Travail Pratique

**Département de génie logiciel et des TI**

|  |  |
| --- | --- |
| Numéro du laboratoire | 3 |
| Étudiant(s) | Benoit, Marie-Ève  Desmarais, Gabriel |
| Code(s) Permanent(s) | BENM22568707  DESG24078908 |
| Cours | GTI310 |
| Session | Automne 2010 |
| Groupe | 01 |
| Professeur | Stéphane Coulombe |
| Chargé de laboratoire | Jean-François Franche |
| Date | 02 Novembre 2010 |

**Table des matières**

[1. Introduction 1](#_Toc275597477)

[2. Choix de design 2](#_Toc275597478)

[3. Implémentation 3](#_Toc275597479)

[4. Analyse du projet 4](#_Toc275597480)

[5. Conclusion 5](#_Toc275597481)

[6. Réponse aux questions supplémentaires 6](#_Toc275597482)

[7. Manuel de l’usager 7](#_Toc275597483)

[ANNEXE 1](#_Toc275597484)

# Introduction

Le musée Labyrinthe-En-Ville désire pourvoir identifier tous les parcours possibles de ses salles.

La présente itération a pour but de compléter une application déjà existante (Unreal Networks Solver) afin d’y ajouter les classes permettant la lecture du fichier d’information (informations du graphe), le calcul des parcours et, finalement, d’écrire la solution dans un fichier texte.

Pour ce faire, une classe Data pour conserver les informations des nœuds et des arches, une classe concreteParser pour prendre les informations provenant d’un fichier texte, une classe ConcreteSolver qui résout la problématique, une classe PathList pour conserver les différentes solutions et une classe ConcreteWriter pour l’écriture du fichier de solutions devront être implémentées.

Le présent document contient une section Choix de Design qui expose les différentes décisions possibles inhérentes à la conception, une section Implémentation contenant les détails de l’implémentation et l’analyse complète du projet est exposée dans la section Analyse du Projet. Un retour sommaire sur les notions abordées se trouve dans la section Conclusion, il s’en suit le guide d’utilisateur ainsi que l’annexe.

# Choix de design

Pour réaliser le projet, il était nécessaire de respecter certains éléments de design. Ces éléments consistaient à devoir parcourir tous les nœuds (salles) du graphe, de respecter le fait que certains arches (couloirs) étaient à sens unique, de pouvoir commencer le parcours à partir de n’importe quel nœud et de pouvoir réutiliser l’application pour d’autres problématiques (Ne pas la limiter au musée). L’application devait également générer un fichier texte contenant tous les parcours solutionnant la problématique.

Plusieurs possibilités ont été envisagées afin de répondre aux demandes.

1. Faire une liste de listes pour les chemins possibles ou faire des tableaux statiques.

Une des principales problématiques rencontrées durant le développement de l’application était le choix d’utiliser une liste de liste

1. Utiliser la récursivité ou faire plusieurs boucles imbriquées pour solutionner le graphe.
2. Faire deux lectures du fichier d’informations ou une seule pour le parser.

# 3. Implémentation

Les deux applications ont été faites dans deux projets différents pour faciliter la conception des solutions aux problèmes posés. En séparant complètement les applications, toutes les modifications faites à une application n’interféraient pas avec l’autre application. Cette manière a aussi facilité l’apprentissage des différentes fonctionnalités d’une application par les membres de l’équipe puisqu’une fois une application terminée, le code de cette application ne subissait plus de changement.

Afin de réaliser les deux programmes, deux classes ont été créées soit ConcreteAudioFilter qui se charge de la conversion au format 8 bits et SNRFilter qui s’occupe de faire l’analyse de qualité de l’échantillon. Les deux classes héritent de AudioFilter et implémentent une fonction process().

En plus de la fonction process(), les deux classes ont les méthodes check(byte[] data, X) qui se charge de la validation des données envoyées en les comparant à la valeur de X, et exit() qui ferme proprement les éléments utilisés dans le programme avant d’ordonner la fermeture de celui-ci.

Le code effectuant le travail principal des deux applications est contenu dans une méthode séparée de process(), soit transformData(byte[] data) pour ConcreteAudioFilter et eatChunk(int chunk) pour SNRFilter. Cette séparation nous permet de faire le travail sur les fichiers audio par morceaux pour ainsi diminuer la mémoire utilisée par l’application.

C’est pour des raisons de mémoire qu’aucune classe supplémentaire pour la gestion des fichiers audio n’a été ajoutée. Nous avons opté pour la lecture octet par octet des fichiers sans revenir en arrière lorsque le travail sur les octets lus était effectué.

La première application, servant à faire le transfert d’un fichier 16 bits vers un fichier 8 bits, appelle directement les méthodes check() pour valider les différents en-têtes. Cette manière de faire est utilisée parce que l’application ne gère qu’un seul fichier à la fois, contrairement à l’application servant à calculer le SNR de plusieurs fichiers. Cette deuxième application utilise la fonction validateHeader(byte[] header) qui, appelée en boucle, valide l’en-tête de tous les fichiers comparés. C’est cette dernière méthode qui appelle les méthodes check().

# 4. Analyse du projet

Nous avons utilisé deux projets complètement séparés pour les deux applications à développer, mais la création d’un seul projet avec des sources partagées aurait permis de mieux réutiliser le code déjà écrit lors de la première application et d’éviter le doublon de code pour certaines fonctions. Cette réutilisation de code nous aurait donné plus de difficulté pour écrire un code s’appliquant à toutes les situations encourues dans les deux programmes à la place de palier seulement aux situations survenant dans un programme spécifique, mais la réutilisation de ces classes partagées aurait alors pu être possible pour un autre projet gérant des fichiers audio tels que ceux-ci.

Nous n’avons pas utilisé de classe gérant le type de fichier audio utilisé et cette méthode reste la meilleure à notre avis pour atteindre les objectifs fixés par l’énoncé du projet. Dans le cas où plusieurs de types de fichiers audio doivent être supportés, le transfert du code servant à décoder des fichiers WAV devra être envoyé dans une classe tierce, nous permettant ainsi d’accomplir ces nouveaux objectifs.

Il n’y a aucune fonction lisant l’en-tête du fichier audio, cette fonction, si elle doit exister, devrait être placée dans la classe gérant le type du fichier audio utilisé. Nous n’avons pas trouvé nécessaire de créer cette fonction tant que nous utilisions le même type de fichier audio ou que nous n’avions pas de classe spécifique à ce type.

La fonction validant les en-têtes a par contre été faite dans la deuxième application, puisqu’elle doit valider l’en-tête de plusieurs fichiers audio. Cette fonction aurait facilement pu être retrouvée dans la première application, mais les objectifs de celle-ci ne nécessitaient pas de la faire. Dans le cas où notre application prendrait de l’ampleur de quelque manière que ce soit, soit en incorporant la deuxième application ou en ajoutant plus de types de fichiers audio possibles pour la lecture, la création de cette méthode serait un excellent ajout.

# 5. Conclusion

Lors de l’écriture de la première application, nous avons eu plusieurs problèmes de compréhension liés au transfert des valeurs en 16 bits vers des valeurs en 8 bits.

En plus de ces quelques problèmes, la gestion de la lecture et la manipulation des fichiers par morceaux sans perdre d’informations c’est avéré un intéressant défi à relever. En effet, ces morceaux servaient à réduire la mémoire utilisée par les applications, mais sectionnaient la lecture et la manipulation des fichiers.

Malgré ces divers problèmes, les objectifs énoncés à travers le projet ont été accomplis en entier. Les différentes complications et problématiques rencontrées ont donc pu être réglées, mais auront nécessité plusieurs explications et recherches.

# 7. Manuel de l’usager

Lors des tests effectués, nous avons utilisé les configurations fournies par Eclipse pour gérer les arguments envoyés aux applications. Ces arguments étaient les suivants :

"Session 3\\GTI310\\AudioResampler-TP2-P1\\App1Test1Stereo16bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P1\\output.wav"

pour l’application 1 et les suivants pour l’application 2 :

" Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Original\_Mono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod1\_AmpBase\_Mono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod2\_30SamplesDelay\_Mono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod3\_Echo1SecMono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod4\_WhiteNoise1\_Mono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod5\_WhiteNoise2\_Mono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod6\_WhiteNoise3\_Mono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod7\_WhiteNoise4\_Mono8bits.wav" " Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod8\_PinkNoise\_Mono8bits.wav"

La méthode qui doit être utilisée en dehors de l’interface de développement est la suivante :

java <programme1> <ficher d’entrée> <fichier de sortie>

java <programme2> <ficher de référence> <fichiers à analyser>

Cette méthode permettra à un utilisateur de lancer les deux applications avec les arguments nécessaires.

# ANNEXE