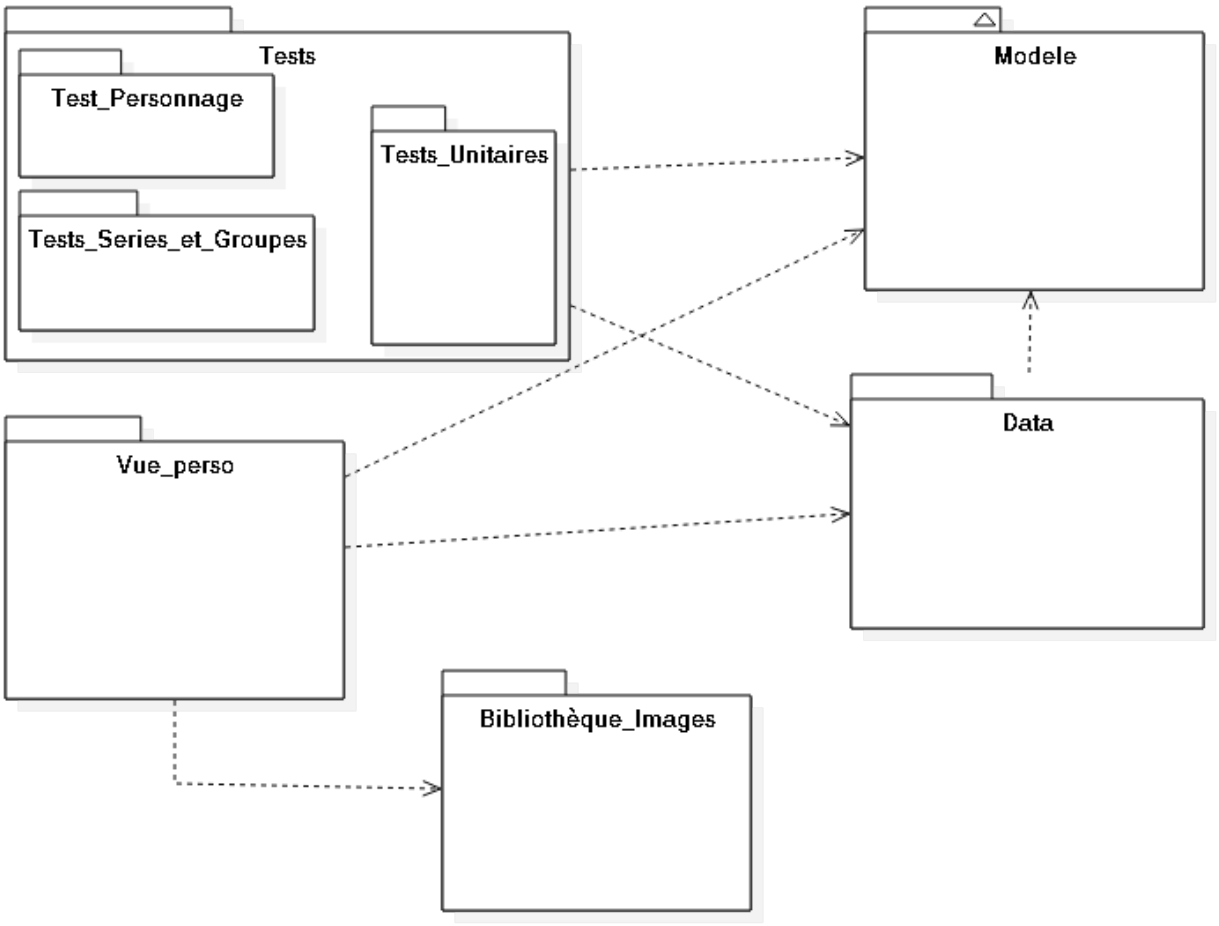
Explication architecture de l’application

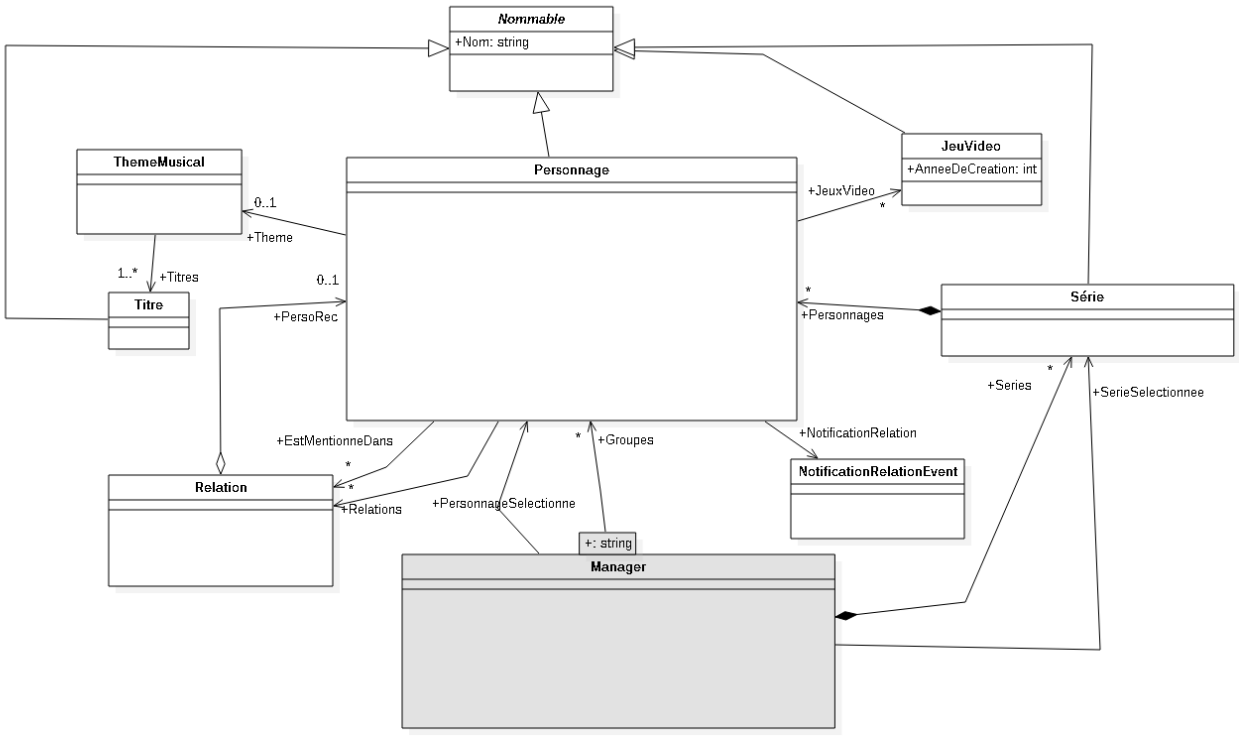
Diagramme de paquetage :



Voici le diagramme de paquetage de notre application. Il est composé de plusieurs projets une bibliothèque d’image qui va contenir les images de l’application, un projet Vue_perso qui contient la partie graphique de notre application, le projet Modele qui contient le cœur de l’application, le projet Data qui contient les classes nécessaires à la mise en place de la persistance (avec notamment le stub qui contient les données utilisées dans les tests), le dossier Tests contient nos 3 projets qui testent les fonctionnalités. Le projet Test_Personnage permet de tester les fonctionnalités qui impactent le personnage. Le projet Tests_Series_et_Groupes permet lui de tester les fonctionnalités sur les séries et les groupes qui contiennent des personnages. Les tests unitaires permettent de tester plus en détail des méthodes essentielles à la mise en place de certaines fonctionnalités.

Diagrammes de Classe :

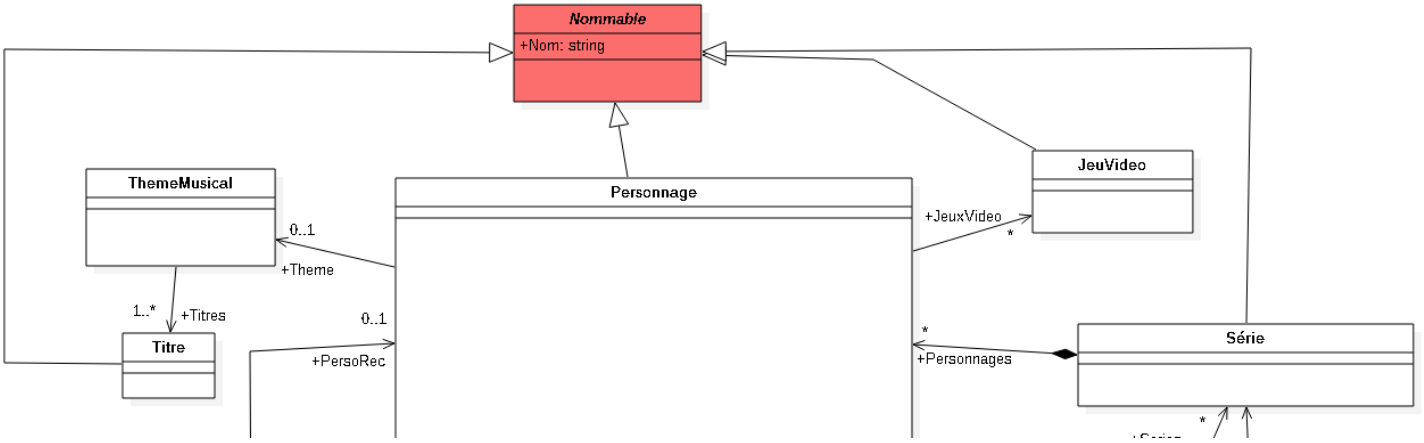
1.Diagramme général non détaillé



Voici le modèle détaillé de notre application.
La classe Manager est la classe Principale, c'est elle qui va gérer les deux autres classes qui ont un rôle très important la classe Serie et la classe Personnage. Elle va donc nous permettre de naviguer dans l'application entre les séries de l'application qui regroupent les personnages, et une fois à l'intérieur d'une série de consulter les éléments qui composent un personnage. Le recherche d'une série lui sera elle aussi intégrée. Cette classe comporte aussi un attribut groupe, un dictionnaire qui permet à l'utilisateur de former des groupes de ses personnages favoris.
Le patron de conception utiliser dans notre modèle est donc la façade.

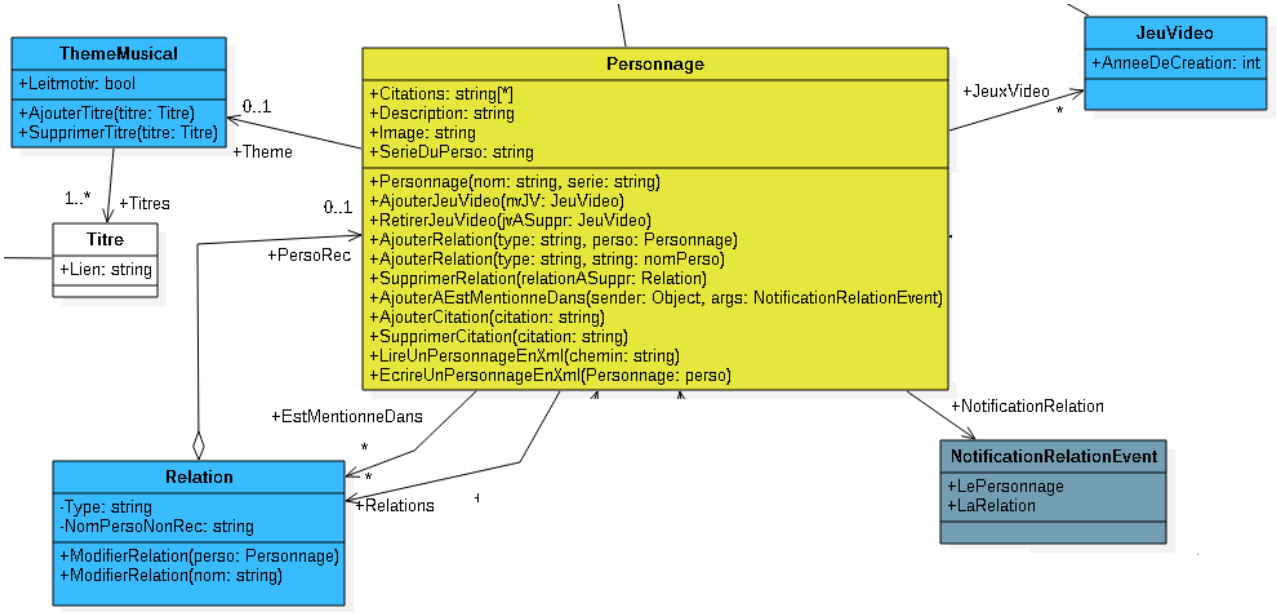
La façade dans notre cas le manager est un patron de conception qui permet de gérer avec une interface simple un sous-système plus complexe. Ce dernier est donc bien plus simple à utiliser, la façade et la classe qui permet de gérer le reste du code. Elle fait aussi le lien entre différentes classes dans notre cas avec série ou avec personnage par l'intermédiaire du dictionnaire Groupes qui rassemblent les personnages que l'utilisateur décide de ranger dans un groupe qu'il crée au préalable. La façade délègue des requêtes à réaliser aux autres classes. Le manager donne des travaux à réaliser aux autres classes en utilisant leurs différentes méthodes, les classes ne font pas référence au manager, c'est ce dernier qui les utilise en leur faisant référence.

2. Zoom sur la classe abstraite Nommable et ces héritiers



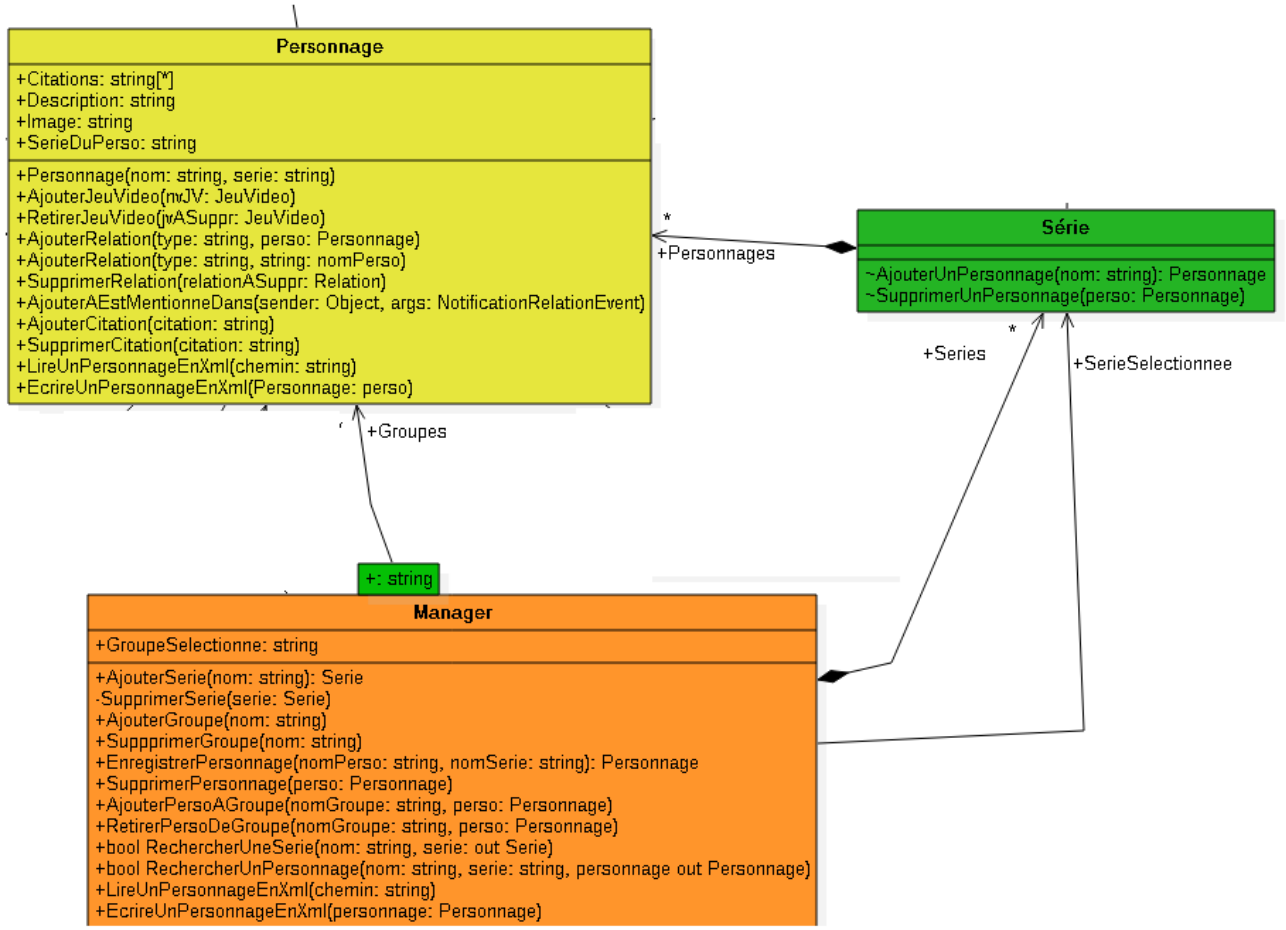
L'utilisation que l'on fait de la classe nommable est un factorisation de code. C'est à dire que la classe nommable fournit l'attribut nom de type string aux autres classes elle redéfinit également le protocole d'égalité qui va être très utile par ces autres classes qui héritent de Nommable. Elles vont cependant avoir un fonctionnement très différent les unes des autres mais leur algorithme concernant le protocole d'égalité est le même (à l'exception de la classe personnage qui aura un complément sur le protocole d'égalité son attribut SerieDuPerso de type string sera lui aussi pris en compte).

3.Zoom sur la classe Personnage



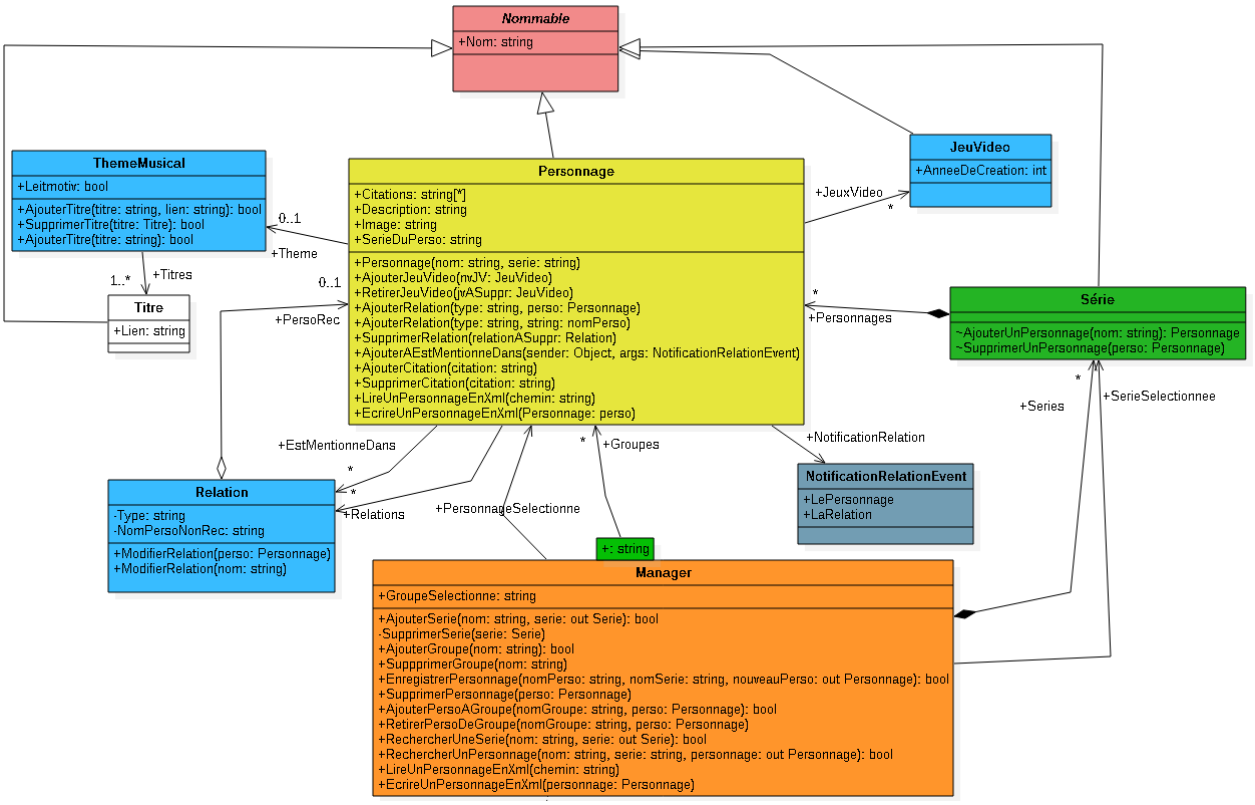
La classe personnage est le cœur de notre application c'est elle qui renferme la partie detail. Un personnage est donc composé de plusieurs éléments qui seront affiché dans la partie detail de l'application. Certains de ces éléments sont des objets que nous avons créés nous même(Relation, JeuVideo, ThemeMusical). Ces classes sont représentés en bleu sur le diagramme de classe.
Nous avons aussi créer un événement en respectant le patron standard des événements qui se nomme NotificationRelationEvent. Le patron qui se cache derrière cet événement est le patron observateur. Il permet de notifier un changement. Dans notre cas l'événement NotificationRelationEvent permet lors de l'ajout d'une relation avec un Personnage enregistré de notifier le personnage de la relation qu'il a été ajouté à une relation. Nous lui avons abonné une méthode qui ajoute donc à ce personnage la nouvelle relation dans l'attribut EstMentionneDans qui est de type Relation.

4.Zoom sur les collections contenues dans le manager



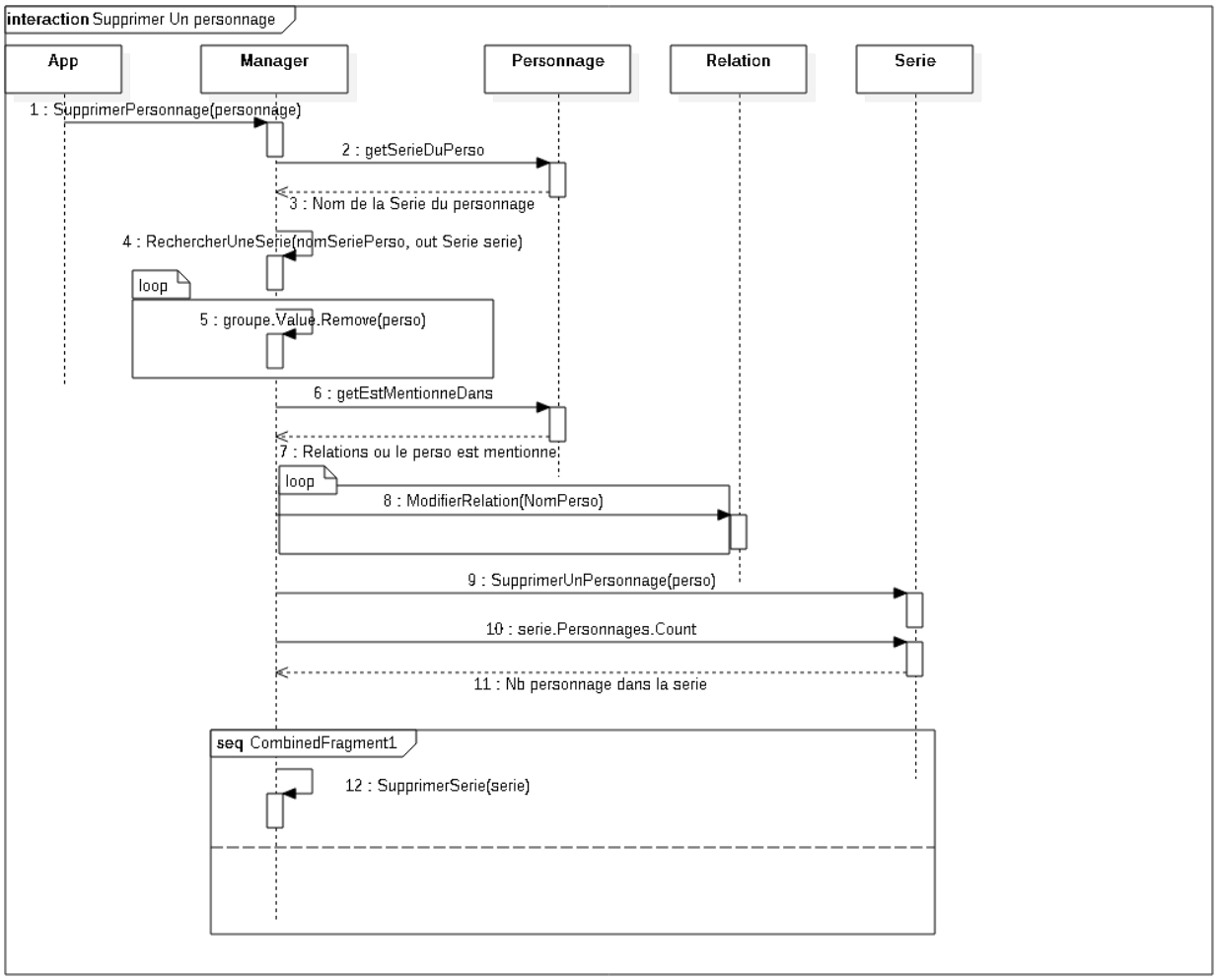
Le Manager comme nous l'avons déjà dit précédemment est la classe principale de l'application. Elle contient plusieurs propriétés qui sont utiles lors de son utilisation avec la vue. Mais les propriétés aux quelles nous nous intéressons ici sont celles du type des classes représentées en vertes. Nous avons un dictionnaire appelés Groupes qui contient les listes de personnages créées par l'utilisateur (par exemple une liste de favoris nommées « Favoris »). Et une collection de Séries de type Serie. Cette collection stocke les personnages dans une Serie (un personnage pouvant appartenir seulement à une série).Ces deux collection seront donc essentielles lors de la mise en place de la persistance se seront elles qu'il faudra sauvegarder.

5.Diagramme de classe complet



Voici notre diagramme de classe détaillé.Pour résumé nous y retrouvons le patron de conception de la façade avec notre classe Manager, qui est la classe principale de l'application. Les deux conteneurs qu'elle contient, qui vont aussi servir à la persistance. La classe Nommable qui permet une factorisation du code notamment pour les protocoles d'égalité. La classe Personnage avec ces propriétés, elle est le cœur de l'application c'est elle qui sera en charge du detail. L'événement NotificationRelationEvent derrière lequel se cache le patron observateur.

Diagramme de Séquence :



Ce diagramme de séquence explique l'algorithme qui est mis en œuvre lors de la suppression d'un personnage. Tout d'abord l'application (App) qui a été sollicitée par l'utilisateur appelle la méthode *void SupprimerPersonnage(personnage : Personnage)* de Manager. (2) Le manger commence par récupérer le nom de la série à la quelle le personnage appartient. (4) Il recherche et récupère la série qui correspond à ce nom *bool RechercherUneSerie(string nom, out Serie serie)* .(5) Le manger parcoure tous les groupes et supprime le personnage dans ces derniers. (6) Le manager cherche à récupéré toutes les relations dans lesquels le personnage apparaît, (7) il les récupère. (8) Il modifie toutes les relations qu'il avait récupéré précédemment en appelant la méthode *void ModifierRelation (string nom)* de Relation sur chaque relation. (9) Le manager supprime le personnage de la série dans laquelle il était contenu, en appelant la méthode *void SupprimerUnPersonnage(Personnage personnage)* de Serie sur la série récupérée à l'étape 4. (10) On récupère le nombre de personnages contenu dans la série qui contenait précédemment le personnage que l'on vient de supprimer en faisant serie.Personnages.Count(). Si ce nombre vaut 0 alors on appelle la méthode *void SupprimerSerie(Serie serie)* de Manager.