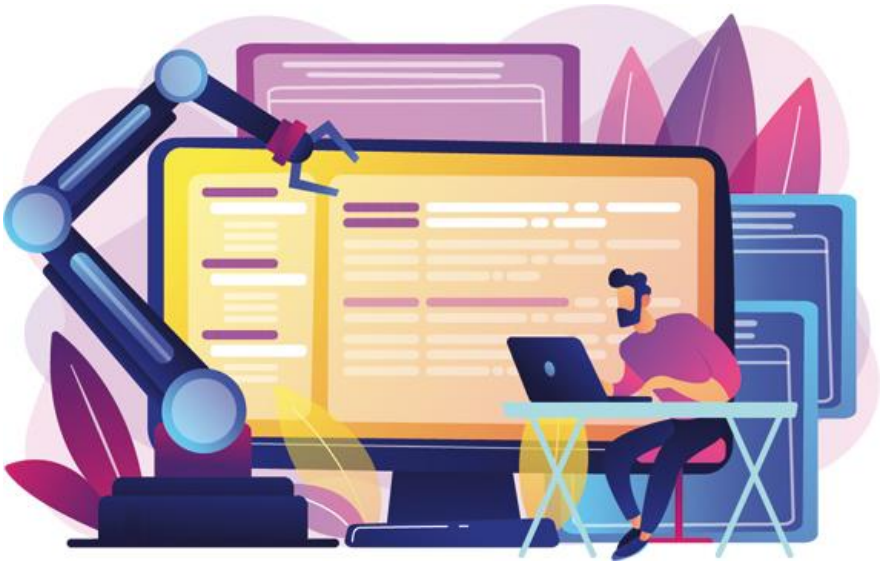

Éditeur de notes C#

Thomas DEVIENNE 4H
BIZET Matthias 3H

26 avril 2021

Diagramme de paquetage

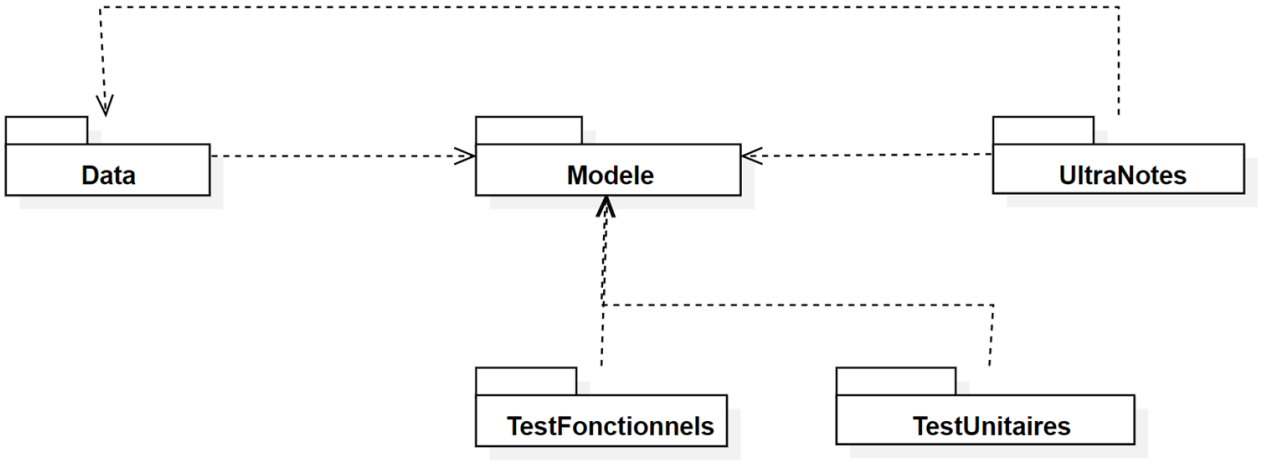

Figure 1 : Diagramme de paquetage de l'application

Le diagramme de paquetage est composé de plusieurs éléments. La première raison de séparer le code de l'application en plusieurs paquages est de pouvoir les rendre interchangeables. En effet, nous développons actuellement une application de bureau. Cependant, peut-être que par la suite nous souhaiterons porter l'application sur appareils mobiles, ou encore sur le web. Pour faire cela, inutile de refaire l'application de zéro : il suffit de remplacer le paquage UltraNotes puis de réutiliser les autres paquages dans notre application. En effet, chaque paquage est indépendant des autres et peut être remplacé par un autre. Encore, si nous souhaitons améliorer la partie persistance de notre application (sauvegarde des données), il nous suffit de remplacer le paquage Data par un autre (exemple utilisation d'une base de données plutôt que des fichiers binaires). Notre application est composée des paquages ci-dessous.

Les **bibliothèques de classes** sont des « dossiers » contenant exclusivement des classes. Modele définit toutes les classes concernées uniquement par la logique de l'application. Ainsi, les collections de notes, les propriétés décrivant ces objets et la classes Manager sont définies dans ce paquage. La classe Manager permet d'effectuer toutes les actions de notre application. Son but est de simplifier l'accès à tout notre sous-système de classes dans le Modele mais ne les interdit pas pour autant. En effet, nous pouvons toujours accéder à des méthodes de classes « derrière » Manager pour faire du Binding par exemple. Manager joue donc le rôle de façade car il fait le pont entre la vue et le Modele, entre l'application et son fonctionnement interne. Ainsi, elle se contente d'appeler les méthodes des autres classes dans ces méthodes. Cela permet une meilleure compréhension du programme et facile aussi l'utilisation de notre application. En effet, il n'est pas nécessaire de connaître toutes les méthodes de toutes les classes du Modele pour utiliser notre application, seulement celles de Manager. Encore, Data est une seconde bibliothèque de classe qui est chargée de la sérialisation (sauvegarder un objet) de l'objet Bouquin. Bouquin est une classe qui regroupe une collection de Note et les gère. Il n'est pas nécessaire de sérialiser la classe Manager car elle ne fait que déléguer le travail à la classe Bouquin, qui contient toutes les données utiles à savoir les notes et les paramètres de l'application. C'est donc la classe Bouquin qu'il faut sérialiser. Toutes les bibliothèques de classes sont en .NET Standard 2.1.

Le paquage TestFonctionnels est une **application console** qui sert à tester les fonctionnalités de l'application. C'est dans ce paquage que nous testons les classes qui gèrent les fonctionnalités de l'application, décrites dans le diagramme de cas d'utilisation (créer une note, supprimer une note, etc.).

TestUnitaires est un paquage de **tests Xunit**. Dans ce paquage, nous testons les méthodes de nos classes avec des tests unitaires.

Enfin, UltraNotes est une **application WPF** .NET Core 3.0. Dans ce paquage sont organisées en sous-dossiers les vues de l'application (la vue principale et les user control) ainsi que les classes qui y sont associés et une classe qui sert à convertir un objet de type FlowDocument en fichier de balisage XML.

On remarque d'ailleurs que tous les paquages dépendent de Modele.

Diagramme de classe

1. Diagramme de classe de Modèle

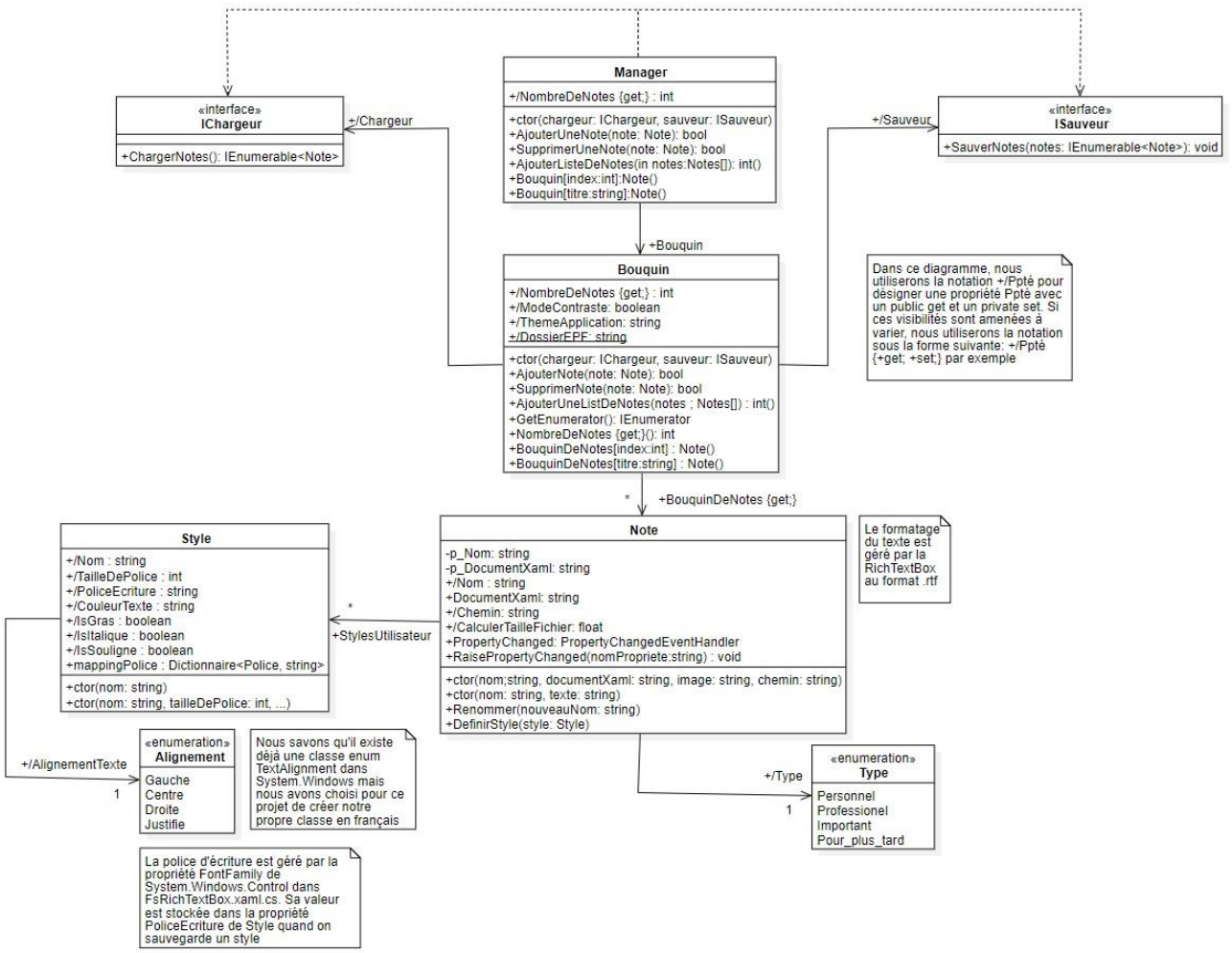

Figure 2 : Diagramme de classe du paquage Modele

La classe **Manager** gère les fonctionnalités de l'application. Cette classe possède une instance de la classe Bouquin et la manipule en appelant ces méthodes. Cela permet à l'utilisateur de seulement instancier dans son code une instance de la classe Manager pour manipuler un Bouquin, sans avoir à connaitre toutes les méthodes de cette dernière. La classe Manager possède une propriété calculée NombreDeNotes qui renvoie le nombre de notes dans le Bouquin. Les méthodes d'ajout et de suppression de notes permettent de fournir et gérer le Bouquin. Les indexeurs permettent d'accéder aux éléments du Bouquin depuis Manager en utilisant la notation manager[index]. Le constructeur de la classe nécessite une instance IChargeur et ISauveur. Ces deux interfaces servent à charger les données dans le bouquin et les sérialiser. Le bouquin de la classe Manager sera ainsi instancié avec ces objets. Ceci permet de pouvoir manipuler manager comme si c'était un Bouquin, mais en gardant les avantages d'une façade.

La classe **Bouquin** gère une collection de notes. Elle possède donc une propriété calculée qui renvoie le nombre de notes dans le Bouquin et de quelques paramètres utilisateurs à sérialiser. La propriété ModeContraste précise si le thème colorimétrique de l'application doit être accentué ou non. Cette option est utile pour les personnes peine à lire des textes écrit en blanc sur les fonds colorés par exemple. L'application est donc en noir et blanc (sauf dans la partie éditeur de texte). ThemeApplication définie la couleur dominante

de l'application, par défaut le bleu. DossierEPF (EPF pour Dossier d'Enregistrement Par Défaut) est le dossier dans lequel les notes vont être enregistrées par default si l'utilisateur ne précise pas de chemin. Ces paramètres sont stockés dans la classe Bouquin car ils font partis des réglages utilisateurs et sont donc propres à son « plan de travail ». Cette propriété est statique pour pouvoir y avoir accès sans forcément créer une instance de Bouquin. Ensuite, la classe Bouquin possède des méthodes qui permettent d'ajouter et de supprimer des notes pour fournir et gérer le Bouquin, ainsi que des indexeurs pour retrouver les informations rapidement dans la collection BouquinDeNotes qui contient donc la collection de Notes. La classe implémente l'interface IEnumerable pour pourvoir être parcourue. Elle redéfinie sa méthode GetEnumerator(). Enfin, le constructeur prend en paramètre deux objets : un de type IChargeur et l'autre de type ISauveur. Ces objets permettent aux méthodes ChargerBouquin() et SauverBouquin() de désérialiser et sérialiser l'objet. La méthode de sérialisation et désérialisation dépends du type d'instance des propriétés Chargeur et Sauveur, ce qui permet à un utilisateur de pouvoir changer très facilement de méthode de sauvegarde et récupération de données. En effet, il lui suffit de changer le type d'instance des paramètres lors de l'instanciation d'un Manager ou d'un Bouquin.

La classe **Note** définie donc une note. Une note possède un Nom donné par la propriété Nom, un contenu qui est la représentation de la note en XML (la note est un FlowDocument) donné par DocumentXaml et un chemin sur le disque donné par Chemin. Une propriété calculée permet de calculer la taille du fichier. Les champs p_Nom et p_DocumentXaml sont les variables associées aux propriétés du même nom. Ces deux propriétés envoient une notification lorsque leur contenu change pour notifier la vue de leur évolution. Une note possède également une liste de styles utilisateur qui lui est propre. Un Style est un ensemble de paramètres qui définissent la mise en forme d'un texte. Par exemple, un titre peut être définie comme écrit en gras, en police Cantarell en taille 15. La classe Note dispose également de d'une propriété Type qui définit son type. Un Type définie quel est le type de document. La note peut avoir été rédigée pour le travail ou à titre personnel par exemple. Ces types sont cumulables. Les méthodes de Note permettent de la renommer et de définir un style. Note implémente l'interface INotifyPropertyChanged qui permet de notifier la vue que la valeur d'une propriété dans la classe a changé.

Enfin, la classe **Style** définie un style utilisateur. Les différentes propriétés de la classe représentent les différents paramètres de mise en page à savoir : la taille de police, la police d'écriture, la couleur du texte, savoir si le texte doit être mis en gras, en italique, et/ou en souligné ainsi que l'alignement du texte. Ce dernier paramètre est donné par une classe d'énumération qui définit qu'un texte peut être aligné à droite, au centre, à gauche ou en justifié (le texte est aligné à gauche et à droite). Les alignements ne sont pas cumulables. Le but de mettre tous ces paramètres dans des propriétés est d'une part de bénéficier du système de notifications de ces dernières (pour informer la vue que leur valeur a changé, pour le binding). Nous plaçons ces paramètres dans le Modele car un style doit être sérialisé.

2. Diagramme de classe de UltraNotes

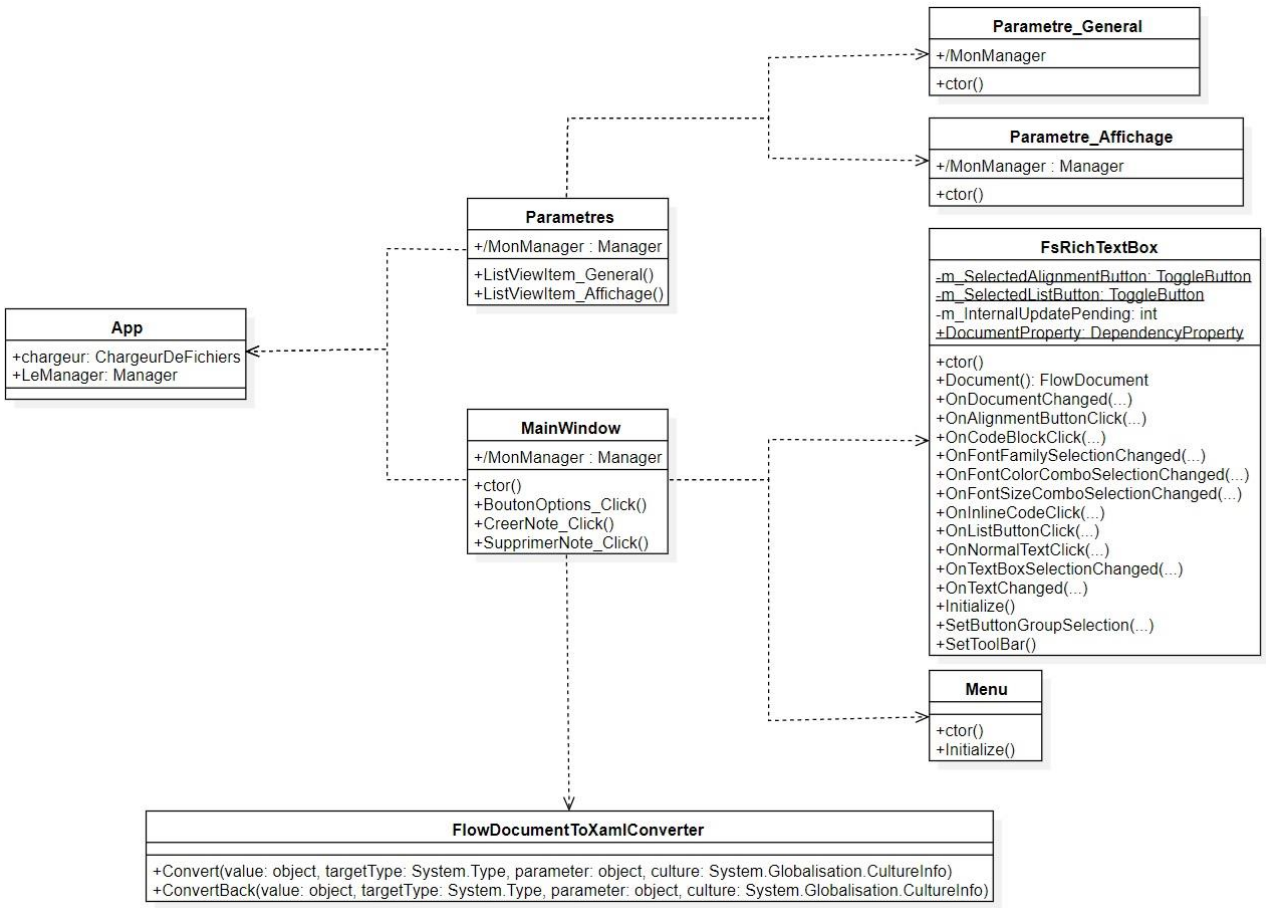

Figure 3 : Diagramme de classe du paquage UltraNotes

La paquage UltraNotes contient toutes les classes liées aux vues de notre application. Ces classes composent le « code behind », ce sont les classes et méthodes qui interagissent directement avec les éléments de la vue. Toutes ces classes sauf FlowDocumentToXamlConverter et App héritent de la classe Window (pour les Vues) et UserControl (pour les UserControls) du package PresentationFramework (Cf Figure 4 en fin de document).

La classe App est réservée ici pour instanciée des propriétés communes à chacune de classes de notre application. La propriété de type Manager nommée LeManager qui est instanciée dans la classe App est donc accessible partout ailleurs, autant dans les classes héritant de Window que de UserControls et autre. Pour cela, il faut déclarer une propriété de type Manager qui pointe vers le même espace mémoire que LeManager de App dans chacune de classes qui a besoin d'utiliser cette propriété. Ainsi dans notre diagramme, LeManager dans App est notre première instance du manager et les propriétés MonManager dans MainWindow, Paramètres et ces UserControls sont des propriétés ayant la même référence que LeManager, ce qui nous permet de travailler et manipuler cette dernière propriété ailleurs que dans App. Ce fonctionnement permet plus de visibilité dans le code. Nous ne sommes pas obligés de passer une instance de manager dans nos méthodes ou notre XAML pour avoir le même objet manager partout. Ici, toutes les instances de type Manager dans les classes de l'application portent le nom MonManager et pointe vers l'instance de LeManager de App. Ces propriétés sont toujours déclarées dans les premières lignes de la classe pour savoir tout de suite si celle-ci utilise l'objet manager.

Les classes MainWindow et Paramètres sont des classes plutôt simples dans leur fonctionnement. Elles contiennent chacune des méthodes gérant des évènements de la vue. La classe Paramètres contient des méthodes qui gèrent un UserControl qui change en fonction de l'onglet sur lequel l'utilisateur a cliqué. Ces

méthodes commencent par ListViewItem et se terminent par le nom de l'onglet en question. Les UserControls instanciées dans ces méthodes portent le nom Paramètre en référence à la fenêtre dans laquelle ils vont s'insérer suivi du nom de l'onglet choisi. Ce système de nommage est utilisé ici pour plus de clarté dans la lecture du code. Une personne extérieure de développement de l'application peut vite comprendre que les trois classes Paramètres, Parametre_Affichage et Parametre_General sont liées. La classe MainWindow se contente aussi de gérer les éléments de sa vue avec des méthodes. Ce sont donc des gestionnaires d'évènements. La classe MainWindow possède également une dépendance vers la classe FlowDocumentToXamlConverter, qui se charge de convertir un objet de type FlowDocument en un fichier de balisage XML représentant ce FlowDocument. Ce convertisseur est utile au bon fonctionnement de la FsRichTextBox qui s'en sert lors que l'on bind le contenu d'une note dans sa propriété Document. Encore, le UserControl Menu est inséré dans le XAML de la MainWindow ce qui explique la dépendance. Nous avons séparé le code du Menu avec celui de la fenêtre pour bien séparer les éléments de celle-ci et rendre les fichiers moins volumineux et plus simple à comprendre. En effet, chaque fichiers XAML existe pour gérer un élément : MainWindow pour le Master, Menu pour la barre des menus et FsRichTextBox pour le Detail.

Enfin, la classe FsRichTextBox est un UserControl contenant une RichTextBox sur laquelle on peut Binder des propriétés. Elle contient de nombreux gestionnaires d'évènement pour les UIElements qu'il contient. Elle possède également une Dependency Property pour la propriété Document, sur laquelle on à binder le contenu d'une note dans MainWindow. Ainsi, une Dependency Property permet de récupérer une valeur depuis la vue, ce pourquoi nous l'utilisons ici. Cette classe possède également des champs que l'on déclare privés car ils ne servent que dans cette classe. Ils n'ont donc pas besoin d'être accessible ailleurs que dans cette classe.

Ci-dessous un diagramme mettant en évidence les relations d'héritage et de dépendance entre UltraNotes et PresentationFramework.

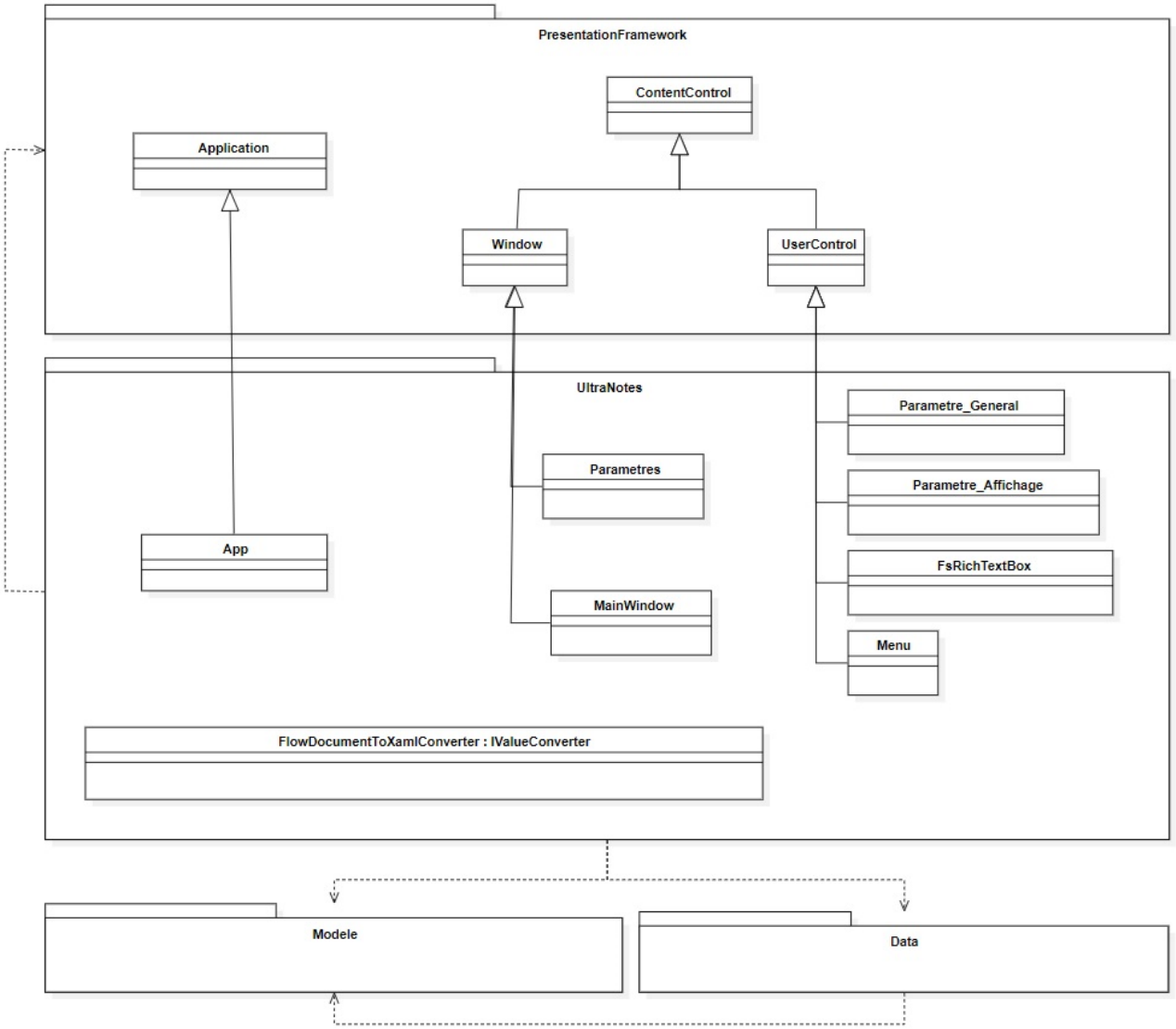

Figure 4 : Mise en évidence des relations d'héritage et de dépendance du package UltraNotes avec PresentationFramework

Exemple de diagramme de séquence

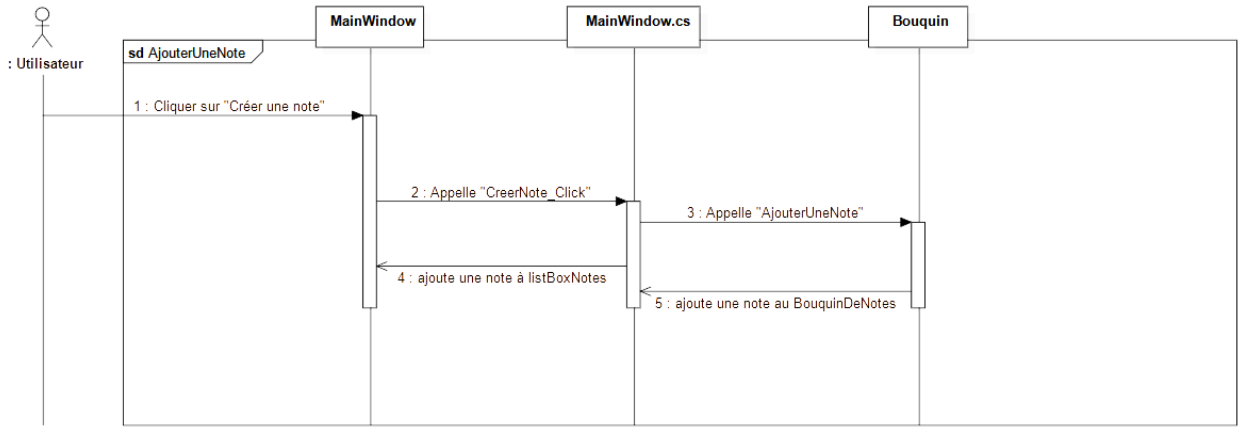

Figure 5 : Diagramme de séquence de la fonctionnalité « Ajouter une note »

Ce diagramme de séquence porte sur l'**ajout d'une note**. Tout d'abord l'utilisateur clique sur le bouton intitulé « Créer une note » sur la vue MainWindow.xaml de notre application wpf. Ensuite la propriété « Click » du bouton appelle la méthode « CreerNote_Click() » du code behind MainWindow.xaml.cs qui va créer une nouvelle note et l'ajouter au BouquinDeNotes grâce à la méthode « AjouterUneNote() » de la classe Bouquin. Si la note n'est pas nulle et n'est pas déja présente dans le BouquinDeNotes alors elle ajoutée. Enfin, étant ajouté dans BouquinDeNotes la note est aussi ajoutée dans la ListBox listBoxNotes de MainWindow.xaml.