Description de l'architecture :

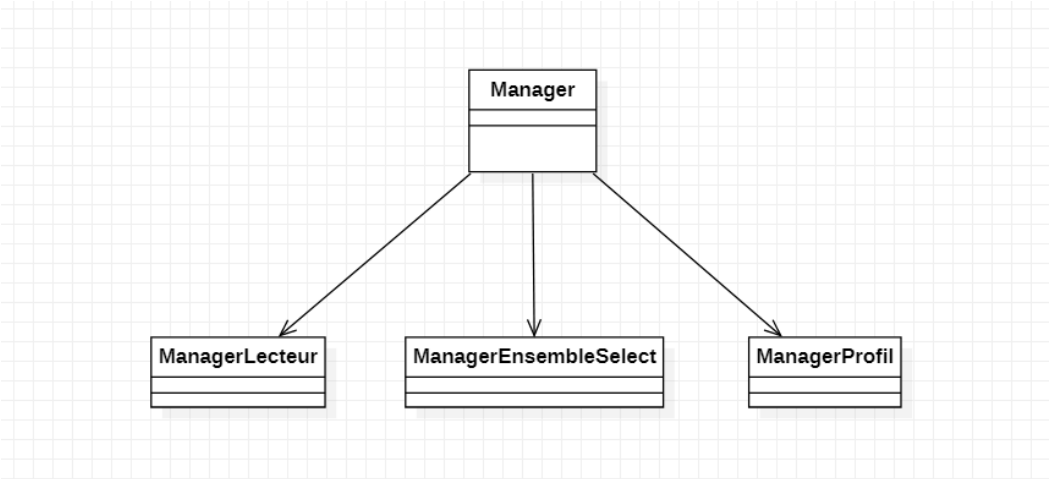
Projets et dépendances

Notre solution est découpée en plusieurs projets, qui possèdent tous un rôle précis ainsi que des relations de dépendances entre eux.

- Le Projet **Audium** : il s'agit de notre vue, il contient toutes les fenêtres XAML et UserControls nécessaires à l'interface de l'application, ainsi que chaque fichier de code-behind qui lui sont associés. C'est notamment dans ce projet qu'est développé toute la logique liée à notre lecteur Audio. Ce projet contient également quelques converters pour faire la liaison et traduire des données du modèle pour un affichage correct et cohérent.
  Ce projet dépend du package Gestionnaires, mais également d'un des projets de persistance, en plus de Material Design qui est souvent utilisé dans notre partie XAML.
- Le Projet **Gestionnaires** : Il contient toutes les classes de Manager venant réguler et gérer les données. C'est aussi ici que sont stockées les principales collections telles que les dictionnaires et les données utilisateurs. Cette bibliothèque dépend du projet Données. Cette bibliothèque constitue en soit le patron de conception Façade qui sera détaillé plus loin.
- Le projet **Données** qui structurent les différentes manières de stocker des informations. Il possède notamment la classe EnsembleAudio et la classe abstraite Piste, qui peut être implémentée différemment en fonction du type de média que l'on stocke. Ce projet ne possède aucune dépendance, et n'a besoin d'aucune référence externe pour fonctionner. Il ne sert qu'à contenir des informations, c'est pourquoi nous l'avons bien séparé de la bibliothèque Gestionnaires.
- **Test_Unitaire** : un projet contenant toutes les classes permettant de tester, avec le Xunit, des méthodes des bibliothèques Gestionnaires et Données. Ce projet dépend de Gestionnaires et de la librairie Xunit.
- **Test_URecherche**, **Test_Manager**, **Test_MangerProfil**, **Test_ManagerEnsembleSelect**, sont une série de projets de test fonctionnels sur les différents éléments de Gestionnaires, ils dépendent donc de celui-ci.
- **Test_Piste** : identique aux autres tests, mais uniquement dépendant de Données (il teste notamment le polymorphisme lié aux objets Pistes).
- **Stub** et **DataContractPersistance** : deux projets qui gèrent la persistance des données à leur manière, dont chaque classe implémente IPersistanceManager, une interface contenue dans Gestionnaires. Stub et DataContractPersistance ont donc chacun une dépendance vis-à-vis de Gestionnaires, et implémentent différemment deux méthodes clés du Gestionnaires, ChargerDonnées et SauverDonnées.
- **Test_DataContract** est un projet de test fonctionnel concernant DataContractPersistance et ne dépend donc que de celui-ci.

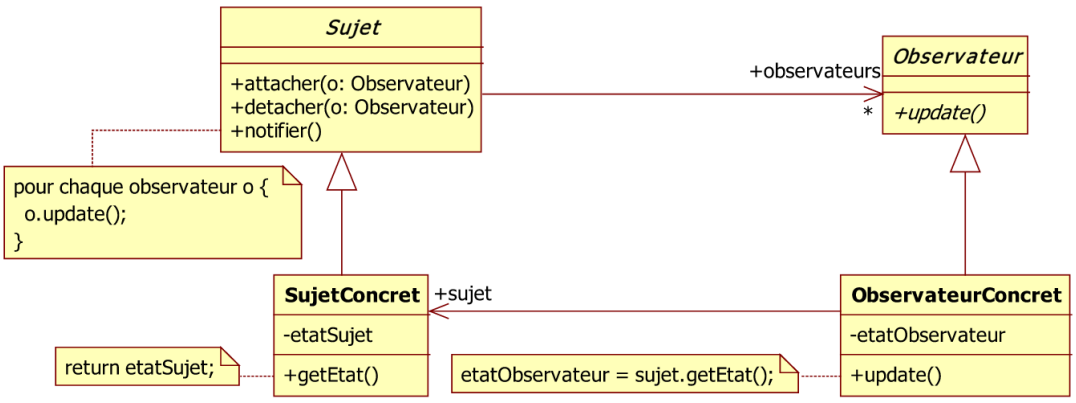
Patrons de Conception utilisés :

- La Façade



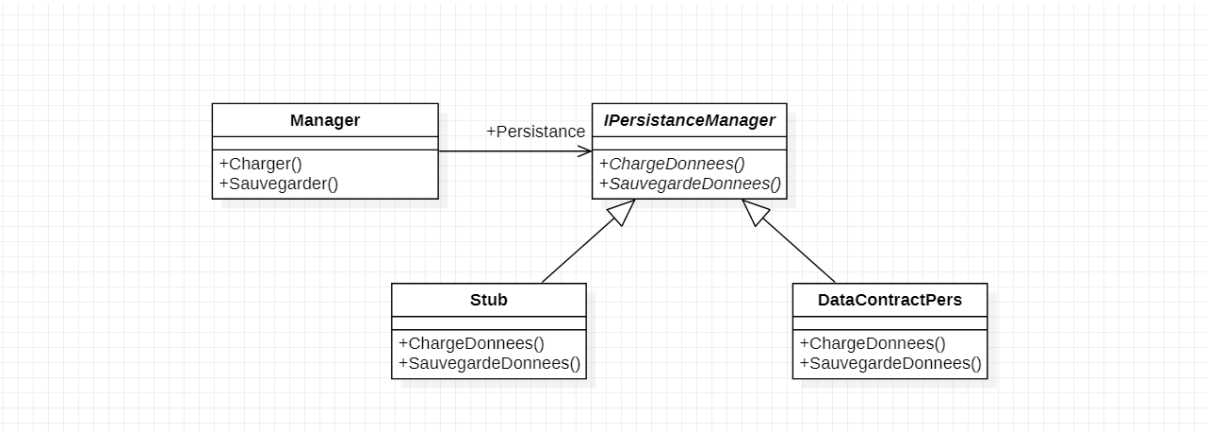
Le premier, et le principal des patrons de conception que nous utilisons est une façade. Le principe est simple, proposer une interface unifiée et simple, qui redirigera les requêtes d'un client au bon service, en fonction du besoin. Notre objet Manager, est le point d'entrée de notre partie Gestionnaires, qui, en plus de stocker notre dictionnaire et notre liste principale, contient également des sous objets qui sont spécialisé et délégué à certains cas d'utilisations. Ainsi, notre façade de Manager permet de simplifier le sous-système logistique, ne redirigeant les demandes de l'application qu'aux services qui lui sont utiles. De plus, on peut penser que si on rajoute ou modifie des fonctionnalités dans le futur, les clients ne verront aucune différence, et aucun souci de compatibilité puisqu'il ne seront toujours qu'en communication directe avec l'interface. L'autre gros avantage est de séparer les différentes fonctionnalités, et de rendre chacun des sous-Managers indépendants, ces derniers ne connaissant même pas la façade pour qui ils travaillent.

- L'Observateur



Le second patron qui nous est aussi fort utile est le patron observateur. Un patron observateur permet de notifier un objet du changement d'état d'un autre objet, l'un dépendant de l'autre. Concrètement, il s'agit de notifier notre vue que des informations, des variables de notre modèles ont été altérées par un programme ou par l'utilisateur et qu'il faut par conséquent mettre à jour l'interface qui correspond. On utilise pour cela l'interface INotifyPropertyChanged sur des objets de notre modèles dont les attributs sont susceptibles de changer pendant l'exécution, et le déclenchement du changement d'un état se fait par un événement appelé OnPropertyChanged.

- La Stratégie



Pour gérer notre persistance, nous utilisons également un patron de conception Stratégie. Un tel patron permet de modifier le fonctionnement et le comportement d'un objet en fonction d'un contexte qui lui est attribué. Ainsi, pour permettre de faire persister des données, nous utilisons deux méthodes différentes, l'une pour charger les données, l'autre pour les sauvegarder. En fonction de l'état de l'attribut Persistance de notre Manager, ces méthodes ne fonctionnent pas de la même manière et n'utilisent pas les mêmes technologies de gestion de données, pourtant elles doivent pouvoir être interchangeables et fonctionnelles. C'est le rôle de notre interface IPersistanceManager, et surtout des objets qui l'implémentent et qui peuvent appliquer une méthode de stratégie adaptée à la demande du Manager.