

Université de Rouen



**Mini-Projet Architecture Logiciel**

**DJUBAKA : Un gestionnaire de listes de lecture**

**2019-2020**

**Réalisée par: ZEKRI Fatma**

## Sommaire

<b>1.</b>	<b>Introduction :</b>	3
<b>2.</b>	<b>Architecture globale de l'application:</b>	4
<b>2.1.1.</b>	<b>Architecture éditeur de liste Diagramme de classe:</b>	5
<b>2.1.2.</b>	<b>Patrons mise en œuvre :</b>	6
✓	<b>Patron Composite:</b>	6
✓	<b>Patron AbstractFactory :</b>	7
✓	<b>Patron Visitor :</b>	8
✓	<b>Patron Builder :</b>	9
✓	<b>Patron Façade :</b>	11
✓	<b>Singleton :</b>	12
<b>2.2.</b>	<b>Architecture Lecteur de liste :</b>	13
<b>2.2.1.</b>	<b>Diagramme de classe :</b>	13
✓	<b>Patron Pont :</b>	14
✓	<b>Patron Observer :</b>	15
✓	<b>Singleton :</b>	16
<b>2.2.2.</b>	<b>Patrons non appliquées :</b>	16
<b>3.</b>	<b>Principes S .O.L.I.D :</b>	16

## 1. Introduction :

Le projet consiste à développer un gestionnaire et un lecteur de listes de lecture multimédia qui sont deux programmes différents.

Le premier programme est un éditeur de listes multimédias qui permet à un utilisateur de créer une liste de lecture multimédia qui est composée de fichiers audio, des fichiers vidéo et / où une sous liste médias de même type que la liste parente.

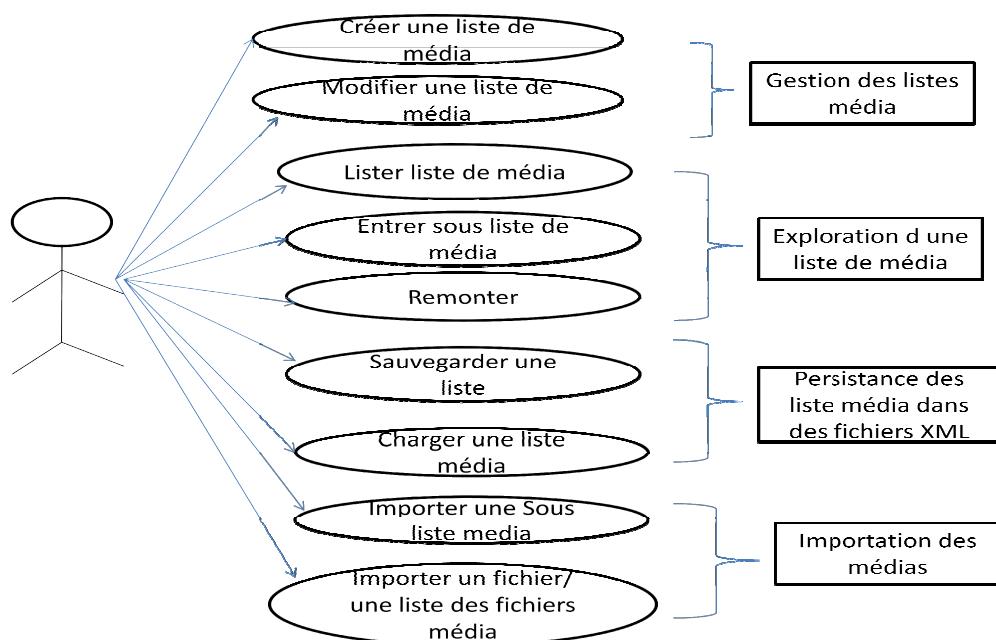
Un utilisateur peut sauvegarder où charger une liste dans où à partir d'un fichier XML et peut aussi importer les différents types de médias.

Le deuxième programme est un lecteur de listes multimédias qui gère la lecture des médias toute en offrant la possibilité de les manipuler (démarrer une lecture, mettre en pause, passer au fichier suivant où liste suivante, revenir au fichier précédent où listes précédentes). Il doit disposer de deux modes d'interactions avec l'utilisateur : mode graphique (avec une interface graphique) et un mode en ligne de commande.

Ce projet donc contient deux principaux modules :

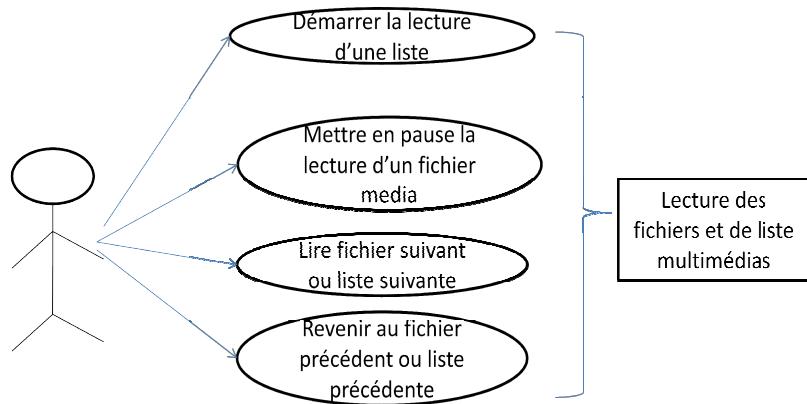
### 1.1. Un Editeur de liste : (en ligne de commande)

- ✓ Gestion des médias
- ✓ Sauvegarder/ Charger des médias en fichier XML
- ✓ Explorer des médias
- ✓ Importer des médias



### 1.2. Un lecteur de liste (interface graphique et ligne de commande)

- ✓ La lecture d'une liste des médias (mode graphique et en ligne de commande)



## 2. Architecture globale de l'application:

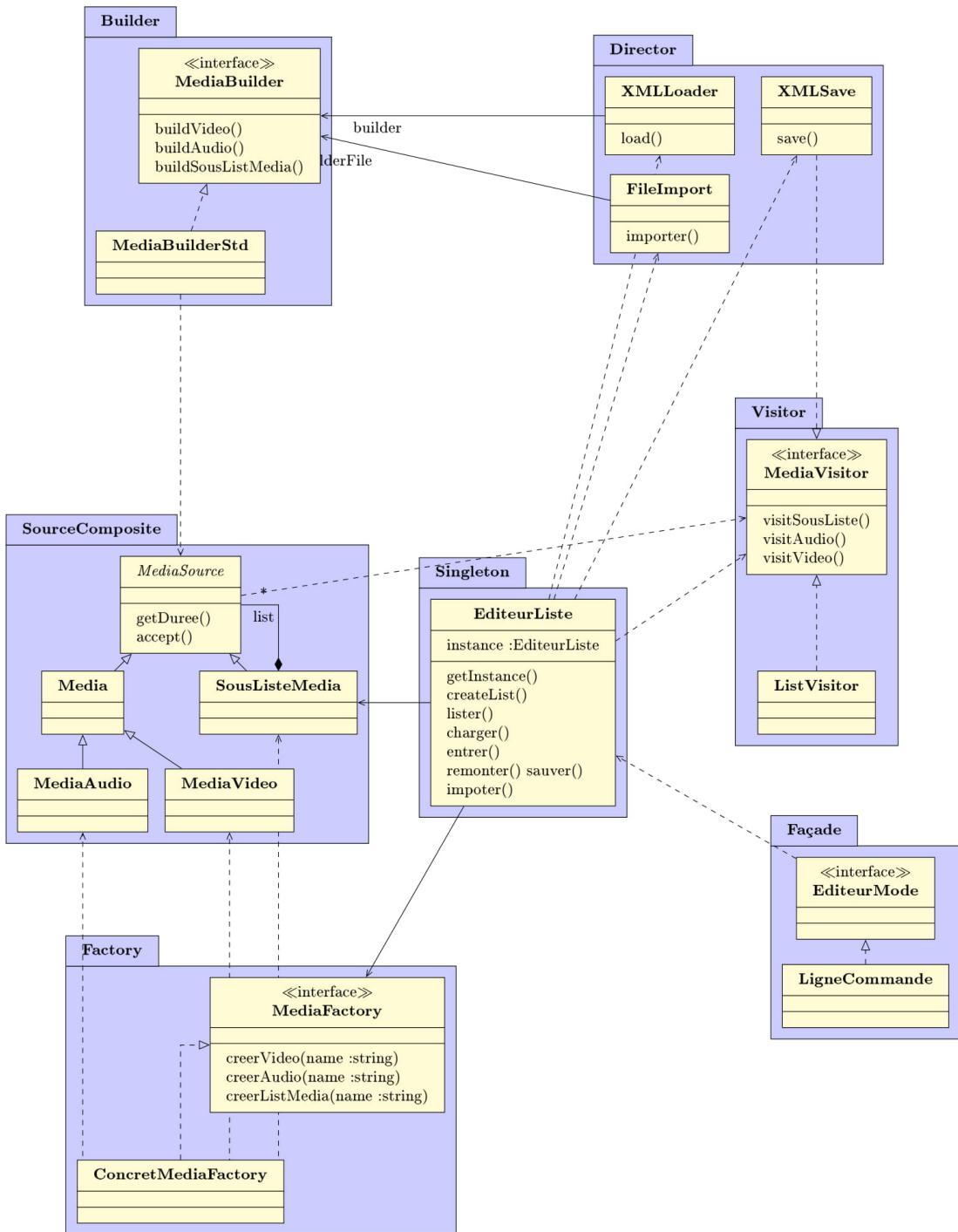
Les besoins exprimés dans l'introduction m'on permis de définir :

Le modèle de l'application est composé de :

- Un média audio : nom de la piste, durée, nom de l'artiste
- Un média vidéo : titre, durée, résolution
- Une sous liste de médias (audio et vidéo): nom, liste de médias
- Un éditeur liste : une liste de media
- Un lecteur liste : une liste de media

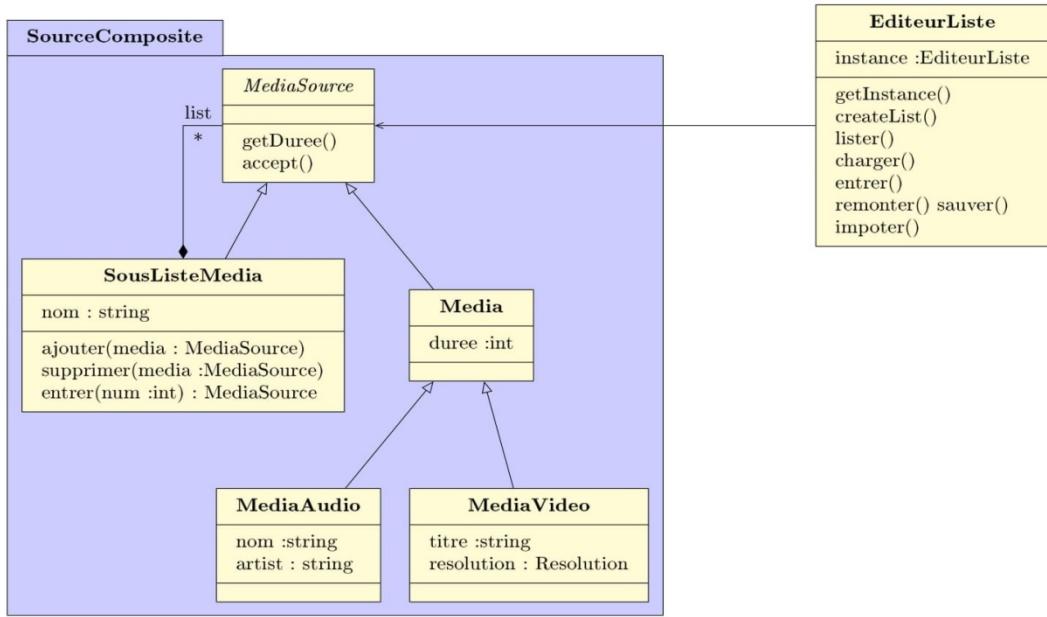
## 2.1.1. Architecture éditeur de liste Diagramme de classe:

1



## 2.1.2. Patrons mise en œuvre :

### ✓ Patron Composite:



### Motivation :

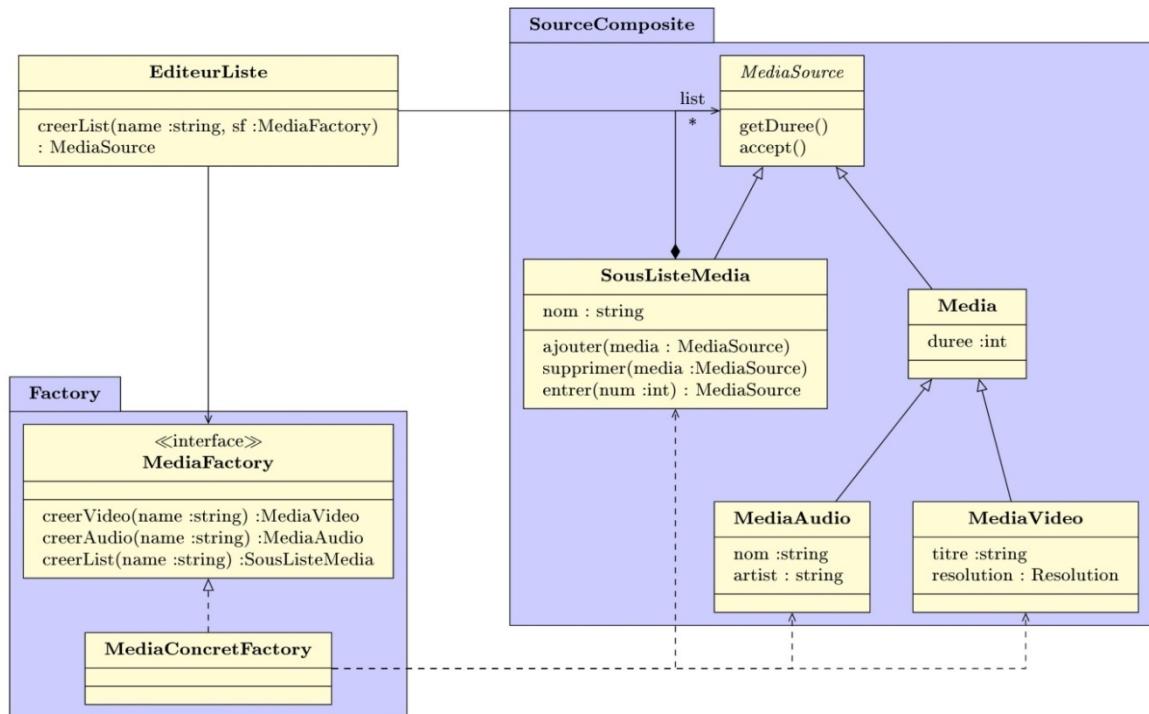
J'ai utilisé le patron composite pour représenter les différents types de média sous forme d'un arbre de : audio (feuille), vidéo (feuille) et la liste de média (nœud) qui est une composition de ces types et d'elle-même.

L'application peut traiter les différents objets du média de façon uniforme et récursive : sans savoir la différence entre les feuilles (**MediaAudio**, **MediaVideo**) et les nœuds de l'arbre (**SousListeMedia**).

### Les Acteurs du patron :

- **Client** : La classe **EditeurListe** manipule une liste de multimédia de façon uniforme
- **Component** : **MediaSource**
- **Leaf** : **MediaAudio**, **MediaVideo**
- **Composite** : **SousListeMedia**

## ✓ Patron AbstractFactory :



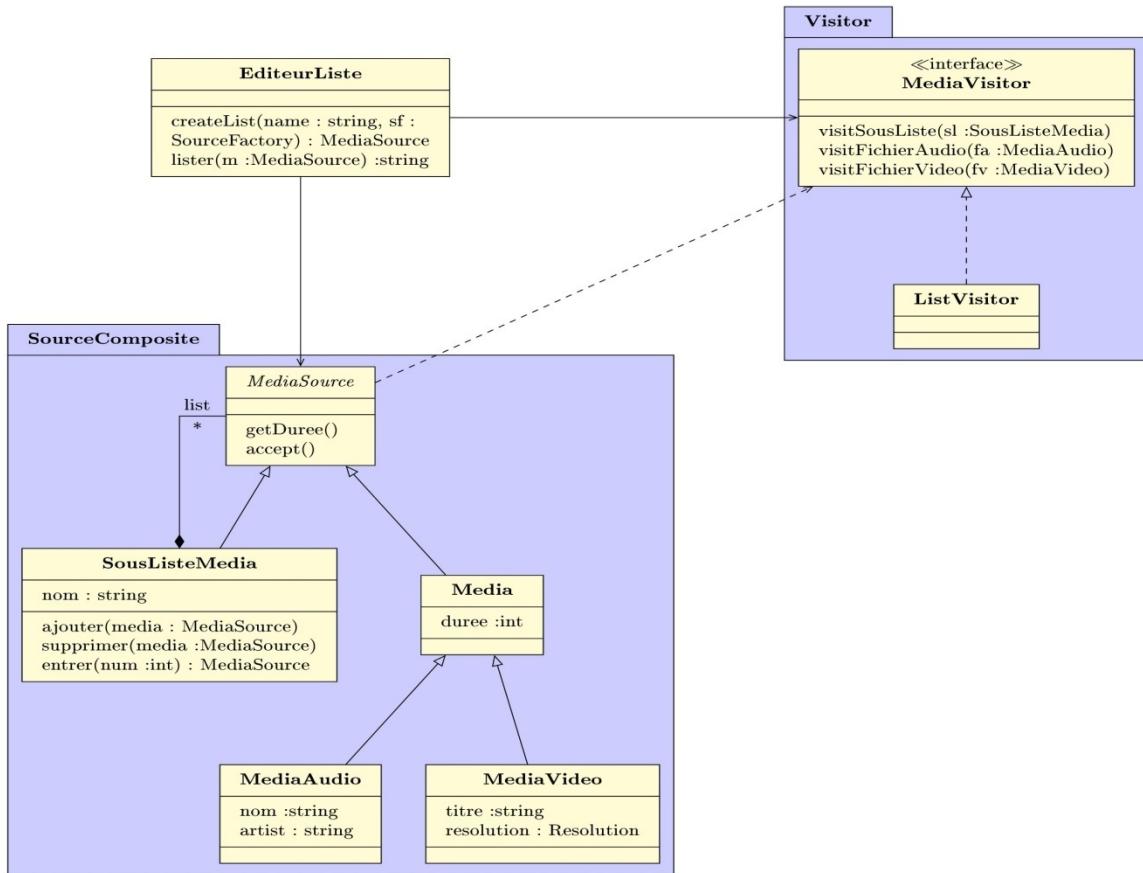
## Motivation :

J'ai utilisé le patron abstract factory pour la création des objets à fin d'isoler l'**EditeurListe** des implémentations des médias et éviter que l'éditeur crée lui-même les différents média(SRP) ce qui facilite l'ajout d'un nouveau type de médias : on peut ajouter un nouveau type de média qui sera créé et récupéré à partir de la classe factory (OCP) .

## Acteurs du patron :

- **Client** : **EditeurListe**
- **AbstractFactory** : **MediaFactory**
- **ConcreteFactory** : **MediaConcreteFactory**
- **AbstractProduct** : **MediaSource**
- **ConcreteProduct** : **MediaAudio**, **MediaVideo**, **SousListeMedia**

## ✓ Patron Visitor :



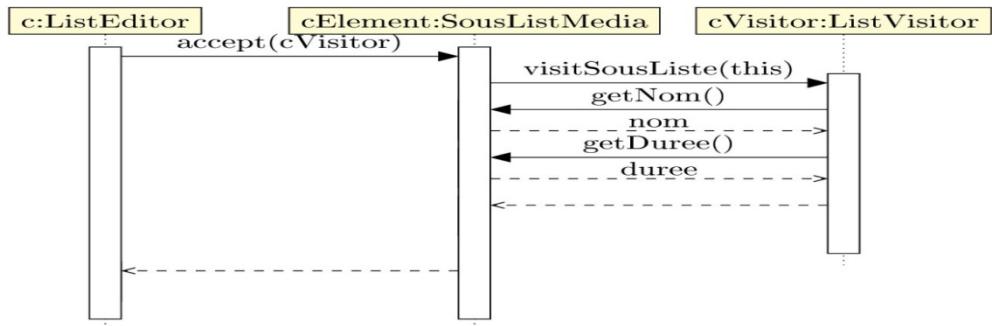
## Motivation :

Puisque j'ai utilisé le patron composite pour représenter les différents objets de média sous forme d'une arborescence, pour la parcourir et traiter ses composants de façon uniforme et surtout récursive il est important d'utiliser le patron visitor.

## Acteurs du patron :

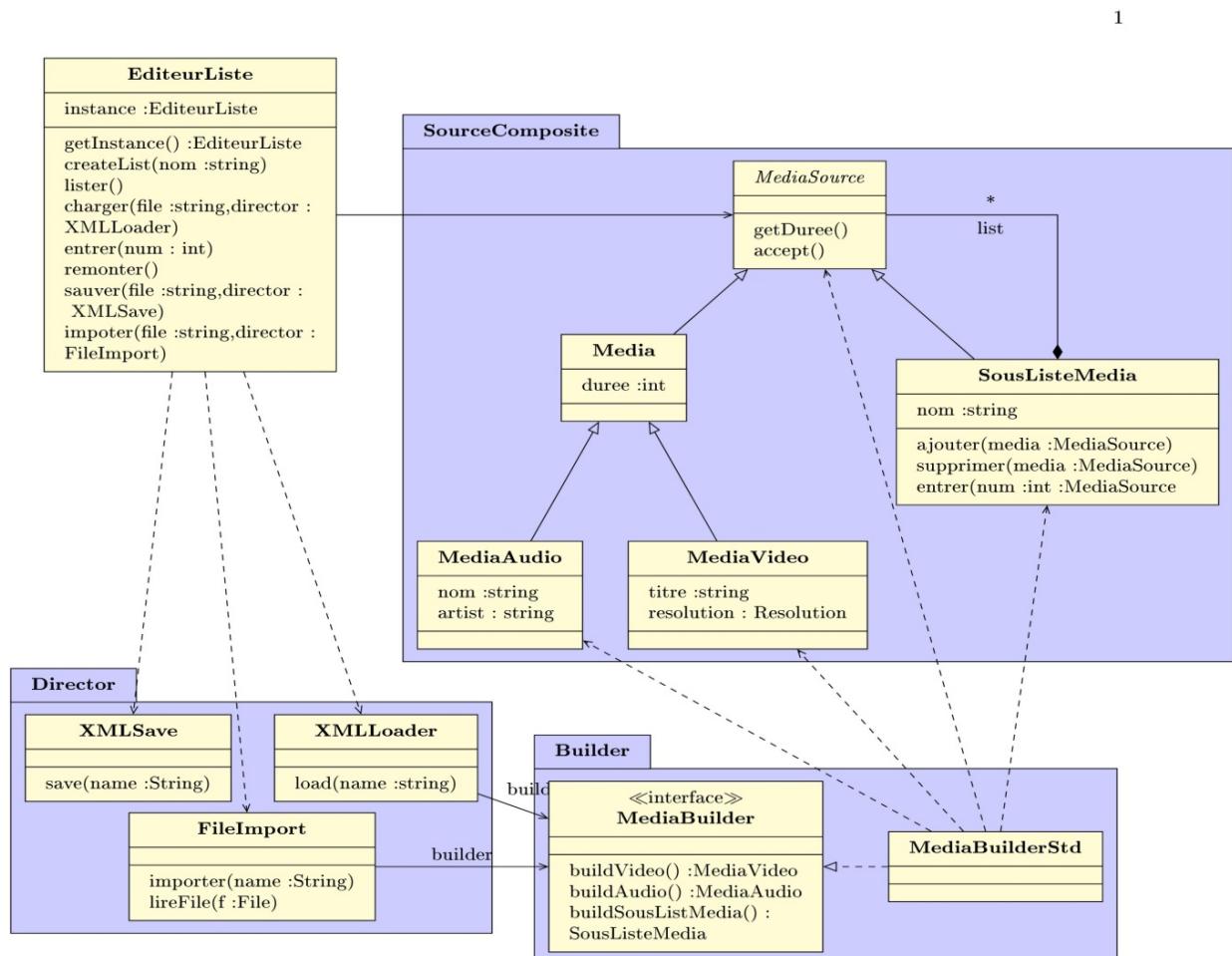
- **Client** : `EditeurListe`
- **Visitor** : `MediaVisitor`
- **ConcreteVisitor** : `listVisitor` : un visiteur pour lister les différents types de médias
- **Element** : `MediaSource`
- **ConcreteElement** : `MediaAudio`, `MediaVideo`, `SousListeMedia` : les classes qui vont être visité par un visiteur pour lister leurs contenus

## Diagramme de séquence :



-Le client ListEditeur demande à une liste de media d'accepter le visiteur ListVisitor pour lister son contenu.

## ✓ Patron Builder :



## Motivation :

J'ai utilisé le patron builder pour construire les objets de type média à partir d'un fichier xml car il sépare le mécanisme de la persistance des objets média de leurs constructions.

Le patron Builder construit le produit étape par étape sous le contrôle du «directeur».

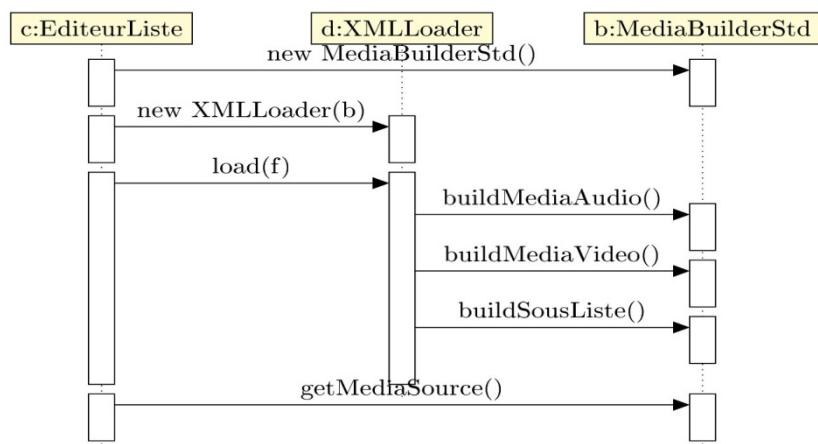
Par exemple le directeur XML lit le fichier xml et récupère les informations nécessaires et invoque l'objet builder pour construire l'objet requis. Ce qui permet de construire des objets média à partir des différents types de fichier xml, html, il suffit juste de changer le directeur XML par un directeur html.

j'ai bien utilisé le même patron builder pour construire les objets médias à partir des fichiers de type mta et mtv.

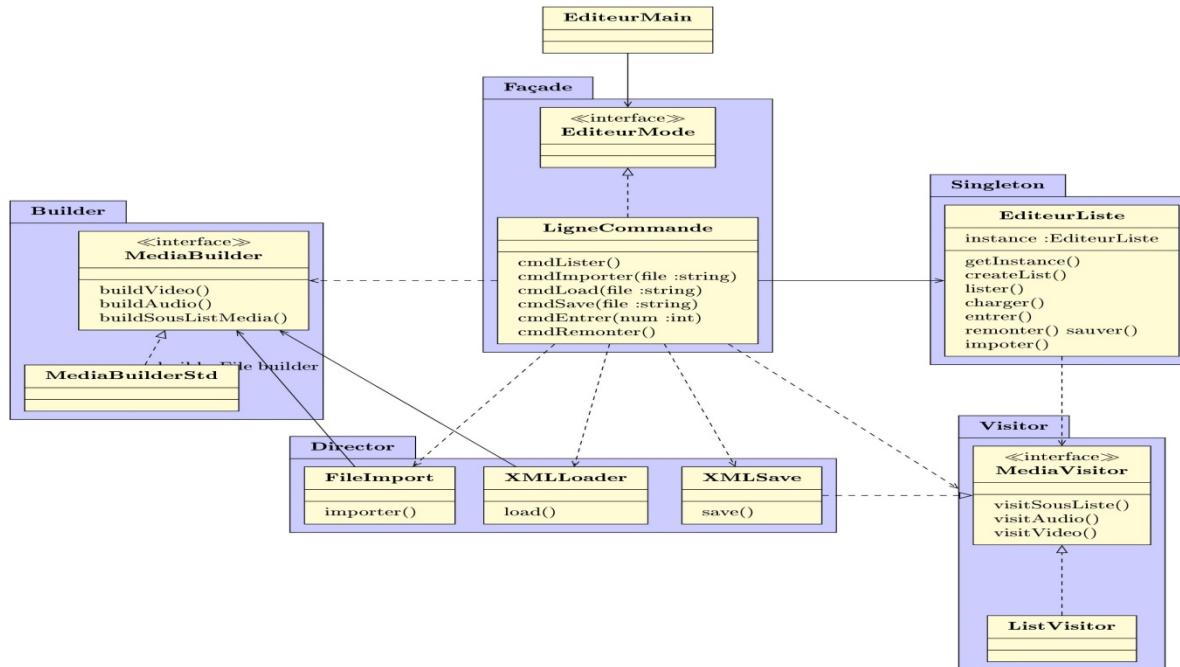
## Acteurs du patron :

- Builder : MediaBuilder
- ConcreteBuilder : MediaBuilderStd
- Director : XMLLoader
- Product : MediaSource

## Diagramme de séquence :



## ✓ Patron Façade :



## Motivation :

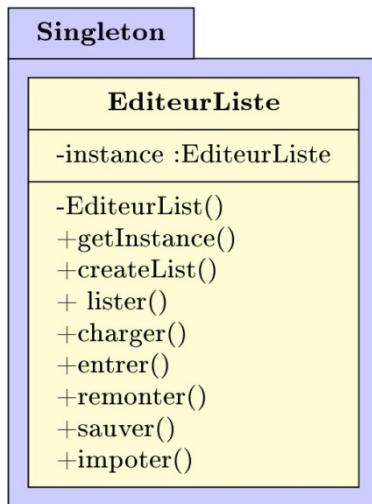
Le programme éditeur liste se compose de plusieurs étapes telle que la création, le chargement, la sauvegarde des médias... Pour chaque étape on doit créer un sous-composant complexe. Par exemple pour charger un media il faut créer un director, un builder , et le produit, cela n'intéresse pas le client qui veut juste charger un média. Pour cela j'ai choisi de fournir au client une classe Façade uniforme qui lui permet d'accéder au système.

Le patron façade masque les fonctionnalités « bas niveau » du client.

## Acteurs du patron :

- **Façade** : `LigneCommande`
- **SubSystem** : `Builder`, `Director`, `Singleton`, `Visitor`

✓ **Singleton :**



**Motivation :**

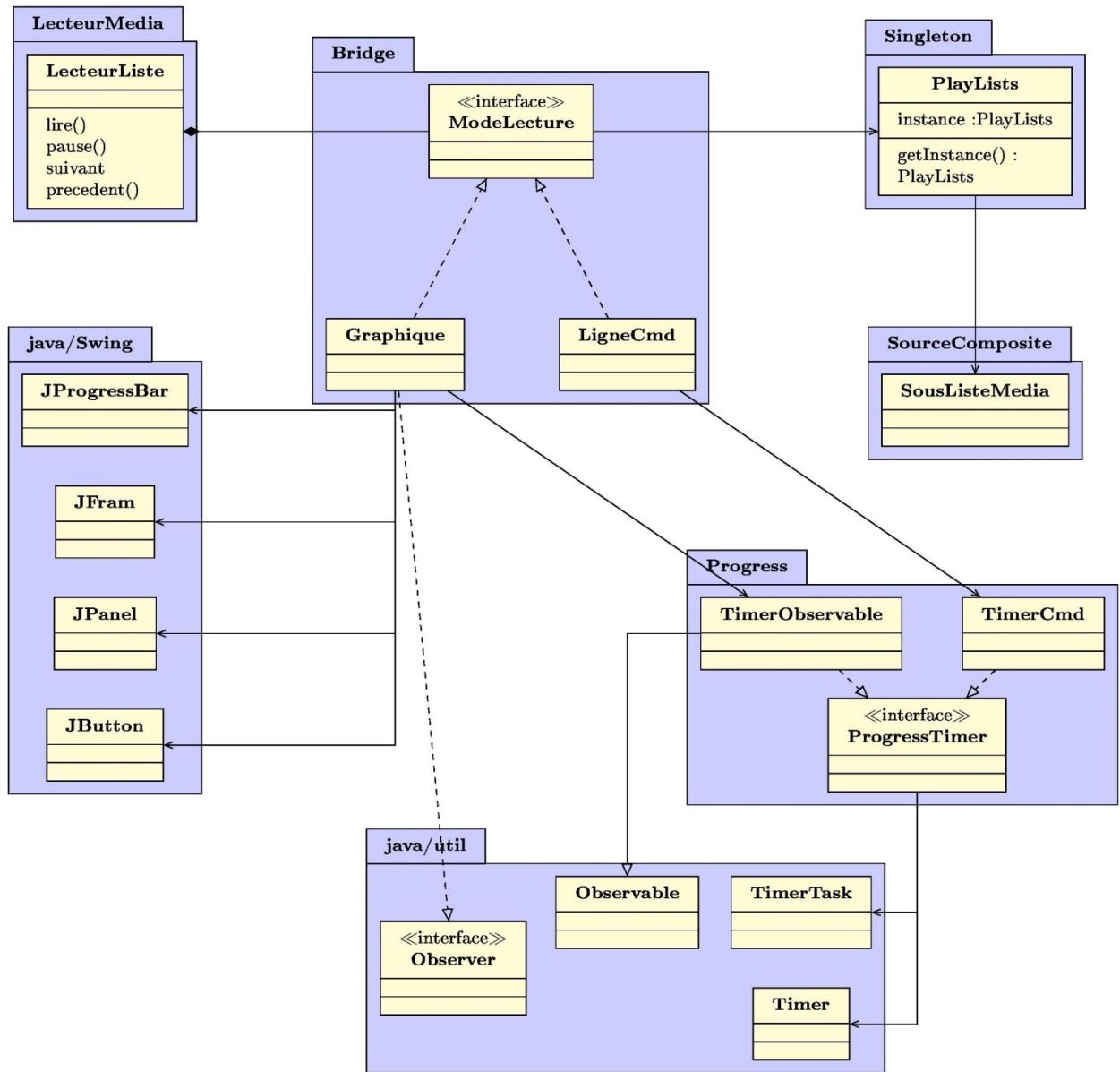
Pour éviter qu'un fichier multimédia soit créé ou modifié par plusieurs éditeurs, j'ai choisi d'appliquer le patron singleton sur la classe **EditeurListe** pour fournir au client une seule instance qui manipule les médias.

**Acteur du patron :**

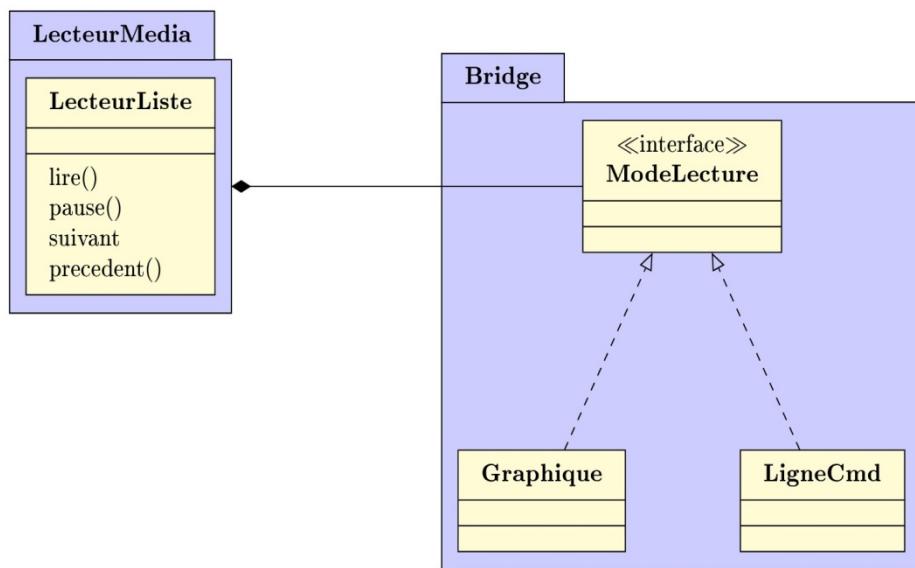
- **Singleton :** **EditeurListe.**

## 2.2. Architecture Lecteur de liste :

### 2.2.1. Diagramme de classe :



### ✓ Patron Pont :



### Motivation :

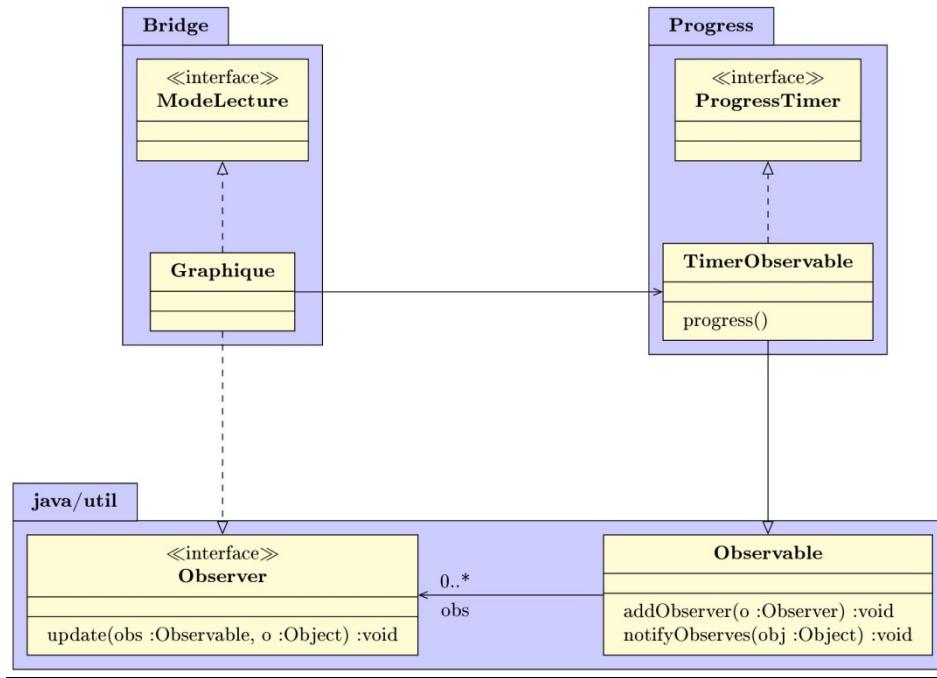
Un lecteur de médias peut être représenté par une interface graphique ou en ligne de commande. Pour permettre à un client de choisir le mode de lecture qu'il le souhaite (graphique ou ligne commande) j'ai appliqué le patron pont qui permet d'extraire la structure lecteur de ses deux implémentations (graphique Ligne de commande.).

Le lecteur délègue le travail aux implémentations qui peuvent varier indépendamment.

### Acteurs du patron :

- **Abstraction** : LecteurListe
- **Implementor** : ModeLecture
- **ConcreteImplementor** : Graphique, LigneCmd

✓ **Patron Observer :**



**Motivation :**

Pour simuler la progression de la lecture d'un média j'ai utilisé l'API Timer avec la classe TimerObservable qui permet de planifier une tache pour une exécution répétée à intervalles réguliers ;

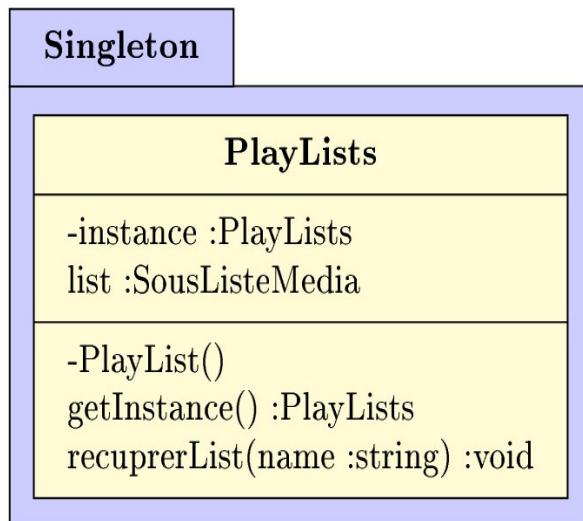
Un lecteur graphique a une barre de progression graphique qui permet de visualiser la progression de la lecture d'un média. Il faut actualiser cette barre par rapport à la progression de lecture de façon qu'à chaque avancement dans la lecture il faut notifier la barre graphique. Pour mettre en place ce système de notification j'ai utilisé le patron Observer.

Le modèle Observer définit une dépendance un à plusieurs entre les objets, de sorte que lorsqu'un objet change d'état, toutes ses dépendances sont notifiées et mises à jour automatiquement.

**Acteurs du patron :**

- **Subject :** Observable
- **ConcreteSubject :** TimerObservable
- **Observer :** Observer
- **ConcreteObserver :** Graphique

✓ **Singleton :**



**Motivation :**

Pour permettre à un client de lire une même liste de médias à partir d'un lecteur graphique et d'un lecteur en ligne de commande j'ai appliqué le patron singleton sur la classe Playlist pour fournir au lecteur graphique et au lecteur en ligne de commande une unique instance de liste.

**Acteur :**

- **Singleton :** PlayLists

**2.2.2. Patrons non appliquées :**

- **Patron Adapté :** il n'y a pas une interface complexe existante à adapter
- **Patron chaîne de Responsabilités :**

**3. Principes S .O.L.I.D :**

- ❖ **SRP :** Le principe SRP est respecté: chaque objet a une unique responsabilité
  - XMLoader : chargement d'une liste:
  - FileImport : Import des listes
  - XMLSave : sauvegarde des listes
  - LecteurMedia : lecture des médias
  - EditeurList : créer des listes ...
- ❖ **OCP :** Le principe est respecté, l'architecture de l'application est ouverte à l'extension et fermée à la modification. Il est garantit par le patron composite qui

permet d'ajouter d'autre type de média et par le patron façade et passerelle qui permettent d'ajouter d'autre type de lecteur et éditeur

- ❖ **LSP** : le principe LSP il doit être respecté pendant le développement de l'application
- ❖ **ISP & DIP**: Ces deux principes sont respectés globalement. J'ai essayé de faire en sorte que chaque interface possède un rôle dans l'architecture de l'application et elle ne dépend pas des classes de bas niveau