Technische Hochschule Ingolstadt

Vorführaufgabe 9: Filtern, Sortieren und Iterieren

Mit der Vorführaufgabe 9 wird die `PlayList` zur Verwaltung von abzuspielenden Audiodateien weiterentwickelt. Es werden Sortier-Kriterien ergänzt und die Möglichkeit, nur einen Teil der `PlayList` abzuspielen. Hierfür soll ein konfigurierbarer Iterator entwickelt werden.

Eine `PlayList`-Instanz soll folgende zusätzliche Funktionalität bieten

- Konfigurierbare Sortierreihenfolge und Suchfilter
- Iterierbarkeit der Klasse `PlayList` über die Schnittstellen `Iterable` und `Iterator`

Für die Sortierreihenfolge sind `Comparator`-Implementierungen zu erstellen, welche nach Album, Author, Title und Duration sortieren. Über den Aufzählungstyp `SortCriterion` soll gesteuert werden, welche Sortierung in der `PlayList` aktuell verwendet wird.

Für die Iteration über die `PlayList` ist ein steuerbarer Iterator notwendig, da der Player sowohl von der Position 0 anfangen kann, Dateien abzuspielen, als auch die Möglichkeit benötigt mit einem bestimmten Lied zu starten. Der Iterator soll daher folgende Funktionalität bieten:

- Iteration über eine gegebene Liste von `AudioFile`-Instanzen, beginnend beim ersten Element
- Springen an eine bestimmte Listenposition und Fortzusetzen der Iteration ab der neuen Position.

Zusätzlich soll mit dieser Vorführaufgabe die Fehlerbehandlung verbessert werden. Eine eigene Exception-Klasse dient dazu, typische Fehler als „checked exceptions“ an entscheidenden Stellen vorzugeben und die Behandlung des Fehlers durch den Java-Compiler zu fordern.

Abschließend sollen alle Klassen in ein Paket verschoben werden, um die Strukturierung der Anwendung für das letzte Aufgabenblatt vorzubereiten.

Abbildung 1 zeigt das finale Klassendiagramm.

Vorbereitungen

Kopieren Sie ihr Projekt zur vorangegangenen Vorführaufgabe, entfernen Sie die Tests im cert-Ordner und ersetzen Sie die Tests im „tests“-Ordner durch die der aktuellen Vorführaufgabe (Archiv tests.zip). Die Klassen in tests.zip sind bereits in einer Package-Struktur, achten Sie auf die korrekte Ordnerstruktur im Ordner tests um das Package `studiplayer.test` abzubilden.

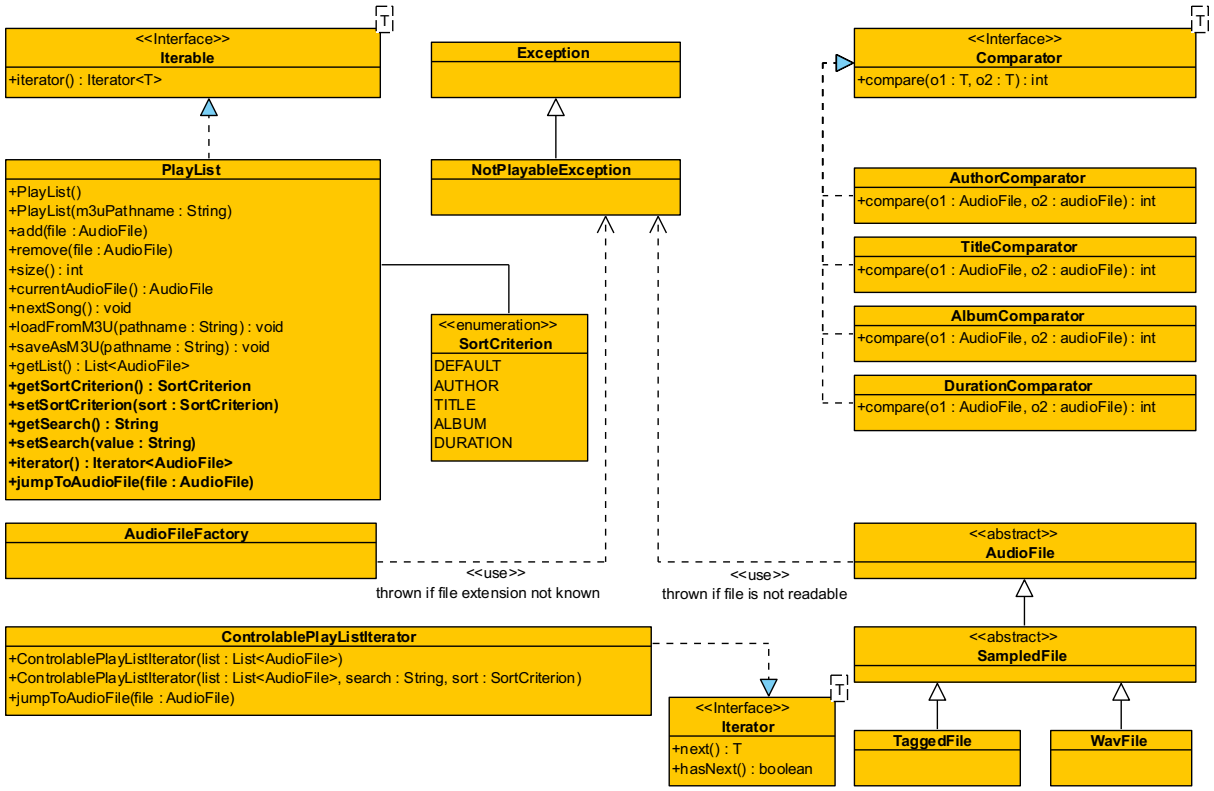

Abbildung 1: Finales Klassendiagramm

Teilaufgabe a) Paketstruktur

Mit der Einführung der Nutzeroberfläche in der nächsten Aufgabe sollen Klassen, die sich mit der visuellen Darstellung befassen, in einem anderen Package liegen als Klassen, die sich mit dem Handling der `AudioFile`-Instanzen befassen. Hierfür soll das Package `studiplayer.audio` eingeführt werden. Dieses umfasst u.a. die Vererbungsstruktur der `AudioFile`-Klassen, die `AudioFileFactory` und die `PlayList`. Erstellen Sie das Package und verschieben Sie alle bisher erstellten Klassen in dieses Package.

Teilaufgabe b) Ausnahmebehandlung mit NotPlayableException

Legen Sie eine Klasse `NotPlayableException` an, die von `Exception` abgeleitet sein soll. Ergänzen Sie anschließend folgende Konstruktoren, verketten Sie diese mit den Konstruktoren der Klasse `Exception` und speichern Sie den Parameter `pathname` in ein geeignetes Attribut.

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

Hinweis: Berücksichtigen Sie bzgl. der Frage, welche Konstruktor-Parameter in welchen Attributen zu speichern sind, dass `Exception` bereits (mindestens) ein Attribut selbst vorgibt!

Im nächsten Schritt soll der Umgang mit auftretenden Ausnahmen an verschiedenen Stellen des Players angepasst werden. Modifizieren Sie hierfür folgende Methoden sowie Verwendungen von

`RuntimeException` so, dass diese die `NotPlayableException` werfen.

- die Methode `play` soll eine `NotPlayableException` werfen, wenn beim Abspielen des Liedes Fehler auftreten
- die Konstruktoren der Klassen `AudioFile`, `SampledFile`, `WavFile` und `TaggedFile` mit einem Pfadnamen als Parameter verwenden die `NotPlayableException`, wenn die Datei nicht lesbar ist
- die Methode `createAudioFile` der Klasse `AudioFileFactory`, soll die `NotPlayableException` werfen, wenn eine unbekannte Dateiextension erkannt wird
- die Methode `readAndStoreTags` der Klasse `TaggedFile`, soll die `NotPlayableException` werfen, wenn die Tags nicht gelesen werden können
- die Methode `readAndSetDurationFromFile` der Klasse `WavFile`, soll die `NotPlayableException` werfen, wenn die Parameter nicht gelesen werden können

Abschließend soll die Methode `loadFromM3U` so angepasst werden, dass Informationen zur `NotPlayableException` (vgl. `printStackTrace`), die bei der Verwendung der `AudioFileFactory` auftreten können, in der Konsole ausgegeben werden. Anschließend soll mit der Verarbeitung der M3U-Datei fortgesetzt werden, das Programm darf sich nicht beenden.

Zum Testen Ihrer Implementierung können Sie die bereitgestellten Tests in der **Datei TestTeilaufgabeB.java** nutzen.

Teilaufgabe c) Steuerbarer Iterator für AudioFile-Instanzen

Erstellen Sie eine neue Klasse `ControllablePlayListIterator` welche die Schnittstelle `Iterator` implementiert. Der Konstruktor der Klasse soll eine Liste von `AudioFile`-Instanzen als Parameter entgegennehmen.

Hinweis: Die Schnittstelle Iterator beschreibt in Java wie über eine Menge von Objekten iteriert werden kann. Zu implementieren sind hierfür die Methoden hasNext und next, welche die Bedingung pro Iterationsschritt und Zugriff auf das aktuelle Element inkl. Springen zum nächsten Element umsetzen. Genaueres finden Sie im Skript.

Implementieren Sie den Iterator so, dass die `hasNext`- und `next`-Methode eine Iteration über alle Elemente einer `PlayList` erlaubt. Probieren Sie den Iterator aus, indem Sie z.B. eine Liste von `AudioFile`-Instanzen übergeben und den Iterator entsprechend verwenden:

```
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());
}
```

Ergänzen Sie anschließend die Methode `AudioFile jumpToAudioFile(AudioFile file)`. Diese soll beim Aufruf den Zustand des Iterators so verändern, dass der nächste Aufruf von `next` das Lied nach der übergebenen `AudioFile`-Instanz zurückliefert. Die Methode selbst soll ihr Argument zurückliefern oder „null“, falls das Argument nicht in der Liste enthalten ist.

Dieselbe Liste wie zuvor verwendend, wäre ein Beispiel:

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

Dieses Beispiel sollte die ersten beiden Lieder überspringen und lediglich das dritte Lied in der Liste auf die Konsole ausgeben.

Zum Testen Ihrer Implementierung können Sie die bereitgestellten Tests in der **Datei TestTeilaufgabeC.java** nutzen.

Teilaufgabe d) Sortieren von AudioFile-Instanzen

Grundlage zur Sortierung nach verschiedenen Kriterien in Java ist die Schnittstelle `Comparator`. Erstellen Sie die vier Klassen `AlbumComparator`, `AuthorComparator`, `TitleComparator` und `DurationComparator`, welche jeweils die Schnittstelle `Comparator` implementieren. Ordnen Sie dabei nach dem jeweiligen Kriterium. Beim Vergleich von Zeichenketten sind Groß- und Kleinschreibung zu berücksichtigen, z.B. ist "TANOM..." < "kein.wav...". Dies entspricht dem Standardvergleich einer Zeichenkette in Java.

Hinweise:

1. Die Schnittstelle `Comparator<T>` definiert die Methode `int compare(T o1, T o2)`. Verwenden Sie für T einen geeigneten Datentyp, damit alle Arten von AudioFile-Instanzen verglichen werden können. In der Methode `compare` ist anschließend für die Parameter `o1` und `o2` ein Vergleich durchzuführen. Details zur Steuerung der Reihenfolge finden Sie im Skript bzw. unter https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#compare-T-T-.
2. Prüfen Sie die Parameter von compare auf null-Werte und werfen Sie ggf. eine Runtime-Exception.

Für die Sortierung nach der Abspieldauer `duration` bzw. dem Album ergibt sich eine Besonderheit: Das für den Vergleich benötigte Attribut ist nicht in der Klasse `AudioFile` definiert! Stellen Sie sicher, dass Instanzen, die keine Duration- bzw. Album-Information haben in der Sortierreihenfolge am Anfang stehen. Der `DurationComparator` sollte zudem aus Effizienzgründen nicht die Methoden `formatDuration` oder `formatPosition` verwenden.

Zum Testen Ihrer Implementierung können Sie die bereitgestellten Tests in der **Datei TestTeilaufgabeD.java** nutzen.

Teilaufgabe e) Zusätzliche Konfiguration der PlayList

Erweitern Sie die Klasse `PlayList` um die Attribute `String search` und `SortCriterion sortCriterion`. Ergänzen Sie entsprechende Getter und Setter. Der Datentyp `SortCriterion` ist ein Aufzählungstyp, welcher die möglichen Sortierreihenfolgen (sowie „keine Sortierung“) abbilden soll. Erstellen Sie den Aufzählungsdatentyp so, dass dieser die folgenden Elemente enthält:

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

Der Default-Wert von `SortCriterion` ist `DEFAULT` und soll bei Erzeugung einer `PlayList`-Instanz für das entsprechenden Attribut verwendet werden.

Zum Testen Ihrer Implementierung können Sie die bereitgestellten Tests in der **Datei TestTeilaufgabeE.java** nutzen.

Teilaufgabe f) Iterieren über AudioFile-Instanzen in PlayList

Die Sortierung (Comparator-Implementierungen und Sortierkriterium) und Suchfilter-String sind vorbereitet. Ziel ist es nun, aus diesen Informationen eine Iteration über die Abspielliste zu erzeugen, die gemäß den Vorgaben gefiltert und sortiert ist. Hierfür wird ein neuer Konstruktor im `ControllablePlayListIterator` erstellt, welcher die aktuelle Abspielliste und die Konfiguration als Parameter erhält (Sortierung und Suchfilter).

Erweitern Sie die Klasse `ControllablePlayListIterator` um einen zweiten Konstruktor, der neben der `AudioFile`-Liste die Parameter `search` und `sortCriterion` erhält. Der Konstruktor soll aus der übergebenen Liste eine neue Liste erzeugen, über die dann iteriert werden kann. Dabei sind folgende Anforderungen zu berücksichtigen:

- Ist search `null` oder leer sollen alle `AudioFile`-Instanzen der `PlayList` verwendet werden. Andernfalls sollen nur `AudioFile`-Instanzen verwendet werden, bei denen entweder im Autor, Titel oder Album die Zeichenkette `search` enthalten ist
- Das Ergebnis der Verarbeitung von `search` soll unter Verwendung von `sortCriterion` sortiert werden. Je nachdem, welchen Wert das Attribut hat, soll die passende Comparator-Implementierung verwendet werden. Ausnahme ist `DEFAULT`, hier soll keine Sortierung verwendet werden

Implementieren Sie anschließend die Schnittstelle `Iterable` in der Klasse `PlayList` unter Verwendung des erstellten Konstruktors.

Nach dieser Anpassung können Sie Ihre `PlayList` in for-each-Schleifen verwenden. Die Ausgabe ändert sich dabei, je nachdem, wie Sie die `PlayList` konfigurieren.

Ein Beispiel könnte wie folgt aussehen:

```
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"));
// Verändern Sie folgende Konfigurationen
pl.setSearch("Eisbach");
pl.setSortCriterion(SortCriterion.ALBUM);
// Beispiel für Iteration mit for-each
for(AudioFile file: pl) {
    System.out.println(file);
}
```

Zum Testen Ihrer Implementierung können Sie die bereitgestellten Tests in der **Datei TestTeilaufgabeF.java** nutzen.

Teilaufgabe g) Anpassung von PlayList

Abschließend ist die Abspielsteuerung der `PlayList` auf den Iterator umzustellen. Bisher wurde ein Zähler verwendet, um die aktuelle Abspielposition zu markieren. Zukünftig soll der Iterator genutzt werden, um die Abspielposition zu verwalten. Die Abspielfolge wird in der `PlayList` über `nextSong` und `currentAudioFile`-Aufrufe genutzt und gesteuert. Ziel ist es, dass diese Methoden den `ControllablePlayListIterator` entsprechend verwenden.

Orientieren Sie sich an folgenden Beispielen für die Verwendung, diese finden sich ebenfalls in der **TestTeilaufgabeG.java** (`testSnippet01` bis `testSnippet05`). Für jedes Beispiel gelten die folgenden Zeilen als gegeben:

```
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	**Ausgabe** (gekürzt)
testSnippet01: PlayList fängt von vorne an, wenn das Ende der Abspielfolge erreicht ist.	

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

testSnippet02: Wird nextSong aufgerufen, wird immer das nächste Lied geliefert, unabhängig von 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 fängt auch bei mehreren Liedern von vorne an, wenn das Ende erreicht ist.	

`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: Wird Suche oder Sortierung verwendet, hat dies Einfluß auf currentAudioFile und nextSong	

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

`pl4.nextSong();`
`System.out.println(pl4.currentAudioFile());`
`pl4.nextSong();`
`System.out.println(pl4.currentAudioFile());`	

testSnippet05: Wird Suche oder Sortierung verwendet, setzt dies die Abspielfolge zurück.	

`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, …`

Beachten Sie bei der Umsetzung:

- Entfernen sie das Attribut `current` und die entsprechenden Getter/Setter-Methoden.
- Modifizieren Sie `currentAudioFile()` und `nextSong()` so, dass die Steuerung der `PlayList` unter Verwendung des `ControllablePlayListIterator` abgebildet wird. Ergänzen Sie ggf. notwendige Attribute, um Iteratoren oder Referenzen auf das aktuelle Lied in PlayList zu speichern.
  - `currentAudioFile()` soll das aktuell abgespielte Lied liefern.
  - `nextSong()` soll die Abspielfolge auf das nächste Lied weiterbewegen.
- Sorgen Sie dafür, dass bei der Veränderung des Suchfilters bzw. des Sortierkriteriums sowie beim Hinzufügen bzw. Löschen eines Lieds oder dem Laden einer neuen M3U-Datei die Abspielfolge von vorne beginnt.

Ergänzen Sie abschließend eine Methode `jumpToAudioFile(AudioFile audioFile)` in `PlayList` und nutzen Sie die entsprechend hierfür umgesetzte Methode des Iterators. Achten Sie darauf, dass die `PlayList` anschließend korrekt ab dieser Position weiter abspielt (siehe `testJumpTo`), zum Beispiel wieder von vorne Beginnt, wenn das Ende der Liste erreicht ist:

```
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());
```

Die Ausgabe für dieses Beispiel wäre:

```
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
```

Zum Testen Ihrer Implementierung können Sie die bereitgestellten Tests in der **Datei TestTeilaufgabeG.java** nutzen.

Hinweise zur Abnahme Ihrer Implementierung der Vorführaufgabe 09

Laden Sie alle zum Aufgabenblatt gehörigen Abnahme-Tests herunter und speichern Sie diese im Source-Folder `cert` Ihres Projekts. Dann führen Sie die Abnahme-Tests in Eclipse aus.

Wenn alle Tests ohne Fehler durchlaufen, mailen Sie bitte folgende Java-Dateien als Dateianhang mit dem Betreff „VA09“ an den 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