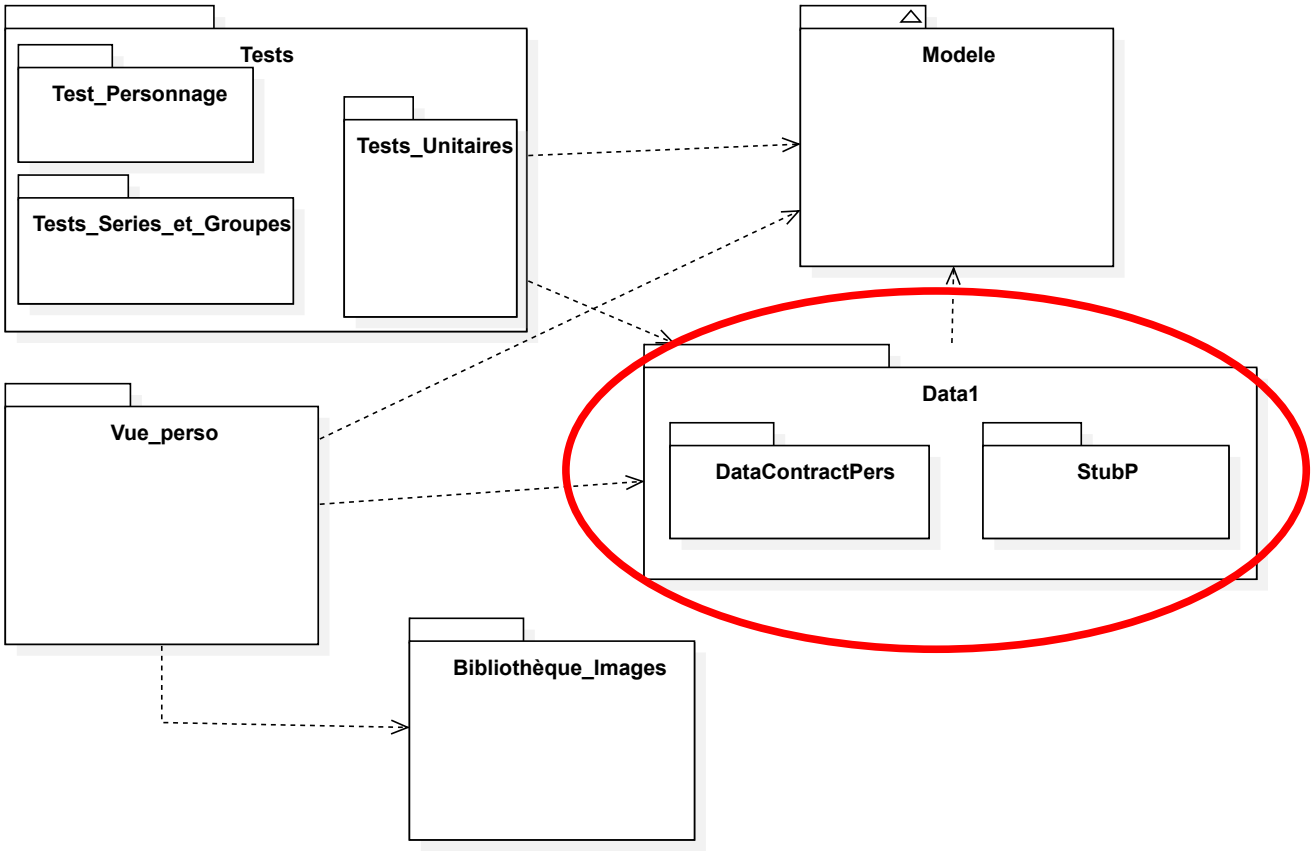
Explication Persistance et Ajout Personnel de notre application

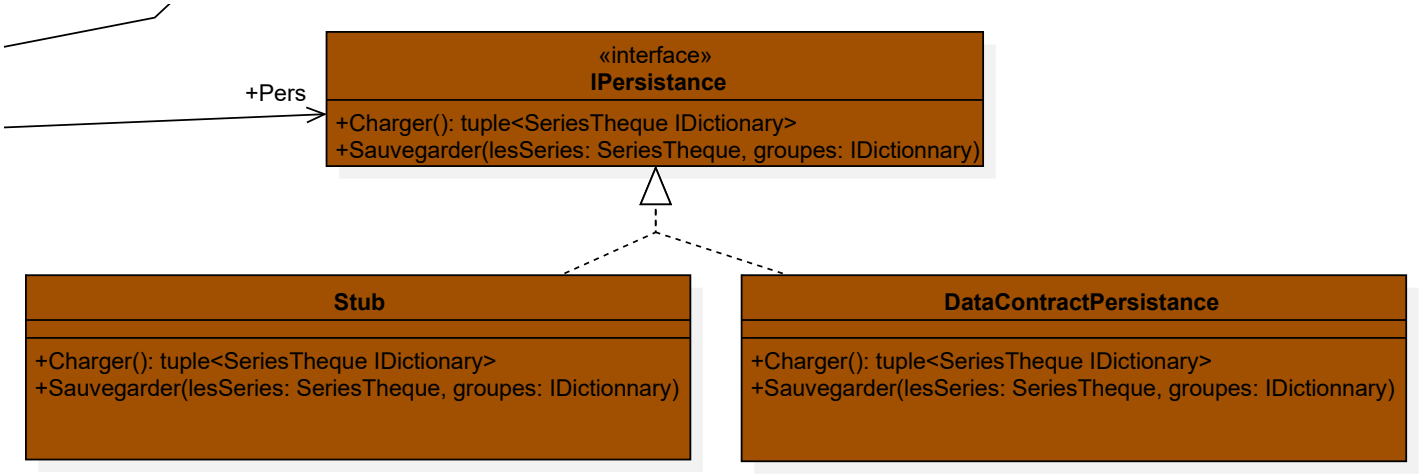
La Persistance :

1.Diagramme de paquetage :



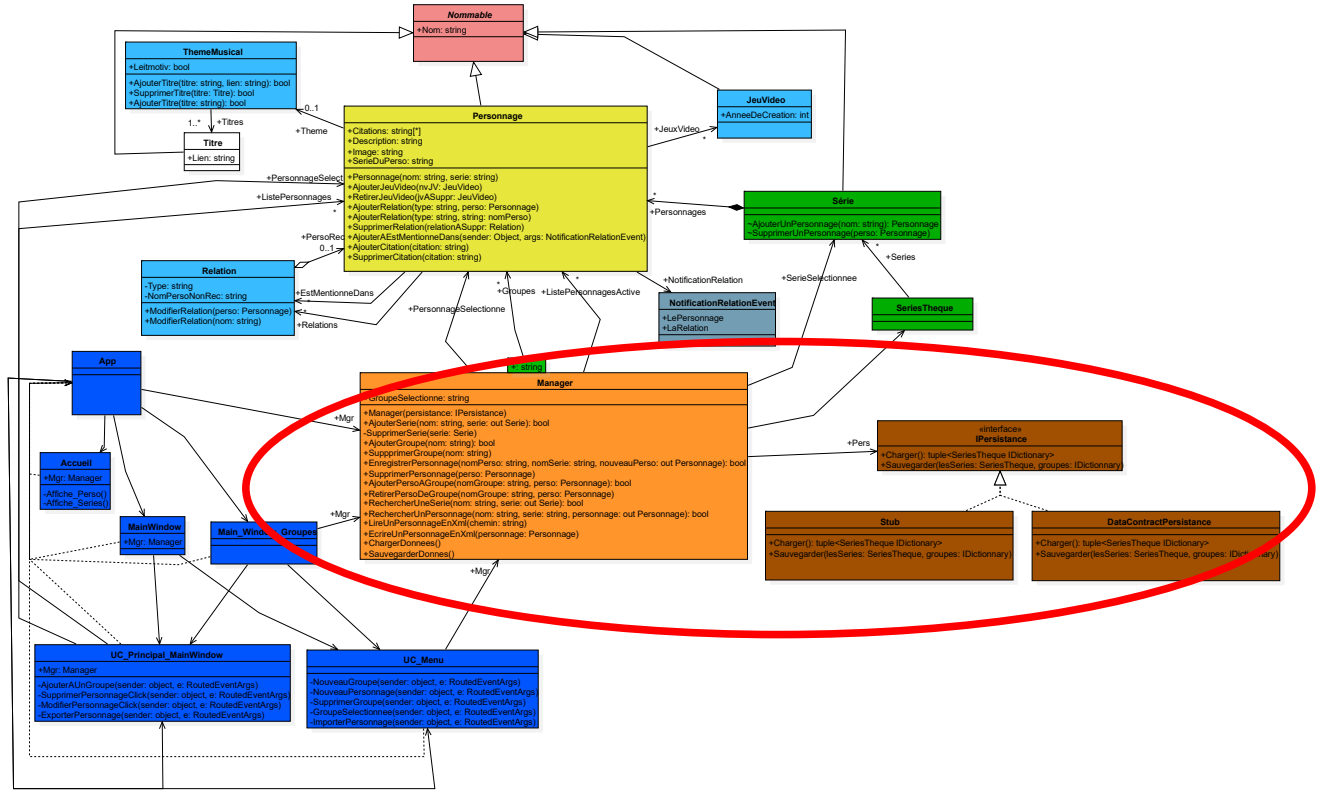
Nous souhaitons mettre en évidence ici la partie Persistance de notre application.
La séparation des méthodes de persistance en assemblages les rend interchangeables, c'est-à-dire que le modèle peut utiliser aussi bien le `Stub` que la `DataContractPers`, il suffit que le modèle qui contient une couche abstraite de persistance, sache avec quelle couche contrainte, il doit communiquer. Cela nous offre aussi la possibilité de rajouter une autre «méthode» de persistance à tout moment sans avoir à modifier le modèle.

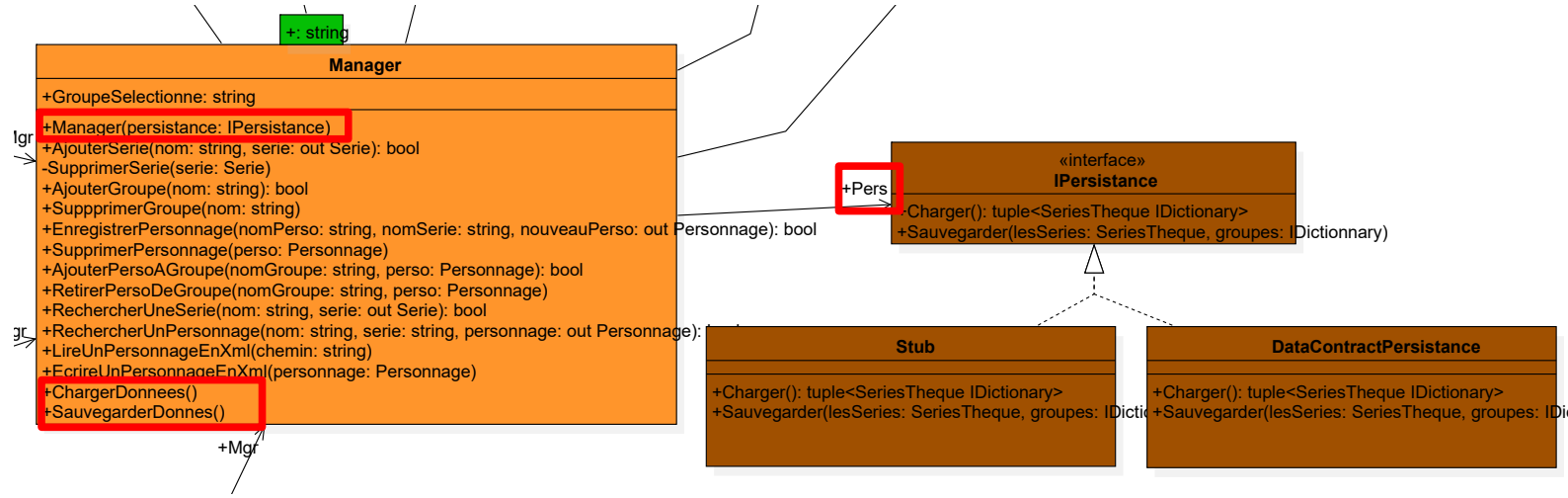
2.Les classes responsables de la Persistance :



Voici les classes responsables de la persistance dans notre modèle. L'interface `Ipersistance` représente la couche abstraite de la persistance. La couche concrète est représentée par les classes `Stub` et `DataContractPersistance` (il est possible d'en rajouter si l'on veut utiliser une autre «méthode» pour sauvegarder et restaurer les données). Le patron de conception utilisé ici est la stratégie, l'interface propose le prototype de l'algorithme, les classes filles ont la même fonction mais définissent l'algorithme de l'interface différemment. Dans notre cas, ces classe filles représentent les différentes stratégies de persistance. On dit plus généralement que seul le comportement diffère entre les classes apparentées.
Une stratégie doit avoir un contexte lors de son utilisation, il va définir quel type de persistance il va utiliser, dans notre modèle le contexte sera le `Manager`.

3.Diagramme de Classes:





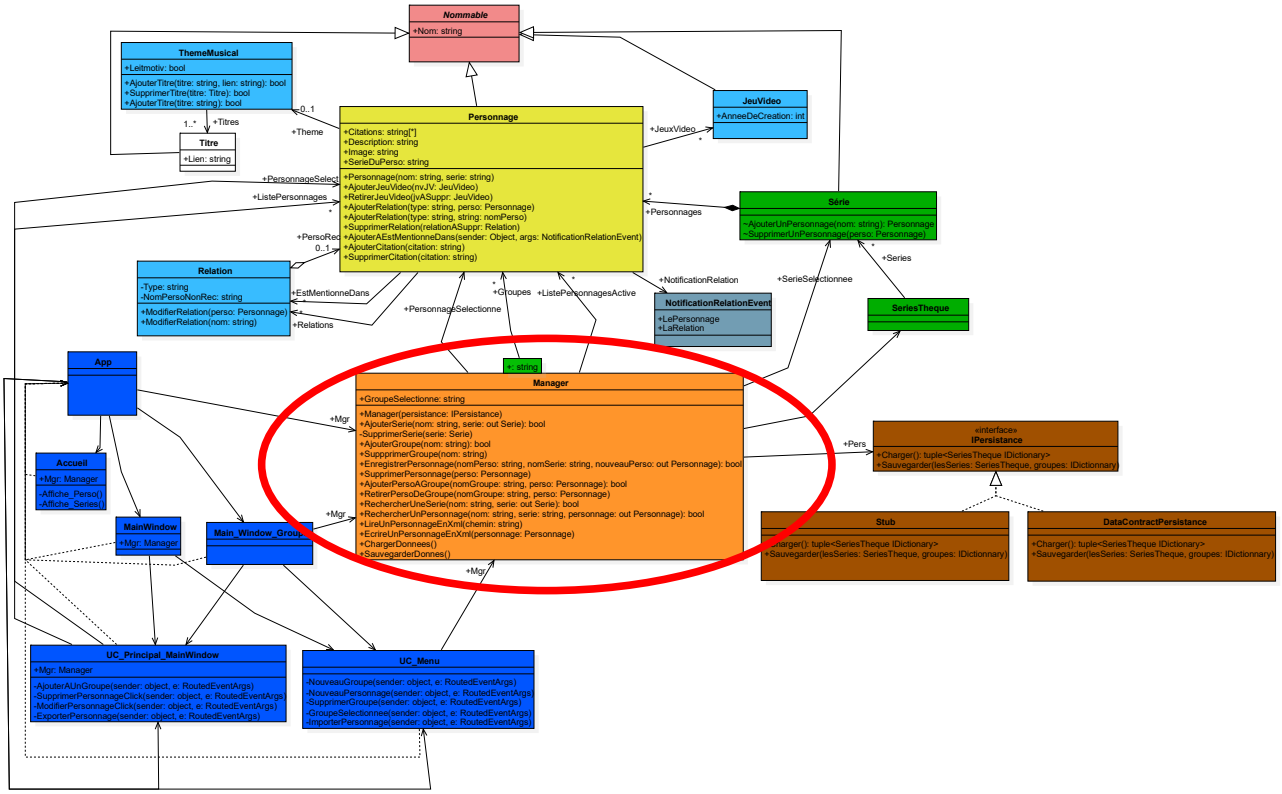
Nous pouvons voir ici la mise en place de la persistance dans le `Manager` qui possède une propriété de type `IPersistance`. Elle est setée dans le constructeur de `Manager` où on lui passe une instance d'une des classes fille de `IPersistance`. On réalise donc une injection de dépendance concrète via le constructeur de `Manager`.

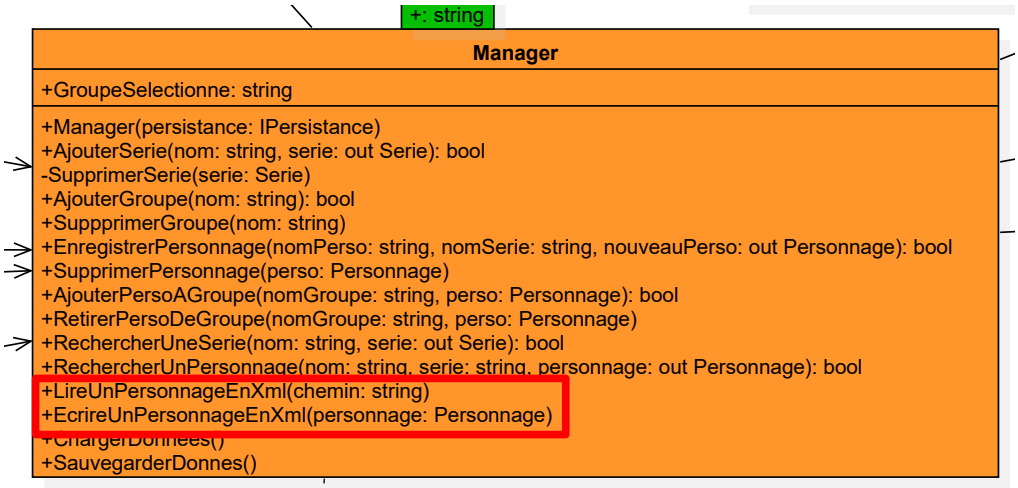
Le `Manager` appelle dans ses méthodes *void ChargerDonnees()* et *void SauvegarderDonnees()* les méthodes *Charger(): tuple<SeriesTheque IDictionary>* et *Sauvegarder(in lesSeries:SeriesTheque, in groupes:IDictionnary)* . Grâce à ces deux méthodes on sauvegarde et on charge les données dont le manager a besoin.

Les Ajouts Personnels :

1.Import/Export :

1.1Diagramme de Classe:





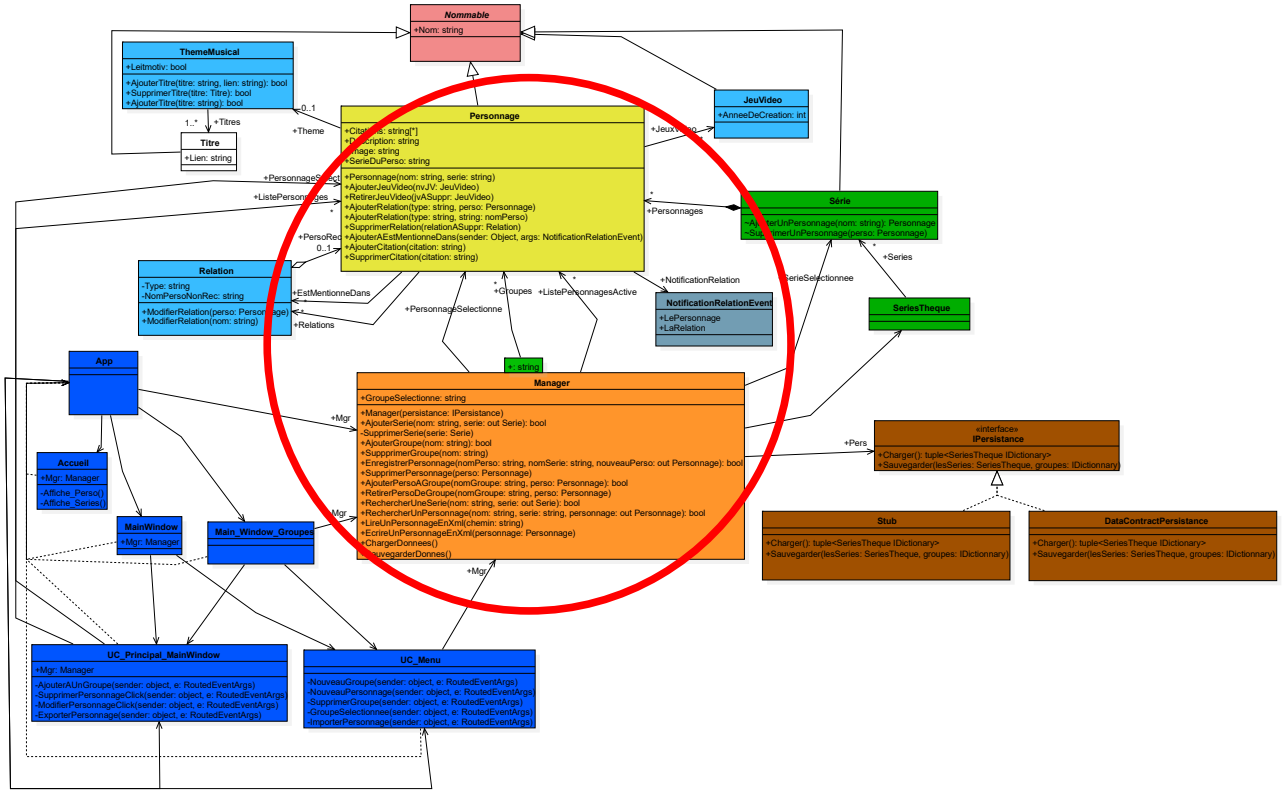
Notre ajout personnel se situe dans la classe `Manager`. Il consiste à donner à l'utilisateur la possibilité d'importer et d'exporter des personnages depuis ou dans des fichiers xml. Grâce à la méthode *void EcrireUnPersonnageEnXml(in personnage:Personnage)* nous pouvons sérialiser un personnage dans un fichier xml. Grâce à la méthode void *LireUnPersonnageEnXml(in chemin:string)* nous pouvons ajouter un personnage à partir d'un fichier xml.

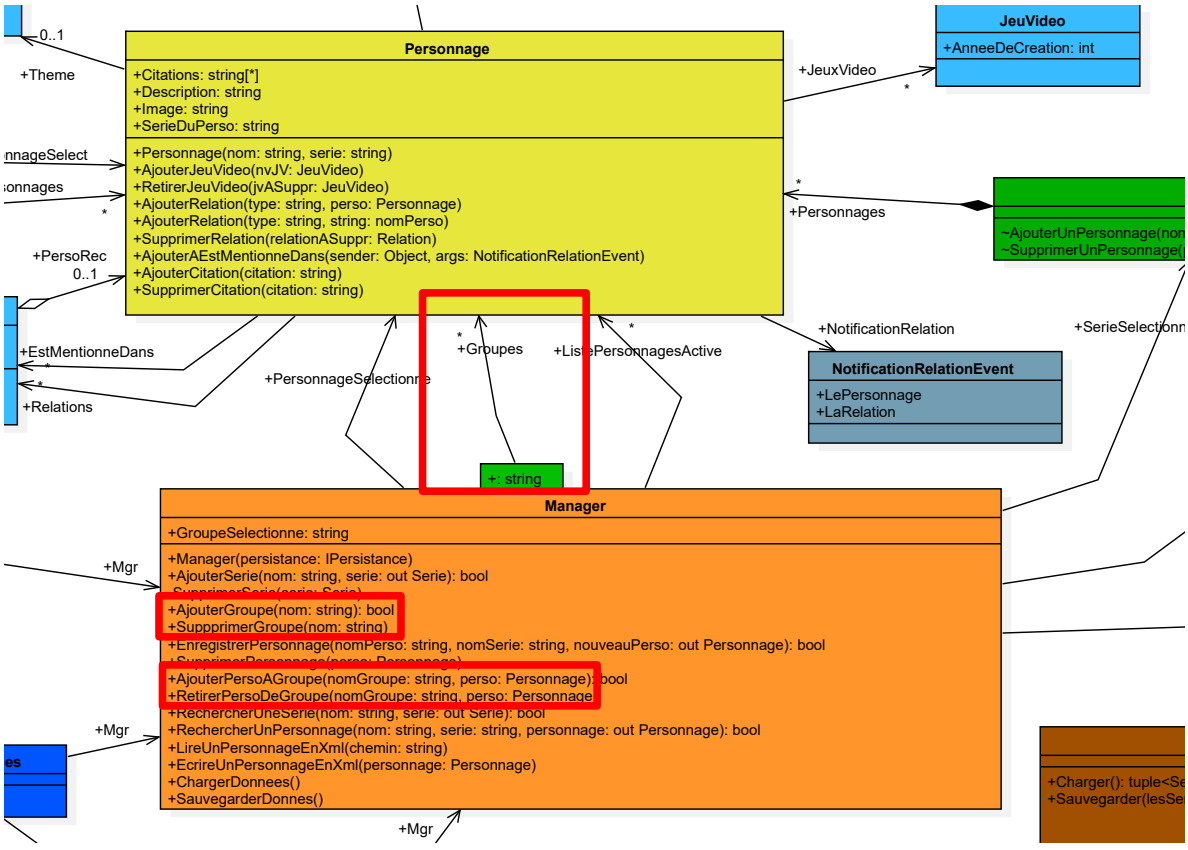
Ces méthodes sont appelées dans la vue lors du clic sur le menuItem de `UC_Menu` « Importer un personnage » (pour l'importation) s'en suit la sélection du fichier à importer, et lors du clic sur le menuItem de `UC_Principal_MainWindow` « Exporter le personnage » (pour l'exportation) s'en suit la sélection du dossier où exporter le personnage.

Nous avons choisi d'exporter les données dans un fichier xml car il est compréhensible par un utilisateur s'il est ouvert avec un éditeur de texte, grâce aux balises qui le composent.

2.Groupe :

2.1Diagramme de classe :





L’utilisateur peut créer des groupes dans lesquels il ajoute les personnages qu’il souhaite. Pour gérer cette fonctionnalité notre classe `Manager` possède un dictionnaire nommé Groupes qui a pour clé un string et pour valeur une liste de personnages. Lors de la création d’un groupe la méthode *AjouterGroupe(in nom:string): bool* de `Manager` est appelé le groupe n’est ajouté que s’il n’existe pas. Pour ajouter et supprimer un `Personnage` on utilise les méthodes *AjouterPersoAGroupe(in nomGroupe:string, in perso:Personnage): bool* et *RetirerPersoDeGroupe(in nomGroupe:string, in perso:Personnage)*. Pour supprimer un groupe nous avons la méthode *SuppprimerGroupe(in nom:string).*

Notre choix d’utiliser un dictionnaire a été motivé, par le fait, que l’on voulait associer un nom de groupe (un string) à une liste de personnages, sans qu’il y ait de doublon au niveau des noms de groupes. Le dictionnaire nous paraît être la collection le plus adaptée, en raison du fait qu’elle contient des paires clef-valeur. Dans notre cas la clef est le nom du groupe et la valeur est la liste de personnage associée.