Exercice de Sérialisation dans l'environnement Visual Studio 2017 en C#

EPSI 2019

Mini-Projet

Soit le modèle de données suivant :

Etudiant : va permettre de porter toutes les informations associées à un étudiant.

Etablissement : porte les informations liées à l'établissement que l'on veut gérer.

Filiere : description de la filière et de sa composition.

Matiere : description d'une matière ou discipline …

Service : est généré par la relation entre une matière et une filière. Le service est dispensé par un enseignant et suivi par les étudiants.

On souhaite définir les différentes classes dans notre application. Développer une application qui nous permette de saisir la configuration de l'établissement, de saisir les différentes notes des étudiants au cours de l'année afin de pouvoir éditer les bulletins de notes des étudiants avant les conseils de classe ou autres structures d'évaluation, d'éditer les services des enseignants.

Avant toute chose, on commence par approfondir notre analyse du problème. Il faut définir les classes et leurs différentes associations. Commençons par les classes avec leurs attributs :

Première analyse :

Etudiant	Etablissement	Filiere	Matiere
-Nom :string -Prenom :string -Email:string -IDetudiant:string …. +Constructeur(s)	-Nom -Adresse - Telephone …. +Constructeur(s)	-Intitule -Description …. +Constructeur(s)	-Nom -Description …. +Constructeur(s)

Pour enseignant et étudiant, on voit qu'ils partagent beaucoup de chose d'où l'insertion d'une nouvelle classe personne.

Enseignant
-Nom :string -Prenom :string -Email:string -Num_employe:string …. +Constructeur(s)

Voilà cinq classes qui vont nous permettre de poser l'édifice de notre application.

Veuillez procéder à la création d'un nouveau projet de type bibliothèque de classe. Nommez le miniprojet_lib. Ce projet va produire un assembly de type DLL (Dynamic Library Link). En ligne de commandes **#dotnet new classlib –o miniprojet_lib**.

Traduction dans C# de ces classes

```
namespace miniprojet_lib
{
    public class Etablissement
    {
        public Etablissement(){ }
        public Etablissement(string nom, string adresse, string telephone)
        {
            Nom = nom;
            Adresse = adresse;
            Telephone = telephone;
        }

        public string Nom {get;set;}
        public string Adresse { get; set; }
        public string Telephone { get; set; }

    }
}
```

CONSTRUCTEURS

PROPRIETES

```
namespace miniprojet_lib
{
    public class Filiere
    {
        public Filiere() {}
        public void filiere(string intitule, string description)
        {
            Intitule = intitule;
            Description = description;
        }

        public string Intitule { get; set; }
        public string Description { get; set; }
    }
}
```

PROPRIETES

```
namespace miniprojet_lib
{
    public class Matiere
    {
        public Matiere(){}

        public Matiere(string intitule, string description)
        {
            Intitule = intitule;
            Description = description;
        }

        public string Intitule { get; set; }
        public string Description { get; set; }
    }
}
```

PROPRIETES

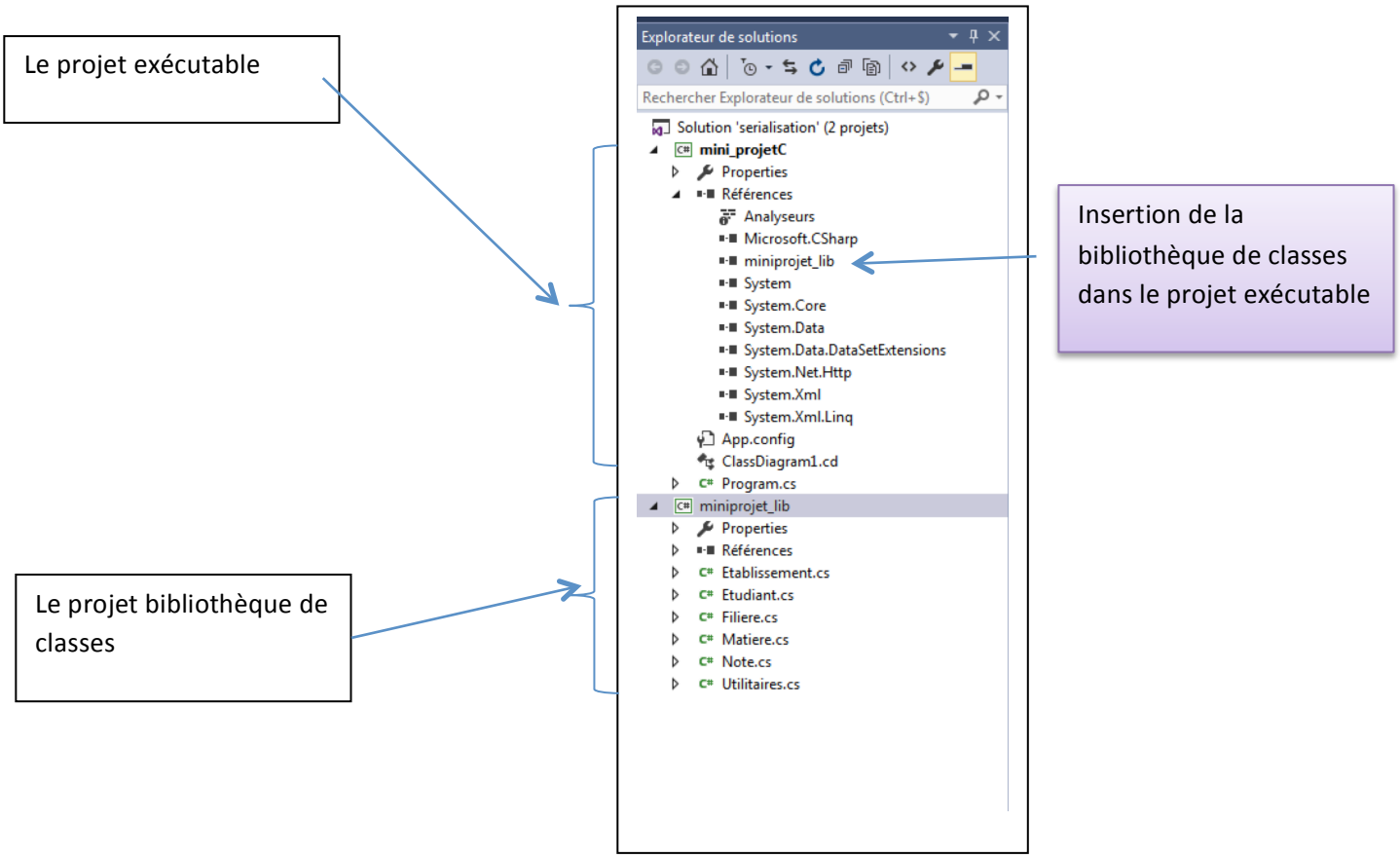
On va pouvoir utiliser les notions d'héritage pour les deux classes Etudiant et Enseignant qui vont pouvoir hériter de Personne.

```
namespace miniprojet_lib
{
    public class Personne
    {
       public Personne()  {}
       public Personne( …..) {
       }
       public int id {get; set; }
       public string Nom {get; set; }
       public string Preom {get; set; }
       public string Email {get; set; }
       public DateTime DateNaissance {get; set; }



    }
```

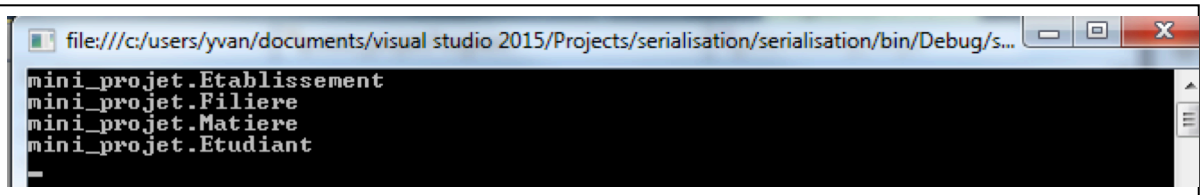
Utilisation de la classe personne en héritage pour etudiant et enseignant.

```
namespace miniprojet_lib
{
    public class Etudiant : Personne
    {
       public string Num_etudiant { get; set; }
    }
}

namespace miniprojet_lib
{
    public class Enseignant: Personne
    {
       public string Num_employe { get; set; }
    }
}
```

Pour utiliser notre modèle, nous allons rajouter à la solution un nouveau projet cette fois-ci en mode console, qui va produire un exécutable. Pour permettre à ce nouveau projet d'utiliser les classes développées dans la DLL il faut lui ajouter une référence supplémentaire sur miniprojet_lib.



```
using miniprojet_lib;

namespace mini_projet
{
    class Program
    {
        static void Main(string[] args)
        {
            Etablissement etablissement = new Etablissement();
            etablissement.Nom = "EPSI Paris";
            etablissement.Adresse = "1 place saint Cyran";
            etablissement.Telephone = "01 49 45 47 45";
            Console.Out.WriteLine(etablissement) ;

            Filiere fil1 = new Filiere();
            Fil1.Intitule = "DEV";
            Fil1.Description="Filière de préparation Ing Devops";
            Filiere fil2 = new Filiere();
            Fil2.Intitule = "NETWORK";
            Fil2.Description="Filière de préparation Ing Réseau ";


            Console.Out.WriteLine(filiere) ;
            Matiere matiere1 = new Matiere("Mathematique");
            Matiere matiere2 = new Matiere("Anglais");
            Console.Out.WriteLine(matiere1) ;
            Etudiant etudiant =new  Etudiant("Stroppa", "yvan", "yvan.stroppa@gmail.com");
            Console.Out.WriteLine(etudiant) ;
            Console.ReadKey() ;
        }
    }
}
```



On peut constater lors de l'affichage des objets que le système affiche l'arborescence des classes utilisées, c'est-à-dire le namespace dans lequel elles sont définies et le nom de la classe.

Dans ce cas précis, le système utilise une fonction membre ToString attachée à OBJECT (objet Top level) pour appliquer cet affichage.

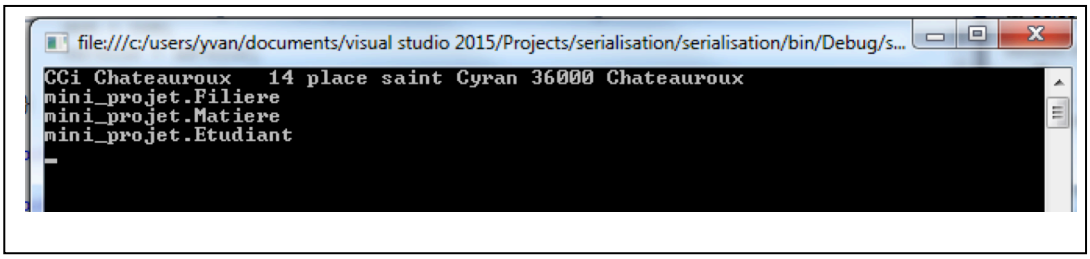
Pour afficher les éléments de façon plus représentative il faut que les classes SURCHARGENT la fonction membre ToString et définissent ainsi le comportement qu'elle doit avoir, ce qui se traduit par l'ajout dans chaque classe. Soit par exemple pour Etablissement :

Aménagement de la classe Etablissement

```
override
public string ToString()
{
    return Nom + "    " + Adresse;
}
```

SURCHARGE

Exécution du programme après modification :



Il faut bien sûr effectuer la même opération pour les autres classes.

Analyse des associations entre les différentes classes

Notre modèle jusqu'à maintenant est incomplet, en effet, il manque les associations entre les différents objets, par exemple, un étudiant est affecté à une filière, une filière appartient à un établissement, un étudiant va recevoir des notes dans le cadre de sa participation à des matières.

Il faut formaliser et créer des structures d'accueil dans les classes pour toutes ces informations supplémentaires. On va étudier association par association.

Analyse de l'association entre établissement et filière

Il faut étudier les associations entres les classes et surtout définir les éléments qui vont devoir porter ces relations.

Tout d'abord, commençons par Etablissement, un établissement est constitué de plusieurs filières. Si on effectue la représentation sous forme UML du diagrammes des classes :

Ce qui se traduit : par un établissement peut avoir 0 à n filières, et une filière est attachée à un seul établissement. La navigation de cette association est bidirectionnelle, car il n'y a aucun sens indiqué.

Pour la classe Etablissement

Etant donné que l'association est multiple d'établissement vers filières, il faut ajouter un nouvel attribut dans Etablissement qui permette de porter cette association. On va choisir une collection de type List qui va contenir des références sur les filières. Et il faudra ajouter les fonctions membres qui vont permettre de manipuler ce nouvel attribut.

```
Classe Etablissement

       private List<Filiere> Liste_filieres = new List<Filiere>();
        public List<Filiere> _Liste_filieres
        {
            Get { return Liste_filieres; }
            set { Liste_filieres = value;}
        }
        public void ajoute_filiere(Filiere fil)
        {
            if (!Liste_filieres.Contains(fil))
            {
                Liste_filieres.Add(fil);
            }
        }

        public void supprime_filiere(Filiere fil)
        {
            if (Liste_filieres.Contains(fil))
            {
                Liste_filieres.Remove(fil);
            }
        }
```

Pour la classe Filiere

Etant donné que l'association dans ce sens est au maximum vers un établissement, il nous suffit d'ajouter un attribut qui va la porter et qui va être du type Filiere. Il faut aussi ajouter les fonctions membres permettant la manipulation de cet attribut.

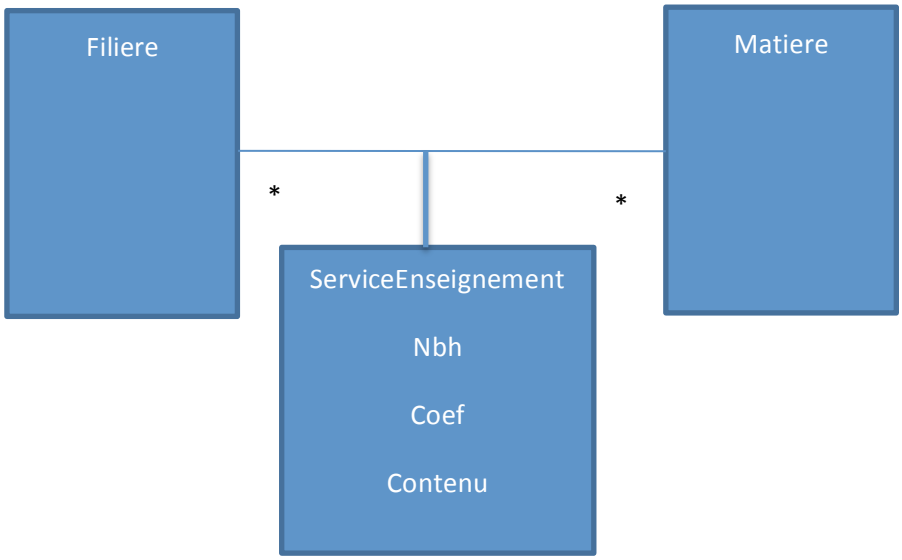
```
Classe filiere

        public Etablissement Etablissement { get; set; }
```

Adaptation de notre programme de tests pour ajouter des deux côtés l'association.

```
            etablissement.ajoute_filiere(filiere);
            filiere.Etablissement = etablissement;
            Console.Out.WriteLine(etablissement);
            Console.Out.WriteLine(filiere);
```

Pour des raisons de clarté et de simplicité, nous avons allégé l'association, maintenant on peut évoquer des aspects complémentaires à ce type d'association et la remplacer par une liaison plus contraignante qui est la composition. Ce qui implique qu'une filière ne peut exister que dans le cadre d'un établissement.

Analyse de l'association entre filière et Matière



L'association entre matière et filière va générer un service d'enseignement qui permettre de définir le contenu de la matière pour la filière, son nombre d'heures et le coefficient. On lui rattachera pas la suite l'enseignant qui va le dispenser. Donc une matière peut être en relation plusieurs avec une filière. Exemple pour la matière Mathématiques on peut avoir pour une même filière un service d'enseignement en géométrie de 10h et de 3 en coefficient et un service d'enseignement Algèbres Linéaires de 15h et de 4 en coefficient.

Dans ce cas, c'est la filière qui va porter les services d'enseignements avec toutes leurs caractéristiques … et qui permettra ensuite de les attacher à des enseignants. Donc dans filière on ajoute l'attribut listes_services qui sera constitué de service généré dès que l'on établit une association entre matière et filière.

Dans ce cas, nous sommes en présence d'une association de plusieurs vers plusieurs, donc pour implémenter cette association, nous allons devoir ajouter dans chacune des classes un attribut supplémentaire qui devra permettre de porter cette association. Ce nouvel attribut est du type List. Il faudra également définir les méthodes qui permettent la manipulation de cette association.

Dans la classe Filiere

```
        private List<Service> _Liste_services = new List<Service>();
        public List<Service> Liste_services { get => _Liste_services; }
        public bool ajoute_matiere(Matiere matiere, int Nb_h, int Coef, string
Descipt) {

            Service service= new Service(){mat=matiere, fil=this,descriptif=Descipt};
            if (!_Liste_services.Contains(service)) {
```

Le but de cette méthode ajoute_matiere, c'est de créer un objet service et de l'ajouter dans l'attribut _Liste_services de Filiere pour le mémoriser. Ce service permettra par la suite d'associer un enseignant pour le dispenser et un lien avec les étudiants pour les notes.

Il faudra à cette méthode en ajouter d'autres comme :

Supprimer_service

Update_service

Afficher_service

Vérifier_service_affectation

Donc il faudra encore ajouter d'autres attributs et méthodes à cette classe pour gérer toutes ses associations.

Dans la classe Service

```
[Serializable]
 public class Service: IEquatable<Service>, IComparable<Service> {
      public int id {get; set; }
      public Filiere fil {get; set; }
      public Matiere mat{get; set; }
      public int  coef{get; set; }
      public int nb_h  {get; set; }
      public string descriptif {get; set; }
      public Enseignant affectation {get; set; }
      public int Codeid {get; set; } // identification du service
      public bool Equals([AllowNull] Service other)
      {
         return (this.mat==other.mat && this.fil==other.fil &&
this.descriptif==other.descriptif);
      }
      public int CompareTo([AllowNull] Service other)
      {
         return this.descriptif.CompareTo(other.descriptif);
      }
      override
      public string ToString() {
            return "Service : " + this.Codeid + "   " + this.descriptif;
      }
   }
```

Utilisation à partir du programme principal :

```
            filiere.ajoute_matiere(matiere1, 10,3,"Géométrie");
            filiere.ajoute_matiere(matiere1, 10,4,"Algèbre");
```

On peut ajouter dans ajoute_matiere de filière.

Pour améliorer l'affichage, vous pouvez ajouter cette méthode dans la classe filiere

Dans la classe filiere

```
        public string affiche_service()
        {
            string chaine = "Filière "+ this.descriptif + "\n";
            foreach (Service serv in _Liste_services)
            {
                chaine += "\t\t" + serv.CodeID + "\n" +
serv.mat.intitule …. ;
            }
            return chaine;
        }
```

Nous avons utilisé la méthode de Contains de la classe List et de la classe Dictionary. Cette méthode a pour objectif de vérifier si un élément est déjà présent dans la collection. Comment fait-elle pour effectuer cette vérification. Lorsque l'on a un élément de type simple (int, float ou string) la comparaison est immédiate et le système peut vérifier qu'au qu'un élément est déjà enregistré dans la collection on le comparant avec le nouvel élément. Par contre lorsque c'est une collection d'objets le système permet de vérifier qu'un objet est déjà présent en vérifiant l'OID de l'élément. Est-ce que c'est le même objet :

Exemple :

```
Matiere mat1 = new Matiere("Info", "Informatique");

Matiere mat2 = new Matiere("Math", "Mathématique");

Matiere mat3 = new Matiere("Anglais", "Anglais");

filiere.ajoute_matiere(mat1,12,2,"Langage C#");

filiere.ajoute_matiere(mat2,12,2,"Langage Java");
```

Si on souhaite ajouter une deuxième fois la même matière, le programme doit l'empêcher :et émettre une erreur ou un avertissement.

```
filiere.ajoute_matiere(mat1, 12,2,"Langage C#");
```

La matière existe déjà

Par contre, dans la situation suivante où on génère une nouvelle matière Matiere mat4 = new Matiere("Info", "nouvelle matière informatique"); et que l'on essaye de l'ajouter, le système va l'autoriser car le contrôle de la présence ne s'appuie que sur l'OID de l'élément et dans ce cas les OID sont différents. Comment faire ?

Pour éviter cette situation il faut doter la classe d'une interface de type IEquatable<Matiere> qui va obliger le développeur à implémenter la méthode Equal et permettre aux collections de savoir comment comparer deux éléments entre eux.

```
public class Service : IEquatable<Service>

…

        public bool Equals(Service other)

        {

                return this.mat.Equals(other.mat) && this.mat.Equals(other.mat) &&

        this.Descriptif.Equals(other.Decriptif);

        }

}
```

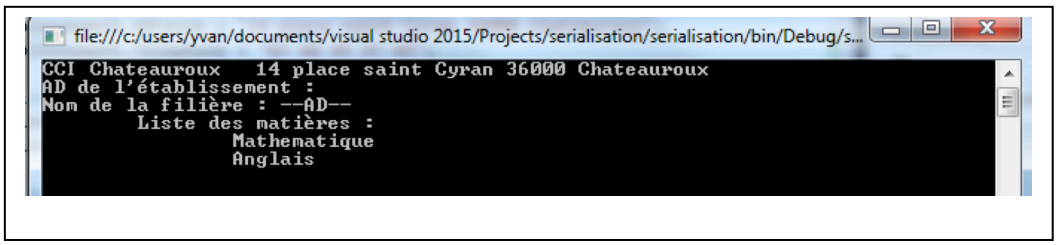
La comparaison s'effectue à partir de la matière, de la filière et de la description

On pourra faire la même chose pour Filiere.

Dans le programme principal :

```
Console.WriteLine(filiere.affiche_detail());
```

La sortie du programme nous donnera :



Une fois que l'on a manipulé les différents éléments, il faudrait penser à les préserver pour cela nous allons utiliser la sérialisation.

La sérialisation

Intérêt de la sérialisation : être capable de manipuler une structure d'objets avec l'ensemble de ses relations, associations de façon simple et complète. Dans le cadre d'une sauvegarde ou de transfert réseau.

Pour ce faire, il faut dans un premier temps aménager les classes avec une directive indiquant au système les fonctionnalités de sérialisations. Il faut faire précéder les déclarations de classe par le mot clé [Serialization]

Ce qui nous donne pour les quatre classes :

```
[Serializable]
    public class Etablissement
```

```
[Serializable]
    public class filiere
```

```
[Serializable]
    public class Etudiant
```

```
[Serializable]
    public class Matiere
```

Ensuite on va pouvoir utiliser la puissance de cette sérialisation. Tous les éléments crées sont attachés à l'objet Etablissement. Il suffira d'ouvrir un descripteur sur une source fichier et d'écrire le contenu de l'objet. La sérialisation fera le reste et ira écrire tous les objets attachés à cette référence.

Ce qui se traduit par :

On ajoute une classe nommée Utilitaires avec deux méthodes statiques :

Une méthode sauve() ;

Une méthode restaure() ;

```
namespace miniprojet_lib
{
    public abstract class Utilitaires
    {
        public static Boolean  sauve(Object obj)
        {
            IFormatter formatter = new BinaryFormatter();
            Stream stream = new FileStream("myfile.dat", FileMode.Create, FileAccess.Write,
FileShare.None);
            formatter.Serialize(stream, obj);
            stream.Close();
            return true;
        }
        public static object  restaure()
        {
            IFormatter formatter = new BinaryFormatter();
            Stream stream = new FileStream("myfile.dat", FileMode.Open, FileAccess.Read,
                                           FileShare.Read);
            Object obj = formatter.Deserialize(stream);
            stream.Close();
            return obj;
        }
    }
}
```

Pour utiliser cet ensemble, il faut ajouter une nouvelle méthode d'affiche détail dans établissement.

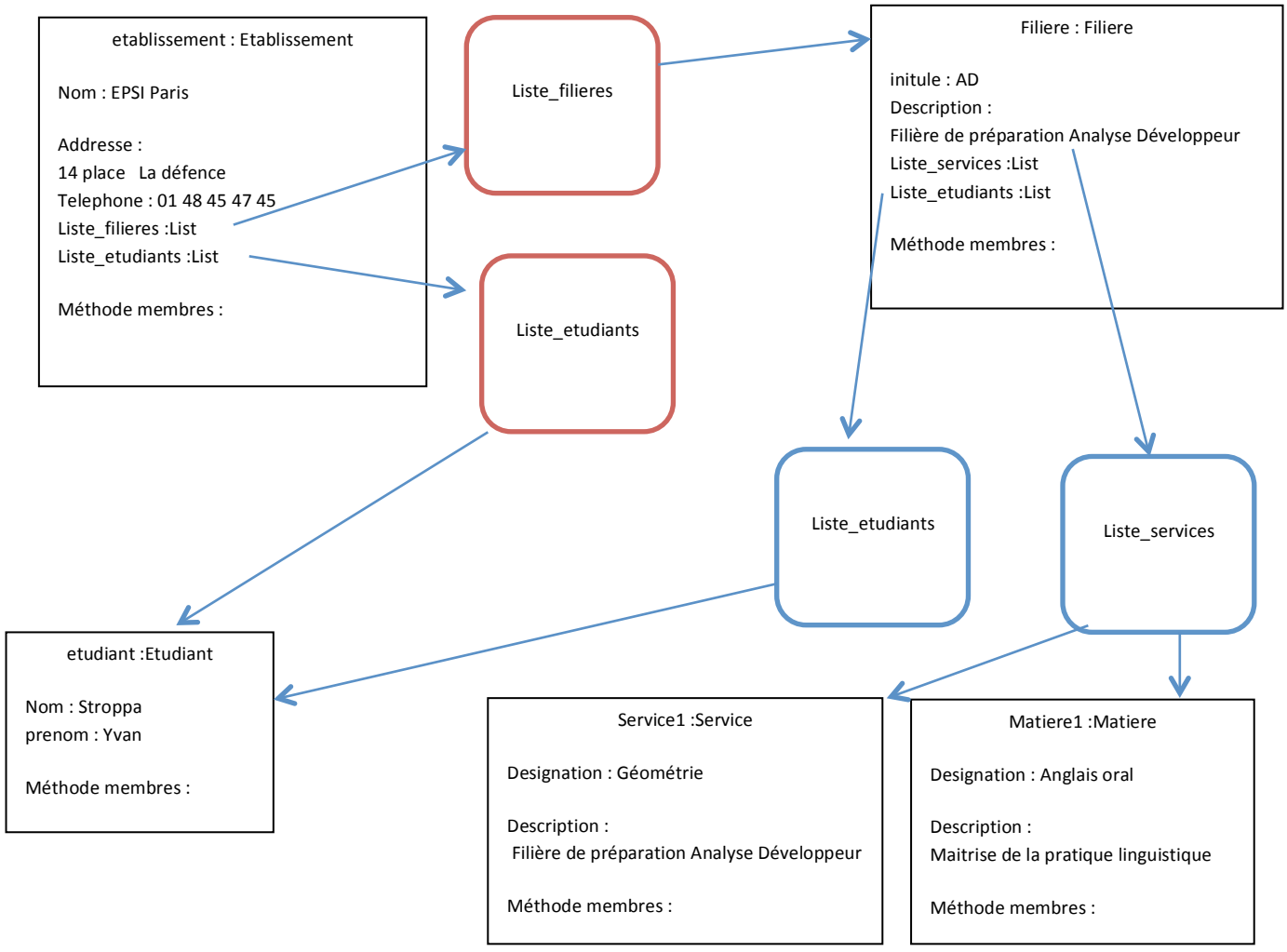
```
Dans la classe Etablissement

        public string affiche_detail()
        {
            string chaine = "";
            chaine+=  Nom + "    " + Adresse + "\n" ;
            chaine += "les filières : \n";
            foreach(Filiere fil in Liste_filieres)
            {
                chaine += fil.affiche_detail_filiere();
            }
            return chaine;
        }
```

Dans le programme principal

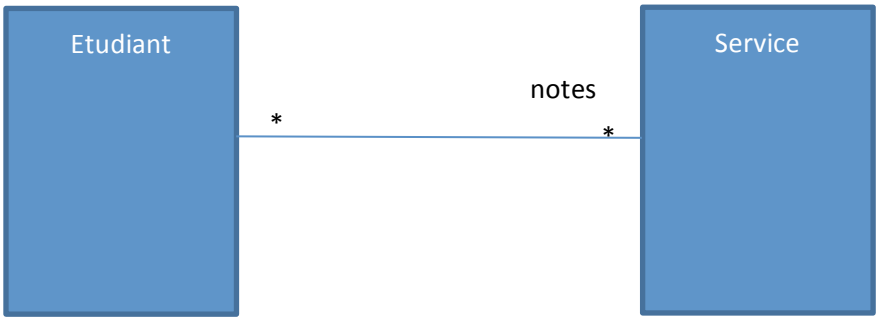
```
Dans le programme principal

            // serialisation
            Console.WriteLine("sauvegarde de l'établissement");
            Utilitaires.sauve(etablissement);*/

            // deserialisation
            Etablissement etabl1 = (Etablissement) Utilitaires.restaure();
            Console.WriteLine(etabl1.affiche_detail());
```

Le résultat est que l'on a préservé dans le fichier myfile.dat sous format binaire l'ensemble des éléments saisis.

Représentation en mémoire de la structure de données :



Ajout de la dernière liaison entre les étudiants, matières ce sont les notes. Il faut formaliser cette association :



Un étudiant est attaché à une filière et parce qu'il est attaché à cette filière il aura des notes pour les différentes matières (services) qu'il suit dans le cadre de cette filière.

Adaptation des classes

```
Dans la classe Etudiant
        // attributs
        private Dictionary<Service, List<Note>> _Notes = new Dictionary<Service,
List<Note>>();

        // propriétés
        public Filiere filiere { get; set; }
        public Dictionary<Service, List<Note>> Notes
        {
            get {return _Notes; }
            set { _Notes = value;}
        }

public Boolean ajoute_notes(string intule, Note note)
        {
        Service service= fil.Liste_services.Find(x => x.descriptif==intitule );
        if (service ==null) {
            Console.WriteLine("L'intitulé de la matière demandé n'existe pas");
            return ;
        }
        if (!_Notes.ContainsKey(service) ) {
            Console.WriteLine("ajouter une nouvelle note");
            List<Note> Lnotes= new List<Note>();
            Lnotes.Add(n);
            _Notes.Add(service, Lnotes);
        } else {
            Console.WriteLine("Complete des notes pour une matiere existante");
            _Notes[service].Add(n);
        }
```

Eviter d'utiliser dans le if Notes[mat] != null car pour extraire une valeur il faut qu'elle existe. Donc on va préférer Notes.ContainsKey pour effectuer cette opération

En argument on a pris l'intitulé mais on aurait pu prendre le code du service …

Dans la classe Etudiant

```
    // on balaye la listes des notes pour effectuer l'affichage
    public string afficher_notes() {
       string chaine="";
       foreach(Service service in _Notes.Keys) {
            chaine +="Matiere: "+ service.descriptif + " \n" ;
            chaine +="Notes"+ " \n" ;
            foreach (Note n in _Notes[service]) {
                chaine +="\t Note"+ n.date + "   " + n.note  +" \n" ;
            }
        }
        return chaine;
    }
```

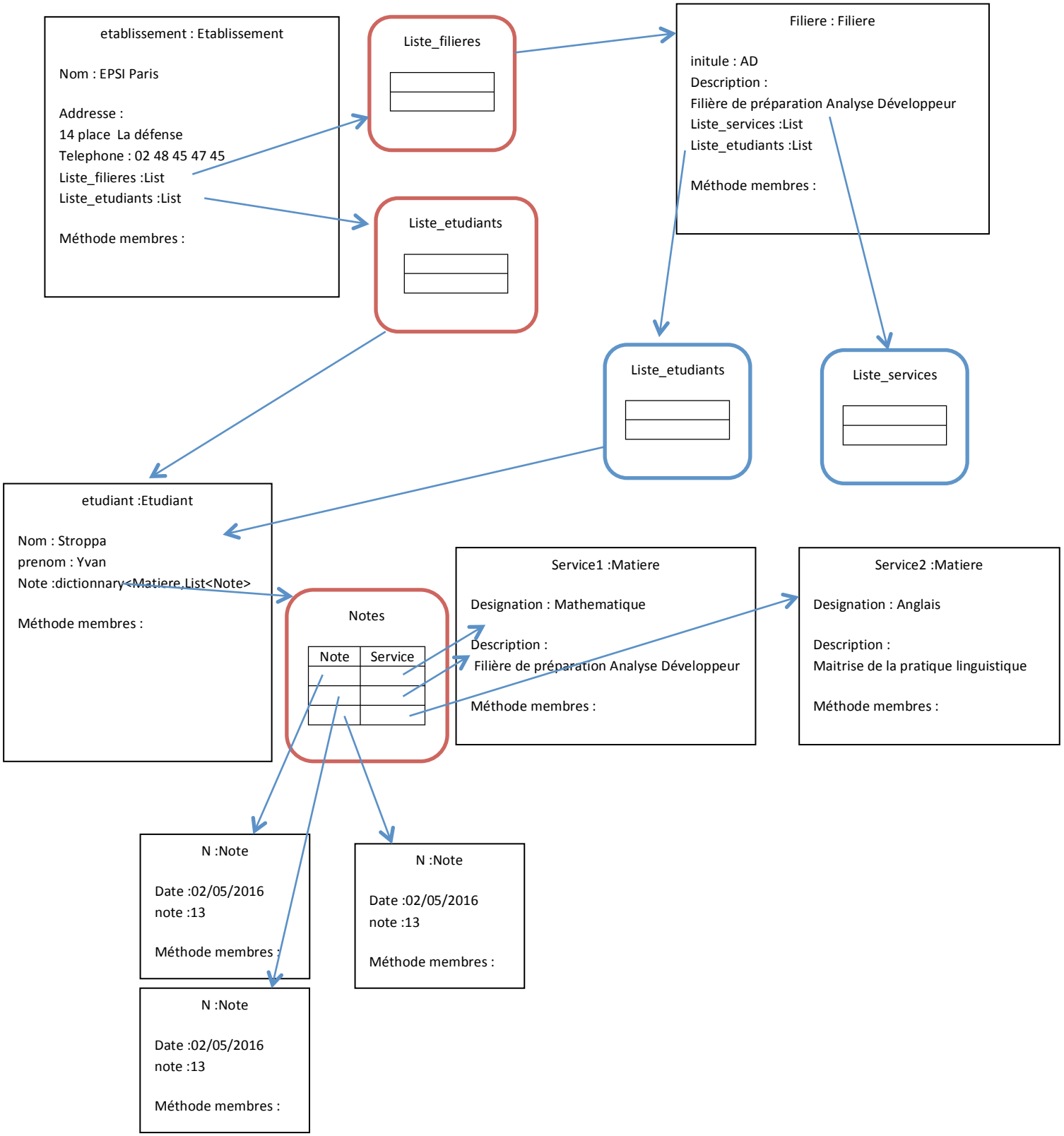
Ajout d'une nouvelle classe :

```
namespace miniprojet_lib
{
    [Serializable]
    public class Note
    {
        public float note { get; set; }
        public DateTime date { get; set; }

    }
}
```

Utilisation de ces fonctions dans le programme principal

```
      // partie note étudiant
      etudiant.ajoute_notes("Langage C#", new Note(10, new DateTime(2016,04,13)));
      etudiant.ajoute_notes(("Langage Java",, new Note(12, new DateTime(2016,04,14)));
      etudiant.ajoute_notes(("Routeur", new Note(13, new DateTime(2016,05,16)));

      // affichage liste des notes par matières pour un etudiant
      Console.WriteLine(etudiant.liste_notes());
```

Ce qui nous donne le nouveau schéma suivant :



Quelques notions supplémentaires :

Notion de classe abstraite

Une classe abstraite ne pas être instanciée par contre une classe peut hériter d'une classe abstraite.

Notion de static

La notion static pour une fonction membre, est une méthode qui est attachée à la classe et non pas à l'objet. Donc on peut l'invoquer en donnant le nom de la classe. Elle ne nécessite pas d'instancier un objet pour l'utiliser.

Pour une variable « Static » : la variable est associée à la classe et non pas à objet, elle permet par exemple de compter le nombre d'objets d'une classe « sorte de compteur ».

Notion d'exception

La notion d'exception : comment lever une exception et gérer un e levée d'exception. Les exceptions permettent de traiter des situations non conforme ou non souhaitée. Exemple j'essaye d'ouvrir un fichier qui n'existe pas, j'essaye de me connecter à une source de données mais je n'ai pas de réseau, j'essaye d'introduire un élément dans une collection, mais il existe déjà. On peut émettre des message d'avertissement sur la console ou en retour d'appel de méthode mais cette solution est mal adaptée et plus difficile à traiter (il faut traiter des string ou des entiers …et si on change il faut recommencer). Pour ce faire, nous avons les exceptions qui vont permettre des objets et des situations particulières.

Donc les zones à risque (pouvant levée une exception sont encadrées par un try { } et catch (Exception} l'instruction try permet d'indiquer au programme qu'il peut avoir un e exception dans l'exécution des instructions dans son bloc et l'instruction catch permet de traiter et d'adapter la situation.

Déclarée une classe de type exception pour le contrôle de la note. Une note ne doit pas être supérieure à 20 et inférieure à 0.

On va déclarer une nouvelle classe nommée InvalidNoteException qui aura l'allure suivante :

```
[Serializable()]
   public class InvalidNoteException : System.Exception
   {
     public InvalidNoteException() : base() { }
     public InvalidNoteException(string message) : base(message) { }

   }
```