Itération 1

Suivi de Planification du Projet

Toutes les fonctionnalités prévues pour cette itération ont été implémentées avec succès, notamment :

- **Lecture de morceaux :** L'application scanne un dossier pour récupérer les fichiers audio (.mp3, .flac, etc.), permettant leur lecture avec affichage de la durée totale et du temps écoulé.
- **File d'attente :** Implémentation d'une file d'attente FIFO pour gérer la lecture séquentielle des morceaux avec possibilité d'ajout, de suppression et de réorganisation.
- **Boutons de contrôle :** Ajout des boutons de lecture (lecture/pause, précédent, suivant) et d'un curseur pour naviguer dans le morceau ou ajuster le volume.
- **Contrôles divers :** Intégration des options avancées telles que la vitesse de lecture, le mode aléatoire et la balance des canaux audio.

Difficultés Techniques et Solutions

Durant cette itération, nous avons rencontré des difficultés notables liées à l'implémentation de la lecture des métadonnées des fichiers audio. Initialement, nous avions tenté une implémentation manuelle de cette fonctionnalité. Cependant, il s'est rapidement avéré que cette approche était excessivement complexe, très "low-level", et entraînait une charge de travail considérable.

Pour résoudre ce problème, et après validation par l'assistant responsable, nous avons décidé d'utiliser une librairie spécialisée existante - JAudioTagger. Ce changement a nécessité un refactoring conséquent du code existant, entraînant une perte de temps significative.

Impact sur la Planification

Ces difficultés techniques ont provoqué une perte de temps substantielle. De plus, la surestimation initiale de la charge de travail a entraîné une répartition inefficace des tâches, laissant certaines équipes avec relativement peu de travail durant une partie de l'itération.

Couverture des Tests

Actuellement, la couverture des tests unitaires n'est pas optimale. Nous sommes conscients de ce problème et avons prévu d'améliorer significativement cette couverture pour la prochaine itération.

Motivation des Choix de Conception

Le principal choix de conception adopté est le design pattern MVC (Model-View-Controller), adapté à l'environnement JavaFX. Ce choix a été motivé par la familiarité de l'équipe avec ce pattern, ainsi que par sa fiabilité éprouvée et sa popularité dans le développement d'applications graphiques. Ce modèle facilite la séparation claire des responsabilités, améliorant ainsi la maintenabilité et la lisibilité du code produit.

Itération 2

Suivi de Planification du Projet

Toutes les fonctionnalités prévues pour cette itération ont été implémentées, notamment :

- **Importation et affichage des paroles** : L'application permet d'importer et d'afficher les paroles des morceaux pendant leur lecture.
- **Radio en ligne** : La possibilité d'écouter des flux web radio et de les ajouter à la bibliothèque a été ajoutée.
- **Multilinguisme** : L'application prend en charge l'affichage des informations dans plusieurs langues, selon la préférence de l'utilisateur.
- **Tags personnalisés** : Il est désormais possible d'ajouter des tags personnalisés aux morceaux pour les classer.
- **Modification des métadonnées** : Les utilisateurs peuvent maintenant modifier les métadonnées des fichiers audio, telles que le titre, l'artiste et le genre.
- **Listes de lecture** : Les utilisateurs peuvent créer et gérer des playlists, avec un titre et une image optionnelle.
- **Récupération des pochettes d'album** : L'application peut associer des images d'album aux morceaux et les afficher.
- **Playlist "Favoris"** : Une playlist "Favoris" a été ajoutée pour permettre d'ajouter facilement des morceaux préférés.
- **Égaliseur audio** : Un égaliseur a été intégré, permettant d'ajuster les fréquences entre -24dB et +12dB.

Ces fonctionnalités sont maintenant disponibles et opérationnelles dans l'application.

Difficultés Techniques et Solutions

Lors de cette itération, nous avons rencontré des difficultés liées à l'architecture **MVC**, où chaque vue devait être fortement couplée à son contrôleur. Avec plusieurs vues et contrôleurs partageant des éléments communs, notre première approche basée sur une hiérarchie classique a rapidement conduit à une duplication de code et une complexité accrue. Pour résoudre ce problème, nous avons adopté le **Curiously Recurring Template Pattern (CRTP)**, permettant de structurer proprement la relation entre vues et contrôleurs via des classes génériques `ViewController` et `View`. Ce choix a nécessité

un **refactoring conséquent**, ralentissant temporairement le développement, mais garantissant une meilleure réutilisabilité et maintenabilité à long terme.

Histoire 13 (Paroles)

Dans un premier temps, nous avons essayé d'intégrer les paroles directement dans les métadonnées des fichiers **`.mp3`** et **`.wav`**. Toutefois, cette approche a rapidement révélé des problèmes de compatibilité avec les fichiers **`.wav`**, en plus d'augmenter significativement la taille des fichiers audio et surtout elle nous exposait à des erreurs critiques si le fichier était mal modifié

Pour surmonter ces difficultés, nous avons décidé de séparer totalement les paroles des fichiers audios. Nous avons donc conçu un système reposant sur 2 éléments : un fichier **`lyrics.json`** qui mappe chaque fichier chanson à un fichier **`.txt`** contenant ses paroles, et un dossier contenant les fichiers **`.txt`** des paroles. Ce choix permet de préserver l'intégrité des fichiers audio tout en facilitant la gestion des paroles.

Histoire 18 (Multilinguisme)

Un problème avec le streaming des flux web pour la radio est apparu en utilisant certains URL. Après pas mal de recherches, il s'avère que c'est en réalité dû à la classe Media de JavaFX qui, pour stream un flux web avec le protocole HTTP, s'attend à recevoir une réponse HTTP standard du serveur de stream afin de lancer l'écoute de la radio. Cependant, cela n'est pas le cas avec chaque serveur auquel nous nous connectons, ce qui limite les URLs que nous pouvons utiliser. Une solution possible pour améliorer la fonctionnalité des radios, serait d'écouter les flux web directement sur le localhost, et ensuite de renvoyer Media vers le localhost. Cependant, cela complique énormément l'implémentation, et les solutions testées jusqu'à présent ne sont pas suffisamment satisfaisantes que pour en faire une feature.

Impact sur la Planification

Le développement de l'**histoire 13 (Paroles)** a été plus long que prévu à cause des changements d'approche et du refactoring architectural. Mais ces ajustements ont permis une intégration plus propre et plus maintenable.

Couverture des Tests

Des tests ont été ajoutés pour couvrir les fonctionnalités de modification des métadonnées et des tags personnalisés, afin de vérifier leur bon fonctionnement et prévenir toute régression.

D'autres tests ont été ajoutés pour vérifier la robustesse du système de mapping JSON, la lecture et l'écriture de fichiers de paroles, ainsi que la tolérance aux erreurs (absence de fichier, mauvais format, ...).

Motivation des Choix de Conception

Histoire 13 (Paroles)

Pour organiser la gestion des paroles, nous avons adopté une strucure claire et modulaire : *DataProvider* centralise l'accès aux fichiers (lyrics.json, fichiers texte), tandis que *LyricsManager* se charge spécifiquement du traitement des paroles (chargement, sauvegarde et mapping). Ce choix renforce l'architecture MVC en séparant clairement l'affichage (*LyricsView*), les interactions utilisateur (*LyricsController*) et la logique des paroles. Ainsi, le code est devenu plus lisible, facile à maintenir et à tester.

Itération 3

Suivi de Planification du Projet

Toutes les fonctionnalités prévues pour cette itération ont été implémentées, notamment :

- **Recherche :** L'utilisateur peut rechercher des morceaux dans sa bibliothèque via le titre, l'artiste, l'album ou les tags.
- **Visualiseur audio :** Un visualiseur graphique affiche en temps réel l'évolution du son pendant la lecture.
- **Autocomplétion des données :** l'application proposera automatiquement des valeurs pour les champs "artiste", "album" et "tags" lors de la modification.
- **Mode Karaoké :** permet à l'utilisateur de chanter en suivant des paroles synchronisées avec la musique en temps réel
- **Mode DJ :** permet à l'utilisateur de modifier la musique jouée en direct, à l'aide de plusieurs sliders et effets

Ces fonctionnalités sont maintenant disponibles et opérationnelles dans l'application.

Difficultés Techniques et Solutions

Histoire 10 (Mode "Dj")

Lors de cette itération, nous avons rencontrés quelques problèmes pour l'implémentation du mode Dj.

Nous supposions initialement que JavaFX fournissait les outils nécessaires à la création d'effets sonores à la volée sur le flux musical. Cependant, la librairie étant très haut-niveau, nous ne pouvions pas manipuler les données comme nous le voulions, ni même appliquer des effets créés "à la main" à l'aide d'une libraire de plus bas niveau (e.g. JavaSound) étant donné que le flux audio était inaccessible avec le player de JavaFX.

L'idée initiale pour contourner ce problème était d'utiliser un player annexe qui nous permettait du traitement de flux plus aisé et avancé, et avons pour cela tenté l'utilisation de deux librairies externes (Minim & TarsosDSP). Cependant, ces deux librairies se sont

révélées trop complexes à intégrer avec la version de java modulaire que le projet utilise, et il nous a été impossible d'aboutir à une solution "propre" et fonctionnelle.

Après avoir pris un retard conséquent sur l'implémentation de l'histoire, nous avons décidé de nous arrêter à quelques effets sonores basés sur l'égalisation des bandes de fréquence, ce qui était relativement simple à faire avec JavaFX et son égaliseur intégré.

Histoire 14 (Mode Karaoké)

Le principal défi a été d'assurer une synchronisation fluide entre l'avancement de la chanson et l'affichage des lignes de paroles.
Nous avons d'abord testé des approches basées sur des positions fixes ou des identifiants, mais cela complexifiait inutilement la logique. Nous avons finalement adopté une méthode simple et fiable, à chaque instant, l'application identifie la dernière ligne dont le timestamp est inférieur au temps de lecture actuel.

Couverture des Tests

Des tests unitaires ont été écrits pour vérifier la robustesse du système de gestion des paroles, notamment : la lecture et l'écriture de fichiers de paroles, la création des mappings dans **`lyrics.json`** et le bon fonctionnement en cas de fichiers manquants ou mal formatés.

Motivation des Choix de Conception

Histoire 10 (Dj mode)

Afin de garder le code propre et cohérent, il était plus "sage" d'utiliser le lecteur intégré de javaFX, tout comme dans le reste de l'application.

C'est pour cela que nos effets se basent exclusivement sur ce que ce player nous permet de faire, et plus précisément sur des changements de fréquence.

Itération 4

Suivi de Planification du Projet

Toutes les fonctionnalités prévues pour cette itération ont été implémentées, notamment :

- **Suggestion de morceaux** : L'application suggère des morceaux similaires à ceux déjà présents dans la playlist, en se basant sur leurs métadonnées (tags, artiste, genre.), afin de faciliter la découverte musicale.
- **Mode transition** : Ajoute une transition où le volume du morceau actuel diminue pendant que le suivant augmente avec une durée réglable
- **Vidéo comme pochette** : Remplace la pochette par une vidéo muette pendant la lecture
- **Comptes utilisateurs** : Permet à plusieurs utilisateurs locaux de créer un compte, chacun avec ses morceaux et playlists. Un dossier global est partagé et un menu permet de changer d'utilisateur sans mot de passe

Ces fonctionnalités sont maintenant disponibles et opérationnelles dans l'application.

Motivation des Choix de Conception

Histoire 20 (Vidéo comme pochette)

Pour implémenter cette fonctionnalité, nous avons choisi d'adopter l'approche utilisée pour les pochettes d'album sous forme d'image : stocker les données sous forme de bytes directement dans un champ personnalisable des métadonnées. Nous avons décidé de ne pas écraser la pochette existante, le cas échéant, mais plutôt de conserver à la fois le clip et la pochette, afin de permettre un affichage personnalisé selon nos besoins.

Histoire 22 (Comptes Utilisateurs)

Cette fonctionnalité ayant un impact significatif sur plusieurs composants clés de notre application, nous avons opté pour une approche prudente. L'objectif était d'intégrer cette nouveauté tout en minimisant les modifications des interactions existantes entre les différentes classes.

Concernant la gestion des paramètres, nous avons introduit un nouveau modèle, conçu pour s'appuyer sur l'existant (`Settings`). Cela nous a permis de préserver l'interface utilisée par nos contrôleurs, tout en révisant en profondeur le fonctionnement interne du système de configuration.

Difficultés Techniques et Solutions

Histoire 15 (Mode transition)

L'idée était d'implémenter un deuxième player afin de pouvoir jouer deux pistes audios en même temps. Nous avions alors décidé d'utiliser un AudioClip, qui permet de lancer un audio quasiment sans délai. Seul problème : quand l'AudioClip est lancé, le volume de ce dernier ne peut être changé, ce qui est assez problématique dans le cas de notre transition. Nous avons alors décidé de le transformer en MediaPlayer, qui est une structure de donnée légèrement plus lourde mais qui permet de changer le volume durant la lecture. Nous avons alors deux *Players:* un pour la transition, et un pour la musique en général.

Histoire 22 (Comptes Utilisateurs)

Lors de l'implémentation de cette fonctionnalité, plusieurs défis techniques se sont présentés :

- L'ajout de la gestion multi-utilisateur nous a contraints à repenser entièrement la manière dont les paramètres étaient sauvegardés, ceux-ci étant jusque-là étroitement liés à l'utilisateur actif.
- Notre système de gestion des playlists s'est également révélé inadapté, chaque utilisateur devant désormais disposer de ses propres listes de lecture personnalisées.

Pour surmonter ces obstacles, nous avons dû revoir en profondeur la structure de nos paramètres et de nos playlists. Cela a nécessité une refonte complète du service et du dépôt responsable de leur stockage persistant.

Modèle MVC

La structure du code n'était pas adaptée à la structure du MVC et il a donc fallu de nombreuses heures pour corriger la structure du projet. Une meilleure compréhension

de ce concept aurait permis de gagner beaucoup de temps et nous éviter d'investir autant d'efforts dans la réduction de couplage entre les contrôleurs et les vues.