Concepteur Développeur en Informatique

Développer des composants d'interface

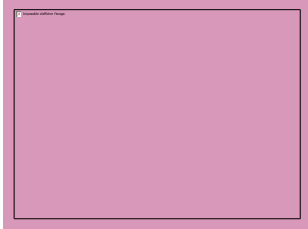
POO – Interfaces et IOC

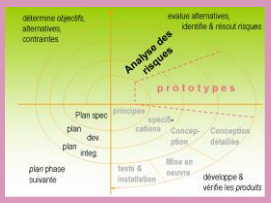
Accueil

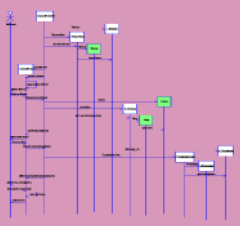
Apprentissage

PAE

Evaluation







Localisation : Module 1 – Objet partie 2

SOMMAIRE

1. Introduction 3
1.1. Implémentation des interfaces : 4
2. Illustration au niveau de nos objets métier 6
3. Inversion de dépendance 8
3.1. Les différentes formes de la dépendance 8
3.2. Principe de l'inversion de dépendance 8
3.3. Un exemple concret d'usage de l'inversion de dépendance. 9

1. Introduction

Nous allons poursuivre dans ce document l'apprentissage de la programmation objet avec la conception et l'implémentation d'interfaces.
Ces classes particulières qui n'implémentent pas les membres (propriété, méthode ou événement) mais uniquement les signatures de ces derniers.

Une interface peut être un membre d'un espace de noms ou d'une classe, et peut contenir les signatures des membres suivants :

- Méthodes
- Propriétés
- Indexeurs
- Événements

Nous traiterons des indexeurs dans un prochain chapitre.

Une **interface** peut hériter **d'une ou de plusieurs interfaces de base**.

Une **classe** peut hériter **d'une ou plusieurs interfaces**.

Ce mécanisme permet donc de mettre en œuvre **l'héritage multiple** qui n'est pas envisageable dans le cadre de l'héritage de classes.
On ne peut en effet hériter de plus d'une classe de base (parente).

Une interface ne contient donc pas le code des opérations ni les données mais précise uniquement les méthodes et propriétés qu'une classe héritant de l'interface devra fournir.

A la différence de l'héritage des classes abstraites, **toutes** les propriétés et méthodes d'une interface héritée doivent être implémentées.

Les interfaces permettent de définir des **comportements similaires** à l'ensemble des objets qui les implémentent.

Les traitements qui seront exécutés par les objets implémentant ces interfaces sont toutefois **spécifiques** à chaque objet et le code implémenté diffère donc d'une classe à une autre.

L'intérêt du recours aux interfaces est de s'assurer de disposer d'un code correctement structuré, de fournir des mécanismes standardisés et uniformes, et d'assurer l'évolutivité du système.

Par convention, les interfaces sont toujours préfixées d'un I majuscule.

Syntaxe d'une interface :

Exemple d'une interface proposant une méthode :

```
interface IComparable
{
    bool CompareTo(object objet);
}
```

Définition d'une interface

```
interface IPersonne : IComparable
{
    string Nom { get; set; }
    string Prenom { get; set; }
    DateTime DateNaissance { get; set; }
    string Sexe { get; set; }
    double Age { get; }
}
```

La restriction essentielle d'une interface est qu'elle ne doit jamais comporter d'implémentation, ce qui implique :

- Pas d'utilisation de champs relatifs aux propriétés (variables locales _n)
- Aucun constructeur, ni destructeur
- Pas de modificateur d'accès : les membres d'une interface sont toujours publics.
- Une interface peut seulement hériter d'une interface. Elle ne peut hériter d'une classe ou d'une structure

Mixité d'héritage : Si une classe est dérivée d'une classe de base et d'interface, la classe de base doit apparaître en premier dans la liste.

1.1. Implémentation des interfaces :

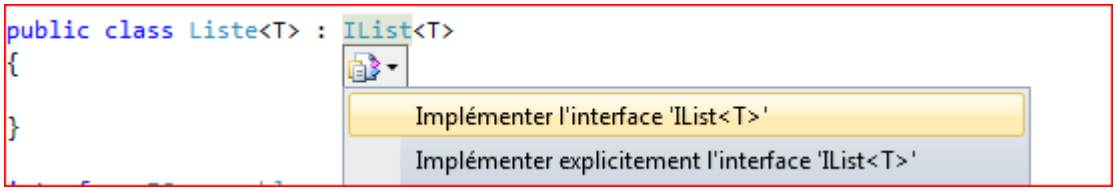
Une classe qui **implémente** une interface **doit implémenter tous les membres** de l'interface.

L'héritage d'interface est une technique qui permet de s'affranchir de la limite de l'héritage de classe : Une classe dérivée (ou fille) ne peut hériter que d'une classe au plus alors qu'une classe peut hériter de plusieurs interfaces.

L'héritage par le biais d'interfaces est un principe couramment mis en œuvre dans les classes fournies par Microsoft au niveau de l'architecture logicielle Dot Net. Nous pourrons donc recourir le plus souvent à l'héritage d'interfaces existantes au niveau du Framework pour compléter la définition de nos objets.

L'exemple suivant montre comment créer son propre type de liste générique à partir d'une de l'interface IList<T>.

La première chose à faire est de déclarer sa classe et d'indiquer que nous souhaitons qu'elle hérite de l'interface IList. Puis de sélectionner l'interface IList<T> et de demander son implémentation (menu contextuel sur click droit) :



Les différents membres de l'interface, propriétés et méthodes, sont alors générés. Il ne vous reste plus qu'à implémenter le code de ces méthodes.

Extrait de l'implémentation de IList<T> :

```
public int IndexOf(T item)
{
    throw new NotImplementedException();
}

public void Insert(int index, T item)
{
    throw new NotImplementedException();
}

public void RemoveAt(int index)
{
    throw new NotImplementedException();
}

public T this[int index]
{
    get
    {
        throw new NotImplementedException();
    }
    set
    {
        throw new NotImplementedException();
    }
}
```

Le corps des méthodes ne peut bien entendu pas être généré car il est nécessairement propre au contexte d'implémentation.

Vous reconnaîtrez quelques méthodes déjà vues lors de la manipulation de listes.

Vous voyez aussi un mécanisme nouveau, l'indexeur de propriété this[int index] qui permet d'indexer les éléments d'une classe à la manière d'un tableau.

Nous pouvons donc réaliser relativement aisément notre propre type de liste pour répondre à une problématique particulière.

Nous pourrons, via le recours aux interfaces, assurer à notre liste un comportement similaire à toute liste : c'est bien le but des interfaces.

2. Illustration au niveau de nos objets métier

Nous souhaitons mettre en place un mécanisme de prise en charge de la persistance au niveau de toutes nos collections d'objets « métier ».

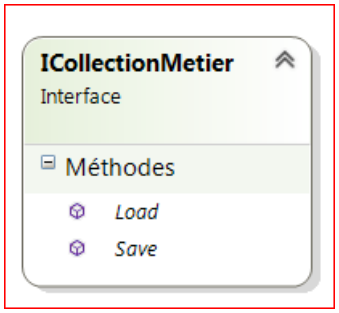
Ce mécanisme reposera sur deux opérations :

- Chargement de l'état de nos objets à partir d'un document
- Ecriture de l'état de nos objets dans un document

La solution : La conception d'une interface.

Ici, j'ai créé une seule interface ICollectionMetier qui définira deux méthodes :

- Une pour la sauvegarde
- Une pour le chargement



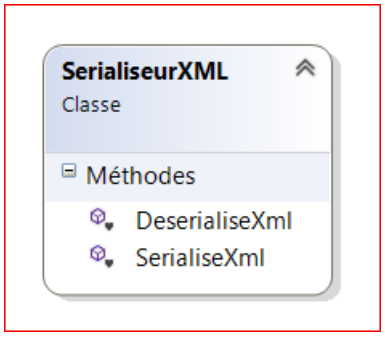

```
3 références
interface ICollectionMetier
{
    9 références
    void Save(string pathRepData);
    8 références
    void Load(string pathRepData);
}
```

Dans quelle mesure ce choix de conception peut-il être considérée discutable ?

Pour extraire les données ou les sauvegarder, nous devons faire appel à un mécanisme de sérialisation ou de traitement de base de données (nous y reviendrons ultérieurement).

Si nous ne passons pas cet objet en référence, nous allons instancier celui-ci dans chaque méthode interne de notre objet et donc, non seulement rendre très fortement dépendant nos objets métier du mécanisme de sauvegarde retenu mais nous n'aurons même pas une connaissance aisée de cette dépendance (masquée par l'encapsulation).

Exemple : Nous avons un objet XML qui permet de sérialiser et désérialiser nos objets métiers.




```
public void SerialiseXml(string pathRepData, object objetASerialiser)
{
    Type type = objetASerialiser.GetType();
    string pathXmlDocument = string.Format("{0}\\{1}.Xml", pathRepData, type.Name);
    using (FileStream fileStream = new FileStream(pathXmlDocument, FileMode.Create,
        FileAccess.Write, FileShare.Read))
    {
        XmlTextWriter xmlTW = new XmlTextWriter(fileStream, Encoding.UTF8);
        XmlSerializer xmlS = new XmlSerializer(type);
        xmlS.Serialize(xmlTW, objetASerialiser);
        xmlTW.Close();
        fileStream.Close();
    }
}
```

Ainsi, la méthode implémentée au niveau de nos classes métiers prendrait la forme suivante :

```
public void Save(string pathRepData)
{
    SerialiseurXML sXML = new SerialiseurXML();
    sXML.SerialiseXml(pathRepData, this);
}
```

Mieux serait donc de passer la référence de l'objet prenant en charge la sauvegarde aux méthodes Save et Load. Modifions notre interface en conséquence.

```
interface ICollectionMetier
{
    2 références
    void Save(SerialiseurXML serialiseur, string pathRepData);
    1 référence
    void Load(SerialiseurXML serialiseur, string pathRepData);
}
```

Nous aurions ensuite ainsi une méthode Save dans notre classe métier qui prendrait la forme suivante :

```
public void Save(SerialiseurXML serialiseur, string pathRepData)
{

    serialiseur.SerialiseXml(pathRepData, this);
}
```

Cette solution est-elle tout à fait satisfaisante ?

Non ! C'est un peu mieux mais nous avons toujours une forte dépendance entre notre objet concret qui permet d'effectuer les sauvegardes et nos objets métier.

Si demain, nous envisageons un autre support de sauvegarde, nous devrons reprendre toutes les méthodes Save et Load de chaque objet métier.

Il n'est pas rare de manipuler plusieurs centaines d'objets métier au sein d'un système d'information d'entreprise.

La solution pour obtenir un couplage plus faible est de définir une dépendance vers un type abstrait. **Nous exprimerons ainsi le principe de dépendance** :

- Chaque collection d'objets métier sera dépendante d'un dispositif de sauvegarde.
- Ce dispositif de sauvegarde devra permettre de sauvegarder les objets métiers et de les restaurer.
- Le dispositif de sauvegarde pourra s'appuyer sur différentes techniques, techniques choisies dans l'application qui utilisera les objets métier.

Comment mettre en œuvre cette solution ?

Nous allons nous appuyer sur un patron de conception très usité, celui de l'inversion de dépendance, en anglais IOC (Inversion Of Control). Je vous le présente au chapitre suivant.

3. Inversion de dépendance

Nous allons découvrir ici 1 design pattern (patron de conception) utilisant la mise en œuvre d'interfaces pour un seul objectif, réduire la dépendance entre deux objets, ou par extension, deux couches de composants. Nous y reviendrons dans le cadre de nos développements au sein d'architectures n-tiers en couches.

Il s'agit du design pattern IOC : Inversion Of Control.

3.1. Les différentes formes de la dépendance

En programmation objet, les objets de type **A** dépendent d'un objet de type **B** si au moins une des conditions suivantes est vérifiée :

- Un objet de type **A** est de type **B** : *dépendance par héritage*
- Un objet de type **A** possède un attribut de type **B** : *dépendance par composition*
- Un objet de type **A** dépend d'un objet de type **C** qui dépend d'un objet de type **B** : *dépendance par transitivité*
- une méthode de **A** prend en argument d'une méthode un objet de type **B**.

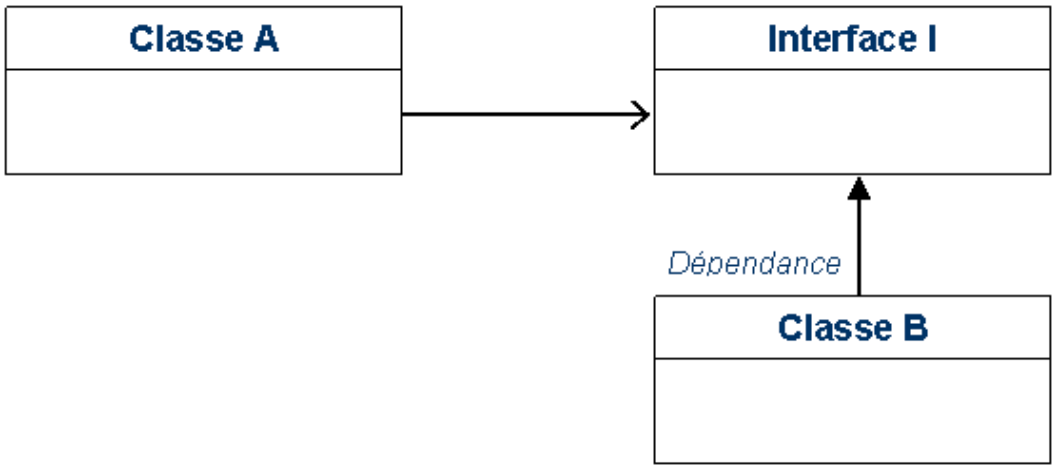
Si **A** dépend de **B**, cela implique que pour créer **A**, on ait besoin de **B** ou que pour mettre en œuvre un traitement propre à **A** il nous faille avoir une instance de **B**.

3.2. Principe de l'inversion de dépendance

L'objectif est de découpler les liens de dépendances entre deux objets figurés ici par les objets A et B.
Pour supprimer la dépendance entre ces deux objets, un moyen possible consiste à créer une interface **I**.

Cette interface contiendra toutes les propriétés et méthodes qu'**A** peut appeler sur **B**.
B implémentera l'interface **I**.

Toutes les références au type **B** seront alors remplacées par des références à l'interface **I** dans **A**.



Il n'y a plus alors de dépendance entre A et B, mais une dépendance entre B et I.
On peut alors remplacer B par B' pour peu que B et B' proposent les mêmes services.

3.3. Un exemple concret d'usage de l'inversion de dépendance.

Reprenons notre exemple avec le dispositif de sauvegarde.

Plutôt que de rendre nos composants métiers dépendants d'un mécanisme de sauvegarde concret, nous allons les rendre dépendants d'une abstraction, fournie sous la forme d'une interface, qui devra prévoir deux comportements :

- Un comportement figurant l'opération de sauvegarde
- Un comportement figurant l'opération de restauration

Créons cette interface dans un assemblage particulier Utilitaires :

```
11 références
public interface ISauvegarde
{
    4 références
    void Save(string pathRepData, IEnumerable objetASauvegarder);
    4 références
    IEnumerable Load(string pathRepData, Type typeACharger);

}
```

Nous allons inverser le contrôle entre la classe Métier et le composant de Sauvegarde.

Au niveau du composant métier :

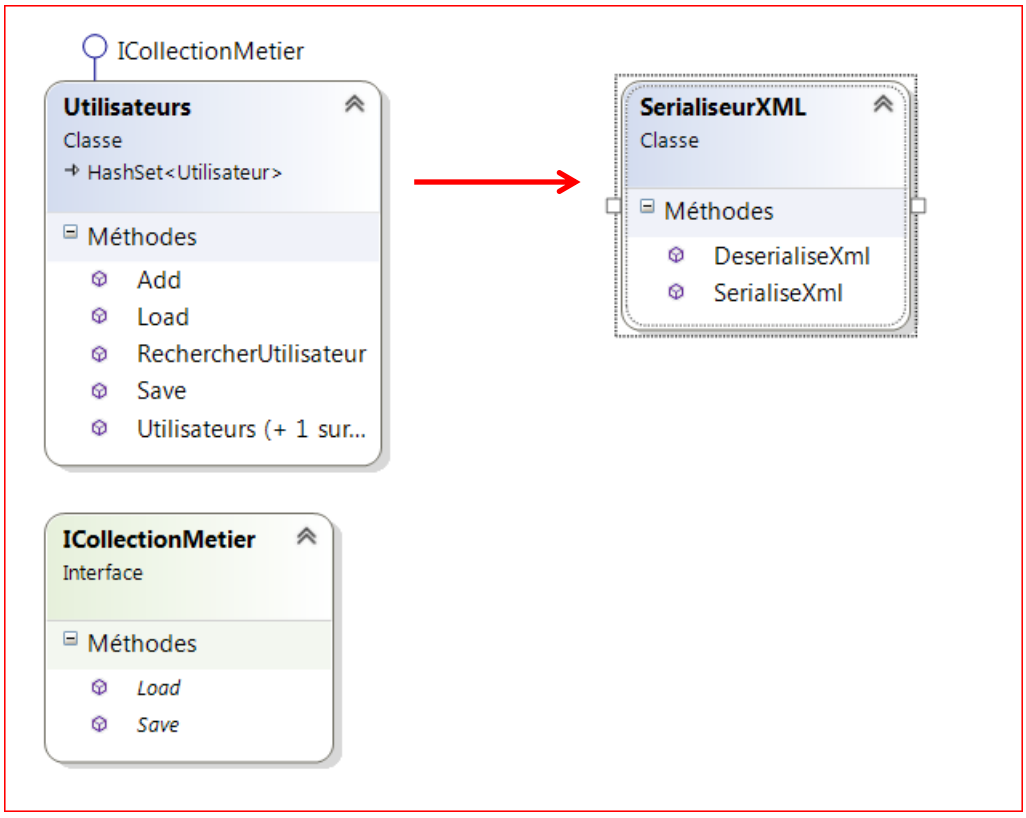
```
3 références
public void Save(Utilitaires.ISauvegarde sauvegarde, string pathRepData)
{
    sauvegarde.Save(pathRepData, this);
}
```

Cette interface devra être implémentée dans tous les mécanismes de sauvegarde concrets. Nous élaborons ici le mécanisme qui permet de sauvegarder les objets métier dans un fichier au format XML :

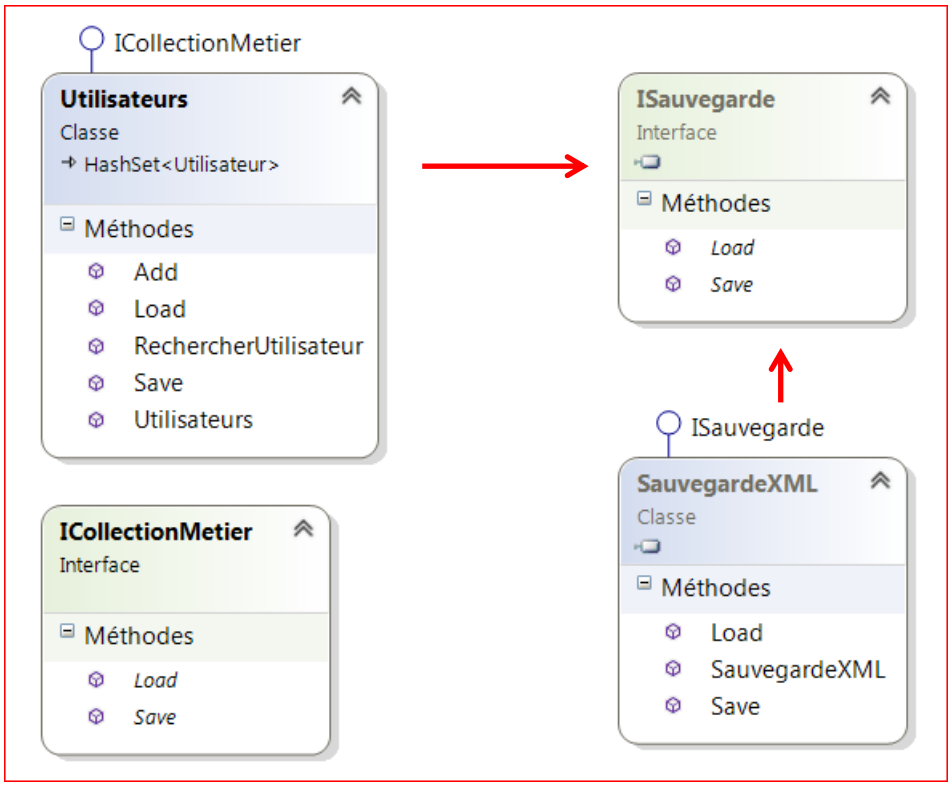
```
2 références
public class SauvegardeXML : ISauvegarde
{


    /// <summary> ...
    4 références
    public void Save(string pathRepData, IEnumerable objetASauvegarder)...
    /// <summary> ...
    4 références
    public IEnumerable Load(string pathRepData, Type typeACharger)...
}
```

Le schéma page suivante vous montre les évolutions et l'inversion des dépendances.

Version 1 : La classe Utilisateurs, dérivée de HashSet, dépend du mécanisme de sauvegarde s'appuyant sur la sérialisation XML.



Version 2 : La même classe Utilisateurs dépend de l'interface de sauvegarde. Le mécanisme concret de sauvegarde dépend de l'interface de sauvegarde.



Nous avons mis ici en œuvre le design pattern d'architecture IOC.

L'implémentation dans l'application exécutée est alors la suivante :

Nous allons conserver les références des objets mis à disposition de tous les objets de l'application dans une classe commune statique Application. Ce qui est important ici, c'est le caractère statique de la référence à l'objet de sauvegarde. **Cette référence sera partagée par tous les composants de l'application.**

```
5 références
internal static class MonApplication
  {

    static ISauvegarde _dispositifSauvegarde = new SauvegardeXML();

    0 références
    public static ISauvegarde DispositifSauvegarde
    {
        get { return MonApplication._dispositifSauvegarde; }
    }

  }
```

Puis dans les autres composants, nous ferons référence à la propriété DispositifSauvegarde.

Exemple ici avec le formulaire de gestion des utilisateurs et la méthode de chargement de la liste des utilisateurs.

```
1 référence
private void ChargerUtilisateurs()
{
    utilisateurs = new Utilisateurs();
    ISauvegarde serialiseur = MonApplication.DispositifSauvegarde;
    utilisateurs.Load(serialiseur,Properties.Settings.Default.AppData);
    foreach (Utilisateur item in utilisateurs)
    {
        cbUtilisateurs.Items.Add(item.Identifiant);
    }
}
```

Nous avons réalisé une application avec une dépendance faible entre objets métier et mécanismes de persistance.

Pour remplacer le mécanisme de sauvegarde actuel, basé sur XML par un dispositif s'appuyant sur la sérialisation binaire, combien de lignes de code devez-vous modifier ?

Nous verrons dans le dernier module de la formation comment injecter automatiquement des dépendances en nous appuyant sur un nouveau patron de conception traitant de l'injection de dépendance ((Dependency Injection).