.setFont(new Font("Tahoma", Font.BOLD, 17));
    lblNewLabel.setHorizontalAlignment(SwingConstants.CENTER);
    frame.getContentPane().add(lblNewLabel, BorderLayout.NORTH);

    JPanel panelCentro = new JPanel();
    frame.getContentPane().add(panelCentro, BorderLayout.CENTER);
    panelCentro.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

    JPanel panel = new JPanel();
    panel.setPreferredSize(new Dimension(300, 200));
```

Figura 5: Vemos que una ventana tiene un JFrame y dentro del JFrame tenemos un JPanel que contiene a su vez otro JPanel

- Observer: este patrón lo utilizamos para notificar el cambio de estado de un objeto a una lista de oyentes, en este caso, sólo al Adaptador, cuando ha cambiado la lista de canciones en el sistema.

```
@SuppressWarnings("unchecked")
private void notificarCambio(CancionesEvent event) {
    Vector<CancionesListener> lista;
    synchronized (this) {
        lista = (Vector<CancionesListener>) oyentes.clone();
    }
    for (CancionesListener cl : lista) {
        cl.enteradoCambioCanciones(event);
    }

}
```

Figura 6: Vemos el uso del método notificar cambio donde se avisa a la lista de oyentes del cambio

- Método plantilla: este patrón de diseño es usado para definir un esqueleto de programa de un método en concreto, la clase padre les da la responsabilidad a los distintos hijos de implementar los métodos dependiendo de la funcionalidad que quieran. En nuestro caso, la clase abstracta Descuento permite a sus hijos implementar el cálculo del descuento como quieran.

```
public abstract class Descuento {

    // Metodo que heredaran los hijos
    int descuento;

    abstract void calcDescuento();

    public int getDescuento() {
        return descuento;
    }
}
```

Figura 7: Utilización el método plantilla en el método abstracto calcDescuento()

5. Explicación componentes

En nuestra aplicación utilizamos estos componentes:

- Componente Luz (Pulsador): este componente, dado por el profesorado, es una especie de pulsador. En nuestro caso práctico, al pulsarlo, abre un JFileChooser que nos sirve para seleccionar el archivo .xml donde tenemos las canciones para cargar en la base de datos llamando al cargador de canciones.

- Cargador de canciones: este componente, creado por nosotros, sirve para pasar de un archivo .xml a objetos de tipo 'Cancion'. También, implementa una clase 'listener' y 'event' que sirven para avisar a cualquier oyente (en nuestro caso, solo el controlador) de que la lista de canciones ha cambiado y poder obtener la nueva.

- JCalendar: este componente se utiliza en la creación de ventanas con JSwing, es un calendario desplegable que sirve para poder seleccionar una fecha. En nuestro caso práctica, lo utilizamos para poder seleccionar la fecha de nacimiento de un usuario que se va a registrar en el sistema.

- Driver Persistencia: este componente, nos da los métodos que va a utilizar el adaptador para poder crear, modificar o eliminar entidades de una base de datos local. Esta base de datos local es con la que va a trabajar nuestra aplicación.

6. Test unitarios

Hemos implementado estas pruebas utilizando JUnit:

- Test Registrar Usuario: en esta prueba intentamos registrar un usuario que ya está registrado en base de datos, comprobamos que el método registrar usuario nos devuelve false indicando que no se puede registrar.

```
@Test
public void testRegistrarUsuario1() {

    String nombre = "carlos";
    String apellidos = "ortiz";
    String email = "carlos.ortizr@um.es";
    String login = "carlos";
    String password = "1";
    LocalDate fechaNacimiento = LocalDate.now();
    boolean resultado = false;
    assertEquals(resultado, AppMusicControlador.getInstancia().registrarUsuario(nombre, apellidos, email, login, password, fechaNacimiento));
}
```

Figura 8. Test Registrar Usuario

- Test Registrar Usuario v2: en esta prueba registramos un usuario en la base de datos. Posteriormente, lo recuperamos del catálogo y vemos si todas las propiedades registradas son las propiedades con las que le hemos registrado originalmente.

```
@Test
public void testRegistrarUsuario3() {

    String nombre = "Pepe";
    String apellidos = "Ortiz22";
    String email = "carlos.ortizr2322@um.es";
    String login = "carlosPrueba23232";
    String password = "1";
    LocalDate fechaNacimiento = LocalDate.now();


    AppMusicControlador.getInstancia().registrarUsuario(nombre, apellidos, email, login, password, fechaNacimiento);

    Usuario registrado = AppMusicControlador.getInstancia().recuperarUsuario(login);

    assertEquals(nombre, registrado.getNombre());
    assertEquals(apellidos, registrado.getApellidos());
    assertEquals(email, registrado.getEmail());
    assertEquals(login, registrado.getLogin());
    assertEquals(password, registrado.getPassword());
    assertEquals(fechaNacimiento, registrado.getFechaNacimiento());
}
```

Figura 9. Test Registrar Usuario v2

- Test Registrar PlayList: en esta prueba registramos una PlayList, luego recuperamos todas las PlayList de nuestra base de datos y recorremos la lista en busca de la PlayList que acabamos de registrar. Si existe, vemos si el nombre y la lista de canciones de la PlayList coinciden.

```java
@Test
public void testRegistrarPlayList() {

    AppMusicControlador.getInstancia().login("carlos", "1");
    List<String> listaInterpretes = new LinkedList<String>();
    listaInterpretes.add("Cosmin");

    Cancion cancion = new Cancion("Cosmin", "C//Cosmin", "Cosminstyle", listaInterpretes);

    List<Cancion> listaCanciones = new LinkedList<Cancion>();
    listaCanciones.add(cancion);

    String nombre = "JUnitPlaylist";
    ListaCanciones playlist = new ListaCanciones(nombre, listaCanciones);


    if(AppMusicControlador.getInstancia().comprobarListaYaExiste(nombre)) {

    }

    else {

        AppMusicControlador.getInstancia().registrarListaCanciones(playlist);
    }

    List<ListaCanciones> a = AppMusicControlador.getInstancia().recuperarTodasListasCanciones();

    ListaCanciones listaRegistrada = null;

    for(ListaCanciones lis: a) {

        if(lis.getNombre().equals(nombre)) {

            listaRegistrada = lis;


        }

    }
    assertEquals(nombre, listaRegistrada.getNombre());
    assertEquals(listaCanciones, listaRegistrada.getCanciones());
    AppMusicControlador.getInstancia().eliminarPlayList(nombre);
}
```

Figura 10. Test Registrar Playlist

- Test Login: en esta prueba intentamos logearnos con un usuario registrado en base de datos y comprobamos que el método login nos devuelve true indicando que hemos podido autenticarnos en el sistema.

```java
@Test
public void testLoginOK() {
    String login = "carlos";
    String clave = "1";

    boolean resultado = true;
    assertEquals(resultado, AppMusicControlador.getInstancia().login(login, clave));
}
```

Figura 11. Test Login

- Test Filtrar Canciones: en esta prueba, vamos a comprobar el método de filtrado de canciones de 3 maneras, filtrado por título, por intérprete y por estilo. Comprobamos que devuelven las canciones que están registradas en el sistema con los datos correspondientes.

```java
@Test
public void filtrarCanciones() {
    String interprete = null;
    String interprete1 = "King Crimson";
    String titulo = null;
    String titulo1 = "I'll take a melody";
    String estilo = null;

    String estilo1 = "Tango";

    List<Cancion> filtrado1 = CatalogoCanciones.getUnicaInstancia().filtrarCanciones(interprete, titulo, estilo1);
    List<Cancion> filtrado2 = CatalogoCanciones.getUnicaInstancia().filtrarCanciones(interprete, titulo1, estilo);
    List<Cancion> filtrado3 = CatalogoCanciones.getUnicaInstancia().filtrarCanciones(interprete1, titulo, estilo);
    Cancion cFiltrada1 = filtrado1.get(0);
    Cancion cFiltrada2 = filtrado2.get(0);
    Cancion cFiltrada3 = filtrado3.get(0);


    int longitud = 1;
    String resultado1 = "La Cumparsita";
    String resultado2 = "Todd Sheaffer";
    String resultado3 = "Rock-sinfonico";

    assertEquals(longitud, filtrado1.size());
    assertEquals(longitud, filtrado2.size());
    assertEquals(longitud, filtrado3.size());
    assertEquals(resultado1, cFiltrada1.getTitulo());
    assertEquals(resultado2, cFiltrada2.getListaInterpretes().get(0));
    assertEquals(resultado3, cFiltrada3.getEstiloMusical());

}
```

Figura 12. Test Filtrar Canciones

- Test Eliminar PlayList: en este test, registramos una playlist en el sistema y seguidamente la borramos con el método eliminar PlayList. Después, recuperamos todas las PlayList del sistema y comprobamos que no encontramos ninguna con el nombre de la PlayList registrada inicialmente.

```java
@Test
public void eliminarPlaylist() {

    List<String> listaInterpretes = new LinkedList<String>();
    listaInterpretes.add("Cosmin");

    Cancion cancion = new Cancion("Cosmin", "C//Cosmin", "Cosminstyle", listaInterpretes);

    List<Cancion> listaCanciones = new LinkedList<Cancion>();
    listaCanciones.add(cancion);

    String nombre = "JUnitEliminarPlaylist";
    ListaCanciones playlist = new ListaCanciones(nombre, listaCanciones);

    AppMusicControlador.getInstancia().registrarListaCanciones(playlist);

    AppMusicControlador.getInstancia().eliminarPlayList(nombre);

    List<ListaCanciones> a = AppMusicControlador.getInstancia().recuperarTodasListasCanciones();

    boolean resultado = false;

    for(ListaCanciones lis: a) {

        if(lis.getNombre().equals(nombre)) {

            resultado = true;


        }


}
assertEquals(resultado, false);

}
```

Figura 13. Test Eliminar PlayList

7. Manual de usuario

- Ventana Login: esta ventana nos sirve para poder ingresar nuestro usuario y contraseña para validarnos y poder entrar al menú principal de la aplicación. Consta también de otro botón para acceder a la ventana de registro. Por último, si nuestros datos introducidos son incorrectos nos saltará un mensaje notificándonos del error.

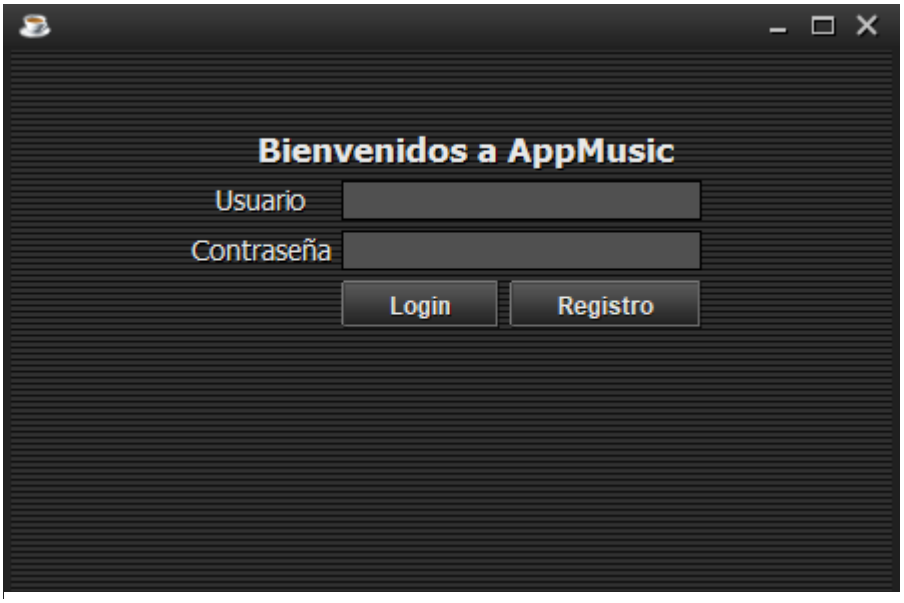

Figura 14. Ventana Login

- Ventana Registro: esta ventana sirve para poder registrar a un usuario, nos pide nuestros datos y los tenemos que introducir de manera correcta. Para seleccionar la fecha de nacimiento, tenemos un calendario desplegable gracias a JCalendar. El botón *cancelar* sirve para limpiar todos los campos de registro. Al pulsar *registrar*, el sistema procederá a registrar con esos datos al usuario. Si algún dato es incorrecto o si el usuario o email está ya registrado en base de datos, nos mostrará un error por pantalla.

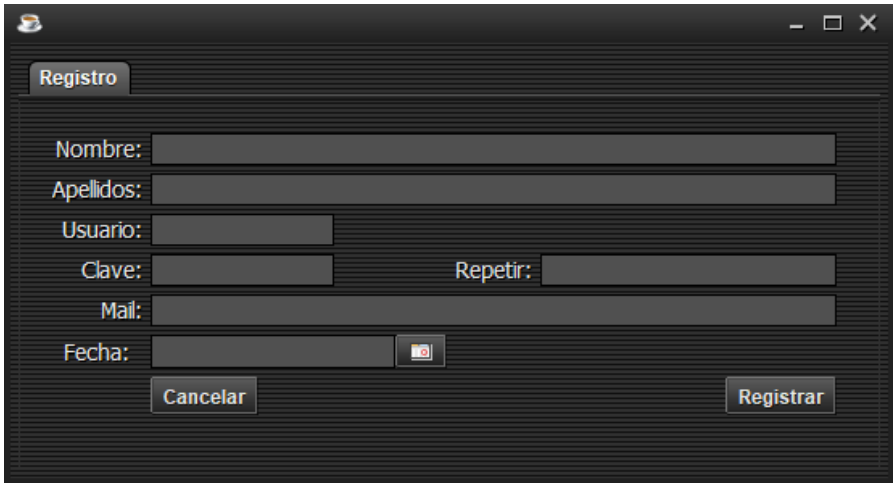

Figura 15. Ventana Registro

- Panel Principal: una vez estamos autentificados en nuestra aplicación, se nos mostrará esta ventana. Aquí podemos ver todas las ventanas disponibles en nuestra aplicación. Además, podemos entrar a *Mejorar cuenta* para poder pagar y convertirnos en usuario premium. También, la ventana consta de un pulsador para poder elegir las canciones y registrarlas en nuestra base de datos.

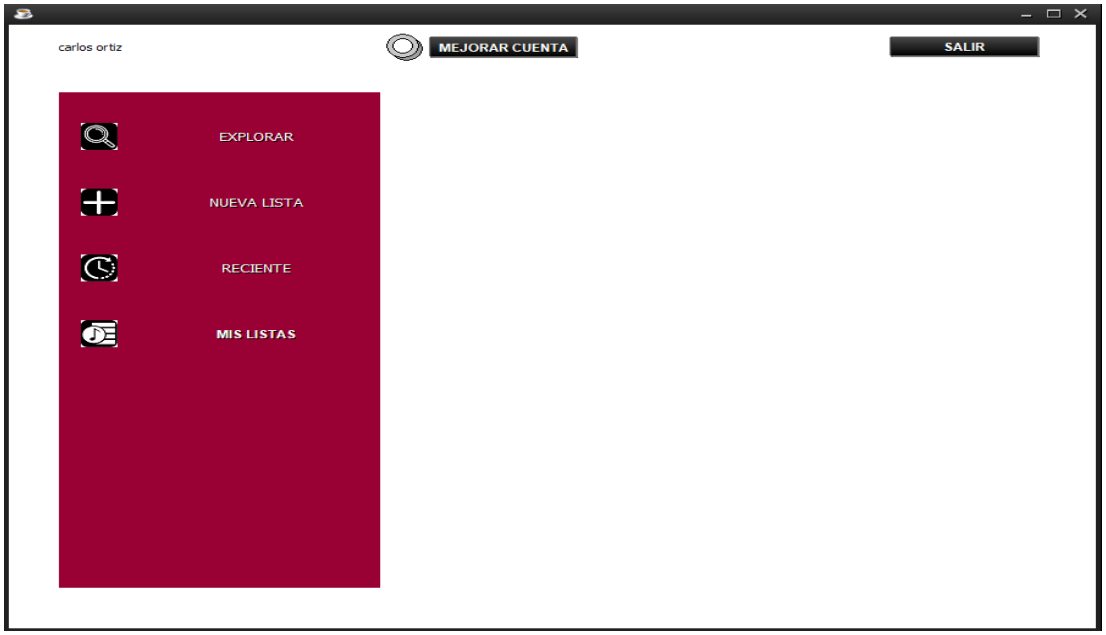

Figura 16. Panel Principal

- Ventana Explorar: esta ventana sirve para poder filtrar canciones por estilo, intérprete y título. Una vez seleccionamos lo que queramos, pulsamos el botón *buscar* y nos mostrará los resultados de nuestra búsqueda. Si no seleccionamos intérprete, título o estilo, obviará este criterio. También, nos da la opción de poder seleccionar una canción y poder reproducirla, pararla o ir a la canción anterior o posterior.

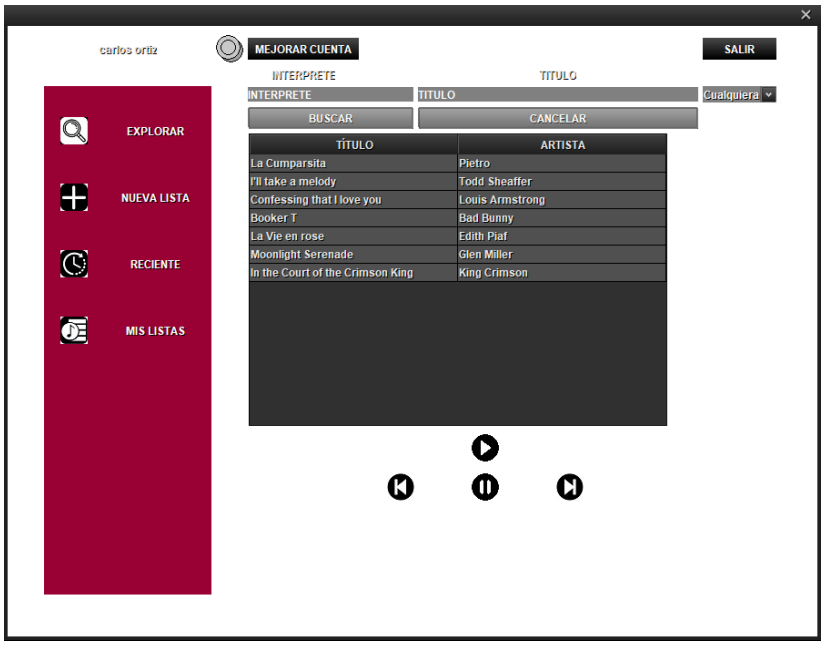

Figura 17. Ventana Explorar

- Ventana Nueva Lista: esta ventana nos pide primero insertar un nombre de lista. Si el nombre de la lista ya existe, se nos abrirá un panel para poder editar la playlist en cuestión o eliminarla. Si el nombre no existe, se nos abrirá el panel con las opciones para crear la playlist. Dentro de ambos paneles, podemos buscar canciones de manera similar a la ventana explorar para poder añadirlas a nuestra playlist.

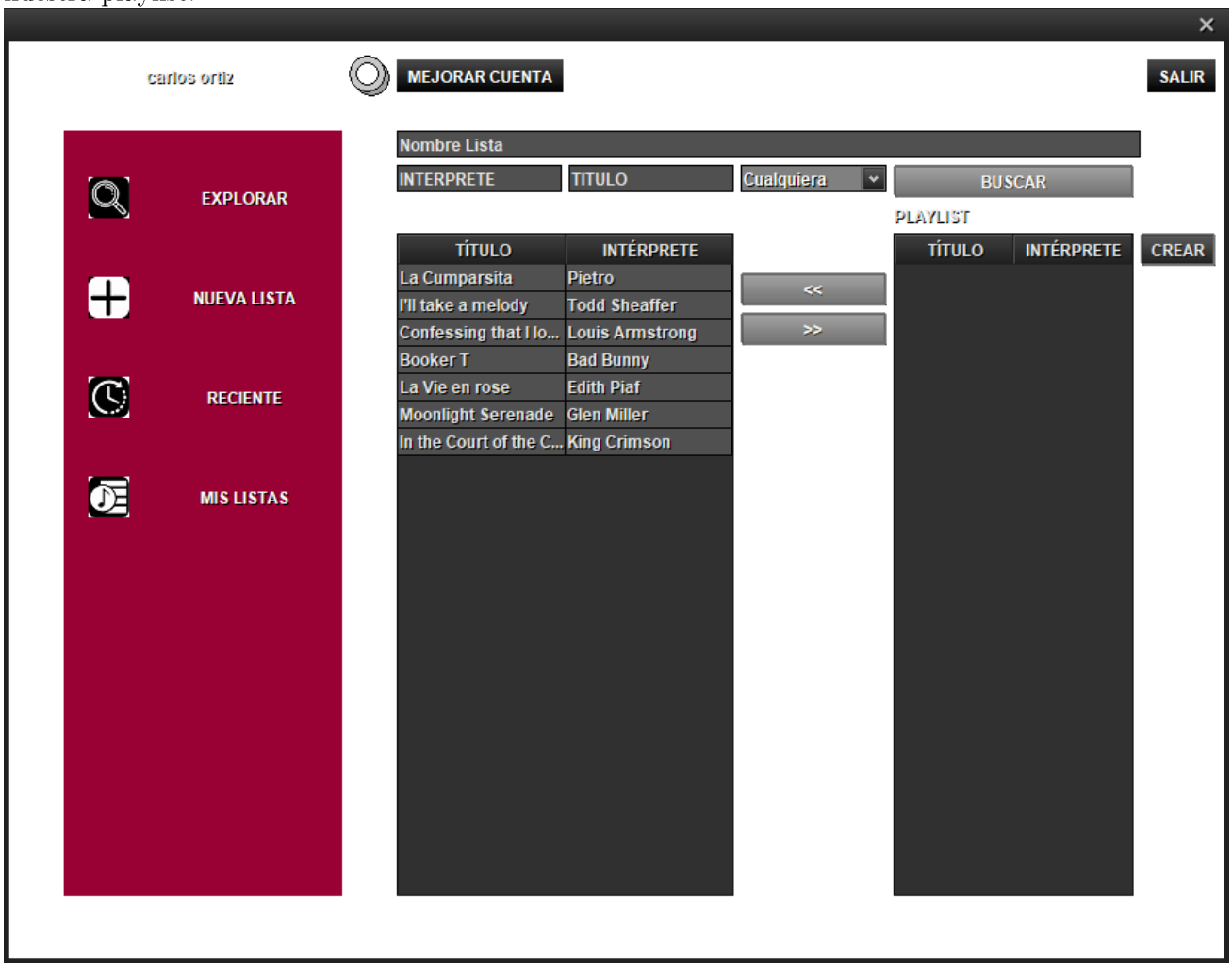

Figura 18. Ventana Nueva Lista

- Ventana Reciente: esta ventana nos muestra cuales son las canciones más recientes escuchadas por el usuario. También nos da la opción de seleccionar una canción y poder reproducirla, pararla o ir a la canción anterior o posterior.

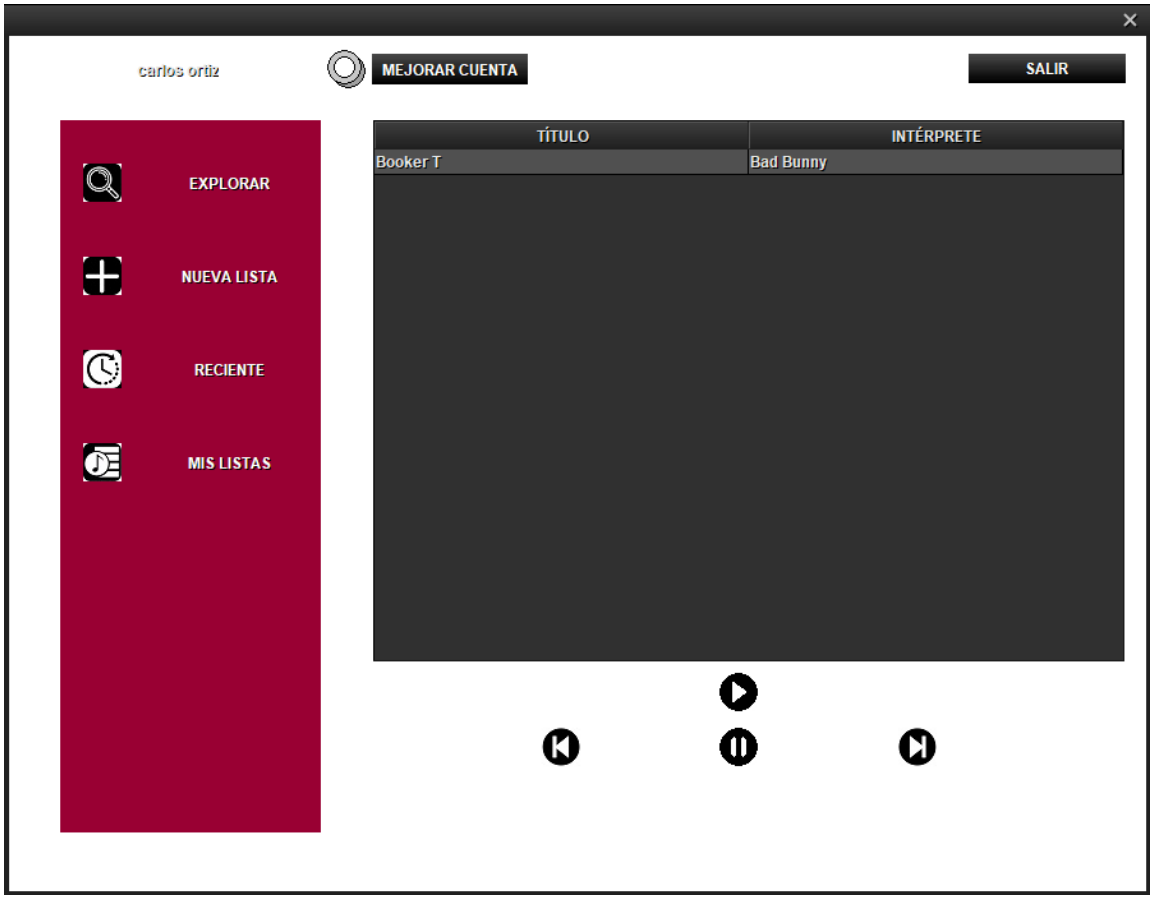

Figura 19. Ventana Reciente

- Ventana Mis Listas: esta ventana nos permite seleccionar abajo a la izquierda una de nuestras playlist creadas para poder ver la lista de canciones completas. Además, si el usuario es premium, podrá seleccionar la opción de *Generar PDF* para poder exportar la lista de playlist con sus canciones a un fichero .pdf Como en la ventana anterior, nos da la opción de seleccionar una canción y poder reproducirla, pararla, o ir a la canción anterior o posterior.

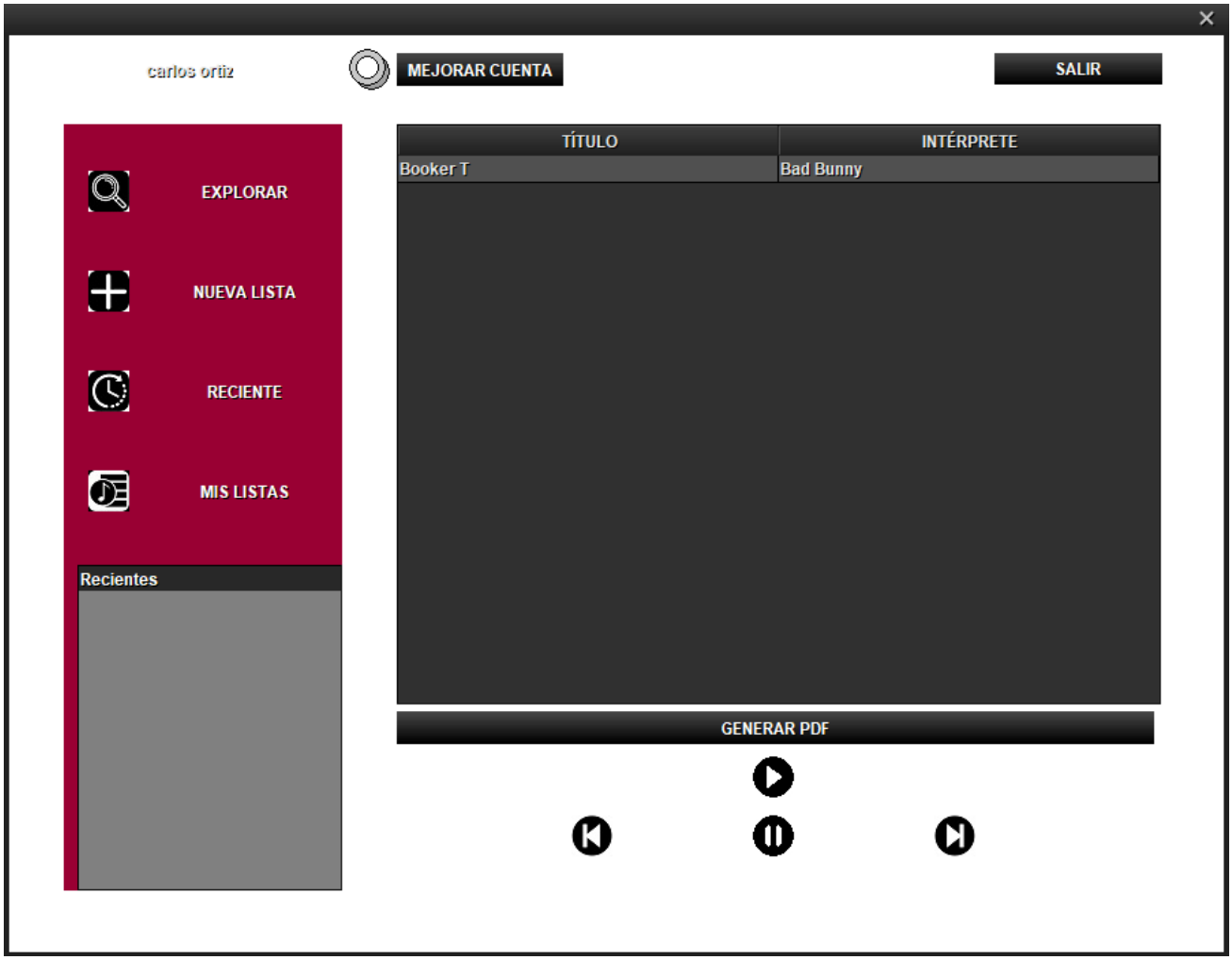

Figura 20. Ventana Mis Listas

- Ventana Mas Reproducidas: esta ventana, exclusiva para usuarios premium, nos permite ver cuáles son las canciones más reproducidas dentro de nuestra aplicación. Podremos ver para cada canción y el número de reproducciones de la misma. También, ofrece la oportunidad de seleccionar una canción y poder reproducirla, pararla, o ir a la canción anterior o posterior.

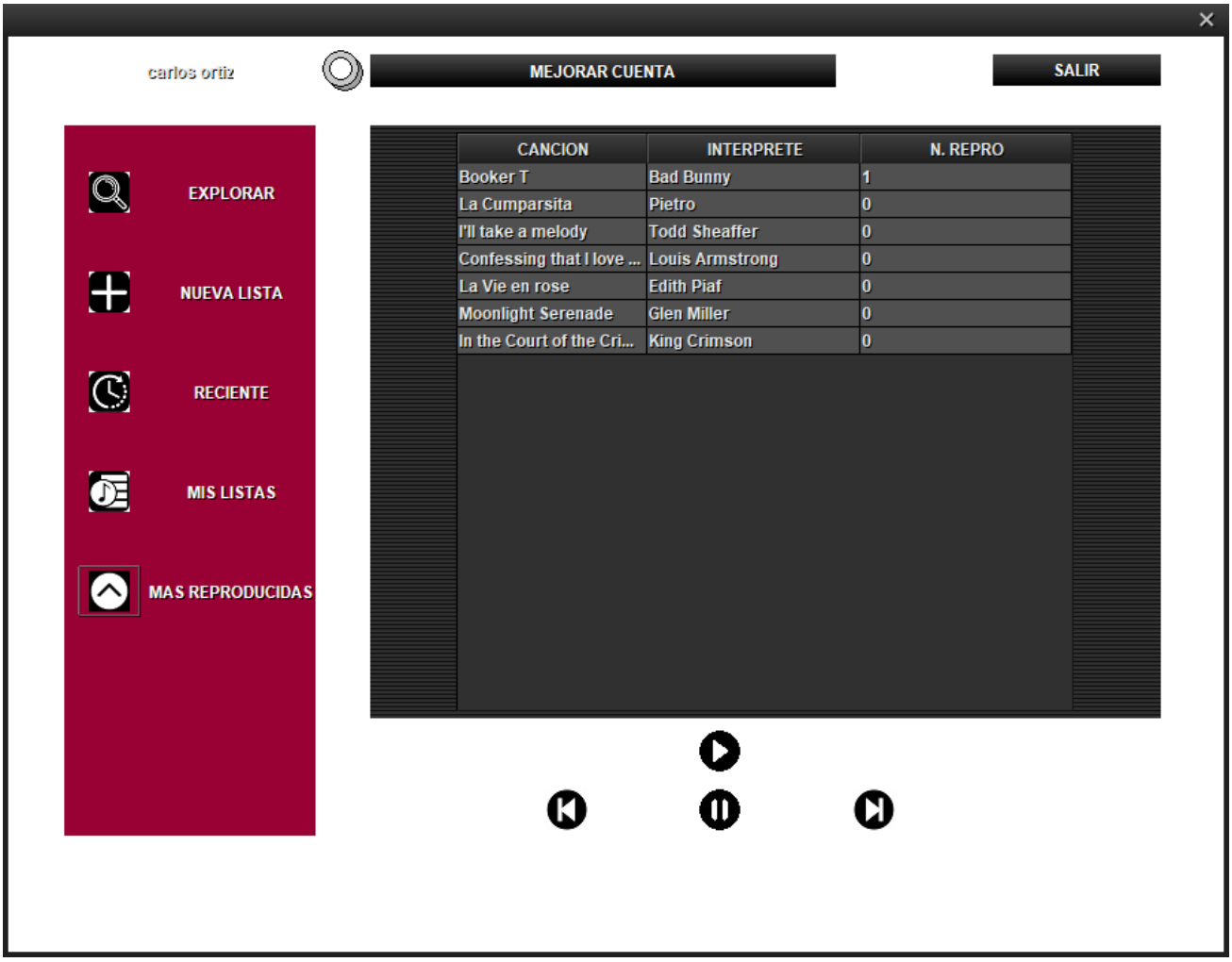

Figura 21. Ventana Mas Reproducidas

- Ventana Mejorar Cuenta: esta ventana, ofrece el servicio de poder convertirnos en premium o revocar nuestra suscripción. Podemos ver los descuentos disponibles, seleccionar uno con un desplegable y poder pulsar el botón *pagar* y así convertirnos en premium. Si queremos dejar de ser premium, podemos pulsar el botón *cancelar suscripción* para convertirnos en un usuario normal.

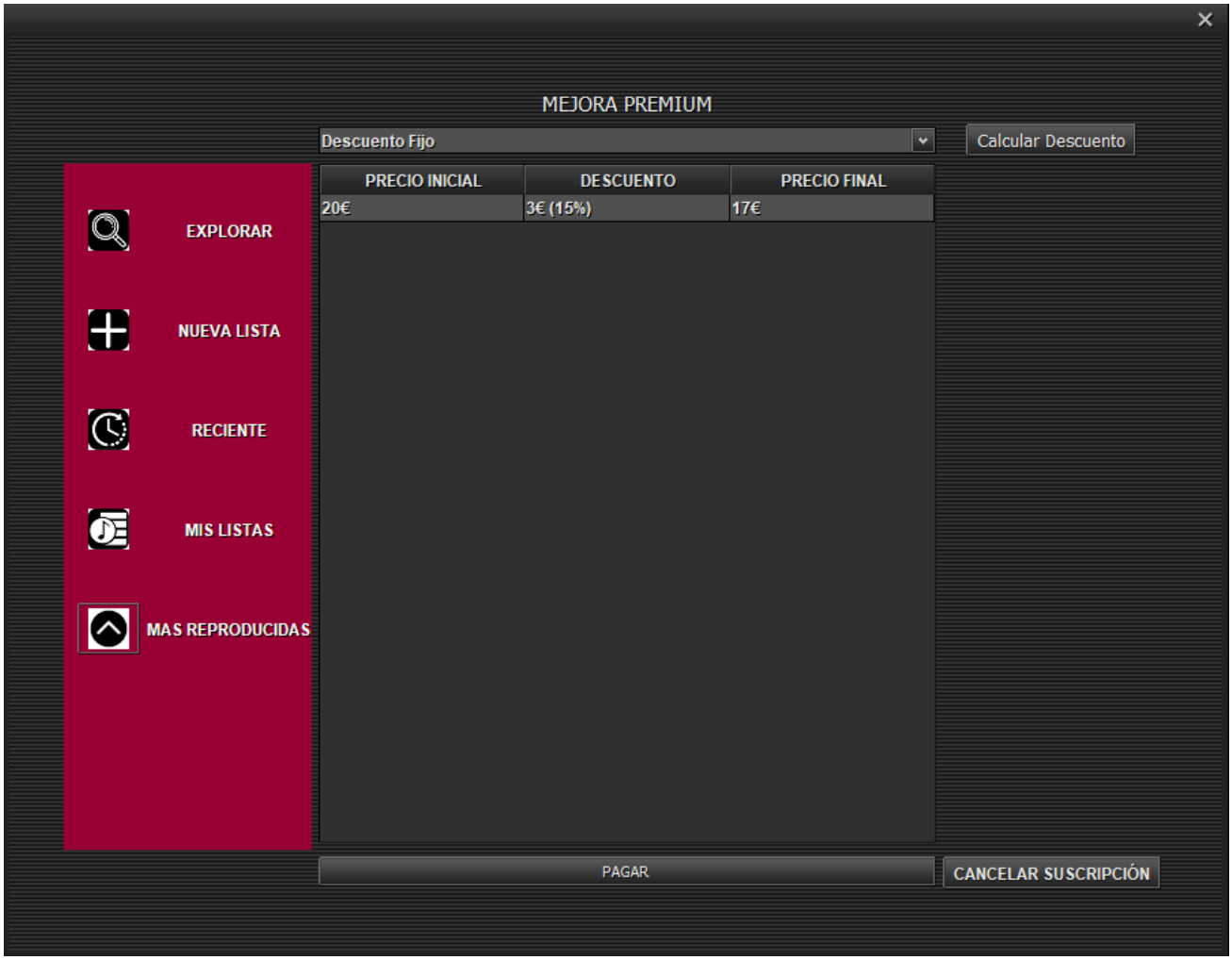

Figura 22. Ventana Mejorar Cuenta

8. Contenido extra

Como contenidos extra, hemos añadido que, dependiendo de en qué ventana estemos situados, el botón correspondiente a la ventana se pondrá de color blanco.

Figura 23. Mejora 1

También, al introducir un email para el registro, obligamos al usuario a que ese email acabe en '.org', '.es' o '.com' y que ese String tenga el carácter '@'. Si no cumple estos requisitos, no se podrá registrar.

```
else if (!email.endsWith(".com") && !email.endsWith(".es") && !email.endsWith(".org")) {
    JOptionPane.showMessageDialog(botonRegistrar, "El correo no es correcto, solo se acepta .org, .com o .es", "Registro usuario",
            JOptionPane.ERROR_MESSAGE, null);
}

else if (!email.contains("@")){
    JOptionPane.showMessageDialog(botonRegistrar, "El formato del correo no es correcto", "Registro usuario",
            JOptionPane.ERROR_MESSAGE, null);
}
```

Figura 24. Mejora 2.

9. Observaciones finales

Este proyecto nos ha resultado ser el más completo hasta la fecha de la carrera. Esto tiene, su parte buena y su parte mala. Por un lado, ha sido la primera vez en la que al menos, nosotros, hemos hecho una aplicación incluyendo la parte del back-end y el front-end y la verdad, es que le hemos puesto mucho empeño para que ambas queden lo más depuradas y perfectas posibles. Pero, por otro lado, también ha sido el proyecto que más tiempo nos ha llevado con muchísima diferencia ya que, estimamos, que le hemos dedicado aproximadamente 100 horas por persona. En conclusión, estamos muy satisfechos con el resultado ya que, en el fondo, no resulta una práctica muy compleja del todo, pero, a la vez, requiere mucha dedicación si quieres obtener un resultado bueno.