Technische Hochschule Ingolstadt
Fakultät für Elektrotechnik und Informatik

Vorführaufgabe 8: die Abspielliste

Ziel der Vorführaufgabe 8 ist die Implementierung einer Abspielliste (Play-Liste) zur Verwaltung der abzuspielenden Audiodateien. Wir werden hierfür die Klasse `PlayList` implementieren.

Neben den offensichtlich dafür benötigten Methoden zum Speichern und Löschen von Einträgen in der Liste möchten wir auch zwei unterschiedliche Modi für das Abspielen der Lieder (sequentielle und zufällige Reihenfolge) und eine Möglichkeit zur direkten Positionierung in der Liste (Wahl des aktuellen Liedes) anbieten.

Entwurfsentscheidung

Bevor wir jedoch auf diese zusätzliche Funktionalität eingehen, müssen wir die grundsätzliche Entscheidung treffen, wie wir die Abspielliste modellieren wollen. In objektorientierten Sprachen stehen hier immer zwei wesentliche Varianten zur Debatte: *Komposition* und *Vererbung*.

Variante 1 mit Komposition (has-a) :
Die Klasse `PlayList` nutzt ein Attribut eines Listen-artigen Typs (Array, Vektor, verkette Liste, …) zum Speichern der Abspielliste.
Die in `PlayList` zu implementierenden Methoden zum Speichern und Löschen delegieren die Arbeit einfach an die entsprechenden Methoden des als Attribut gespeicherten Listen-Objekts. Zusätzlich implementiert die Klasse `PlayList` die oben angesprochene Funktionalität für Abspielmodi und Positionierung.

Variante 2 mit Vererbung (is-a):
Die Klasse `PlayList` wird *abgeleitet* von einem Listen-artigen Typ und *erbt* somit bereits die Methoden zum Speichern und Löschen. Zusätzlich implementiert die Klasse `PlayList` die oben angesprochene Funktionalität für Abspielmodi und Positionierung.

Beide Varianten haben sowohl Vorteile als auch Nachteile:

Komposition ist flexibler als Vererbung, da

- die Zusammensetzung der Komponenten auch zur Laufzeit noch geändert werden können
- keine Auflagen bezüglich eines zur Basisklasse passenden Verhaltens entstehen
  Thema: Liskow'sches Substitutionsprinzip (LSP)

Komposition ist im allgemeinen aber aufwendiger zu implementieren als Vererbung, da die Klasse alle Methoden ihrer Schnittstelle selbst implementieren muss. Es gibt nichts zu erben.

Je größer oder flexibler das Software-Projekt werden soll, desto eher entscheidet man sich für die Komposition.

Da unser Media-Player eine relativ einfache Software-Struktur hat und auch keine komplizierten Plugin-Mechanismen zur Verfügung stellen soll, entscheiden wir uns für die einfachere, dafür aber weniger flexible Variante 2 der Vererbung.

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 7 implementiert haben, und geben Sie dem Klon einen neuen Namen. Wenn Sie zB. das Projekt der Aufgabe 7 im Projektverzeichnis `MediaPlayer_GdPII_VA07` gespeichert haben, legen Sie nun eine Kopie des Projekts an und geben der Kopie den Namen `MediaPlayer_GdPII_VA08`[1].

Löschen Sie sodann alle Dateien im Unterverzeichnis `cert`. Das Unterverzeichnis beinhaltet aufgrund des Klonens noch die Server-Tests der Aufgabe 7. Laden Sie nun die zur Aufgabe 8 gehörigen Zip-Archive `cert.zip` und `build.zip` aus dem Intranet (Seite für VA08) und entpacken Sie beide im Wurzelverzeichnis Ihres Projektes. Das neue Archiv `cert.zip` enthält die Server-Tests für die VA08. Das Archiv `build.zip` enthält wieder die Ant-Build-Umgebung für die VA08. Die Dateien im Archiv `build.zip` werden im Wurzelverzeichnis ausgepackt und überschreiben die alten Dateien der Aufgabe 7.

Für diese Vorführaufgabe benötigen Sie auch wieder Audio-Dateien, die Sie im Intranet auf der Seite der Aufgabe 8 im Archiv `audiofiles.zip` finden. Die im Archiv enthaltenen Dateien entsprechen genau den Dateien aus der Aufgabe 7, die Sie durch das Klonen bereits kopiert haben. Sie müssen also das Archive `audiofiles.zip` der Aufgabe 8 nicht unbedingt neu laden und entpacken.

Hinweis 1: 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 8 finden.

Auch diese Dateien habe Sie bereits durch das Klonen kopiert. Je nach dem aber, wie Sie bei Aufgabe 7 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.

Hinweis 2: Es gibt Anarchisten, die die Dateien .project und .classpath in einem guten Editor (also nicht notepad.exe) direkt bearbeiten. Man kann ein Projekt auch mit dem Explorer oder direkt in der MinGW-Shell klonen (cp -r). Dafür aber unbedingt Eclipse vorher herunterfahren! Danach dann die Datei .project anpassen (Projektname) und in der Datei .classpath evtl. Bibliotheken hinzufügen/ändern. Danach Eclipse einfach wieder starten und das Projekt im Package-Explorer importieren (via “Import“ → “General“ → “Existing Projects into Workspace“ → “Browse“)
Dabei die Option “Copy Project into Workspace“ nicht setzen!
Noch besser ist es, die Projekte gar nicht erst im Workspace anzulegen. Dazu beim Anlegen des neuen Projekts die Option “Use Default Location“ deaktivieren und einen zum Workspace externen Ordner angeben.
Aber so etwas machen natürlich nur Anarchisten, die Spaß am Risiko haben.
Der normale Weg ist die Benutzung der Eclipse-internen Funktionen!

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

Parametrisierte Klassen (generische Datenstrukturen)

Wie eingangs erwähnt, möchten wir durch die Klasse `PlayList` eine Abspielliste implementieren, in der wir beliebige Objekte der Klasse `AudioFile` abspeichern können. Genauer gesagt, möchten wir Objekte der konkreten Klassen `TaggedFile` oder `WavFile` oder anderer konkreter Klassen abspeichern. Wichtig ist dabei nur, dass die Klassen von `AudioFile` abgeleitet sind. Objekte anderer Klassen, wie z.B. Objekte der Klassen `String`, `GiroKonto` oder `Fahrzeug`, haben hingegen in einer Abspielliste für Audio-Dateien nichts verloren.

In Java gibt es zur Umsetzung solcher Typ-Einschränkungen das Konzept der *parametrisierten Klassen*, das unter anderem auch zur Implementierung von sogenannten *Container-Klassen* verwendet wird. Man kann damit Datenstrukturen wie Listen, Bäume, Mengen oder dergleichen implementieren und gleichzeitig durch die Parametrisierung der Klassendefinition mit einer *Typ-Variablen* die Festlegung der konkreten Klasse des Elementtyps noch offen lassen. Wegen der noch nicht gefallenen Entscheidung bzgl. des Typs (Klasse) der in der Datenstruktur zu speichernden Elemente spricht man von *generischen* bzw. *parametrisierten Datentypen* (generics).

In Java gibt es mehrere Container-Klassen, die Listen-artige parametrisierte Datenstrukturen implementieren (parametrisiertes Interface `List<T>`). Wir werden für unsere Klasse `PlayList` die konkrete Datenstruktur `LinkedList<T>` verwenden, die eine doppelt verkettete Liste realisiert und das Interface `List<T>` implementiert.

Eine mittels der Typ-Variablen T parametrisierte Datenstruktur, etwa `LinkedList<T>`, muss zuerst mit einem konkreten Typ instantiiert werden (etwa `Fahrzeug`), bevor Objekte erzeugt werden können. Im Beispiel wäre so ein Objekt eine verkettete Liste über dem Typ `Fahrzeug`.

Die Java-Syntax für eine *Objektdefinition* sieht, passend für unser Beispiel, wie folgt aus:

```
List<Fahrzeug> liste = new LinkedList<Fahrzeug>();
```

Hiermit wird ein Objekt `liste` erzeugt, das eine doppelt verkettete Liste von Fahrzeugen darstellt. Es ist im Allgemeinen von Vorteil, auf der linken Seite den Typ des Interfaces zu verwenden.

Die *Ableitung* einer Klasse von einer parametrisierten Datenstruktur wie `LinkedList<T>`, wobei der Typ konkret instantiiert wird, sieht in Java wie folgt aus:

```
public class FahrzeugList extends LinkedList<Fahrzeug> {
   ...
}
```

Hiermit wird eine Klasse `FahrzeugList` definiert, die abgeleitet wird von der Klasse der verketteten Listen, die nur noch Objekte vom Typ `Fahrzeug` speichern können. Zuerst wird also die parametrische Datentstruktur `LinkedList<T>` instanziert zu `LinkedList<Fahrzeug>`, dann wird von dieser Klasse abgeleitet.

Teilaufgabe a (Ableitung PlayList von LinkedList<T>)

Legen Sie nun zunächst eine Klasse `PlayList` an. Diese Klasse soll von `LinkedList<T>` abgeleitet sein und so parametrisiert sein, dass sie nur Objekte vom Typ `AudioFile` aufnehmen kann.

Den Rumpf der Klasse `PlayList` können Sie hierfür zunächst leer lassen. Falls Eclipse Sie mit folgender Warnung beglückt:

`The serializable class PlayList does not declare a static final serialVersionUID field of type long`

können Sie diese durch Voranstellen von `@SuppressWarnings("serial")` zum Verstummen bringen.

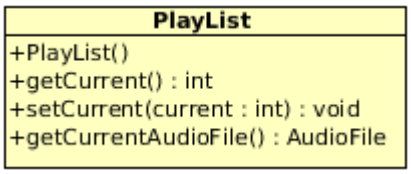
Anforderungen an das zusätzliche Verhalten von PlayList

In der Teilaufgabe a haben wir durch die Ableitung der Klasse `PlayList` von `LinkedList<T>` erreicht, dass `PlayList` eine bestimmte Art von Liste ist (is-a). Somit verfügt die Klasse PlayList über allen Methoden ihrer Basisklasse[2].

Darüber hinaus soll eine `PlayList` aber noch weitere Eigenschaften und Verhalten haben, die nicht von `LinkedList<T>` geerbt werden können.

In den nun folgenden Teilaufgaben werden Sie die noch fehlende Funktionalität der Play-Liste implementieren.

Im nebenstehenden UML-Diagramm ist die Schnittstelle der Klasse in einer ersten Version dargestellt. Später werden noch weitere Methoden hinzugefügt.



Klasse PlayList: Erste Version

Teilaufgabe b (Aktuelle Position; setCurrent() und getCurrent())

Die Klasse `PlayList` soll die „aktuelle Position“ in der Abspielliste in einem Attribut verwalten. Entsprechend der ererbten Eigenschaften einer `LinkedList` bietet sich hierfür ein Attribut vom Typ `int` an.

Statten Sie die Klasse mit einem geeigneten Attribut aus (Index für die Abspielposition), und setzen Sie die Sichtbarkeit des Attributs so restriktiv wie möglich. Fügen Sie zudem noch Setter (`setCurrent`) und Getter (`getCurrent`) hinzu.

Fügen Sie des weiteren einen Konstruktor `PlayList()` hinzu, der das Attribut geeignet initialisiert.

Hinweis: um die Frage, ob das Attribut tatsächlich eine valide Position in der Liste bezeichnet, kümmern wir uns erst in der nächsten Teilaufgabe (`getCurrentAudioFile`)

Teilaufgabe c (das aktuelle Lied; getCurrentAudioFile())

Die Klasse `PlayList` soll die Methode `getCurrentAudioFile()` bereitstellen, die dasjenige `AudioFile` zurückgibt[3], das an der aktuellen Position in der Liste gespeichert ist.

Nun kann es aber vorkommen, dass das interne Attribut, das die aktuelle Position speichert, eine Position angibt, die es in der Liste gar nicht gibt (invalide Position).

Das ist zum Beispiel dann der Fall, wenn die Liste leer ist, oder wenn eine zuvor valide Abspielposition durch Löschung von Liedern aus der Liste plötzlich auf eine Position zeigt, die es nicht mehr gibt.

Implementieren Sie die Methode `getCurrentAudioFile()` und sichern Sie die Methode so ab, dass im Fall einer invaliden aktuellen Position statt einer Referenz auf ein Objekt der Klasse `AudioFile` die Referenz `null` zurückgegeben wird.

2 Studieren Sie die API-Dokumentation der Klasse `LinkedList: add, remove, get, isEmpty,...`

3 Genauer gesagt: eine Referenz auf ein Objekt der Klasse `AudioFile`

Unit-Tests für getCurrentAudioFile()

Da die Methode `getCurrentAudioFile()` bereits ein nicht-triviales Verhalten hat, sollte sie durch Unit-Tests abgesichert werden. Im folgenden werden Ihnen beispielhaft einige Unit-Tests präsentiert. Fügen Sie diese zu Ihrem Projekt im Verzeichnis `tests` hinzu und schreiben Sie selbst zusätzliche Unit-Tests, die weitere Fehlermöglichkeiten der bisher implementierten Funktionalität aufdecken können.

```
public class UTestPlayList2 {
    @Test
    public void test_getCurrentAudioFile_01() throws Exception {
        PlayList pl = new PlayList();
        assertEquals("Wrong current AudioFile", null, pl.getCurrentAudioFile());
    }

    @Test
    public void test_getCurrentAudioFile_02() throws Exception {
        PlayList pl = new PlayList();
        TaggedFile tf0 = new TaggedFile("audiofiles/Eisbach Deep Snow.ogg");
        pl.add(tf0);
        pl.setCurrent(10); // Wrong index; however, the setter is not checked
                           // However, getCurrentAudioFile() is checked
        assertEquals("Wrong current AudioFile", null, pl.getCurrentAudioFile());
    }

    @Test
    public void test_getCurrentAudioFile_04() throws Exception {
        PlayList pl = new PlayList();
        TaggedFile tf0 = new TaggedFile("audiofiles/Eisbach Deep Snow.ogg");
        TaggedFile tf1 = new TaggedFile("audiofiles/Rock 812.mp3");
        pl.add(tf0);
        pl.add(tf1);
        pl.setCurrent(1);
        assertEquals("Wrong current AudioFile", tf1, pl.getCurrentAudioFile());

        pl.remove(0); // Removing the first element invalidates current index
                      // pointing at position 1. Now, list is too short.
        assertEquals("Wrong current AudioFile", null, pl.getCurrentAudioFile());
    }
```

Teilaufgabe d (changeCurrent())

Die Klasse `PlayList` soll die Methode `changeCurrent()` bereitstellen, die den Index für die aktuelle Abspielposition in der Liste um eine Position weiter schaltet. Falls der Index bereits auf das letzte Lied in der Liste zeigt, soll der Index an den Anfang zurück auf die Position des ersten Liedes (Position 0) gesetzt werden. Des weiteren soll ein gerade invalider Index durch Aufruf der Methode `changeCurrent()` ebenfalls auf Position 0 gesetzt werden.
Die Methoden `setCurrent(), getCurrent()` und `changeCurrent()` geben der Klasse `PlayList` das Verhalten einer Ringstruktur.

Implementieren Sie die Methode `changeCurrent()` nach obigen Vorgaben.

Unit-Tests für changeCurrent()

Sichern Sie Ihre Implementierung durch folgenden Unit-Test ab.

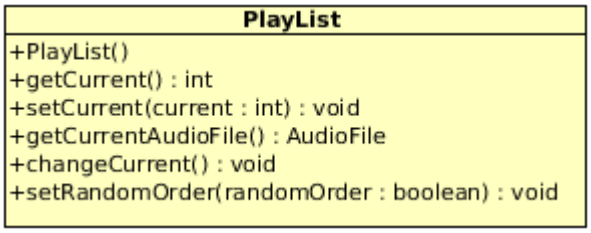
```
@Test
public void test_changeCurrent_01() throws Exception {
    PlayList pl = new PlayList();
    TaggedFile tf0 = new TaggedFile("audiofiles/Eisbach Deep Snow.ogg");
    TaggedFile tf1 = new TaggedFile("audiofiles/tanom p2 journey.mp3");
    TaggedFile tf2 = new TaggedFile("audiofiles/Rock 812.mp3");
    pl.add(tf0);
    pl.add(tf1);
    pl.add(tf2);
    pl.setCurrent(0);
    assertEquals("Wrong current index", 0, pl.getCurrent());
    pl.changeCurrent();
    assertEquals("Wrong change in current index", 1, pl.getCurrent());
    pl.changeCurrent();
    assertEquals("Wrong change in current index", 2, pl.getCurrent());
    pl.changeCurrent();
    assertEquals("Wrong change in current index", 0, pl.getCurrent());
}
```

Fügen Sie weitere Tests hinzu, etwa zum Testen der Wirkung von changeCurrent() auf einen invaliden Positionsindex.

Teilaufgabe e (setRandomOrder())

Die Klasse `PlayList` soll für das Abspielen von Liedern zwei Abspielmodi bereitstellen. Zum einen das Abspielen in sequentieller Reihenfolge, zum anderen aber auch das Abspielen in einer zufälligen Reihenfolge. Der Modus soll durch die Methode `setRandomOrder()` eingestellt werden können. Der Aufruf von `setRandomOrder(true)` stellt zufällige Wiedergabe ein, der Aufruf von `setRandomOrder(false)` stellt normale sequentielle Wiedergabe ein.
Das nebenstehende UML-Diagramm zeigt einen Überblick über die derzeit gefordert Schnittstelle der Klasse `PlayList`.

Fügen Sie der Klasse `PlayList` ein Attribut mit geeignetem Typ hinzu, in dem der Abspielmodus gespeichert wird. Wie üblich sollten Sie die Sichtbarkeit des Attributs möglichst restriktiv setzen. Statten Sie die Klasse zusätzlich mit dem Setter `setRandomOrder(boolean)` aus, mit dem der Modus von außen gesetzt werden kann. Sehen Sie auch eine geeignete Initialisierung des Attributs vor (bei der Definition oder im Konstruktor).



PlayList: zweite Version

Teilaufgabe f (Implementierung der zufälligen Reihenfolge)

Um eine Zufallswiedergabe zu realisieren, gibt es verschiedene denkbare Wege. Ein einfacher Weg besteht darin, die zufällige Reihenfolge **nicht** etwa über den Positionsindex der Liste zu organisieren, sondern vielmehr einfach zum richtigen Zeitpunkt die Elemente der Liste zufällig anzuordnen. Wenn man dann die Lieder in der *in Unordnung gebrachten* Liste der Reihe nach von Anfang bis Ende abspielt, hat man von außen den Eindruck, als ob die Lieder in zufälliger Reihenfolge abgespielt würden.

Zur Lösung der vorliegende Teilaufgabe sind also zwei Fragen zu klären:

1. wie bringt man eine Liste in eine zufällige Unordnung?
2. wann ist der oben angesprochenen *richtige Zeitpunkt* dafür gekommen?

Die Antwort auf die erste Frage liefert die statische Methode `shuffle` der Klasse `Collections`[4]. Diese Methode erwartet als Parameter ein Objekt, welches das Interface `List` implementiert, und ordnet die Elemente dieses Objektes in einer pseudo-zufälligen Reihenfolge neu an.
Da unsere Klasse `PlayList` von `LinkedList` erbt, und diese das Interface `List` implementiert, können zum Beispiel mit dem Aufruf `Collections.shuffle(myPlayList);` die in einer `PlayList myPlayList` abgelegten Lieder am Platz (in place) entsprechend umsortiert werden. Am Platz bedeutet, dass keine neue Liste erzeugt wird, sondern dass die bestehende Liste umsortiert wird.

Um den richtigen Zeitpunkt für das *Verwürfeln* (Shuffle) einzustellen, müssen wir den Code der Klasse `PlayList` an zwei Stellen anpassen.

- In der Setter-Methode `setRandomOrder(randomOrder)` rufen wir die Methode `Collections.shuffle()` genau dann auf, wenn das Argument `randomOrder` den Wert `true` hat. Wer Unordnung haben will, soll sie auch gleich bekommen.
- In der Methode `changeCurrent()`, wenn aufgrund des Weiterschaltens der aktuellen Position ein Sprung an den Anfang der Liste ansteht (Rotation). Wenn der Zufallsmodus gesetzt ist, setzten wir den Index nicht nur an den Anfang der Liste zurück, sondern bringen die Liste zusätzlich in eine neue Unordnung. Diese Unordnung reicht aus für einen vollen Durchgang durch die Liste.

Unit-Tests für den zufälligen Abspielmodus

Die Wirkung der gerade implementierten Funktionalität lässt sich mit folgendem Unit-Tests veranschaulichen. Einen richtige Test findet Sie in `cert/PlayListTest.testSetGetChangeCurrent()`

```
@Test
public void test_changeCurrent_02() throws Exception {
    PlayList pl = new PlayList();
    TaggedFile f0 = new TaggedFile("audiofiles/Eisbach Deep Snow.ogg");
    TaggedFile f1 = new TaggedFile("audiofiles/tanom p2 journey.mp3");
    TaggedFile f2 = new TaggedFile("audiofiles/Rock 812.mp3");
    WavFile f4 = new WavFile("audiofiles/wellenmeister - tranquility.wav");
    pl.add(f0); pl.add(f1); pl.add(f2); pl.add(f4);
    pl.setRandomOrder(true);
    // Note: Only the content of the list is shuffled ;-)
    for (int i = 0; i < 5 * pl.size(); i++) {
        System.out.printf("Pos=%d Filename=%s\n",pl.getCurrent(),
                pl.getCurrentAudioFile().getFilename());
        assertEquals("Wrong current index", i % pl.size(),pl.getCurrent());
        pl.changeCurrent();
        if (pl.getCurrent() == 0) System.out.println("");
    }
}
```

4 `import java.util.Collections;`

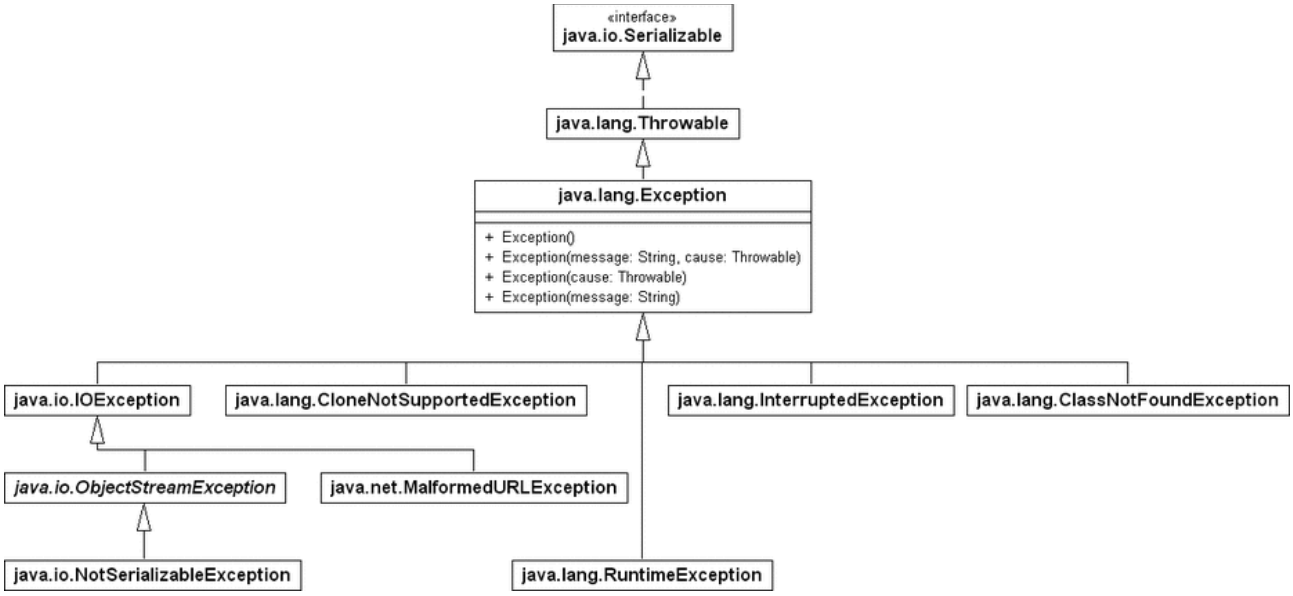
Vorbereitende Erklärung zu Exceptions

Damit unsere Abspiellisten wieder verwendet werden können, soll es möglich sein, diese als Datei abzuspeichern und zu einem späteren Zeitpunkt auch wieder zu laden. Hierzu muss über entsprechende Klassen der Java-Bibliothek auf das Dateisystem zugegriffen werden.

Beim Zugriff auf das Dateisystem muss immer damit gerechnet werden, dass Fehler auftreten. So kann beim Öffnen einer Datei zum Lesen der Pfadname der Datei falsch angegeben sein, oder die Datei darf vom Benutzer nicht gelesen werden. Beim Schreiben kann die Berechtigung für das Anlegen von Dateien fehlen, oder das Dateisystem kann voll (Platte voll) oder beschädigt sein (Dateisystem korrupt).

In Java werden beim Auftreten von Fehlern bei Operationen auf dem Dateisystem Ausnahmen (Exceptions) der Klasse `IOException` ausgelöst. Diese Klasse gehört zu den sogenannten *kontrollierten Ausnahmen*, die vom Java-Compiler kontrolliert werden und vom Programmierer explizit behandelt werden müssen. Daneben gibt es noch die Klasse `RuntimeException`, die für die *nicht-kontrollierten Ausnahmen* steht. Diese werden vom Compiler nicht so streng geprüft und *können* vom Programmierer behandelt werden, müssen aber nicht.

Das nachfolgende Bild ist dem Werk von Christian Ullenboom[5] entnommen und zeigt einen Ausschnitt aus der Exception-Hierarchie von Java:



Will man in einer Java-Applikation die Behandlung von Exceptions (vorerst) einfach halten, kann man an den Stellen, an denen mit kontrollierten Exceptions zu rechnen ist, diese Exceptions auffangen und nach einer Umwandlung in nicht-kontrollierte Exceptions „weiter schmeißen“ (re-throw). Dieses Vorgehen erleichtert zwar die Behandlung von Exceptions, sollte aber nur eine vorübergehende Lösung sein, bis man die Ausnahmen auch im umschließende Code ordentlich behandelt. Durch das Umschreiben verliert man nämlich die zuvor gegebene Granularität der Fehlermeldungen.
Das Code-Muster für die oben angesprochene Umschreibung der Exceptions findet sich in den nachfolgenden Beispielen. Es sieht wie folgt aus:

```
try {
      // kritischer Code, der kontrollierte IOException auslösen kann
} catch (IOException e) {
      throw new RuntimeException (e)  // re-throw einer unkontrollierten
}                                     // Ausnahme
```

5 Java ist auch eine Insel, 9.Auflage: http://openbook.galileocomputing.de/javainsel/

Muster-Code zum Schreiben von Textdateien

Bevor wir uns nun an das Abspeichern und Lesen von Play-Listen heranwagen, soll anhand eines Beispiels gezeigt werden, wie man in Java eine Text-Datei korrekt beschreibt. Die meisten von Ihnen werden zu diesem Zeitpunkt noch nicht mit den IO-Klassen von Java und speziell mit dem Mechanismus der Exceptions vertraut sein. Daher ist ein einführendes Beispiel angebracht.

Der nachfolgende Unit-Test zeigt ihnen exemplarisch, wie Sie in Java eine Text-Datei `file.txt` mit drei Strings beschreiben können, einer pro Zeile. Der Code wird im Anschluss erklärt.

```
327     @Test // @Ignore
328     public void test_WriteLinesToFile_01() throws Exception {
329         FileWriter writer = null;
330         // Write lines to file using line.separator
331         String fname = "file.txt";
332         String linesep = System.getProperty("line.separator");
333         // String linesep = "\n";
334         try {
335             // Create a FileWriter
336             writer = new FileWriter(fname);
337             // Since this worked out we know that the file is writable
338             // Write some strings to file
339             writer.write("Line1" + linesep);
340             writer.write("Line2" + linesep);
341             writer.write("Line3" + linesep);
342         } catch (IOException e) {
343             throw new RuntimeException(
344               "Unable to write to file " + fname + ":" + e.getMessage());
345         } finally {
346             // Close file handle in any case!!
347             try {
348                 writer.close();
349             } catch (Exception e) {
350               // Just swallow any exception caused by the 'finally' block
351             }
352         }
353     }
```

In Zeile 329 wird ein `FileWriter`[6] definiert, der Name der Ausgabedatei `file.txt` wird in Zeile 331 festgelegt. Die nächste Zeile definiert, wie das Ende einer Zeile kodiert werden soll. Die System-Eigenschaft `line.separator` liefert passend zum Betriebssystem den Zeilentrenner. Unter Windows ist das CarriageReturn (`'\r'`) gefolgt von LineFeed (`'\n'`), unter Unix entspricht er nur LineFeed (`'\n'`) und unter MacOs meistens nur CarriageReturn (`'\r'`).

Im try-Block wird der Code ausgeführt, der möglicherweise eine Ausnahme auslöst. Zuerst öffnen wir die Datei in Zeile 336 zum Schreiben und schreiben danach in den Zeilen 339-341 drei Strings in die Datei, Beachten Sie bitte die Verwendung der Hilfsvariablen `linesep`.
Falls eine `IOException`[7] ausgelöst wird, wird diese im catch-Block gefangen und umkodiert in eine `RuntimeException` mit entsprechender verbesserter Fehlermeldung.

Egal, ob nun ein Fehler auftritt (`IOException`) oder nicht, soll in jedem Fall am Schluss die Datei wieder geschlossen werden, damit die Puffer des Dateisystems sauber geleert und auf Platte geschrieben werden. Dies erreicht man durch den finally-Block, in dem in Zeile 348 die Datei geschlossen wird. Das Schließen einer Datei kann ebenfalls einen Fehler auslösen, und daher ist auch hier eine Verwendung von try-catch notwendig.

6 `import java.io.FileWriter`

7 `Import java.io.IOException`

Nachdem Sie diesen Unit-Test ausgeführt haben, können Sie sich die neu erzeugte Text-Datei `file.txt` mit einem Editor anschauen. Bitte verwenden Sie hierzu einen vernünftigen Editor und nicht etwas Notepad.exe. Unter Windows stehen hier unter anderem Notepad++ oder UltraEdit zur Debatte. Sie können aber natürlich auch in einer MinGW/msys-Shell den vim verwenden.

Bitte betrachten Sie zusätzlich die Hex-Kodierung der Datei, entweder in einem Hex-Dump-Modus Ihres Editors oder aber durch Ausgabe des Hex-Dumps in der MinGw-Shell ('`xxd file.txt`').

Danach aktivieren Sie die Zeile 333 (Zeile 332 deaktivieren) und wiederholen den Test. Je nach Betriebssystem werden Sie einen Unterschied feststellen oder nicht. Experimentieren Sie auch mit dem Zeilntrenner '`\r`'. Falls möglich, führen Sie den Test auch auf unterschiedlichen Betriebssystemen aus.

Muster-Code zum Lesen von Textdateien

Der nachfolgende Unit-Test zeigt ihnen, wie Sie in Java eine Text-Datei einlesen können.

```
355     @Test // @Ignore
356     public void test_ReadLinesFromFile_01() throws Exception {
357         String fname = "file.txt";
358         Scanner scanner = null;
359         String line;
360         try {
361             // Create a Scanner
362             scanner = new Scanner(new File(fname));
363             // Since this worked out we know that the file is readable
364             // Read line by line
365             int i = 1;
366             while (scanner.hasNextLine()) {
367                 line = scanner.nextLine();
368                 System.out.println("Got line " + i + "|" + line + "|");
369                 i++;
370             }
371         } catch (IOException e) {
372             // e.printStackTrace();
373             throw new RuntimeException(e);
374         } finally {
375             // Close file handle in any case!!
376             try {
377                 scanner.close();
378             } catch (Exception e) {
379             // Just swallow any exception caused by the 'finally' block
380             }
381         }
382     }
```

Der Aufbau ähnelt dem Code für das Schreiben von Text-Dateien. In Zeile 357 wird der Name der zu lesenden Textdatei festgelegt. In Zeile 358 wird ein Objekt der Klasse `Scanner`[8] definiert. In Zeile 362 öffnen wir die Datei zum Lesen (Objekt der Klasse `File`[9]) und übergeben das Objekt dem Scanner zu Analyse. Die Methode `hasNextLine()` prüft, ob noch weitere Eingabezeilen vorhanden sind, und `nextLine()` liefert die aktuelle Zeile als String zurück.
Für gewöhnlich würde man dann nach dem Lesen der aktuellen Zeile den erhalten String weiter mit Hilfe von Methoden der Klasse `String` analysieren, z.B mit `String.matches()`.

8 `import java.util.Scanner`

9 `import java.io.File`

Erklärung zu Kodierung der Play-Listen in Dateien im M3U-Format

Wir werden zur Speicherung unserer Abspiellisten das offene M3U-Format[10] verwenden. Im einfachsten Fall ist eine in diesem Format kodierte Abspielliste eine Liste von durch Zeilenumbrüchen getrennten Pfadnamen der einzelnen Audio-Dateien.
Sofern eine Zeile mit einem '#' beginnt, handelt es sich um eine Kommentarzeile, die bei der Auswertung übergangen werden muss. Leere Zeilen (White-Space) sind auch erlaubt und müssen ebenfalls ignoriert werden.

Beispiel für eine im M3U-Format gespeicherte Play-Liste:

```
# My best songs

Alternative\Band – Song.mp3
Classical\Other Band - New Song.mp3
Stuff.mp3
D:\More Music\Foo.mp3
..\Other Music\Bar.mp3
```

Fleißaufgabe!
Spezifizieren Sie die Syntax von M3U-Dateien in EBNF

Teilaufgabe g (saveAsM3U())

Implementieren Sie nun in der Klasse `PlayList` eine Methode

```
public void saveAsM3U(String pathname)
```

welche die im Objekt gespeicherte Play-Liste in einer Datei mit Namen `pathname` im M3U-Format abspeichert. Orientieren Sie sich dabei am oben angegebenen Muster zum Schreiben von Textdateien.

Fleißaufgabe!
Speichern Sie dabei zusätzlich Ihren Namen und das Tagesdatum nebst Uhrzeit als Kommentare

Teilaufgabe h (Klasse AudioFileFactory)

Eine noch ausstehende Aufgabe ist das Laden einer Play-Liste aus einer M3U-Datei. Neben dem reinen Lesen und Analysieren der einzelnen Zeilen müssen wir, nachdem wir den Inhalt einer Zeile als Kandidat für einen Pfadnamen einer Audio-Datei klassifiziert haben, immer die folgende Frage beantworten:

- welche konkrete Klasse soll das zu instantiierende Objekt haben ?

Derzeit können wir konkret Objekte der Klassen `TaggedFile` oder `WavFile` erzeugen. Die Entscheidung macht man am besten an der Endung des Dateinames fest, der Teil des Pfadnames der Audio-Datei ist. Im Fall der Endungen `.mp3` oder `.ogg` möchten wir ein Objekt der Klasse `TaggedFile` erzeugen, im Fall der Endung `.wav` ein Objekt der Klasse `WavFile.`

Des weiteren ist damit zu rechnen, dass irgendwann eine weiterer Unterklasse von `AudioFile` hinzukommt. Kurzum, die Logik für die Entscheidung, welchen Typ das neue Objekt haben soll, ist nicht ganz trivial, und sie muss vielleicht auch irgendwann geändert werden. In einem solchen Fall lagert man diese Logik zusammen mit dem Akt der Objekterzeugung am besten in eine eigene Klasse aus.

Für den objektorientierten Entwurf (object oriented design) und die fortgeschrittene objektorientierte Programmierung gibt es für diesen Zweck spezielle Entwurfsmuster (design patterns),

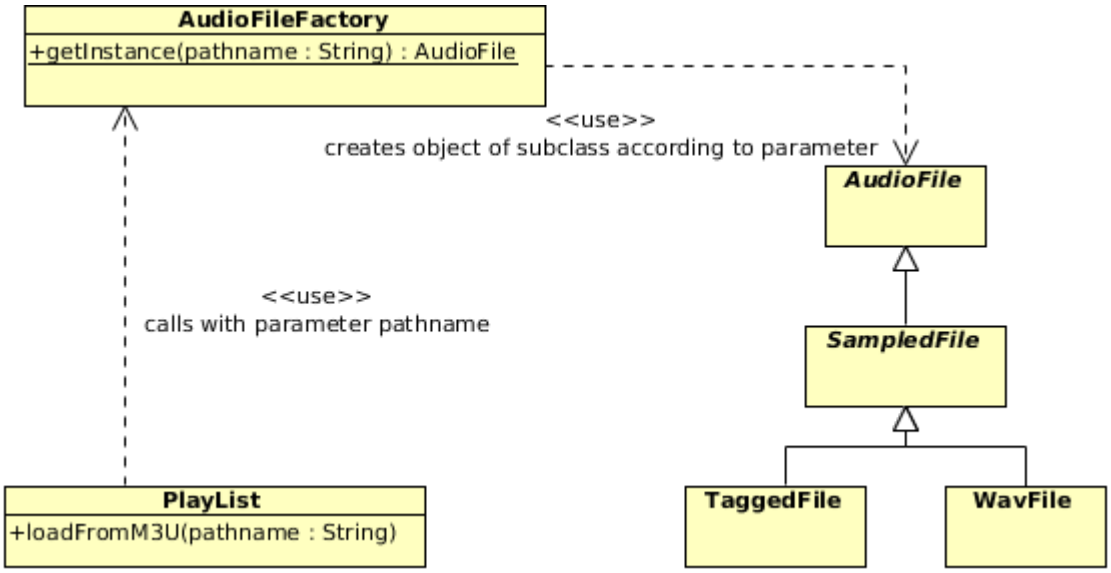
10 Siehe http://en.wikipedia.org/wiki/M3U

die sogenannten Erzeugungsmusters (creational patterns)[11]

Eines dieser Entwurfsmuster ist die sogenannte Fabrik-Methode, die wir in der speziellen Ausprägung der parametrisierten Fabrik-Methode (parameterized factory method) zum Einsatz bringen werden.

Die parametrisierte Fabrik-Methode ist eine statische Methode einer Klasse, deren Aufgabe einzig und allein darin besteht, Objekte anderer Klassen zu bauen. Welcher Klasse die zu bauenden Objekte angehören, entscheidet die parametrisierte Fabrik-Methode anhand eines Parameters, der ihr als Argument mitgegeben wird. Die statische Methode heißt sehr oft `getInstance()`.

Das folgende Klassendiagramm verdeutlicht noch einmal die Funktionsweise der parametrisierten Fabrik-Methode. Im Diagramm zeigen wir das Entwurfsmuster gleich im hier vorliegenden Kontext.



Im Auftrag der noch zu schreibenden Methode `loadFromM3U()` erzeugt die statische Methode `AudioFileFactory.getInstance()` anhand des ihr übergebenen Pfadnames der Audio-Datei ein Objekt einer Sub-Klasse von `AudioFile`. Ob ein Objekt der Klasse `TaggedFile` oder `WavFile` erzeugt werden soll, entscheidet die Fabrik-Methode anhand der Dateiendung des im Pfadnamen enthaltenen Dateinamens.

Durch die Auslagerung der Erzeugungslogik in die Fabrik-Methode kann die Klasse `PlayList` die erzeugten Objekte polymorph nutzen. Aus Ihrer Sicht werden von der Fabrik-Methode immer Objekte der Klasse `AudioFile` erzeugt. Man erreicht eine schöne Kapselung.

Legen Sie nun die Klasse `AudioFileFactory` an, die eine Methode

```
public static AudioFile getInstance(String pathname)
```

implementiert. Falls der Dateiname in `pathname` mit `.wav` endet, soll diese Methode ein Objekt der Klasse `WavFile` erzeugen, falls der Dateiname mit `.mp3` oder `.ogg` endet, ein Objekt der Klasse `TaggedFile`. Die Schreibweise (groß, klein oder gemischt) der Dateiendung ist dabei irrelevant. Im Anschluss an die Erzeugung wird das Objekt zurückgegeben. Sollte der potentielle Pfadname nicht über eine bekannte Endung verfügen, soll die folgende Exception von der Fabrik-Methode erzeugt werden.

```
throw new RuntimeException("Unknow suffix for AudioFile: \""+ pathname + "\"");
```

11 Siehe etwa Buch Design Patterns; Gamma,Helm,Johnson,Vlissides; Addison-Wesley; 1995

Unit-Tests für die Fabrik-Methode

Testen Sie Ihre Implementierung der Klasse `AudioFileFactory` durch Unit-Tests der folgenden Bauart.

Tests, die die Erzeugung von Ausnahmen prüfen:

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

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

Tests für die Erzeugung von Objekten der richtigen Sub-Klasse:

```
@Test
public void test_getInstance_03() throws Exception {
    AudioFile af1 = AudioFileFactory
            .getInstance("audiofiles/Eisbach Deep Snow.ogg");
    assertTrue("Expecting object of type TaggedFile",
            (af1 instanceof TaggedFile));

    AudioFile af2 = AudioFileFactory
            .getInstance("audiofiles/wellenmeister - tranquility.wav");
    assertTrue("Expecting object of type WavFile",
            (af2 instanceof WavFile));

    AudioFile af3 = AudioFileFactory.getInstance("audiofiles/special.oGg");
    assertTrue("Expecting object of type TaggedFile",
            (af3 instanceof TaggedFile));
}
```

Teilaufgabe i (Load from M3U)

Jetzt sind wir endlich in der Lage, eine Play-Liste aus einer M3U-Datei einzulesen. Implementieren Sie nun in der Klasse `PlayList` eine Methode

```
public void loadFromM3U(String pathname)
```

die eine Datei mit dem Namen `pathname` einliest. Gemäß der Kodierung von M3U-Dateien sollen dabei Kommentarzeilen und Leerzeilen ignoriert werden. Alle anderen Zeilen sollen als Kandidaten für Pfadnamen von Audio-Dateien behandelt werden.

Nutzen Sie für das Einlesen der M3U-Datei den oben angegebenen Muster-Code zum Lesen von Textdateien. Wenn Sie eine Zeile eingelesen haben, die Ihrer Meinung nach einen Kandidaten für eine Pfadnamen enthält, rufen Sie die Fabrik-Methode auf. Falls diese erfolgreich ein Objekt einer Sub-Klasse von `AudioFile` erzeugen kann, speichern Sie dieses in der Play-Liste.

Hinweise:

- Leeren Sie die bestehende Play-Liste, bevor Sie neue Elemente aus der M3U-Datei einfügen (`this.clear()`)
- Einfügen von Elementen in die Play-Liste macht man mit `this.add()`

Teilaufgabe j (Konstruktor PlayList(pathname))

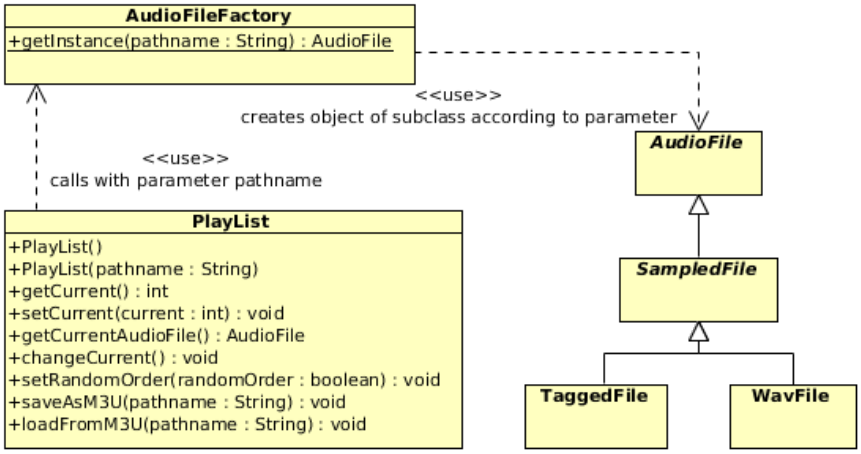
Zu guter Letzt implementieren Sie noch einen weiteren Konstruktor der Klasse `PlayList`, der als Argument den Namen einer M3U-Datei erwartet. Im Rumpf des Konstruktors rufen Sie zunächst den Konstruktor ohne Parameter auf (Initialisierung) und danach die Methode zum Laden der Play-Liste `loadFromM3U(pathname)`.

Unit-Tests für Laden und Speichern von M3U-Dateien

Schreiben Sie einen Unit-Test, der ein Objekt der Klasse `PlayList` erzeugt und mehrere Audio-Dateien darin speichert. Schreiben Sie diese Play-Liste dann mit Hilfe der Methode `saveAsM3U()` in eine M3U-Datei. Kontrollieren Sie die entstandene Text-Datei. Schreiben Sie so dann einen weiteren Unit-Test, der die im vorigen Test erzeugte M3U-Datei mit Hilfe der Methode `loadFromM3U()` wieder in ein Objekt der Klasse `PlayList` einliest.

Zusammenfassung

Durch die Bearbeitung aller Teilaufgaben dieses Aufgabenblattes haben Sie die im folgende UML-Diagramm dargestellten Klassen `PlayList` und `AudioFileFactory` implementiert.



Hinweise zur Abnahme Ihrer Implementierung der Vorführaufgabe 08

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

- `AudioFile.java`
- `AudioFileFactory.java`
- `PlayList.java`
- `SampledFile.java`
- `TaggedFile.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: `VA08`
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.