Task 10: A GUI for the Player

The aim of this task is to implement a user interface (graphical user interface, GUI for short) that allows for convenient control of the functionality developed so far.

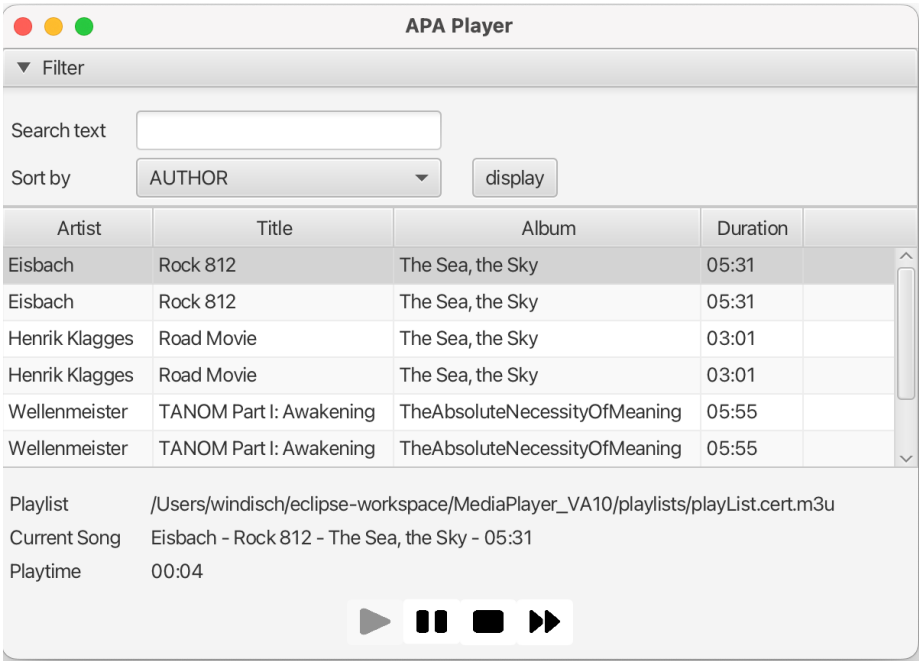


Figure 1: User interface of the Media Player

Project Setup

Copy your project from task 9, remove the tests in the cert folder and replace the tests in the "tests" folder with those of the current task (archive tests.zip). The classes in tests.zip are already in a package structure, pay attention to the correct folder structure in the tests folder to map the studiplayer.test package.

There are further archives, please handle them as follows after downloading

- „examples.zip: Contains the introductory application "GuessANumber"
  After unpacking, save all files in a new source folder "examples"
- "playlists.zip": Contains prepared playlists in M3U format
  Save all files in a folder "playlists" after unpacking
- "icons.zip": Contains graphic files for the buttons
  After unpacking, save all files under "src" in a new folder "icons"

How to use JavaFX with Eclipse

Before the development of a JavaFX application can begin, some preparations have to be made[1]:

1. Java Version: Make sure that you are using Java 13 or higher in Eclipse, see figure below.

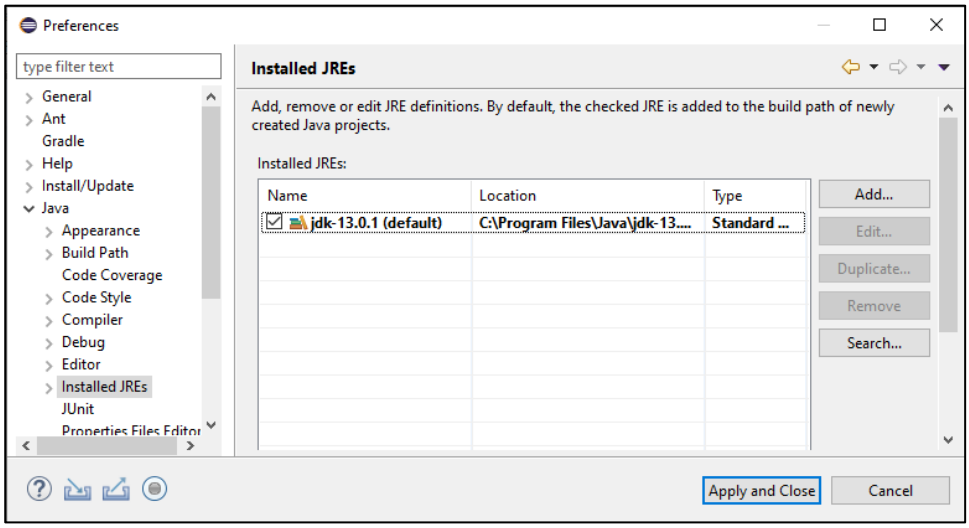


Figure 2: Verifying your installed Java version

2. Install the SDK (software development kit) for JavaFX from https://gluonhq.com/products/javafx/, selecting the appropriate version for your operating system and processor architecture. For Windows, the installation can take place under C:\bin\javafx-sdk-11.0.2, for example.
3. Create a new "user library" called "JavaFX" under Window > Preferences > Java > Build Path > User Libraries by clicking on the "New…" button and entering the name of the library in the following dialog. Then use the "Add external jars" button to add all jar files to the lib directory of your JavaFX installation. For the installation location of point 2 (Windows!), select all files in C:\bin\javafx-sdk-11.0.2\lib:

[1] Please refer to https://openjfx.io/openjfx-docs/#IDE-Eclipse.

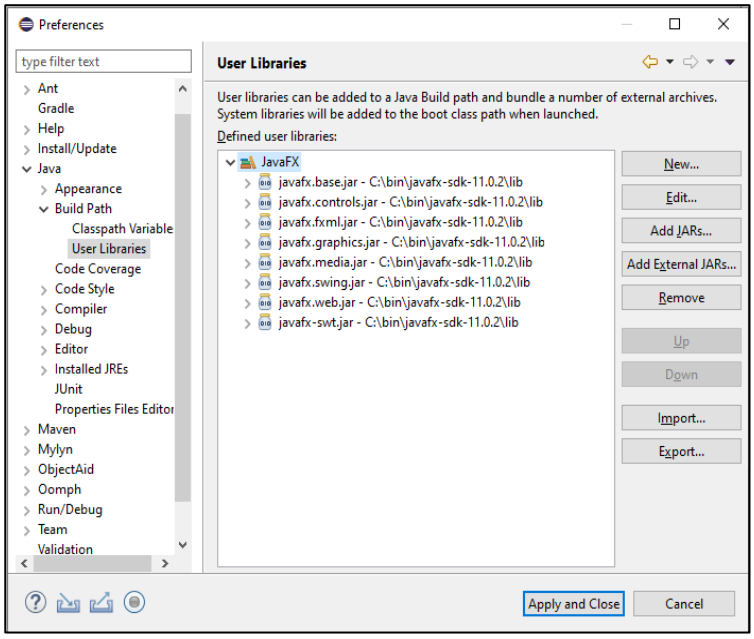


Figure 3: Defining the JavaFX User Library

4. Add the user library to your build path. To do this, open the dialog (right mouse button on the project name in the Package Explorer > Java Build Path) and add the new user library "JavaFX" (Add Library > Select User Library > next > Select JavaFX).
5. Starting a program: With the help of the "JavaFX" user library, your classes that rely on JavaFX can be compiled. For execution, the JVM must be informed where the JavaFX jars are located. For this purpose, a "run configuration" must be created for each application in Eclipse (analogue "debug configuration" for debugging). To do this, right-click on your main class (e.g. GuessANumber.java) in the Package Explorer and select the menu item Run As > Run Configurations. In the following dialog, select "Java Application" and click on the New button (top left in the dialogue), whereupon fields for editing the properties of your Run Configuration open. It is particularly important to ensure that the entry

   ```
   --module-path "C:\bin\javafx-sdk-11.0.2\lib" --add-modules javafx.controls
   ```

   exists in field „VM arguments“. Also check option “-XstartOnFirstThread…”.

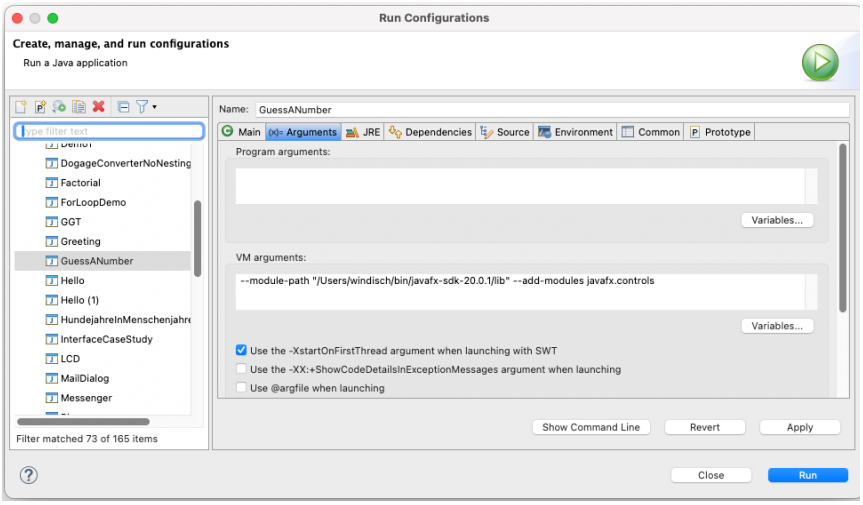


Figure 3: Defining a Run Configuration for GuessANumber

6. You should now be able to run the application (e.g. GuessANumber). After the first execution, the application can also be started as usual via the green arrow button or via the bug button (for debugging).

Number guessing: an introductory example

Before you start implementing the graphical user interface for your media player, please study the example GuessANumber.java in the examples subdirectory.

This example code creates a simple graphical user interface for a number guessing game. Experiment with the code and change

- Layout
- Event creation
- Event handling

Note: In the GuessANumber.java class, the elements of the graphical user interface (buttons etc.) are implemented together with the handling of the generated events (lambda expressions). This procedure is only advisable for small examples. For more complex interfaces, the events should always be handled in separate classes. See also the OO design pattern MVC: (Model-View-Control).

Subtask a) Package studiplayer.ui

Create a new package "studiplayer.ui" and save the predefined Java classes "Song.java" and "SongTable.java" (see APA course room/VA10) there.

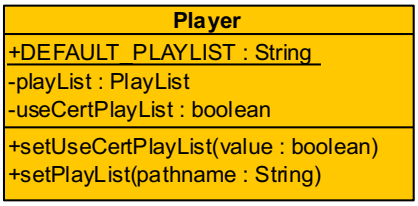
Now create the Player class. The player should fulfil the following initial requirements:

- Player specialises `javafx.application.Application` and provides a default constructor
- Player implements a `start(Stage stage)` method and a `main` method with the typical code to start the application
- The `start()` method sets the title and sets its main pane with a size of 600x400 pixels in its first (and only) scene.

After implementing these requirements, you should see an empty window with the appropriate title.

To test your package you can use the tests provided in file **TestSubtaskA.java**.

Subtask b) Loading a PlayList



The player needs a PlayList instance which should be loaded when the application is started in one of two ways. Either the user is presented with a dialog to select an m3u file or a default m3u file name is used to load the PlayList. Implement the following logic:

- A JavaFX `FileChooser` should be used for the selection of a m3u filename, see e.g. https://docs.oracle.com/javafx/2/ui_controls/file-chooser.htm
- It should be possible to deactivate the FileChooser for the execution of the tests. Define a boolean attribute `useCertPlayList` and a setter for it. The attribute should initially be set to "false".

Note: `useCertPlayList` controls whether the default m3u file "playList.cert.m3u" is loaded at the beginning or whether the user selects an m3u file via a file selection dialog, which is then loaded. This method is used in the acceptance test to run the player without a file selection.

Also implement the method `loadPlayList(String pathname)`, which reloads the player PlayList based on the given path name. If the name is "null" or an empty string, the default PlayList should be loaded. To do this, define a public constant called DEFAULT_PLAYLIST and initialize it with the value "playList.cert.m3u".

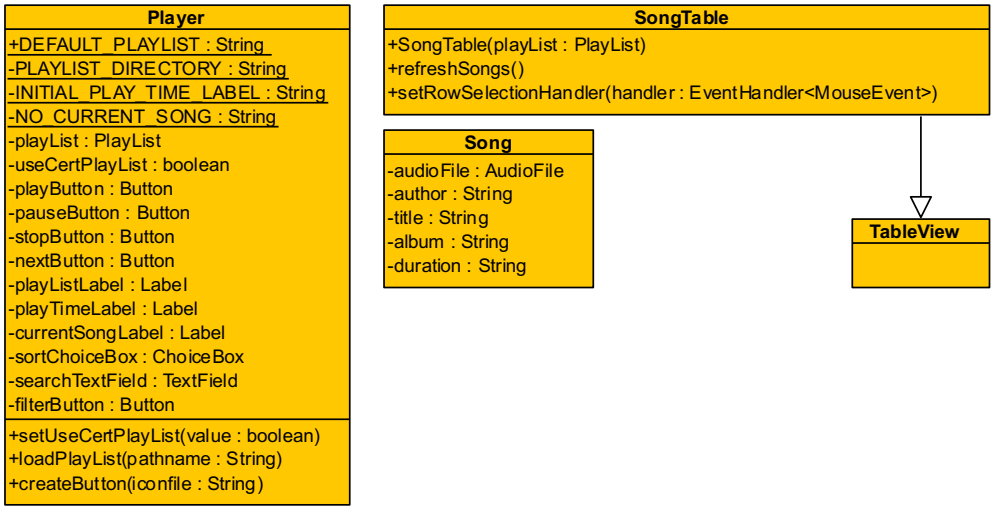
To test your code you can use the tests provided in the file **TestTeilaufgabeB.java**.

Subtask c) Basic Dialog Layout

This subtask is about developing the dialog with regard to the controls and the layout, but still without event handling. Create all controls as shown in Figure 1 and use suitable JavaFX panes to achieve the required layout.

Note: Sketch a layout hierarchy on a notepad first to think about the structure for controls and panes. Details on layout hierarchies can be found in the script. The sketch does not have to be handed in.

Some controls must be referenced in corresponding attributes for tests to work correctly. Define the attributes according to the following class diagram.



The following considerations can be taken into account for the layout:

- The PlayList table in the middle of the dialog should grow accordingly when the window is enlarged, which is why a `BorderPane` with a `TableView` in the centre area is recommended. You can use the `SongTable` for this purpose (see class diagram), please also refer to note 2 below.
- The filter form can be realized with a `TitledPane`, which should be placed in the top area of the `BorderPane`. See https://docs.oracle.com/javafx/2/ui_controls/accordion-titledpane.htm.
- The bottom area has two elements displayed one below the other, which is why a `VBox` is

suitable here:

- The first area provides information on the current song in tabular form (see `Grid`)
- The second area displays the buttons for song control. The method `createButton(String iconfile)` can be used to create the buttons, see note 3 below. The parameter `iconfile` specifies the file name of the graphic file to be loaded (e.g. "play.jpg")

Create the corresponding panels and controls and connect the elements to achieve the layout shown in Figure 1.

Notes:

1. Set up the following additional constant player attributes:
   - `PLAYLIST_DIRECTORY` (private): directory with playlists to choose from
   - `INITIAL_PLAY_TIME_LABEL` (private): „00:00“
   - `NO_CURRENT_SONG` (private): " - " // no song chosen yet
2. The `SongTable` class is available to make it easier to work with tables. A `SongTable` instance is a `TableView` instance that displays the `AudioFiles` contained in the `PlayList`. The `Song` class is used as the "row model" for each table row, which contains the values to be displayed (artist, title, album, duration) as well as a reference to the underlying `AudioFile`. The latter is relevant for event handling when a row is selected. The click handler is set using the `setRowSelectionHandler()` method. A `SongTable` can be updated using the method `refreshSongs()`, e.g. when a new m3u file is loaded.
3. The following method can be used to create a button with an icon:

```
private Button createButton(String iconfile) {
    Button button = null;
    try {
        URL url = getClass().getResource("/icons/" + iconfile);
        Image icon = new Image(url.toString());
        ImageView imageView = new ImageView(icon);
        imageView.setFitHeight(20);
        imageView.setFitWidth(20);
        button = new Button("", imageView);
        button.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
        button.setStyle("-fx-background-color: #fff;");
    } catch (Exception e) {
        System.out.println("Image " + "icons/"
            + iconfile + " not found!");
        System.exit(-1);
    }
    return button;
}
```

4. Correct labels for selection of sort method
   `SortCriterion` elements are displayed to select the sorting method. By default, these elements return the result of `name()`, i.e. uppercase strings from the enumeration type. A simple way to use the terms displayed in the table columns is to specify a `toString()` method in the enumeration type that returns a corresponding String for each `ordinal()` value. This can be done by specifying an array with the names and using `ordinal()` as an index into the array.

To test your code you can use the tests provided in **TestSubtaskC.java**.

Subtask d) Event Handling – Filtering and Sorting the PlayList

This subtask is intended to connect the chosen sort method with the `PlayList`. After clicking on the "display" button in the filter, the current values for the search text and the sort method need to be set in the `PlayList`, resulting in a correspondingly filtered or sorted play list. Finally, the `SongTable` must be updated using the `refreshSongs()` method (see Figure 4).

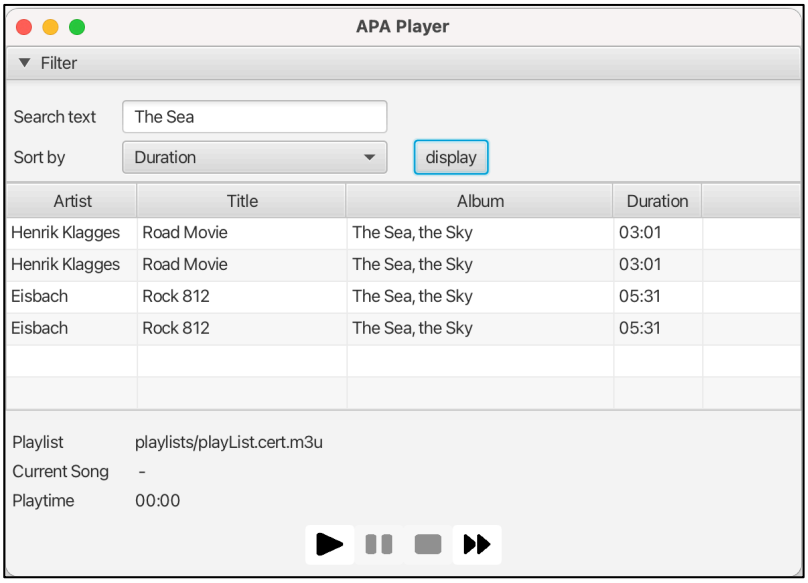


Figure 4: A filtered and sorted PlayList

Subtask e) Event Handling – Controlling the playback sequence

The actual control is carried out via the buttons and by clicking on rows in the table ("jump to this song and play it").

For the event handling of the buttons, it is recommended to define a local method and call it in the handler lambda. The result is a better modularised application with the option of reusing code. Example for the play button:

```
playButton.setOnAction(e -> {
    playCurrentSong();
});
```

The first step is to establish the basic event handling. The actual playing of songs and counting up the playback time is programmed in subtask f). Both activities must be executed in parallel to the user interface in separate threads, as otherwise the user interface would "freeze".

Implement the above idea for all buttons with the following initial event handling, whereby instead of the actual `AudioFile` action (play/pause/stop) only information is to be output to the console (see example output below):

- Play Button:
  → update the information about the current song and set the playback time to "00:00"
  → adjust the button states (`setDisable(true|false)`), Figure 1 and Figure 4 show the state of the buttons when a song is playing or the player is stopped.
  → start playing the current song

- Pause Button:
  → when a song is being played: pause playback
  → when a song is paused: continue playback
- Stop Button:
  → stop the current playback
  → adapt button states
  → reset playtime
- Next Button:
  → stop current playback (if active)
  → jump to next song
  → display song information and set playtime to „00:00“
  → adapt button states

Corresponding console outputs for a sample click sequence

```
play > stop > play > pause > next > stop
```

could look as follows based on the PlayList "playList.cert.m3u":

```
Playing Wellenmeister - TANOM Part I: Awakening - TheAbsoluteNecessityOfMeaning - 05:55
Filename is wellenmeister_awakening.ogg

Stopping Wellenmeister - TANOM Part I: Awakening - TheAbsoluteNecessityOfMeaning - 05:55
Filename is wellenmeister_awakening.ogg

Playing Wellenmeister - TANOM Part I: Awakening - TheAbsoluteNecessityOfMeaning - 05:55
Filename is wellenmeister_awakening.ogg

Pausing Wellenmeister - TANOM Part I: Awakening - TheAbsoluteNecessityOfMeaning - 05:55
Filename is wellenmeister_awakening.ogg

Switching to next audio file: stopped = false, paused = true
Stopping Wellenmeister - TANOM Part I: Awakening - TheAbsoluteNecessityOfMeaning - 05:55
Filename is wellenmeister_awakening.ogg
Playing Henrik Klagges - Road Movie - The Sea, the Sky - 03:01
Filename is special.oGg
Switched to next audio file: stopped = false, paused = true

Stopping Henrik Klagges - Road Movie - The Sea, the Sky - 03:01
Filename is special.oGg
```

Finally, use the method `setRowSelectionHandler` of your `SongTable` instance for event handling to play selected rows in the table. Use the corresponding logic in the `PlayList`, stop a currently playing song if necessary and start the selected one. You obtain the selection via the method `getSelectionModel()`.

Hints:

1. It makes sense to set the button states (disabled yes/no) calling an auxiliary method
   ```
   private void setButtonStates(boolean playButtonState,
       boolean pauseButtonState, boolean stopButtonState,
       boolean nextButtonState)
   ```
   as this is needed at various points.
2. It also makes sense to provide a method for updating the current song and the current playback time. The following is a basic framework as a starting point:
   ```
   private void updateSongInfo(AudioFile af) {
       Platform.runlater(() -> {
           if (af == null) {
               // set currentSongLabel and playTimeLabel
           } else {
   ```

```
                // set currentSongLabel and playTimeLabel
            }
        });
    }
```

`Platform.runLater(<lambda>)` causes `<lambda>` to be executed in parallel in a JavaFX thread. This is necessary if, for example, the timer thread (see subtask f) updates the playback time. Non-JavaFX threads cannot access JavaFX controls, which would lead to an `Exception`.

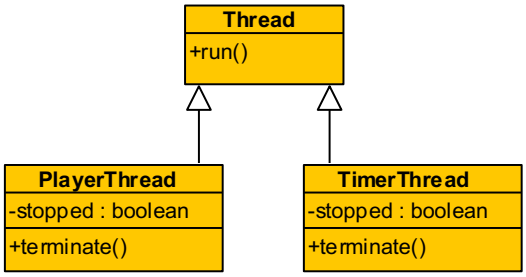
Note: the above method always updates both labels, which is not always necessary. Feel free to define your own methods to improve this!

Subtask f) Player- and Timer-Thread

The final step is to implement the playback of songs. The obvious approach here would be to call `audioFile.play()` directly in the `playCurrentSong()` handler method. However, this would block the JavaFX event handler thread as long as the song is being played. The result would be that no more event handling could be carried out, our dialog would be "frozen".

The solution is to execute the playback of songs in a separate thread - the PlayerThread. It should implement the following logic:

- The thread can be stopped externally by calling the `terminate()` method. It sets a boolean attribute `stopped`, which should exit the loop in `run()`.
- As long as the thread has not been stopped, the current song is being played. Before starting the playback process, the song is selected in the table (see class `SongTable`).
- After a song has finished playing, the next song in the PlayList should be played unless the thread is not to be stopped.



Implement the thread described above as an inner class of the `Player` class. This has the advantage that attributes (e.g. `playList`) and methods (e.g. `updateSongInfo()`) of the `Player` class can be accessed.

Start and stop the `PlayerThread` as part of the event handling at the appropriate points. It is a good idea to introduce an attribute referencing the thread instance. A non-zero value indicates whether a playback process is currently active or not.

A second thread - the TimerThread – should now be defined to update the playback time. This new thread implements the following logic:

- Just like the player thread, the timer can be stopped externally by calling `terminate()`.
- As long as the thread has not been stopped, it updates the song information (playback time and current song).
- Updates should be performed periodically to avoid updating unnecessarily often.

Implement the thread described above as an inner class of the `Player` class. Start and stop the timer thread as part of the event handling at the appropriate points. It is a good idea to introduce an attribute as a reference to the thread instance and, if necessary, methods to simplify starting and stopping the threads.

Notes:

1. The two threads do not always have to be created and terminated together. With "pause", for example, the player thread can remain active while the timer thread is terminated and may have to be created again later.
2. A modular approach to realising the above statement would be to introduce and use the following methods:

```
private void startThreads(boolean onlyTimer) {
    // create and start TimerThread
    // if !onlyTimer: create and start PlayerThread
}
private void terminateThreads(boolean onlyTimer) {
    // terminate TimerThread, set thread reference to null
    // if !onlyTimer: terminate PlayerThread,
    // set thread reference to null
}
```

Attention: Test your application interactively and extensively before you start the acceptance tests!

Hints on Acceptance of Your Solution

Download all acceptance tests for task 9 and save them in the source folder “cert” of your project. Then run the acceptance tests in Eclipse.

Note: You can run all tests together by using the right mouse button on the “cert” source folder and selecting "run as > JUNIT test". However, this requires that you have previously created a "run configuration" similar to the one you created for the player. The "VM arguments" field must also be filled accordingly!

If all tests run without errors, please email the following Java files as file attachments with the subject "ex-10" to the APA server:

- From package `studiplayer.audio`:
  AlbumComparator.java, AudioFileFactory.java, AudioFile.java, AuthorComparator.java, ControllablePlayListIterator.java, DurationComparator.java, NotPlayableException.java, PlayList.java, SampledFile.java, SortCriterion.java, TaggedFile.java, TitleComparator.java, WavFile.java
- From package `studiplayer.ui`:
  Player.java, SongTable.java, Song.java