TOULOUSE INP N7

ENSEEIHT

Rapport Final - Projet long TOB
Logiciel de montage vidéo

Equipe KL-08 :
Alexandre Lescot
Camille Meyer
Doryan Benoit
Iman-Norr Draou
Mathier Trahand
Oscar Mautin
Sophie Girardot
Yacine Meziani

Professeur :
G. Dupont

May 29, 2025

Contents

1 Objectif du projet

Ce projet vise à développer une application de montage vidéo intuitive et performante, accessible aussi bien aux professionnels qu'aux débutants. L'objectif est de proposer un outil ergonomique permettant d'importer, d'éditer et de manipuler facilement du contenu multimédia.

2 Préparation du projet

2.1 MVP

Nous avons choisi le patron de conception Model-View-Controller (MVC) car c'est celui qui s'adapte le mieux au contexte de notre projet.

2.1.1 Timeline

La timeline constitue la base du logiciel de montage. Il s'agit d'une frise chronologique sur laquelle les utilisateurs peuvent organiser leurs fichiers multimédias (vidéos, audios, images, textes). L'objectif est d'intégrer un système de *drag and drop* permettant d'ajouter du contenu à la timeline et de le visualiser en temps réel dans la *preview*.

Il sera également possible de manipuler les éléments sur la timeline : déplacer une vidéo de quelques secondes, assembler plusieurs fichiers ou ajuster leur durée. Les pistes vidéo (vidéos, images, textes) seront distinctes des pistes audio pour une meilleure gestion du contenu.

2.1.2 Preview

Un espace de prévisualisation sera intégré dans l'interface, permettant aux utilisateurs de visualiser les modifications effectuées sur la timeline en temps réel. La prévisualisation sera située dans la partie supérieure droite de la fenêtre, selon la conception définie.

2.1.3 Importation de vidéos

L'application offrira la possibilité d'importer des vidéos à partir du stockage local. Les utilisateurs pourront parcourir leurs documents et sélectionner un fichier à intégrer dans leur projet.

2.1.4 Gestion des fichiers importés

Une bibliothèque de médias affichera les fichiers importés, facilitant leur accès et leur organisation. Cet espace sera situé en haut à gauche de l'interface, conformément au design prévu.

2.1.5 Ajout de vidéos à la timeline

Les vidéos importées pourront être ajoutées à la timeline pour être manipulées. L'utilisateur pourra sélectionner un point précis de la vidéo à afficher en *preview*, à l'aide d'un curseur dédié. D'autres actions de modification seront ajoutées ultérieurement.

2.1.6 Lecture et manipulation des fichiers audio et vidéo

Les fichiers vidéo et audio présents sur la timeline seront lisibles directement via la *preview*, permettant un contrôle précis du montage.

2.2 Fonctionnalités intermédiaires

2.2.1 Découpage de vidéos

L'application permettra de découper une vidéo en plusieurs segments directement sur la timeline, à l'aide d'un curseur de sélection. Chaque segment pourra être déplacé indépendamment, supprimé ou réorganisé.

2.2.2 Fusion de vidéos

Les utilisateurs pourront assembler plusieurs vidéos pour en faire un seul fichier, facilitant le montage et l'édition de contenus combinés.

2.2.3 Édition et lecture des fichiers audio

Un fichier audio pourra être manipulé dans la timeline de la même manière qu'une vidéo. L'utilisateur pourra écouter un fichier, ajuster son placement et effectuer des modifications basiques.

2.2.4 Ajout d'images à la timeline

Les images pourront être intégrées dans la timeline et manipulées comme des vidéos, avec la possibilité de les visualiser dans la *preview*.

2.2.5 Exportation du montage

L'application offrira une option d'export permettant de regrouper l'ensemble des éléments montés sur la timeline en un fichier final unique (ex. MP4).

2.3 Fonctionnalités supplémentaires

2.3.1 Ajout de texte à la timeline

L'application permettra d'ajouter du texte personnalisable sur la timeline (possibilité de personnalisation de la police, la taille, la couleur, etc). L'utilisateur pourra définir la durée d'affichage du texte et le manipuler comme une vidéo ou une image. Ce texte pourra être positionné et édité.

2.4 Croquis

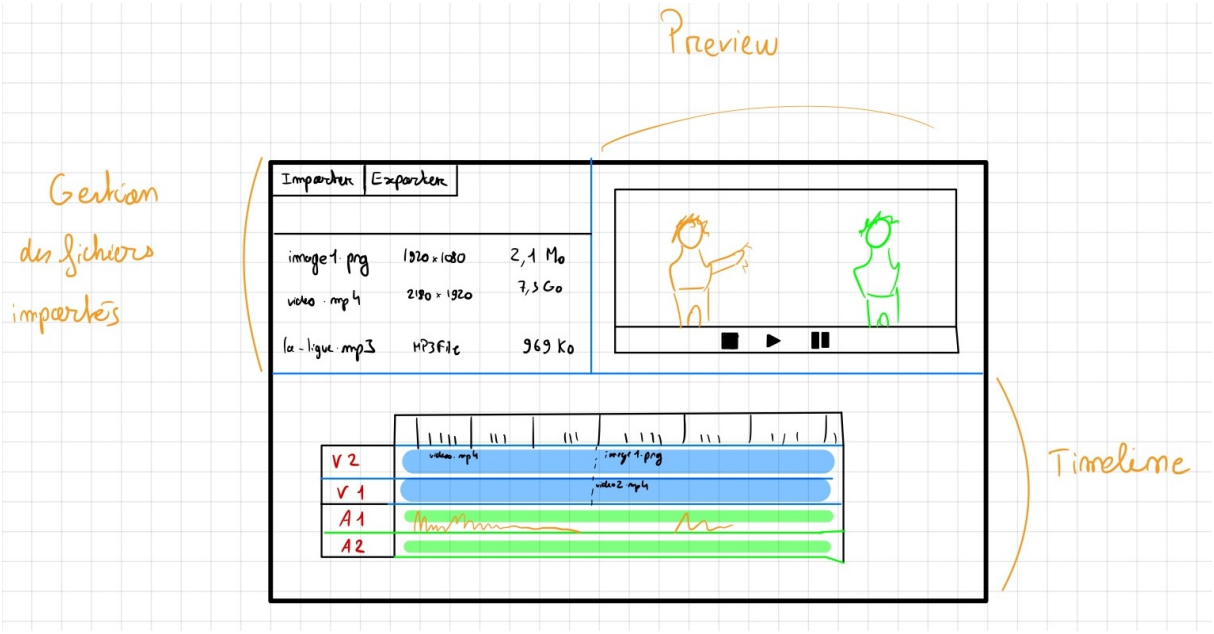

Figure 1: Premier croquis de l'application

3 Architecture de l'application

3.1 Introduction à la conception de l'application

L'interface utilisateur (UI) de notre logiciel de montage est développée à l'aide de **JavaFX**, un framework Java dédié à la création d'interfaces graphiques modernes et modulaires. Conformément aux bonnes pratiques enseignées durant notre cursus, nous avons adopté le **patron de conception Modèle-Vue-Contrôleur (MVC)** afin d'assurer une séparation claire entre la gestion des données, la présentation graphique et la logique applicative.

La couche **Vue** est majoritairement conçue en utilisant des fichiers **FXML**, un langage déclaratif XML dédié à la description des interfaces JavaFX. L'utilisation de FXML nous permet de simplifier considérablement la création et la maintenance des interfaces graphiques. Elle offre également la possibilité d'employer l'outil **SceneBuilder**, facilitant la conception visuelle des fenêtres et panneaux de l'application. Une formation rapide sur SceneBuilder a été assurée à tous les membres de l'équipe afin d'harmoniser la production des vues et accélérer le développement.

La couche **Contrôleur** est naturellement intégrée à JavaFX, qui gère automatiquement la liaison entre les fichiers FXML et leurs contrôleurs associés. Pour chaque vue, il suffit de préciser dans le fichier FXML le contrôleur correspondant, ce qui permet à la classe `FXMLLoader` de faire le lien entre les deux. Cette fonctionnalité simplifie grandement la gestion des événements et des interactions utilisateur. Ainsi, pour chaque panneau principal de l'application — tels que la prévisualisation (Preview), l'importation des médias (Importation), la timeline (Timeline) et l'inspecteur de propriétés (Inspecteur) — un fichier FXML dédié est associé à un contrôleur spécifique (par exemple, `Import.fxml` est lié à la classe `ImportController.java`).

La couche **Modèle** est centralisée autour d'une classe principale nommée `VideoProject`, qui représente l'état global du projet vidéo d'un utilisateur. Cette classe contient toutes les informations pertinentes au projet en cours, telles que les clips importés, la structure

temporelle, et les paramètres d'exportation. Tous les contrôleurs accèdent à cette instance pour lire ou modifier les données. Par ailleurs, le modèle est réparti sur plusieurs classes spécialisées afin de structurer clairement les différents aspects fonctionnels. Par exemple, pour la gestion des clips vidéo, la classe `VideoProject` délègue l'accès à un registre de clips nommé `ClipRegistry`, qui gère spécifiquement l'import, le stockage et la récupération des clips. Cette organisation modulaire du modèle permet de conserver une architecture claire, facilitant l'évolution et la maintenance du logiciel.

3.2 Découpage de l'application en sous-systèmes (paquetages)

3.3 Découpage de l'application en sous-systèmes (paquetages)

Le découpage en paquetages a constitué une étape importante et relativement complexe de la conception. Initialement, il n'était pas évident de déterminer si l'organisation devait se faire selon les couches classiques du patron MVC (modèle, vue, contrôleur) ou plutôt en fonction des fonctionnalités principales de l'application. Après plusieurs réflexions et discussions au sein de l'équipe, nous avons convenu d'adopter une architecture hybride qui combine ces approches.

Le paquetage racine `com` contient les éléments de base utilisés de manière transversale dans l'ensemble du projet. Ce paquet regroupe notamment les classes centrales suivantes :

- `VideoProject` : classe principale du modèle représentant l'état global du projet vidéo.
- `Main` : classe principale lançant l'application.
- `MainController` : classe responsable de l'initialisation de tous les contrôleurs et de l'attribution de l'instance `VideoProject` à chacun d'eux.
- `Clip` : classe définissant la notion de clip vidéo, élément fondamental du logiciel utilisé à travers plusieurs modules.

En parallèle, nous avons structuré l'application en plusieurs paquetages dédiés aux panneaux (ou *panels*) de l'interface graphique, correspondant aux fonctionnalités principales :

- `Timeline` : gestion de la timeline et des éléments temporels.
- `Import` : gestion de l'importation des médias et ressources.
- `Preview` : gestion de la prévisualisation des vidéos montées.

Enfin, les fichiers `.fxml` correspondant aux vues JavaFX sont organisés dans un dossier `ressources/views`, séparé des classes Java, ce qui facilite la maintenance et la clarté de l'arborescence du projet.

3.4 Diagrammes de classe

4 Réalisation du projet

4.1 Conception [It. 0]

4.2 Mise en place de l'environnement [It. 0 et 1]

L'itération 1 a principalement porté sur la mise en place de l'environnement de développement, une phase qui a pris plus de temps que prévu en raison de la diversité des systèmes d'exploitation utilisés par les membres de l'équipe.

4.2.1 Spécification initiale et planification

Nous avons tout d'abord formalisé l'ensemble des fonctionnalités, rédigé en LaTeX. Le code source de cette spécification se trouve dans le répertoire `docs/fonctionnalites`. À partir de ce référentiel, un tableau Kanban a été mis en place pour visualiser l'avancement, et des *epics* ont été créés pour chacune des fonctionnalités définies, afin de faciliter le découpage et le suivi des tâches.

4.2.2 Choix et configuration des bibliothèques

Pour la partie interface utilisateur, nous avons retenu `JavaFX` (avec *Scene Builder*) et pour le traitement bas-niveau de la vidéo, `GStreamer`. L'installation de `GStreamer` s'effectue sans difficulté sous Linux et macOS, mais s'est avérée instable sur Windows pour des raisons indéterminées. Afin d'uniformiser l'environnement et d'éviter les incompatibilités, nous avons opté pour une solution de conteneurisation.

4.2.3 Conteneurisation via Docker

Le recours à Docker s'est imposé pour assurer la portabilité et la cohérence de notre environnement de développement, indépendamment du système hôte :

- **Isolation des dépendances** : chaque conteneur embarque les versions exactes de JavaFX, GStreamer et de leurs bibliothèques associées, évitant ainsi les conflits de versions ou de configuration entre Windows, Linux et macOS.
- **Reproductibilité** : grâce à un Dockerfile, tout membre de l'équipe peut reconstruire, à l'identique, l'image de développement avec un simple `docker build`, garantissant que le programme de test `FXPlayer` fonctionne de manière identique sur toutes les machines.
- **Simplicité de déploiement** : la conteneurisation évite l'installation manuelle de GStreamer sur l'hôte Windows, problème dont la cause exacte (versions de DLL manquantes, conflits de chemin, etc.) n'a pas pu être résolue en temps utile.

La création d'un Dockerfile pleinement opérationnel a néanmoins présenté plusieurs challenges :

1. **Choix de l'image de base** : après plusieurs itérations, nous avons retenu `openjdk:17-jdk-slim` pour sa légèreté et sa compatibilité JavaFX ; un premier prototype basé sur `ubuntu:22.04` était trop volumineux et nécessitait trop de traitements de nettoyage.
2. **Installation de GStreamer et plug-ins** : il a fallu identifier et installer les paquets `gstreamer1.0-plugins-base`, `-good`, `-bad` et `-ugly`, ainsi que leurs dépendances système (libc, libglib), puis configurer `LD_LIBRARY_PATH` pour que JavaFX puisse les localiser.
3. **Gestion des volumes et des permissions** : pour monter un répertoire de projet en lecture/écriture, nous avons dû ajuster les UID/GID dans le conteneur et ajouter un utilisateur non-root afin de respecter les bonnes pratiques de sécurité.
4. **Optimisation de l'image** : un build multi-étapes a été mis en place pour séparer la compilation Java et l'exécution, réduisant considérablement la taille finale de l'image (passée de 1,2 Go à 400 Mo).
5. **Validation automatique** : enfin, un script d'entrée a été développé pour lancer `FXPlayer` au démarrage du conteneur et signaler par code de sortie le succès ou l'échec de l'initialisation des bibliothèques.

4.2.4 Gestion du dépôt et workflow collaboratif

Compte tenu des contraintes d'accès au réseau INP (nécessité d'un VPN), le code source principal est hébergé sur GitHub. À intervalles réguliers, un mirroring automatique est effectué vers notre instance GitLab interne. Pour limiter les conflits de fusion, chaque membre utilise une branche dédiée pour ses développements. Une fois le travail terminé et validé localement, une demande de fusion (*merge request*) est émise vers la branche `main`, après revue du code et tests unitaires.

4.3 Développement des fonctionnalités principales (MVP) [It. 2]

Lors de cette deuxième itération, plusieurs fonctionnalités clés ont été développées ou finalisées, tout en amorçant une structuration plus claire de l'architecture logicielle. Le travail a été réparti autour de trois axes principaux : la gestion des imports et des clips, l'intégration de la timeline, et la mise en place d'une prévisualisation fonctionnelle.

4.3.1 Prévisualisation et gestion des flux

Une première version du système de prévisualisation a été implémentée. Elle repose sur une pipeline qui permet de récupérer une vidéo d'exemple en ligne. Les flux audio et vidéo sont désormais gérés dynamiquement, et connectés aux sorties (écran et haut-parleur) au moment de l'exécution.

4.3.2 Modélisation du modèle de données

Un diagramme de classes a été produit afin de modéliser la structure de la partie *model* du logiciel. Deux classes principales ont été introduites :

- `Clip`, définie sous forme d'interface, représentant une séquence vidéo ou audio.

- `VideoProject`, une classe centrale qui regroupe les informations relatives aux vidéos importées ainsi qu'à la timeline du projet.

4.3.3 Importation de vidéos

La fonctionnalité d'import a été finalisée. Elle comprend :

- Un bouton pour importer des vidéos via une fenêtre système.
- Une `TableView` listant les vidéos importées avec leurs métadonnées (miniature, nom, taille, dimensions, durée, date de création).
- Une barre de recherche permettant de filtrer les vidéos par nom.
- Un bouton pour supprimer une vidéo de la liste.
- Une fonctionnalité de glisser (*drag*) affichant la miniature de la vidéo (sans gestion du *drop* à ce stade).

4.3.4 Création de clips depuis les vidéos importées

Un travail spécifique a été mené sur la création d'un objet `Clip` à partir d'une vidéo :

- Implémentation de la classe `GStreamerVideoImporter`, avec la méthode `importVideo` qui crée un `ImportedClip` à partir d'une URI.
- Extraction d'informations locales (nom, type) sans passer par GStreamer.
- Utilisation de GStreamer pour récupérer la durée, la taille et une miniature (prise à 5% ou à 1 seconde).
- Redimensionnement de la miniature pour une taille standard.

4.3.5 Travail sur la timeline

Des éléments visuels ont été ajoutés pour la timeline :

- Réservation d'espaces pour l'échelle de temps, le chronomètre et les contrôles.
- Création des boutons `addTrack`, `supprimerTrack`, et `séparer` (seul le premier est fonctionnel).
- Ajout d'un champ texte pour le chronomètre (non encore lié aux données).

4.3.6 Implémentation de la classe `Track`

Les méthodes suivantes ont été développées :

- `addTimelineObject` : insertion ordonnée selon le début des clips.
- `getObjectAtTime` : identification de l'objet actif à un instant donné.
- `changeTimelineName` : renommage de la piste.
- `getTotalDuration` : durée totale de la piste.
- `enableDrop` : amorce de la gestion du glisser-déposer.

4.3.7 Ajout d'un curseur temporel

Un curseur permettant de naviguer dans la timeline a été créé :

- Ajout de l'affichage dans le fichier FXML dédié.
- Mise en place d'une nouvelle classe `Curseur` pour gérer ce composant.
- Le curseur est visible mais non encore synchronisé avec l'affichage de la prévisualisation.

4.3.8 Affichage des `TimelineObject` dans la timeline

La fonctionnalité d'affichage des clips dans la timeline a été finalisée en deux étapes :

- Création d'un `TimelineObject` (initialement à partir d'un clip test avec des métadonnées et une miniature factice).
- Affichage du `TimelineObject` dans la timeline sous forme d'un rectangle dont la taille et le texte varient en fonction de la durée et du nom du clip.

4.3.9 Déplacement des `TimelineObject` par *Drag&Drop* [It. 2 et 3]

Après avoir réussi à afficher les objets, la possibilité de les déplacer dans la timeline a été ajoutée afin de positionner les clips au moment souhaité dans la vidéo. Initialement, les clips pouvaient se superposer lors du déplacement, ce qui était inapproprié. Pour remédier à cela, une forme de « hitbox » a été attribuée aux objets, permettant de bloquer le déplacement lorsqu'un clip rencontre un autre. Lorsque la souris s'éloigne suffisamment, l'ordre des clips peut s'inverser. Plusieurs solutions ont été testées avant d'adopter celle-ci, notamment l'interrogation du modèle pour vérifier les chevauchements, mais cela engendrait un délai bloquant les objets concernés. Cependant, un problème persiste : lorsqu'un clip « téléporte » de l'autre côté d'un clip adjacent pour inverser leur ordre, et qu'un troisième clip est proche, une superposition temporaire peut survenir. Ce phénomène, bien que non optimal, ne perturbe pas significativement l'utilisateur et n'entraîne pas de crash.

4.3.10 Ajout de clips dans une piste depuis le panneau d'import

Enfin, une fonctionnalité permettant d'ajouter un clip depuis le panneau d'import vers une piste spécifique de la timeline a été implémentée. L'utilisateur sélectionne un clip dans le panneau d'import, puis utilise un bouton pour choisir la piste d'ajout. Cette fonctionnalité implique la liaison entre deux panneaux distincts : l'import et la timeline. L'ajout dans le modèle est direct, mais la mise à jour de la timeline dans l'interface n'est pas immédiate. Plusieurs solutions ont été explorées, notamment la mise à jour périodique automatique. La méthode retenue utilise des objets `Observable`, plus précisément une `ObservableList` pour les clips importés et les `TimelineObject` affichés. Ainsi, lorsqu'un objet est ajouté à une piste, un événement est déclenché, capté par le contrôleur qui met à jour la vue. Cette approche évite des modifications lourdes dans les diagrammes de classes et préserve les performances de l'application.

4.4 Prévisualisation du pipeline GStreamer [It. 3]

Pour mettre en place la fonctionnalité de prévisualisation, nous nous appuyons sur l'API de **GStreamer**, un framework multimédia modulaire.

GStreamer repose sur un modèle en *pipeline*, dans lequel différents blocs, appelés *filtres* (ou *éléments*), sont chaînés ensemble. Chaque filtre possède :

- des ports de sortie appelés **sources**, qui émettent un flux de données,
- et des ports d'entrée appelés **sinks**, qui reçoivent ce flux.

Les éléments sont connectés entre eux via ces ports afin de former un pipeline fluide de traitement des données audio, vidéo ou autres.

Figure 2: Illustration simplifiée d'un pipeline GStreamer.

Dans cette représentation :

- Le **nœud source** (à gauche) ne possède qu'un port *source*, car il initie le flux.
- Le **filtre intermédiaire** possède un port *sink* pour recevoir les données et un port *source* pour les transmettre.
- Le **nœud sink** (à droite) ne possède qu'un port *sink*, car il termine le traitement du flux.

Dans notre application JavaFX, le nœud *sink* correspond à un objet `ImageView` de JavaFX, connecté à notre pipeline GStreamer.

Nous jouons une *timeline* contenant plusieurs vidéos, donc plusieurs sources potentielles. Cependant, une seule vidéo est affichée à la fois. L'idée est donc de changer dynamiquement la source du pipeline en fonction de l'objet actuellement actif dans la timeline.

Pour cela, nous utilisons un élément spécial de GStreamer appelé `input-selector`. Ce composant permet de connecter un grand nombre de sources et de sélectionner dynamiquement celle que l'on souhaite utiliser en sortie. (Figure 2)

Pour assurer un bon fonctionnement de cette architecture, nous mettons en place un `ScheduledExecutorService`, exposé en tant qu'*Observable*, qui permet de récupérer régulièrement le temps écoulé dans la timeline. Ce mécanisme nous sert à synchroniser le changement de source dans le pipeline avec la lecture courante de la timeline, garantissant ainsi une mise à jour fluide de la prévisualisation.

Nous ajoutons également une source spéciale, appelée **blackSource**, utilisée pour représenter visuellement les sections vides (ou "trous") dans la timeline.

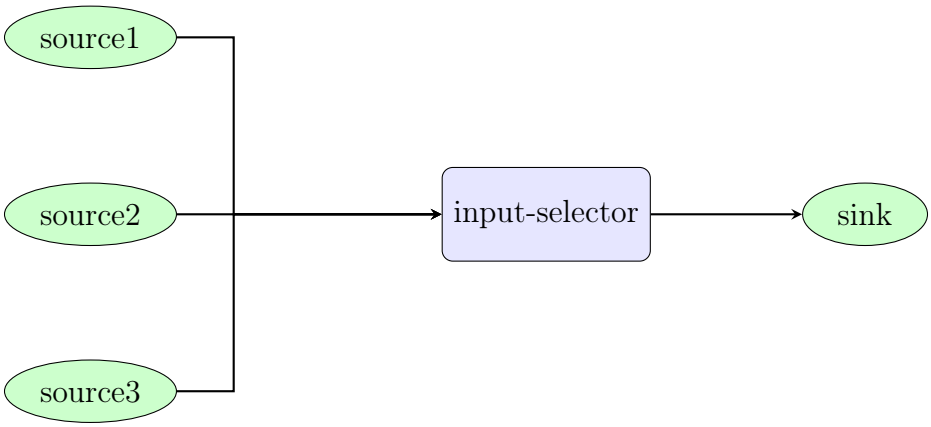

Figure 3: Utilisation de `input-selector` pour choisir dynamiquement une source dans le pipeline GStreamer.

Enfin, afin d'optimiser les performances et éviter toute latence lors des changements de source, nous préchargeons l'ensemble des clips vidéo dans le pipeline avant de lancer la lecture. Grâce aux capacités multithreadées de GStreamer, chaque clip est initialisé en amont et connecté à l'`input-selector`, ce qui permet une commutation instantanée au moment opportun.

4.5 Itération 3 : corrections, optimisations et finalisation

L'itération 3 a principalement consisté en une phase de stabilisation et d'optimisation du logiciel. Aucun ajout majeur de nouvelles fonctionnalités n'a été réalisé, à l'exception de la finalisation de la prévisualisation vidéo. Cette phase a permis de corriger de nombreux bugs, de résoudre plusieurs erreurs conceptuelles identifiées lors des itérations précédentes, et d'améliorer la qualité globale du code.

Parmi les actions entreprises, on compte notamment :

- La correction des comportements imprévus dans la gestion des `TimelineObject`, notamment lors des opérations de déplacement et de superposition des clips.
- L'amélioration de la cohérence entre le modèle et la vue, notamment grâce à un usage plus systématique des structures observables (`ObservableList`), permettant une mise à jour plus réactive et efficace de l'interface.
- La simplification et le nettoyage du code source, avec un refactoring visant à renforcer la lisibilité, la modularité et la maintenabilité.
- L'optimisation des performances dans les parties critiques, notamment la gestion du pipeline GStreamer et l'affichage de la timeline.
- La rédaction et la mise à jour des documents de suivi du projet, en particulier le rapport final et le manuel utilisateur, afin de refléter fidèlement l'état actuel du logiciel et ses fonctionnalités.

Cette dernière itération a ainsi permis de consolider les acquis et d'assurer une base solide pour d'éventuelles évolutions futures du logiciel.

5 Organisation de l'équipe

5.1 Répartition des tâches

	Mathieu Trahand	Doryan Benoit	Yacine Meziani	Iman-Norr Draou	Alexandre Lescot	Sophie Girardot	Oscar Mautin	Camille Meyer
Configuration de GStream et Docker								
Import								
Prewiew								
Timeline								
Rapport								
Presentation								
Diagrame UML								
Manuel utilisateur								

Figure 4: Répartition des tâches

5.2 Mise en place des méthodes agiles

Dans le cadre de notre projet informatique, nous avons adopté une organisation fondée sur les **méthodes agiles**, en mettant en œuvre notamment le framework *Scrum*. Pour cela, nous avons utilisé l'outil **Jira** afin de gérer nos *user stories* et de suivre l'avancement des tâches via un *tableau Kanban*. **Yacine Meziani** a assuré le rôle de *Scrum Master*, tandis que l'ensemble des huit membres de l'équipe, y compris lui-même, occupait à la fois les rôles de **Product Owner** et de **développeur**. La coordination et le suivi de l'avancement se faisaient principalement via **Discord**, tandis que des aspects plus logistiques ou organisationnels étaient discutés sur **Messenger**.

Nous avons notamment utilisé Discord pour réaliser des réunions daily tous les 2 à 3 jours durant les itérations 2 et 3. Ces réunions rapides nous permettaient de faire le point sur l'avancement de chacun et d'éviter que certains membres restent bloqués sur des problèmes spécifiques.

À la fin de chaque itération, nous organisions une réunion plus longue afin de faire un point global sur l'état du projet. À l'issue de la première itération, cette réunion nous a permis de définir collectivement les axes prioritaires pour l'itération suivante, en nous basant sur les retours d'expérience des premières phases du projet. C'est à ce moment-là, à la fin de l'itération 1, que nous avons décidé de mettre en place les réunions "daily" tous les 2 à 3 jours pour faciliter la communication et améliorer notre efficacité.

Après la deuxième itération, nous avons défini les fonctionnalités que nous allions finaliser, ainsi que celles que nous allions abandonner pour la version finale du projet.

Grâce à toutes ces méthodes, nous avons pu bien communiquer et nous organiser efficacement pour développer notre projet. Nous avons constaté une nette amélioration de notre organisation au fil du temps, notamment grâce à la mise en place de ces différents outils et pratiques.