Mission n°1 – *Application de gestion des notes de frais*

Une application de gestion des notes de frais des visiteurs médicaux de la société GSB est en cours de développement. Cette application, permettant de gérer les différentes dépenses devant être remboursées, a été commencée mais demande certaines modifications avant d'être reprise par l'équipe de développement. Vous devez effectuer les modifications suivantes :

I. Sérialisation

La sérialisation correspond au processus de conversion d'un objet en un formulaire facilement transportable. Par exemple, vous pouvez sérialiser un objet et le transporter par Internet via HTTP entre un client et un serveur. À l'autre extrémité, la désérialisation reconstruit l'objet du flux de données.
La sérialisation est donc un mécanisme qui permet de donner une représentation "à plat" d'un objet en mémoire. La désérialisation est le mécanisme inverse qui reconstruit en mémoire un objet à partir d'une représentation "à plat". La sérialisation permet de conserver l'état d'un objet pour une manipulation ultérieure. Nous allons utiliser ici la sérialisation binaire, nous utiliserons plus tard la sérialisation XML.

Travail à réaliser :

- Dans l'application, ajouter une classe PersisteServiceCommercial avec deux méthodes statiques serialiser() et deserialiser().
- Utiliser les méthodes statiques de cette classe pour persister notre service commercial en complétant le menu Fichier>Ouvrir et Fichier>Enregistrer.

II. Principe d'encapsulation

Le principe de l'encapsulation ne permet pas l'accès direct aux attributs. Il est donc nécessaire de définir des méthodes qui permettent d'obtenir l'information voulue. Ces méthodes (publiques) permettant d'accéder aux attributs (privés) d'un objet sont appelés accesseurs.
Il existe deux types d'accesseurs :

- les accesseurs en lecture (qui commencent par get suivi du nom de l'attribut),
- les accesseurs en écriture (qui commencent par set suivi du nom de l'attribut).

Notre application met en place plusieurs accesseurs dans les différentes classes utilisées mais ces accesseurs sont de simples accesseurs par défaut. Le but d'un accesseur est de permettre de vérifier que l'on ne fait pas « n'importe quoi » avec les attributs de l'objet ; c'est pour cela que nous allons vraiment utiliser la fonction de nos accesseurs.

Travail à réaliser :

- Dans la classe Commercial, modifier le code pour que le nom du commercial soit toujours en majuscule et que son prénom commence seulement par une majuscule.
- Dans la classe FraisTransport, modifier le code pour qu'un kilométrage saisi ne soit jamais négatif. Si un kilométrage négatif est saisi, la valeur 0 sera retenue.

III. Héritage

Une classe peut être déclarée comme héritant d'une autre classe si elle peut utiliser toutes les méthodes de cette dernière et si elle contient également tous ses attributs.

Par exemple, la classe NoteFrais, appelée classe mère, transfère toutes ses caractéristiques (attributs et méthodes) à la nouvelle classe FraisTransport, appelée classe fille. On dit que la classe fille FraisTransport hérite de la classe mère NoteFrais.
Une classe fille

- peut utiliser tous les attributs et les méthodes de sa classe mère (protected et public),
- possède en plus des attributs et méthodes propres.

Un objet de la classe fille est un objet de la classe mère avec des caractéristiques supplémentaires.

Travail à réaliser :

- Actuellement, une note de frais nuitée ne prend en charge que le petit déjeuner et la nuit d'hôtel car la société GSB ne rembourse pas le repas du soir. La nouvelle politique est de prendre en charge ce repas s'il a été payé par le visiteur médical (il arrive parfois que le visiteur médical soit invité par un client). Il existe donc des nuitées simples et des nuitées repas. La logique de calcul du remboursement du repas du soir est la même que le repas de midi.

IV. Polymorphisme

Polymorphisme est un mot grec qui signifie « de forme multiple ». Une classe mère peut définir et implémenter des méthodes virtuelles et ses classes filles peuvent les substituer, ce qui signifie qu'elles fournissent leur propre définition et implémentation. Par exemple, au moment de l'exécution, lorsque le code client appelle la méthode calculMontantARembourser(), le type de l'objet est recherché et cette substitution de la méthode virtuelle peut avoir lieu. Par conséquent, dans votre code source vous pouvez appeler une méthode d'une classe mère et provoquer l'exécution de la version d'une classe fille de la méthode.

Travail à réaliser :

- Ecrire la méthode polymorphe entete() qui retourne Nuitée, Transport, Repas ou Nuitée avec repas en fonction du type de note de frais. Cette méthode sera ensuite utilisée dans la méthode ToString().

V. Gestion des exceptions

La gestion des erreurs est un élément indispensable en programmation. Les erreurs qui pourraient survenir à l'exécution de cette application doivent être prises en compte. On appelle exception une erreur générée lors de l'exécution d'un programme ou d'une fonction. Cette exception peut être le résultat d'une erreur ou bien être provoquée par le développeur afin d'alerter le programme appelant d'une situation anormale.

Travail à réaliser :

- Gérer le problème si l'utilisateur ne sélectionne pas un fichier valide à désérialiser.
- Prévoir les erreurs de saisies sur les différents formulaires.
- … (Eviter un maximum d'erreurs par la gestion des exceptions).

VI. Tests unitaires

L'activité de test est extrêmement importante dans le développement. Tester revient à vérifier que l'application respecte son cahier des charges. Si l'on veut tester les fonctionnalités attendues, il faut effectuer des tests dès les premières méthodes écrites. On parle alors de tests unitaires. Un test unitaire est un test portant sur une seule méthode.
Dans son état actuel, l'application fonctionne correctement et les différentes fonctions ont été testées. Dans le futur, l'application devra être modifiée et il arrive parfois que ces modifications impactent le code existant. Pour être certain du bon fonctionnement du code d'origine et ainsi être certain que le code n'a pas

subi de régression, des tests unitaires doivent être écrits pour permettre de vérifier le code tout au long du développement.

Travail à réaliser :

- Développer au minimum une méthode de test pour chacune des méthodes de l'ensemble des classes de l'application.

VII. Les classes interfaces

La notion d'interface est importante en POO. Elle permet de seulement présenter les services utiles à une classe utilisatrice. Cela permet ainsi de masquer l'implémentation en exposant une interface.

La classe ServiceCommercial utilise les services proposés par la classe Commercial (ajout de notes, ...). Dans la suite de notre application, nous pouvons envisager que ces mêmes commerciaux participent à un autre processus, par exemple le service comptable responsable des salaires de tous les employés. De nombreux services proposés par la classe Commercial ne sont pas utiles pour établir les bulletins de salaire.

La programmation objet permet à une classe de présenter différentes interfaces suivant les consommateurs de la classe. Ainsi, la même classe Commercial proposera une interface au service commercial en fournissant les services dont il a besoin mais proposera une autre interface à une autre.

Travail à réaliser :

- Créer l'interface VisiteurMedical.
- Créer l'interface Salarie utile au nouveau ServiceComptable.

Annexe n°1 : Sérialisation

https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/concepts/serialization/walkthrough-persisting-an-object-in-visual-studio

Annexe n°2 : Tests unitaires

https://msdn.microsoft.com/fr-fr/library/hh694602.aspx

Annexe n°3 : Gestion des exceptions

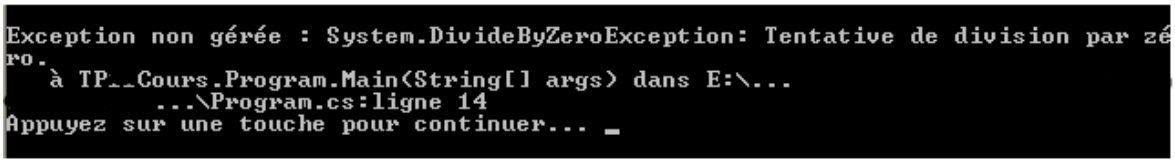
La gestion des erreurs est un élément indispensable en programmation. Nous abordons ici les erreurs qui surviennent à l'exécution d'un programme. On appelle exception une erreur générée lors de l'exécution d'un programme ou d'une fonction. Cette exception peut être le résultat d'une erreur ou bien être provoquée par le développeur afin d'alerter le programme appelant d'une situation anormale. Les langages objets disposent de classes d'exception correspondant aux causes de l'exception. En C# la classe la plus haute hiérarchiquement est la classe Exception.

1. Exemples

Le code suivant :

```
int i, n=5;
for (i=-1 ; i<3 ; i++)
      n/=i;
```

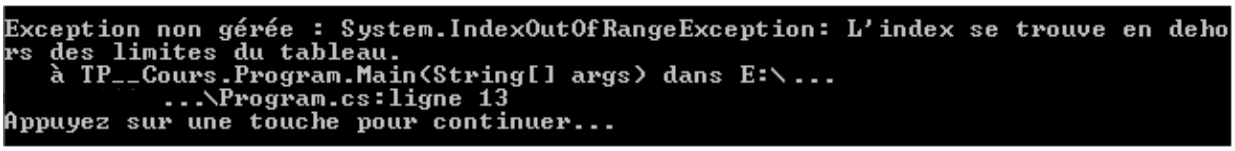
provoque une erreur à l'exécution et génère le message suivant :



Le code suivant :

```
int[] t ={1,2,3,4,5};
int n = t[6];
```

provoque aussi une erreur à l'exécution mais avec un message différent :



2. Gestion des exceptions – try/catch

Pour éviter l'arrêt d'exécution du programme, nous allons intercepter l'erreur en prenant en compte la gestion des erreurs. Pour cela, nous allons utiliser un bloc *try*.

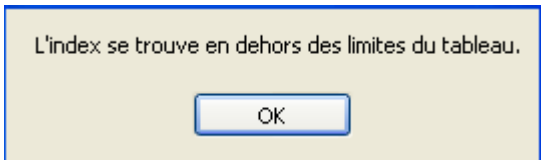
```
try
{
      int[] t ={1,2,3,4,5};
      int n = t[6];
}
```

Ce contexte doit être suivi d'instructions qui indiquent au programme ce qu'il doit faire en cas d'erreur :

```
catch (System.IndexOutOfRangeException ex)
{
```

```
            MessageBox.Show(ex.Message);
        }
```

L'exception est maintenant gérée. L'erreur n'arrête plus le programme mais ouvre seulement une boite de dialogue avec le message correspondant :



La syntaxe générale d'interception des exceptions est la suivante :

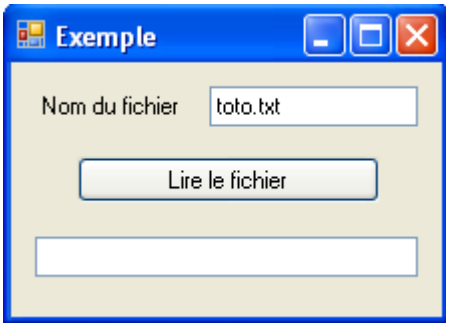
```
try
{
      // code à tester
}
catch (<classe d'exception> nom1)
{
      // code à exécuter pour cette exception
}
catch (<classe d'exception> nom2)
{
      // code à exécuter pour cette exception
}
finally
{
      // code qui s'exécute dans tous les cas
}
```

Commentaires :

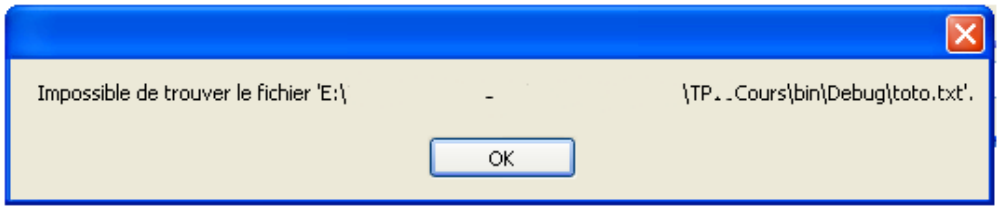
- A un bloc try peut correspondre plusieurs catch, si plusieurs catch sont dans la même hiérarchie de classe, il faut commencer par les exceptions les plus génantes.
- Dans le bloc finally (facultatif), on place le code dont on désire l'exécution dans tous les cas. Par exemple si une connexion est ouverte dans le try, on peut souhaiter la fermer dans le finally et non à la fin du try.

3. Déclenchement des exceptions

Il est possible de lever une exception en utilisant l'instruction throw. Prenons l'exemple suivant qui affiche le contenu d'un fichier dans une zone de texte :



Si le nom du fichier donné dans la zone de texte est incorrect, une exception est générée de la classe System.IO.FileNotFoundException :



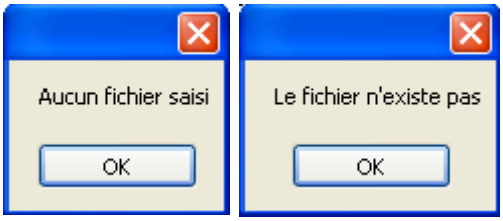
Le code utilisé est le suivant :

```
private void Affiche(string nomFichier, TextBox tx)
{
    try
    {
        StreamReader fichier = new StreamReader(nomFichier);
        string ligne = fichier.ReadLine();
        tx.Text = ligne;
    }
    catch (System.IO.FileNotFoundException ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Cette procédure est appelée dans l'évènement clic du bouton :

```
Affiche(txtNomFic.Text, txtSortie);
```

Nous pouvons émettre des messages différents pour la gestion d'une exception en modifiant notre code et en envoyant nous même une nouvelle exception via throw :



Le code doit être modifié de la façon suivante :

```
private void Affiche(string nomFichier, TextBox tx)
{
    try
    {
        StreamReader fichier = new StreamReader(nomFichier);
        string ligne = fichier.ReadLine();
        tx.Text = ligne;
    }
    catch (FileNotFoundException e1)
    {
        throw new FileNotFoundException("Le fichier n'existe pas");
    }
    catch (Exception e2)
    {
        throw new Exception("Aucun fichier saisi");
    }
}

private void button1_Click(object sender, EventArgs e)
{
    try
    {
        Affiche(txtNomFic.Text, txtSortie);
    }
    catch (FileNotFoundException e1)
    {
        MessageBox.Show(e1.Message);
```

```
            }
            catch (Exception e2)
            {
                MessageBox.Show(e2.Message);
            }
        }
```

4. Validation de saisie

On peut également utiliser ce mécanisme pour lever une exception qui n'est pas une erreur mais seulement une exception dans un contexte particulier :

Supposons que l'on saisisse des notes et que celles-ci doivent être comprises entre 0 et 20 :

```
        private void ValideNote(int n)
        {
            if (n < 0 || n > 20)
                throw new Exception("Note invalide");
        }
```

L'appel *ValideNote(22)* va provoquer une exception que l'on peut intercepter dans un bloc try/catch :

```
            try
            {
                ValideNote(22);
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
```

Ce qui produit :

Il est possible d'améliorer cette gestion en utilisant une exception mieux appropriée : *ArgumentOutOfRangeException* qui permet de récupérer l'argument objet de l'exception :

```
        private void ValideNote(int n)
        {
            if (n < 0 || n > 20)
                throw new System.ArgumentOutOfRangeException("note", n, "non valide");
        }
```

Le bloc catch de l'appel doit référencer maintenant ce type d'exception :

```
            try
            {
                ValideNote(25);
            }
            catch (System.ArgumentOutOfRangeException ex)
            {
                MessageBox.Show("La note " + ex.ActualValue.ToString() + " n'est pas
valide.");
            }
```

5. Résumé

- Utilisez systématiquement les blocs try/catch.
- Catchez des exceptions appropriées à la situation.
- Si plusieurs exceptions sont énumérées, placez la plus spécifique en premier.
- Il faut prévoir, lorsque l'on écrit une fonction susceptible de lever une exception, de la propager à l'aide de l'instruction throw.
- N'utiliser l'instruction throw que pour des exceptions exceptionnelles.

Annexe n°4 : Interface

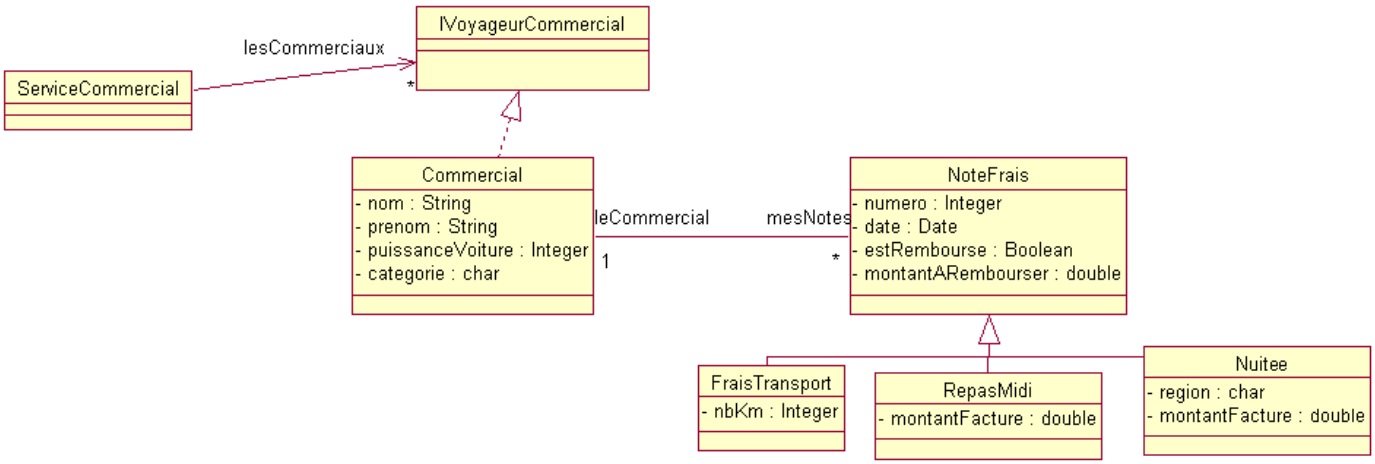
La notion d'interface est très importante en programmation objet. Elle permet de présenter uniquement les services utiles à une classe cliente. On masque ainsi l'implémentation en exposant une interface.
Les interfaces permettent de définir un ensemble de propriétés, méthodes, évènements mais elles ne contiennent aucun code. L'implémentation doit s'effectuer au niveau de la classe elle-même. L'interface constitue un contrat que vous signez. En déclarant que votre classe implémente une interface, vous vous engagez à fournir, dans votre classe, tout ce qui est défini dans l'interface.

6. Mise en œuvre d'interfaces dans la gestion des commerciaux

a. Interface voyageur commercial

Le service commercial développé précédemment utilise les services proposés par la classe Commercial (ajout de notes, ...). Si notre domaine d'étude était plus vaste et les commerciaux auraient pu intervenir dans un autre processus, dans le service comptable responsable des salaires de tous les employés par exemple. De nombreux services disponibles de la classe Commercial ne seraient alors pas utiles pour l'établissement des salaires (ajout de notes de frais par exemple).
La programmation objet permet à une classe de présenter différentes interfaces suivant les *consommateurs* de la classe. Ainsi, la même classe Commercial proposera une interface au service commercial en fournissant les services dont il a besoin mais proposera une autre interface à une autre classe correspondant à ses seuls besoins.
La notion d'interface est proche de la notion de spécialisation si ce n'est qu'une classe peut exposer plusieurs interfaces distinctes (ce qui n'est pas le cas de la spécialisation).

Le diagramme de classe ci-dessous présente la nouvelle classe (interface) qui va collaborer avec le service commercial. Cette dernière ne connaîtra pas l'implémentation du commercial mais seulement ses services utiles.



Notation : la relation *implémente* (entre une interface et une classe) se représente avec une flèche pointillée.

b. Code de la nouvelle interface

On déclare une classe interface qui ne contient que les signatures des méthodes (sans le code) utiles au service commercial :

```
public interface IVoyageurCommercial
{
    void ajouterNote(DateTime date, int nbKm);
    void ajouterNote(DateTime date, double montantFacture);
    void ajouterNote(DateTime date, double montantFacture, char region);
    NoteFrais getNoteFrais(int i);
    string Nom { get; }
    string Prenom { get; }
    List<NoteFrais> MesNotes { get; }
    int nbNotes();
}
```

Le niveau de visibilité (toujours public) des méthodes d'une interface n'est pas indiqué.

La classe Commercial est inchangée mise à part le fait qu'elle déclare implémenter l'interface IVoyageurCommercial : `public class Commercial : IVoyageurCommercial`

Cette déclaration est un engagement - qui sera vérifié à la compilation - d'écrire le code des méthodes de l'interface. La classe Commercial peut bien sûr posséder des méthodes autres que celles de son interface.
Le code de la classe ServiceCommercial va faire référence, non plus à la classe Commercial mais à son interface :

```
public class ServiceCommercial
{
    List<IVoyageurCommercial> lesCommerciaux;

    public ServiceCommercial()
    {
        lesCommerciaux = new List<IVoyageurCommercial>();
    }

    public IVoyageurCommercial getCommercial(int i)
    {
        return lesCommerciaux[i];
    }

    public void ajouterCommercial(IVoyageurCommercial commercial)
    {
        lesCommerciaux.Add(commercial);
    }
   …
}
```

La classe Commercial a été remplacée par son interface.

On peut donc écrire le code suivant dans le Main :

```
IVoyageurCommercial c = new Commercial("Dupont", "Jean", 7, 'B');
ServiceCommercial sc = new ServiceCommercial();
sc.ajouterCommercial(c);
```

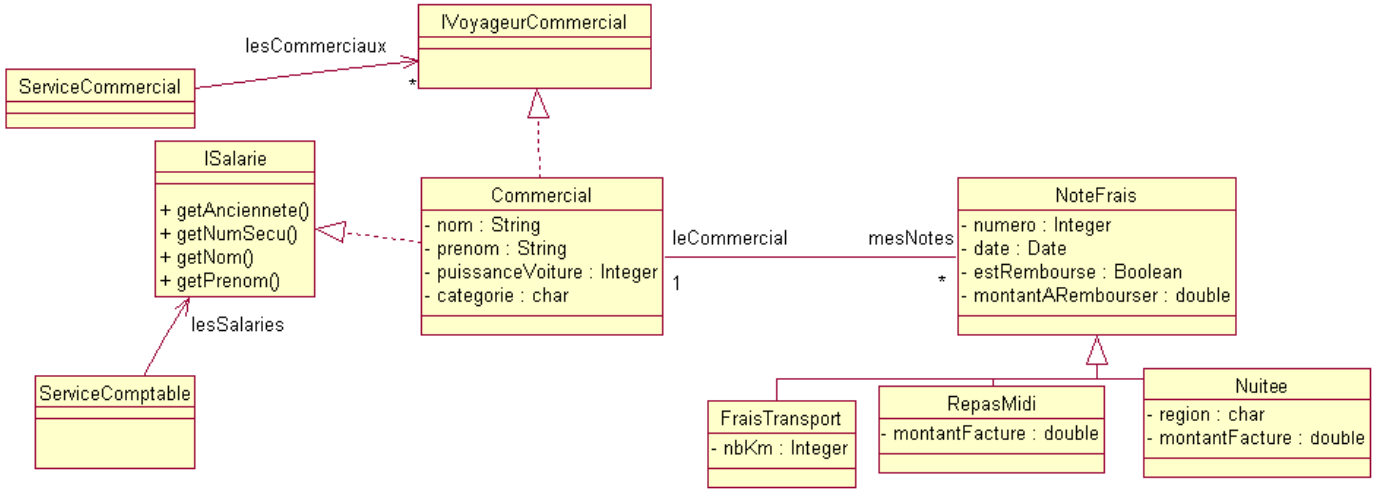
Certes, on crée bien un Commercial que l'on ajoute au service (on ne peut pas instancier une interface) mais le service commercial n'a accès qu'aux méthodes de IVoyageurCommercial, son interface, ce qui était le but recherché.

c. Interface salarié

Le service comptable est responsable de la paye des commerciaux et a besoin de leur numéro de sécu et de leur date d'embauche pour établir la paye.
Ajoutons dans un premier temps ces champs dans la classe Commercial :

```
private string numSecu;
private DateTime dateEmbauche;
```

Créons le service comptable et ajoutons une nouvelle interface ISalarie à la classe Commercial :



Pour simplifier, les méthodes de l'interface IVoyageurCommercial ne figure pas sur le diagramme de classes.

d. Code de l'interface salarié

La classe *ISalarie* se présente ainsi :

```
interface ISalarie
{
    string Nom { get; }
    string Prenom { get; }
    string getNumSecu { get; }
    int getAnciennete() ;
}
```

C'est à la classe Commercial d'écrire (implémenter le code) de ces méthodes (les deux premières sont déjà écrites).

On peut maintenant indiquer que la classe Commercial implémente la nouvelle interface :

```
public class Commercial : IVoyageurCommercial, ISalarie
```

Il est nécessaire d'ajouter un nouveau constructeur correspondant à l'interface *ISalarie* :

```
public Commercial(string nom, string prenom, DateTime dateEmbauche, string numSecu)
{
    this.nom = nom;
    this.prenom = prenom;
    this.dateEmbauche = dateEmbauche;
    this.numSecu = numSecu;
    this.mesNotes = new List<NoteFrais>();
}
```

Complétons au minimum la classe *ServiceComptable* :

```
public class ServiceComptable
{
    private List<ISalarie> lesSalaries;

    public ServiceComptable()
    {
        lesSalaries = new List<ISalarie>();
    }

    public void ajouterSalarie(ISalarie s)
    {
        lesSalaries.Add(s);
    }
```

```
        public ISalarie getSalarie(int i)
        {
            return lesSalaries[i];
        }
    }
```

Il est maintenant possible d'utiliser la nouvelle interface dans le programme principal.

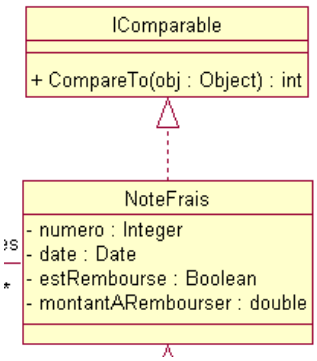
7. Intérêt des interfaces dans un framework

Le framework .Net propose de nombreuses classe *Interface ;* le grand bénéfice de cette organisation est de faire bénéficier ses propres classes des services prévus pour les interfaces. Il suffit ainsi de déclarer que sa classe implémente une interface pour bénéficier de ces services ; à condition, bien sûr, d'implémenter la ou les méthodes exigées par l'interface.

a. L'interface IComparable

Nous allons mettre en œuvre cette situation dans l'exemple de la méthode **Sort** de la classe List<>.
La classe List<> est capable de trier (Sort) les éléments de sa liste à condition que la classe implémente une interface du framework : *IComparable*.

Nous allons, par exemple, trier les notes de frais par date croissante :
Ceci n'est possible que si la classe NoteFrais implémente IComparable dont la seule méthode est CompareTo :



Dans la classe NoteFrais, on déclare l'interface :

```
    public class NoteFrais : IComparable
```

On s'engage ainsi à implémenter la méthode CompareTo :

```
        public int CompareTo(object obj)
        {
            return this.date.CompareTo(((NoteFrais)obj).date);
        }
```

Dans ce cas, le tri se fait sur les dates des notes de frais. On peut envisager un autre tri, plus élaboré, sur les montants à rembourser :

```
        public int CompareTo(object obj)
        {
            int result = -1;
            if (this.montantARembourser > ((NoteFrais)obj).montantARembourser)
                result = 1;
            return result;
        }
```

La classe Commercial peut maintenant trier sa collection :

```
        public void trierNotes()
        {
            this.mesNotes.Sort();
        }
```

On peut tester ainsi :

```
DateTime embauche = new DateTime(1999, 03, 07);
Commercial c = new Commercial("Senneton", "Bénédicte", 7, 'B', embauche, "27012457852154");
DateTime d = new DateTime(2016, 01, 21);
DateTime d1 = new DateTime(2016, 02, 20);
DateTime d2 = new DateTime(2016, 03, 22);
c.ajouterNote(d, 100);
c.ajouterNote(d1, 15.5);
c.ajouterNote(d2, 75, '3');
c.ajouterNote(d2, 89, '2');
c.ajouterNote(d2, 70, '1');
c.trierNotes();
Ecran.affiche(c);
```

b. L'interface IEnumerable

Dans la classe Ecran, une méthode *affiche* permet d'afficher les commerciaux d'un service commercial :

```
public static void affiche(ServiceCommercial sc)
{
    for (int i = 0; i < sc.nbCommerciaux(); i++)
        affiche(sc.getCommercial(i));
}
```

Nous parcourons tous les commerciaux avec une boucle for ; il n'est pas possible d'utiliser un *foreach* car la classe *ServiceCommercial* ne peut *itérer* comme le fait –par exemple- la classe ArrayList. Il est néanmoins possible de fournir à la classe *ServiceCommercial* ce service, en utilisant le mécanisme des classes Interfaces.
Pour pouvoir être *itéré* (c'est à dire pouvoir utiliser *foreach*), la classe doit implémenter *IEnumerable*.
Commençons par ajouter à la classe *ServiceCommercial*, son engagement à implémenter les méthodes de *IEnumerable* : `public class ServiceCommercial : IEnumerable`

L'environnement propose alors d'ajouter la seule méthode à implémenter :

```
public IEnumerator GetEnumerator()
{
}
```

Cette méthode retourne une Interface ; on doit créer une classe implémentant cette interface :

```
class EnumereCommerciaux : IEnumerator
```

L'environnement nous précise alors les méthodes à implémenter :

- *public object Current*
- *public bool MoveNext()*
- *public void Reset()*

Remarque : *Current* est une propriété et non une méthode classique.
Le rôle de la classe est de parcourir les éléments d'un service commercial ; ces éléments sont les commerciaux de la collection. Les trois méthodes font références à ce parcours des commerciaux. C'est pourquoi cette classe *EnumereCommerciaux* doit avoir une référence sur un service commercial.
Les champs peuvent être :

```
private ServiceCommercial leService;
private int index;
```

On ajoute le code des méthodes :

```
class EnumereCommerciaux : IEnumerator
{
    private ServiceCommercial leService;
    private int index;

    public EnumereCommerciaux(ServiceCommercial leService)
    {
        this.leService = leService;
        this.index = -1;
```

```
        }

        public object Current
        {
            get { return this.leService.getCommercial(this.index); }
        }

        public bool MoveNext()
        {
            this.index++;
            return this.index < this.leService.nbCommerciaux();
        }

        public void Reset()
        {
            index = -1;
        }
    }
```

Remarques :

- Le constructeur doit récupérer le service commercial
- Comme *Current* est une propriété, on doit écrire le *geter –la méthode get-* .
- La méthode *MoveNext* fait avancer l'index et indique si on est en fin de parcours.

Terminons en écrivant le code de la méthode à implémenter dans le classe ServiceCommercial :

```
        public IEnumerator GetEnumerator()
        {
            EnumereCommerciaux en = new EnumereCommerciaux(this);
            return en;
        }
```

On peut maintenant modifier la méthode Affiche vue plus haut :

```
        Public static void affiche(ServiceCommercial sc)
        {
            /*for (int i = 0; i < sc.nbCommerciaux(); i++)
                affiche(sc.getCommercial(i));*/
            foreach (Commercial c in sc)
                affiche(c);
        }
```

Il faut maintenant réaliser un test dans le Main pour vérifier le bon fonctionnement de la méthode.