Vorführaufgabe 9: Ausnahmebehandlung, Pakete und Sortierung

In der Vorführaufgabe 9 beschäftigen wir uns mit drei Themen. Wir verbessern die bisherige Ausnahmebehandlung, führen eine Paketstruktur ein und implementieren unterschiedliche Möglichkeiten zum Sortieren der Elemente unserer Abspiellisten.

Vorbereitungen

Bevor Sie mit der Implementierung der Teilaufgaben dieses Aufgabenblatts beginnen, müssen Sie wieder ein paar vorbereitende Schritte ausführen.

Empfehlung: Klonen Sie zunächst im Package-Explorer von Eclipse dasjenige Projektverzeichnis, in dem Sie die Aufgabe 8 implementiert haben, und geben Sie dem Klon einen neuen Namen. Wenn Sie zB. das Projekt der Aufgabe 8 im Projektverzeichnis `MediaPlayer_GdPII_VA08` gespeichert haben, legen Sie nun eine Kopie des Projekts an und benennen die Kopie mit `MediaPlayer_GdPII_VA09`[1].

Löschen Sie sodann alle Dateien im Unterverzeichnis `cert`. Das Unterverzeichnis beinhaltet aufgrund des Klonens noch die Server-Tests der Aufgabe 8. Laden Sie nun die zur Aufgabe 9 gehörigen Zip-Archive `cert.zip` und `build.zip` aus dem Intranet (Seite für VA09) und entpacken Sie beide im Wurzelverzeichnis Ihres Projektes. Das neue Archiv `cert.zip` enthält die Server-Tests für die VA09. Das Archiv `build.zip` enthält wieder die Ant-Build-Umgebung für die VA09. Die Dateien im Archiv `build.zip` werden im Wurzelverzeichnis ausgepackt und überschreiben die alten Dateien der Aufgabe 8.

Für diese Vorführaufgabe benötigen Sie auch wieder Audio-Dateien, die Sie im Intranet auf der Seite der Aufgabe 9 im Archiv `audiofiles.zip` finden. Die im Archiv enthaltenen Dateien entsprechen genau den Dateien aus der Aufgabe 8, die Sie durch das Klonen bereits kopiert haben. Sie müssen also das Archive `audiofiles.zip` der Aufgabe 9 nicht unbedingt neu laden und entpacken.

Unser Code nutzt natürlich nach wie vor Audio-Bibliotheken, die wir Ihnen im Intranet zum Download zur Verfügung stellen. Die Bibliotheken sind alle im Archiv `lib.zip` enthalten, welches Sie auf der Download-Seite der Aufgabe 9 finden.

Auch diese Dateien habe Sie bereits durch das Klonen kopiert. Je nach dem aber, wie Sie bei Aufgabe 8 die Bibliotheken eingebunden haben, haben Sie entweder relative Verweise auf die Bibliotheken erzeugt (via “Build-Path“ → “Configure Build Path“ → “Libraries“ → “Add Jars“) oder absolute Verweise (via “Build-Path“ → “Add external Archives“) generiert.
Falls Sie absolute Verweise erzeugt haben, müssen Sie diese nun ändern bzw. besser löschen und neu als relative Verweise einfügen, denn durch das Klonen werden die absoluten Verweise leider nicht angepasst. Falls Sie jedoch gleich relative Verweise erzeugt haben, müssen Sie nichts tun, denn die relativen Verweise zeigen bereits richtig in das neu geklonte Projekt.

1 In Eclipse im Projekt-Explorer im Wurzelverzeichnis des Projekts Rechtsklick → Copy

Ausnahmebehandlung (Klasse NotPlayableException)

Zuerst wollen wir uns der Ausnahmebehandlung annehmen.

Bisher wurden auftretende Ausnahmen gefangen, in einer `RuntimeException` gekapselt und weiter geworfen. Dies haben wir so gehandhabt, damit die aufrufenden Methoden sich nicht explizit um die Ausnahmen kümmern müssen. Ausnahmen vom Typ `RuntimeException` sind sogenannte nicht-kontrollierte Exceptions. Sie werden vom Java-Compiler nicht kontrolliert, d.h. der Compiler meldet keinen Fehler, wenn diese Ausnahmen nicht mittels `try/catch` behandelt oder mittels `throws` im Kopf der Methoden deklariert werden.
Dies soll nun geändert werden, in dem wir eine eigene Klasse `NotPlayableException` von `Exception` ableiten und somit als kontrollierte Ausnahme einführen. Natürlich müssen wir dann an den entsprechenden Stellen im Code eine explizite Deklaration bzw. Behandlung der kontrollierten Ausnahme `NotPlayableException` vorsehen.

Teilaufgabe a: Implementierung der Klasse `NotPlayableException`

Legen Sie eine Klasse `NotPlayableException` an. Diese Klasse soll von `Exception` abgeleitet sein und ein Attribut `pathname` vom Typ `String` haben.

Das Erzeugen von Ausnahmen kann in unterschiedlichen Kontexten erfolgen. Im ersten Fall erzeugt man selbst die Ausnahme und verpackt Fehlerdaten sowie eine Fehlernachricht im Ausnahmeobjekt. Für diesen Anwendungsfall braucht man einen geeigneten Konstruktor für die zu erzeugende Ausnahme.

In einem anderen Kontext fängt man eine Ausnahme eines anderen Typs und verpackt diese in einem selbst generierten Objekt (Umschreiben von Ausnahmen). Je nach dem, ob man selbst noch eine Nachricht und Daten hinzufügen will, braucht man auch für diesen Fall einen oder mehrere geeignete Konstruktoren für die Ausnahme.

Fügen Sie der Klasse `NotPlayableException` nun folgende drei Konstruktoren hinzu, die die unterschiedlichen Kontexte abdecken, in denen wir unsere neue Ausnahme einsetzen wollen:

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

Bemerkung: Die Klasse `Throwable` ist die Basisklasse von `Exception`. Die Klassen zur Behandlung von Ausnahmen der Java-Standard-Bibliothek nutzen diesen allgemeineren Typ in Ihren Schnittstellen. Da wir uns in dieses System integrieren wollen, müssen wir ebenfalls diesen allgemeineren Typ verwenden.

Alle Konstruktoren weisen in ihrem Rumpf den Parameter `pathname` dem gleichnamigen Attribut der Klasse `NotPlayableException` zu, rufen zuvor jedoch mittels `super` den jeweils passenden Konstruktor der Basisklasse `Exception` mit den verbleibenden Parametern auf.

- Überschreiben Sie nun noch in der Klasse `NotPlayableException` die Methode `toString()` so, dass diese die Zeichenkette
  `pathname + ": " + super.toString()`
  zurück gibt.

Ausnahmebehandlung (NotPlayableException)

Im nächsten Schritt soll nun der Umgang mit auftretenden Ausnahmen angepasst werden:

Teilaufgabe b: Deklarieren von NotPlayableException

Folgende Methoden sollen mit `throws NotPlayableException` das potentielle Werfen unserer Ausnahme deklarieren:

- die abstrakte Methode `play` der Klasse `AudioFile` und deren Implementierung in der Klasse `SampledFile`
- der Konstruktor der Klasse `AudioFile` sowie die Konstruktoren der abgeleiteten Klassen `SampledFile, WavFile` und `TaggedFile.` Natürlich macht das Werfen einer Exception nur in den Konstruktoren einen Sinn, die einen Pfadnamen als Argument haben. In den Konstruktoren ohne Parameter passiert nichts, was zur `NotPlayableException` führen könnte.
- die Methode `getInstance` der Klasse `AudioFileFactory`
- die Methode `readAndStoreTags` der Klasse `TaggedFile`
- die Methode `readAndSetDurationFromFile` der Klasse `WavFile`

Teilaufgabe c: Werfen von NotPlayableException

An den Stellen, an denen der Umgang mit Audio-Dateien einen Fehler verursacht hat, und wir bisher eine `RuntimeExceptions` geworfen haben, soll nun die neue Ausnahme `NotPlayableExceptions` erzeugt und geworfen werden. Das erste Argument für den Konstruktor der Ausnahme soll dabei stets der Pfadnamen der betroffenen Audio-Datei sein, während das zweite Argument eine prägnante Fehlermeldung enthalten soll.
Betroffen sind folgende Methoden:

- der Konstruktor mit einem Pfadnamen als Parameter der Klasse `AudioFile`
- die Methode `getInstance` der Klasse `AudioFileFactory`

Teilaufgabe d: Umwandeln von anderen Exceptions in NotPlayableException

Von Bibliotheksfunktionen geworfene `RuntimeExceptions` werden mittels `try/catch`-Block gefangen, in eine `NotPlayableException` gekapselt und weiter geworfen. Achten Sie darauf, dass der Konstruktor der `NotPlayableException` jeweils den Pfadnamen der betroffenen Audio-Datei, sowie die gefangene `RuntimeException` als Parameter bekommt. Bei Bedarf können Sie auch selbst noch eine erläuternde Fehlermeldung hinzufügen.
Betroffen sind die folgende Aufrufe von Bibliotheksfunktionen:

- Aufruf von `BasicPlayer.play` in der Methode `play` der Klasse `SampledFile`
- Aufruf von `TagReader.readTags` in der Klasse `TaggedFile`
- Aufruf von `WavParamReader.readParams` in der Klasse `WavFile`

Hinweis: Die Methoden in denen eine Umwandlung von Ausnahmen in `NotPlayableException` erfolgt müssen im allgemeinen dann die Ausnahme `NotPlayableException` mit `throws` deklarieren.

Ausnahmebehandlung (Fangen von NotPlayableException)

Jede Ausnahme erbt von der Klasse `Throwable` die Methode `printStackTrace`, die Infomationen zu einer Ausnahme auf die Standard-Fehlerausgabe (z.B. auf die Konsole) ausgibt. So können mit

```
try {
      // Code der Exception auslösen kann
} catch (Exception e) {
      e.printStackTrace();
}
```

diese Informationen zusammen mit einem Stack-Trace zu einer gefangenen Ausnahme sichtbar gemacht werden. Da die `Exception` dabei gefangen und nicht bis zur Laufzeitumgebung weiter gereicht wird, wird das Programm nicht abgebrochen.

Eine derartige Behandlung einer Ausnahme macht natürlich nur dann Sinn, wenn die reine Darstellung der Fehlerinformation in Form eines Stack-Trace als Fehlerbehandlung genügt. Falls man eine Ausnahme so behandelt bedeutet das, dass das Auftreten der Ausnahme zwar zur Kenntnis gebracht werden soll (Trace), jedoch keine weiteren Maßnahmen zur Behebung des Fehlers notwendig erscheinen.

Teilaufgabe e: Fangen von NotPlayableException

Ändern Sie nun die Methode `loadFromM3U` der Klasse `PlayList` so, dass eine von `AudioFileFactory.getInstance` ausgelöste `NotPlayableException` gefangen und wie oben beschrieben mittels `printStackTrace` geloggt wird, ohne dass das Programm dabei abbricht. Dies soll bewirken, dass fehlerhafte Audiodateien beim Einlesen übersprungen werden.

Ausnahmebehandlung (Unit-Tests)

Durch die Einführung der neuen Ausnahme `NotPlayableException` haben wir einige Änderungen am Code durchgeführt, die wir durch entsprechende Unit-Tests absichern sollten. Schreiben Sie nach folgendem Muster eigene Unit-Tests die prüfen, ob Ihre Klasse `AudioFileFactory` in den vorgesehenen Situation eine `NotPlayableException` erzeugt.

```
public class UTestAudioFileFactory {
    @Test
    public void test_getInstance_01() throws Exception {
        try {
            AudioFileFactory.getInstance("unknown.xxx");
            fail("Unknow suffix; expecting exception");
        } catch (NotPlayableException e) {
            // Expected
        }
    }

    @Test
    public void test_getInstance_02() throws Exception {
        try {
            AudioFileFactory.getInstance("nonexistent.mp3");
            fail("File is not readable; expecting exception");
        } catch (NotPlayableException e) {
            // Expected
        }
    }
```

Des weiteren sollten Sie durch folgenden Unit-Test prüfen, ob Ihre Implementierung der Methode `loadFromM3U` eine von der `AudioFileFactory` geworfene `NotPlayableException` sauber fängt und den Stack-Trace generiert, ohne dass die Ausnahme weiter propagiert wird und die Anwendung dadurch beendet wird.

```java
@Test //@Ignore
public void test_loadFromM3U_02() throws Exception {
    String m3u_pathname = "playlist.m3u";
    String mp3_pathname = "corrupt.mp3";

    // Create the M3U file with one entry for a non-existent mp3 file
    FileWriter writer = null;
    try {
        // Create a FileWriter
        writer = new FileWriter(m3u_pathname);
        writer.write(mp3_pathname + System.getProperty("line.separator"));
    } catch (IOException e) {
        throw new RuntimeException("Unable to store M3U file: "
                        + m3u_pathname, e);
    } finally {
        try {
            writer.close();
        } catch (IOException e) {
            // Just swallow
        }
    }
    // OK, the playlist for testing is in place
    PlayList pl = new PlayList();
    // The next statement will cause a stack trace to be printed onto
    // the console. However, execution is not terminated with an error
    // since we catch the exception in PlayList.loadFromM3U()
    // The test succeeds.
    pl.loadFromM3U(m3u_pathname);
    // cleanup
    new File(m3u_pathname).delete();
}
```

Paketstruktur (Paket studiplayer.audio)

Da das Projekt nun bereits eine gewisse Größe erreicht hat, ist es sinnvoll die Klassen in Paketen - meistens wird der englische Begriff Package verwendet - zu organisieren. Dies erhöht die Übersichtlichkeit und damit die Wartbarkeit (nähere Details hierzu in der Vorlesung Software Engineering). Auf Dateisystemebene wird dabei die Paketstruktur über einzelne Ordner abgebildet, in denen nach der Kompilierung der Byte-Code der einzelnen Klassen liegt[2].

Wenn Sie eine Klasse in ein anderes Paket verschieben, kann es nötig sein, `import`-Anweisungen in anderen Klassen einzufügen oder anzupassen. Einige IDEs, wie etwa Eclipse, sind in der Lage, diesen Vorgang zu automatisieren.

In Eclipse gehen Sie wie folgt vor, um bestehende Klassen in ein neues Paket zu verschieben:

- Rechtsklick auf den `src`-Ordner des Projekts im Package-Explorer und Menüpunkt *New → Package* auswählen.
- geben Sie den Namen des neuen Paketes ein und bestätigen sie mit den Button *Finish*
- markieren Sie im Package-Explorer die Klassen, die sie verschieben wollen (Mehrfachauswahl wie bekannt mit `<Shift>`-Klick oder `<Strg>`-Klick)
- Auf den markierten Klassen führen Sie einen Rechtsklick aus und wählen den Menüpunkt

2 Eclipse organisiert zusätzlich auch den Quellcode der Klassen in einer den Paketen entsprechenden Ordnerstruktur. Dies wird von der Java-Sprachdefinition nicht gefordert, da der Class-Loader nur mit Byte-Code arbeitet, führt aber zu mehr Übersichtlichkeit in der Organisation des Quellcodes!

Refactor → Move

- setzen Sie im erscheinenden Fenster *Move* den Haken bei *Update references to the moved element(s)*, und bestätigen Sie mit Klick auf den Button *OK*

(Für das Vorgehen in anderen IDEs sehen Sie bitte in deren Applikationsdokumentation nach.)

Teilaufgabe f: (Erzeugen und Befüllen des Pakets studiplayer.audio)

- Erzeugen Sie, wie eben beschrieben, das neue Paket `studiplayer.audio`
- Verschieben Sie alle bisher im Ordner `src` erstellten Klassen in das Paket `studiplayer.audio`

Sofern nicht anders angegeben, sind alle in den weiteren Aufgaben zu implementierenden Klassen in diesem Paket abzulegen.

Sortierung (Enum SortCriterion)

Als letztes wollen wir in dieser Vorführaufgabe die Sortierung der Lieder in einer `PlayList` nach verschiedenen Kriterien ermöglichen. Wir werden hierfür das `Comparator`[3]-Interface benutzen[4], wobei die eigentliche Sortierung von der Methode `sort()` der Klasse `Collections` der Java-API erledigt wird. Wir müssen lediglich geeignete Methoden implementieren, die zwei Einträge einer `PlayList` vergleichen können. Den Rest erledigt die Methode `Collections.sort()`.

Um uns die Unterscheidung der einzelnen Sortierkriterien zu vereinfachen, werden wir eine `Enum`-Klasse, also eine Aufzählung, mit den vier möglichen Sortierkriterien einführen. Der Aufzählungstyp `Enum` ist Ihnen bereits in C begegnet. Bei einer Aufzählung können Bezeichner festgelegt werden, die im Hintergrund durch eindeutige Werte ersetzt werden. Die Zuweisung der Werte zu den einzelnen Bezeichnern erfolgt dabei durch die Laufzeitumgebung (bei C durch den Compiler).

Teilaufgabe g: (Enum Klasse SortCriterion)

- Legen sie die Aufzählungsklasse
  `public enum SortCriterion`
  mit den vier Konstanten
  `AUTHOR, TITLE, ALBUM` und `DURATION`
  an.

Sortierung (Vorbetrachtung: totale Quasi-Ordnungen)

Nun muss für jedes Sortierkriterium eine entsprechende Klasse geschrieben werden, die das `Comparator`-Interface durch eine eigene Implementierung der Methode

```
public int compare(<E> obj1, <E> obj2)
```

erfüllt. Diese Methode gibt für zwei übergebene Objekte einen ganzzahligen Wert zurück, der den *logischen Abstand* zwischen den Objekten angibt. Das `Comparator`-Interface ist parametrisierbar, um Typensicherheit zu gewährleisten.

Die Implementierung der Methode `compare(obj1, obj2)` für den logische Abstand zwischen zwei Objekten soll indirekt eine Ordnungsrelation auf den zu vergleichenden Objekten definieren. Falls `obj1` kleiner als `obj2` anzusehen ist, muss die Methode eine negative Integer-Zahl zurück liefern. Sollen `obj1` und `ob2` gleich sein, muss `compare(obj1,obj2)` eine `0` liefern. Wenn aber `obj1` nach der zu definierenden Ordnung größer als `obj2` sein soll, muss `compare(obj1,obj2)` eine positive Integer-Zahl zurück liefern.

---

3 `import java.util.Comparator;`

4 Nicht zu verwechseln mit dem Interface `Comparable`!

Die Definition der Methode `compare(obj1,obj2)` darf allerdings nicht leichtfertig geschehen. Eine willkürliche Abbildung auf Integer-Zahlen definiert lediglich eine Relation zwischen den Objekten. Für den erfolgreichen Einsatz der Vergleichs-Methode `compare()` im Rahmen einer Sortierung muss es sich bei der durch `compare()` indirekt definierten Relation um eine totale Quasi-Ordnung[5] handeln. Andernfalls ist die korrekte Durchführung des Sortieralgorithmus nicht garantiert!

Neben anderen Eigenschaften der Ordnungstheorie müssen alle Implementierungen der vom Interface `Comparator` vorgeschriebenen Methode `compare()` insbesondere folgende Eigenschaften aufweisen:

1. aus `compare(o1,o2) > 0` folgt stets `compare(o2,o1) < 0`
2. aus `compare(o1,o2) < 0` folgt stets `compare(o2,o1) > 0`
3. aus `compare(o1,o2) = 0` folgt stets `compare(o2,o1) = 0`

Wird eine dieser Eigenschaften nicht erfüllt, so stellt die daraus abgeleitete Relation keine totale Quasi-Ordnung dar.

Sortierung (Mehrfache Implementierung des Interface Comparator)

Für die vier Sortierkriterien `AUTHOR, TITLE, ALBUM` und `DURATION` werden im folgenden jeweils die entsprechenden Implementierungen des Interface `Comparator` ausgeführt.

Da wir beim Sortieren nach den Kriterien `AUTHOR, TITLE` und `ALBUM` nur `String`-Attribute miteinander vergleichen, können wir die Methode `compareTo` der Klasse `String` für die einzelnen Vergleiche nutzen. Mit `myString1.compareTo(myString2)` werden mit dieser Methode zwei `Strings` miteinander verglichen, wobei der zurückgegebene ganzzahlige Wert dem von uns benötigten logischen Abstand entspricht.

Teilaufgabe h: AuthorComparator

- Implementieren Sie nun eine Klasse `AuthorComparator,` die das Interface `Comparator<E>` implementiert. Instantiieren Sie den Typ-Parameter `E` des Interface dabei so, dass nur Objekte vom Typ `AudioFile` verglichen werden können.
- Setzen Sie die vom `Comparator`-Interface vorgeschriebene Methode
  `public int compare(AudioFile af1, AudioFile af2)`
  so um, dass im Fall von zwei definierten Autoren-Strings (nicht Null-Pointer) der Rückgabewert dem Ergebnis des Vergleichs von `af1.getAuthor()` und `af2.getAuthor()` mittels der Methode `compareTo()` der Klasse `String` entspricht.

  Werfen Sie eine `NullPointerException` mit einer aussagekräftigen Fehlermeldung, falls eines der beiden an die Methode `compare()` übergebenen Objekte vom Type `AudioFile` ein `Null`-Pointer ist. Sorgen Sie zudem für sinnvolle Ergebnisse auch für den Fall, dass einer oder beide Autoren-Strings Null-Pointer sind.

Teilaufgabe i: TitleComparator

- Implementieren Sie nun analog zu der Klasse `AuthorComparator` die Klasse `TitleComparator`. Stützen Sie sich bei der Implementierung der Methode `compare()` auf die Getter-Methode `getTitle()`.

5 Eine totale Quasi-Ordnung ist reflexiv, transitiv und es lassen sich alle Elemente vergleichen (total). Die Eigenschaft der Antisymmetrie muss jedoch nicht gelten! Elemente können gleich sein, müssen dabei aber nicht identisch sein. Durch eine Quotientenbildung lässt sich daraus eine totale Ordnung erzielen. Siehe auch http://de.wikipedia.org/wiki/Quasiordnung

Für die Sortierkriterien `ALBUM` und `DURATION` ergibt sich eine Besonderheit:
Die für den Vergleich benötigten Attribute sind nicht in der Klasse `AudioFile` definiert[6]. Daher kann nicht für alle zu vergleichenden Elemente (vom Typ AudioFile) auf die benötigten Attribute zugegriffen werden. Aus diesem Grund ist vor dem eigentlichen Vergleich eine Überprüfung nötig, ob es sich bei den beiden zu vergleichenden Objekten um Objekte der jeweils benötigten (Kind-)Klasse handelt.

Bei der nun nachfolgenden Festlegung des Verhaltens der Methode `compare(o2,o1)` müssen wir stets daran denken, dass wir die Anforderungen einer totalen Quasi-Ordnung erfüllen müssen!

Sollte **nur** das erste übergebene Objekt nicht vom benötigten Typ sein, geben wir `-1` zurück. Sollte **nur** das zweite Objekt nicht vom benötigten Typ sein, geben wir `1` zurück. Objekte vom falschen Typ sind also immer kleiner als die mit dem richtigen Typ. Sind aber beide Objekte vom benötigten Typ, wird ein echter Vergleich durchgeführt. Denken Sie auch an den noch verbleibenden Fall!

Da in der Elternklasse das benötigte Attribut jedoch nicht bekannt ist, muss ein Cast, also ein explizites Umwandeln in den entsprechenden Typ, ausgeführt werden. So kann etwa mit `TaggedFile tf = (TaggedFile) af1` aus einem Objekt der Klasse `AudioFile` eine Objekt der Klasse `TaggedFile` gemacht werden und dann mittels `tf.getAlbum()` auf das zum Vergleich benötigte Attribut zugegriffen werden.

Teilaufgabe j: AlbumComparator

- Implementieren sie nun die Klasse `AlbumComparator` für den Vergleich der `AudioFiles` gemäß der evtl. vorliegenden Information über das Album. Führen Sie jedoch zusätzlich die beschriebene Überprüfung mittels `instanceof` und evtl. nachfolgende Casts durch, bevor Sie den eigentlichen Vergleich anhand eines gespeicherten Albums machen. Natürlich müssen Sie auch hier wieder Null-Pointer für Objekte der Klassen `AudioFile` und `String` speziell behandeln.

Teilaufgabe k: DurationComparator

- Implementieren Sie nun analog zur Klasse `AlbumComparator` die Klasse `DurationComparator` für den Vergleich der Audio-Dateien gemäß der gespeicherten Information über die Abspieldauer. Je nach Implementierung müssen Sie dabei Type-Casts durchführen oder nicht. (In welcher Klasse haben Sie das Attribut für die Abspieldauer eingehängt?). Finden Sie eine geeignete Möglichkeit zur Berechnung des logischen Abstands auf Basis der Abspieldauer. Auch hier müssen wieder Null-Pointer für Objekte der Klasse `AudioFile` speziell behandelt werden.

Hinweis: der logische Abstand muss nicht unbedingt durch Berechnung einer Differenz gefunden werden. Sie haben hier diverse Optionen.

Sortierung (Sortieren der Play-Liste: PlayList.sort())

Da wir für jedes Sortierkriterium eine entsprechende Klasse haben, die das `Comparator`-Interface implementiert, können wir nun die statische Methode `sort` der Klasse `Collections` nutzen, um unsere Abspielliste zu sortieren. Die Methode `sort` erwartet dabei als erstes Argument eine Implementierung des Interface `List` (in unserem Fall also eine Instanz der Klasse `PlayList`), und als zweites Argument ein Objekt der entsprechenden `Comparator`-Klasse.

So kann beispielsweise mit

```
Collections.sort(myPlayList, new TitleComparator());
```

die `PlayList` mit dem Namen `myPlayList` nach dem Kriterium `TITEL` sortiert werden.

6 Beim für die Abspieldauer zuständigen Attribut kann man darüber diskutieren, ob es besser in `AudioFile` oder in `SampleFile` aufgehoben ist. Aber das Attribut, in dem wir den Namen des Albums speichern, macht definitiv nur in der Klasse `TaggedFile` Sinn

Teilaufgabe I: Implementierung der Methode PlayList.sort()

- Implementieren Sie in der Klasse `PlayList` eine Methode
  `public void sort(SortCriterion order)`
  welche die Abspielliste (`this`) in Abhängigkeit vom übergebenen Sortierkriterium mittels der Methode `Collections.sort` passend sortiert.

Sortierung (Unit-Tests für PlayList.sort())

Überprüfen Sie, ob das Sortieren nach den einzelnen Kriterien bei einer Abspielliste mit gemischten Audioformaten wie vorgesehen funktioniert. Verwenden Sie hierzu Unit-Tests nach folgendem Muster.

```
@Test
public void test_sort_byTitle_01() throws Exception {
  PlayList pl1 = new PlayList();
  // Populate the playlist
  pl1.add(new TaggedFile("audiofiles/Eisbach Deep Snow.ogg"));
  pl1.add(new WavFile("audiofiles/wellenmeister - tranquility.wav"));
  pl1.add(new TaggedFile("audiofiles/wellenmeister_awakening.ogg"));
  pl1.add(new TaggedFile("audiofiles/tanom p2 journey.mp3"));
  pl1.add(new TaggedFile("audiofiles/Rock 812.mp3"));
  // Sort the playlist by title
  pl1.sort(SortCriterion.TITLE);
  // Store the toString() strings of the sorted play lists into an array
  // and compare the arrays
  // Note: "T*" < "t*"
  String exp[] = new String[] {
   "Eisbach - Deep Snow - The Sea, the Sky - 03:18",
   "Eisbach - Rock 812 - The Sea, the Sky - 05:31",
   "Wellenmeister - TANOM Part I: Awakening - TheAbsoluteNecessityOfMeaning - 05:55",
   "Wellenmeister - TANOM Part II: Journey - TheAbsoluteNecessityOfMeaning - 02:52",
   "wellenmeister - tranquility - 02:21" };
  String sorted[] = new String[5];
  int i = 0;
  for (AudioFile af : pl1) {
      sorted[i] = af.toString();
      i++;
  }
  assertArrayEquals("Wrong sorting by title", exp, sorted);
}
```

```java
@Test
public void test_sort_byDuration_01() throws Exception {
 PlayList pl1 = new PlayList();
 // Populate the playlist
 pl1.add(new TaggedFile("audiofiles/Eisbach Deep Snow.ogg"));
 pl1.add(new WavFile("audiofiles/wellenmeister - tranquility.wav"));
 pl1.add(new TaggedFile("audiofiles/wellenmeister_awakening.ogg"));
 pl1.add(new TaggedFile("audiofiles/tanom p2 journey.mp3"));
 pl1.add(new TaggedFile("audiofiles/Rock 812.mp3"));
 // Sort the playlist by duration
 pl1.sort(SortCriterion.DURATION);
 // Store the toString() strings of the sorted play lists into an array
 // and compare the arrays
  String exp[] = new String[] {
 "wellenmeister - tranquility - 02:21",
 "Wellenmeister - TANOM Part II: Journey - TheAbsoluteNecessityOfMeaning - 02:52",
 "Eisbach - Deep Snow - The Sea, the Sky - 03:18",
 "Eisbach - Rock 812 - The Sea, the Sky - 05:31",
 "Wellenmeister - TANOM Part I: Awakening - TheAbsoluteNecessityOfMeaning - 05:55" };
 String sorted[] = new String[5];
 int i = 0;
 for (AudioFile af : pl1) {
     sorted[i] = af.toString();
     i++;
 }
 assertArrayEquals("Wrong sorting by duration", exp, sorted);
}
```

Hinweise zur Abnahme Ihrer Implementierung der Vorführaufgabe 09

Im Zuge der Bearbeitung aller Teilaufgaben dieses Blattes haben Sie im Unterverzeichnis `src/studiplayer/audio` die Klassen

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

erzeugt. Diese Java-Dateien müssen Sie im Anhang einer E-Mail an den APA-Server schicken (Anhang als einzelne Dateien oder als Archiv). Der richtige Betreff der E-Mail lautet: `VA09`
Nähere Details zum Abnahme-Prozess finden Sie im Intranet.

Im Unterverzeichnis `tests` haben Sie parallel dazu eine oder mehrere Testklassen mit Unit-Tests erzeugt. Diese müssen und sollen Sie nicht einschicken!

Damit Sie bei der Abnahme durch den Service im Netz nicht zu viel Zeit durch etwaige Fehlversuche vergeuden, empfiehlt es sich, die Abnahme-Tests vorher aus dem Intranet herunterzuladen und lokal auf Ihrem Rechner auszuführen.
Erst wenn die Tests bei Ihnen lokal erfolgreich ausgeführt werden, lohnt es sich, die Tests vom Abnahme-Service prüfen zu lassen.

Hinweise zum Laden der Abnahme-Tests:
Laden Sie alle zum jeweiligen Aufgabenblatt gehörigen Abnahme-Tests herunter und speichern Sie diese im Unterverzeichnis `cert` Ihres Projekts (siehe Abschnitt Vorbereitung am Anfang dieses Aufgabenblatts). Dann führen Sie die Abnahme-Tests als Junit3-Tests in Eclipse aus.