Task 8: The Playlist

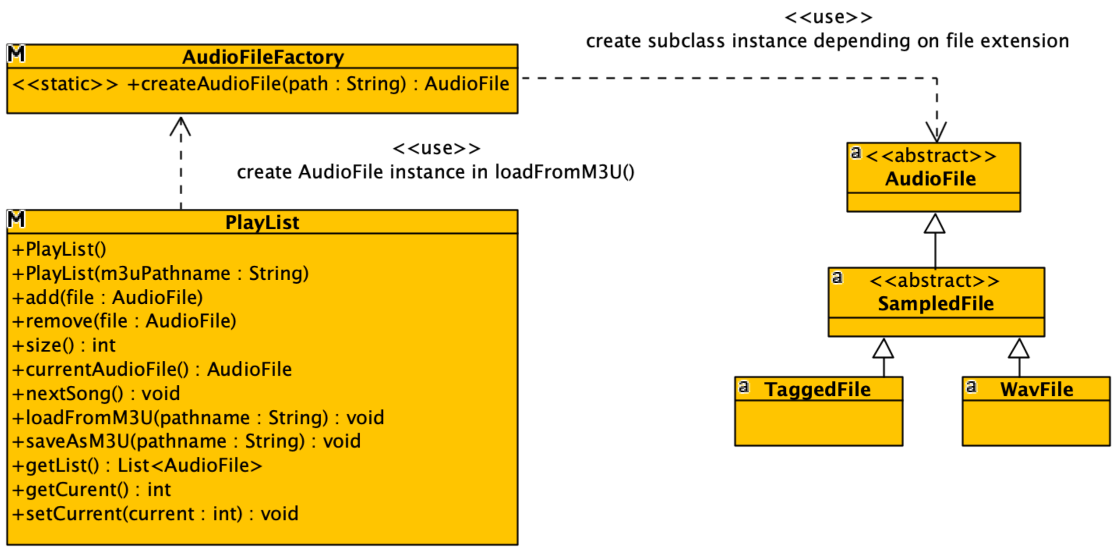
The aim of task 8 is to implement a play list to manage the audio files to be played. The class `PlayList` is to be implemented for this purpose.

A `PlayList` instance should offer the following functionality

- Saving and deleting `AudioFile` instances in and from the list
- Current playback position in the list:
  - Set and read the current `AudioFile` instance
  - Switching to the next `AudioFile`
- Sequential playback mode
- Loading the list from a text file in M3U format
- Saving the list in M3U format

A M3U file contains file paths of audio files, one per line. When reading in an M3U file, either a `WavFile` or a `TaggedFile` object must be created according to the file extension of the audio file. This task is to be performed by a simple implementation of a so-called "factory method"[1]. The advantage of the factory method lies in the centralisation of object creation, which reduces the possibility of errors: If additional `AudioFile` types are added, only the `createAudioFile()` method needs to be adapted, which introduces only small changes in one class.

The following illustration shows the final class diagram.



<<use>>
create subclass instance depending on file extension
AudioFileFactory
<<static>> +createAudioFile(path : String) : AudioFile
<<use>>
create AudioFile instance in loadFromM3U()
PlayList
+PlayList()
+PlayList(m3uPathname : String)
+add(file : AudioFile)
+remove(file : AudioFile)
+size() : int
+currentAudioFile() : AudioFile
+nextSong() : void
+loadFromM3U(pathname : String) : void
+saveAsM3U(pathname : String) : void
+getList() : List<AudioFile>
+getCurent() : int
+setCurrent(current : int) : void
<<abstract>> AudioFile
<<abstract>> SampledFile
TaggedFile
WavFile

[1] Factory method design pattern, see e.g. here: https://refactoring.guru/design-patterns/factory-method

Project Setup

Make a copy of your project for task 7, remove the tests in the cert folder and replace the tests in the "tests" folder with those of the current task (archive tests.zip). Also create a "source folder" called "examples" and unzip "examples.zip" into it.

Subtask a) Storing Audio Files in a List

Before we can begin with the implementation of the methods of class `PlayList`, the question of where the `AudioFile` references of the `PlayList` should be stored must be answered. There are basically two options: Inheritance - for example by deriving from `LinkedList` - or aggregation, i.e. introducing an attribute of type `LinkedList` or another list implementation.

Due to the greater flexibility of the aggregation variant, the `AudioFile` instances of the play list should be saved in a collection of type `LinkedList`. Define a corresponding attribute in the `PlayList` class and implement the corresponding getter `getList()`.

Notes:

- The getter `getList()` is primarily used here to simplify the JUNIT tests. In the next task you will develop an iterator that allows for iteration of the `AudioFiles` contained in a `PlayList`. The getter can then be omitted.
- The result type of `getList()` is `List<AudioFile>`. This means that any object reference to a type that implements the List interface with base type AudioFile can be returned. If you use a `LinkedList<AudioFile>` instance, for example, this requirement is met, just as with an `ArrayList<AudioFile>` instance. This specification of the result type of the method therefore allows more flexibility with regard to the implementation.

Then add methods for managing the list content:

- `add(AudioFile file)`: Insert the given AudioFile reference into the list
- `remove(AudioFile file)`: Remove the given AudioFile reference from the list
- `int size()`: Return the number of AudioFile references currently in the list

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

Subtask b) Current Position and Switching to the Next Song

The class `PlayList` should manage the current position in the play list. Realise the corresponding getter and setter methods for reading and setting the position. The parameterless constructor should initialise the position appropriately.

Also realise the method `currentAudioFile()`, which returns the `AudioFile` object at the current position. If there is no valid current position, i.e. the internally managed index is invalid, the method should return null. An invalid index results, for example, if the list is empty or the index is at a position that is no longer valid after deleting `PlayList` entries.

The `nextSong()` method is intended to advance the playback position to the next song. If the end of the list is reached, the playback position should be set to the first song in the list (position 0). If the internal index is at an invalid position when the method is called, the index should also be set to position 0.

To test your implementation, you can use the tests provided in file **TestSubtaskB.java**.

Sample code for writing and reading text files

The remaining two methods `saveAsM3U()` and `loadFromM3U()` write or read text files in M3U format. In preparation for this, you will find an example programme in the provided class `TextFileIO` (see examples.zip), which writes an array of strings to a text file and then reads the created text file again. Each line is preceded by a line number when it is read in. The main programme then outputs the read-in strings to the console.

Use the comments in the `TextFileIO` class and execute or debug the code to gain the necessary basic understanding for the implementation of the two remaining methods.

Subtask c) Saving a PlayList in M3U Format

The `saveAsM3U()` method is intended to save the `AudioFile` list of a `PlayList` in a text file in M3U format.

An M3U file has the extension "m3u" and can contain comment lines, empty lines or lines with file paths for audio files. A comment line begins with a "#" followed by any text. An empty line contains no characters or only whitespace characters.

The M3U format is illustrated by the following example:

```
# My best songs
Alternative\Band - Song.mp3

Classical\Other Band - New Song.mp3
Stuff.mp3
D:\More Music\Foo.mp3
..\Other Music\Bar.mp3
```

The above file contains a comment, an empty line and 5 file paths.

Now implement the method `saveAsM3U(String pathname)`, which saves the file paths of the AudioFile instances contained in the list in a file with the name given by pathname.

To test your implementation, you can use the tests provided in file **TestSubtaskC.java**.

Subtask d) Factory Method for Creating AudioFile instances

Until now, `AudioFile` instances (`TaggedFiles`, `WavFiles`) were created where they were needed. As practical as this procedure seems to be, it is also prone to errors! The type of `AudioFile` instance to be created depends on the extension of the file name. For a correct program, the proper constructor must be called depending on the extension of the file name.

To avoid such sources of error, the creation of `AudioFile` instances is centralised in a factory method. The static method `createAudioFile(String path)` creates the corresponding `AudioFile` instance depending on the extension contained in path and returns it:

- A `WavFile` instance is created for the "wav" extension.
- If the extensions "ogg" or "mp3" are present, a `TaggedFile` instance is created.
- If none of the above extensions are present, a `RuntimeException` with a meaningful error message should be thrown.
- Example: `AudioFileFactory.createAudioFile("media/example.txt")` should throw a `RuntimeException` with the error text 'Unknown suffix for AudioFile "media/example.txt"'.

Note: Ignore upper and lower case for the file extension comparisons.

You can use the tests provided in file **TestSubtaskD.java** to test your implementation.

Subtask e) Initialising a PlayList from a M3U File

Now implement the method `loadFromM3U(String path)`, which opens the M3U file with the given file path and creates an audio file for each audio file path contained in the file using the factory method. The created audio files should be saved in the PlayList.

Note: Initialise the `AudioFile` list before you start reading the file. This includes deleting existing list entries and resetting the current playback position.

Finally, add the correct implementation of the second constructor, which uses the given path name to initialize the PlayList from the corresponding M3U file.

To test your implementation, you can use the tests provided in **TestSubtaskE.java**.

Hints on Acceptance of Your Solution

Download all acceptance tests for task 8 and save them in the source folder “cert” of your project. Then run the acceptance tests in Eclipse.

If all tests run without errors, please email the following Java files as file attachments with the subject "ex-08" to the APA server:

- AudioFile.java,
- AudioFileFactory.java,
- PlayList.java,
- SampledFile.java,
- TaggedFile.java,
- WavFile.java