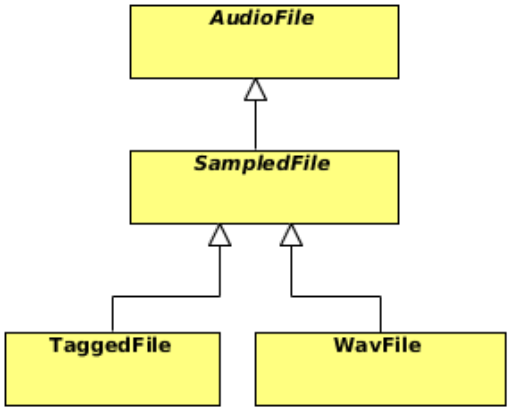
Technische Hochschule Ingolstadt
Fakultät für Elektrotechnik und Informatik

Vorführaufgabe 7: Vererbungshierarchie

Ziel der Vorführaufgabe 7 ist es, die rechts abgebildete Klassenhierarchie und die Basisfunktionalitäten des Audioplayers zu implementieren.

Die Basisklasse `AudioFile` wurde bereits in Aufgabe 6 erstellt. Von ihr wird eine Klasse `SampledFile` abgeleitet, die gesampelte Audiodateien - etwa MP3, OggVorbis oder RIFF WAVE- repräsentiert. Die Klasse greift zum Dekodieren der Audiodateien auf externe Bibliotheken zurück (genauer gesagt auf Java-Archive, sogenannte JAR-Dateien).

In unserem Projekt werden von `AudioFile` außer `SampledFile` keine weiteren Klassen abgeleitet. Streng genommen könnte man also die beiden Klassen zu einer zusammenfassen. Es gibt aber Audiodateien wie z.B. MIDI, die nicht gesampelt sind und daher sauberer direkt von `AudioFile` abgeleitet werden würden. Aus diesem Grund nehmen wir diese Trennung vor.



Finale Klassenhierarchie

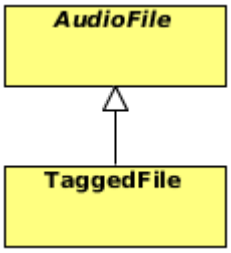
Die von `SampledFile` abgeleitete Klasse `TaggedFile` repräsentiert Audiodateien mit Meta-Daten wie etwa ID3-Tags bei MP3 oder Vorbis Comments bei OggVorbis. Die andere von `SampledFile` abgeleitete Klasse repräsentiert RIFF WAVE-Dateien, die keine Metainformationen in Form von Tags bereitstellen. Solche Dateien erhält man beispielsweise, wenn man CDs mit einem entsprechendem Tool "rippt".

Das obige Klassendiagramm stellt die finale Version der Klassenhierarchie dar, in der die Funktionalität zur Manipulation von Audio-Dateien über die einzelnen Klassen verteilt ist. Diese Version werden wir erst am Ende dieses Arbeitsblatts erreicht haben.

Den Prinzipien der objektorientierten Programmierung folgend, wird die Klasse `SampledFile` dabei Attribute und Methoden enthalten, die gemeinsam von den Klassen `TaggedFile` und `WavFile` genutzt werden können. Die Klasse `AudioFile` an der Wurzel der Hierarchie wird Attribute und Methoden enthalten, die zum einen von der Klasse `SampeldFile` und den von ihr abgeleiteten Klassen genutzt werden, zum anderen aber auch von Klassen genutzt werden können, die direkt von `AudioFile` ableiten.

Die Einsicht, welche Attribute und Methoden in gemeinsamen Oberklassen implementiert werden können, fällt jedoch nicht vom Himmel. Im allgemeinen entdeckt man das Potential zur Generalisierung erst, wenn man mehrere Klassen implementiert hat und danach bei deren Inspektion feststellt, das das ein oder andere Attribut oder bestimmte Methoden in eine Basisklasse verschoben werden können. Manchmal gibt es eine solche Basisklasse schon, oft wird sie aber auch erst nach der Entdeckung von Generalisierungspotential angelegt.

Da wir Ihnen diesen zentralen Erkenntnisprozess nicht vorenthalten wollen, beginnen wir zunächst mit der nebenstehenden einfacheren Hierarchie. Nachdem wir die erste Klasse TaggedFile implementiert haben und uns an die Umsetzung der Klasse WavFile machen, werden wir bereits das Potential zu Generalisierung entdecken und gemeinsame Daten und Funktionalität in die dann neu zu erstellende Klasse SampledFile auslagern.
(Siehe Teilaufgabe j)



Vorläufige Klassenhierarchie

Vorbereitungen

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

Empfehlung: Klonen Sie zunächst im Package-Explorer von Eclipse das Projektverzeichnis, in dem Sie die Aufgabe 6 implementiert haben, und geben Sie dem Klon einen neuen Namen. Wenn Sie z.B. das Projekt der Aufgabe 6 im Verzeichnis `MediaPlayer_GdPII_VA06` gespeichert haben, legen Sie nun eine Kopie des Verzeichnisses an und benennen die Kopie `MediaPlayer_GdPII_VA07`[1].

Löschen Sie sodann alle Dateien im Unterverzeichnis `cert`. Das Unterverzeichnis beinhaltet aufgrund des Klonens die Server-Tests der Aufgabe 6. Laden Sie nun die zur Aufgabe 7 gehörigen Zip-Archive `cert.zip` und `build.zip` aus dem Intranet (Seite für VA07[2]) und entpacken Sie beide im Wurzelverzeichnis Ihres Projektes. Das neue Archiv `cert.zip` enthält die Server-Tests für die VA07. Das Archiv `build.zip` enthält wieder die Ant-Build-Umgebung für die VA07. Die Dateien im Archiv `build.zip` werden im Wurzelverzeichnis ausgepackt und überschreiben die Dateien der Aufgabe 6.

Ab dieser Vorführaufgabe sollten Sie die Funktionalität grundsätzlich mit echten Audio-Dateien testen. Zu diesem Zweck finden Sie im Intranet auf der Seite der Aufgabe 7 eine Auswahl an geeigneten Audio-Dateien in den Formaten MP3 (*.mp3), OggVorbis (*.ogg) und RIFF WAVE (*.wav). Die Dateien sind im Archiv `audiofiles.zip` enthalten. Entpacken Sie dieses Archiv im Wurzelverzeichnis Ihres Projektes für die Aufgabe 7. Beim Entpacken entsteht automatisch ein Unterverzeichnis `audiofiles`, welches die Audio-Dateien enthält.

Der Code, den Sie im folgenden entwickeln werden, wird mehrfach Methoden aus Audio-Bibliotheken nutzen, 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 7 finden. Entpacken Sie dieses Archiv wieder im Wurzelverzeichnis Ihres Projektes. Beim Entpacken entsteht ein Unterverzeichnis `lib`, welches die nachfolgenden Jar-Bibliotheken enthält.

```
jl1.0.jar,        mp3spi1.9.4.jar,       tritonus_share.jar,
jogg-0.0.7.jar,   jorbis-0.0.15.jar,     vorbisspi1.0.2.jar,
studiplayer.jar
```

Fügen Sie dann **jede** dieser Jar-Bibliotheken dem Klassenpfad Ihres Projektes hinzu. In Eclipse finden Sie dazu im Kontextmenü des Projektes, dass Sie in der Standardansicht mit einem Rechtsklick auf das Projekt im Package-Explorer auf der linken Seite erreichen, den Eintrag "Build-Path“ → “Configure Build Path“ → “Libraries“ → “Add Jars“.
Eine Mehrfachselektion von Bibliotheken ist möglich!

In einigen Teilaufgaben werden Klassen verwendet, die Sie nicht selbst schreiben werden, die sich aber auch nicht im Paket `java.lang` befinden, welches automatisch importiert ist. Diese zusätzlichen Klassen müssen jeweils durch eine entsprechende `import`-Anweisung in Ihrem Code eingebunden werden (siehe Fußnoten).

Teilaufgabe a (Lesetest für Datei in Klasse AudioFile)

Bauen Sie in den Konstruktor `AudioFile(String pathname)` der Klasse `AudioFile` einen Test ein, der prüft ob der im Parameter des Konstruktors übergebene Pfadname tatsächlich auf eine lesbare Datei zeigt. Nutzen Sie für den Zugriffstest den normalisierten Pfadnamen, den die Methode `getPathname()` liefert (dh. den Test erst nach Bereinigung des Pfadnames ausführen).

Hinweis: die Methode `canRead` der Klasse `File`[3] erlaubt Ihnen, die Lesbarkeit einer Datei zu

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

2 VA07 bedeutet Vorführaufgabe 07

3 `import java.io.File;`

prüfen. Dazu müssen Sie aber zunächst ein Objekt der Klasse `File` erzeugen.

Falls die Datei nicht lesbar ist, soll die Anwendung mittels

```
throw new RuntimeException(<Ihr Fehlertext mit Nennung des Pfadnamens>);
```

eine Ausnahme (Exception) generieren. Da wir erst später eine Fehlerbehandlung (catch) für diese Ausnahme an geeigneter Stelle einführen, resultiert das Erzeugen der Exception (throw exception) derzeit auch im sofortigen Abbruch der Anwendung.

Teilaufgabe b (Erzeugen der vorläufigen Klassenhierarchie)

Häufig ist es sinnvoll, in Basisklassen („Elternklassen“) Methoden zu implementieren, damit diese von allen Subklassen („Kindklassen“) genutzt werden können. Teilweise macht es dabei jedoch keinen Sinn, Instanzen der Basisklassen selbst zu erlauben. In diesem Fall kann die Klasse durch das Schlüsselwort `abstract` vor dem Klassennamen zu einer abstrakten Klasse erklärt werden. Damit kann zwar von dieser Klasse geerbt werden, es ist jedoch nicht möglich, direkt ein Objekt dieser Klasse zu instantiieren.

Diese Technik wenden wir zunächst auf die Klasse `AudioFile,` später auch auf die Klasse `SampledFile` an. Die Motivation hierzu erfolgt in Teilaufgabe c.

Erstellen Sie nun die vorerst leere Klasse `TaggedFile` entsprechend der oben skizzierten Vererbungshierarchie (siehe Abbildung 'Vorläufige Klassenhierarchie) und deklarieren Sie die Klasse `AudioFile` als abstrakte Klasse.

Teilaufgabe c (abstrakte Methoden der Klasse AudioFile)

Weiter ist es möglich, in einer abstrakten Klasse abstrakte Methoden zu definieren. Diese bestehen nur aus einem Methodenkopf (auch Signatur genannt), der wie die abstrakte Klasse selbst auch mit dem Schlüsselwort `abstract` versehen ist. Abstrakte Methoden selbst implementieren keine Funktionalität, fordern aber von jeder nicht-abstrakten Kindklasse die Implementierung einer Methode mit der vorgegebenen Signatur.

Die abstrakte Klasse `AudioFile` soll eine Schnittstelle festlegen, die allen von ihr abgeleiteten Klassen gemeinsam ist. Egal um welche Kindklasse von `AudioFile` es sich genau handelt , soll es für alle `AudioFiles` möglich sein, den Abspielvorgang zu starten '`play()`', eine Pausierung an- oder auszuschalten '`togglePause()`' oder den Abspielvorgang zu beenden '`stop()`'.

Weiterhin brauchen wir für die später entstehende grafische Oberfläche die Möglichkeit, die Gesamtspieldauer eines Liedes und die aktuelle Position im gerade gespielten Lied als String in einem gut lesbaren Format darzustellen. Diesen Dienst sollen die beiden Methoden `getFormattedDuration()` und `getFormattedPosition()` erbringen.

Je nach Kindklasse muss die Berechnung von Dauer und Position unterschiedlich erfolgen, wir möchten jedoch bereits in der abstrakten Basisklasse `AudioFile` die pure Existenz und Signatur der Methoden `getFormattedDuration()` und `getFormattedPosition()` festlegen.

Zu diesem Zweck versehen Sie nun bitte die Klasse `AudioFile` mit Deklarationen für folgende abstrakte Methoden:

- `public abstract void play();`
- `public abstract void togglePause();`
- `public abstract void stop();`
- `public abstract String getFormattedDuration();`
- `public abstract String getFormattedPosition();`

Teilaufgabe d (Implementierung Klasse TaggedFile, Teil 1)

In dieser Teilaufgabe implementieren wir die Funktionalität der in `AudioFile` geforderten Methoden `play()`, `togglePause()` und `stop()`in der Klasse `TaggedFile`.

Später werden wir diese Methoden nach `SampledFile` verschieben. Da wir aber schrittweise vorgehen und es die Klasse `SampledFile` noch nicht gibt, erfolgt die Implementierung zunächst in der Klasse `TaggedFile`.

Die tatsächliche Audio-Wiedergabe von Audio-Dateien ist nicht ganz einfach. Daher werden Ihnen die niederen Zugriffsfunktionen auf Audio-Dateien in Form der Bibliothek `studiplayer.jar` zur Verfügung gestellt. Diese Bibliothek haben Sie bereits im Rahmen der Vorbereitungen am Anfang dieses Angabenblatts in Ihr Projekt eingebunden.

Implementieren Sie nun die drei Methoden `play()`, `togglePause()` und `stop()` in der Klasse `TaggedFile` mit folgenden Funktionalitäten:

- Die Methode `play()` in der Klasse `TaggedFile` ruft die gleichnamige statische Methode der Klasse `studiplayer.basic.BasicPlayer` auf, die als Parameter den Dateinamen der abzuspielenden Datei erwartet. Sie können also etwa mit
  ```
  BasicPlayer[4].play("lied.mp3");
  ```
  die Audio-Datei `lied.mp3` abspielen.
  Natürlich sollte Ihre Implementierung der Methode `play()` in der Klasse `TaggedFile` als Argument den im Objekt gespeichert Pfadnamen der Audio-Datei verwenden.
- Die Methoden `togglePause()` und `stop()` rufen die gleichnamigen und parameterlosen statischen Methoden der Klasse `BasicPlayer` auf.

In der späteren Teilaufgabe h werden wir auch die Methoden `getFormattedDuration()` und `getFormattedPosition()` in der Klasse TaggedFile mit sinnvollen Methodenrümpfen implementieren. Zunächst jedoch implementieren Sie bitte diese Methoden nur mit trivialen Rümpfen.

- Implementieren Sie in der Klasse `TaggedFile` die Methode `getFormattedDuration` wie folgt mit trivialem Rumpf:
  ```
  public String getFormattedDuration() { return ““; }
  ```
- Implementieren Sie in der Klasse `TaggedFile` die Methode `getFormattedPosition` wie folgt mit trivialem Rumpf:
  ```
  public String getFormattedPosition() { return ““; }
  ```

Zu guter Letzt müssen wir noch die Konstruktoren für die Klasse `TaggedFile` implementieren.

- Implementieren Sie den Konstruktor `TaggedFile()` ohne Argumente;
  Er soll einfach den Konstruktor der Superklasse aufrufen.
  Diesen Konstruktor werden wir in Unit-Tests verwenden
- Implementieren Sie den Konstruktor `TaggedFile(String s)`
  Dieser Konstruktor soll als einziges Argument den Pfadnamen einer Audio-Datei entgegennehmen und im Rumpf die Arbeit an den entsprechenden Konstruktor (den mit einem Argument) der Superklasse `AudioFile` delegieren.

Teilaufgabe e (Unit-Tests für TaggedFile, Teil 1)

Eine aktuelle Praxis der Software-Entwicklung ist die enge Verzahnung der Erstellung von Produktiv-Code mit der Erstellung von Test-Code. Hierbei kommt im Umfeld der Java-Entwicklung das JUnit-Framework zum Einsatz. Das künstliche Hinzufügen von main-Methoden in Klassen des Produktiv-Codes, wobei diese main-Methoden lediglich geschrieben werden, damit sie Test-Code aufnehmen können, wird im professionellen Entwicklungsumfeld nicht mehr praktiziert.
Test-Code wird strikt getrennt von Produktiv-Code!

Um die Vorgabe der Trennung von Produktiv-Code und Test-Code einzuhalten, speichert man Unit-Tests nicht im Unterverzeichnis `src`, in dem der Produktiv-Code liegt , sondern in einem anderen Verzeichnis. In diesem Projekt verwenden wir hierfür das Verzeichnis `tests`.

4 `import studiplayer.basic.BasicPlayer;`

- Legen Sie im Unterverzeichnis `tests` eine Test-Klasse `UTestTaggedFile` mit nachfolgendem Inhalt an und führen Sie dann den enthaltenen Test aus.

```
import org.junit.Test;
public class UTestTaggedFile {
    @Test
    public void test_play_01() throws Exception {
        TaggedFile tf = new TaggedFile("audiofiles/Rock 812.mp3");
        tf.play();
        // Note: cancel playback in eclipse console window
    }
}
```

- Fügen Sie weitere Tests nach obigem Muster mit anderen Audio-Dateien (*.mp3, *.ogg) aus dem Verzeichnis `audiofiles` hinzu. Bitte sparen Sie dabei aber die Dateien mit Namensmuster ***.cut.*** aus. Diese Dateien sind absichtlich fehlerhaft kodiert. Sie werden nur für Unit-Tests benutzt, die absichtlich Fehler provozieren.

Hinweis:
Man kann Unit-Tests gezielt einzeln ausführen. Weiterhin kann man einzelne Unit-Tests auch von der Ausführung ausnehmen (ignorieren). Hierzu muss man nur vor den Test (vor oder nach der Annotation `@Test`) die Annotation `@Ignore`[5] schreiben.

Teilaufgabe f (time Formatter)

Innerhalb des vorliegenden Projektes sollen Zeitangaben wie Abspieldauer oder die aktuelle Position im Lied bei der Ausgabe in der grafischen Oberfläche immer im Format *mm:ss* dargestellt werden, d.h. es gibt immer zwei Ziffern für die Minutenangabe und zwei Ziffern für die Sekundenangabe, wobei die beiden Angaben durch einen Doppelpunkt getrennt sind.

Für die interne Verarbeitung selbst werden die Zeiten in Mikrosekunden[6] verwaltet, so dass hier eine Umrechnung erforderlich ist. Nach der Umrechnung von Mikrosekunden in das Format *mm:ss* werden verbleibende Sekundenbruchteile ignoriert. d. h. etwaige Nachkommastellen werden verworfen und es wird nicht gerundet. So führt etwa die intern verwaltete Abspieldauer von 305862000 Mikrosekunden zu einer Ausgabe des Strings "05:05" (05 Minuten und 05 Sekunden).

Die Aufbereitung der intern in Mikrosekunden gespeichert Zeiten als formatierten String wird sowohl für die Angabe der Gesamtabspieldauer als auch für die Ausgabe der aktuellen Position im Lied benötigt. Es bietet sich daher an, diese Funktionalität in eine Hilfsfunktion auszulagern.

- Implementieren Sie in der Klasse `TaggedFile` die Methode
  `public static String timeFormatter(long microtime)`

Die Methode wandelt eine Zeitangabe in Mikrosekunden nach den obigen Angaben in das Format "mm:ss" um.

Erstellen und nutzen Sie während der Implementierung der Methode `timeFormatter()` zusätzlich Unit-Tests. Fügen Sie Ihre Tests der Datei `UTestTaggedFile` hinzu.
Im folgenden ist ein Muster für einen Unit-Test angegeben:

```
@Test
public void test_timeFormatter_10() throws Exception {
    assertEquals("Wrong time format", "05:05",
            TaggedFile.timeFormatter(305862000L));
}
```

5 `import org.junit.Ignore`

6 Eine Sekunde besteht aus 10^6 Mikrosekunden

Die Methode `timeFormatter()` soll auch die beiden nachfolgenden Möglichkeiten für fehlerhafte bzw. problematische Argumentwerte behandeln:

1. Der übergebene Zeitwert in Mikrosekunden ist negativ. In diesem Fall soll die Methode folgende Ausnahme (Exception) generieren:

   ```
   throw new RuntimeException("Negativ time value provided");
   ```

2. Der übergebene Zeitwert in Mikrosekunden ist zu groß für das Format "mm:ss". In diesem Fall soll die Methode folgende Ausnahme (Exception) generieren:

   ```
   throw new RuntimeException("Time value exceeds allowed format");
   ```

   Hinweis: überlegen Sie sich zuerst, ab welchem Wert ein Überlauf auftreten wird.

Die korrekte Auslösung von Ausnahmen (Exceptions) kann durch Unit-Tests der folgende Art getestet werden. Fügen Sie Ihre Tests wieder der Datei `UTestTaggedFile` hinzu.

```
@Test
public void test_timeFormatter_08() throws Exception {
    try {
        // Call method with time value that underflows our format
        TaggedFile.timeFormatter(-1L);
        // We should never get here
        fail("Time value underflows format; expecting exception");
    } catch (RuntimeException e) {
        // Expected
    }
}
```

Teilaufgabe g (readAndStoreTags)

Wie weiter oben bereits beschrieben, können einige Audio-Dateien (mp3, ogg) Meta-Daten (Tags) enthalten. Zur Bearbeitung dieser speziellen Art von Audio-Dateien haben wir die Klasse `TaggedFile` vorgesehen. In diesen Tags können unter anderem Informationen zu Titel, Interpret und Album oder auch Informationen zu der Audio-Datei selbst enthalten sein.

Die statische Methode `readTags()` der Klasse `studiplayer.basic.TagReader`[7] erlaubt es, die einzelnen Tags, also die einzelnen Werte der Meta-Daten, auszulesen.
Das nachfolgende Code-Fragment, welches wir zu Testzwecken in einen Unit-Tests verpackt haben, zeigt die Verwendung der Schnittstelle. Es werden alle in der Audio-Datei gespeichert Tags auf der Konsole ausgegeben:

```
// Read all tags from a TaggedFile
// This test demonstrates the functionality of TagReader.readTags()
@Test  // @Ignore
public void test_readTags_01() throws Exception {
    TaggedFile tf = new TaggedFile("audiofiles/Rock 812.mp3");
    Map<String, Object> tag_map = TagReader.readTags(tf.getPathname());
    for (String key : tag_map.keySet()) {
        System.out.printf("\nKey: %s\n",key);
        System.out.printf("Type of value: %s\n", tag_map.get(key).getClass().toString());
        System.out.println("Value: " + tag_map.get(key));
    }
}
```

7 Import studiplayer.basic.TagReader

Der Code im Unit-Test nutzt das Interface `Map`[8] der Java-Container-Bibliothek. Studieren Sie in der Java-Dokumentation das Interface (API) von `Map`, insbesondere die für Sie wichtige Methode `get()`, die den Zugriff auf einen unter einem Schlüssel (key) gespeicherten Wert erlaubt.

- Implementieren Sie in der Klasse `TaggedFile` die Methode
  `public void readAndStoreTags(String pathname)`

  Die Methode nutzt `studiplayer.basic.TagReader.readTags()`, um die Tags "`title`", "`author`", "`album`" und "`duration`" aus derjenigen Datei auszulesen, deren Pfadname im Parameter `pathname` übergeben wird.

  Falls die Tags belegt sind und *sinnvolle* Werte enthalten (also nicht null-Referenz oder leerer String oder Wert 0), sollen die ausgelesenen Werte in geeigneter Weise in Attributen der Klasse (oder in Attributen einer Basisklasse) gespeichert werden. Sofern nötig, ändern Sie hierzu die Sichtbarkeiten (`private → protected`) der Attribute in der Basisklasse.

  Sollten einige Attribute bereits mit Werten belegt sein, hat der aus dem Tag gelesene Wert (falls sinnvoll) höhere Priorität und soll den alten Wert des Attributs überschreiben.

  Beispiel: evtl. speichern sie bereits einen Titel oder Autor, weil die entsprechenden Attribute schon beim Aufruf von `parseFilename()` im Code der Basisklasse belegt wurden.

- Erweitern Sie den Konstruktor der Klasse `TaggedFile`, der einen Pfadnamen als Argument erwartet, dahingehend, dass nach Aufruf des Konstruktors der Basisklasse (super) die Methode `readAndStoreTags()` mit einem geeigneten Argument aufgerufen wird.
  Hinweis: Sie sollten ausnützen, das der Konstruktor der Basisklasse bereits eine Normalisierung und Überprüfung des Pfadnamens ausführt. Holen Sie sich dieses Ergebnis und nutzen Sie nicht den ungeprüften String.

Teilaufgabe h (getFormattedDuration, getFormattedPosition)

Mit Hilfe der statischen Methode `timeFormatter()` aus der früheren Teilaufgabe f können wir nun die Rümpfe der Methoden `getFormattedDuration()` und `getFormattedPosition()` sinnvoll ausprogrammieren. Bisher enthielten diese nur Dummy-Code (siehe Teilaufgabe d).

- Implementieren Sie in der Klasse `TaggedFile` die Methode
  `public String getFormattedDuration()`
  Lesen Sie im Rumpf der Methode das Attribut aus, das Sie für die Speicherung des unter dem Schlüssel "duration" (Tag) hinterlegten Werts verwendet haben (siehe letzte Teilaufgabe). Zur Aufbereitung als String im Format "mm:ss" nutzen Sie die statische Methode `timeFormatter()`

- Implementieren Sie in der Klasse `TaggedFile` die Methode
  `public String getFormattedPosition()`
  Diese Methode soll die aktuelle Abspielposition der Audio-Datei ausgeben. Die momentane Abspielposition in Mikrosekunden ab Anfangsposition lässt sich durch den Aufruf
  `studiplayer.basic.BasicPlayer.getPosition()`
  in Erfahrung bringen. Zur Aufbereitung als String im Format "mm:ss" nutzen Sie wieder die statische Methode `timeFormatter()`
  Hinweis: die Methode `getFormattedPosition()` können wir nur testen, wenn wir mit zwei Threads arbeiten. Der eine spielt das Lied ab, der andere greift die Abspielposition ab. Falls Sie so einen Test schreiben möchten, finden Sie ein Beispiel in der Methode `testPlayPauseStop()` in der Datei `cert/SampledFileTest.java`

8 import java.util.Map

Teilaufgabe i (getAlbum und toString)

Zur Abrundung der Klassenschnittstelle fehlen uns noch zwei Methoden:

- Implementieren Sie die Methode
  `public String getAlbum()`
  welche den Namen des Albums zurückgibt (siehe `readAndStoreTags()`).

- Die Klasse soll eine `toString()`-Methode haben, die einen wie folgt formatierten String zurück gibt:
  - falls das Album nicht bekannt ist: „`Interpret – Titel – Dauer`“
  - ansonsten: „`Interpret – Titel – Album – Dauer`“

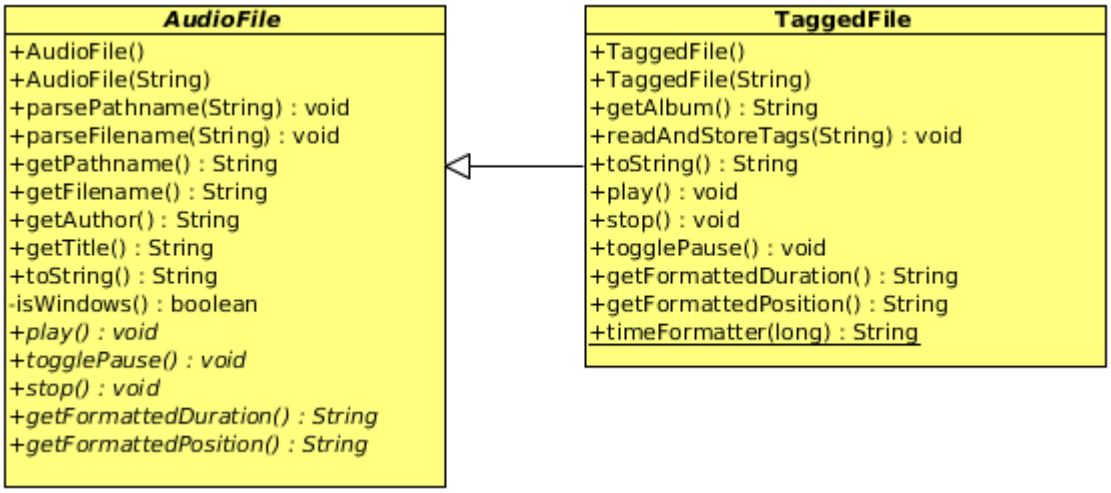
  Hinweise:
  - verwenden Sie die Methode `getFormattedDuration()`
  - verwenden Sie die `toString()`-Methode der Basisklasse.

Der nachfolgende Unit-Test zeigt Ihnen, wie Sie Ihre Methode `readAndStoreTags()` testen können.

```
@Test
public void test_readAndStoreTags_03() throws Exception {
    TaggedFile tf = new TaggedFile();
    tf.readAndStoreTags("audiofiles/Rock 812.mp3");
    assertEquals("Wrong author", "Eisbach",tf.getAuthor());
    assertEquals("Wrong title", "Rock 812",tf.getTitle());
    assertEquals("Wrong album", "The Sea, the Sky",tf.getAlbum());
    assertEquals("Wrong time format", "05:31",tf.getFormattedDuration());
}
```

Zwischenbilanz

Die Implementierung der Klasse `TaggedFile` ist nun im wesentlichen abgeschlossen. Das nachfolgende UML-Klassendiagramm zeigt die bisher implementierten Klassen mit ihren Schnittstellen.



Vorbetrachtung zur Generalisierung

Nun wollen wir in einem zweiten großen Schritt die Klasse `WavFile` implementieren. Sie soll ebenfalls von der Klasse `AudioFile` ableiten. Wie eingangs erwähnt, soll die Klasse `WavFile` Audio-Dateien repräsentieren, die zwar gesampelt sind, aber keine Meta-Daten in Form von Tags bereitstellen.

Obwohl keine Meta-Daten in Form von Tags in diesen Audio-Dateien gespeichert sind, lassen sich doch, über die Nutzung von Audio-Bibliotheken, Informationen über die Audio-Dateien auslesen. Über die Klasse `studiplayer.basic.WavParamReader` und deren Methoden[9] lassen sich beispielsweise Basisdaten extrahieren, die bei geschickter Kombination die Berechnung einer Gesamtspieldauer der Audio-Datei erlauben. Im Unterschied zu Dateien mit Tags lässt sich aber keine Information über ein Album extrahieren. Diese Information ist einfach nicht enthalten.

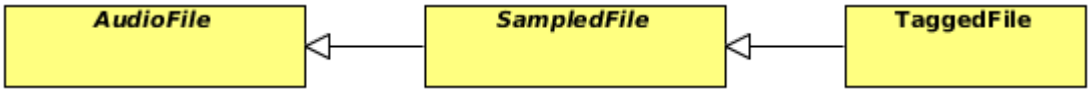
Analog zur Klasse `TaggedFile` müssten wir in der Klasse `WavFile` die in der Klasse `AudioFile` geforderten Methoden `play(), togglePause(), stop(), getFormattedDuration()` und `getFormattedPosition()` implementieren. Für die Implementierung der beiden letztgenannten Methoden stünde natürlich auch wieder eine Formatierung der Gesamtspieldauer und Position im Format “mm:ss“ an.

Hier zeigt sich nun das Potential für eine Generalisierung. Anstatt all diese Methoden und die von Ihnen benötigten Attribute (zB die Gesamtspieldauer) in `WavFile` erneut zu implementieren, versucht man alle Elemente (Attribute und Methoden) der Klasse `TaggedFile`, die auch für `WavFile` verwendet werden können, in eine gemeinsame Oberklasse auszulagern. Da es sich hierbei aber zumeist um Elemente handelt, die nur bei gesampelten Audio-Dateien vorliegen, wäre es nicht adäquat, die bereits bestehende Oberklasse `AudioFile` für diesen Zweck zu nutzen[10].

Aus diesem Grund führen wir nun schlussendlich die Oberklasse `SampledFile` ein, die aber selbst wiederum von `AudioFile` abgeleitet ist.

Teilaufgabe j (Generalisierung, Einführung SampledFile)

Erweitern Sie nun Ihre Klassenhierarchie um die Klasse `SampledFile`, so dass im Ergebnis die nachfolgende Klassenhierarchie resultiert.



- Machen Sie die Klasse `SampledFile` abstrakt. Es macht keinen Sinn, Instanzen (Objekte) dieser Klasse zu erzeugen.
- Statten Sie die Klasse `SampledFile` mit zwei Konstruktoren aus. Einer hat keinen Parameter, der andere hat erwartungsgemäß einen String-Parameter. Beide delegieren per (`super`) die Konstruktion an die Oberklasse.
- Verschieben Sie folgende Methoden von `TaggedFile` nach `SampledFile`:
  - `play(), togglePause(), stop(), getFormattedDuration(), getFormattedPosition(), timeFormatter()`
- Durch die Verschiebung der Methoden müssen Sie evtl. auch das ein oder andere Attribut der Klasse `TaggedFile` nach `SampledFile` verschieben. Evtl. muss die Sichtbarkeit des Attributs angepasst werden.

Teilaufgabe k (Implementierung Klasse WavFile)

Implementieren Sie nun die Klasse `WavFile`, die abgeleitet wird von `SampledFile`.

9 Details über die Methoden gibt es weiter unten
10 Lediglich die statische Methode `timeFormatter()` könnte man nach AudioFile verschieben

Bei diesem Typ von Audio-Datei kann die Gesamtspielzeit nicht direkt ausgelesen werden. Man kann jedoch über die drei nachfolgenden statischen Methoden der Klasse `WavParamReader`[11]

- `WavParamReader.readParams(Dateiname);`
- `WavParamReader.getFrameRate();`
- `WavParamReader.getNumberOfFrames();`

die Gesamtanzahl der Frames (`getNumberOfFrames()`) und die Anzahl der Frames pro Sekunde (`getFrameRate()`) auslesen. Daraus lässt sich mit einer einfachen Berechnung die Gesamtabspieldauer des Liedes errechnen. Die Methode `readParams(Dateiname)` muss dabei vor Verwendung der anderen beiden Methoden aufgerufen werden, damit diese auch die zur Audiodatei `Dateiname` gehörenden Daten liefern.

- Schreiben Sie eine statische Hilfsmethode
  ```
  public static long computeDuration(long numberOfFrames,
                                     float frameRate)
  ```
  für die Klasse `WavFile`, die aus der Gesamtanzahl der Frames und der Frame-Rate (Frames pro Sekunde) die Gesamtabspieldauer in Mikrosekunden berechnet.

  Nutzen Sie Unit-Tests der folgenden Bauart:

  ```
  @Test
  public void test_computeDuration_02() throws Exception {
      assertEquals("Wrong duration",2000000L ,
                   WavFile.computeDuration(88200L,44100.0f));
  }
  ```

- Schreiben Sie für die Klasse `WavFile` eine Methode
  ```
  public void readAndSetDurationFromFile(String pathname)
  ```
  die die Werte für Gesamtanzahl der Frames und Frame-Rate aus der mit `pathname` bezeichneten Audio-Datei mittels der Methoden der Klasse `WavParamReader` ausliest und in einem geeigneten Attribut speichert. Diese Methode nutzt die Hilfsmethode `computeDuration()` für die Berechnung der Abspieldauer.

  Hinweis: das Attribut zur Speicherung der Abspieldauer kann auch in einer Basisklasse liegen

- Erweitern Sie den Konstruktor der Klasse `WavFile`, der einen Pfadnamen als Argument erwartet, dahingehend, dass nach Aufruf des Konstruktors der Basisklasse (`super`) die Methode `readAndSetDurationFromFile()` mit einem geeigneten Argument aufgerufen wird.
  Hinweis: Sie sollten ausnützen, das der Konstruktor der Basisklasse bereits eine Normalisierung und Überprüfung des Pfadnamens ausführt. Holen Sie sich dieses Ergebnis und nutzen Sie nicht den ungeprüften String.

  Nutzen Sie Unit-Tests der folgenden Bauart:

```
@Test
public void test_readAndSetDurationFromFile_01() throws Exception {
    WavFile wf = new WavFile();
    wf.parsePathname("audiofiles/wellenmeister - tranquility.wav");
    wf.readAndSetDurationFromFile(wf.getPathname());
    assertEquals("Wrong time format", "02:21", wf.getFormattedDuration());
}
```

- Die Klasse soll eine `toString()`-Methode haben, die einen wie folgt formatierten String zurück gibt:

11 `import studiplayer.basic.WavParamReader;`

- „Interpret – Titel – Dauer“

Hinweise:
- verwenden Sie die Methode `getFormattedDuration()`
- verwenden Sie die `toString()`-Methode der Basisklasse.

Teilaufgabe I (abstrakte Methode fields() und ihre Implementierungen)

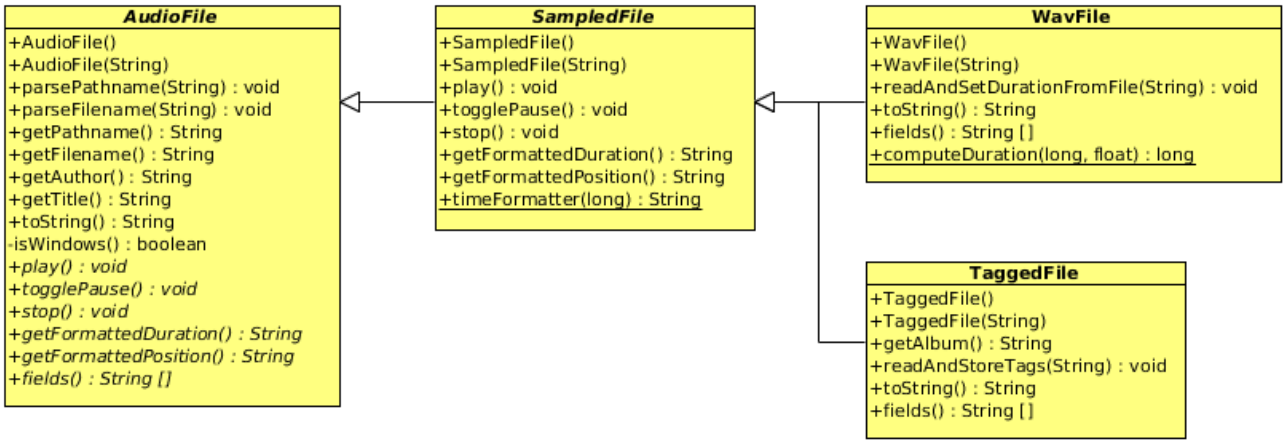
Für die Darstellung in der später zu entwickelnden grafischen Oberfläche brauchen wir eine Getter-Methode `fields()`, die ein String-Array der Länge 4 mit den Werten für Interpret, Titel, Album und Abspieldauer zurück liefert.

Da diese Methode für alle Audio-Dateien nützlich ist, soll bereits die Basisklasse `AudioFile` eine Signatur dafür festlegen.

- Legen Sie in der Klasse `AudioFile` eine abstrakte Methode
  ```
  public abstract String[] fields();
  ```
  an. Damit fordern wir die Existenz einer solchen Methode in den nicht-abstrakten abgeleiteten Klassen.
- Implementieren Sie nun in den nicht-abstrakten abgeleiteten Klassen die Methode `fields()`. Werte, die in der entsprechenden Klasse nicht verfügbar sind oder nicht belegt sind, sind im zurückgelieferten Array mit einem leeren String zu belegen.

Abschließende Bemerkungen

Sie haben nun alle Teilaufgaben dieses Aufgabenblatts gelöst, und die Klassen Ihrer Implementierung bilden die folgende Hierarchie.



Pflege der Unit-Tests, Testen abstrakter Klassen (freiwillig)

Falls Sie im Rahmen der Vorführaufgabe 6 bereits Unit-Tests für die dort noch konkrete Klasse `AudioFile` geschrieben haben, müssen Sie diese nun im Projekt der Aufgabe 7 umschreiben, da Sie von der nun abstrakten Klasse `AudioFile` keine Instanzen mehr bilden können.

Da die alten Unit-Tests Funktionalität der Klasse `AudioFile` testen und diese Funktionalität immer noch in der abstrakten Basisklasse implementiert ist, wäre es nicht adäquat, die Unit-Tests so umzuformulieren, dass Objekte der konkreten Klassen `TaggedFile` oder `WavFile` genutzt werden. Die Unit-Tests der Klasse `AudioFile` sollen unabhängig von diesen Klassen bleiben.

Als Abhilfe führt man im gleichen Verzeichnis, in dem die Unit-Test Klassen liegen, eine konkrete Klasse `DummyAudioFile` ein, die direkt von der abstrakten Klasse `AudioFile` abgeleitet ist. Alle in der abstrakten Klasse geforderten Methoden implementiert man in der Klasse `DummyAudioFile` mit trivialen Rümpfen.

Dann schreibt man die Unit-Tests entsprechend um. Statt Objekte der Klasse `AudioFile` zu instantiieren, nutzt man in den Unit-Tests nun Objekte der Klasse `DummyAudioFile`.

Ein entsprechender Umbau empfiehlt sich auch für die Unit-Tests gegen Funktionalität der Klasse `SampledFile`. Dorthin haben wir ja Methoden der Klasse `TaggedFile` verschoben, damit sie auch für Objekte der Klasse `WavFile` zur Verfügung stehen. Beispielsweise könnte man die Unit-Tests für die Methode `SampledFile.timeFormatter()` in eine Klasse `UTestSampledFile` verschieben.

In einem analogen Vorgehen leitet man eine konkrete Klasse `DummySampledFile` von `SampledFile` ab. Die alten Tests, die Objekte der Klasse `TaggedFile` genutzt haben, schreibt man so um, dass Objekte der Klasse `DummySampledFile` verwendet werden. Damit sind dann die Tests der Funktionalität der Klasse `SampledFile` wieder unabhängig von den Tests der Klasse `TaggedFile`.

Hinweise zur Abnahme Ihrer Implementierung der Vorführaufgabe 07

Im Zuge der Bearbeitung aller Teilaufgaben dieses Blattes haben Sie im Unterverzeichnis `src` die Klassen `AudioFile.java` , `SampledFile.java`, `TaggedFile.java` und `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: `VA07`
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.