Visual Studio 2012 & Langage C#

ED72/ED73 – Développer les classes des couches Présentation et Service

Thèmes abordés:

- **Séparer l'IHM en classes de Présentation et classes de Service**
- **Structurer une application en couches**

1. Séparer l'IHM en classes de Présentation et classes de Service

Dans une application Winform 'classique', les forms assurent finalement 2 fonctions : la *présentation* des données (affichage et saisie contrôlée) et le *traitement sur ces données* (actions de prise en compte de ces données, enchaînements entre écrans voire mémorisation en base de données).

L'idée est donc ici de limiter les Winforms à leur rôle de *présentation* et d'isoler la 'logique applicative' dans d'autres classes qui seront en charge de gérer les données saisies, les événements et les enchaînements d'écrans.
On parle de classes '**Contrôleur'** en ce sens qu'elles **contrôlent le déroulement des opérations** : *elles instancient les forms nécessaires au dialogue et réagissent aux événements survenus sur ces forms*.

L'intérêt principal de ce découplage Présentation/Contrôleur est de *favoriser la réutilisation* des forms dans des contextes différents (le même form peut être utilisé dans différents enchaînements d'écrans et pour des fonctionnalités différentes – consultation ou modification, actions variables…).
De plus, comme le Contrôleur assure la logique applicative, il peut être adapté à peu de frais à des présentations différentes (la même logique d'une application se retrouve dans des écrans Windows ou des pages Web, seuls la présentation et les enchaînements varient).
Ce découplage aura bien entendu un coût en complexité de développement.

L'objet de cet exercice est de transformer une application Winform 'classique' en architecture Présentation/Contrôleur.

2. Structurer une application selon le 'design pattern' MVC

La conception et le développement orientés objet sont des techniques relativement récentes ; malgré tout, l'expérience est déjà suffisante pour dégager un certain nombre de '*bonnes pratiques'*. Certains chercheurs ont ainsi établi un recueil de cas typiques et déterminé pour chacun une 'bonne' architecture : on appelle ces modèles des *'design patterns'*.

La logique générale de ces bonnes pratiques est de *favoriser les associations entre classes plutôt que les héritages et de jouer sur des types d'interfaces plutôt que les classes elles-mêmes*. Et tout cela au prix d'une *complexité* certaine, au service de la *réutilisabilité des composants* et de la *facilité de maintenance de l'application*.

Il ne s'agit pas ici d'étudier les *design patterns* mais de mettre en œuvre une architecture applicative encore simple s'inspirant des *design pattern* 'MVC' et 'x-tiers'.
Selon MVC, l'application est divisée en 3 'couches', le 'Modèle' des données, représenté dans un premier temps par les classes Métier et leurs collections, la 'Vue', concrétisée par les classes Winforms, et le 'Contrôleur', assuré par ces classes de la couche Service qui nous restent à écrire.

3. Application au cas de gestion des stagiaires.

Choix d'implémentation pour les classes Présentation (Winform)

Pour cette application, il est fait ici un certain nombre de choix techniques d'implémentation :

a) chaque classe de Présentation Winform est instanciée par une classe Contrôleur ; elle reçoit la référence de l'objet Métier à afficher/mettre à jour (un Winform ne 'sait pas' d'où proviennent les données à afficher) ***;***

b) ***chaque classe de Présentation Winform prend en charge essentiellement les affichages, les contrôles de saisie et la mise à jour des données Métier ; en général, il n'est pas associé de procédure événementielle aux boutons de commande*** mais les commandes 'd'intérêt local' de type fermer le form peuvent rester implémentées dans le Winform ***;***

c) ***les Winforms exposent aux Contrôleurs certains contrôles graphiques de commande*** *(boutons…)* ***afin que les Contrôleurs puissent y associer des procédures événementielles de traitement*** (un Winform ne 'sait pas' ce qui sera fait des données qu'il permet de saisir ou de choisir) ***;***

d) ***un Winform expose aux Contrôleurs certaines propriétés*** permettant au Contrôleur de savoir ce qui s'est passé pendant l'affichage/la saisie des données, et ***des services sous forme de méthodes permettant de traiter les objets Métier*** en fonction des données saisies dans les ***contrôles graphiques qui restent privés au Winform ;*** à fonctionnalités identiques, un Winform doit pouvoir évoluer (Textbox ➔ ListBox par exemple) sans remettre en cause la logique des traitements ;

e) ***chaque classe de Présentation est définie indépendamment de son usage par un Contrôleur particulier*** (et reste donc réutilisable pour une même présentation mais une autre logique de traitement) ;

f) ***les Winforms de création de données instancient eux-mêmes les nouveaux objets Métier correspondants*** et exposent des propriétés permettant aux Contrôleurs d'y accéder (bien souvent, il s'agit d'un objet Métier spécialisé que seul le form saura instancier en fonction des données saisies dans ses contrôles graphiques) ;

g) ***le form principal d'une application MDI n'a pas besoin de Contrôleur et il reste instancié par la classe de lancement de l'application*** (`Program.cs`) ; il gère toute la logique d'instanciation des Contrôleurs associés aux différentes fonctions offertes par ses menus ;

En bref, un Winform n'est pas « dénué d'intelligence », mais il ne prend aucune initiative.

Choix d'implémentation pour les classes Service

Pour cette application, il est fait ici un certain nombre de choix techniques d'implémentation :

a) ***chaque classe Contrôleur correspond à un cas d'utilisation élémentaire*** ; elle gère la logique applicative associée à ce '*use case*' ***en assurant les traitements sur les données saisies/mises à jour ainsi que les enchaînements d'écrans*** nécessaires (bien souvent, un élémentaire se limite à un seul écran Winform) ;

b) ***chaque classe Contrôleur instancie le form*** initial associé au scénario nominal du use case et ***l'enrichit des événements prévus dans le scénario afin de déclencher ses propres procédures événementielles en réponse aux sollicitations*** de l'utilisateur sur le Winform ;

c) ***les classes Contrôleur assurent la gestion des objets Métier et des collections d'objets Métier*** (recherches et fourniture aux classes de Présentation, gestion des contenus de collections, gestion de la Persistance des données) ;

d) ***chaque classe Contrôleur communique avec les Winforms en leur passant éventuellement des paramètres lors de leur instanciation*** (de préférence, des références aux objets Métier à afficher/mettre à jour) et ***en invoquant certaines méthodes*** exposées par les Winforms (contrôles de saisie, réaffichage, mise à jour effective des objets Métier par exemple) ;
e) ***une classe Contrôleur peut instancier le Contrôleur associé au use case élémentaire*** suivant/inclus en lui passant éventuellement les paramètres nécessaires ;
f) ***le 'constructeur' du Contrôleur réalise donc le scénario principal/nominal du use case ;*** *les scenarii alternatifs sont assurés par d'autres méthodes du Contrôleur correspondant à des procédures événementielles liées aux forms mais implémentées dans le Contrôleur lui-même ;*
g) la ***couche Service*** est essentiellement assurée par les classes ***Contrôleurs***, complétées de ***classes 'Outils' d'intérêt général*** (exposant habituellement des méthodes et propriétés sous forme 'static' -comme `Math.Sqrt()` ou `Outils.EstEntier()`-) ;

En bref, un Contrôleur pilote le déroulement des opérations qui concernent son rôle, en enchaînant des appels de méthodes exécutées par les classes des autres couches.

Notre mini-application permet de gérer les stagiaires d'un centre de formation ; pour chacun on souhaite gérer des données générales d'identité et coordonnées ainsi que des informations de suivi des travaux. Les stagiaires sont regroupés en sections possédant un identifiant, un libellé et des dates de début et fin de formation. On peut considérer que la relation entre un stagiaire et une section est une 'composition' : une section est composée de stagiaires et un stagiaire ne peut exister sans sa section. En conséquence, une section gère elle-même la collection privée des références aux objets `MStagiaire` qui lui sont associés. Les différentes sections de stagiaires sont cataloguées dans une collection générale gérée par une classe `MSections`.

Dans cette mini-application, on pourra dès le départ accéder aux différentes sections, puis d'une section choisie, à ses stagiaires, mais on ne permet pas la recherche directe d'un stagiaire ; en conséquence seule la collection générale des sections sera nécessaire (chaque section contenant la collection de ses propres stagiaires), et un stagiaire n'a pas besoin de référencer sa section (pas d'implémentation de la relation réciproque).

Pour cet exercice, les données restent volatiles en mémoire ; la gestion de la '*persistance'* sera étudiée dans un exercice ultérieur.

Trois use case sont identifiés : 'afficher la liste des stagiaires d'une section', 'ajouter un stagiaire dans une section' et 'consulter/modifier un stagiaire particulier' ; *il s'agira donc de développer 3 classes Contrôleur.*

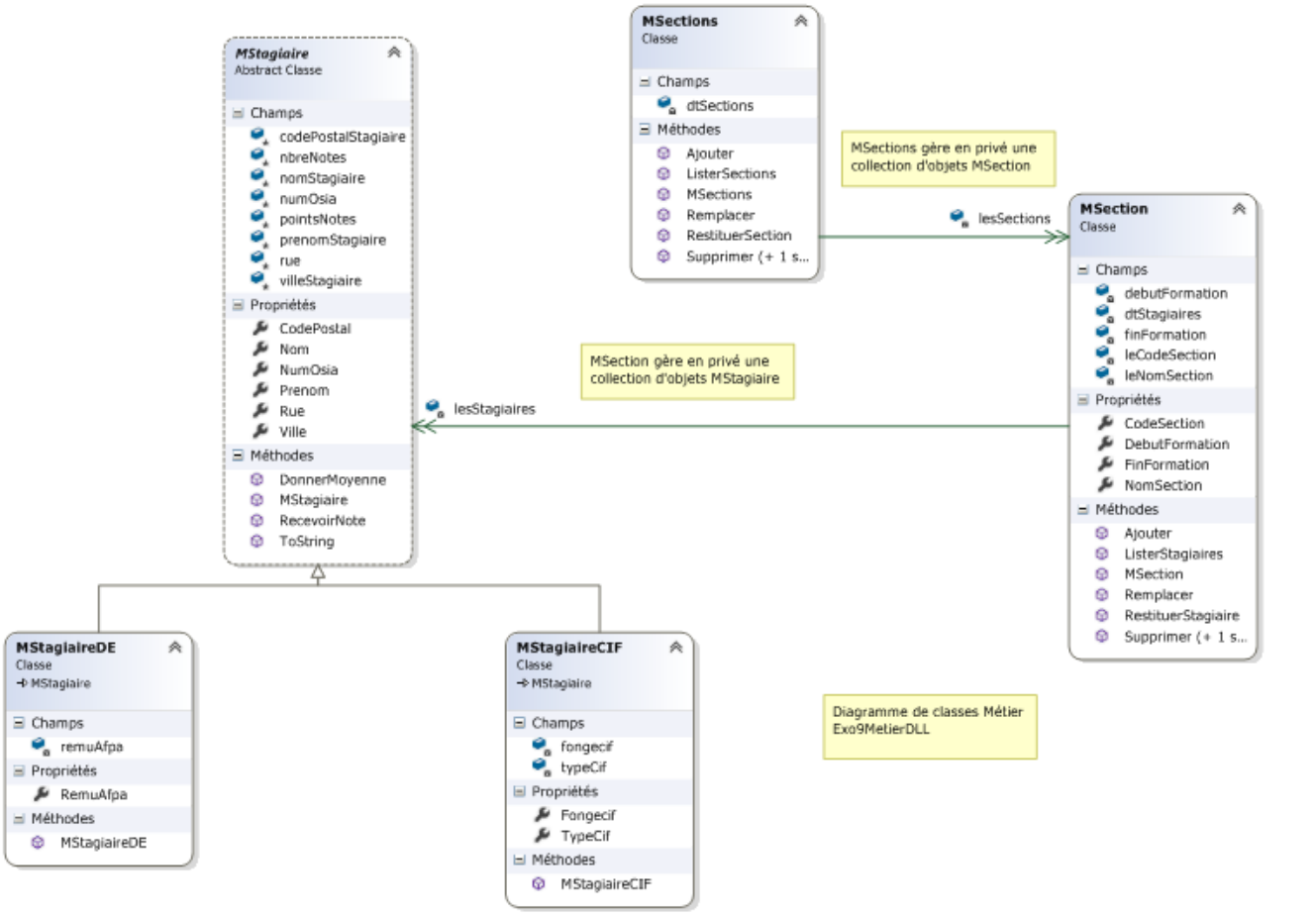
L'application fournie *(Exo9Métier)* ***est opérationnelle*** *(vous pouvez la faire tourner à tout moment à travers Visual Studio ou bien lancer sa version compilée `Exo9.exe` en dehors de Visual Studio)* ***et ne nécessite aucune adaptation de sa logique applicative ; il s'agit ici 'simplement' de restructurer cette application construite en 2 couches (classes Métier et Winform) vers la même application construite en 3 couches 'MVC'.***

Pour illustrer le fonctionnement interne de l'application finale, vous pouvez étudier à tout moment l'animation 'Exo9-métier-vue-controleur.ppsx'.

4. Première étape : développer le Contrôleur du use case 'afficher la liste des stagiaires d'une section'.

Ouvrir le projet `Exo9MetierDLL` et observer le diagramme de classes fourni. Il représente uniquement les classes Métier :

- la classe des stagiaires, `MStagiaire`, est spécialisée en 2 classes `MStagiaireDE` et `MStagiaireCIF` selon le statut des personnes ;
- la classe `Donnes` expose en `static` un objet `MSections` représentant la collection des sections de stagiaires ;
- chaque section est représentée par un objet `MSection` ;



Le form MDI de démarrage, opérationnel, `frmMDI.cs`, permet d'afficher le form `frmExo9.cs` ; celui-ci affiche la liste des stagiaires d'une section et permet d'en supprimer ou bien d'enchaîner sur les use case '*ajouter un stagiaire dans une section'* et '*consulter/modifier un stagiaire particulier*'.

Il s'agit donc de développer le Contrôleur '*afficher la liste des stagiaires d'une section'* qui doit assurer :

- instancier le form `frmExo9.cs` ;
- l'enrichir afin de pouvoir exécuter le traitement de suppression d'un stagiaire choisi par l'utilisateur dans le `DataGridView` ;

- l'enrichir afin de pouvoir instancier le Contrôleur du use case '*ajouter un stagiaire dans une section*' si l'utilisateur clique le bouton *Ajouter* ;
- l'enrichir afin de pouvoir instancier le Contrôleur du use case '*consulter/modifier un stagiaire particulier*' si l'utilisateur double-clique sur un stagiaire affiché dans le `DataGridView` ;

Opérations à réaliser sur le projet fourni :

- Ajouter au projet **`Exo9`** une nouvelle classe `CtrlListerStagiairesSection.cs` ; préciser sa visibilité `public` ;
- Instancier un objet `CtrlListerStagiairesSection` dans le form de démarrage de l'application pour l'option de menu *Fichier/Ouvrir/Section/CDI* : modifier `frmMDI.cs` :

```
private void openCDIToolStripMenuItem_Click(object sender, EventArgs e)
{
    // instancier le Contrôleur lister les stagiaires d'une section
    CtrlListerStagiairesSection ctrl = new
CtrlListerStagiairesSection();
}
```

- Faites de même pour le code associé à l'outil `Ouvrir` :

```
private void OpenFile(object sender, EventArgs e)
{
    // instancier le Contrôleur lister les stagiaires d'une section
     CtrlListerStagiairesSection ctrl = new
CtrlListerStagiairesSection();
}
```

- Ecrire le constructeur de la classe `CtrlListerStagiairesSection` ; il s'agit essentiellement d'instancier un objet `frmExo9` en lui passant la référence de la section à gérer et d'implémenter les événements double-clic du `DataGridView`, clic du bouton *Ajouter* et clic du bouton *Supprimer*. En effet, le form conservera les fonctions d'affichage mais ne sera plus en charge lui-même d'ajouter ou supprimer un stagiaire ou encore enchaîner sur un autre écran car c'est maintenant le rôle du Contrôleur (noter que pour pouvoir implémenter les événements, ces contrôles graphiques (boutons, `DataGridView`) ne doivent plus être privés au form `frmExo9` mais de visibilité `internal` (ou `public`) pour être maintenant accessibles par d'autres composants du projet comme les Contrôleurs) :

```
/// <summary>
/// constructeur : instancie et personnalise le form frmExo9 et
l'affiche en non modal
/// </summary>
public CtrlListerStagiairesSection()
{
    // pour commencer : initialisation de quelques données
...// (la section aurait du etre recue en parametre)…
    this.init();
    // instancier le form initial
    this.leForm = new frmExo9(this.laSection);
    // le renseigner
    this.leForm.AfficheStagiaires(this.laSection);
    // implémenter l'événement bouton ajouter clic
    this.leForm.btnAjouter.Click += new EventHandler(btnAjouter_Click);
    // implémenter l'événement bouton supprimer clic
    this.leForm.btnSupprimer.Click += new
EventHandler(btnSupprimer_Click);
```

```
    // implémenter l'événement double-clic en datagridview
   this.leForm.grdStagiaires.DoubleClick += new
EventHandler(grdStagiaires_DoubleClick);
    // afficher le form
    this.leForm.MdiParent = Donnees.FrmMDI;
    this.leForm.Show();
}
```

- Ecrire la procédure d'initialisation qui crée 'en dur' une section et un stagiaire dans la section :

```
/// <summary>
/// pour commencer  : instanciation d'une section et d'un stagiaire
/// </summary>
private void init()
{
    // initialisation de la collection de sections
    Donnees.Sections = new MSections();
    // pour commencer, une seule section référencée "en dur" dans ce
programme
    // instancie la section
    this.laSection = new MSection("CDI1", "Concepteur Développeur
Informatique 2012");
    // l'ajoute dans la collection des sections gérée par la classe de
collection
    Donnees.Sections.Ajouter(this.laSection);
    // ajoute en dur un stagiaire à cette section
    MStagiaire unStagiaire;
    unStagiaire = new MStagiaireDE(11111, "DUPOND", "Albert", "12 rue
des Fleurs", "NICE", "06300",false);
    this.laSection.Ajouter(unStagiaire);
}
```

- Ajouter les **membres privés de niveau classe** nécessaires au bon déroulement de ce Contrôleur : `laSection` de type `MSection` et `leForm` de type `frmExo9`.
- Il reste maintenant à 'déporter' les traitements de suppression, ajout et affichage du détail d'un stagiaire depuis `frmExo9` vers son Contrôleur `CtrlListerStagiairesSection`:
  - supprimer les 'handler' d'événements maintenant inutiles sur `frmExo9` (sans supprimer le code associé car il faudra le déplacer vers le Contrôleur - le mieux est de neutraliser les lignes correspondantes dans le fichier de code `frmExo9.designer.cs`-) ; la classe `frmExo9` ne contiendra plus que les membres :

```
public partial class frmExo9 : Form
{
    /// <summary>
    /// la section de stagiaires gérée par ce form
    /// </summary>
    private MSection laSection;

    /// <summary>
    /// Constructeur
    /// </summary>
    public frmExo9(MSection uneSection)
    { …… }

    /// <summary>
    /// rétablit la source de données de la dataGridView
```

```
    /// et rafraîchit son affichage (public car appelé par le
ctrleur)
    /// </summary>
    /// <param name="laSection">la section dont il faut lister les
stagiaires</param>
    public void AfficheStagiaires(MSection laSection)
    { …… }

/// <summary>
/// bouton fermer : fermer le form (action locale)
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnFermer_Click(object sender, EventArgs e)
    { …… }
}
```

On y ajoutera la méthode événementielle suivante qui permettra d'activer ou désactiver le bouton Supprimer selon qu'une ligne est sélectionnée ou non dans le DataGridView :

```
/// <summary>
/// active ou désactive le bouton supprimer
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void grdStagiaires_SelectionChanged(object sender,
EventArgs e)
    { // à vous de jouer… }
```

- Déplacer et adapter selon les besoins le code permettant la suppression d'un stagiaire depuis `frmExo9` vers `CtrlListerStagiairesSection` (il peut être nécessaire de modifier la visibilité de certains membres des classes…) ; compiler et tester : la suppression de stagiaire devrait être effective mais le double-clic en `DataGridView` et le clic sur bouton *Ajouter* ne répondent plus ;
- Déplacer de même le code permettant l'affichage du détail d'un stagiaire. Mais attention, ici, la consultation/modification d'un stagiaire **fait l'objet d'un autre use case donc d'un autre Contrôleur** : il s'agit donc d'instancier un nouveau Contrôleur (et non plus un form `frmVisuStagiaire`) en lui passant les paramètres nécessaires à son bon déroulement :

```
/// <summary>
/// double-clic sur DataGridView du form frmExo9 :
/// instancier le form détail en y affichant
/// le stagiaire correspondant à la ligne double-cliquée
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void grdStagiaires_DoubleClick(object sender, EventArgs e)
{
    // ouvrir la feuille détail en y affichant
    // le stagiaire correspondant à la ligne double-cliquée
    MStagiaire leStagiaire;
    Int32 laCle; // clé (=numOSIA) du stagiaire dans la collection
    // récupérer clé du stagiaire cliqué en DataGridView
```

```
            laCle =
        (Int32)this.leForm.grdStagiaires.CurrentRow.Cells[0].Value;
            // instancier un objet stagiaire pointant vers
            // le stagiaire d'origine dans la collection
            leStagiaire = this.laSection.RestituerStagiaire(laCle);
            // instancier un Contrôleur de form détail pour ce stagiaire
            CtrlVisuModifStagiaire ctrlVisu = new
        CtrlVisuModifStagiaire(leStagiaire);
            // en sortie, régénérer l'affichage du dataGridView
            this.leForm.AfficheStagiaires(this.laSection);
        }
```

- Ajouter tout de suite une classe `CtrlVisuModifStagiaire` de visibilité `public` et déclarer la signature de son constructeur qui reçoit en paramètre une référence de `MStagiaire` ;
- Retour au Contrôleur `CtrlListerStagiairesSection` : écrire/déplacer le code associé à la procédure événementielle clic sur bouton *Ajouter* : il s'agit pour l'instant d'instancier un Contrôleur `CtrlNouveauStagiaire` en lui passant la référence à la section courante ; ajouter tout de suite la classe publique `CtrlNouveauStagiaire.cs` et déclarer la signature de son constructeur qui reçoit en paramètre une référence de `MSection` ; compiler et tester ; le Contrôleur `CtrlListerStagiairesSection` est maintenant opérationnel et ***il contrôle bien le déroulement des traitements accessibles depuis la classe de Présentation*** `frmExo9` (il reste bien entendu à écrire la suite des Contrôleurs `CtrlVisuModifStagiaire` et `CtrlNouveauStagiaire`).

5. Deuxième étape : développer le Contrôleur du use case 'consulter/modifier un stagiaire particulier'.

A ce stade, la classe `CtrlVisuModifStagiaire` existe ainsi que la signature de son constructeur (à tout moment, n'hésitez pas à consulter l'animation PowerPoint pour illustrer le fonctionnement interne de l'application finale).

- Ajouter au Contrôleur `CtrlVisuModifStagiaire` les 2 membres privés de niveau classe nécessaires au bon fonctionnement, `leForm` de type `frmVisuStagiaire` et `leStagiaire` de type `MStagiaire` ;
- Ecrire le code du constructeur de la classe `CtrlVisuModifStagiaire` : il s'agit d'instancier un form `frmVisuStagiaire` en lui passant en paramètre la référence du stagiaire à consulter/modifier, et d'implémenter la procédure événementielle correspondant au clic sur le bouton '*Saisir note*' ; rien de bien nouveau… (éventuellement déplacer la demande d'affichage des données depuis l'événement `Load` vers ce constructeur car l'utilisation de cet événement n'est plus totalement justifiée en construction orientée objet) :
- Ecrire le code associé au clic du bouton '*Saisir note*' en le déplaçant et en l'adaptant si nécessaire depuis `frmVisuStagiaire` vers `CtrlVisuModifStagiaire` ; noter que ce form de saisie de note est ici défini comme faisant partie du même use case et il n'est donc pas nécessaire de définir un nouveau Contrôleur pour cette action ;
- Adapter le code du form **`frmVisuStagiaire`** de manière à y conserver uniquement ce qui concerne l'affichage et le contrôle de saisie du stagiaire ; il ne devrait rester que :

```
/// <summary>
/// le stagiaire à afficher / modifier
/// (on ne travaille pas directement sur le stagiaire fourni
/// et l'utilisateur pourra abandonner par bouton Annuler)
```

```
/// </summary>
private MStagiaire leStagiaire;

/// <summary>
/// Constructeur
/// </summary>
/// <param name="unStagiaire">ref au stagiaire à visualiser / 
modifier</param>
public frmVisuStagiaire(MStagiaire unStagiaire)
{…}

/// <summary>
/// fermer sans incidence (action locale-> reste private)
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnFermer_Click(object sender, EventArgs e)
{…}

/// <summary>
/// abandon de modif en cours ==> réafficher anciennes valeurs
/// (action locale-> reste private)
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnAnnuler_Click(object sender, EventArgs e)
{…}

/// <summary>
/// impacter les modifications saisies sur le form :
/// modifie les données de l'objet MStagiaire en fonction des valeurs
/// courantes dans les contrôles graphiques
/// l'action reste locale pour ne pas exposer tous les controles 
graphiques
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnValider_Click(object sender, EventArgs e)
{…}

/// <summary>
/// au démarrage, afficher le stagiaire reçu sur le form
/// (pourrait être réintégré dans le constructeur)
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void frmVisuStagiaire_Load(object sender, EventArgs e)
{…}

/// <summary>
/// affiche en textbox les données d'un stagiaire reçu
/// appelable aussi bien depuis le controleur -> internal
/// </summary>
/// <param name="unStagiaire">la référence du stagiaire à afficher</param>
```

`internal void AfficheStagiaire(MStagiaire unStagiaire)``{…}`

Dans cette configuration, la classe de Présentation `frmVisuStagiaire` gère elle-même les actions de ses boutons *Annuler* et *Fermer* qui présentent un intérêt 'local', concernant uniquement l'affichage. Par contre, l'action associée au bouton *Valider*, qui met à jour les données, reste locale, ce qui signifie que la mise à jour de l'objet Métier échappe au Contrôleur (alors que c'est bien le cœur du traitement). Pour aller plus loin dans le découpage, 2 solutions s'offrent à nous :

- déporter l'action du bouton *Valider* dans le Contrôleur, mais cela impliquerait que le Contrôleur 'connaisse' les contrôles graphiques du form pour accéder à leurs contenus (les données saisies par l'utilisateur) ; au final, le form devrait donc exposer en `public` ou `internal` tous ses contrôles graphiques (ou des propriétés permettant d'accéder à leurs valeurs) ce qui multiplierait les liens entre ces classes au détriment de la réutilisabilité et de l'évolutivité.
- ne pas associer d'action au bouton *Valider* dans le form et exposer au Contrôleur une méthode permettant de mettre à jour l'objet Métier en fonction du contenu des contrôles graphiques.

Choisissons cette dernière solution, plus 'élégante' en terme de construction orientée objet…

Dans le form `frmVisuStagiaire`, transformer la procédure événementielle btnValider_Click :

```
/// <summary>
/// Affecte les modifications apportées en contrôles graphiques sur l'objet stagiaire courant
/// </summary>
internal void Valider()
{
    try
    {
        // modifier les valeurs du stagiaire pointé par la ref temporaire
        // (sauf numéro OSIA non modifiable)
       this.leStagiaire.Nom = this.txtNom.Text;
       this.leStagiaire.Prenom = this.txtPrenom.Text;
       this.leStagiaire.Rue = this.txtAdresse.Text;
       this.leStagiaire.Ville = this.txtVille.Text;
       this.leStagiaire.CodePostal = this.txtCodePostal.Text;
       // fermer le form
       this.DialogResult = DialogResult.OK;
    }
    catch (Exception ex)
    {
        MessageBox.Show("Erreur : \n" + ex.Message, "Modification de stagiaire");

    }
}
```

NB : on suit ici les conventions Microsoft : les noms des méthodes maintenant exposées commencent par une MAJUSCULE

Supprimer le handler d'événement local au form sur le bouton *Valider* et exposer ce bouton en `public` ou `internal`.

Dans le Contrôleur `CtrlVisuModifStagiaire`, ajouter l'implémentation de l'action associée au bouton *Valider* du form et écrire la procédure événementielle associée :

```
/// <summary>
/// constructeur : instancie et personnalise le form et l'affiche en modal;
/// </summary>
/// <param name="unStagiaqire">le ref du stagiaqire à mettre à jour</param>
```

```
public CtrlVisuModifStagiaire(MStagiaire unStagiaire)
{
    // mémo ref au stagiaire à mettre à jour
    this.leStagiaire = unStagiaire;
    // instancier le form initial
    this.leForm = new frmVisuStagiaire(this.leStagiaire);
    this.leForm.Text = this.leStagiaire.ToString();
    // implémenter l'événement bouton Saisir Note
    this.leForm.btnSaisirNote.Click += new EventHandler(btnSaisirNote_Click);
    // implémenter l'événement bouton valider
    this.leForm.btnValider.Click += new EventHandler(this.btnValider_click);

    // afficher le form
    this.leForm.ShowDialog();
}

/// <summary>
/// appelle le service de mise à jour par le form de l'objet stagiaire
actuellement affiché
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnValider_click(object sender, EventArgs e)
{
    this.leForm.Valider();

}
```

Cette dernière transformation peut sembler inutile et coûteuse mais elle présente l'avantage d'un meilleur partage des rôles : maintenant le form se limite au dialogue avec l'utilisateur grâce à ses contrôles graphiques privés, sans savoir comment seront exploitées les données ; il expose un service de mise à jour de l'objet Métier reçu. Le Contrôleur contrôle bien la logique applicative du use-case qu'il représente en déclenchant lui-même la mise à jour.
Les relations entre les classes Contrôleur et Présentation restent limitées à des références d'objets Métier et de boutons de commande, ce qui assure une meilleure indépendance entre classes et donc une meilleure maintenabilité et évolutivité de l'application.

6. Troisième étape : développer le Contrôleur du use case 'ajouter un stagiaire dans une section.

A ce stade, la classe **`CtrlNouveauStagiaire`** existe ainsi que la signature de son constructeur (à tout moment, n'hésitez pas à consulter l'animation PowerPoint pour illustrer le fonctionnement interne de l'application finale).

Le fonctionnement de ce Contrôleur va un peu varier car, ***comme les objets `MStagiaires` sont dérivés en `MStagiaireDE` et `MStagiaireCIF`, c'est bien le form `frmAjoutStagiaire` qui 'saura' quel type réel instancier et renseigner ses propriétés.*** Aussi la référence du stagiaire à ajouter ne peut-elle être passée ni au form ni au Contrôleur `CtrlNouveauStagiaire` et ces derniers devront exposer un accesseur à la référence de l'objet stagiaire instancié pour en informer l'extérieur (c'est bien le Contrôleur appelant `CtrlListerStagiairesSection` qui doit insérer cet objet dans la collection des stagiaires de la section qu'il est en charge de gérer).

De plus, ce Contrôleur `CtrlNouveauStagiaire` devra aussi informer le Contrôleur appelant sur le résultat du dialogue en form `frmAjoutStagiaire` afin que le Contrôleur appelant `CtrlListerStagiairesSection` prenne en compte ou non les données saisies, en fonction du bouton *OK* ou *Annuler* cliqué par l'utilisateur dans le dialogue ; pour ce faire on choisit ici de 'propager' tout simplement de Contrôleur en Contrôleur une variable de type `DialogResult`, reflet du `DialogResult` du form de saisie.
NB : `DialogResult` est un type de *l'espace de nom* `System.Windows.Forms` ; il est plus spécialement 'destiné' à la gestion des Winform et ce n'est donc pas une 'excellente' solution d'utiliser ce type dans un *Contrôleur* puisque les *Contrôleurs* devraient rester facilement adaptables d'un contexte technique à l'autre. Le monde n'étant pas parfait et la perfection ayant un coût élevé, on se permettra ici cette 'transgression' aux 'choix d'implémentation' énoncées au début de ce TP.

- Ajouter les membres privés de niveau classe et les accesseurs nécessaires au bon fonctionnement du Contrôleur **`CtrlNouveauStagiaire`**:

```
/// <summary>
/// ref au form frmAjoutStagiaire
/// </summary>
private frmAjoutStagiaire leForm;

/// <summary>
/// ref au stagiaire saisi par le form frmAjoutStagiaire
/// </summary>
private MStagiaire leStagiaire;

/// <summary>
/// obtient la ref au stagiaire saisi par le form frmAjoutStagiaire
/// </summary>
public MStagiaire LeStagiaire
{
    get
    {
        return this.leStagiaire;
    }
}
```

```
/// <summary>
/// ref à la section du stagiaire traité
/// </summary>
private MSection laSection;

/// <summary>
/// résultat du dialogue modal assuré par le form frmAjoutStagiaire
/// </summary>
private DialogResult resultat;

/// <summary>
/// obtient le résultat du dialogue modal assuré par le form
frmAjoutStagiaire
/// </summary>
public DialogResult Resultat
{
    get
      {
        return this.resultat;
      }
}
```

- Compléter le constructeur : il doit mémoriser dans une variable de niveau classe la référence à la section reçue en paramètre, instancier le form `frmAjoutStagiaire` et lui personnaliser le titre (préciser la section concernée) et l'afficher en mode 'modal'. En fin de dialogue, ce Contrôleur doit mémoriser le résultat du dialogue modal et récupérer la référence du stagiaire créé par le form `frmAjoutStagiaire`.
- Adapter le code du from `frmAjoutStagiaire` : il n'a plus à connaître la section concernée par ce stagiaire (c'est le rôle du Contrôleur) et il se contente de faire son travail d'affichage, de saisie et de contrôle de validité des données stagiaires entrées dans ses contrôles graphiques (méthode exposée `Controle()`) ; sa méthode exposée `Instancie()` peut travailler sur une *référence d'objet* `MStagiaire` *générique* et *instancie le stagiaire spécialisé* en lui renseignant ses valeurs de propriétés ; ce form est instancié par le Contrôleur `CtrlNouveauStagiaire` qui ne gère pas la collection des stagiaires et qui a été instancié par le Contrôleur `CtrlListerStagiairesSection` ; **c'est donc le Contrôleur `CtrlListerStagiairesSection` qui réalisera l'insertion du nouveau stagiaire dans la collection** des stagiaires de sa section, en fonction du résultat du dialogue exposé par le Contrôleur `CtrlNouveauStagiaire`.

    - dans **`CtrlListerStagiairesSection`** :

```
/// <summary>
/// bouton ajouter du form frmExo9 :
/// instancier un form de saisie via le controleur
CtrlNouveauStagiaire
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnAjouter_Click(object sender, EventArgs e)
{
    // instancier un ctrl de saisie de stagiaire
    // pour afficher le form en modal
```

```
    CtrlNouveauStagiaire ctrlNouveau = new CtrlNouveauStagiaire
(this.laSection);
    // si on sort de la saisie par OK
    if (ctrlNouveau.Resultat == DialogResult.OK)
    {
      // ajouter la référence de l'objet MStagiaire créé par le
ctrleur dans la collection des stagiaires de la section
      this.laSection.Ajouter(ctrlNouveau.LeStagiaire);

      // régénèrer l'affichage du dataGridView
      this.leForm.AfficheStagiaires(this.laSection);
    }
  }
```

- dans **`frmAjoutStagiaire`**, laisser en l'état les actions locales associées aux 2 radio-boutons et au bouton *Annuler* (cela ne concerne que l'affichage). On pourrait laisser en local l'action associée au bouton de validation *OK* mais il va être préférable de ne conserver que les opérations de contrôles de saisie (méthode `controle()`) et d'instanciation de l'objet Métier (méthode `instancie()`) en déportant le contrôle de ces opérations… dans le Contrôleur `CtrlNouveauStagiaire` : supprimer le *handler* d'événement local au form concernant le bouton *OK*, exposer en `public` ou `internal` les 2 méthodes `Controle()` et `Insancie()`, et déplacer (en l'adaptant) le code associé au clic du bouton OK depuis le form vers le Contrôleur :

```
/// <summary>
/// constructeur : instancie et personnalise le form et l'affiche en
modal;
/// </summary>
/// <param name="uneSection">la section du stagiaire à créer</param>
public CtrlNouveauStagiaire (MSection uneSection)
{
    this.laSection = uneSection;
    // instancier le form initial
    this.leForm = new frmAjoutStagiaire();
    this.leForm.Text += this.laSection.ToString();
    // variante avec plus de contrôle du form par le Contrôleur
    this.leForm.btnOK.Click += new EventHandler(this.btnOK_Click);
    // afficher le form
    this.leForm.ShowDialog();
}

/// <summary>
/// bouton OK du from : demande au form de se contrôler puis
d'instancier un objet stagiaire spécialisé
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnOK_Click(object sender, EventArgs e)
{
    if (this.leForm.Controle()) // controles sur les données saisies
dans le form
    {
      if (this.leForm.Instancie()) // instancie objet stagiaire
spécialisé
```

On suit ici les conventions Microsoft : les noms des méthodes maintenant publiques commencent par une MAJUSCULE

```
      {
        this.leForm.DialogResult = DialogResult.OK; // fermer le form si
OK
        // en fin de dialogue modal récupérer le résultat de la saisie
        this.resultat = this.leForm.DialogResult;
        // et récupérer la ref du stagiaire spécialisé instancié par le
form
        this.leStagiaire = this.leForm.LeStagiaire;
      }
      else // erreur d'instanciation d'objet stagiaire
     {
       this.resultat = DialogResult.No; // erreur à remonter
     }
   }
}
```

Attention de bien déplacer, depuis le form vers le contrôleur, l'instruction permettant d'ajouter dans la collection de `MStagiaire` la référence au nouvel objet Métier.

Ici encore, ce dernier découpage a pour seul but de bien séparer les rôles entre les classes Présentation et Contrôleur.

Précisons encore une fois que cette répartition des traitements entre forms et Contrôleurs reste arbitraire et définie dans les choix d'implémentation exposés en début de ce document. Nous vous invitons à les relire pour mieux comprendre ces choix maintenant que vous les avez mis en œuvre.

Synthèse : classes Contrôleur en C#

- Une classe Contrôleur représente un use case et en implémente les actions ; elle contrôle le déroulement des opérations ;
- Le constructeur de la classe déroule le scénario principal/nominal du use case ;
- Le Contrôleur assure les traitements des scenarii alternatifs du use-case grâce à des méthodes supplémentaires (bien souvent, des méthodes événementielles correspondant à des événements survenus sur les forms) ;
- Chaque Contrôleur instancie les forms nécessaires au déroulement du dialogue correspondant au use case et leur passe en paramètre les références des objets Métier nécessaires aux affichages et aux saisies ;
- De plus, comme les forms tendent à être relativement indépendants de leurs usages, chaque Contrôleur adapte les forms instanciés de manière à coller aux actions prévues dans le use case (implémentation de méthodes événementielles, personnalisation des affichages et de la barre de titre, détermination du 'parent MDI'…) ;
- Les objets Métier à afficher/modifier dans les forms sont recherchés/instanciés par les Contrôleurs et fournis lors de l'instanciation du form ;
- Les nouveaux objets Métier instanciés par les forms eux-mêmes sont exposés par eux sous forme de propriétés pour être récupérés par les Contrôleurs ;
- Les forms exposent aux Contrôleurs les services qu'ils sont capables d'assurer, sous forme de méthodes ; ils conservent en privé leurs contrôles graphiques (meilleure indépendance entre classes et meilleure évolutivité) et exposent simplement leurs boutons de commande de manière à ce que les Contrôleurs puissent y associer des procédures événementielles qui utilisent les services exposés par les forms ;
- Le dialogue entre objets de Présentation et Contrôleurs se fait donc essentiellement par passation/exposition de référence d'objets Métier, ainsi que par levées/interception d'erreurs (Exceptions) ;
- Un Contrôleur peut instancier un autre Contrôleur (en lui passant/récupérant éventuellement les références des objets Métier nécessaires) de manière à refléter le niveau de découpage en use cases effectué lors de l'analyse de l'application ('include'/'extend'), et à assurer la réutilisabilité des composants développés ;
- La couche Service est constituée des classes Contrôleur et de classes 'outils' d'intérêt général, non-spécifiques à une application en particulier ;