Software Engineering Project Report

Federica Ardizzone

March 2025

1 Documentation

1.1 Description

This software simulates the operation of a music player - it implements the **Design Pattern State** in visualizing how the playlist is shown to the user, and the **Design Pattern Façade** by offering the user an interface to interact with the playlist itself, which allows the user to sort the playlist and make song queries.

1.2 Ancillary classes; `SongLibrary` and `Song`

The `Song` class and the `SongLibrary` class act as aid for the `MusicPlayer`.

The `Song` class defines the objects which represent the songs; the class `Song` has four attributes: `String title`, `String author`, `int secondsDuration` and `int publishingYear`. It contains getter methods, and a `String toString()` override in order to print the details of the song.

The `SongLibrary` class has two private attributes: `List<Song> myLibrary` contains the songs inside the playlist, and `int whereInList` represents the "head" of `SongLibrary` playlist, the current song which is "selected" by the playlist. It is possible to add and remove songs from the `List<Song> myLibrary` list, through the methods `public void addSong(Song s)`, and `public void removeCurrentSong()`, which removes the current song, if the list is not empty.

The `SongLibrary`'s method `public Optional<Song> next()` checks and then if it is possible to get a next, it returns it (or the absence of it) through an `Optional<Song>` object through which an `Song` object can be accessed through the `.get()` method implemented in the `Optional` library; the `Optional` library was introduced in Java 8.

The same is done by the `public Optional<Song> previous()` method, but for the previous song. Both methods work in a "circular" way, looping back to

the end if one goes before the first, and going up to the beginning if one tries to go after the last `Song` in the list.

The method `public Optional<Song> getCurrentSong()` works in a similar way as the `next()` and the `prev()`. It checks if the `List MyLibrary` is empty. If it is not, then it returns an `Optional` object which contains the current song, which is at the index given by the `int whereInList`. If the list is empty, then this method returns an empty `Optional`.

The `SongLibrary` class contains two methods which can be used in order to modify the `List MyLibrary` and the `int whereInList`: respectively, `public void modifyPlaylist(List<Song> modifiedList)` and `public void modifyWhereInList( int modifiedHead)`. These are used by the `AdvancedControl` class.

1.3 `AdvancedControl` classes

The `AdvancedControl` class implements the **Façade Design Pattern** and provides a way to offer more control and features to the music player, by either calling personalized queries or sorting the library by different criteria. It is a Façade to the `ClientListener` class, simplifying the interface to the `AdvancedControl` subsystem, which includes the class `Sorter` and the class `Finder`.

In this implementation, the `ClientListener` has an instance of the `AdvancedControl` class, and is able to call its methods, which will in turn call the methods of the subsystem of which it is Façade of, such as specific sorting methods and specific finding methods, which are here implemented through the `Stream API`, introduced in Java 8.

The `AdvancedControl` class has seven private attributes: The `SongLibrary songlib`, which allows access to the list of songs, the `Sorter sorter` and the `Finder finder`, which are specialized classes in which the logic of the controls is placed, and then `List<Song> currentUserQuery`, `Optional<Song> singleQuery`, and the two attributes `List<Song> libraryStoring` and `int headStoring`, which are useful in order to momentarily store queries, and to restore the original library when the Client desires.

The `AdvancedControl` class defines three calls to its instance of `Sorter`, and three calls to its instance of `Finder`. In the case of the `Sorter`, the `AdvancedControl` class merely calls the respective `Sorter` method, whereas the finding methods have a different logic, which includes storing a temporary List of `Song` objects based on the selected query.

The `Sorter` class has a private attribute `SongLibrary songlib` - which is passed to it at the creation - and three sorting methods, `sortByTitleName()`, `sortByAuthorName()` and `sortByPublishingYear()`, which are implemented

through the Stream API. Each sorts the instance of `SongLibrary` by the specified `Song` attribute.

The `Finder` class has too a private attribute `SongLibrary songlib` which is passed to it at the creation, plus the private attribute, `List<Song> queryList`, which is used by the class to temporarily store user queries before returning them. The previous queries' "buffer" is cleaned with the `clearQuery()` before being reassigned by a new query.

The first of the finder methods is `Optional<Song> longestSongFinder()`, which returns an `Optional<Song>` that contains the longest song of the playlist, if the playlist is not empty; if it is empty, it will return an empty Optional; the `AdvancedControl` class won't modify the playlist head if there is nothing to be played.

The second finder method is `List<Song> songsByWhom(String author)`, which allows the user to make a query with a string containing the author name. If there are any songs which match the query, this method will insert them in the `queryList` and return the list to the `AdvancedControl`. The `AdvancedControl` class will temporarily store the current library into the `List<Song> libraryStoring` and the `int headStoring` attribute, and modify the playlist with the current query made by the user. It is left up to the user to call the method `backToLibrary()` in the `AdvancedControl` class in order to go back to the original library as it was before the query was made.

The third finder method, `findAllAuthors()`, prints to the screen all the distinct names of the authors in the playlist, by calling the `List<String> songAuthors()` method in the `Finder` class.

1.4 The Display classes: `MyDisplay`, `FullPlaylistMode`, `SongFocusMode`; and `MusicPlayer`

These classes implement the way that the playlist is shown to the user - in this case, printed through `System.out.println`. They implement the Design Pattern State.

The `interface MyDisplay` is the **State** interface, which is implemented by two **ConcreteStates**, `FullPlaylistMode` and `SongFocusMode`. The `MusicPlayer` class plays the role of the **Context**, which the Client is able to interact with. The `MusicPlayer` class also has a variable, `boolean isFocus`, which signals which ConcreteState is currently implementing the `MyDisplay` interface.

The "default" mode, which is assigned to the private `MyDisplay displayState` attribute in the `MusicPlayer` instance is `FullPlaylistMode`; and the variable

`boolean isFocus` starts as negative.

The `MusicPlayer` defines a method `switchMode()` which allows the Client to swithc the status. This specific implementation implies a two-states architecture, but more states could be added by using an `int` variable instead of a `boolean` one as the variable which keeps track of which ConcreteState is active.

The `MusicPlayer` class also contains the method `void playMusic()` which is called by the `ClientListener` in order to display the songs themselves. This method calls the `visualize()` method of the current instance of `MyDisplay`. It is silent in case of an empty playlist - when the `SongLibrary musiclibr` contains an empty playlist thus returns an empty `Optional`. This is checked with the `.isPresent()` method of the Optional library. The same is done with the `nextSong()` and the `prevSong()` methods.

The two ConcreteStates `FullPlaylistMode` and `SongFocusMode` both implement the `public void visualize(Song s)` method and the `String getModeTitle()` method, in different ways: while the `FullPlaylistMode` class prints the whole playlist every time it is called - using the private method `void printAllList()` too in order to do that – the `SongFocusMode` only prints to screen the "current" song.

Both states have a method to return the name of their state when called: `String getModeTitle()`.

1.5 Final comments, and the class `ClientListener`

The duty to create songs and to store them inside the playlist has been left to the class `ClientListener`, and this is where the main function is, too. This class creates the instance of SongLibrary, a MusicPlayer, and an AdvancedControl, and can use their public methods in order to enjoy their songs. The instances of a few `Song` objects have been created for the purpose of showing the project's functionalities.