RAPPORT

Nom	***Prénom***	***Adresse e-mail***
Menaouer	*Ammari*	`ammari.menaouer@etu.cyu.fr`
Angelise	*Enzo*	`enzo.angelise@etu.cyu.fr`

MÉTADONNÉES DE FICHIERS MP3 & PLAYLISTS

Projet POO & Java L2-I 2025–2026

Application : ***MP3 Explorer / Playlist Builder***

Classe : *L2 Informatique*
Établissement : *CY Paris Cergy Université*
Date de modification : *22 décembre 2025*

Enseignant(s) : *Marc Lemaire, Liu Tiaxino*
Groupe : *A15*
Version du document : *1.0*

Table des matières

1 Introduction

*Ce projet porte sur l'analyse de fichiers musicaux au format **MP3** et sur l'exploitation de leurs métadonnées **ID3** (titre, artiste, album, année, genre, numéro de piste, durée, etc.). L'objectif est de concevoir une application Java orientée objet capable, d'une part, d'extraire et d'afficher proprement les informations associées à un fichier MP3 donné, et d'autre part, d'explorer automatiquement un répertoire et ses sous-répertoires afin d'identifier uniquement les fichiers MP3 de manière fiable. Pour éviter les faux positifs, la détection ne se limite pas à l'extension `.mp3` mais s'appuie également sur le **type MIME** et, si nécessaire, sur une vérification de signature. À partir des pistes détectées, l'application construit ensuite une playlist cohérente et l'exporte dans des formats standards largement compatibles : **XSPF** (XML), **JSPF** (JSON) et **M3U8** (UTF-8). Enfin, le logiciel propose deux modes d'utilisation complémentaires : un mode **console (CLI)** pour des traitements rapides et scriptables, et un mode **graphique (GUI)** pour une exploration plus visuelle des pistes, des métadonnées et des exports.*

2 Organisation du projet

2.1 Organisation du travail (démarrage, UML, Git/GitHub et réunions Discord)

*Nous avons démarré par une phase courte de **conception UML** (cas d'utilisation puis diagramme de classes), afin de clarifier le périmètre et de stabiliser l'architecture avant d'implémenter. Cette étape a servi de **référence commune** : quelles entités manipule-t-on (piste MP3, métadonnées, playlist), quels services sont nécessaires (lecture ID3, scan de répertoire, export), et comment la CLI et la GUI consomment ce noyau.*

*À partir de l'UML, le projet a été découpé en trois blocs : (i) **noyau métier** (modèle + services), (ii) **interface console (CLI)**, (iii) **interface graphique (GUI)**. Pour garantir une intégration fluide, nous avons défini **à deux dès le début** les classes structurantes (modèle et contrats de services), car elles conditionnent la cohérence de tout le code :*

- *classes du modèle (ex. `Track`, `Metadata`, `Playlist`) : attributs, invariants, API ;*
- *signatures des services (lecture ID3, scan, export) : entrées/sorties, comportements en cas d'erreur ;*

- *règles minimales d'export (XSPF/JSPF/M3U8) : contenu attendu, encodage UTF-8, gestion des chemins.*

Le développement a ensuite été réalisé majoritairement à distance. ***GitHub*** *a été utilisé pour centraliser le code, conserver un historique clair et faciliter l'intégration progressive des contributions, tandis que* ***Discord*** *a servi de canal principal de coordination. Nous avons maintenu une* ***réunion hebdomadaire*** *sur Discord (avec des points courts si nécessaire) structurée de manière constante : (1) avancement, (2) blocages, (3) décisions techniques, (4) plan d'action jusqu'à la séance suivante. Cette organisation a permis une progression régulière, une cohérence entre CLI/GUI et noyau métier, et une maîtrise partagée du projet par les deux membres du binôme.*

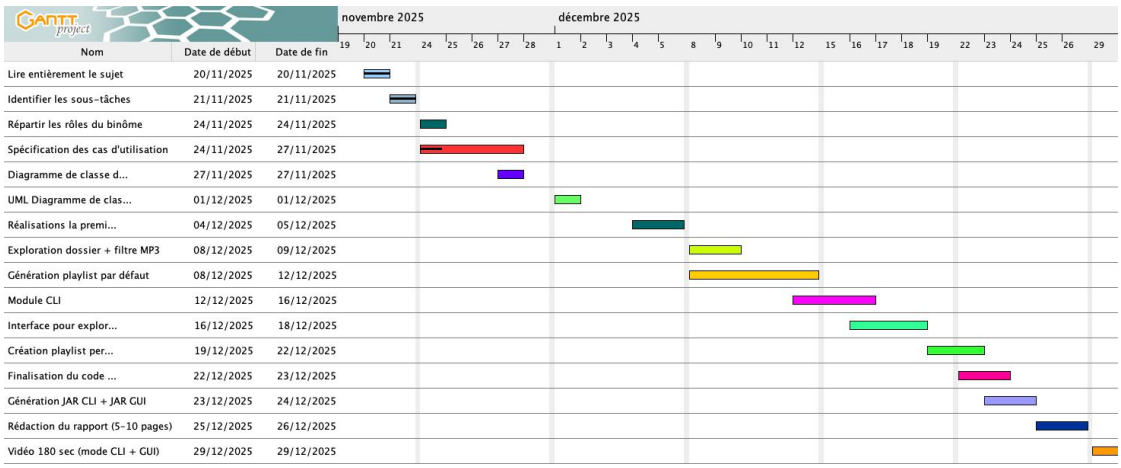

FIGURE 1 – Diagramme de Gantt (planning et répartition des tâches).

3 Conception et diagrammes UML

3.1 Méthodes et algorithmes clés

L'application repose sur une chaîne de traitement déterministe allant (1) de la découverte des fichiers à (2) l'extraction des métadonnées, puis (3) la construction d'une playlist et (4) l'export conforme au format choisi. Les méthodes ci-dessous sont les points techniques clés garantissant ***fiabilité****,* ***robustesse*** *et* ***conformité****.*

3.1.1 Exploration de l'arborescence (DFS avec pile) — résumé

Le parcours des dossiers est réalisé en ***profondeur d'abord (DFS)*** *en utilisant une* ***pile explicite*** *(ex.* `Deque<Path>`*) afin de contrôler la récursivité et d'éviter des appels récursifs non maîtrisés.*

- *parcours DFS de l'arborescence via une* ***pile*** *: on empile le dossier racine, puis on dépile/empile ses sous-dossiers ;*

- *filtration par type* ***fichier régulier*** *(`Files.isRegularFile`) : exclusion des dossiers, liens, sockets, etc. ;*
- *gestion d'erreurs* ***locales*** *dossier/fichier (try/catch par entrée) sans arrêt global du scan ;*
- *compteur global du* ***nombre de MP3 détectés*** *et constitution d'une* ***liste ordonnée*** *des pistes.*

Complexité : *le scan est en $O(N)$ où N est le nombre total d'entrées visitées. Chaque entrée subit un surcoût constant (tests extension/MIME/*magic bytes*). La mémoire est en $O(H)$ (taille maximale de la pile), où H correspond au nombre de dossiers simultanément empilés pendant la traversée.*

3.1.2 Filtrage MP3 fiable (extension + MIME)

Pour éviter les faux positifs (fichiers renommés), la détection combine ***plusieurs heuristiques*** *:*

- *test d'extension `.mp3` (****insensible à la casse****) ;*
- *test MIME via `Files.probeContentType(path)` (quand disponible) attendu : `audio/mpeg` ;*
- *repli par* ***signature*** *(*magic bytes*) : lecture de quelques octets en tête du fichier pour reconnaître un en-tête `ID3` (ID3v2) ou un motif compatible avec une trame MP3 (synchronisation).*

La règle retenue est volontairement stricte : un fichier est accepté comme MP3 uniquement s'il passe l'extension ***et*** *(MIME* ***ou*** *signature). Ainsi, un `.mp3` renommé ou non audio n'est pas exporté.*

3.1.3 Organisation et cohérence des fichiers

- *éviter les doublons en comparant les chemins des fichiers ;*
- *utiliser des chemins cohérents pour l'export, quel que soit le système ;*
- *afficher les pistes dans un ordre clair et lisible.*
- *éviter les doublons en comparant les chemins des fichiers ;*
- *utiliser des chemins cohérents pour l'export, quel que soit le système ;*
- *afficher les pistes dans un ordre clair et lisible.*

3.1.4 Extraction ID3 et construction du modèle métier

L'extraction des métadonnées est réalisée via une bibliothèque dédiée (ex. `Jaudiotagger`) et encapsulée derrière un service (`Id3Reader`) afin de découpler la dépendance externe du reste de l'application :

- *lecture des tags ID3 (ID3v1/ID3v2.x) et conversion en objets métier (`Metadata`, `Track`) ;*
- *politique de **valeurs par défaut** : champs absents → chaîne vide / `Optional` / `"Unknown"` ;*
- *gestion des fichiers corrompus : exception capturée, piste ignorée (ou marquée invalide) sans arrêter le traitement.*

3.1.5 Extraction et rendu du *cover art*

Quand une image est présente (frame APIC), elle est extraite en binaire puis convertie pour l'affichage :

- *récupération de l'`Artwork` (si présent) ;*
- *conversion en `BufferedImage` via `ImageIO` ;*
- *redimensionnement (*scaling*) côté GUI pour garder un rendu stable (miniature) ;*
- ***fallback** : placeholder en absence d'image.*

3.1.6 Génération et export des playlists (XSPF / JSPF / M3U8)

La génération repose sur une interface d'export (ex. `PlaylistWriter`) et trois implémentations, une par format :

- ***XSPF (XML)** : échappement XML, structure `<playlist><trackList><track>...` ;*
- ***JSPF (JSON)** : échappement JSON, structure `{playlist:{track:[...]}}` ;*
- ***M3U8 (UTF-8)** : écriture ligne par ligne (`#EXTM3U` optionnel), chemins en UTF-8.*

L'encodage UTF-8 est imposé à l'écriture (ex. `OutputStreamWriter(..., UTF_8)`) afin de préserver accents et caractères non latins. Chaque writer garantit : (i) un titre exporté ↔ une piste valide détectée, (ii) un ordre stable, (iii) aucune rupture en cas de métadonnées manquantes.

3.1.7 Parsing CLI : validation des arguments

Le CLI est traité comme un ensemble d'options à valider avec des contraintes strictes :

- *exclusivité `-f` ou `-d` ;*
- *`-o` implique exactement un format (`-xspf` ou `-jspf` ou `-m3u8`) ;*
- *en absence d'arguments : message d'erreur et suggestion `-h/-help`.*

Cette validation se fait avant tout traitement lourd (scan/lecture ID3) afin d'éviter un état partiellement construit.

3.1.8 Stratégie d'erreurs

Le comportement retenu vise la robustesse :

- *un fichier illisible/corrompu n'arrête pas l'application : il est signalé puis ignoré ;*
- *les erreurs d'accès lors du scan sont isolées par sous-dossier ;*
- *l'application finit avec un résultat partiel cohérent (playlist construite à partir des fichiers valides).*

3.2 Diagramme de cas d'utilisation

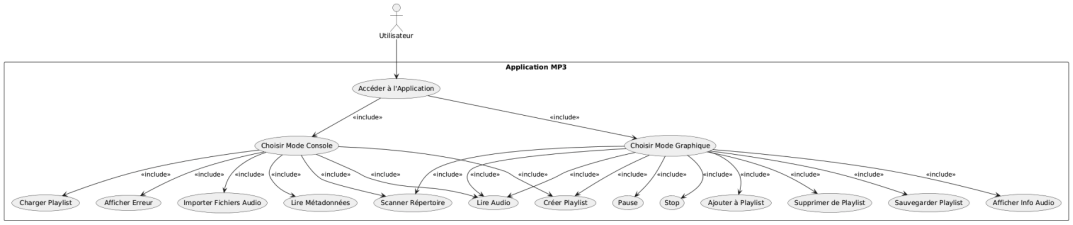

FIGURE 2 – Diagramme de cas d'utilisation.

3.3 Diagramme de classes

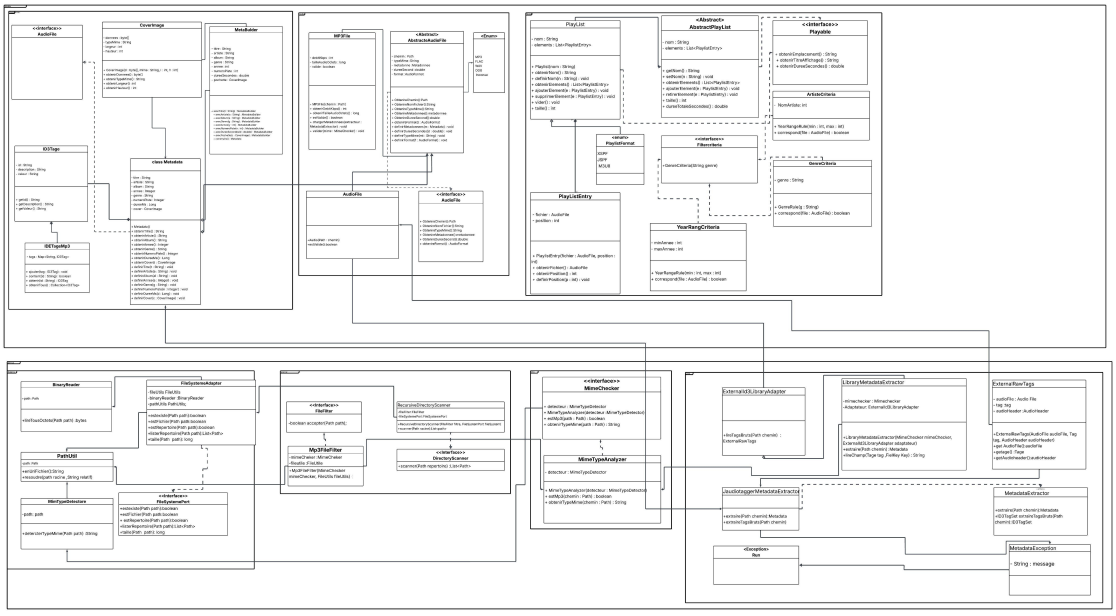

FIGURE 3 – Diagramme de classes (modèle + services + interfaces).

3.4 Principes de conception retenus

- ***Responsabilité unique*** *: parsing CLI, lecture MP3, génération playlist, UI séparées.*

- ***Faible couplage*** *: le modèle métier ne dépend pas des interfaces (CLI/GUI).*
- ***Extensibilité*** *: ajout futur d'un nouveau format de playlist sans refonte du reste.*

4 Architecture logicielle globale

L'application est organisée en ***quatre couches*** *afin de structurer clairement les responsabilités et de garantir la* ***maintenabilité****. (1) La couche* ***Modèle*** *représente les données manipulées (pistes, métadonnées, playlists). (2) La couche* ***Services*** *regroupe la logique technique réutilisable (scan de répertoires, filtrage MP3, extraction ID3, génération/écriture XSPF–JSPF–M3U8). (3) La couche* ***Cas d'utilisation*** *orchestre les enchaînements métier (*scanner → extraire → afficher → exporter*) en appliquant les règles de validation et de cohérence. (4) Enfin, la couche* ***Interfaces*** *propose deux points d'entrée indépendants,* ***CLI*** *et* ***GUI****, qui consomment les mêmes cas d'utilisation et garantissent un comportement identique. Cette séparation permet à chaque membre de travailler sur une couche ou un module sans bloquer l'autre, limite les conflits d'intégration et assure une base de code rigoureuse. Le projet a été* ***documenté*** *(rôles des classes, choix techniques, scénarios) et* ***versionné sur GitHub*** *avec des commits réguliers pour tracer l'évolution et les contributions.*

Les dépendances externes (ex. lecture ID3, lecture audio) sont encapsulées derrière des interfaces afin de limiter le couplage et de faciliter l'évolution.

5 Mode console (CLI)

5.1 Objectifs du CLI

- *analyser un fichier MP3 et afficher ses métadonnées (`-f`) ;*
- *analyser un répertoire et produire une playlist par défaut (`-d`) ;*
- *exporter la playlist dans un fichier de sortie (`-o`) au format choisi.*

5.2 Gestion des options et validation

- *si aucun argument : message d'erreur + suggestion `-h` ;*
- *`-f` et `-d` exclusives ;*
- *`-o` impose un format (`-xspf` ou `-jspf` ou `-m3u8`) ;*
- *aide claire via `-h`/`-help`.*

5.3 Schéma d'options

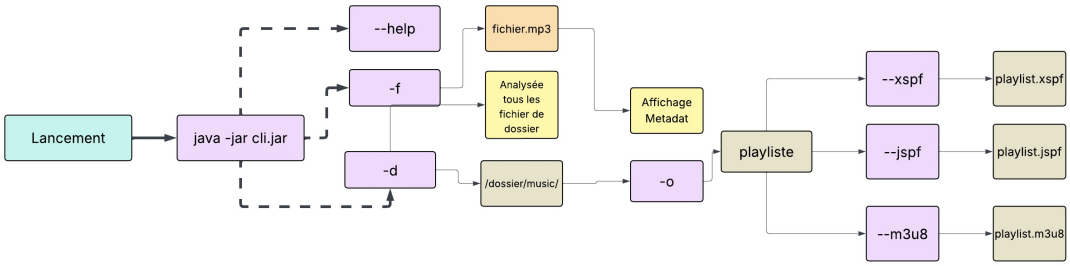

FIGURE 4 – Schéma récapitulatif des options CLI.

6 Mode graphique (GUI)

6.1 Fonctionnalités GUI

- *exploration d'arborescence et liste des MP3 ;*
- *affichage des métadonnées d'un MP3 sélectionné ;*
- *playlist par défaut et playlist personnalisée ;*
- *sauvegarde et réouverture d'une playlist.*

6.2 Gestion des illustrations associées aux pistes

Lorsqu'une image est intégrée aux métadonnées (pochette d'album), elle est récupérée et affichée automatiquement. En l'absence d'illustration, l'interface affiche un visuel par défaut afin de conserver un rendu homogène.

6.3 Maquette de l'interface graphique

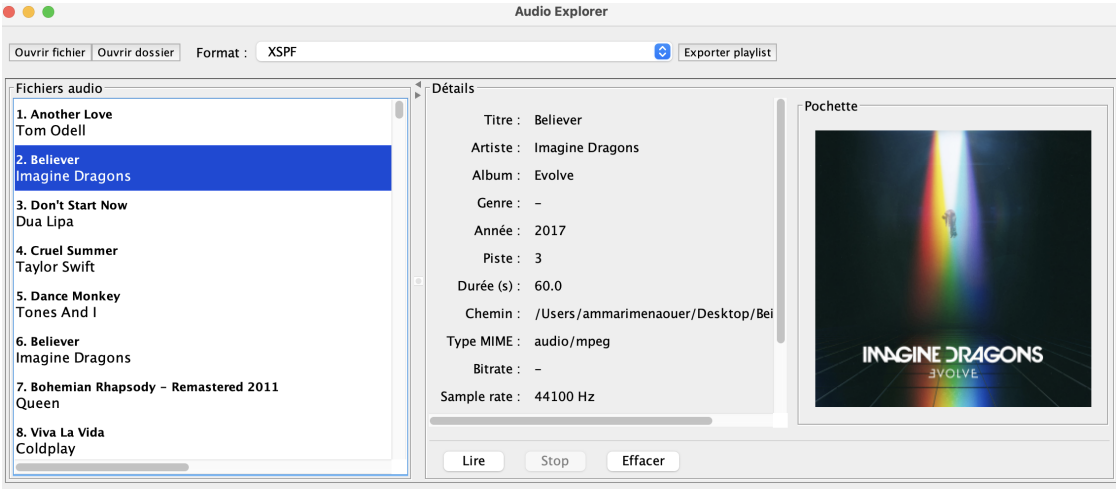

FIGURE 5 – Capture d'écran de l'interface graphique.

7 Tests de robustesse de l'application

Pour vérifier la robustesse de l'application, nous avons mis en place une série de tests couvrant les cas réels et les situations d'erreur. Le filtrage des MP3 a été validé avec des fichiers aux extensions trompeuses et des formats non audio, afin de confirmer que la détection par extension et par type MIME reste fiable. Nous avons aussi testé la robustesse des entrées et sorties en utilisant des dossiers avec des droits restreints, des chemins longs ou contenant des caractères spéciaux, ainsi que des fichiers absents ou corrompus, pour s'assurer que l'application gère correctement les exceptions sans interrompre l'exécution. Des tests ont également porté sur les encodages (UTF-8, accents, caractères non latins) afin de garantir un affichage correct des métadonnées et des noms de fichiers. Enfin, l'extraction et l'affichage du cover art *ont été testés sur des MP3 contenant des images de tailles et de formats variés, ainsi que sur des fichiers sans image intégrée, pour vérifier un comportement stable et cohérent dans tous les cas.*

8 Défis techniques

Le projet a soulevé plusieurs défis techniques majeurs liés au traitement de fichiers audio et à la fiabilité globale de l'application. D'abord, il a fallu garantir un filtrage MP3 robuste en combinant une vérification par extension (`.mp3`) et une détection par type MIME (`audio/mpeg`) via analyse de contenu (signature/"magic bytes") pour éviter les faux positifs (ex. fichiers renommés). Ce point a été étendu à d'autres formats audio potentiels rencontrés lors des scans (ex. `audio/flac`, `audio/wav`, `audio/aac`) afin de gérer correctement les cas mixtes et les fichiers non conformes.

Ensuite, la robustesse I/O a constitué un enjeu central : parcours récursif de l'arborescence avec une profondeur maximale contrôlée, gestion des erreurs de lecture (`AccessDenied`, chemins trop longs, liens symboliques), tolérance aux fichiers partiellement corrompus, et continuité du traitement sans arrêt global (stratégie "fail-safe" par fichier). Un autre défi important concernait l'encodage : prise en charge cohérente de l'UTF-8, des caractères accentués et des métadonnées ID3 (ID3v1/ID3v2.x), en évitant les problèmes de décodage lors de l'affichage GUI/CLI et lors des exports.

Enfin, la génération et la validation des playlists ont demandé une attention particulière : export XSPF (XML), JSPF (JSON) et M3U8 (UTF-8) avec contrôle de conformité (balises, échappements, encodage) et vérification structurelle (nombre de titres/lignes exportées, ordre, chemins relatifs/absolus) pour garantir qu'une playlist reflète exactement les fichiers détectés lors du scan, y compris dans des arborescences profondes.

9 Bibliothèques utilisées

- *lecture et parsing des métadonnées ID3 :* `Jaudiotagger` *(extraction titres, artistes, album, genre, année, track, etc.) ;*
- *extraction/gestion du* cover art *: via* `Jaudiotagger` *(APIC/ID3v2) + conversion en* `BufferedImage`/`ImageIcon` *pour l'affichage ;*
- *GUI : Swing (fenêtres, menus, tableaux, filtres, aperçu cover art) ;*
- *lecture audio (bibliothèque de lecture de musique) :* `JavaFX Media` *(*`Media`/`MediaPlayer`*) ou* `JLayer` *(MP3) pour la lecture des fichiers audio.*

10 Gestion du code et livrables

10.1 Gestion de versions

Nous avons utilisé ***Git/GitHub*** *pour sauvegarder l'historique du code, suivre chaque modification via des* ***commits*** *et faciliter la collaboration surtout pout fusionner le code .*

11 Fonctionnalités non implémentées

Toutes les fonctionnalités demandées ont été implémentées. Des améliorations restent toutefois possibles optimisations Algorithmique , tri/recherche avancés, interface plus ergonomie GUI, lecteur audio plus complet .

12 Leçons apprises et perspectives

Ce projet a mis en évidence plusieurs enseignements essentiels liés à la conception, au développement et à l'évolution d'une application logicielle.

- ***UML et architecture avant le code :*** *la définition préalable de l'architecture logicielle à l'aide de diagrammes UML s'est révélée indispensable pour structurer efficacement le projet, clarifier les rôles des classes et garantir la cohérence globale avant la phase d'implémentation.*
- ***Tests sur des données réelles :*** *l'utilisation de données réelles lors des phases de test a permis de valider le bon fonctionnement de l'application, de détecter des cas particuliers et d'améliorer la robustesse du programme.*
- ***Évolutions futures :*** *des améliorations pourront être apportées ultérieurement, notamment l'ajout de fonctionnalités de filtrage, de tri et de recherche, ainsi que l'intégration d'un lecteur audio en tant que fonctionnalité complémentaire.*

13 Conclusion

Ce projet nous a fait progresser à la fois sur le plan ***technique*** *et* ***méthodologique****. D'un point de vue technique, il nous a permis de consolider une pratique réelle de la POO en Java à travers une application complète : modélisation des entités (piste, métadonnées, playlist), mise en place de services réutilisables (lecture ID3, exploration récursive, export) et génération de playlists conformes aux formats* ***XSPF****,* ***JSPF*** *et* ***M3U8****. Cette réalisation nous a aussi obligés à traiter des aspects essentiels souvent sous-estimés : robustesse des entrées/sorties, gestion de données manquantes ou hétérogènes, cohérence d'encodage (UTF-8) et fiabilité du filtrage MP3. Sur le plan méthodologique, le projet a renforcé une démarche de développement « justifiable » : partir de l'UML pour stabiliser l'architecture, découper en modules à responsabilités claires, définir des contrats de services communs à la* ***CLI*** *et à la* ***GUI****, puis intégrer progressivement sans perdre la cohérence globale. Enfin, le travail en binôme a amélioré notre organisation et notre synchronisation (répartition des tâches, décisions partagées, maîtrise croisée du code), se rapprochant d'une démarche professionnelle où la qualité dépend autant de l'architecture et de la rigueur que des fonctionnalités.*