Technische Hochschule Ingolstadt

Task 9: Filtering, Sorting and Iterations

In task 9 we extend class `PlayList` for managing audio files. Sorting criteria are added and the option to play only a part of the PlayList. A configurable iterator is to be developed for this purpose.

A `PlayList` instance should offer the following additional functionality

- Configurable sort order and search filter
- All `AudioFile` instances of a `PlayList` instance should be iterable via the `Iterable` and `Iterator` interfaces

`Comparator` implementations are to be developed to realize the following sorting options: by album, by author, by title and by duration. The `SortCriterion` enumeration type is used to control which sorting is currently applied in the `PlayList`.

A controllable iterator is required for iteration of a `PlayList` instance, since the player user interface can start playing files from position 0 and also requires the option of starting with a specific song. The iterator should therefore offer the following functionality:

- Iteration over a given list of AudioFile instances (a `PlayList` instance), starting with the first element
- Jumping to a specific list position and continuing the iteration from the new position.

In addition, this task is intended to improve error handling. We use our own exception class for throwing typical errors as "checked exceptions" on order to request that the error be handled by the developer.

Finally, all classes should be moved into a package in order to prepare the final structure of the application.

Figure 1 shows the final class diagram.

Project Setup

Copy your project from task 8 into a new project, 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.

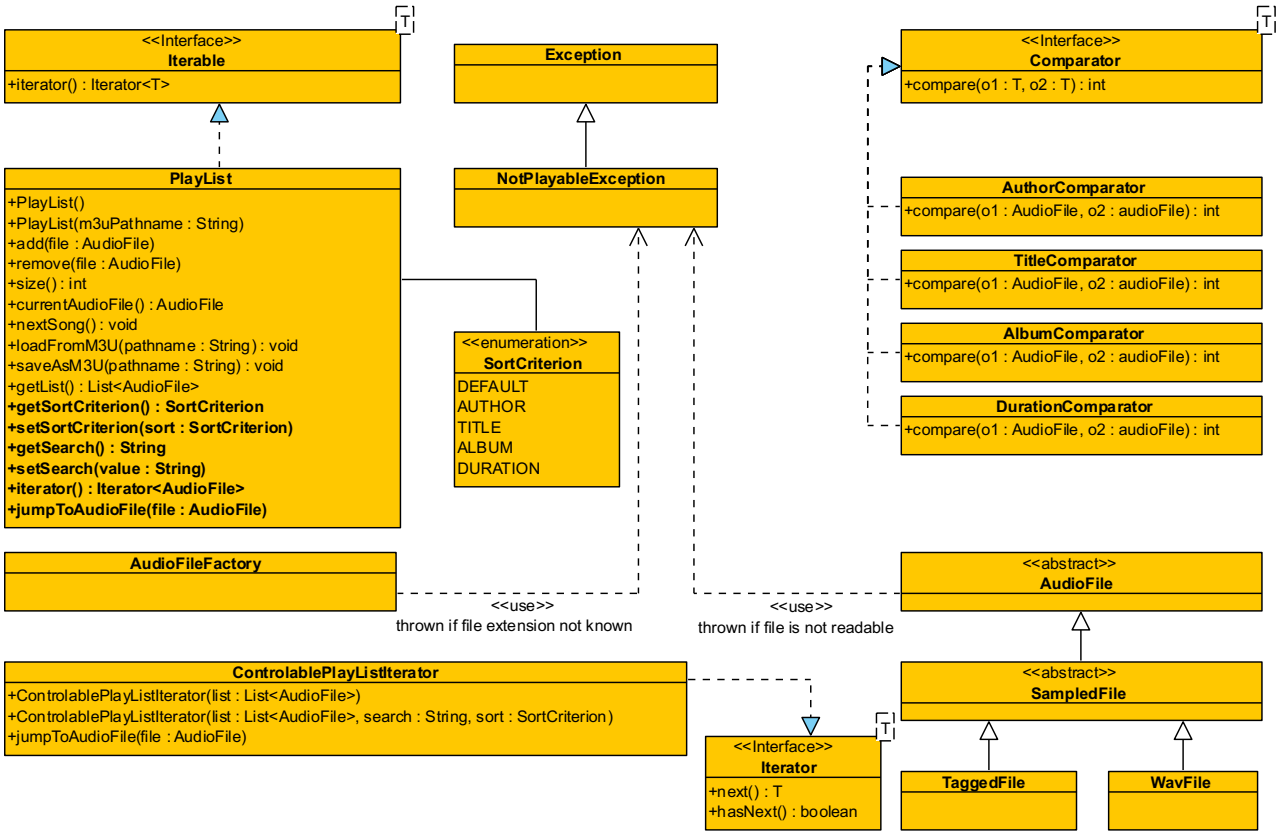

Figure 1: Final Class Diagram

Subtask a) New Package Structure

With the introduction of the user interface in the next task, classes that deal with the user interface should be in a different package than classes that deal with the handling of `AudioFile` instances. The `studiplayer.audio` package should be introduced for this purpose. This package includes the inheritance structure of the `AudioFile` classes, the `AudioFileFactory` and the `PlayList`. Create the package and move all previously created classes into this package.

Subtask b) Exception Handling with NotPlayableException

Create a class `NotPlayableException`, which should be derived from `Exception`. Then add the following constructors, link them to the constructors of the `Exception` class and save the pathname parameter in a suitable attribute.

- `NotPlayableException(String pathname, String msg)`
- `NotPlayableException(String pathname, Throwable t)`
- `NotPlayableException(String pathname, String msg, Throwable t)`

Note: With regard to the question of which constructor parameter is to be stored in which attribute, bear in mind that `Exception` already specifies (at least) one attribute itself!

In the next step, replace `RuntimeException` by `NotPlayableException`:

- the `play` method should throw a `NotPlayableException` if errors occur when playing the song
- the constructors of the classes `AudioFile`, `SampledFile`, `WavFile` and `TaggedFile` with a path name as a parameter should use `NotPlayableException` if the file is not readable
- method `createAudioFile` of class `AudioFileFactory` should throw a `NotPlayableException` if an unknown file extension is recognised
- method `readAndStoreTags` of class `TaggedFile` should throw a `NotPlayableException` if the tags cannot be read
- method `readAndSetDurationFromFile` of class `WavFile` should throw a `NotPlayableException` if the parameters cannot be read

Finally, method `loadFromM3U` should be adapted so that information contained in a caught `NotPlayableException` (see `printStackTrace`), is output to the console. Processing of the M3U file should then be continued, the program must not terminate!

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

Subtask c) Controllable iterator for AudioFile instances

Create a new class `ControllablePlayListIterator` which implements the `Iterator` interface. The constructor of the class should accept a list of `AudioFile` instances as a parameter.

Note: In Java, the Iterator interface describes how to iterate over a set of objects. The methods hasNext and next must be implemented for that matter. You can find more details in the script.

Implement the iterator so that the `hasNext` and `next` methods allow iteration over all elements of a `PlayList`. Try out the iterator by passing a list of `AudioFile` instances, for example:

```
List<AudioFile> files = Arrays.asList(
    new TaggedFile("audiofiles/Rock 812.mp3"),
    new TaggedFile("audiofiles/Eisbach Deep Snow.ogg"),
    new TaggedFile("audiofiles/wellenmeister_awakening.ogg"));
ControllablePlayListIterator it =
    new ControllablePlayListIterator(files);
while(it.hasNext())
    System.out.println(it.next());
```

Now add method `AudioFile jumpToAudioFile(AudioFile file)`. The method should change the state of the iterator so that the next call of `next` returns the song after the given AudioFile reference. `jumpToNextAudioFile` should return its argument or "null" if the argument is not contained in the list.

Using the same list as before, an example would be:

```
ControllablePlayListIterator it =
    new ControllablePlayListIterator(files);
it.jumpToAudioFile(files.get(1));
while(it.hasNext()) {
    System.out.println(it.next());
}
```

This example should skip the first two songs and only output the third song in the list to the console.

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

Subtask d) Sorting AudioFile instances

The basis for sorting according to various criteria in Java is the interface `Comparator<T>`. Create the four classes `AlbumComparator`, `AuthorComparator`, `TitleComparator` and `DurationComparator`, each of which implements `Comparator<T>`. Each comparator should establish an ordering for AudioFile instances according to the respective criterion. When comparing character strings, upper and lower case must be taken into account, e. g. is "TANOM..." smaller than "no.wav...". This corresponds to the standard comparison for character strings in Java.

Notes:

1. The Comparator<T> interface defines the method int compare(T o1, T o2). Use a suitable data type for T so that all types of AudioFile instances can be compared. A comparison must then be carried out for the parameters o1 and o2 in the compare method. Details on controlling the sequence can be found in the script or here: https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#compare-T-T-.
2. Check the parameters of compare for null values and throw a runtime exception if necessary.

There is a specialty when sorting according to the playback duration or the album: The attribute required for the comparison is not defined in the `AudioFile` class! Make sure that instances that have no duration or album information appear at the beginning of the sorting sequence. Moreover, the `DurationComparator` should not use the methods `formatDuration` or `formatPosition` for efficiency reasons.

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

Subtask e) Additional PlayList Configuration

Add two attributes in class `PlayList`: `String search` and `SortCriterion sortCriterion`. Also add the corresponding getters and setters. The `SortCriterion` data type is an enumeration type that is intended to map the possible sorting sequences (as well as "no sorting"). Create the enumeration data type so that it contains the following elements:

```
DEFAULT, AUTHOR, TITLE, ALBUM, DURATION
```

The default value of `SortCriterion` is `DEFAULT` and should be used for the corresponding attribute when creating a PlayList instance.

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

Subtask f) Iterate over AudioFile instances in PlayList

Now the basis for sorting (comparator implementations and sort criterion) and a search filter string are ready to use. The aim is now to iterate the play list with respect to the specified filter and sorting criteria. For this purpose, a new constructor is created in the `ControllablePlayListIterator` which receives the current play list and the configuration as parameters (sorting and search filter).

Extend class `ControllablePlayListIterator` with a second constructor that receives the

parameters search and sortCriterion in addition to the `AudioFile` list. The constructor should create a new list from the given one, which can then be iterated. The following requirements must be taken into account:

- If search is `null` or empty, all `AudioFile` instances of the `PlayList` should be used. Otherwise, only `AudioFile` instances containing the string search in either the author, title or album should be used
- The resulting list should be sorted using sortCriterion. Depending on the value of the attribute, the appropriate comparator implementation should be used. The exception is DEFAULT, where no sorting is needed

In a next step you can now implement the `Iterable` interface in the class `PlayList` based on the created constructor.

After this customisation, you can use your PlayList in for-each loops. The output will change depending on how you configure the PlayList. For example:

```
PlayList pl = new PlayList();
pl.add(new TaggedFile("audiofiles/Rock 812.mp3"));
pl.add(new TaggedFile("audiofiles/Eisbach Deep Snow.ogg"));
pl.add(new TaggedFile("audiofiles/wellenmeister_awakening.ogg"));
// set different values for search and sortCriterion below...
pl.setSearch("Eisbach");
pl.setSortCriterion(SortCriterion.ALBUM);
// example iteration with for-each
for(AudioFile file: pl) {
    System.out.println(file);
}
```

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

Subtask g) Adapting Class PlayList to use the New Iterator

Finally, the playback control of the PlayList must be switched to use the new iterator. Up to now, a counter was used to mark the current playback position. In the future, the iterator will be used to manage the playback position. The playback sequence is in a PlayList should be controlled via nextSong and currentAudioFile calls. To make this work, these methods need to use the ControllablePlayListIterator accordingly.

Please refer to the following usage examples, which can also be found in file TestSubtaskG.java (testSnippet01 to testSnippet05).

The following lines are given for each example:

```
AudioFile tf1 = new TaggedFile("audiofiles/Rock 812.mp3");
AudioFile tf2 = new TaggedFile("audiofiles/Motiv 5. Symphonie von Beethoven.ogg");
AudioFile tf3 = new TaggedFile("audiofiles/Eisbach Deep Snow.ogg");
```

Snippet	**Output** (shortened)
testSnippet01: PlayList starts from the beginning when the end of the playback sequence is reached.	
`PlayList pl1 = new PlayList();`	`Eisbach - Rock 812 - The Sea, …`

`pl1.add(tf1);`
`System.out.println(pl1.currentAudioFile());`
`pl1.nextSong();`
`System.out.println(pl1.currentAudioFile());`
	`Eisbach - Rock 812 - The Sea, …`

testSnippet02: If nextSong is called, the next song is always delivered, regardless of currentAudioFile.	

`PlayList pl2 = new PlayList();`
`pl2.add(tf1);`
`pl2.add(tf2);`
`pl2.nextSong();`
`System.out.println(pl2.currentAudioFile());`
	`Motiv 5. Symphonie von Beethoven …`

testSnippet03: PlayList also starts from the beginning with several songs when the end is reached.	

`PlayList pl3 = new PlayList();`
`pl3.add(tf1);`
`pl3.add(tf2);`
`System.out.println(pl3.currentAudioFile());`
`pl3.nextSong();`
`System.out.println(pl3.currentAudioFile());`
`pl3.nextSong();`
`System.out.println(pl3.currentAudioFile());`
	`Eisbach - Rock 812 - The Sea, …`
	`Motiv 5. Symphonie von Beethoven …`
	`Eisbach - Rock 812 - The Sea, …`

testSnippet04: If search or sorting is used, this has an influence on currentAudioFile and nextSong	

`PlayList pl4 = new PlayList();`
`pl4.add(tf1);`
`pl4.add(tf2);`
`pl4.add(tf3);`
`pl4.setSortCriterion(SortCriterion.DURATION);`
`System.out.println(pl4.currentAudioFile());`
`pl4.nextSong();`
`System.out.println(pl4.currentAudioFile());`
`pl4.nextSong();`
`System.out.println(pl4.currentAudioFile());`
	`Motiv 5. Symphonie von Beethoven …`
	`Eisbach - Deep Snow - The Sea, …`
	`Eisbach - Rock 812 - The Sea, …`

testSnippet05: If search or sort is used, the playback sequence is reset.	

`PlayList pl5 = new PlayList();`
`pl5.add(tf1);`
`pl5.add(tf2);`
`pl5.add(tf3);`
`System.out.println(pl5.currentAudioFile());`
`pl5.nextSong();`
`pl5.setSearch("Eisbach");`
`System.out.println(pl5.currentAudioFile());`
`pl5.nextSong();`
`System.out.println(pl5.currentAudioFile());`
	`Eisbach - Rock 812 - The Sea, …`
	`Eisbach - Rock 812 - The Sea, …`
	`Eisbach - Deep Snow - The Sea, …`

Notes regarding your implementation:

- Remove the attribute `current` and the corresponding getter/setter methods.
- Modify `currentAudioFile()` and `nextSong()` so that the control of the PlayList is handled by the `ControllablePlayListIterator`. Add any necessary attributes to save iterators or references to the current song in `PlayList`.

    - `currentAudioFile()` should return the currently playing song.
    - `nextSong()` should move the playback sequence to the next song.
- Make sure that the playback sequence starts from the beginning when changing the search filter or the sorting criterion as well as when adding or deleting a song or loading a new M3U file.

Finally, add a method `jumpToAudioFile(AudioFile audioFile)` to `PlayList` and use the corresponding method of the iterator. Make sure that the `PlayList` then continues to play correctly from this position (see testJumpTo), for example starting from the beginning again when the end of the list is reached:

```
PlayList list = new PlayList();
list.add(tf1);
list.add(tf2);
list.add(tf3);
list.jumpToAudioFile(tf2);
System.out.println(list.currentAudioFile());
list.nextSong();
System.out.println(list.currentAudioFile());
list.nextSong();
System.out.println(list.currentAudioFile());
```

The output for this example would be:

```
Motiv 5. Symphonie von Beethoven - Musikschnipsel - 00:06
Eisbach - Deep Snow - The Sea, the Sky - 03:18
Eisbach - Rock 812 - The Sea, the Sky - 05:31
```

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

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.

If all tests run without errors, please email the following Java files as file attachments with the subject "ex-09" to the APA server:

- AlbumComparator.java,
- AudioFile.java,
- AudioFileFactory.java,
- AuthorComparator.java,
- ControllablePlayListIterator.java,
- DurationComparator.java,
- NotPlayableException.java,
- PlayList.java,
- SampledFile.java,
- SortCriterion.java,
- TaggedFile.java,
- TitleComparator.java,
- WavFile.java