Conception et Programmation Orientées Objets (C#, .NET)

Diagramme de classe de l'application :

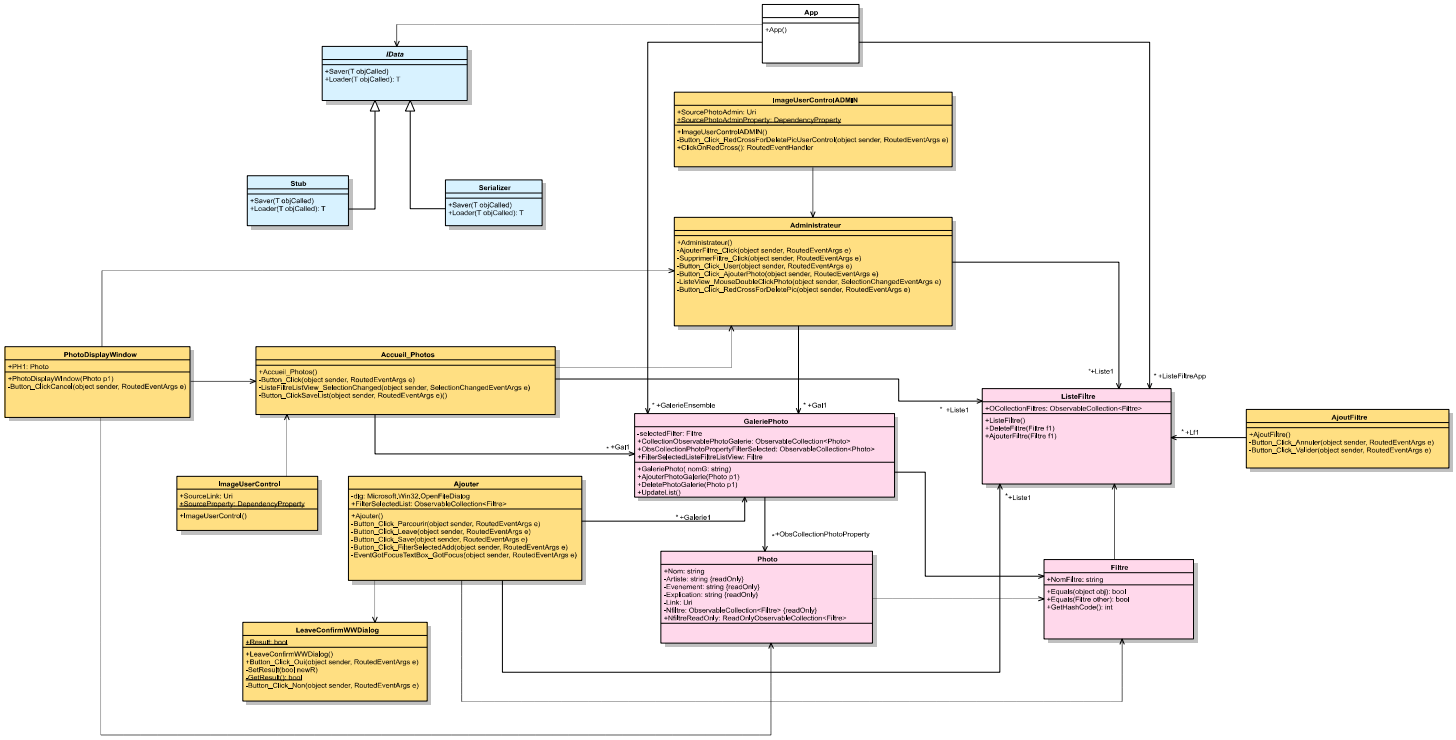


Figure 1 : Diagramme de Classe de l'application (cf Documents annexes n°1)

Les classes en **Jaune** représentent les éléments de la vue, les fenêtres.

Les classes en **Rose** représentent les classes métier.

Les classes en **Bleu** représentent les classes traitant les données : La Serialization et le Stub.

La seule classe Blanche est l'App, c'est dans cette classe que toutes les autres classes de l'application récupère les listes communes : ListeFiltre et Galerie.

Sur ce Diagramme, nous pouvons remarquer qu'il y a 2 grandes classes principales : Galerie et ListeFiltre. Ces dernières sont utilisées par l'ensemble des autres classes (métier et vue inclus).

GaleriePhoto utilise Filtre et Photo puisque GaleriePhoto est une collection observable de Photo. Filtre est utilisé pour récupérer le Filtre sélectionné dans la ListView d'Administrateur et Accueil Photo

En regardant le diagramme nous pouvons aisément voir qu'il y a deux grandes « fenêtres » :

- Accueil Photo
- Administrateur

Ces deux fenêtres récupèrent sensiblement les mêmes listes – GalerieEnsemble et ListePhoto -, tenues via la classe « App », qui elle récupère ses données grâce au Stub ou à la Serialization donc si il y a un changement de fenêtre, de Accueil Photo vers Administrateur ou inversement, l'Utilisateur retrouvera les mêmes listes de photos et de filtre.

Il y a également deux autres classes utilisées - par très peu de classes - mais sont essentielles au fonctionnement de l'application puisque les listes précédemment énoncées manipulent des Filtres et des Photos.

Pourquoi avoir fait ces choix de conception ?

Nous avions un cahier des charges à respecter avec des contraintes à respecter, l'une d'elle était qu'une photo puisse contenir plusieurs Filtres afin de faciliter encore plus leurs gestions. Donc par conséquent nous devions introduire une ListeFiltre dans une photo. Nous obtenions donc deux listes de filtres : Une dans le Detail de l'application, et une dans la photo. L'application contiendra forcément plus d'une photo donc elle stockera *N* liste filtre pour *X* Photos, plus celle du Detail : (N*X)+1.

Vous pourriez légitimement vous demander, « pourquoi « App » est précisée dans ce diagramme ? ». Car « App » nous sert de *« distributeur »* : Nous devions trouver un élément commun à Accueil Photo et Administrateur où stocker nos différentes listes, car après chaque ajout, suppression… elles devaient toutes se mettre à jour et être identiques pour avoir les mêmes listes dans Accueil Photo et Administrateur, donc lors de l'instanciation de la fenêtre, le constructeur récupère le contenu des listes dans l'App et les stocks dans une propriété. *(cf image ci-dessous)*

```
36            Liste1 = (Application.Current as App).ListeFiltreApp;
37            Gal1 = (Application.Current as App).GalerieEnsemble;
38
```

Figure 2 : Récupération du contenu des différentes liste dans App

Rappelons également que « App » puise ses données depuis l'interface *IData* qui possède deux classes filles : *Serializer* et le *Stub*.

Notons que ces deux classes filles possèdent des particularités uniques : Elles sont génériques. C'est-à-dire que peu importe l'élément passé en paramètre des méthodes de Serializer – le cas qui nous intéresse, puisque la généricité du Stub n'est pas essentielle – elles fonctionneront, de plus, le Serializer est *automatique* : Il trouve le nom des fichiers requis, et donne un nom de fichier automatiquement à la sauvegarde.

La méthode « Saver » du Serializer permet de simplement sauvegarder la classe appelée en paramètre, et c'est ici que la généricité intervient : N'importe quel objet peut être passé en paramètre de Loader et il sera sauvegardé avec un nom précis : *Save+* ***type de l'objet*** *+.dat*.

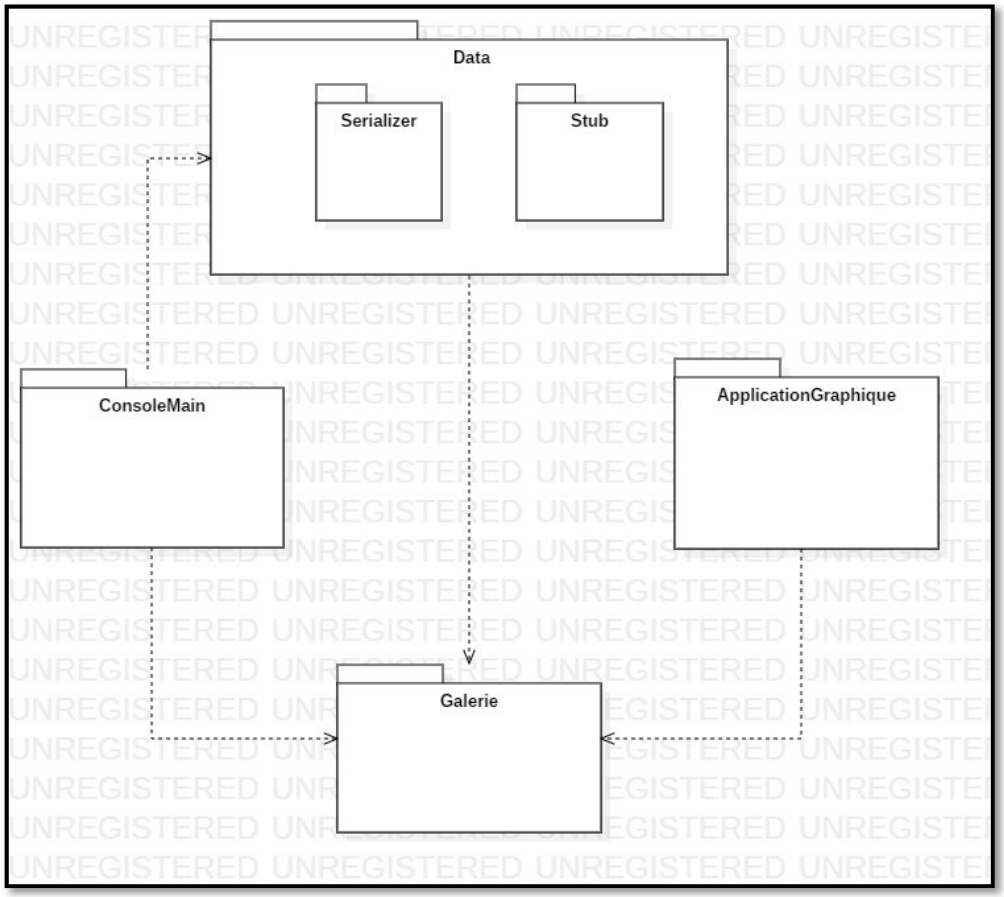
Edit *:* Désormais, les sauvegardes s'effectuent dans le dossier « Mes Documents » de l'utilisateur, et pareil pour les chargements. Si les fichiers demandés ne sont pas présent, les données sont chargées à partir du stub, et les fichiers créés pour faciliter les manipulations.

```
string nomFichAEnregistre = "Save";
nomFichAEnregistre += objCalled.GetType().Name;
nomFichAEnregistre += ".dat";
```

Figure 3: Code permettant la génération automatique d'un nom de fichier

L'ouverture d'un fichier – via la méthode Loader – fonctionne de la même manière : Elle récupère un objet en paramètre, puis le type de l'objet donné, et la méthode trouvera seule le fichier de sauvegarde.

Diagramme de paquetage de l'application :



Ce diagramme décrit les différents package utilisés par l'application pour fonctionner. Nous pouvons voir le package IData qui possède deux classes filles : Serializer, qui obtient les données de l'application à partir des fichiers de sauvegarde. Comme expliquer dans la partie **La persistance** du fichier **Projet Tuteuré** : si les fichiers de sauvegarde ne sont pas présent, ils sont créés et des données

prétéléchargées son chargées à partir du Stub et sauvegardées dans le fichier Mes Documents dans un dossier nommé PhotoStockSave.

Les différents package utilisent un des package les plus important de l'application : Galerie , c'est le package qui contient toutes les classes métiers utiles pour la création des données (Photos, Filtres ….). Ce dernier ne dépend d'aucun autre package, mais *a contrario* tous les autres packages l'utilise afin de traiter les différentes données.

ConsoleMain est le package qui permet de tester l'application durant le développement de l'application, afin de repérer les différentes erreurs, les différents cas d'erreur, conflits... Ce package est donc important pour le développement de nos classes et comme il sert qu'aux différents test, il ne dépend pas du package Data, et interagie directement avec les classes métier de Galerie.

ApplicationGraphique est le package de la Vue, contenant les fenêtres et les code-behind.