Document Preuves

Thomas DEVIENNE (groupe 4H)
Matthias BIZET (groupe 3H)

Conception et programmation orienté objet (C#, .NET)

Je maîtrise les bases de la programmation C# (classes, structures, instances, …)

Chaque classe représente un objet. Par exemple, notre classe Bouquin représente une collection de notes. La classe Note représente une notre.

```
namespace Modele
{
    [DataContract]
    public class Note : INotifyPropertyChanged
    {
        Champs

        Propriétés

        INotifyPropertyChanged Members

        Constructeurs

        Méthodes publiques

        Méthodes redéfinies
    }
}
```

Figure 1 : la classe Note

On remarque ici que notre classe est dans le namespace Modele. Un namespace contient un ensemble de classes. Notre classe contient des instances d'autres classes, certaines build-in comme string, d'autres utilisateurs comme TypeDocument. Type Document est une classe de type énumération qui contient un ensemble de valeurs.

```
        #region Champs

        // Déclaration des champs de la classe;
        // variable de la propriété Nom;
        private string p_Nom;
        // variable de la propriété DocumentXaml;
        private string p_DocumentXaml;

        #endregion

        #region Propriétés

        /// <summary>
        /// Propriété qui est chargée d'envoyer des notifications à la vue pour notifier le changement d'une propriété
        /// </summary>
        void OnPropertyChanged(string nomPropriete) => PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nomPropriete));
        /// <summary>
        /// Liste des styles utilisateurs
        /// </summary>
        public IList<Style> StylesUtilisateur { get; } = new List<Style>();
        /// <summary>
        /// Propriété Nom qui représente le nom de la Note (son titre)
        /// </summary>
        [DataMember]
        public string Nom
        {
            get { return p_Nom; }
```

Figure 2 : corp de la classe Note (extrait)

La partie statique de l'instance (à gauche de l'égal) est toujours de type le plus haut. C'est pour cela que nous avons une propriété de type IList<> qui instancie un objet de type List<>. Les propriétés en C# sont des objets permettant de contenir des valeurs dont l'assignation et la récupération sont contrôlées.
Nous n'avons pas de structures dans notre projet. Une structure est un type valeur, c'est-à-dire stocké sur la pile. Elle s'écrit comme une classe, peut implémenter des interfaces mais ne peux pas faire d'héritage. Il est préférable d'écrire ces structures immuables. En effet, comme tous les types valeur qui sont immuables, on attend de la structure le même comportement.

Je sais utiliser l'abstraction à bon escient (héritage, interfaces, polymorphisme, …)

Nous utilisons des interfaces pour implémenter un système de notification de la vue sur nos propriétés dans certaines classes. C'est le cas de la classe Note pour les propriétés Nom et DocumentXaml (Cf *figure 1*). L'héritage s'écrit de la même manière dans le code. Une classe qui hérite d'une autre peux récupérer ces méthodes, les utiliser et les redéfinir, ainsi que ces champs.

```
namespace Modele
{
    public class Style : INotifyPropertyChanged
    {
        Champs privés

        Propriétés

        #region INotifyPropertyChanged Members

        public event PropertyChangedEventHandler PropertyChanged;

        #endregion

        Constructeurs

        Méthodes redéfinies
    }
}
```

Figure 3 : exemple d'implémentation d'interface

Je sais gérer des collections simples (tableaux listes, …)

Nous utilisons des collections simples dans la classe Note ligne 32 (Cf *figure 2*).
Également, dans la classe Bouquin la méthode AjouterListeDeNotes() prends en paramètres un tableau de notes que nous parcourons dans la méthode.

```
public int AjouterUneListeDeNotes(Note[] notes)
{
    int code_err = 0; // 0 = pas d'erreur;
    bool gravite_err = true; // true = erreur critique; false = erreur mineure;
    bool temoin; // récupère le code retour de AjouterUnFichier();
    // on ajoute les notes une par une;
    foreach (var elm in notes)
    {
        // si une note existe déja, elle sera ignorée;
        temoin = AjouterUneNote(elm);
        // si temoin = true, un fichier à été ignoré;
        if (temoin) code_err = 2; else gravite_err = false;
    }
    // si gravite_err est passé à true, alors cela signifie que toutes les opérations ont été refusées, alors on passe code_err
    // à 1 (erreur critique);
    code_err = gravite_err is true ? 1 : code_err;
    // on retourne le résultat de l'opération (0 si tout s'est bien passé, 1 si erreur critique (tout les fichiers ignorés)
    // et 2 si erreur mineure (certains fichiers ignorés));
    return code_err;
}
```

Figure 4 : utilisation d'un tableau

Je sais gérer des collections avancées (dictionnaires)

Nous utilisons un objet de type dictionnaire dans Bouquin pour associer le nom d'une couleur à son équivalent en hexadécimal.

```
public Dictionary<string, string> ThemeApplicationCouleurs { get; set; } = new Dictionary<string, string>()
{
    {"Défaut", "#64BED8" },
    {"Vert", "#59F0A2" },
    {"Orange", "#E6A05A" },
    {"Bleu profond", "#3A9981" },
    {"Bleu nuit", "#428B99" },
    {"Violet", "#845399" },
    {"Violet profond", "#995489" },
    {"Marron", "#995F3D" },
    {"Contraste", "#252C2E" }
};
```

Figure 5 : exemple de création de dictionnaire

Nous utilisons ce dictionnaire à la ligne 51 de Parametre_Affichage.xaml.cs.

Je sais contrôler l'encapsulation au sein de mon application

Dans notre application, nous initialisons des certaines variables et méthodes publiques, d'autres privées. En général, nous déclarons privé (private) tous les objets que nous utilisons dans la classe où ils ont été déclarés. Nous déclarons publique (public) tous les objets qui doivent être appelés en dehors du package. Ceux qui ne sont pas précédés de mots clés sont dans les classes package private, c'est-à-dire accessibles uniquement dans le package actuel (le namespace en l'occurrence).

Je sais tester mon application

Les tests unitaires sont des tests destinés à vérifier que les méthodes de nos classes ont bien le comportement attendu. Pour nos tests unitaires, nous utilisons Xunit.

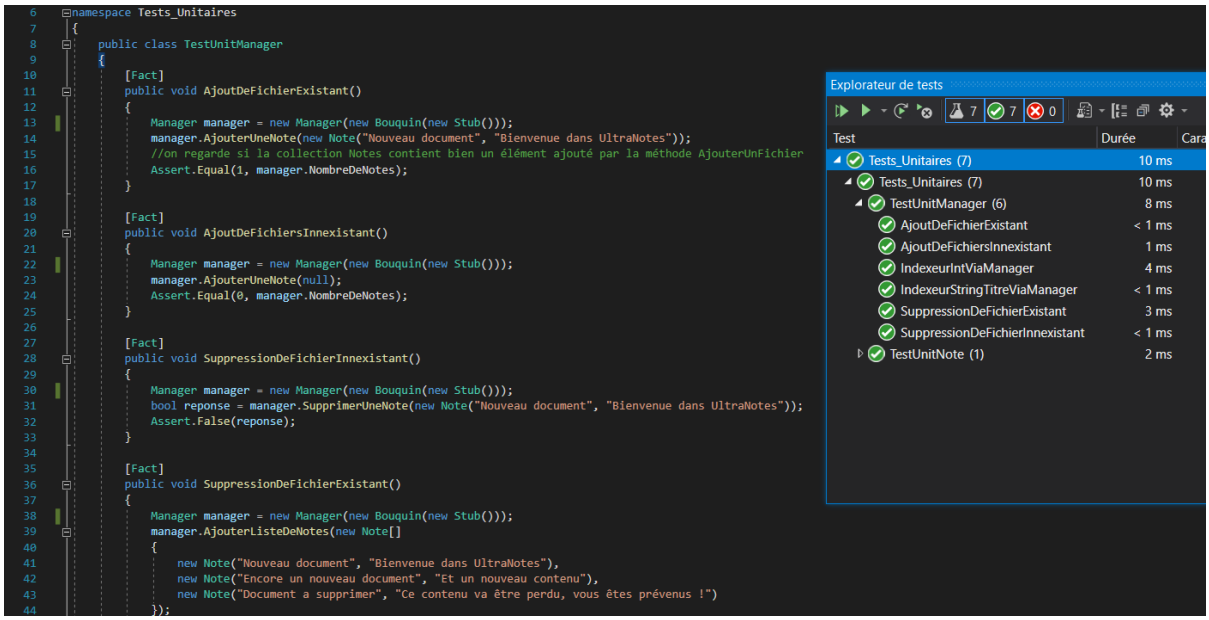

Figure 6 : exécution des tests unitaires pour le Manager

Les tests fonctionnels eux servent à tester les fonctionnalités de l'application. On observe le résultat des tests dans une application console.

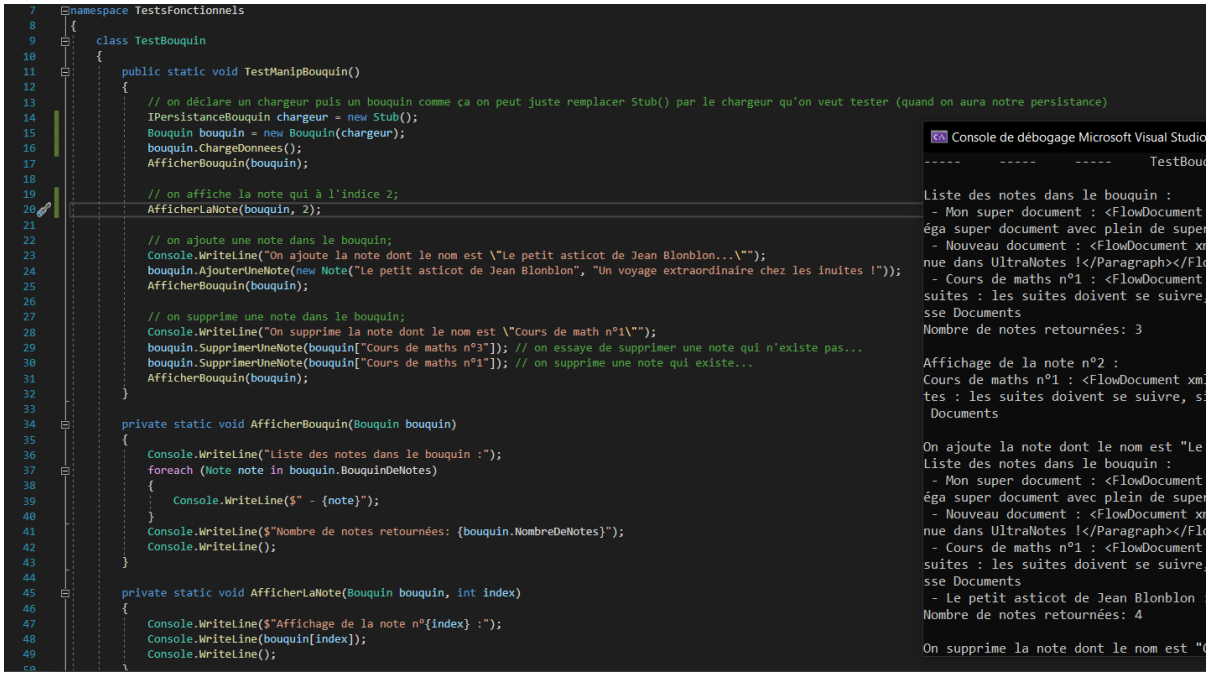

Figure 7 : exécution des tests fonctionnels pour l'ajout et la suppression de fichiers dans un bouquin

Je sais utiliser LINQ

Nous utilisons LINQ dans MainWindow.xaml.cs à la ligne 98 pour récupérer le nombre d'occurrences d'une chaine de caractères dans une liste.

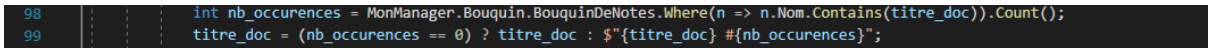

Figure 8 : utilisation de LINQ

Je sais gérer les évènements

Nous utilisons les évènements en implémentant l'interface INotifyPropertyChanged dans Style.

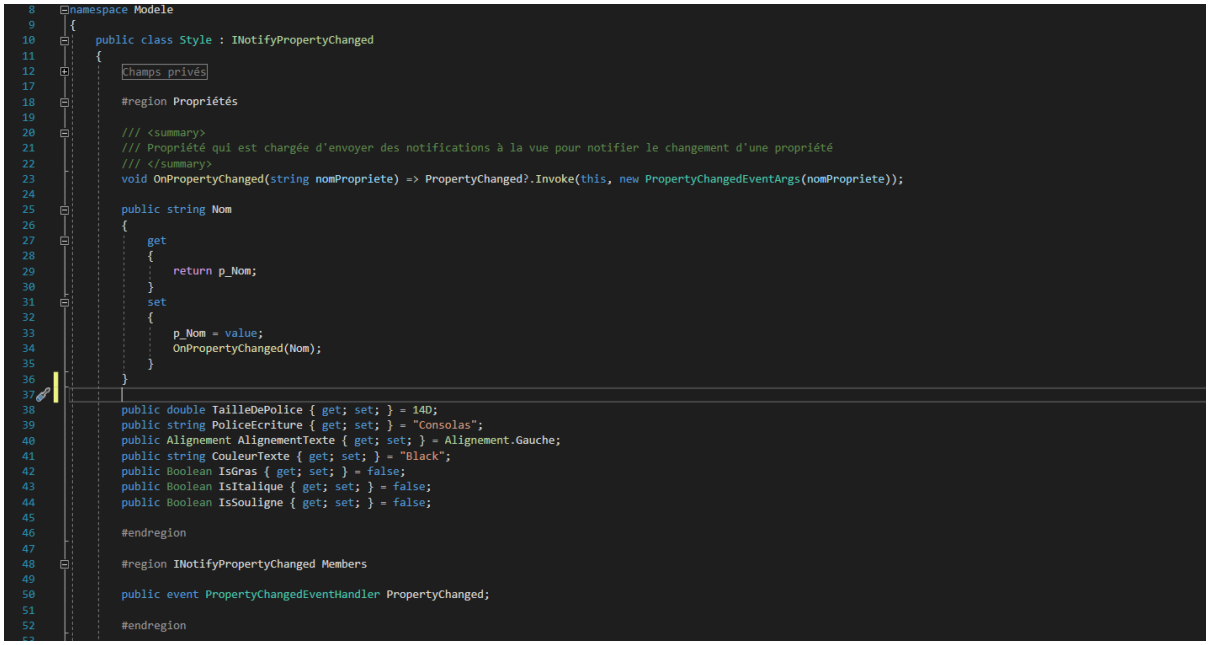

Figure 9 : gestion des évènements

On utilise la méthode Invoke() dans la propriété en lecture seule OnPropertyChanged pour déclencher la notification.

Interface Homme-Machine (XAML, WPF)

Je sais choisir mes layouts à bon escient

Nous utilisons les Dock Panel pour organiser nos IUElements dans la vue de manière classique (toutes les fenêtres). Les Grids nous servent à organiser nos layouts de manière que leur disposition dans l'espace s'adapte à la taille de la fenêtre (89 dans MainWindow pour barre de menu de l'éditeur). Les Wrap Panels nous servent à aligner un ensemble d'éléments dans la vue et faire en sorte qu'ils s'adaptent à largueur de la fenêtre. Ainsi, ils sont tous directement visibles sans avoir à utiliser une scroll bar (l.53 dans FsRichTextBox pour barre d'outils). Nous nous servons des Stack Panel comme conteneur (l.41 FenetreStyle)

Je sais choisir mes composants à bon escient

Nous choisissons des composants différents pour chaque bouton de notre barre d'outils :

- Des boutons qui supportent un click pour une actions spécifique
- Des toggles boutons qui restent activés lorsqu'ils sont activés, parfais pour les effets de gras, italique, soulignés et les alignements

- Des combobox pour permettre à l'utilisateur de choisir entre plusieurs options

```
154                 <Image Source="../icons/texteditor/puces.png" Margin="3,3"/>
155             </ToggleButton>
156             <!-- Insérer une puce numérotée -->
157             <ToggleButton x:Name="NumberingButton" Command="EditingCommands.ToggleNumbering"
158                   CommandTarget="{Binding ElementName=TextBox}"
159                   Background="{x:Null}" BorderBrush="{x:Null}" Height="27"
160                   Template="{StaticResource FlatToggleButtonControlTemplate}"
161                   Cursor="Hand" ToolTip="Puce numérotée (Ctrl+Shift+N)"
162                   Click="OnListButtonClick">
163                 <Image Source="../icons/texteditor/pucesnum.png" Margin="3,3"/>
164             </ToggleButton>
165             <!-- Changer la couleur du texte -->
166             <xctk:ColorPicker Name="FontColorCombo" SelectedColorChanged="OnFontColorComboSelectionChanged" Margin="5,0" Width="40" Height="23" ></xctk:ColorPicker>
167             <!-- Police d'écriture  -->
168             <ComboBox x:Name="FontFamilyCombo" Background="White" BorderBrush="{x:Null}" Cursor="Arrow" Margin="5,0" Height="23"
169                       ToolTip="Police d'écriture" IsEditable="True" SelectionChanged="OnFontFamilyComboSelectionChanged">
170                 <ComboBox.ItemsPanel>
171                     <ItemsPanelTemplate>
172                         <VirtualizingStackPanel Width="250" />
173                     </ItemsPanelTemplate>
174                 </ComboBox.ItemsPanel>
175                 <ComboBox.ItemTemplate>
176                     <DataTemplate>
177                         <TextBlock Text="{Binding}" FontFamily="{Binding}" FontSize="15" Height="20"/>
178                     </DataTemplate>
179                 </ComboBox.ItemTemplate>
180             </ComboBox>
181             <!-- Taille de police -->
182             <ComboBox x:Name="FontSizeCombo" Background="White" BorderBrush="{x:Null}" Cursor="Arrow" Margin="5,0" Height="23" Padding="3"
183                       ToolTip="Taille de police" IsEditable="True" SelectionChanged="OnFontSizeComboSelectionChanged" >
184             </ComboBox>
185             <!-- Styles utilisateur -->
186             <ComboBox x:Name="StyleCombo" Background="White" BorderBrush="{x:Null}" Cursor="Arrow" Margin="5,0" Height="23"
187                       ToolTip="Style utilisateur" IsEditable="True" SelectionChanged="OnStyleComboSelectionChanged">
188                 <ComboBox.ItemTemplate>
189                     <DataTemplate>
190                         <TextBlock Text="{Binding Nom}" FontFamily="{Binding PoliceEcriture}" FontSize="{Binding TailleDePolice}" Foreground="{Binding CouleurTexte}"/>
191                     </DataTemplate>
192                 </ComboBox.ItemTemplate>
193             </ComboBox>
194         </WrapPanel>
195         <RichTextBox x:Name="TextBox" TextOptions.TextFormattingMode="Ideal" TextOptions.TextRenderingMode="Aliased"
```

Figure 10 : Choix des composants

Nous utilisons les balises Images pour insérer des images fixées dans les fenêtres. Pour l'éditeur, nous avons choisi une RichTextBox car elle permet d'éditer des fichiers RTF (Rich Text Format), reconnus par Word, WordPad et d'autres éditeurs de texte. Ce type de fichiers permet notamment la mise en forme et en page du texte et l'ajout d'images (*figure 10*). Les TextBox nous permettent d'éditer des textes simples ne nécessitant pas de mise en page, comme le titre du document (l .101 MainWindow) et les TextBlock d'afficher simplement du texte sans donner la possibilité à l'utilisateur de le modifier (*figure 10*).

Également, nous avons importé via Nugget un kit d'outils WPF permettant d'importer dans notre projet des composants amélioré. Parmi ceux-ci, nous avons utilisés un color pickler permettant de choisir une couleur spécifique pour le texte (l.166).

Je sais créer mes propres composants

Pour ce projet, nous avons crées une FsRichTextBox qui permet de pouvoir se binder sur une propriété (l.153 dans MainWindow). Pour cela, nous avons créé une DependencyProperty dans le code behind de notre composant pour récupérer la valeur que nous lui passions dans ces paramètres.

```
152             <!-- La RichTextBox -->
153             <fsc:FsRichTextBox x:Name="EditBox" Document="{Binding DocumentXaml, Converter={StaticResource flowDocumentConverter}, Mode=TwoWay}"
154                                Tag="{Binding DocumentXaml, Converter={StaticResource flowDocumentConverter}, Mode=TwoWay}"
155                                Grid.Row="0" Margin="10,10,10,5" />
```

Figure 11 : La FsRichTextBox

Cette dépendance de propriétés récupère les valeurs passées dans la propriété Document et s'en sert dans le code behind de notre composant pour effectuer diverses manipulations.

```
#region Dependency Property Declarations

// Document property
public static readonly DependencyProperty DocumentProperty =
    DependencyProperty.Register("Document", typeof(FlowDocument),
    typeof(FsRichTextBox), new PropertyMetadata(OnDocumentChanged));

#endregion

Constructeur

#region Propriétés

/// <summary>
/// FlowDocument WPF contenu dans le controle.
/// </summary>
public FlowDocument Document
{
    get { return (FlowDocument)GetValue(DocumentProperty); }
    set { SetValue(DocumentProperty, value); }
}
```

Figure 12 : Dependency Property de la FsRichTextBox

Je sais personnaliser mon application en utilisant des ressources et des styles.

Nous avons utilisé des styles pour les Boutons (l.14 FsRichTextBox).

Je sais utiliser les DataTemplates

Idem qu'au-dessus, à partie de ligne 14.

Je sais intercepter les évènements de la vue

On utilise des gestionnaires d'évènements pour intercepter les événements de la vue (les méthodes concernées finissent par _Click ou _SelectionChanged). Exemple l.118 dans FsRichTextBox.xaml.cs.

Je sais notifier la vue depuis des événements métier

On utilise l'interface INotifyPropertyChanged dans Note.cs dans le Modele, sur lequel on utilise un évent.

```
namespace Modele
{
    [DataContract]
    public class Note : INotifyPropertyChanged
    {
        Champs

        Propriétés

        #region INotifyPropertyChanged Members

        public event PropertyChangedEventHandler PropertyChanged;

        #endregion
```

Je sais gérer le DataBinding sur mon master

l.55 MainWindow pour databing
Constructeur MainWindow pour datacontext

Je sais gérer mon DataBing sur mon détail

l.195 FsRichTextBox pour databinding
Constructeur FsRichTextBox pour datacontext

Je sais gérer Dependency property

Voir nos dépendency property dans la FsRichTextBox.xaml.cs et MainWindow (lorsqu'on utilise le composant).

Je sais développer un master/detail

l.55 MainWindow pour master
l.195 FsRichTextBox pour detail
Voir toutes les méthodes abonnées

Je sais coder une fonctionnalité qui m'est propre

FenetreStyle.xaml.cs et toutes les méthodes associées (non prévus au départs, décidé vers mai).