Projet Tuteuré

La persistance et partie personnelle :

Dans le Diagramme de classe présent dans *DOC/Conception et Programmation Orientées Objets (C#, .NET)* nous pouvons mettre en évidence trois classes qui vont nous intéresser pour la partie suivante : La Persistance.

Dans cette rubrique je vais vous présenter les exigences que j'avais imposé par rapport au cahier des charges du projet et par quel moyen(s) j'ai répondu à ces exigences.

Je présenterai également ***pourquoi*** j'ai effectué ces choix de conception et plus précisément à quel(s) besoin(s) j'ai répondu via ces exigences.

Voici ci-dessous, un découpage du diagramme de classe afin de mettre en évidence la partie qui nous intéresse :

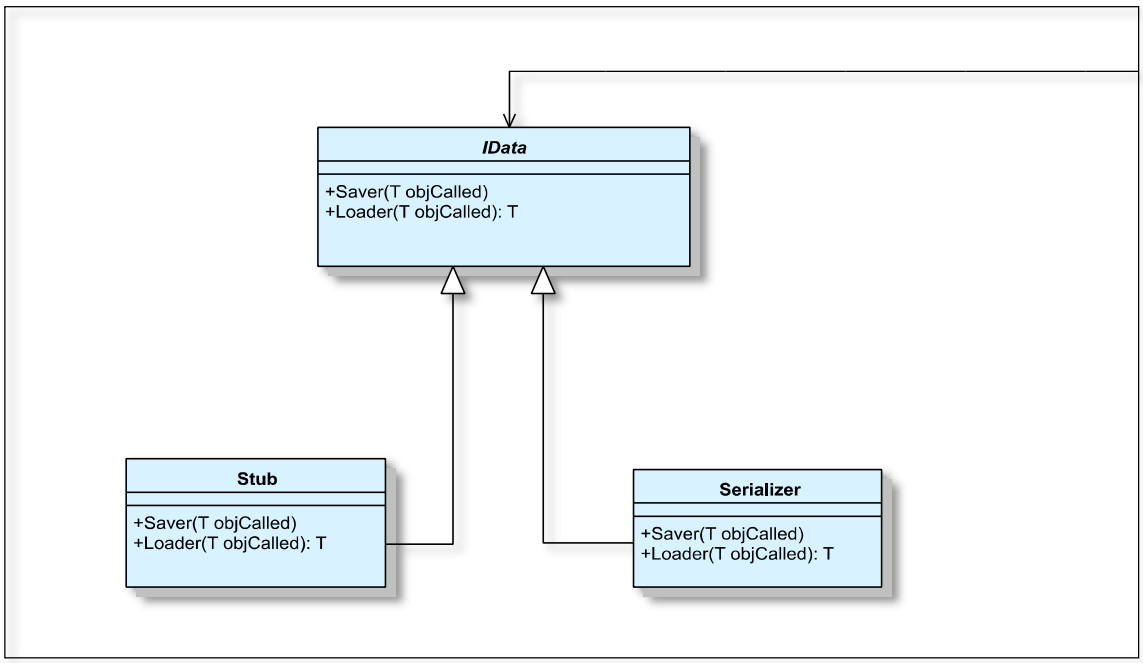

Figure 1 : Classes permettant d'effectuer la persistance (extrait du diagramme de classe de l'application)

Pourquoi avoir fait une interface IData ?

Nous devions utiliser deux classes traitant les données mais de manières différentes : En Mémoire et à partir d'un fichier de sauvegarde. Ces deux techniques nécessitaient alors une séparation des responsabilités puisque ces deux classes possédaient les mêmes méthodes, mais définies différemment. Nous avons alors décidé de créer une classe - interface - mère avec deux méthodes : Saver et Loader.

Quelle est la différence entre le Stub et le Serializer ? Pourquoi les séparer ?

Les classes Stub et Serializer sont séparées car elles n'ont pas les mêmes définitions pour leurs méthodes.

Le Stub lui est chargé de fournir des données simulées – instanciées, en mémoire – pour tester l'application, elle ne peut donc **pas** effectuée de sauvegarde, d'où la méthode vide. (cf code) mais elle peut charger les données via la méthode *« Loader » (Cf Figure 1)* .

Contrairement au Stub, le Serialiazer a ses deux méthodes définies. *« Loader »* lui permet de charger les données de l'application à partir d'un fichier de sauvegarder localisé dans « Mes Documents », mais si le fichier n'est pas trouvé, ou pas chargé, les données du Stub sont alors utilisées et les fichiers sont créés. La deuxième méthode est *« Saver »,* elle permet de sauvegarder l'élément passé en paramètre de la méthode dans un fichier.

Comment fonctionne cette sauvegarde avec la Serialization ?

Nous voulions que l'utilisateur n'ait à choisir ni le chemin du fichier, ni le nom du fichier afin de parer toute erreur humaine. Donc nous avons choisit de faire ces sauvegarde et chargement, **générique.** C'est-à-dire que le fichier sera sauvegardé dans un emplacement spécifique, avec un nom spécifique et cela avec n'importe quel objet.

Pour sauvegarder un objet il suffit d'initialiser et instancier un Serializer et d'appeler la méthode « Saver » avec comme paramètre l'objet que l'on souhaite sauvegarder… et c'est tout. La méthode Saver est définit de telle manière que le nom de fichier sera choisi en fonction du **type** de l'objet et sera constitué de la manière suivante : *Save+* ***type de l'objet*** *+.dat*. Et cela est possible grâce à ces 3 lignes :

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

Figure 2 : Code permettant la sélection automatique du nom de fichier

Mais où ce fichier est-il sauvegardé ?

Le fichier sera créé dans le répertoire « PhotoStockSave » dans le dossier « Mes Documents », et si ce dossier n'est pas présent il sera automatiquement généré pour que la sauvegarde se fasse.

Et le chargement ?

Il fonctionne – dans la logique – de la même manière que la méthode *« Saver » :* la méthode récupère le nom du fichier en fonction du type de l'objet passé en paramètre et le charge automatiquement. Mais si les fichiers n'existent pas, la méthode Loader du Stub est appelé et les fichiers sont créés avec la méthode Saver du ***Serializer.***

La persistance dans le diagramme de paquetage :

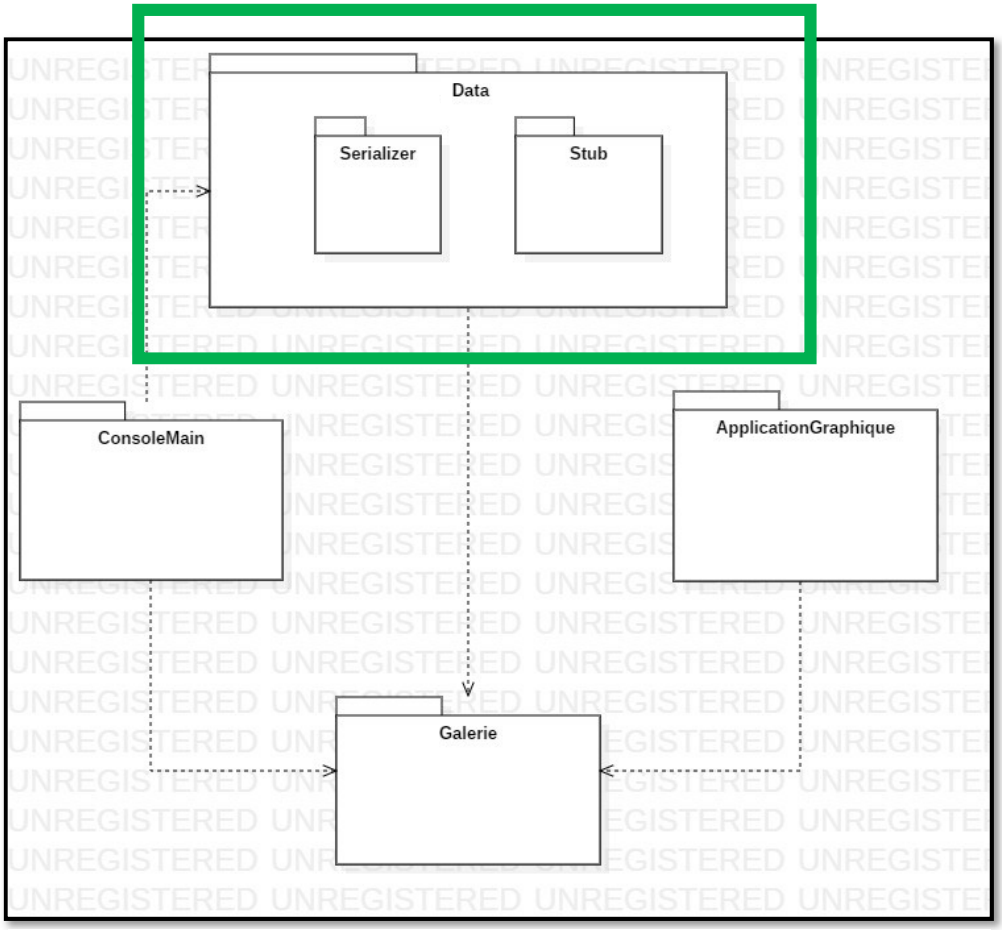

Figure 3 : Diagramme de paquetage incluant la persistance

Le rectangle vert permet de mettre en avant la partie nécessaire à la persistance, comment dit précédemment, incluant le Serializer et le Stub.

Pourquoi un package « à part » du reste ? Pourquoi ne pas l'avoir mis dans Galerie ?

Le paquetage Data est séparé des autres paquetages car elle n'est utilisé que par Galerie et ConsoleMain. Donc pourquoi mettre Data en dehors de Galerie alors que ConsoleMain utilise Galerie ? Car Data n'est pas essentielles aux autres classes métier, elles fonctionnent sans cette dernière, ce qui permet alors à Data d'être « interchangeable » : Ce paquage pourrait être remplacé ou supprimé sans créer des conflits à l'intérieur du projet par exemple.

L'autre avantage de cette séparation est d'éviter à la bibliothèque Galerie d'être surchargé, notamment de classes non essentielles, alors que cette dernière est essentielle au bon fonctionnement de l'application.