Technische Hochschule Ingolstadt
Fakultät für Elektrotechnik und Informatik

Vorführaufgabe 6: Eine einfache Klasse

Wie schon im ersten Semester wird auch der zweite Teil ihrer grundlegenden Programmier-ausbildung durch ein Praktikum begleitet. Die fünf zu erstellenden Vorführaufgaben bauen dabei aufeinander auf und werden Sie schrittweise durch die Implementierung eines einfachen Audioplayers mit folgender graphischen Oberfläche begleiten.

Jede Vorführaufgabe wird durch ein Aufgabenblatt beschrieben, das Sie zusammen mit weiterem Material aus dem Intranet herunterladen können.

Die Aufgabenblätter enthalten neben erklärendem Text auch immer eine Reihe von Teilaufgaben, die Sie im Rahmen der Implementierung des Praktikums erfüllen müssen.
Die Abnahme (Testate) erfolgt bei diesem Praktikum über den speziellen APA-Service im Intranet.

Die Beschreibung der von Ihnen zu erbringenden Aufgaben erfolgt, wie heute in der Praxis üblich, durch eine Mischung aus mehreren Beschreibungsformen. Wir verwenden

- textuelle Anforderungsbeschreibungen
- graphische Beschreibungen anhand von UML-Diagrammen
- tabellarische Darstellung von Argument/Result Kombinationen (IO-Verhalten) und deren Implementierung in Form von Unit-Tests

Die Form der textuellen Anforderungsbeschreibung kennen Sie schon aus dem Praktikum des ersten Semesters. Die beiden anderen lernen Sie nun im Rahmen dieses Praktikums kennen.

Graphische Beschreibungen, wie UML-Diagramme, sind ein in der Praxis häufig genutztes Mittel, um Anforderungen auszudrücken und werden insbesondere bei Diskussionen zwischen Gesprächspartnern mit unterschiedlichem technischen und fachlichen Hintergrund verwendet. Wir werden uns bei diesem Praktikum auf einfache UML-Klassendiagramme beschränken.

Unit-Tests spielen in den letzten Jahren eine immer stärker werdende Rolle bei der Erstellung und der Wartung von Software-Projekten, und daher wollen wie Sie möglichst früh mit dieser Technik vertraut machen.

Ein weiterer Grund für den Einsatz von Unit-Tests im Praktikum besteht aber auch darin, dass wir über Unit-Tests die von Ihnen zu lösenden Aufgaben nebst der resultierenden Funktionalität genau beschreiben können, ohne Sie in Ihren Lösungsansätzen zu sehr einschränken zu müssen.

Die Unit-Tests werden in wenigen Fällen auf den Aufgabenblättern ausgedruckt, was bedeutet, dass Sie die Tests dann abtippen müssen. Meistens werden Ihnen die Unit-Tests im Intranet zum

Download bereitgestellt oder Sie werden aufgefordert, die Unit-Tests selbst zu formulieren und natürlich auch zu programmieren.

Vorbereitung: Aufsetzen der Entwicklungsumgebung

Bevor wir mit den Aufgaben dieses Blatts beginnen, müssen Sie zunächst Ihre Entwicklungsumgebung aufsetzen. Sie sind völlig frei in Ihrer Wahl hinsichtlich eingesetzter IDE (Integrated Development Environment = Entwicklungsumgebung) und Plattform (Windows, MAC, Linux, …). Die für die Abnahme (Testate) zu erbringenden Leistungen sind davon unabhängig.

Auf den Aufgabenblättern werden wir jedoch bisweilen Hinweise zur Benutzung der wohl meistens eingesetzten IDE Eclipse bringen und mögliche Probleme bei Verwendung der Plattformen Windows oder Linux ansprechen. Wir empfehlen Ihnen die Verwendung der IDE Eclipse, da wir Ihnen nur für die IDE Eclipse Support geben können.

Legen Sie nun in Ihrer IDE ein neues Projekt an, beispielsweise mit Namen
MediaPlayer_GdPII_VA06

Falls Sie keine IDE verwenden, was durchaus auch möglich ist, so legen Sie bitte aus Gründen der besseren Strukturierung zumindest ein neues Unterverzeichnis an.

Sorgen Sie durch geeignete Einstellungen in Ihrer IDE dafür, dass:

- ein Unterverzeichnis `src` zur Speicherung der Java-Quellen angelegt wird[1]
- ein Unterverzeichnis `tests` zur Speicherung der JUnit-Tests angelegt wird
- ein Unterverzeichnis `cert` zur Speicherung der Abnahme-Tests angelegt wird
- alle kompilierten Java-Class-Dateien im Unterverzeichniss `classes` abgelegt werden[2]
- das Jar-Archiv für Junit-Tests der Version 3 eingebunden wird (für Abnahme-Tests)[3]
- das Jar-Archiv für Junit-Tests der Version 4 eingebunden wird (für Ihre Unit-Tests)

Teilaufgaben der Vorführaufgabe 6

Die offensichtliche Aufgabe eines Audioplayers ist die Wiedergabe von Audiodaten. Diese können auf unterschiedliche Weise in einen Audioplayer geladen werden. Neben der heute schon häufig genutzten Möglichkeit, einen Media-Stream eines im Internet verfügbaren Media-Servers zu öffnen, gibt es noch die herkömmliche Art, Audiodaten zu laden, nämlich das Lesen einer Audiodatei.

Der im Rahmen des Praktikums entstehende einfache Audioplayer kann Audiodaten nur in Form von Audiodateien (Audiofiles) verarbeiten, die über ein Dateisystem zugreifbar sind. Daher beginnen wir die Implementierung des Audioplayers mit dem Erstellen der Klasse `Audiofile`.

Jedes Objekt der Klasse `AudioFile` bezieht sich auf eine einzelne Audiodatei. Der sogenannte *Pfadname* dieser Datei besteht aus dem optionalen Pfad und dem Dateinamen. Wird ein Pfad angegeben, so kann dies eine absolute oder eine relative Angabe sein. Es kann also der vollständige Pfad, ein relativer Pfad oder auch gar kein Pfad mit angegeben sein.

Beispiele für Unix/Linux Systeme:

`/home/meier/Musik/Falco_-_Rock_Me_Amadeus.mp3`	(absoluter Pfad, beginnt mit `/`)
`../musik/Falco_-_Rock_Me_Amadeus.mp3`	(relativer Pfad)
`Falco_-_Rock_Me_Amadeus.mp3`	(ohne Pfad)

1 Bei Eclipse: Rechtsklick im Package-Explorer → New → Source Folder

2 Bei Eclipse: Window → Preferences → Java → Build Path: Output folder name

3 Bei Eclipse: Rechtsklick im Package-Explorer → Build Path → Add Libraries→ JUnit

Beispiele für Windows Systeme:

`D:\Daten\Musik\Falco_–_Rock_Me_Amadeus.mp`	(absoluter Pfad, beginnt mit Laufwerksangabe)
`..\Musik\Falco_–_Rock_Me_Amadeus.mp3`	(relativer Pfad)
`Falco_–_Rock_Me_Amadeus.mp3`	(ohne Pfad)

(Das Zeichen '_' steht hier, ebenso wie im Rest der Angabe, jeweils für ein Leerzeichen.)

Bitte beachten Sie, dass der Pfadseparator, also das Trennzeichen für die einzelnen Verzeichnisse, je nach Betriebssystem, unterschiedlich ist, etwa ein '/' unter Unix /MacOS und ein '\' unter Windows.

Das statische Attribut (also das „objektorientierte Gegenstück" einer Konstante)

```
java.io.File.separatorChar
```

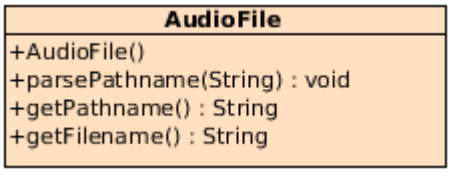
beinhaltet aus diesem Grund den Pfadseparator betriebssystemabhängig. Dieser kann mit

```
System.out.println(java.io.File.separatorChar);
```

immer passend zur Plattform ausgegeben werden.

Nach diesen umfangreichen einleitenden Bemerkungen wollen wir nun endlich mit der Beschreibung der Anforderungen beginnen.

Wie angekündigt, setzen wir zur Beschreibung der Anforderungen UML-Klassendiagramme ein.



Das Klassendiagramm rechts beschreibt in Ausschnitten die Klasse `AudioFile`. Unterhalb der Titelleiste, die den Namen der Klasse enthält, sind in einer eigenen Abteilung die Methoden der Klasse aufgeführt. Neben dem an erster Stelle stehenden Konstruktor `AudioFile()`, der keine Argumente hat, sind noch drei weitere Methoden aufgelistet.
Das Zeichen '+' vor dem Namen einer Methode bedeutet in der UML-Notation, dass die Methode öffentliche Sichtbarkeit (public) hat. Im Gegensatz zu Java, wo der Typ des Rückgabewertes einer Methode vor dem Methodennamen steht, wird dieser in der UML-Notation hinter dem Methodennamen angegeben.

Statt wie in Java:

```
public String getPathname()
```

steht also im UML-Klassendiagramm

```
+getPathname(): String
```

Bitte beachten Sie, dass die Klasse `AudioFile` neben den im Diagramm angegeben Methoden durchaus noch weitere Methoden besitzen kann. Diese stehen nur derzeit nicht im Fokus unserer Aufmerksamkeit und sind daher im Diagramm nicht aufgeführt. Diagramme stellen also keine vollständige Spezifikation dar.

Teilaufgabe (a)

Legen Sie im Verzeichnis `src` eine Klasse `AudioFile` an, die die vier im Klassendiagramm gezeigten Methoden zur Verfügung stellt. Lassen Sie dabei die Rümpfe der Methoden zunächst leer bzw. sehen Sie im Fall eines geforderten Rückgabewerts eine beliebige triviale Implementierung vor (trivial heißt in diesem Fall z.B. immer, einen leeren String zurück zu geben).

Teilaufgabe (b)

In dieser Aufgabe implementieren wir den Code der im Klassendiagramm gezeigten Methoden. Zunächst spezifizieren wir das Verhalten der Methoden in textueller Form. Danach konkretisieren wir die Beschreibung durch mehrere Beispiele und Unit-Tests.

Wie Ihnen sicher auffallen wird, gehen wir dabei nicht näher auf mögliche Attribute der Klasse `AudioFile` ein. Dies ist Absicht, da wir Sie nicht in Ihrer Kreativität einschränken möchten. Ihre Aufgabe ist unter anderem, die Klasse mit geeigneten Attributen so auszustatten, dass die

Methoden das geforderte Verhalten erbringen können.

Hier nun die Anforderungen an die öffentlichen (public) Methoden:

`parsePathname(String): void`

- Der Parameter der Methode vom Typ `String` beinhaltet den sogenannten Pfadnamen der Audiodatei, also den wie oben schon beschriebenen optionalen Pfad und den Dateinamen. Die Methode soll das Argument analysieren und zur späteren Verwendung durch andere Methoden (evtl. schon normalisiert) in geeigneten Attributen der Klasse abspeichern.

`getPathname(): String`

- Liefert den kompletten an `parsePathname(String)` übergebenen Pfadnamen in normalisierter Form zurück. Normalisierung bedeutet hierbei zum einen, dass mehrfach hintereinander auftretende Pfadseparatoren nur einfach ausgegeben werden (also zB. '///' als '/'). Zum anderen sollen Laufwerksbuchstaben plattformabhängig behandelt werden[4]. Bitte beachten Sie, dass Sie den unter Windows gebräuchlichen Pfadseparator '\' im Java-Code als „\\“ angeben müssen, da er andernfalls als Escape-Character interpretiert wird. Nutzen Sie für die Normalisierung zudem den von der Plattform abhängigen Pfadseparator `java.io.File.separatorChar.`

`getFilename(): String`

- Liefert den Dateinamen zurück, d.h. den an `parsePathname(String)` übergebenen Pfadnamen ohne Pfad zurück. Falls kein Dateiname enthalten war, soll ein leerer String zurückgegeben werden (in diesem Fall hat der Pfadname mit einem Separator geendet).

Die obigen Angaben sind nun mehr oder weniger genau und lassen noch Spielraum für Interpretationen frei. Das ist bei textuellen Beschreibungen zwangsläufig der Fall. Aus diesem Grund konkretisieren wir die Anforderungen durch Beispiele in folgender Tabelle (IO-Verhalten):

Argument für parsePathname()	**Resultat getPathname()**	**Resultat getFilename()**
Leerer String	Leerer String	Leerer String
Nur Leerzeichen oder Tabs	Unverändertes Argument von `parsePathname()`	Unverändertes Argument von `parsePathname()`
file.mp3	file.mp3	file.mp3
/my-tmp/file.mp3	Linux: /my-tmp/file.mp3 Windows: \my-tmp\file.mp3	file.mp3
//my-tmp////part1//file.mp3/	Linux: /my-tmp/part1/file.mp3/ Windows: \my-tmp\part1\file.mp3\	Leerer String
`d:\\\\part1///file.mp3`	Linux: **/d/**part1/file.mp3 Windows: **d:**\part1\file.mp3	file.mp3

Selbst durch Angabe der obigen Beispiele verbleibt möglicherweise immer noch ein Rest von Unklarheit, was die zu erbringende Implementierung betrifft. Will man diesen letzten Rest auch noch eliminieren, so muss man zu formalen Beschreibungsmitteln greifen. Im vorliegenden Fall bietet sich die Verwendung der EBNF (Erweiterte Backus-Naur Form) an.

Mit Hilfe der EBNF kann exakt definiert werden, was unter den oben informell eingeführten Begriffen Pfadname, Pfad und Dateiname zu verstehen ist. Eine entsprechende Definition finden Sie in der Datei `PathnameEBNF.pdf` im Intranet auf der WEB-Seite, von der Sie auch dieses

4 Siehe letztes Beispiel der Tabelle

Aufgabenblatt bezogen haben.

Das Studium dieser formalen EBNF-Spezifikation ist zur Lösung der vorliegenden Praktikumsaufgabe nicht zwingend notwendig. Die Spezifikation demonstriert jedoch den praktischen Einsatz der EBNF und bietet Ihnen zudem die Möglichkeit, Ihre Kenntnisse über die EBNF aufzufrischen.

Hinweise zur Implementierung:
1) Bei der in `parsePathname()` anfallenden String-Analyse sollten Sie zunächst das Problem in Teilprobleme zerlegen und dann schrittweise lösen. Ein mögliche Zerlegung ist *Behandlung von Laufwerksbuchstaben, Normalisierung der Separatoren* und zuletzt *Zerteilung in Pfad und Dateiname*. Die Verwendung von regulären Ausdrücken führt i.a. zu sehr kompaktem Code, ist aber schwerer zu programmieren als eine explizite Schleife, die die Zeichen einzeln ersetzt.

2) Die statische Methode `Character.isLetter` prüft, ob das übergebene Zeichen ein Buchstabe ist. Des weiteren könnte der Einsatz folgender Methoden der Klasse `String` hilfreich sein:

```
indexOf, lastIndexOf, valueOf, charAt, substring, trim
```

Bevor Sie sich nun daran machen, die obigen Beispiele zu Testzwecken mühsam in einer main-Methode der Klasse AudioFile zu programmieren, legen Sie bitte stattdessen im Verzeichnis `tests` eine JUnit-Testklasse `UTestAudioFile` mit dem folgenden Inhalt an[5].

```
1  import org.junit.Test;
2  import static org.junit.Assert.assertEquals;
3  public class UTestAudioFile {
4      @Test
5      public void test_parsePathname_03() throws Exception {
6          AudioFile af = new AudioFile();
7          af.parsePathname("/my-tmp/file.mp3");
8          char sepchar = java.io.File.separatorChar;
9          // On Unix    we expect "/my-tmp/file.mp3"
10         // On Windows we expect "\my-tmp\file.mp3"
11         assertEquals("Pathname stored incorrectly",
12                 sepchar + "my-tmp" + sepchar + "file.mp3" ,
13                 af.getPathname());
14         assertEquals("Returned filename is incorrect",
15                 "file.mp3",
16                 af.getFilename());
17     }
18 }
```

Die einzige in der Testklasse enthaltene Testmethode `test_parsePathname_03()` implementiert den Test aus der vierten Zeile in obiger Tabelle.

Um Ihnen den Einstieg in die Benutzung des Junit-Frameworks zu erleichtern, wollen wir im folgenden den Code dieser Testmethode besprechen.

Zeilen 1,2:
Die benötigten Klassen des Junit-Test-Frameworks werden eingebunden.
Wir binden zunächst nur die Methode `assertEquals()` ein, die die Gleichheit zweier Werte prüft. Das Junit-Framework bietet noch eine Reihe weiterer assert-Methoden, von denen Sie später noch einige kennen lernen werden.

Zeile 3:
Hier beginnt die Testklasse. Es ist üblich, einen Namen zu wählen, der sowohl Bezug auf die zu testende Klasse nimmt (hier `AudioFile`), als auch die Klasse bezüglich einer beliebigen Konvention als Testklasse ausweist (hier Präfix `UTest`).

5 Im Package-Explorer von Eclipse: Rechtsklick → New → Junit Test Case

Zeile 4:
Markiert durch die Annotation @TEST die nachfolgende Methode als Testfall (neu in JUnit4).

Zeile 5:
Hier wird die Testmethode mit Namen `test_parsePathname_03` definiert.
Der Name der Methode wird meist so gewählt, dass er Bezug auf die zu testende(n) Methode(n) nimmt. Wir verwenden hier immer den Präfix `test_`, dann den originalen Namen der zu testenden Methode (CamelCase) und dann, nach einem `'_'`, eine laufende Nummer.
Durch diese Konvention wird der originale Name der Methode nicht verändert, was bei der sonst üblichen reinen CamelCase Schreibweise der Fall wäre (`testParsePathname03`).

Der Zusatz `throws exception` ist optional.
Falls Ihnen *Exceptions* noch nichts sagen, nehmen Sie es bitte vorerst einfach als gegeben hin.

Zeile 6:
Nutzt den parameterlosen Konstruktor `AudioFile()`, um ein leeres Testobjekt anzulegen.
Beim Einsatz von Unit-Tests ist es üblich, neben anderen Konstruktoren immer auch einen Konstruktor ohne Parameter zur Verfügung zu stellen, der das Objekt evtl. mit Default-Werten initialisiert. Das Befüllen von Attributen mit den *richtigen Werten* erfolgt dann durch separate öffentliche Methoden, hier etwa durch `parsePathname()`.
Dieses Vorgehen erhöht die Testbarkeit der Klasse.

Zeile 7:
Ruft die zu testende Methode `parsePathname()` des Objektes `af` mit einem Testbeispiel auf.

Zeile 8:
Definiert eine Abkürzung für den plattformabhängigen Pfadseparator.

Zeilen 11-13:
Hier wird eine der zentralen Methoden des Test-Frameworks `assertEquals()` aufgerufen.
Das erste Argument beinhaltet die Fehlermeldung, die ausgegeben wird, falls die Zusicherung des Tests (assertion) fehlschlägt. Das zweite Argument beinhaltet den erwarteten Wert.
Das dritte Argument liest mit Hilfe der öffentlichen Zugriffsmethode `getPathname()`den tatsächlichen, durch Funktionalität der Klasse `AudioFile` erbrachten Wert.

Bemerkung: man geht davon aus, dass die triviale Implementierung eines Getters wie `getPathname()` keine Fehler enthält. Die Intention des Tests ist die Prüfung der nicht-trivialen Methode `parsePathname()`, die aufgrund der Anforderungen einigen Code enthält und somit fehleranfällig ist.

Zeilen 14-16:
Hier wird ein weiterer Teil der Funktionalität der Methode `parsePathname()` getestet. Man prüft über die Getter-Methode `getFilename()`, ob der Dateiname korrekt zurückgegeben wird.
Wahrscheinlich werden Sie die Zerlegung des Pfadnames in Pfad und Dateiname in der Methode `parsePathname()` implementieren. Auch hier geht man davon aus, dass der dann triviale Getter `getFilename()` fehlerfrei implementiert ist.

Nachdem Sie das Test-Framework aufgesetzt haben und auch schon ein Muster für einen Test eingegeben haben, sind Sie nun in der Lage, Ihren Code wesentlich komfortabler und zielgerichteter zu testen, als über die konventionelle Methode durch Code in der main-Methode.

Reichern Sie nun die Klasse `AudioFile` solange um neuen Code an, bis der Testfall `test_parsePathname_03()` erfolgreich abgearbeitet wird. Beim Anlegen von Attributen achten Sie bitte darauf ,die Sichtbarkeiten so restriktiv wie möglich zu vergeben (im Regelfall `private`).

Die anderen Tests aus der Tabelle lassen sich analog zu dem gezeigten Test implementieren. Fügen Sie für jedes Testbeispiel der Tabelle eine eigene Testmethode in die Testklasse `UTestAudioFile` ein, und implementieren Sie parallel dazu solange Funktionalität in der Klasse `AudioFile`, bis alle Testfälle erfolgreich abgearbeitet werden.

In den Testmethoden müssen Sie für die Erzeugung der geforderten Ausgabe bisweilen feststellen,

auf welcher Plattform Ihr Code abläuft. Dazu ist folgende Hilfsmethode nützlich, die Sie ebenfalls in Ihre Testklasse einkopieren können.

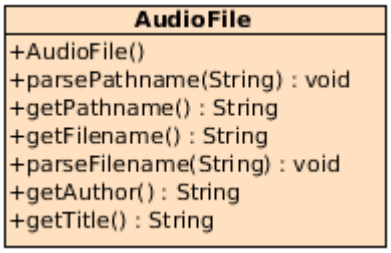
```
/*-------------------------------------------------------------------------
 * Auxiliary methods
 */

private boolean isWindows(){
     return System.getProperty("os.name").toLowerCase().indexOf("win") >= 0;
}
```

Teilaufgabe (c)

In dieser Teilaufgabe widmen wir uns der weiteren Analyse des Dateinamens (filename), den wir dank der Vorarbeiten schon mittels `getFilename()` auslesen können.

Häufig finden sich bei Audiodateien Dateinamen, die nach folgendem Schema aufgebaut sind:
`␣Interpret␣–␣Titel.Endung` (siehe hierzu auch die Beispiele der nachfolgenden Tabelle)
Die Leerzeichen vor und nach dem Bindestrich gehören nicht zu Interpret und Titel. Ebenso spielt ein eventuell angegebener Pfad natürlich bei der Ermittlung von Interpret und Titel keine Rolle.

Dem nebenstehenden Klassendiagramm können Sie entnehmen, dass wir drei weitere öffentliche Methoden von der Klasse `AudioFile` verlangen. Die erste Methode `parseFilename(String)` erledigt das Extrahieren und Speichern von Interpret und Titel aus dem Dateinamen, der als Argument übergeben wird. Die Methoden `getAuthor()` und `getTitle()` hingegen liefern die beiden Bestandteile Interpret und Titel zurück, soweit dies möglich ist.
Hinweis: im Englischen verwenden wir Author statt Interpret.



Entnehmen Sie nachfolgender Tabelle Beispiele, die die Funktionsweise der neuen Methoden genauer festlegen. Im Anschluss an die Tabelle finden Sie wieder ein Muster für einen Unit-Test, anhand dessen Sie alle Beispiele der Tabelle als Unit-Tests implementieren können.

Argument für parseFilename()	Resultat getAuthor()	Resultat getTitle()
`␣Falco␣-␣Rock␣me␣␣Amadeus␣.mp3␣␣`	Falco	`Rock␣me␣␣␣Amadeus`
`Frankie␣Goes␣To␣Hollywood␣-␣The␣Power␣Of␣Love.ogg`	`Frankie␣Goes␣To␣Hollywood`	`The␣Power␣Of␣Love`
`audiofile.aux`	Leerer String	audiofile
`␣␣␣A.U.T.O.R␣␣␣-␣␣T.I.T.E.L␣␣.EXTENSION`	`A.U.T.O.R`	`T.I.T.E.L`
`Hans-Georg␣Sonstwas␣-␣Blue-eyed␣boy-friend.mp3`	`Hans-Georg␣Sonstwas`	`Blue-eyed␣boy-friend`
`.mp3`	`Leerer String`	`Leerer String`
`Falco␣-␣Rock␣me␣Amadeus.`	Falco	`Rock␣me␣Amadeus`
`-`	`Leerer String`	`-`
`␣-␣`	`Leerer String`	`Leerer String`

Eine formale Definition via EBNF für Interpret (author) und Titel (title) überlassen wir als Übung.

Hier nun, wie versprochen, das Muster für einen Unit-Test, mit dem der vierte Test der Tabelle implementiert wird.

```
@Test
public void test_parseFilename_38() throws Exception {
    AudioFile af = new AudioFile();

    af.parsePathname("/tmp/test/   A.U.T.O.R   -  T.I.T.E.L  .EXTENSION");
    af.parseFilename(af.getFilename());

    assertEquals("Filename stored incorrectly",
            "   A.U.T.O.R   -  T.I.T.E.L  .EXTENSION",
            af.getFilename());
    assertEquals("Author stored incorrectly", "A.U.T.O.R", af.getAuthor());
    assertEquals("Title  stored incorrectly", "T.I.T.E.L", af.getTitle());
}
```

Teilaufgabe (d)

Implementieren Sie nun einen weiteren Konstruktor `AudioFile(String)`, der als Argument den Pfadnamen einer Audiodatei erwartet. Im Rumpf des Konstruktors rufen Sie dann in geeigneter Reihenfolge die bereits zur Verfügung stehenden Methoden `parsePathname()`, `parseFilename()` und `getFilename()` auf, so dass neben dem Dateiname auch Interpret und Titel analysiert und gespeichert werden (Hinweis: wie bei `test_parseFilename_38()`)

Die Verwendung dieses Konstruktors stellt die normale Schnittstelle zum Erzeugen eines Objekts der Klasse AudioFile dar.

Da der Code im Rumpf dieses Konstruktors verhältnismäßig einfach ist, sind hierfür derzeit noch nicht unbedingt Unit-Tests notwendig. In einem späteren Aufgabenblatt werden wir jedoch die Funktionalität des Konstruktors erweitern, was dann Anlass zu Unit-Tests geben wird.

Teilaufgabe (e)

Sofern ein Objekt in einem String-Kontext verwendet wird, wird von der Laufzeitumgebung automatisch die Methode `toString()` des Objektes aufgerufen. So wird etwa

```
String x = "Objekt: " + obj;
```

automatisch umgesetzt zu

```
String x = "Objekt: " + obj.toString();
```

Die Methode `toString()`, die jede Klasse von `java.lang.Object` erbt, kann natürlich durch eine eigene Methode überschrieben werden, um die Ausgabe den eigenen Bedürfnissen anzupassen.

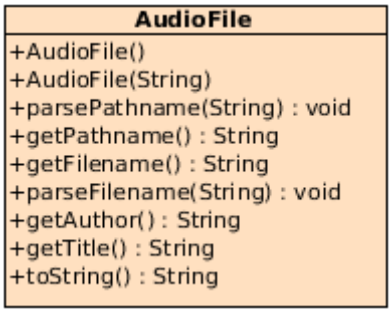
Um eine komfortablere Ausgabe zu ermöglichen, überschreiben wir nun in der Klasse `AudioFile` die Methode `java.lang.Object.toString()` mit einer an unsere Zwecke angepassten Methode. Die Methode soll einen String nach dem folgenden Schema zurück geben:

- falls die Methode `getAuthor()` einen leeren String liefert[6], soll nur der Titel ausgegeben werden.
- Ansonsten soll die Methode Interpret und Titel getrennt durch “_-_“ ausgeben

Formulieren Sie für obige Spezifikation geeignete Unit-Tests und implementieren Sie in der Klasse `AudioFile` den erforderlichen Code.

6 Bitte testen Sie auf leeren String mittels `isEmpty()` oder `equals(“”)`, nie jedoch via `== “”`

Mit Abschluss dieser letzten Teilaufgabe haben Sie alle Aufgaben dieses Angabenblatts erledigt. Mittlerweile sollte Ihre Klasse `AudioFile` zumindest über die im nachfolgenden Klassendiagramm gezeigten öffentlichen Methoden verfügen.



Zusammenfassung

Als wesentliche Lernziele dieses Aufgabenblatts sind folgende Punkte hervorzuheben:

- Zerteilung (parsen) und Manipulation von Zeichenketten (Strings)
- Der mit der Implementierung verzahnte Einsatz von Unit-Tests
- Beschreibung von Anforderungen durch textuelle Formulierungen, UML-Klassendiagramme und Beispiele bzw. deren Implementierung durch Unit-Tests.
- Die exakte Formulierung von Anforderungen gelingt nur durch den Einsatz formaler Beschreibungsmittel (hier EBNF)

Und nun viel Spaß und Erfolg beim Bearbeiten dieses Aufgabenblatts!

Hinweise zur Abnahme Ihrer Implementierung der Vorführaufgabe 06

Im Zuge der Bearbeitung aller Teilaufgaben dieses Blattes haben Sie im Unterverzeichnis `src` eine Klasse `AudioFile.java` erzeugt. Diese Java-Datei müssen Sie an den APA-Server schicken. 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 (APA-Server) prüfen zu lassen.

Wegen der plattformabhängigen Behandlung der Separatoren `'/'` und `'\'` kann es, abhängig von der Plattform auf der Java ausgeführt wird, zu unterschiedlichen Testergebnissen kommen. Das Verhalten auf einer Linux-Plattform, auf der der APA-Server betrieben wird, können Sie bei diesem Aufgabenblatt auch unter Windows prüfen, indem Sie in der Testklasse `AudioFileTest` die Anweisung

```
EmulateOtherOs.emulateLinux();
```

aktivieren!

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.