TD5 : Structuration des codes – Application à la gestion de fichiers mp3

I. Objectifs du TD

Ce TD a pour origine un TD de Peter Sander. Il a été adapté aux nouveaux enseignements.
Les objectifs pédagogiques de ce TD sont :

- Découper un cas d'étude simple, en autonomie, en séparant interaction utilisateur et partie métier et en distribuant les responsabilités sur différentes classes ;
- Mettre en place de la programmation défensive ;
- Utiliser des énumérés ;
- Découvrir le système de fichiers en java ;
- Lire un .jar.

En termes applicatifs, nous allons écrire un ensemble de classes pour nous aider à organiser nos fichiers musicaux stockés sur un ordinateur.
Pendant ce TD **baissez le son de l'ordinateur** et quand ce n'est pas gênant pour les tests, coupez le son !

II. Codes pour lire des fichiers mp3

Vous trouvez ci-dessous des codes que vous aurez à utiliser pendant ce TD.

II.1 Codes à télécharger et installer

Q.1. Téléchargez les codes du lecteur de MP3

Q.2. Pour voir le contenu du .jar

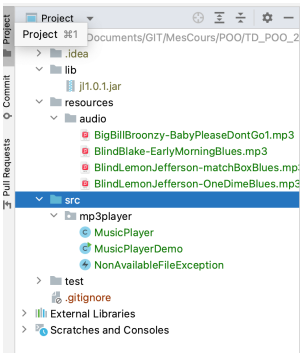
```
$ jar tf mp3playerSRC.jar
src/mp3player/
src/mp3player/MusicPlayer.java
src/mp3player/MusicPlayerDemo.java
src/mp3player/NonAvailableFileException.java
test/mp3player/
test/mp3player/MusicPlayerTest.java
lib/jl1.0.1.jar
resources/audio/
resources/audio/BigBillBroonzy-BabyPleaseDontGo1.mp3
resources/audio/BlindBlake-EarlyMorningBlues.mp3
resources/audio/BlindLemonJefferson-OneDimeBlues.mp3
resources/audio/BlindLemonJefferson-matchBoxBlues.mp3

…
```

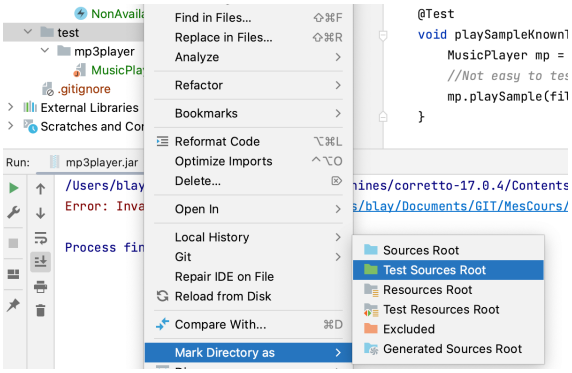
Q.3. Pour extraire les codes par exemple

```
$ jar xf mp3playerSRC.jar
```

Q.4. Placez les codes dans intelliJ en respectant leur structure.



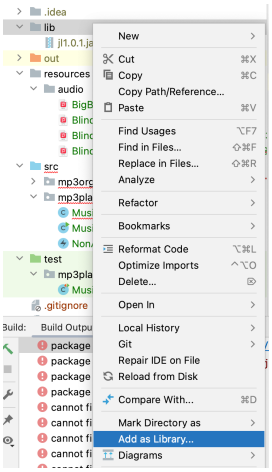
Q.5. Déclarez le répertoire test comme un répertoire de test (rappel, TD1) et bien sûr liez les tests à junit 5.8.



Q.6. Mettre à jour le classpath pour que la bibliothèque *javazoom* soit connue dans notre environnement, cf. ci-dessous.

- Nous ajoutons le répertoire lib comme une « library » sur notre projet, ce qui a pour conséquence de la rendre visible de nos classes (voir ci-dessous).

Javazoom n'est pas une bibliothèque standard de code donc votre environnement ne peut pas résoudre seul cette référence. La bibliothèque vous a été fournie avec les codes, et elle se trouve sous ***lib***.



Q.7. Exécuter le programme principal `mp3player.MusicPlayerDemo`, vous devriez avoir une trace qui ressemble à (les path ont été remplacés par - - -) :

```
/resources/audio/BlindBlake-EarlyMorningBlues.mp3

Placez vos fichiers audio sous : ---/TD5

What file do you want to listen to (to stop type 's')?/resources/audio/BlindBlake-Ea
rlyMorningBlues.mp3

What file do you want to listen to (to stop type 's')?s

What file do you want to listen to for a long time (to stop type 's')?/resources/aud
io/BlindBlake-EarlyMorningBlues.mp3

To stop to listen type (s) or a new file name s
```

Q.8. Un test Junit vous est également donné. Il fonctionne si votre environnement est bien défini, dont notamment le positionnement des fichiers audio.

II.2 Fichiers musicaux

Nous supposons que chaque fichier musical représente une seule piste musicale. Les fichiers d'exemples que nous avons fournis avec le projet contiennent à la fois le nom de l'artiste et le titre de la piste intégrés dans le nom du fichier ; nous utiliserons cette fonctionnalité plus tard.

Par exemple :
BigBillBroonzy-BabyPleaseDontGo1.mp3 :

Big Bill Broonzy *est le nom de l'artiste*
BabyPleaseDontGo est le *titre*.

II.3 Un lecteur de fichiers MP3

La bibliothèque Java standard n'a pas de classe adaptée à la lecture de fichiers MP3, qui est le format audio avec lequel nous voulons travailler. Cependant, de nombreux programmeurs individuels mettent à la disposition d'autres personnes les classes utiles qu'ils développent. Celles-ci sont souvent appelées « bibliothèques tierces» et elles sont importées et utilisées exactement de la même manière que les classes de bibliothèque Java standard.

Q.9. Vous devez comprendre les éléments suivants avant de continuer.

Les trois méthodes de la classe *`MusicPlayer`* que nous allons utiliser sont *`playSample`*, *`startPlaying`* et *`stop`*.

Les deux premières prennent en paramètre la référence vers le fichier audio à jouer. La première lit quelques secondes du début du fichier et revient quand elle a fini de jouer, tandis que la seconde commence sa lecture en arrière-plan puis rend immédiatement le contrôle à l'appelant - d'où la nécessité de la méthode d'arrêt, pour arrêter la lecture.

La classe *`MusicPlayerDemo`* vous donne un exemple d'utilisation de la classe `MusicPlayer`.

Vous êtes prêts à développer !

Pour la suite, vous aurez à la fois à tester automatiquement vos codes (tests unitaires Junit sous le répertoire *test*), et également à tester par des interactions avec votre code (cela simule la connexion à une interface utilisateur) à l'instar de *MusicPlayerDemo.java* qui vous a été donné.

III. (V1) Premier pas vers l'architecture : un organiseur de fichiers musicaux

On veut donc pouvoir stocker des fichiers mp3 (uniquement les références vers les fichiers) puis demander d'en écouter certains. Par souci de simplicité, cette première version de ce projet fonctionnera simplement avec les noms de fichiers des morceaux de musique individuels.

III.1 Des spécifications au code

Pour l'instant, voici les opérations de base à implémenter dans cette version initiale de notre organiseur.
Dans cette version, nous utilisons l'ordre des ajouts comme index des fichiers.

1. Un *MusicOrganizer* permet d'enregistrer des références vers les fichiers, que nous nommerons **piste (track)** .
    a. Ajouter une piste (un fichier)
    b. Ajouter une liste de pistes (fichiers).
2. Il n'a pas de limite prédéterminée sur le nombre de pistes qu'il peut stocker.
3. Il peut nous dire combien de pistes il a enregistré.
4. Il peut donner les pistes qu'il a enregistrées dans l'ordre des pistes.
5. Il nous permet d'écouter une piste et de stopper son écoute. L'index de la piste du point de vue utilisateur commence à 1 : Il écoute la première piste.

Q.10. Proposez une architecture AVANT d'implémenter et justifiez-la.
- ATTENTION, vous devez séparer les entrées/sorties (messages vers l'utilisateur, lecture des lignes de commandes) de la partie métier (organisateur de fichiers de musique). Donc vous n'êtes autorisé à utiliser System.out.println que dans une unique classe, de même pour la lecture de commande au clavier qui doit se trouver dans cette même classe !

Q.11. Créez un package ***mp3Organizer*** dans lequel vous placerez vos codes.

Q.12. Implémentez et testez votre application par des tests unitaires sur vos classes de base
- Pour les étudiants en difficulté avec les ArrayList voir HELP 1.
- Soyez incrémental : définissez une méthode, par exemple une méthode pour ajouter une piste ; testez cette méthode par un test Junit, puis testez la en utilisant votre classe de démonstration. Recommencez avec une autre méthode.

Q.13. Implémentez un "*main*" (*MusicOrganizerDemo*) pour permettre des interactions avec un organiseur.

- Pensez à regarder les codes de la démo qui vous a été donnée *MusicPlayerDemo* pour développer votre propre démo.
- Vous pouvez aussi utiliser un enum pour gérer les commandes.

Q.14. RAPPEL : Aucun code dans la classe MusicOrganizer ne fait référence à une interaction homme machine (println ou read).

III.2 *Protégez* votre code : un organisateur de fichiers musicaux un peu plus robuste

III.2.1 Spécifications à implémenter, si vous ne l'avez pas déjà fait

Q.15. On souhaite éviter les erreurs lors de l'accès à des index erronés. Reprenez votre code et proposez une solution telle que :

- Vous protégez votre code en interdisant un accès à un index erroné dans votre organiseur. En cas d'erreur, **levez une exception spécifique, en signalant qu'il n'y a pas de piste à cet index, et** le message d'erreur indique quelle est la plage valide pour les pistes (entre 1[1] et size()).
- Ne dupliquez pas le code de test de l'index (on ne doit donc trouver le test de l'index qu'une seule fois dans votre code**)** ! (Tip : si besoin pensez à utiliser IntelliJ pour un *extract method checkIndex* par exemple)
- Testez votre code avec des paramètres valides et non valides.
- Votre méthode fonctionne-t-elle toujours lorsque la collection est vide ?

Q.16. Comment gérez-vous le cas où l'on ajoute une piste qui ne correspond pas à un fichier musical, i.e., pas un fichier ou une extension différente de mp3. Dans ce cas, nous considérons qu'il s'agit d'une erreur utilisateur et vous levez l'exception ***IllegalArgumentException*** avec un message d'erreur adapté.

Q.17. Cependant nous aimerions pouvoir ajouter une liste de fichiers et si parmi ceux-ci certains sont erronés, nous retournons la liste des erreurs mais nous ajoutons tous les fichiers qui respectent le bon format.

Q.18. Le programme principal (demo) ne doit pas se planter en cas d'erreur d'indice ou de fichier inexistant.

[1] Ce serait trop bizarre d'accéder à la piste 0…

Q.19. De quelle manière avez-vous implémenté - *Il listera le nom de tous les fichiers*- ? Pour ceux qui auraient implémenté cette fonctionnalité en renvoyant simplement la liste des noms de fichiers (`return files`), que se passe-t-il si vous modifiez cette liste ?

Q.20. Pour qu'il ne soit pas possible de modifier la liste des pistes enregistrées en dehors de l'organiseur, ne renvoyez pas la liste mais une copie de la liste.

Q.21. RAPPEL : Aucun code dans la classe MusicOrganizer ne fait référence à une interaction homme machine (println ou read). Vérifiez que vous avez bien respecté cette contrainte.

III.3 Comparer les éléments d'une liste : un organisateur permettant la sélection par critère.

III.3.1 Spécifications à implémenter

Q.22. Ajoutez une fonctionnalité permettant de lister uniquement les pistes dont le nom contient une chaîne donnée de caractères.

Q.23. Testez-la également avec une chaîne de recherche qui ne correspond à aucun des noms de fichiers. Comment se comporte votre code métier? et la partie "Main/demo", qui doit bien évidemment rattraper l'erreur et ne pas s'interrompre ?

Q.24. Ajoutez une fonctionnalité qui lit des échantillons de toutes les pistes d'un artiste particulier, l'une après l'autre.

- Dans vos tests, assurez-vous de choisir au moins un artiste avec plus d'un fichier.
- Utilisez la méthode `playAndWait` du MusicPlayer, plutôt que la méthode `startPlaying` ; sinon, vous finirez par lire toutes les pistes correspondantes en même temps. La méthode `playAndWait` lit le début d'une piste (environ 15 secondes) puis revient.

IV. (V2) Objets et Responsabilités : Vers un organisateur de pistes de musique

Bien sûr, à ce stade, nous pouvons constater que de stocker des listes de noms de fichiers contenant certains détails de la piste musicale n'est pas vraiment satisfaisant. Par exemple, supposons que nous voulions trouver toutes les pistes avec le mot "love" dans le titre. En utilisant la technique de correspondance peu sophistiquée mise en œuvre précédemment, nous trouverons également des morceaux d'artistes dont les noms contiennent cette séquence de caractères (par exemple, Glover). OK, faisons mieux, pour un produit qui a plus de valeur avec juste un peu plus d'effort.

IV.1 Des spécifications au code

Q.25. Avant de vous lancer dans le développement, réfléchissez à l'architecture résultante. "Modélisez"-la puis, à la fin de cette partie, comparer votre "modélisation" avec ce que vous avez vraiment implémenté.

Q.26. Implémentez les spécifications suivantes et testez vos implémentations.

1. Nous souhaitons pouvoir manipuler maintenant des pistes mieux identifiées, c'est-à-dire une piste contient la référence à un fichier, mais aussi l'artiste, le titre, etc.
2. Nous souhaitons ainsi pouvoir rechercher les pistes dont le titre contient un terme particulier, dont le nom de l'artiste contient un terme particulier, etc.
3. Cependant, nous voulons continuer à ajouter des fichiers musicaux comme précédemment qui ont l'artiste et le titre dans le nom du fichier. Nous n'avons pas envie de tout réécrire !
4. Attention, nous souhaitons pouvoir aussi ajouter des fichiers qui ne respectent pas ce format, voire s'appuient sur d'autres formats comme titre__Artiste__Année, etc. Pour l'instant nous ne les avons pas.
   - *Astuce* : Votre architecture doit donc tenir compte de cette possibilité à venir. Quel objet a la responsabilité de transformer un nom de fichier en une piste ?
5. Il est nécessaire de savoir combien de fois une piste a été écoutée. Cela nous permettra par la suite de faire des recommandations.
6. Si vous lisez deux pistes sans arrêter la première, les deux joueront simultanément. Ce n'est pas très utile.
   - *Astuce* : Modifiez **votre programme** afin qu'une piste en cours de lecture soit automatiquement arrêtée lorsqu'une autre piste démarre. Ne touchez pas au code de *MusicPlayer*.
7. Nous souhaitons disposer d'une fonction de "lecture aléatoire", *i.e.*, qu'elle sélectionne une piste au hasard et la lit.
   - *Astuce :* Le package `java.util` contient la classe `Random` dont la méthode `nextInt` génère un entier aléatoire dans une plage limitée. Vous devrez importer `Random` et créer un objet `Random`, soit directement dans la nouvelle méthode, soit dans le constructeur et stocké dans un champ. Vous devrez trouver la documentation de l'API pour la classe Random et étudier ses méthodes pour choisir la bonne version de nextInt.

8. Lire les pistes “préférées” dans un ordre aléatoire. On considère qu’une piste est préférée si elle a été écoutée plus que la moyenne des écoutes (pour les plus avancés au-dessus de la médiane). Attention, même quand on écoute une piste préférée son nombre d’écoutes augmente.

Astuce : Une façon de procéder consiste à mélanger l’ordre des pistes, puis à les lire du début à la fin. Une autre façon serait de faire une copie de la liste, puis de choisir à plusieurs reprises une piste au hasard dans la liste, de la lire et de la supprimer de la liste jusqu’à ce que la liste soit vide.
Essayez de mettre en œuvre l’une de ces approches. Si vous essayez la première, est-il facile de mélanger la liste pour qu’elle soit véritablement dans un nouvel ordre aléatoire ? Existe-t-il des méthodes de bibliothèque qui pourraient aider à cela ?
(`Collections.shuffle(preferedTracks);`)

9. On désire pouvoir trier les pistes selon :
   - le nom de l’artiste
   - le titre
   - le nombre d’écoutes.

Mais attention, cela ne doit pas modifier l’ordre des pistes, sinon nous ne saurions plus dans quel ordre les pistes ont été ajoutées et l’ordre changerait à chaque ajout (déjà qu’il bouge peut-être au retrait…)!

   – *Astuce* : Est-ce que d’implémenter “Comparable” répondrait au problème ? Pensez à utiliser des comparators.

10. Uniquement si vous avez tout fini et que vous vous ennuyez : gérez des “playLists” à votre convenance par exemple: playListduSoir.

V. HELP 1 : Pour manipuler des ArrayList

V.1 Importer une classe

Avant l’entête de la classe et après l’accroche au package.

```
import java.util.ArrayList;
```

V.2 Déclarer une variable de type ArrayList

```
ArrayList<String> files;
```

The ArrayList indicates that the variable *files* is of type ArrayList and that this ArrayList stores objects of type String. When using collections, we have to declare the type of the collection and the type of the objects in the collection, so that ArrayList can be read as ArrayList of String.
Enfin si vous vous souvenez, on préférera la définition la plus générale qui lie *files* à une interface *List* au lieu d’une de ses implémentations *ArrayList*

```
List<String> files;
```

V.3 Créer une instance de ArrayList

```
files = new ArrayList<>();
```

V.4 Numérotation dans les collections

La position d’un élément dans une collection est connue sous le nom d’index, et le premier élément par convention a toujours l’index 0.

La méthode **get** de la collection ArrayList renvoie un élément de la liste (mais ne le supprime pas)

```
String fileName = files.get(fileNumber));
```

La plage d'index valide est 0…(size()-1), où la méthode **size** donne le nombre d'objets stockés dans une collection, par exemple, *files.size().*

V.5 Ajouter un élément à une liste

```
files.add("music.mp3");
```

V.6 Parcourir une liste

Il existe différentes façons de parcourir une liste comme sur cet exemple où nous utilisons un foreach :

```
List<String> matchingList = new ArrayList<>();
for (String filename : files) {
    if (filename.contains(searchString)) {
        // A match.
        matchingList.add(filename);
    }
}
```

Cette boucle signifie :

Création d'une Liste matchingList
Pour chacun des éléments `filename` de type `String` de `files`
Si la chaine contenue dans `filename` contient la string `searchString`
alors l'ajouter à la liste `matching`.

V.7 Copier une liste

Pour que la liste ne puisse pas être modifiée de l'extérieur, vous devez éviter de retourner vos listes, par exemple, si vous renvoyez la liste des morceaux gérés par votre organiseur alors elle pourra être modifiée de l'extérieur sans que vous puissiez contrôler les changements opérés.

Pour éviter de briser ainsi l'encapsulation, passer par une copie.

```
List<String> listFileList(){
List<String> copy = new ArrayList<>(files);
return copy;
}
```

VI. HELP 2 : Quelques fonctions pour manipuler des fichiers

VI.1 Pour vérifier que les fichiers existent

```
import java.io.File;

private void checkMusicFile(String filename) {
    if (filename == null || filename.equals("")) {
        throw new IllegalArgumentException("The file name " + filename + " is not valid.");
    }
    File musicFile = new File(filename);
    if (!musicFile.exists()) {
        throw new IllegalArgumentException("The file " + filename + " does not exist.");
    }
    if (!musicFile.isFile()) {
        throw new IllegalArgumentException("The file " + filename + " is not a file.");
    }
```

```
    if (!musicFile.canRead()) {
        throw new IllegalArgumentException("The file " + filename + " cannot be read.");
    }
    if (!filename.endsWith(".mp3")) {
        throw new IllegalArgumentException("The file " + filename + " is not a mp3 file.");
    }
}
```