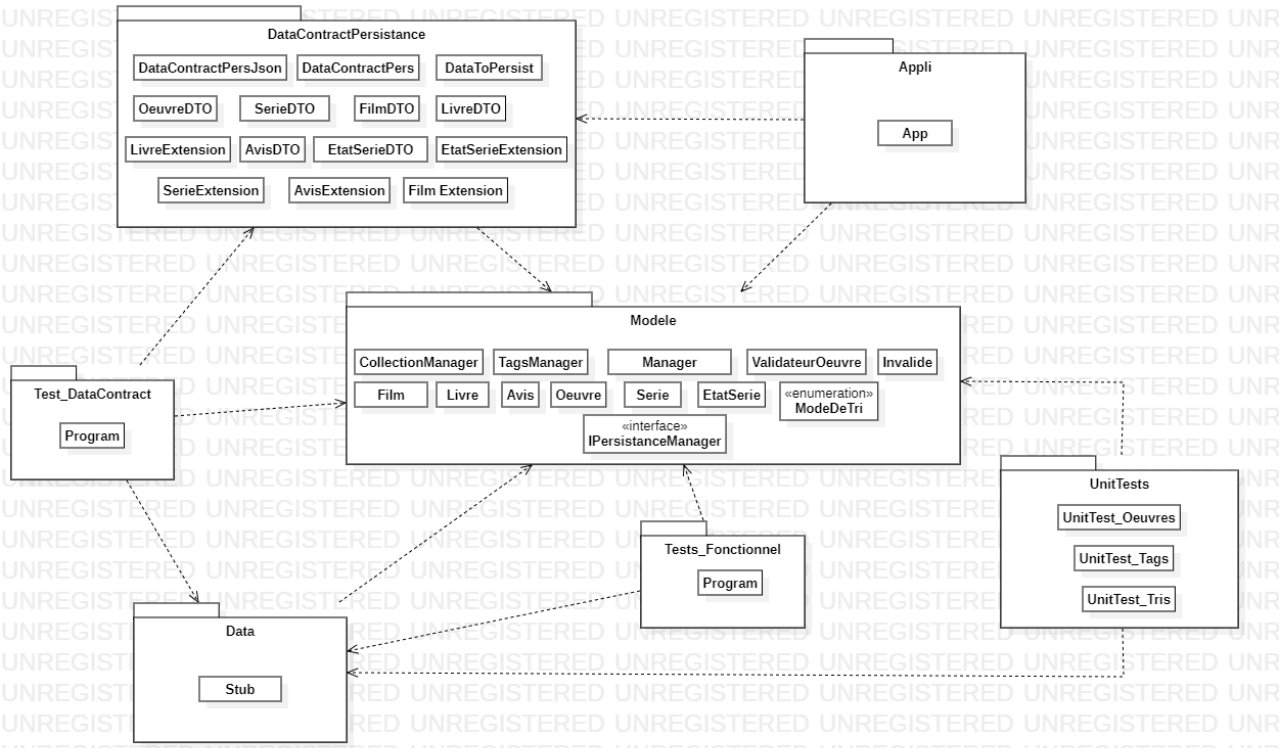
Documentation Conception et Programmation Orientées Objets (C#, .NET)

Diagramme de paquetage



L'intégralité des paquetages dépendent du *Modèle* qui n'a aucune dépendance.

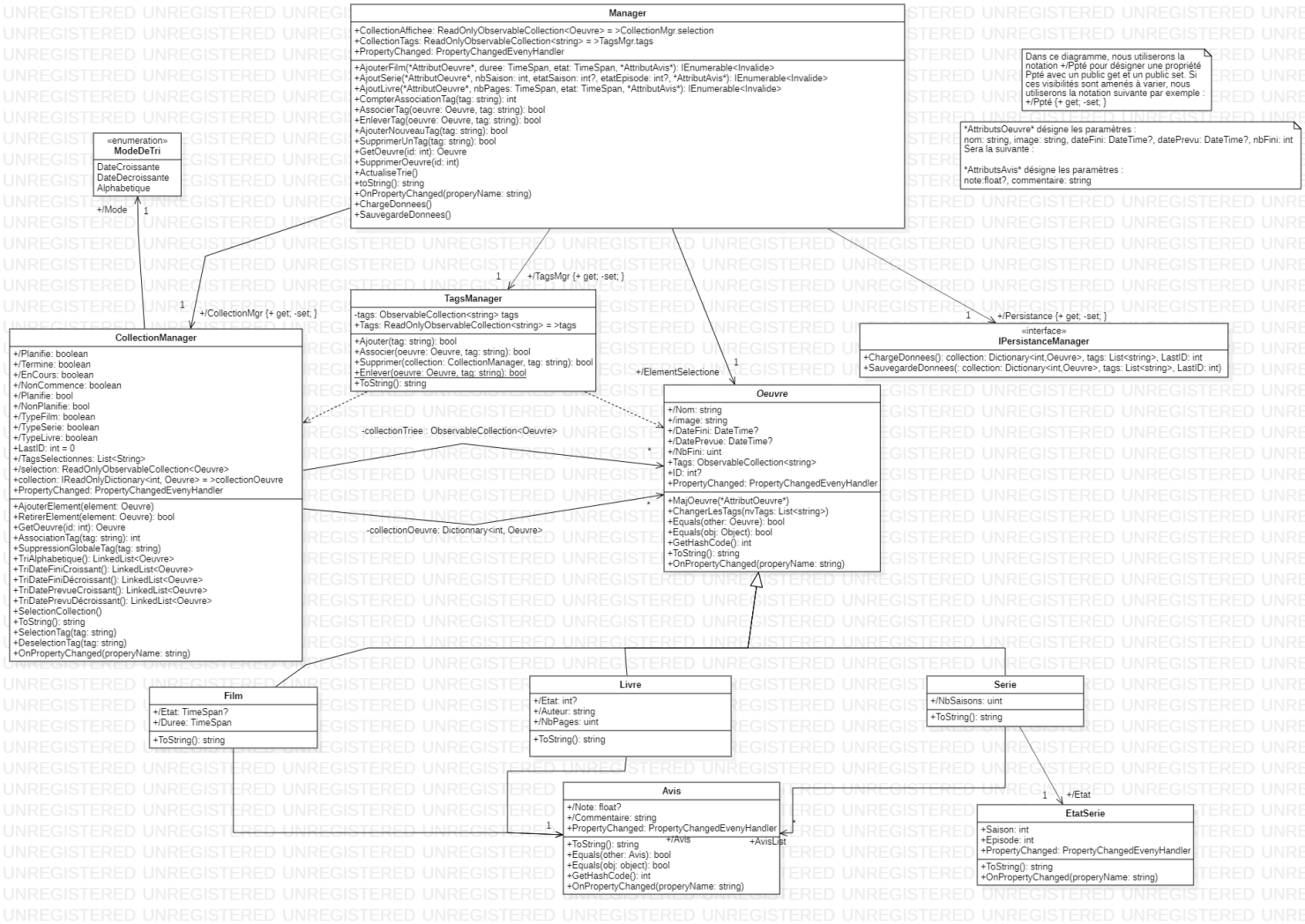
Les paquetages *Test_Fonctionnel* et *UnitTests* dépendent du modèle car ils testent les différentes classes du modèle. Ils dépendent aussi de *Data* car ils ont besoin de données.

Ensuite le *DataContractPersistance* dépend uniquement du *Modèle* car il utilise les classes pour prévoir leur sérialisation.

Test_DataContract en plus de sa dépendance au Modèle, utilise le paquetage *Data* et donc sa classe Stub pour tester la persistance des données. *Data* dépend donc lui aussi du modèle. Il a aussi une dépendance envers *DataContractPersistance* puisqu'il test la sauvegarde et le chargement des données.

Le paquetage *Appli* gère l'interface graphique est doit donc instancier un Manager pour pouvoir afficher des Œuvres, la collections, les tags… Pour permettre cela il dépend également de *Modèle*. Il a aussi besoin de *DataContractPersistance* pour permettre la sauvegarde et le chargement de données.

Diagramme de classes



L'objectif de l'application est de répertorier des Film, des Livres et des Séries, pour cela nous créons 3 classes. Ces classes héritent d'une classe *Œuvre* car elles ont beaucoup d'attributs en commun.

La classe *Œuvre* comporte entre autres un nom, une image, des informations sur la progression de l'utilisateur dans l'œuvre (dates, nombre de fois terminée...) et aussi une liste de tags, chaque tag peut être associé à plusieurs Œuvres.

Les classes filles de la classe *Œuvre* sont composées d'un état et d'une longueur adaptée à chaque classe, par exemple pour Livre il y a la page d'arrêt et le nombre de pages. L'état doit être « nullable » pour pouvoir indiquer qu'un élément n'est pas en cours de visionnage/lecture. (Pour la classe Série nous créons une classe *EtatSerie* pour pouvoir indiquer la saison et l'épisode d'arrêt.)

Les classe *Film*, *Livre* et *Serie* ont aussi des avis grâce à la classe *Avis* qui comporte une note et un commentaire. Pour la classe *Serie,* il y a un avis pour chaque saison donc les avis sont dans une liste d'avis (List<Avis>) et classés dans l'ordre des saisons.

Une classe statique *ValidateurŒuvre* permet de vérifier la validité d'une œuvre en vérifiant la cohérence de ses attributs entre eux, par exemple la date planifiée pour voir un film doit être une date future…etc. Chacune des méthodes qui vérifient la validité retourne un IEnumerable d'enum *Invalide* qui représente la ou les causes de l'invalidité.

Ensuite, il faut stocker ces œuvres dans une collection propre à l'utilisateur, pour cela on utilise la classe *CollectionManager*. Elle doit aussi permettre le tri de la collection et a donc plusieurs attributs de tri : « Planifie », « Termine », « EnCours », « tagsSelectionnes »… pour filtrer la collection et un attribut de type enum *ModeDeTri* représente le mode de tri des œuvres.

Un dictionnaire d'œuvre (Dictionary<int, Œuvre>) contient l'ensemble des Œuvres de l'utilisateur. Lors de l'ajout d'une Œuvre on lui attribut un ID qui correspond à son ordre d'ajout dans la collection (utile pour trier), c'est pour ça qu'une œuvre à une propriété ID. Celle-ci est null si l'œuvre n'appartient pas à une collection. Une propriété LastID est incrémentée à chaque ajout pour permettre ce fonctionnement.

L'autre collection est une ObservableCollection pour *Œuvre* dans laquelle sont chargées les œuvres sélectionnées de façon triée pour l'affichage.

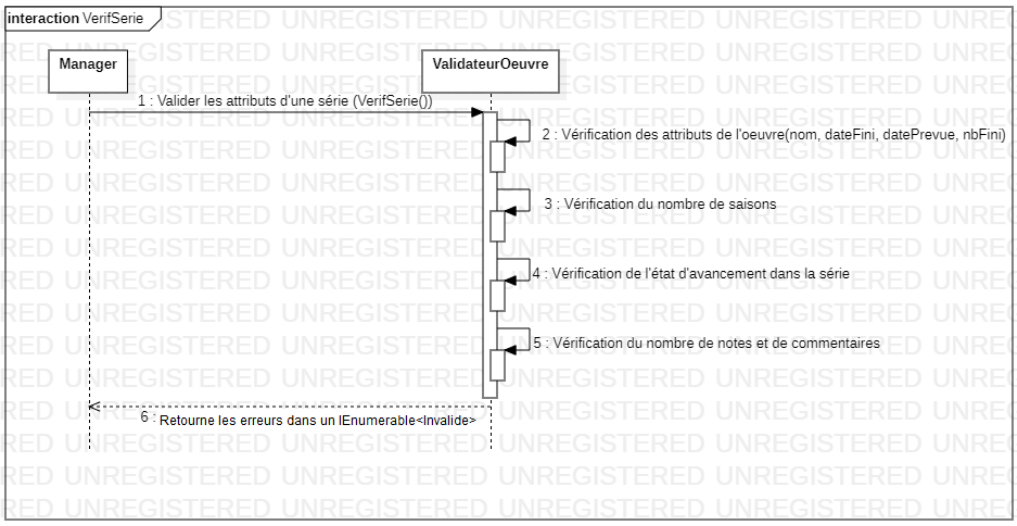
Les méthodes de *CollectionManager* servent aux différents tris, permettent de gérer l'ajout et de supprimer des œuvres ainsi qu'effectuer des opérations sur l'ensemble des œuvres d'une collection comme par exemple supprimer un tag de manière globale avec SuppressionGlobalTag ou compter son nombre d'association avec AssociationTag.

La classe *TagsManager* permet de gérer les tags disponibles (création et suppression) qui sont stockés dans une ObservableCollection de string. Elle permet aussi d'en associer ou d'en enlever à une œuvre. La méthode Enlever qui enlève un tag d'une œuvre est statique car elle ne dépend pas d'une collection de tag, elle peut donc être utilisée par CollectionManager pour supprimer globalement un tag.

Enfin la classe Manager correspond à la façade du Modèle, elle utilise toutes les autres classes, elle possède une instance de *CollectionManager* et de *TagsManager* pour gérer simultanément l'ensemble des œuvres avec les tags. Elle possède aussi un attribut de type *Oeuvre* ElementSelectionne qui permet la sélection d'une œuvre par l'utilisateur. Des propriétés calculées permettent de récupérer les collections de tags et d'œuvres destinées à l'affichage. Les méthodes reprennent pour la plupart des méthodes de *CollectionManager* et de *TagsManager* pour simplifier l'utilisation du Modèle. Il y a des méthodes pour ajouter des œuvres qui ont comme paramètres les attributs des classes impliquées (utile pour le Stub et pour vérifier la validité sans avoir à instancier un Œuvre). Il y a aussi une méthode pour ajouter une œuvre déjà instanciée qui elle n'utilise pas le validateur d'œuvre.

Enfin il faut mentionner que *Manager* possède un attribut *IPersistanceManager* qui permet d'utiliser des méthodes de chargement et de sauvegarde de données pour la persistance.

Diagramme de séquence : ValidateurOeuvre.VerifSerie(..)



Ici on parle de la Méthode VerifSerie(..) qui prend en paramètre le nom, la date à laquelle l'élément à été vu la dernière fois, la date prévue, le nombre de fois qu'il a été fini, le nombre saisons, la saison d'arrêt et l'épisode d'arrêt, un tableau de notes et un tableau de commentaires.

On récupère la Liste d'enum Invalide de la méthode VerifOeuvre après avoir passé les paramètres propres à Oeuvre puis on vérifie les attributs propres à une série :

Le nombre de saisons doit être strictement supérieur à zéro.

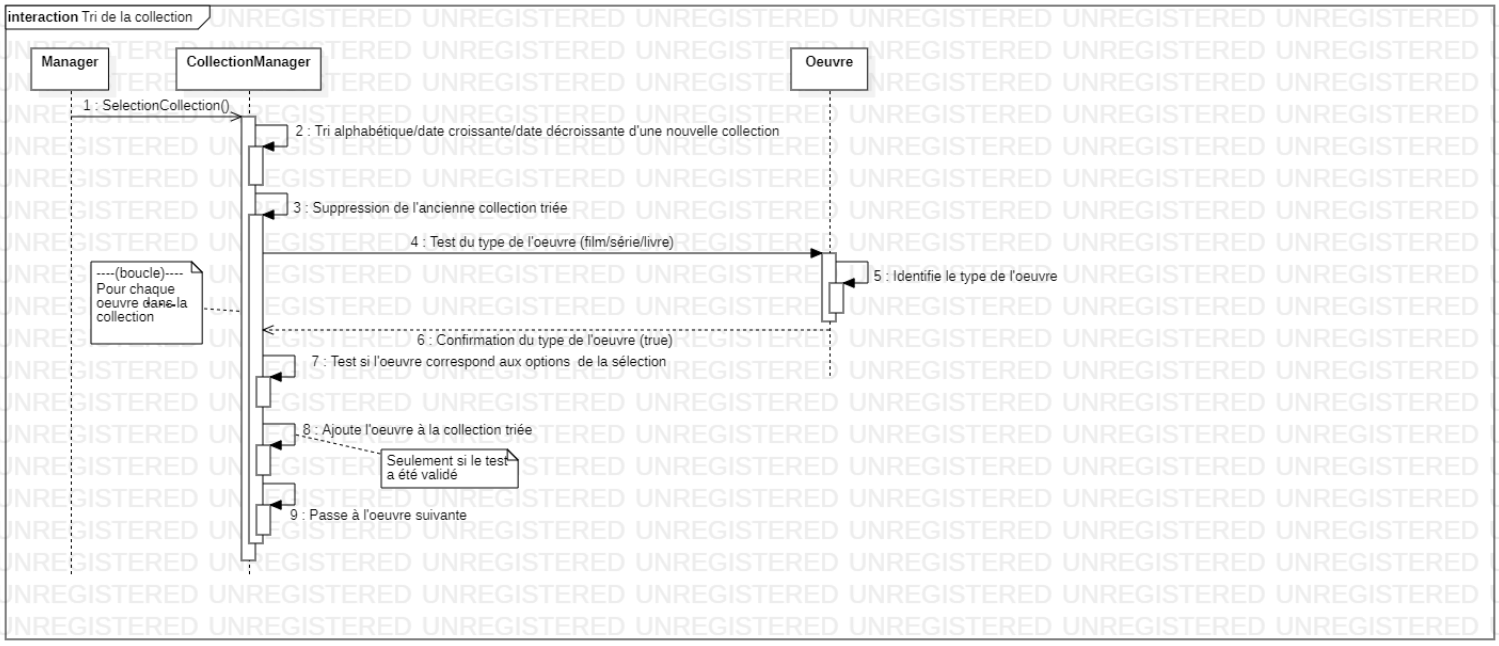
L'état de l'avancement (si n'est pas null) doit être composé d'une saison située entre 1 et le nombre de saisons et d'un épisode strictement supérieur à 0.

Le nombre de notes et de commentaires doit être le même, c'est-à-dire qu'il doit s'agir de tableaux de mêmes tailles. Et chaque note doit être en 0 et 10.

A chaque test s'il y a erreur on ajoute un enum Invalide approprié dans l'IEnumerable.

Enfin la méthode retourne les erreurs à la méthode appelante.

Diagramme de séquence : CollectionManager.SelectionOeuvre()



Cette méthode est appelée à chaque fois que la collection a besoin d'être triée puis affichée.

D'abord on regarde la propriété Mode pour connaître le ModeDeTrie (alphabétique, date croissante…etc.). On appelle ensuite la méthode de tri qui correspond.

Cette méthode retourne une LinkedList d'œuvres car c'est ce qu'elle utilise pour trier la collection.

L'ancienne collection triée est vidée de ses œuvres puis pour chaque élément de la LinkedList d'œuvres on effectue différents tests en prenant en compte les options de tri (Planifié, Terminé…etc.). Il faut cependant caster l'œuvre dans sa classe réelle (Film/Série/Livre) pour pouvoir faire l'intégralité des tests.

On regarde également si les tags correspondent aux tags sélectionnés.

Si l'œuvre correspond aux critères elle est ajoutée à l'ObservableCollection triée qui sera affichée.

Quand on arrive à la fin de la LinkedList le tri est effectué et la collection triée peut être affichée.

Description de l'architecture

Le modèle de notre application utilise le patron de conception « Façade ». C'est la classe *Manager* qui joue le rôle d'interface pour tout le système du modèle.

Manager dépend de plusieurs classes pour remplir son rôle : elle dépend de *CollectionManager* classe qui gère la collection d'œuvres, de *TagsManager* qui gère les tags et *d'Oeuvre* pour pouvoir savoir quelle œuvre est sélectionnée.

TagsManager se charge simplement de stocker des « tags » en string, néanmoins elle possède des méthodes qui nécessitent des instances de classe ŒUvre pour associer et enlever des tags. Et une méthode nécessite une *CollectionManager* en tant que paramètre pour supprimer tous les tags dans une collection.

Pour gérer le mode de tri, la classe possède un attribut enum *ModeDeTri*.

CollectionManager dépend d'*Oeuvre* car elle doit stocker plusieurs instances de la classe ŒUvre et donc de classe *Film*, classe *Livre* et de classe *Serie* qui sont les classes filles de la classe mère ŒUvre.

Les méthodes pour manipuler la collection utilisent des ŒUvres pour les ajouts, les suppressions…etc.

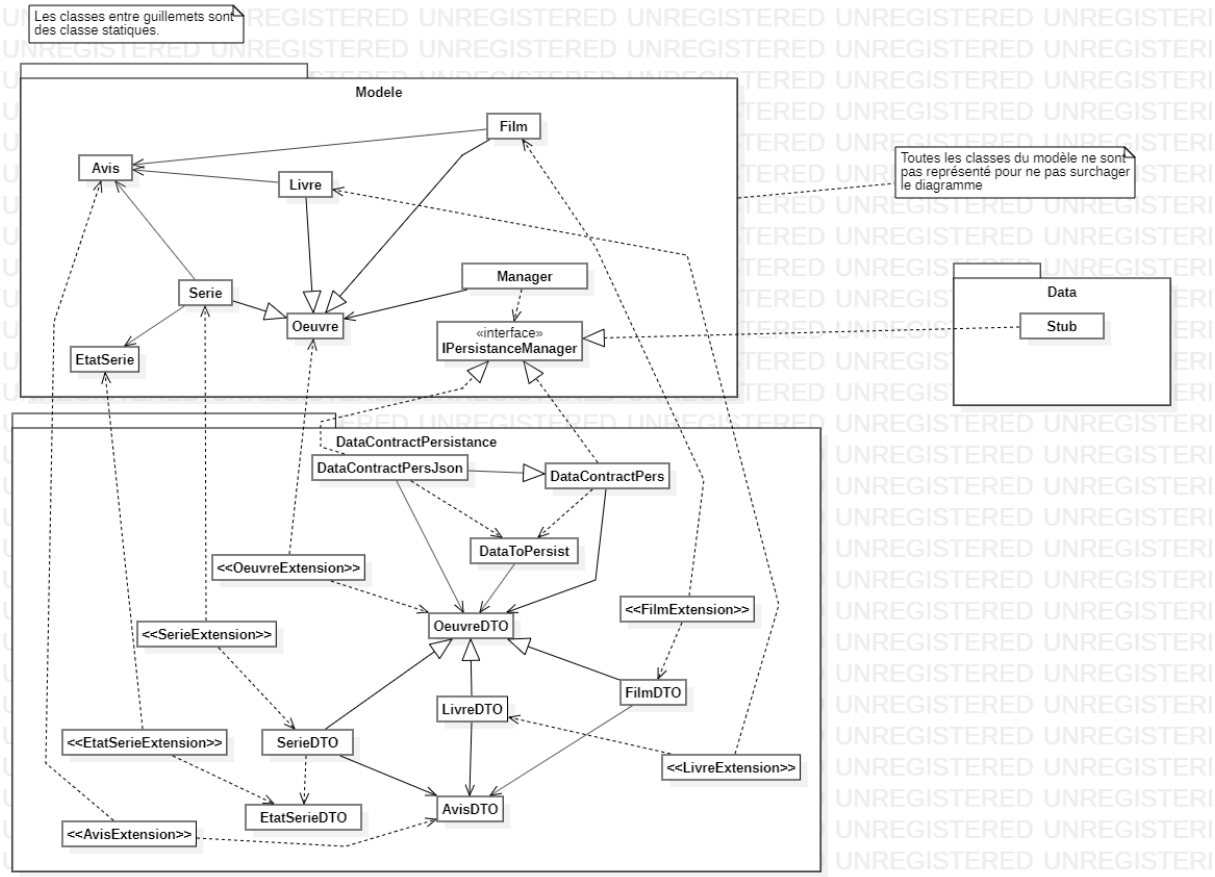
La classe *Film* dépend de la classe *Avis* car elle possède un avis, de même pour *Livre* et *Serie* (qui, elle peut en posséder plusieurs dans une liste). *Serie* dépend aussi de la classe *EtatSerie* qui est imbriquée dans la classe *Serie* car elle est utilisée exclusivement par *Serie*.

La classe *ValidateurOeuvre* est utile pour valider les ŒUvres grâce à des méthodes de vérifications qui retournent des IEnumerables de type enum *Invalide*. Ces IEnumerables sont récupérables par les utilisateurs du modèle pour connaître les causes d'invalidité. Même si elle est considérée comme un utilitaire *Manager* utilise cette classe.

Pour la persistance on utilise une injection de dépendance car *Manager* possède un attribut *IPersistanceManager,* il n'y a donc pas de dépendance directe entre la classe instanciée et *Manager*. Mais la classe instanciée dans Manager doit être une classe qui implémente *IPersistanceManager* et donc avoir les méthodes ChargesDonnées() et Sauvegarde() pour permettre la persistance des données.

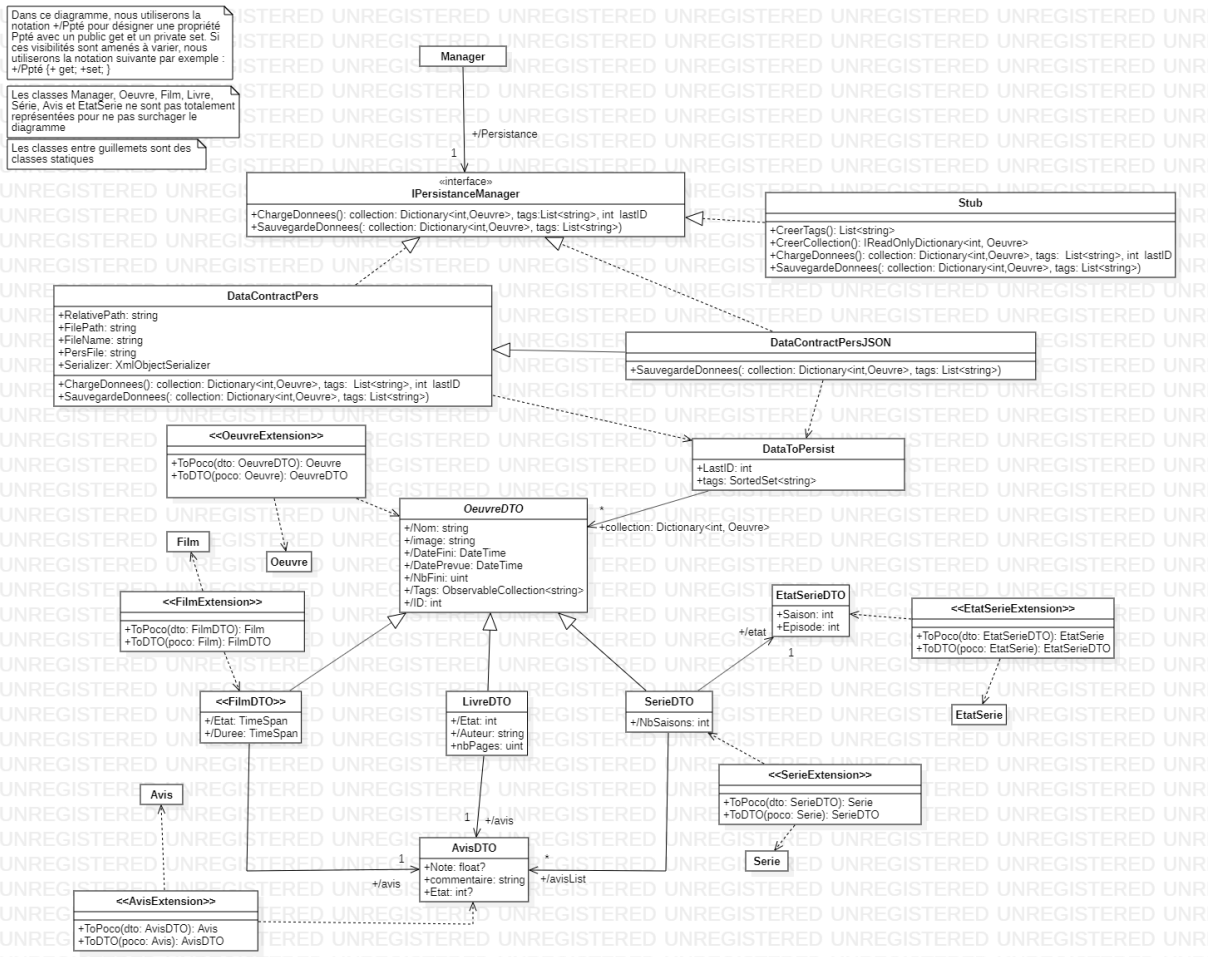
Documentation sur la persistance (Projet Tuteuré S2)

Diagramme de paquetage sur la persistance (détaillé)



Cf. Diagramme de paquetage.

Diagramme de classes sur la persistance



Manager possède un attribut persistance de type *IPersistanceManager* qui sera initialiser par son constructeur.

IPersistanceManager est une interface ayant une méthode de chargement et de sauvegarde de données permettant de rendre le modèle et l'application indépendants du choix de la stratégie de persistance.

3 classes implémentes cette interface :

-*DataContractPers* permet sauvegarder les données dans un fichier XML

- *DataContractPersJSON* permet sauvegarder les données dans un fichier XML et hérite de la fonction de chargement de *DataContractPers.*

-*Stub* permet de charger des données préexistantes et sa méthode de sauvegarde est inutile.

DataToPersist possède en attribut les types de données à persister. Les classes DataContract en ont donc besoin pour connaitre le type de données à persister

Les classes *DTO* rendent le code du modèle plus clair car elles permettent de retirer tous les dataContract et dataMember. Elles possèdent seulement les attributs à persister de leur classe miroir du modèle.

Les classes d'extensions sont des classes statiques permettant de convertir des classes *DTO* en classe *POCO* (leur classe miroir du modèle) et inversement. Elles dépendent donc des classes *DTO* et POCO qui leur sont associées.