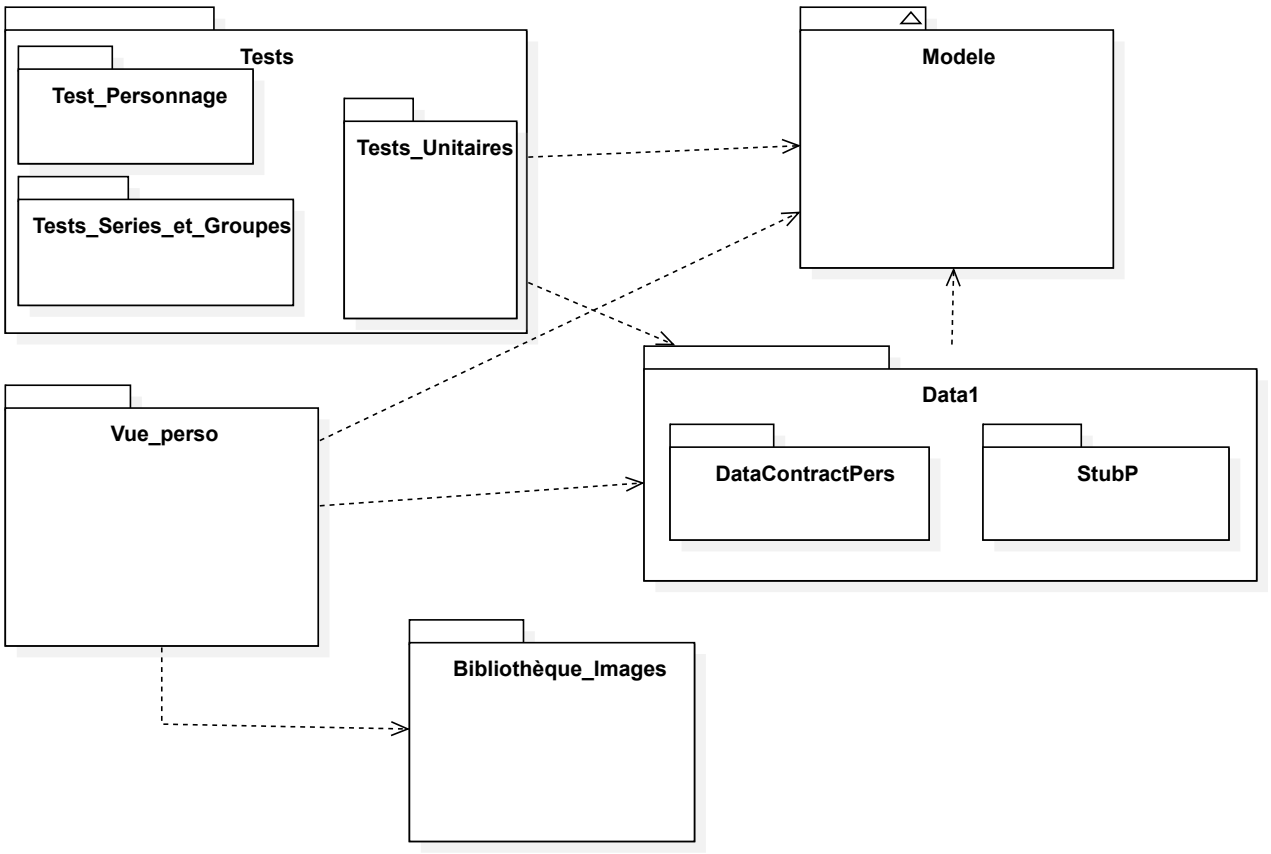
Explication architecture de l'application

Diagramme de paquetage :

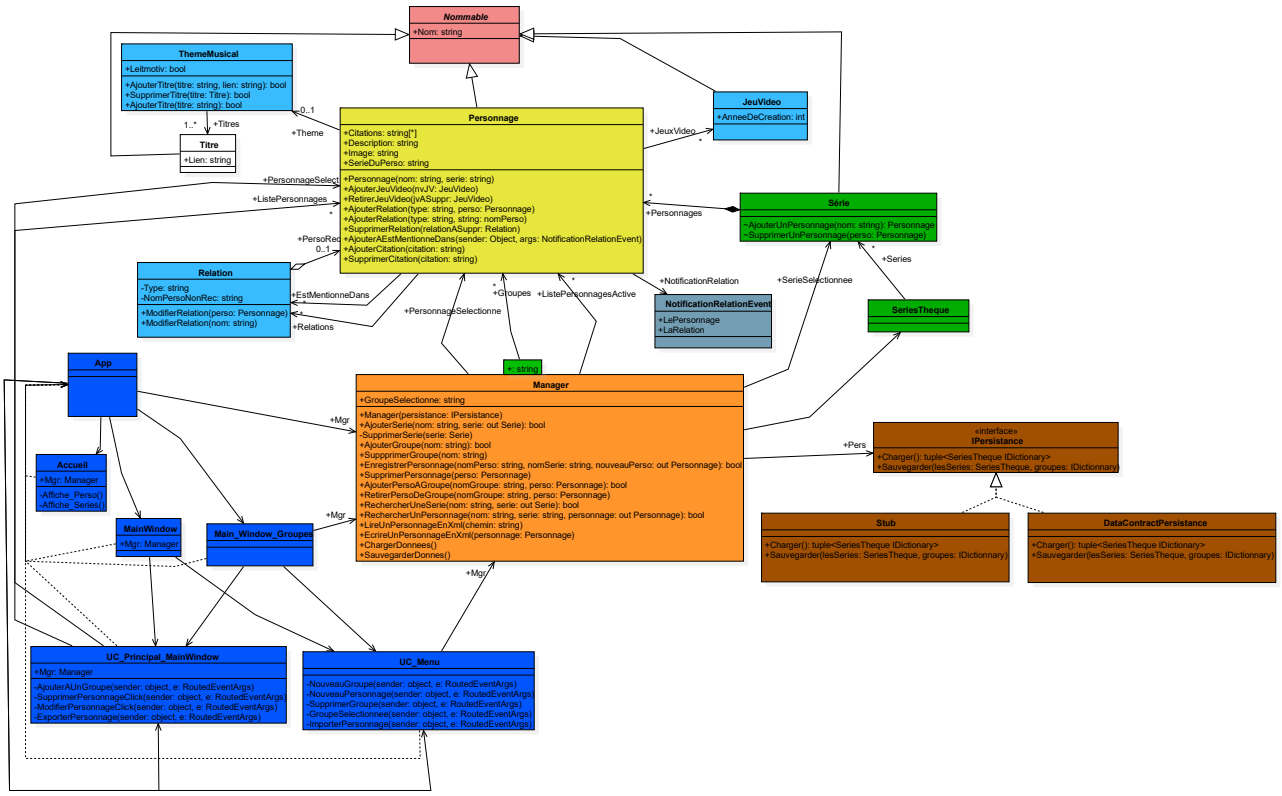


Voici le diagramme de paquetage de notre application. Il est composé de plusieurs projets :
• une bibliothèque d'image comprenant les images de l'application
• Vue_perso en charge de la partie graphique de l' application
• Modele contenant le coeur de l'application
• Data1 composé des projets nécessaires à la mise en place de la persistance, avec notamment le stub qui contient les données utilisées dans les tests
• Tests lui même constitué de 3 sous-projets qui testent les fonctionnalités de l'application : Test_Personnage se concentre sur les méthodes impactant le personnage ; Tests_Series_et_Groupes est centré sur les fonctionnalités des séries et des 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.

Le fait de séparer le modèle de la vue nous permet de réutiliser notre modèle avec des vues différentes. La séparation des méthodes de persistance en assemblage 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.

Diagrammes de Classe :

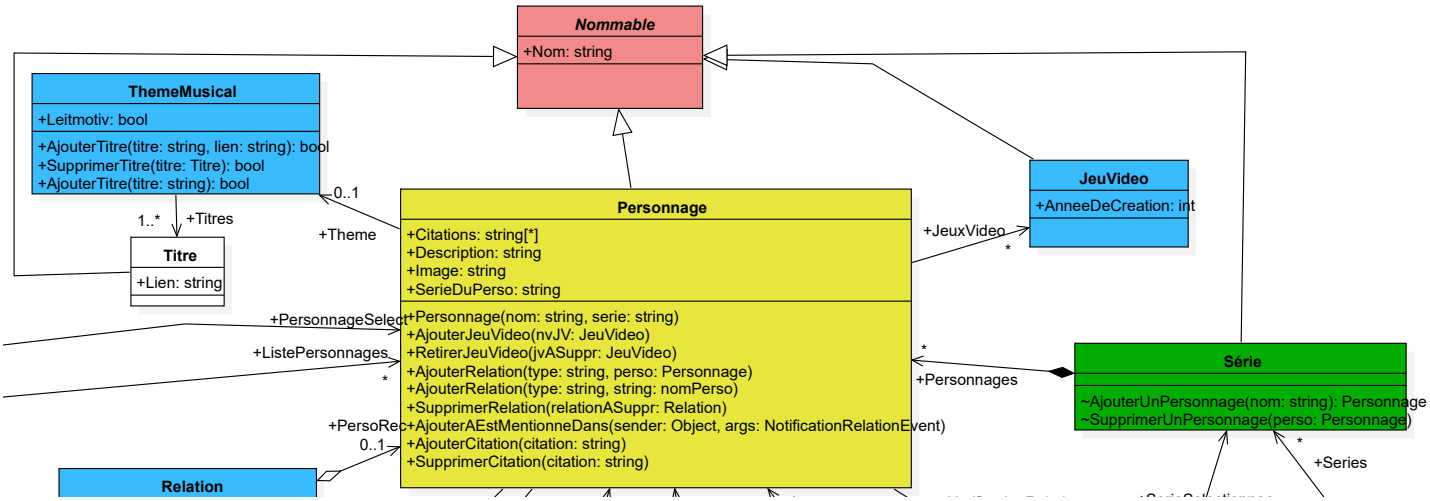
1.Diagramme général



Voici le modèle de notre application.
La classe `Manager` gère les interactions entre l'utilisateur, via la vue, et les principales classes du modèle : les classes `Serie et Personnage`. La classe Manager permet notamment de consulter les attributs des séries et des personnages qui leur sont associés. La recherche d'une série sera elle aussi intégrée. Cette classe comporte un attribut groupe, un dictionnaire qui permet à l'utilisateur de former des groupes de ses personnages favoris.
Le patron de conception utilisé dans notre modèle est donc la façade.

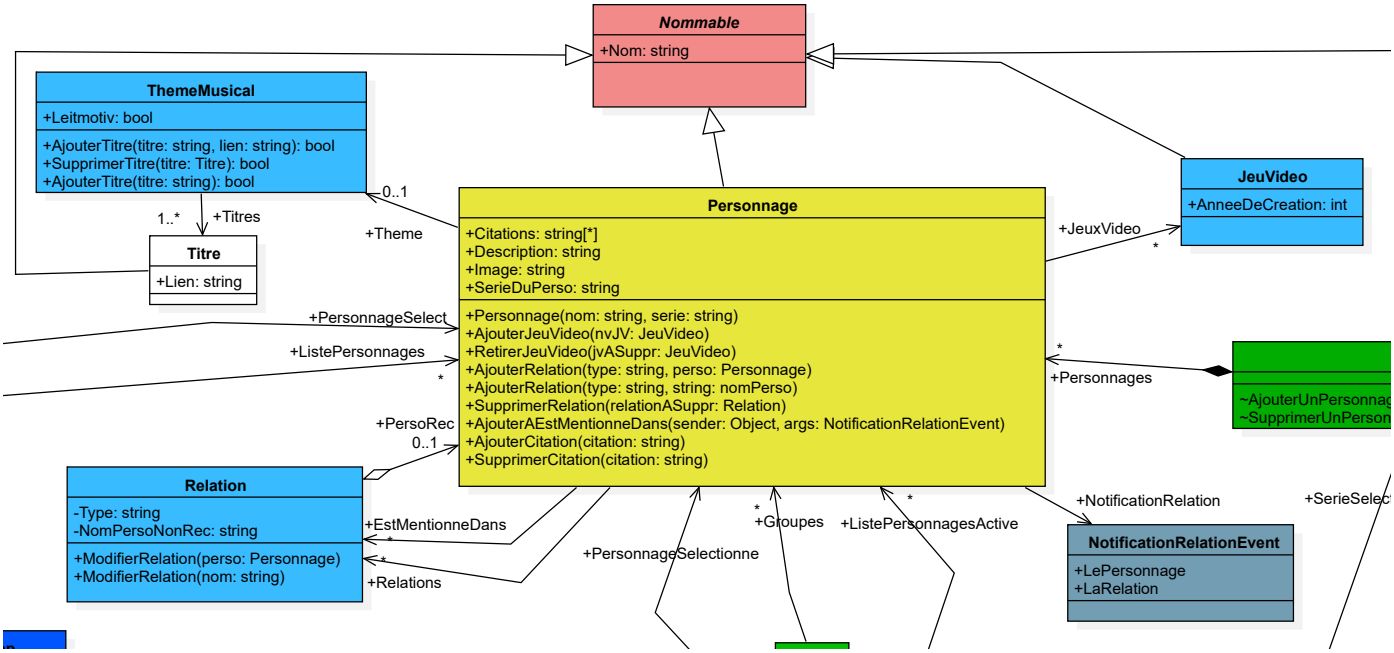
La façade, dans notre cas la classe `Manager,` permet de gérer avec une interface simple un sous-système plus complexe. Elle fait le lien entre différentes classes dans notre cas avec série ou avec personnage par l'intermédiaire du dictionnaire Groupes qui rassemblent les personnages que l'utilisateur décide de ranger dans un groupe qu'il crée au préalable. La façade délègue des requêtes à réaliser aux autres classes en utilisant leurs différentes. 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.La classe abstraite Nommable et ces héritiers



L'utilisation que l'on fait de la classe nommable est une factorisation de code : elle fournit l'attribut `nom` de type string et redéfinit également le protocole d'égalité pour ces sous-classes. Ce protocole d'égalité sera identique pour toutes les sous-classes de `Nommable`, bien qu'elles puissent avoir des comportements différents. Les sous-classes peuvent également bénéficier du polymorphisme issu de l'héritage : c'est le cas de Personnage qui redéfinit ce protocole d'égalité afin de le complémenter avec un attribut `SerieDuPerso` de type string.

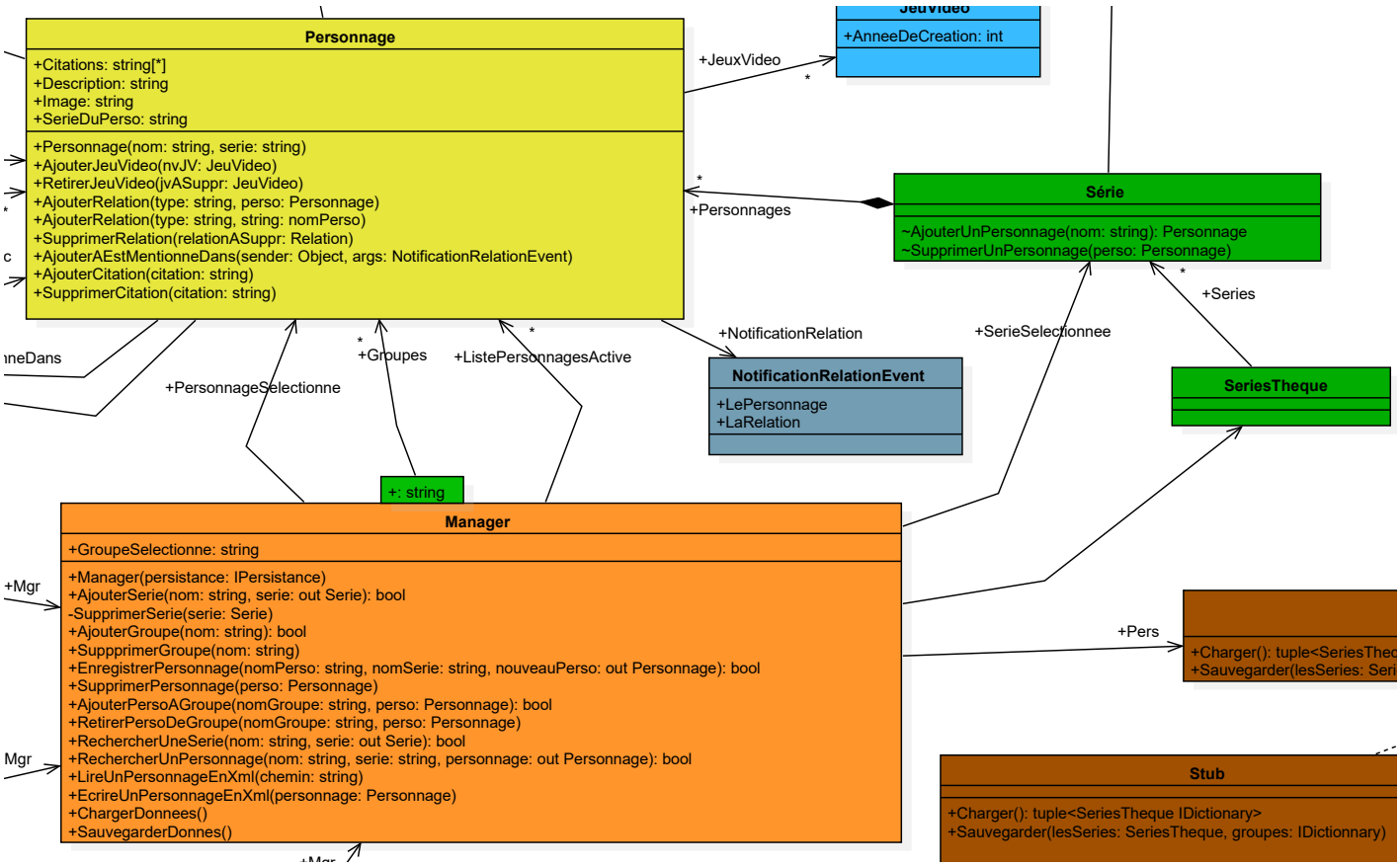
3.La classe Personnage



La classe personnage est le cœur de notre application, c'est elle qui renferme les éléments de la partie *detail* de l'application. Certains de ces éléments sont des objets issus de classes que nous avons créés nous même(Relation, JeuVideo, ThemeMusical). Ces classes sont représentées en bleu sur le diagramme de classe.

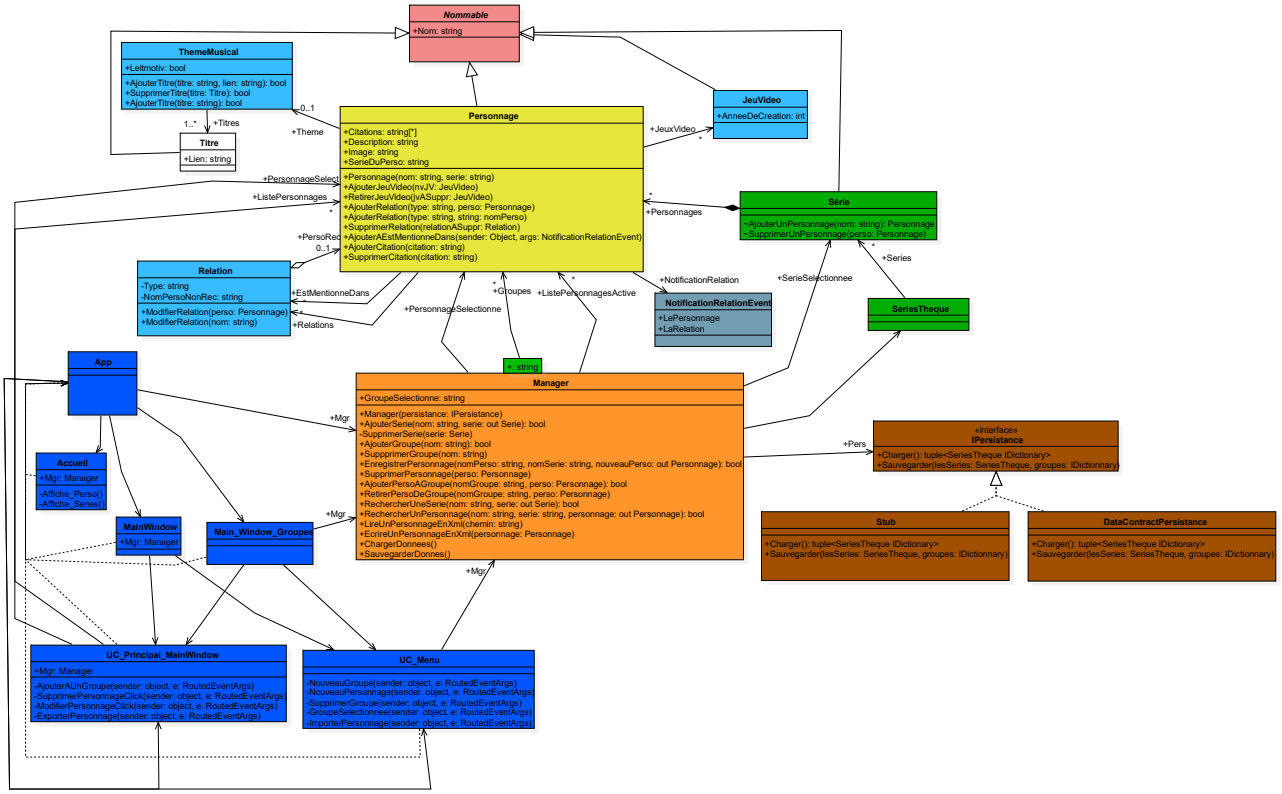
Nous avons aussi mis en place un événement en respectant le patron standard des événements qui se nomme `NotificationRelationEvent` en C#. Le patron qui se cache derrière cet événement est l'observateur. Il permet de notifier un changement. Dans notre cas l'événement `NotificationRelationEvent` permet, 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. Nousl'avons abonné à une méthode qui ajoute donc à ce personnage la nouvelle relation dans l'attribut EstMentionneDans qui est de type Relation. Ce choix d'utiliser un événement est plus évolutif, il pourrait nous permettre de mettre en place une fonctionnalité qui ajouterait une relation dans le personnage mentionne dans la relation.

4.Les collections contenues dans le `Manager`



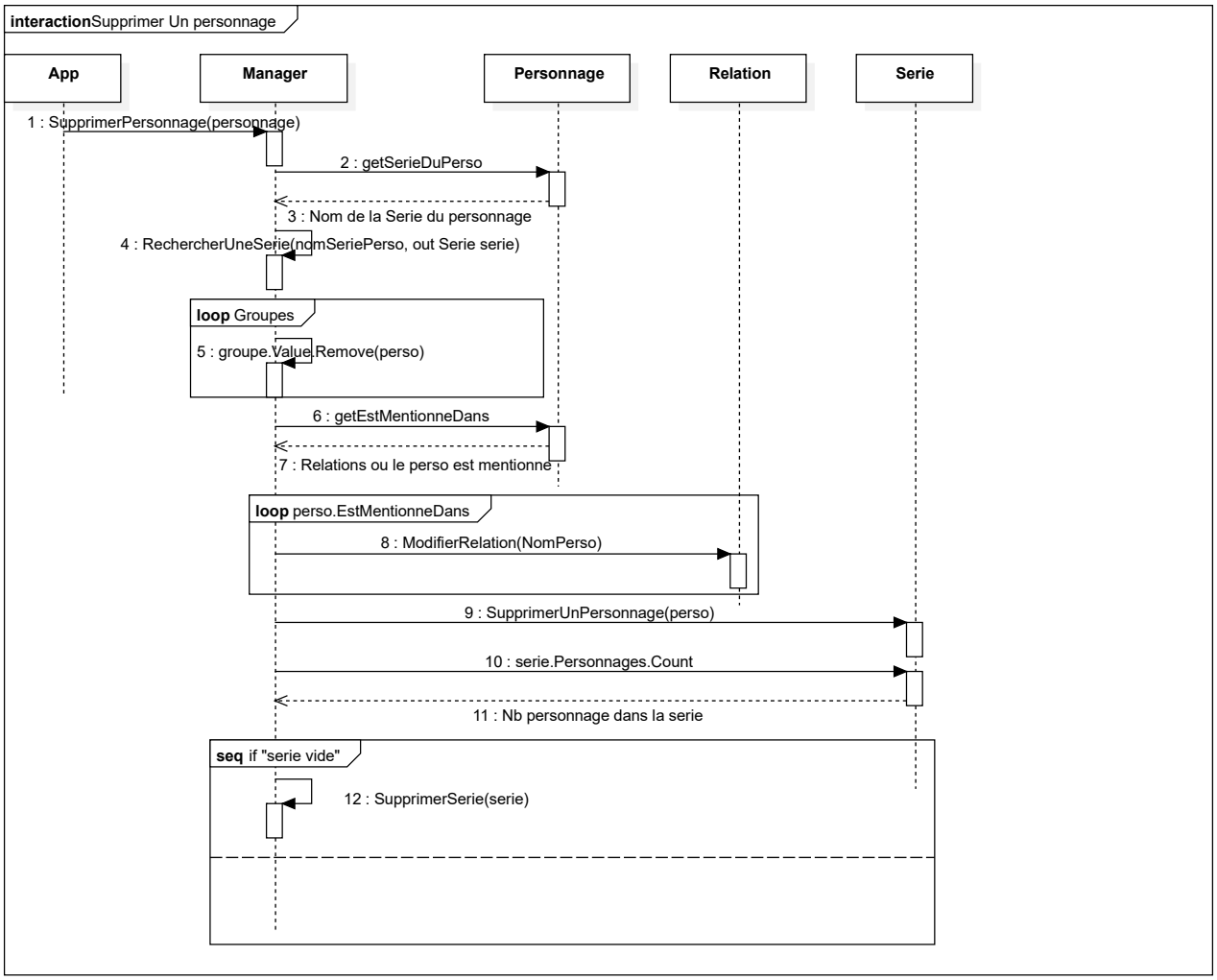
Le `Manager,` comme nous l'avons déjà dit précédemment, est la classe principale de l'application. Elle contient plusieurs propriétés qui sont utiles lors de son utilisation avec la vue. Mais les propriétés aux quelles nous nous intéressons ici sont celles du type des classes représentées en vertes. Nous avons un dictionnaire appelé `Groupes` qui contient les listes de personnages créées par l'utilisateur, la clé est un string et sa valeur associée est une liste de personnages.(par exemple une liste de favoris nommées « Favoris ») ; une propriété de type `SeriesTheque` qui contient une collection de `Serie,` chaque `Serie` stocke des personnages qui lui appartiennent (un personnage pouvant appartenir seulement à une série).Ces deux collections seront donc essentielles lors de la mise en place de la persistance : ce 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`, le cœur de l'application est en charge de fournir ses propriétés à la partie *detail* de l'application. L'événement `NotificationRelationEvent` derrière lequel se cache le patron observateur.
On peut également voir la vue (en bleu foncé), qui apporte quelques informations sur la manière dont elle fonctionne en utilisant le modèle.

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 `Manager` commence par retrouver 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 `Manager` parcoure tous les groupes et supprime le personnage dans ces derniers. (6) Le manager cherche 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 obtenue à l'étape 4.
(10) On récupère le nombre de personnages présents 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`.