Problème :

Choisir les composants pour construire un pc peut être un vrai enfer.

Ma solution :

Un outil pour choisir ses composant celons ces besoins/ressources et rendre le choix d'un build plus facile.

Structure :

Ma solution est une appli MAUI codé en XAML et C#.

La structure est composée de différentes classes :

- La classe "Utilisateur", composée des attributs username, password, listBuild et listFav. Elle ne possède pas de méthode autre que son constructeur.
- La classe "Composant", composé des attributs id, prix, lien et type. Encore une fois pas d'autre méthode que le constructeur.
- La classe "Build", composé des attributs prix et image. Même chose que les 2 autres pour les méthodes.

Pour la classe "Utilisateur" :

Les attributs username et password sont des strings qui servent à la connexion des utilisateurs à l'appli.

L'attribut listBuild est une liste d'int, représentant les indices, dans la liste de build du manager, des builds qui ont été fait par cet utilisateur.

L'attribut listFav fonctionne de la même façon que listBuild : c'est une liste d'int des adresses des builds qui ont été mis en favoris par l'utilisateur.

Pour la classe "Composant" :

L'attribut id est un string qui est unique est qui est le nom (désignation) précis du composant, et permet de le retrouver sur d'autre site (exemple : "Gigabyte B550M AORUS ELITE" pour une carte mère).

L'attribut prix est un float qui représente le prix de chaque composant pour pouvoir estimer le prix total d'un ordinateur.

L'attribut lien est un string qui est un lien http vers un site où le composant est disponible à la vente pour pouvoir rediriger un utilisateur qui aurais envie de l'acheter.

L'attribut type est un TypeComposant, qui est une énumération des différents types de composants (par exemple carte graphique, boitier, processeur, ect …).

Pour la classe “Build” :

L'attribut prix est un float qui sert à stocker le prix total de tous les composants d’un build. Permet aussi de comparer les prix des ordis entre eux.

L’attribut image est un string qui contient le nom de l’image correspondant au pc, pour, au travers de data binding, afficher l’image.

Toutes ces classes sont regroupées et stocké dans le manager de l’application.

Ce manager, en plus des listes d’objets, possède un attribut Persi qui définit quelle persistance est utilisé pour le chargement et la sauvegarde des données : sois le stub, sois le data contract persistance. Ces deux classes sont des classes filles de l’interface “IPersistance”.

IPersistance :

C’est une interface définissant 2 méthodes : sauvegardeDonnee et chargeDonnee. Cette interface permet de donnée un type commun au stub et au data contract pour pouvoir passer de l’un a l’autre facilement dans le manager.

Stub :

Implémente les méthodes de sa classe mère, c’est à dire sauvegardeDonnee et chargeDonnee : sauvegardeDonnee n’implémente pas une vraie façon de sauvegarder car cette méthode n’est jamais sensée être utilisée. La méthode chargeDonnee quand à elle crée des valeurs par ligne de code et les passe au manager. La sauvegarde s’effectue donc via le data contract.

DataContractPersi :

Comme le stub, implémente les deux fonctions de IPersistance :

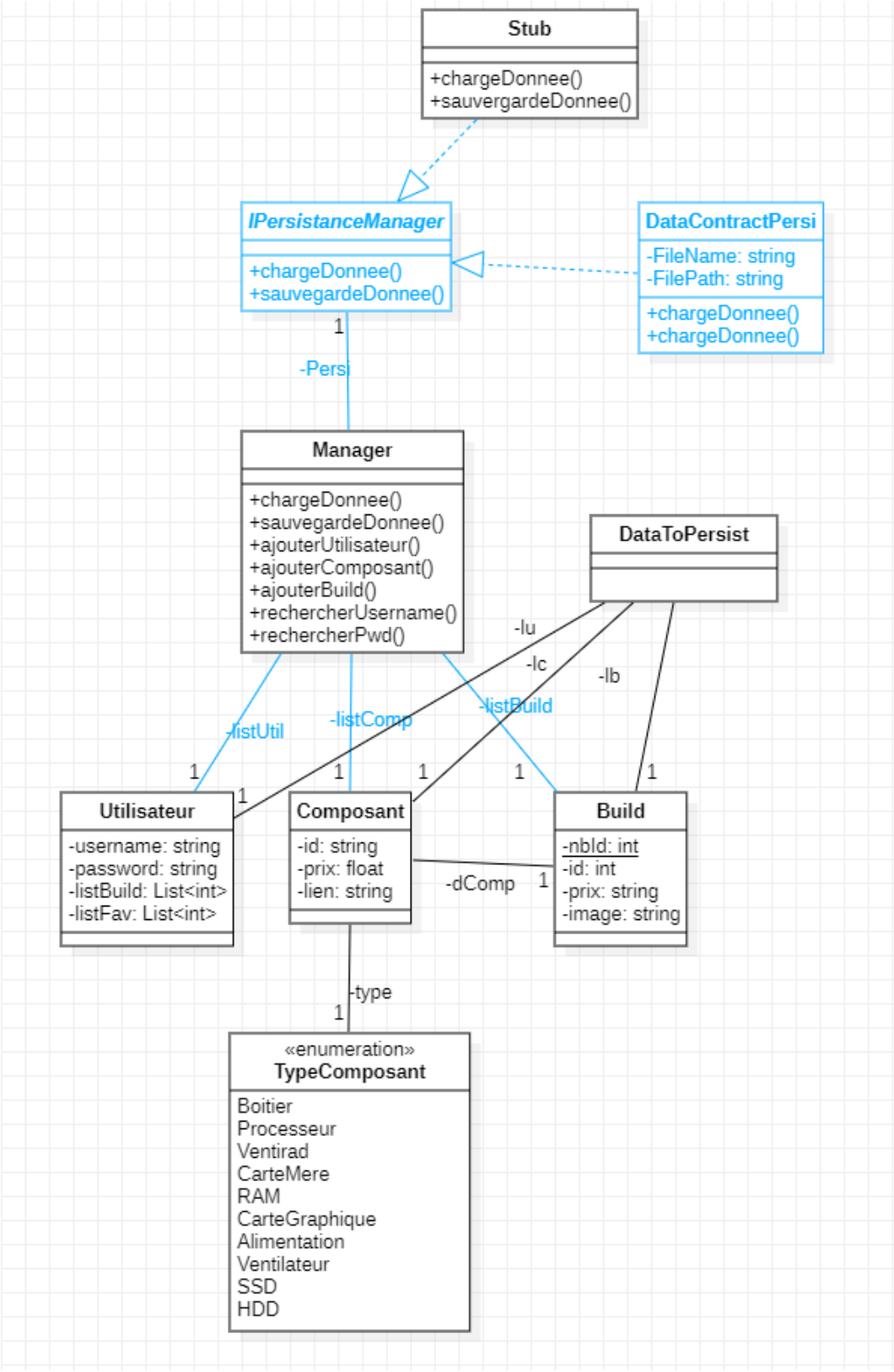
sauvegardeDonnee :

Prend les valeurs des listes du manager et les enregistre dans un fichier xml. C’est cette méthode de sauvegarde qui est utiliser dans tous les cas.

ChargeDonnee :

Lis les données écrites dans le fichier xml et les mets dans le manager.

Diagramme de classe :



Diagrammes de séquence :