NappMusic

A toy multiuser desktop application for reproducing online and offline music

Proyect for Software Development Tools[1]

Authors: Emilio Domínguez Sánchez
Rubén Gaspar Marco
Due Date: 7th January 2021
Group: PCEO

[1]Subject original's name *Tecnologías de desarrollo del software*

Contents

1 Introduction

This project corresponds to the practical part of the subject «Tecnologías de Desarrollo del Software» («Software Development Tools»), part of the degree in Computer Engineering in the University of Murcia.

There are many web and desktop application aimed at reproducing music via streaming, such as Spotify or StreamSquid. The practical part of this subject requires the alumni to develop a music desktop application following a list of requisites.

1.1 Requisites

The specification, written in Spanish, is attached along with this document. To sum it up, it describes an application that can be used by different users. The application allows users to reproduce songs and playlists stored in their device or online. The song collection is common to every user although each of them can store their own playlists. They can also apply filters to search for specific songs or check their recently reproduced songs.

The application also features a premium system that grants users two extra functionalities. One is checking the ten most reproduced songs of the whole application. The other is obtaining a PDF with their playlists. Depending on many factors, the users can benefit from different discounts. The application automatically selects the best discount for each user and shows the lowest price.

The implementation of a payment system was considered too hard for this task. Therefore, the payment is simulated by clicking a button.

1.2 Technologies

This subject has the intention of teaching the students how to work in a professional environment. Hence, the project management is also evaluated.

The application had to be entirely written in Java, using Swing to build the graphical user interface (GUI) and JUnit to program unit tests. The project management had to be done through Maven, which handles the build, dependencies, tests and installation. In addition, the project had to be versioned controlled using Git.

We also decided to follow a consistent coding style. We chose Google's Java Style Guide because it was concise and easily enforceable. and we integrated it into the project in an editor independent manner, for which we used git-code-format-maven-plugin, a build plugin for Maven that also serves to install a Git hook which automatically formats the code before each commit.

2 Application Architecture

The music system needs to execute code in two different tiers: the client machine and the persistency server. The interface as well as the application logic run on the client machine, while the server handles the database. The communication between both is established through a TCP connection. The server was part of the specification that was already given to us and we deployed it in the same machine.

In order to avoid class coupling and achieve flexible and reusable code we decided to use a 3-layer architecture deployed on the two tiers we have. This pattern has also allowed us to reach a certain level of parallelism when working, since we have been able to develop simultaneously in different layers without knowing what changes were taking place in other layers.

The 3 layers used are composed as follows:

- **Presentation Layer**: Interface, typically graphical, with which the user interacts directly. It is responsible for handling incoming events and attending to user requests. The classes of this layer have been implemented with the Java Swing Library.
- **Business Logic Layer**: Domain classes that represent the application logic (those that appear in the fig. 1). They determine the representation of the data and how it can be used. It is somehow the highest-level part of the operational part of the application.
- **Data Access Layer**: This layer contains the classes related to the persistence service access. We have used a persistent database that has been provided by the professors. However, we have applied the Data Access Object (DAO) pattern to make the application independent of the persistence service used. This will be explained in detail in section 5.2.

Often, the user requests involve accessing and modifying domain classes. In this cases we follow the Model-View Separation Principle through the use of a controller class (Model-View-Controller architectural pattern or MVC) that acts as a *facade* between the presentation layer (view or interface) and the business layer (model). Thus, the presentation layer merely displays the options to the user, but does not implement any business logic, and the model classes have no reference to any user interface classes.

3 Domain Classes

The application handles three types of persistent objects. Songs, Playlists and Users. Each song is authored by an artist and is of a concrete music style. Both could be considered persistent entities, but we decided to store them encoded with the rest of the song's data.

3.1 Domain Classes Diagram

Out first task was writing a classes diagram that contained the business layer (see fig. 1), which we have mostly followed, although we changed the names of some classes.

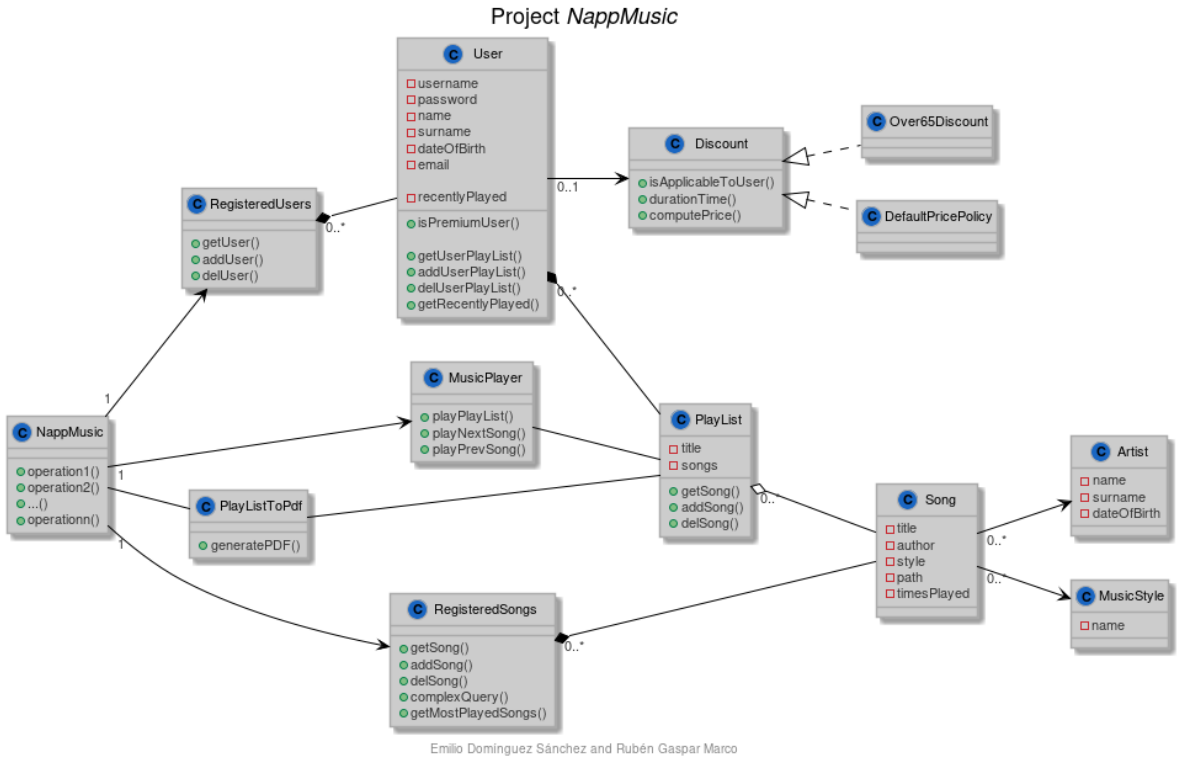


Figure 1: Domain Classes and Business Layer Diagram

- `Artist` and `MusicStyle`: As we pointed out, these classes ended not being part of the domain. The program that run the persistency server in the server, which we had to use, was not prepared to store them. However, we developed the application with the idea that these entities would be represented by their own class if this were a real application.
- `Song`: The `Song` class contains the title, author, style, URI and number of reproductions. It does not reference other classes.
- `Playlist`: The `Playlist` class is a ordered list of songs which also has a title. Every user has a list of playlists. No two playlists from the same user can share the title. Hence, the title can be used to identify the playlist and to display it in the GUI. Additionally, the temporary results that the controller returns to the GUI are also playlists, although they do not come from the database, nor will they be saved there.

  For example, if the GUI asks the controller for the list of songs result of a search, the controller returns a playlist whose title can be something like `Search Results`.

The `MusicPlayer` can also receive a playlist and an index to reproduce a song in a playlist, in which case it reproduces the next songs of the playlist when that one is finished. Thanks to that, the GUI can easily offer the user the posibility of reproducing the search results, saving them as a new playlist, etc.

- `User`: The `User` class contains the name, surname, username, password, date of birth, email, list of playlists, the playlist of recently played songs, a discount (which is selected to be the best one when computing the price), and the role (standard or premium).
- `Discount`: The Discount interface allows defining discounts that are only applicable to certain users. More about the discount interface and the patterns applied in section 5.1.
- `RegisteredSongs` and `RegisteredUsers`: These classes are now named `SongCatalog` and `UserCatalog`. They handle the domain classes `Song` and `User` in memory, while their persistent version is handled by the persistent layer. They support searching the objects that follow some criteria.

  For example, it is the task of the `SongCatalog` to index the songs in a way that allows fast look up. This functionality is exposed to the GUI via the controller.
- `NappMusic`: The class `NappMusic` represented the controller and is now named `Controller`.

4 Design Principles application

In the development of the project we have considered some principles of Domain-Driven Design (DDD). These include guiding the model to the code generation, the MVC pattern itself, the existence of service classes that implement a specific operation on the domain (SongLoader) or the existence of catalogs. Catalogs (or repositories) are globally accessible classes responsible of providing an encapsulation for all query operations on the certain type of objects that they store. They are not created for all of the persistence objects in the domain but for those that are query roots. In this project we have created catalogs for the users and for the songs, but not for the playlists since these objects are added to a certain user and, therefore, queries are made on him.

4.1 Usage of singletons

The singleton is a design pattern that ensures that a class has a single instance and that the instance in globally accessible. Singletons may be used to encapsulate the access to shared resources, like a database. However, a singleton is not different from a global variable[2].

[2]Actually there is a small benefit. Some implementations can benefit from inheritance, deciding which singleton to instantiate at the beginning of the program.

To prevent users from creating new instances, the constructor is marked as private. A single instance can be obtained via a static method called `getInstance()`, which creates it only when it is called for the first time.

We see many problems with singletons.

1. Even if in the form of a class, a singleton is global state inside the program. This means that it is difficult to reason about functions that use the singleton, because their result has to be considered depending on the state of it.
2. Singletons make the program difficult to unit test. For example, when the singleton represents a database, we usually need to mask the results for the tests. When the rest of the classes receive a reference to the database, you only need to pass a class that follows the required interface. However, changing the behaviour of a singleton requires more complex solutions. One possibility is inheriting from the singleton and calling a function that initializes the single instance using the derived class.
3. Singletons may depend on other singletons, which in turn can create a cycle of dependencies for initialization.

Therefore, we try to avoid using singletons. However, we have used them where they make sense.

- The DAO (Domain Access Object) is an abstraction built over the persistency system. Later in section 5.2 we make clear that we have made use of an abstract factory, implemented as a singleton.

  In this case it makes sense to have a single entry point to the persistency system. Furthermore, the persistency system does not depend on anything else and the factory already provides a way of instantiating different objects for the tests.

- The controller is the class that wraps the Business Layer for the Presentation Layer. Almost every graphical component we designed uses it. In this case it is a main class of the application and it packs some global state that is necessary for the efficiency of the application. For example, it wraps the catalogs, which provide fast access via indexing.

And avoid them when we feel we don't need them.

- The `UserCatalog` and `SongCatalog` are the catalogs of our application. Although there is only going to be a single instance of a catalog, we do not think that is enough to justify making them singletons. If global access was required, they could be accessed from the controller. And conceptually, it is much easier to think them as collections that allow fast access with some search criteria. Why would a collection be a singleton even if we only plan on instantiating one?

4.2 Design Patterns

During the development of the application we have used several design patterns. Among them we can find the following.

- Creational patterns.
  - Factory method.
  - Abstract factory.
- Structural patterns.
  - Adapter.
  - Bridge.
  - Decorator.
  - Facade.
- Behavioral patterns
  - Iterator.
  - Strategy.
  - Observer.
  - Template Method.
  - Null Object.

We will describe most of them where we used them, but first we want to explain some patterns that are present multiple times.

4.3 Bridge and Decorator

The bridge and decorator patterns are used by Swing because they are very practical for GUI frameworks. For example, you can pass a `JText` component to a decorator which adds a border around it. Similarly, as users we can implement the `JListModel` interface which is used by the `JList` component to obtain data to display, following the bridge pattern. In a way, the bridge pattern is used to rely on code implemented by the user of the framework.

4.4 Facade

The facade pattern is the creation of a class or a set of methods that act as a simplified interface to a complex system. This methods are shortcuts for common operations that use many of the original methods.

4.5 Strategy Pattern

This pattern consists of defining a family of algorithms and allowing the class (context class) who use them to exchange those algorithms by adding an attribute to that class that stores a reference to a particular algorithm (strategy object). We used this pattern when calculating the discount applied on a user when upgrading to premium.

4.6 Observer Pattern

This pattern, also called the Listener pattern, suggests that you add a subscription mechanism for the objects that give rise to events, instead of calling a function from another class when an event occurs. We use this pattern in the GUI for example, to notify the visual components when certain domain objects have changed, so that the components can update themselves.

5 Application Logic

5.1 Discount Selection

Satisfying the Open-Closed principle, the discounts allow introducing new ways to calculate the price of upgrade to premium without modifying the existing code in the User class, to which a `calculatePremiumPrice` method has been added so that whoever wants to know this data does not need to know the discount object. Also the Null Object pattern has been used in this class, so that a User can have no discount without setting the attribute to null.

The idea of the discount selection is to allow the introduction of new discounts modifying as less code as possible. We thought reflection would be a nice technique to accomplish this. The reflection in a coding language is the possibility of analyzing the code programmatically. In the case of Java, it allows, among other things, to manipulate classes as objects of the generic type `Class<T>`, being able to call methods on them to retrieve some of the information of the class. In this case, we used it to get their constructors (another object that represents code) and then get instances from it only knowing the name of the package these classes are in.

Thanks to the reflection offered by java and the simplification that offers the Reflections library, we can analyze the discount package and go through the different discount interface implementations defined in it, instantiating them and seeing which one gives us the lowest price.

5.2 Data Access Object

The persistency system is based on a closed source library we were given. The API of the library is based on the concepts entity and property. An entity is an object of the database. It consists of a entity name and an unique id generated by the system. The

entity name is useful for retrieving all of the objects of the same class name. Similarly, a property is a value object of the database associated with an entity. Properties consist of a name and a string value. Properties are uniquely identified by the entity they relate to and their name.

Following this scheme, we store classes as entities. The entity name depends upon the class, which lets us retrieve all the objects of a given class or determine the class of an entity. And for each field of the class we want to store we encode it in a property. If a class references another persistent class then we use its entity id.

To make the application independent of the persistence service used, we have applied the Data Access Object (DAO) pattern. This pattern is based on the use of the *Adapter* and the *Abstract Factory* patterns. We will explain it with the classes of our application.

The abstract factory class `DaoFactory` is used to generate a set of related DAO adapter objects. According to the DAO pattern, for each type of persistent object there is a DAO adapter responsible of storing that object in the persistency system. To create a concrete adapter, `DaoFactory` is extended to a concrete factory, in our case `TdsDaoFactory`, that constructs a family of DAO adapters that need to follow an interface for the application to use them (`Dao<T>` interface in our case). The application can then change the persistency system and, just by creating a new concrete `DaoFactory` and a new set of concrete DAO adapters, and run just as before.

5.2.1 Classes organization

Our implementation of the persistency system has been very general. We have structured the code in a way that tries to make very simple to create many DAO adapters. To do that, we need to share the logic of accessing the database and write classes that merely decide which members of the objects need to be stored for a particular class. We divide the role into three main classes.

- `TdsDao<T>` and `TdsPoolDao<T>`: This classes are generic DAO adapters. They encapsulate the logic that is needed to register new objects in the database. To know how to encode the object they use a `BiEncoder<T>`, another important class we will describe next. And to interface with the database they use a wrapper called `PersistencyWrapper`. The difference between both classes is that the second one uses a pool of objects internally, that is, it stores which objects it has retrieved previously. This serves the purpose of avoiding infinite loops when retrieving entities of the database which reference each other forming a loop.

  For example, take a look at this code from `TdsPoolDao<T>`.

```
1 @Override
2 public void register(T obj) {
3   // This step is very important to avoid cycles when registering
4   // an entity that triggers more registers
5   if (pool.get(obj.getId()) != null) return;
6
7   // We include our object into the pool to avoid cycles
```

```
8     // We first need to register the entity without fields
9     Entidad emptyEntity = new Entidad();
10    emptyEntity.setNombre(encoder.getEntityName());
11    // Registering the entity in the server gives it a unique id
12    wrapper.registerEntity(emptyEntity);
13    obj.setId(emptyEntity.getId());
14    pool.put(obj.getId(), obj);
15
16    encoder.encodeIntoEntity(obj, emptyEntity);
17    // Now the entity is not empty
18    wrapper.updateEntity(emptyEntity);
19  }
```

The steps needed to register an object and avoid cycles are very specific. We do not want to rewrite them in every DAO adapter that uses a pool, because it is very easy to introduce a new bug. Algorithmically, the only thing which changes is how to create an `entity` from the object `T`. That is a task the `BiEncoder<T>` performs for us. This way of reusing code is sometimes called the Template Method pattern.

- `BiEncoder<T>`: The interface `BiEncoder<T>` defines which methods an encoder needs to support. These include initializing an object from an entity, and encoding an object into an entity.

  The important design decision here is to wrap a lot of encoding methods into the `PersistencyWrapper`. The reason is that different encoders need to encode the same types of fields. For instance, booleans. Or a more complex example, a list of `Song`. Instead of making each `BiEncoder<T>` aware of each other existing `Dao<T>` to be able to register the member objects, we make each `BiEncoder<T>` just aware of the wrapper. This is like reducing the coupling from n^2 possible relations to just n, plus all the tangible benefits of code reuse.

Listing 1: Example method of the class `UserEncoder` which implements `BiEncoder<User>`

```
1   @Override
2   public Entidad encodeIntoEntity(User user, Entidad entity) {
3     entity.setPropiedades(
4     new ArrayList<Propiedad>(
5     Arrays.asList(
6               new Propiedad(NAME_FIELD, user.getName()),
7               new Propiedad(SURNAME_FIELD, user.getSurname()),
8               new Propiedad(BIRTHDATE_FIELD, wrapper.encodeLocalDate(user.
      getBirthDate())),
9               new Propiedad(EMAIL_FIELD, user.getEmail()),
10              new Propiedad(USERNAME_FIELD, user.getUsername()),
11              new Propiedad(PASSWORD_FIELD, user.getPassword()),
12              new Propiedad(PREMIUM_FIELD, wrapper.encodeBoolean(user.isPremium())),
13              new Propiedad(PLAYLISTS_FIELD, wrapper.encodePlaylistList(user.
      getPlaylists())),
14              new Propiedad(RECENT_FIELD, wrapper.encodePlaylist(user.getRecent()))
      ));
15    return entity;
16  }
```

- `PersistencyWrapper` As already commented, this is the class that contains helper

methods to use the persistency system. Hence, it contains a great deal of basic operations, encoding and decoding methods, but all of them are relatively simple.

Listing 2: Example method of the class `PersistencyWrapper`

```
1 public String encodeSong(Song song) {
2   songDao.register(song);
3   return Integer.toString(song.getId());
4 }
```

Typically, these helper methods are static but because they may need to use the different DAO adapters and we did not want to make each class a singleton we decided to keep them as class methods and to pass the wrapper as a reference.

The class `DaoFactory` from the same package is the concrete factory that can return the reference to each `Dao<T>`. An schema of how everything relates together can be seen in fig. 2. A sequence diagram of the process of creating a playlist, as asked from the GUI to the controller and how it involves the DAO, is represented in fig. 3.

6 External Components

Big software companies can reuse code among projects. Thus, they use a design based in components, which tries to break an application into small reusable parts. In the end, this may also help the development and maintenance processes, since it is usually easier to work with many small completely independent projects.

In our application, we have used some open source external components like `JCalendar` or `iTextPdf` and we have developed another two.

6.1 SongLoader

The `SongLoader` is a component that allows to load a list of songs from a XML file. The interface `SongLoader` uses the Observer Pattern to notify the listeners with a `Songs` object[3] when the operation invoked via the method `loadSongs(String xmlPath)` is completed.

Listing 3: Example XML file

```
1 <?xml version="1.0" ?>
2 <canciones xmlns="http://www.tds.es/canciones" xmlns:xs="http://www.w3.org/2001/
      XMLSchema-instance"
3   xs:schemaLocation="http://www.tds.es/canciones canciones.xsd">
4
5   <cancion titulo="I'll take a melody">
6     <URL><![CDATA[https://ia801406.us.archive.org/8/items/tsheaffer2020-11-08/16%201%27
      11%20Take%20A%20Melody.mp3]]></URL>
7     <estilo>Folk</estilo>
8     <interprete>Todd Sheaffer</interprete>
```

[3]Note that the component defines its own classes. There is a `Song` class in the component and in the main application.

```
9    </cancion>
10
11   <cancion titulo="La Vie en rose">
12     <URL><![CDATA[https://ia801605.us.archive.org/16/items/78_la-vie-en-rose-slow-
       chante_edith-piaf-louiguy-edith-piaf-chansons-parisiennes-guy_gbia0000684a/La%20Vie
       %20En%20Rose%20%28Slow%20Chante%29%20-%20Edith%20Piaf-restored.mp3]]></URL>
13     <estilo>Cabaret</estilo>
14     <interprete>Edith Piaf</interprete>
15   </cancion>
16
17   <cancion titulo="In the Court of the Crimson King">
18     <URL><![CDATA[https://ia800102.us.archive.org/7/items/cd_in-the-court-of-the-
       crimson-king_king-crimson/disc1/05.%20King%20Crimson%20-%20The%20Court%20of%20the
       %20Crimson%20King%20%28including%20The%20Return%20of%20the%20Fire%20Witch%20and%20
       The%20Dance%20of%20the%20Puppets%29_sample.mp3]]></URL>
19     <estilo>Rock-sinfonico</estilo>
20     <interprete>King Crimson </interprete>
21   </cancion>
22
23   <cancion titulo="Confessing that I love you">
24     <URL><![CDATA[https://ia600108.us.archive.org/13/items/78_west-end-blues_louis-
       armstrong-and-his-orchestra-spencer-williams_gbia0031327/06%20-%20Confessin%27%20
       That%20I%20Love%20You%20-%20Louis%20Armstrong%20And%20His%20Orchcestra.mp3]]></URL>
25     <estilo>Jazz</estilo>
26     <interprete>Louis Armstrong</interprete>
27   </cancion>
28
29   <cancion titulo="Moonlight Serenade">
30     <URL><![CDATA[https://ia800803.us.archive.org/24/items/78_moonlight-serenade_glenn-
       miller-and-his-orchestra-glenn-miller_gbia0015151b/Moonlight%20Serenade%20-%20Glenn
       %20Miller%20and%20his%20Orchestra-restored.mp3]]></URL>
31     <estilo>Ballad</estilo>
32     <interprete>Glen Miller</interprete>
33   </cancion>
34
35   <cancion titulo="La Cumparsita">
36     <URL><![CDATA[https://ia801609.us.archive.org/14/items/78_la-cumparsita_pietro-g.h
       .-matos-rodriguez_gbia0001088b/La%20Cumparsita%20-%20Pietro%20-%20G.H.%20Matos%20
       Rodriguez-restored.mp3]]></URL>
37     <estilo>Tango</estilo>
38     <interprete>Pietro</interprete>
39   </cancion>
40
41 </canciones>
```

6.2 UiButton

The UiButton is a graphical component. A simple colored button the user clicks in our application to load new songs.

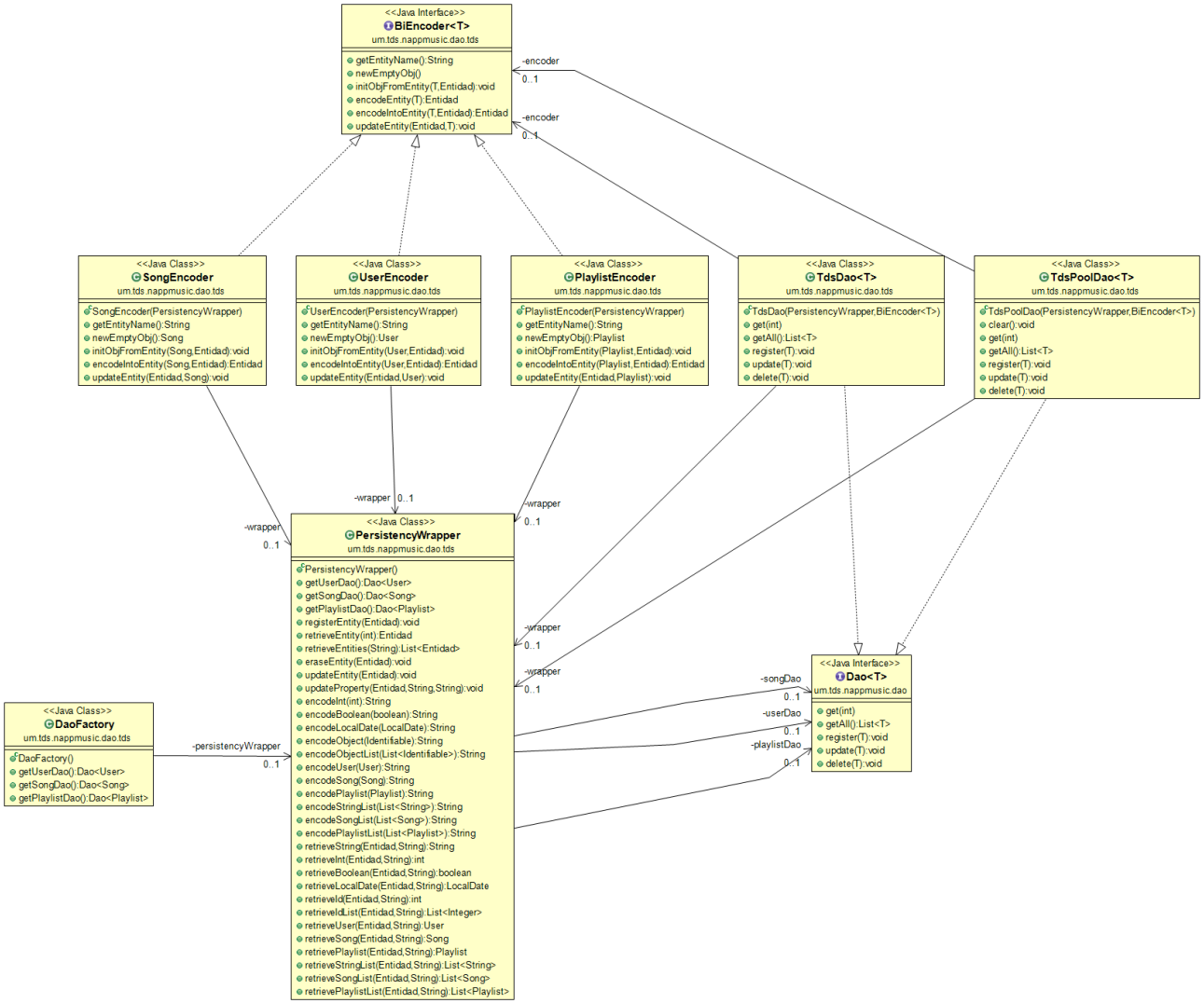


Figure 2: DAO classes

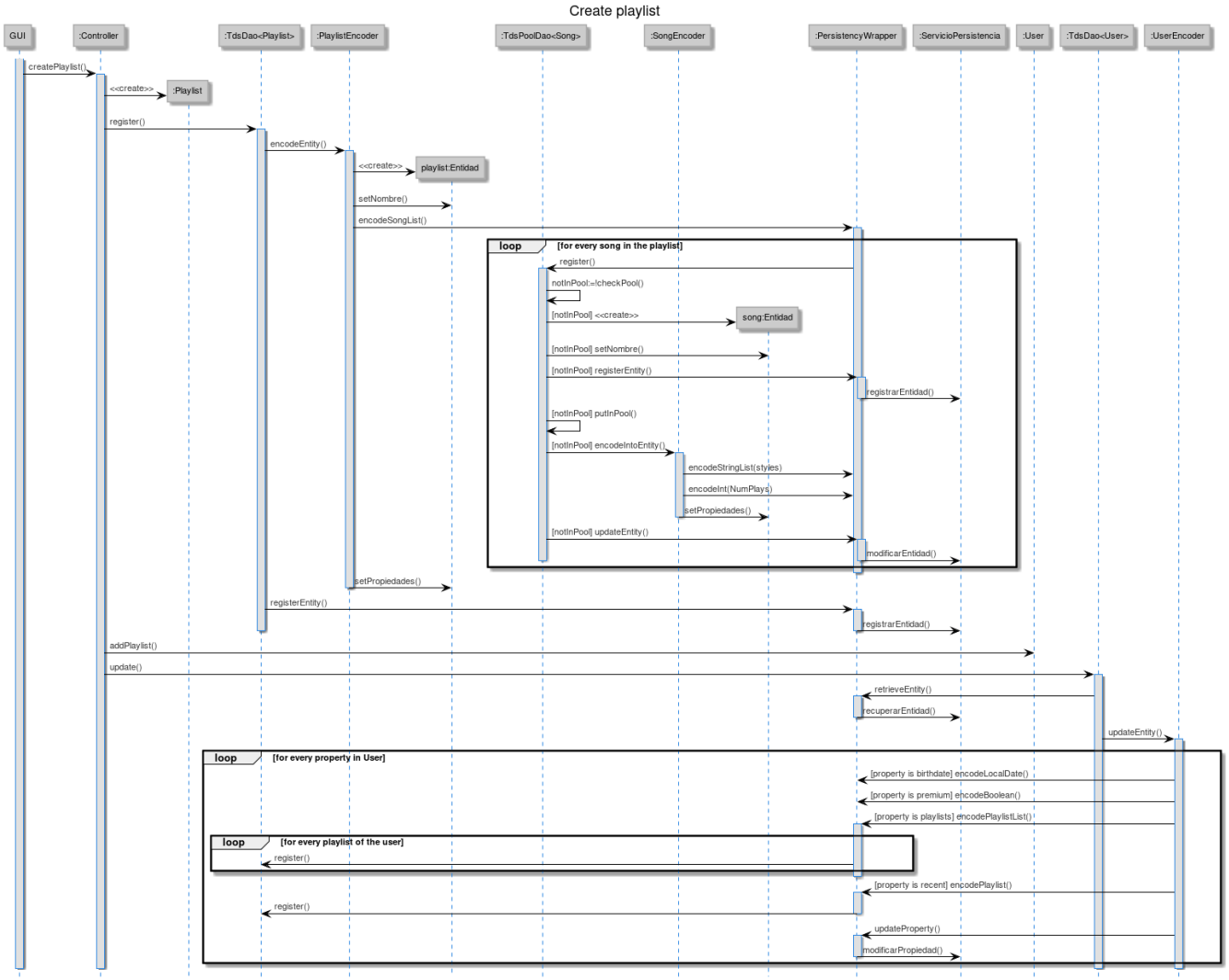


Figure 3: Creating a new playlist. Sequence Diagram

7 Tests

Our project also integrates developer-side testing through JUnit 5. We have a strong preference for automated unitary tests, but in some cases we could only implement integration tests.

We have prepared unitary tests for the persistency system, the DAO, the catalogs and the discount selection system. However, an integration test has been created for the PDF generator. 25 tests in total. These are the main systems below the presentation layer. In contrast, the graphical user interface has been tested running the application several times. We would have liked to have a system that would help us automate the process of testing the GUI.

The file `RetrieveStoreTests.java` contains tests for the class `PersistencyWrapper`. It checks that it is possible to register and retrieve entities from the persistency system.

The file `DaoTests.java` contains tests for the `User`, `Playlist` and `Song` DAOs. It tests each one registering objects via the dao (which encodes them), and them retrieving them and checking that they are the same. It also performs some test to check that it is also possible to modify objects.

The file `CatalogTests.java` contains fake data and tests for the `SongCatalog`. It is centered on testing the queries implemented for the catalog: searching by artist, searching by title, getting the most reproduced songs, etc.

Some other tests can be seen under the tests folders of each project.

8 Graphical User Interface

8.1 Handling events

As we anticipated, the entire graphic interface has been implemented using Java's graphic library Swing, which follows a single-threaded programming model based on the `EventQueue`, which is a platform-independent class that queues events and dispatches them sequentially in what is called the event-dispatch thread. Therefore, no two GUI components are accessed simultaneously.

To handle the events, Swing components usually define a method called `addActionListener` , an example of the Observer pattern. In general, we either use anonymous classes that implement the `actionListener` interface or small lambda expressions, since `actionListener` is a functional interface.

8.2 Lists and tables

Graphical lists (resp. tables) apply the Adapter pattern to make the object, a `JList` (resp. `JTable`), independent of the displayed information. `JList` (resp. `JTable`) uses the

`ListModel` (resp. `TableModel`) interface. For both lists and tables there is an abstract class that partially implements the interface and from which it has been inherited to implement our own `PlaylistListModel`, which obtains data from a `List<Playlist>`, and `PlaylistTableModel`, which obtains data from a `Playlist`.

8.2.1 PlaylistJList

This class is a facade in charge of making easy the use of a JList whose model is PlaylistListModel. It can get a playlist object from the selected item in the table.

8.2.2 PlaylistJTable

It is the analogous class to the previous one. In addition, this class has an extensible JPopupMenu (you can pass a JPopupMenu in its construction) that allows to add the selected song of the table to any playlist, including to a new playlist, for which it appears an input dialog in which it is specified the name.

8.2.3 Updating

Because the `PlaylistJTable` graphical component allows the modification of domain classes, it is also responsible of notifying the other graphical components that a `Playlist` has been modified. This is one of the complex parts of the GUI, as both the tables and the lists have to be updated when the data they contain changes. This kind of updates are not a problem when the modifications take place when the list or the table is not visible because the panels are revalidated[4] every time they are shown. However, as changes can be made on the data while a panel is visible, a notification system was built following the Observer pattern to notify the tables/lists owners from the part of the code that involves the change.

This is a singleton (`GuiNotifier`) containing a list of the listeners. Components interested can subscribe implementing the interface `PlaylistListListener` if they want to be notified of a change in the list of playlists or the interface `PlaylistListener` if they want to be notified only when a particular playlist changes.

8.3 MusicPlayer

This class contains a simple panel with `BorderLayout` where to the west, in a subpanel with `GridBagLayout`, are the data (`JLabels`) of the song being played and in the center we have the buttons that control playback (Previous, Play/Pause and Next).

The audio files can be located through a system path or through a URL that uses the HTTP protocol. In the latter case, the GUI is responsible for downloading the song to a temporary file.

[4] A Swing operation which refreshes some properties of a component.

Song playback is performed by the `MediaPlayer` class of the JavaFX library. This library has allowed us to do a few cool things, like establishing markers in the songs and configuring event handlers for the player. We have used the markers to consider a `Song` reproduced only after the user has listened the first ten seconds.

As all song collections are managed through a playlist object, (that is, the GUI does not know of songs which don't belong to playlists, because the search results are playlists) we also set an event to reproduce the next song in the playlist when the song being reproduced ends.

9 Conclusions

The development of this application has taken each of us more than 80 hours of work (approximately distributed as listed in table 1). During this time we have been able to value the role that design patterns play as a way of reusing experience, that is, to solve problems with previously implemented soutions on similar problems. It has also been our most intense contact with a graphical interface, forcing us to understand the difficulties that it entails. Furthermore, this development has also led us to continue improving our teamwork, managing a common source code repository with git.

Project: *NappMusic*			Start date:		*7/10/2020*
Team:	Emilio Dominguez Sánchez Rubén Gaspar Marco		End date:		*3/1/2021*
	Design	Implement.	Validation	Refactoring	**TOTAL**
TOTAL (hours)	25	20	15	5	80
PERCENTAGES	31.25%	25%	18.75%	25%	100%

Table 1: Time spent on the project

User Manual

Upon launching the application, the first thing you will see is the Login Window (fig. 4a). If you do not have a user yet, you will need to create one by clicking on the `Register` button. It will bring up the Register Window (fig. 4c), where you can create an account if you are at least ten years old.

When you log in you encounter the Main Window with the Home Panel selected (fig. 5a). If this is the first time you launch the application in this computer, you will want to load some songs first, which you can do by pressing the button located above your username and selecting an XML file.

On the left hand side of the application there is a menu to navigate the different panels. The search panel (fig. 5c) allows you to browse the available songs. The title and the artist entries will filter the songs whose title an artist contain the substring you input. You can also restrict the search to songs of a given genre. You can select any song and reproduce it by clicking in the play button that will appear in the music player at the bottom.

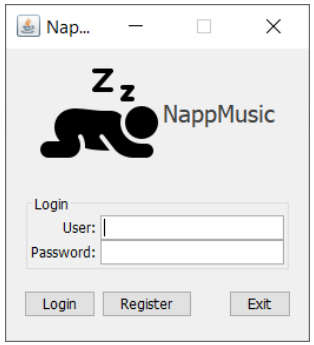
In addition, if you right click on a song, you will be able to add it to a playlist (fig. 6b). If you want to create a new playlist that contains that song, you can also do that by clicking on new playlist. The playlists panel (fig. 6c) lets you browse your playlists, which you will need to do if you want to remove songs from any playlist. The recently played panel (fig. 7a) allows you to see your most recently reproduced songs. Note that a song will only count as reproduced if you play it for more than ten seconds.

9.1 Premium functionality

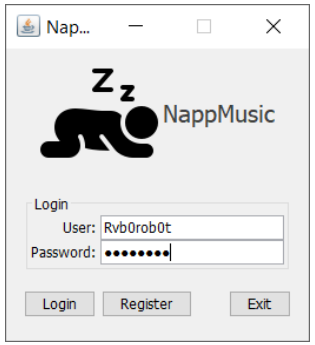
If you wish to become premium, right click on your name and click on `Upgrade` (fig. 8a). If you are a premium user, you will be able to access the most played panel. As the name suggests, it lets you see the ten most reproduced songs of the application. If you try to use without being premium a dialog will stop you (fig. 7c).

Once you become premium, the `Upgrade` button will be replaced by `Save playlists in pdf` (fig. 8), this will let you save your playlists as a PDF file in your system.

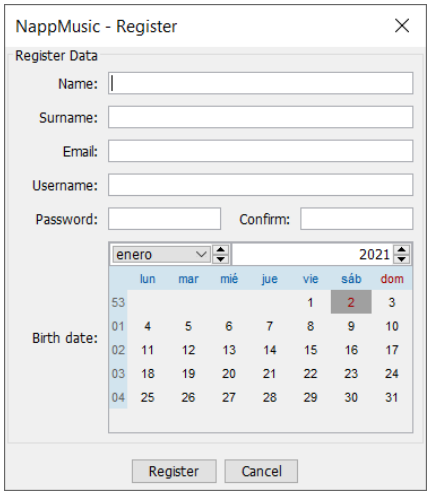
Gallery



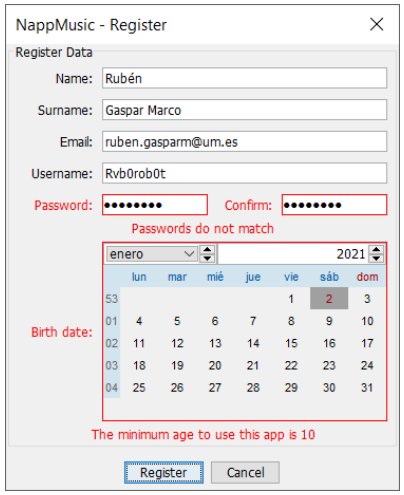
(a) Login Window



(b) Entering credentials

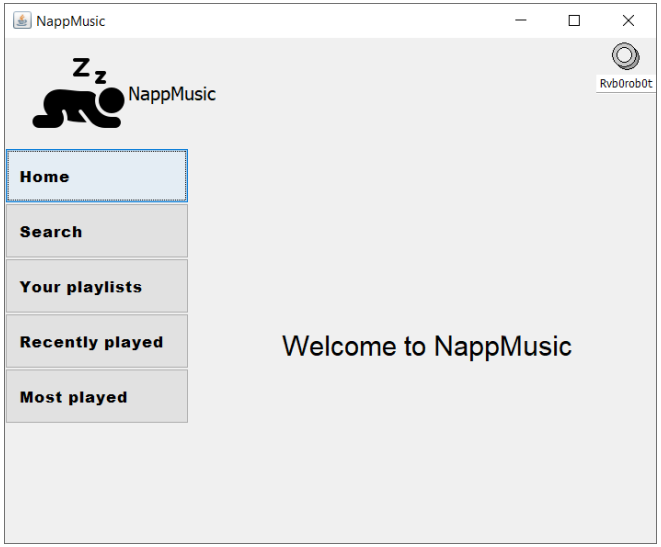


(c) Register Window

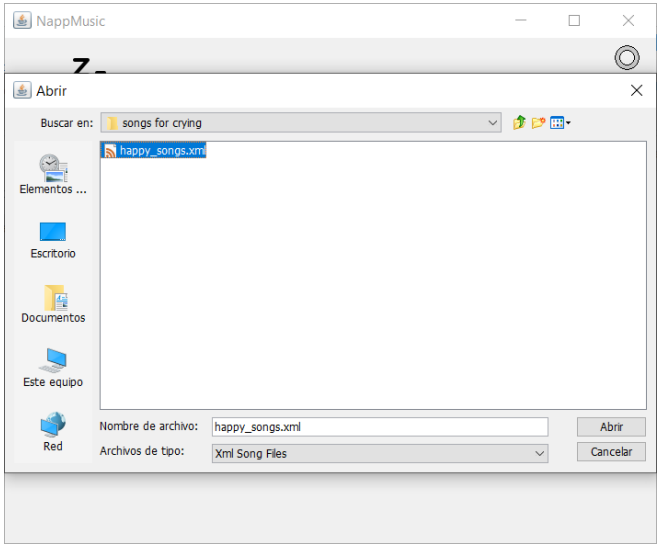


(d) Error messages in the Register Window

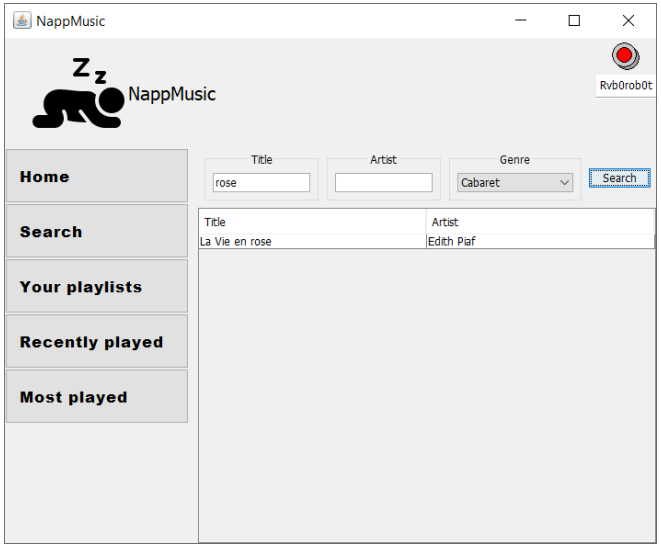
Figure 4: Login and Register Windows



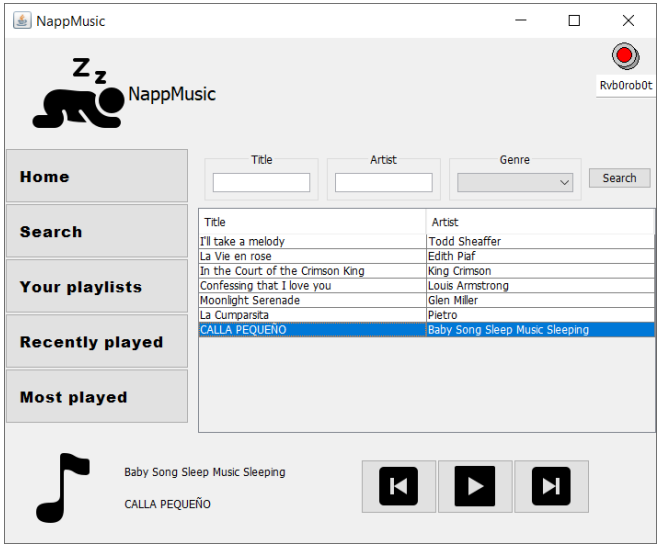
(a) Home Panel



(b) File chooser opened for selecting an XML file with song descriptions

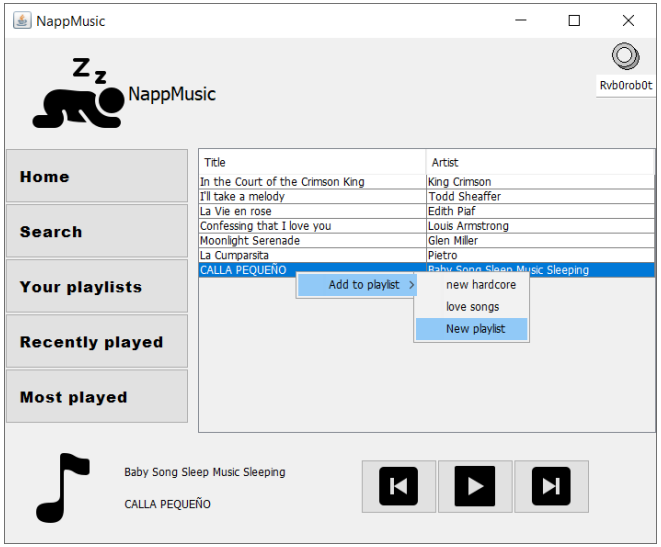


(c) Search Panel

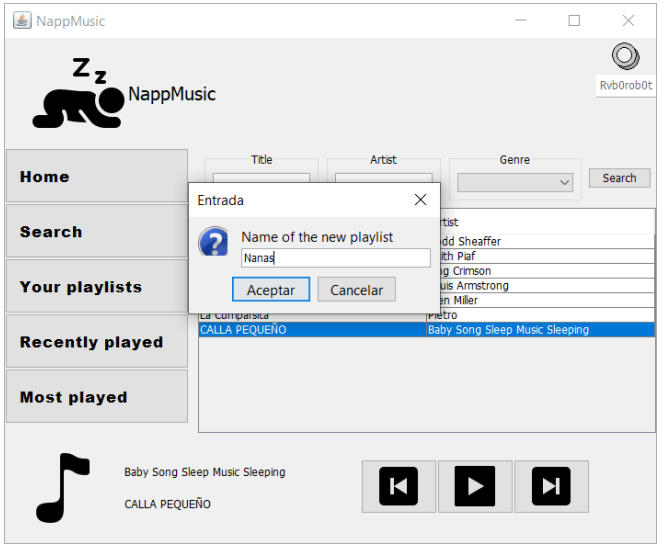


(d) Media Player

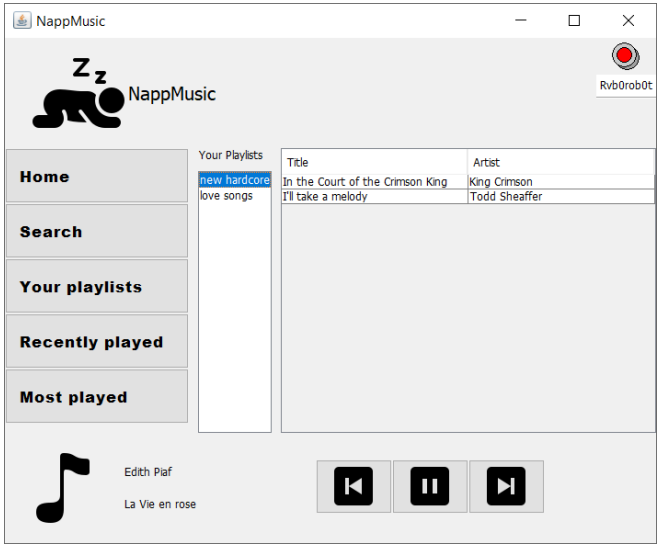
Figure 5: First Steps: Loading and reproducing songs



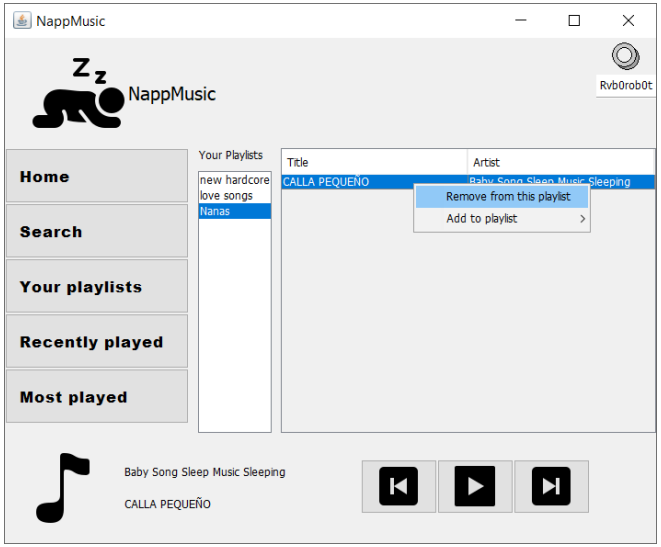
(a) Adding a song to a playlist



(b) Naming a new playlist

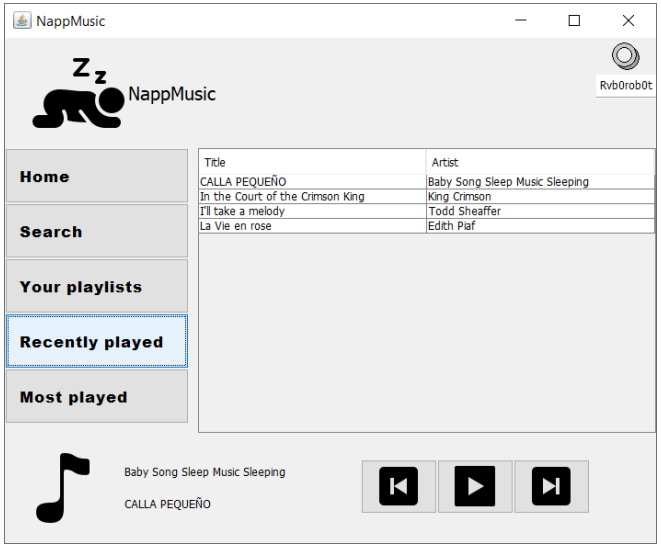


(c) Playlists Panel

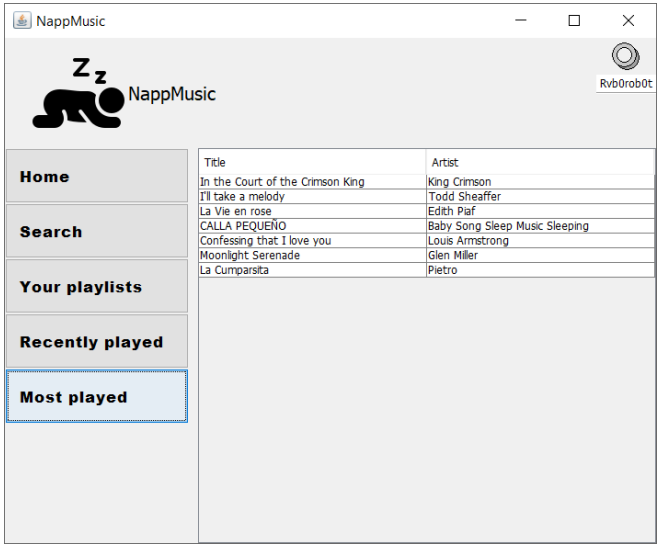


(d) Removing a song from a playlist

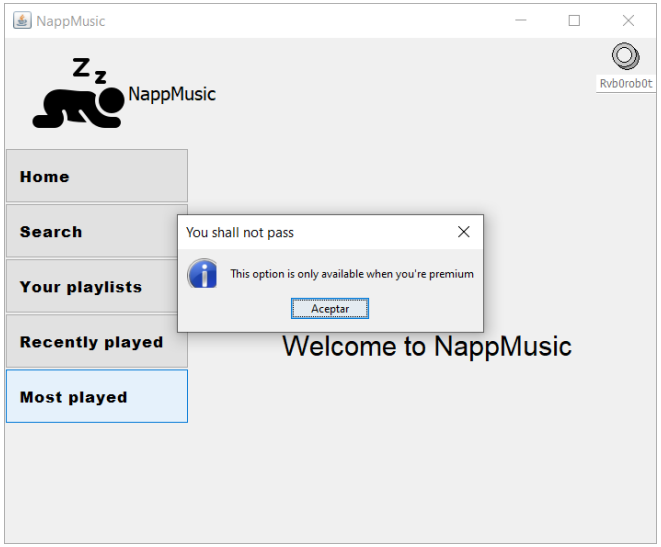
Figure 6: Managing the user playlists



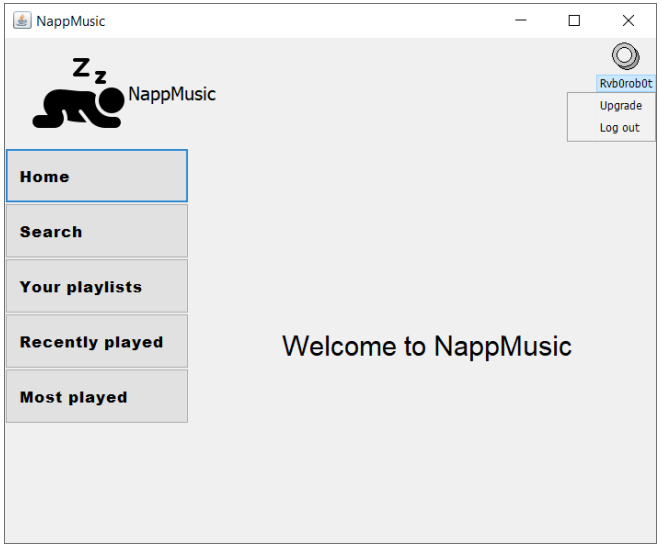
(a) Recently played panel



(b) Most viewed panel

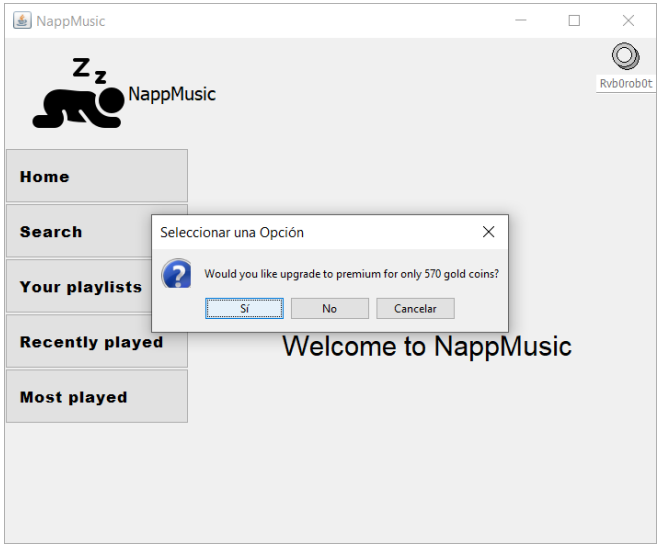


(c) Most viewed songs panel is only available to premium users

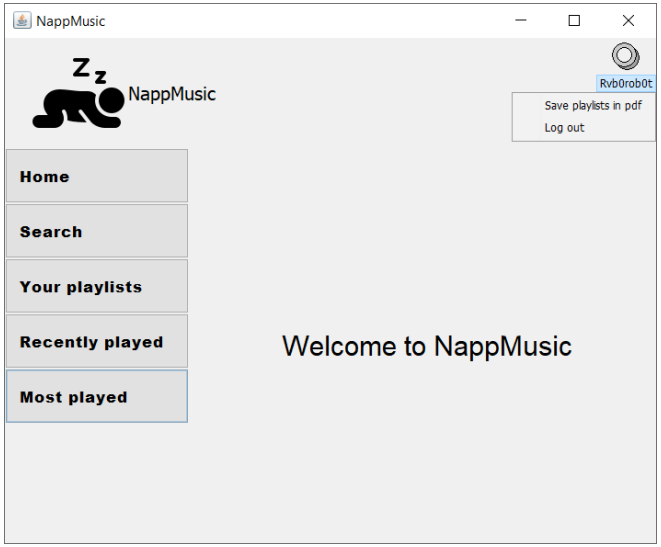


(d) Upgrade Button

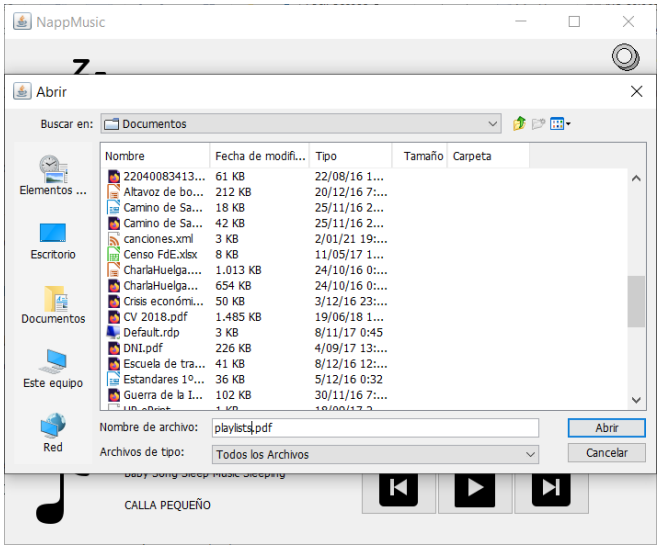
Figure 7: Non premium user restrictions



(a) Becoming premium dialog



(b) Premium user menu



(c) File chooser to save a playlist pdf

Hello Rvb0rob0t, these are your playlists

new hardcore

Title	Artist	Style
In the Court of the Crimson King	King Crimson	Rock-sinfonico
I'll take a melody	Todd Sheaffer	Folk

love songs

Title	Artist	Style
La Vie en rose	Edith Piaf	Cabaret

Nanas

Title	Artist	Style
CALLA PEQUEÑO	Baby Song Sleep Music Sleeping	Nana

(d) Example playlist pdf

Figure 8: Premium functionality