Vous devrez terminer le TD6 Partie 1 avant de réaliser celui-ci.

1. ***Patron de conception « Injection de dépendances »***

Principes de l'injection de dépendances

En programmation orientée objet, de façon classique, un objet (classe ou module) contient un ensemble de dépendances envers d'autres objets, auxquels il va déporter tout ou partie de ses traitements. Le bon côté de la chose est que l'on évite ainsi que les objets contiennent trop de comportements (les rendant difficiles à maintenir). Le mauvais côté est que chacun de ces objets référencés devient une dépendance forte, car l'objet appelant doit connaitre chacun des objets qu'il va utiliser avant de les instancier.
C'est ce que vous faites quand vous écrivez : `Calcul calcul = new Calcul();`
Même si ce code fonctionne, ce n'est pas une bonne façon de coder, car cela créé une dépendance forte.

L'injection de dépendances (*Dependency Injection - DI*), parfois nommée *Inversion de Dépendance*, est un patron de conception utilisé pour résoudre la problématique des dépendances entre objets tout en permettant un découplage des objets liés, en passant par une indirection.
https://fr.wikibooks.org/wiki/Patrons_de_conception/Injection_de_dépendance
Il consiste à créer dynamiquement (injecter) les dépendances entre les différents objets en s'appuyant sur une description (fichier de configuration ou métadonnées, comme, par exemple, une interface) ou de manière programmatique. Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique (pas d'instanciation, et donc de `new`) mais déterminées dynamiquement à l'exécution : les composants n'ont plus besoin de connaitre la manière dont sont créées leurs dépendances.
Ce pattern est utilisé dans beaucoup de frameworks de développement récents tels qu'Angular avec le décorateur `@Injectable()` (https://angular.io/api/core/Injectable), Symfony, Laravel (https://laravel.sillo.org/cours-laravel-5-5-les-bases-injection-de-dependance-conteneur-et-facades/), Android (https://www.raywenderlich.com/146804/dependency-injection-dagger-2), etc.

Mise en œuvre de l'injection de dépendances

Pour mettre en œuvre l'injection de dépendances, le processus générique est le suivant :

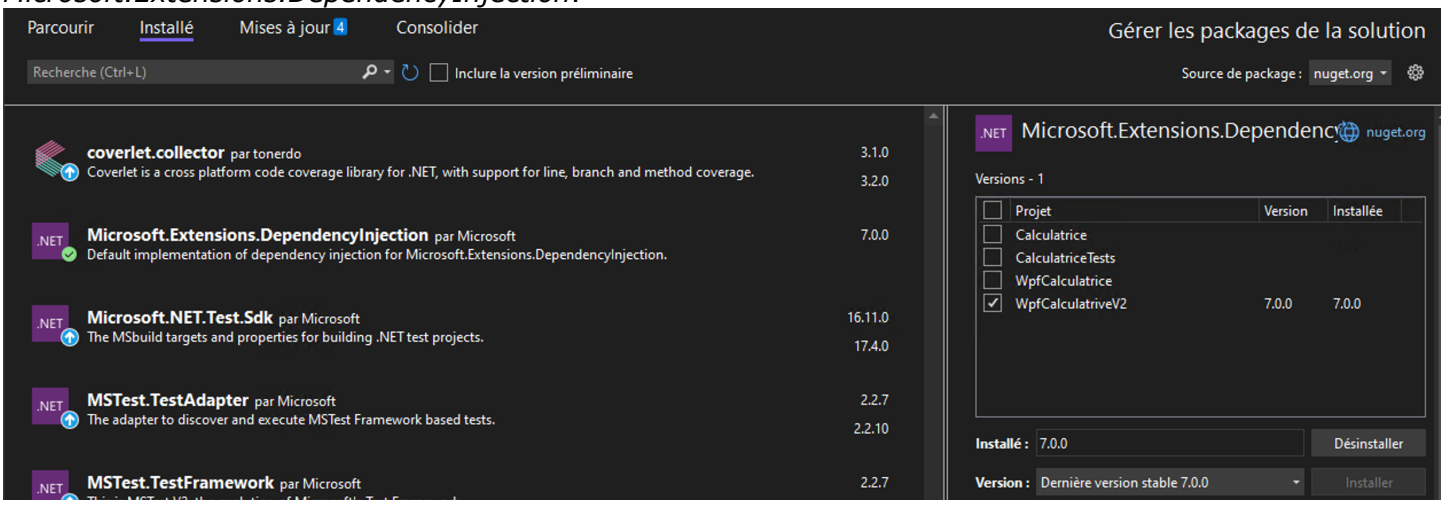
1. Il faut créer une interface I déclarant les méthodes de la classe B utilisées par la classe A. C'est ce que nous avons fait en Partie 1 en créant l'interface `ICalcul` de la classe `Calcul` qui est utilisée par la classe WPF ou la classe `CalculAvance`.
2. Ensuite, déclarer la classe B comme implémentation de cette interface I. C'est ce que nous avons fait en écrivant : `class Calcul:ICalcul.`
3. Enfin, remplacer toute référence à la classe B par des références à l'interface I. Ainsi, nous avons remplacé **`Calcul`** `calcul` par **`ICalcul`** `calcul`.
4. Si besoin, ajouter une méthode pour spécifier l'instance de l'interface I à utiliser. En clair, il s'agit d'indiquer quelle classe sera à instancier lors de l'utilisation d'une interface. **Ce point n'a pas encore été traité. Nous allons le faire ci-dessous.**

Pour vérifier que ce point n'a pas été traité, remplacer le code suivant dans l'application WPF (interface au lieu de la classe) :

```
public ICalcul ObjCalcul
{
  get {
    return ICalcul.Instance;
  }
}
```

Vous devrez rajouter la méthode `Instance` dans l'interface si cela n'a pas été fait.
Vous verrez à l'exécution une exception *NullReferenceException*, indiquant qu'aucun objet de la classe `Calcul` a été instancié. En clair, on ne sait pas qu'il faut instancier un objet de la classe `Calcul`, car l'interface ne sait pas quelle classe lui est liée.
Pour associer l'interface `Calcul` à la classe `ICalcul`, nous allons utiliser un fournisseur de services. Ainsi, quand vous voudrez créer un objet de type `ICalcul`, il instanciera automatiquement la classe `Calcul`.

Pour utiliser le framework d'injection de dépendance, installer le package Nuget *Microsoft.Extensions.DependencyInjection*.



Ajouter le code suivant dans le fichier App.xaml.cs :

```
/// <summary>
/// Interaction logic for App.xaml
/// </summary>
public partial class App : Application
{
    /// <summary>
    /// Gets the instance to resolve application services.
    /// </summary>
    public ServiceProvider Services { get; }

    public App()
    {
        /// <summary>
        /// Configures the services for the application.
        /// </summary>
        ServiceCollection services = new ServiceCollection();
        services.AddTransient<ICalcul, Calcul>();
        Services = services.BuildServiceProvider();
    }

    /// <summary>
    /// Gets the current app instance in use
    /// </summary>
    public new static App Current => (App)Application.Current;

}
```

Le code en jaune permet de configurer les services qui seront créés au démarrage de l'application et utilisables par la suite.
Dans notre cas, nous enregistrons l'interface `ICalcul` qui sera associée à la classe `Calcul`, sous la forme d'un service. Il est possible d'ajouter plusieurs services (`services.Add…`). Les services sont stockés dans un container (*container d'injection de dépendances*), c'est-à-dire une sorte de cache, dans lequel chaque service constitué d'un contrat (l'interface) et de son implémentation est stocké.
Le code en vert nous permettra d'avoir accès à l'application courante, et ainsi, à ses services.

`AddTransient` : L'enregistrement d'un service avec Transient signifie que pour chaque objet qui fera appel à ce service, le conteneur d'IoC va fournir une instance de ce dernier. Concrètement, une nouvelle instance sera créée à chaque appel.

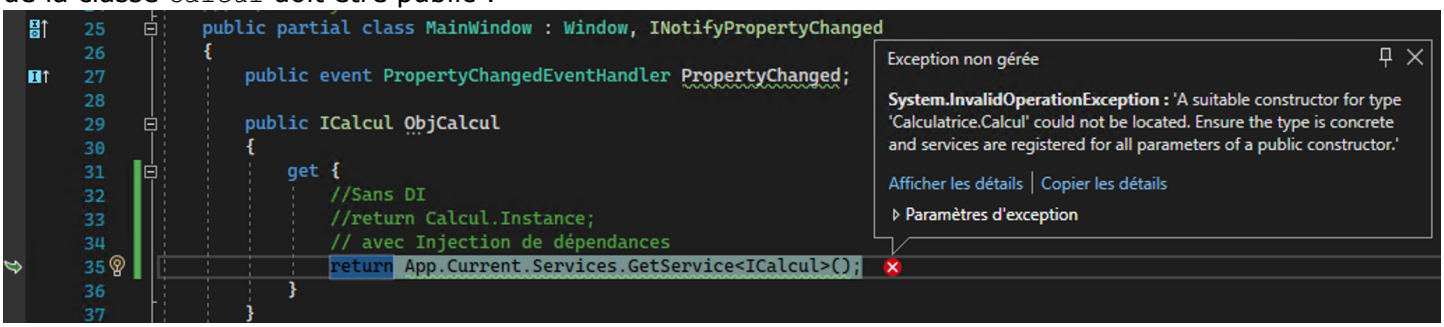
Pour l'instant, cela ne résout pas le problème *NullReferenceException.*
Pour cela, la récupération de la classe `Calcul` et l'instanciation automatique de l'objet se fait ainsi :

```
public ICalcul ObjCalcul
{
    get {
        //Sans DI
        //return Calcul.Instance;
```

```
        // avec Injection de dépendances
        return App.Current.Services.GetService<ICalcul>();
    }
}
```

`App.Current` correspond à : `public new static App Current => (App)Application.Current;`
Et `Services` correspond à : `public ServiceProvider Services { get; }`

Exécuter l'application. Cette fois, l'exception n'est plus levée, mais une erreur indique que le constructeur de la classe `Calcul` doit être public :



Cela montre bien qu'un objet va être instancié, sans avoir besoin de faire un `new`, ce qui prouve le découplage.
Passer le constructeur en `public`.
Cette fois l'application fonctionne, mais cela empêche d'utiliser le singleton. Mettre en commentaires le code du singleton qui n'est plus utile. Faire de même pour la méthode `Instance` de l'interface.

Implémenter un singleton quand nous avons mis en place un framework de DI est très simple : il suffit de remplacer la méthode `AddTransient` par `AddSingleton` :

```
public partial class App : Application
  {
    /// <summary>
    /// Gets the instance to resolve application services.
    /// </summary>
    public ServiceProvider Services { get; }

    public App()
    {
      /// <summary>
      /// Configures the services for the application.
      /// </summary>
      ServiceCollection services = new ServiceCollection();
      services.AddSingleton<ICalcul, Calcul>();
      Services = services.BuildServiceProvider();
    }

    /// <summary>
    /// Gets the current app instance in use
    /// </summary>
    public new static App Current => (App)Application.Current;

  }
```

`AddSingleton` : Singleton est utilisé pour un service qui doit être instancié une seule fois et dont la même instance sera utilisée par tous les composants de l'application qui en auront besoin. Le service est créé pour le premier composant qui en fait la demande, et utilisé pour le reste.
Si votre application nécessite un Singleton, au lieu d'implémenter le pattern Singleton, il est recommandé d'utiliser ce cycle de vie de service.

Test du mécanisme de DI

Imaginons que nous ayons 2 classes qui fassent la même chose (mais différemment). Elles fournissent les mêmes méthodes (mais avec du code différent) et implémentent la même interface.
Pour illustrer cet exemple, créer une nouvelle classe `Calcul2` qui fasse la même chose que `Calcul`, mais différemment.

Copier le code de `Calcul` dans `Calcul2`. `Calcul` implémentera la méthode `Factorielle` en utilisant une boucle. La méthode `Factorielle` de `Calcul2` utilisera la récursivité.
Modifier le service : services.AddSingleton<ICalcul, **Calcul2**>();
On voit qu'il n'y a qu'une seule modification à réaliser pour remplacer une classe par une autre, ce qui prouve un haut niveau de découplage.

Mettre un point d'arrêt dans la méthode `Factorielle` de `Calcul2`. Vous verrez que c'est bien cette méthode qui sera exécutée.

Remarque : L'inversion de contrôle (IoC) (https://fr.wikipedia.org/wiki/Inversion_de_contrôle) est un patron d'architecture très utilisé en développement orienté objet. Ce concept veut que lorsqu'un module effectue un traitement, le contrôle du traitement soit déporté vers l'appelé, et non pas vers l'appelant. En pratique, on va chercher à diminuer au maximum la connaissance qu'a l'appelant de la mécanique interne de l'appelé.
L'inversion de contrôle est un terme générique. L'injection de dépendances est une forme d'IoC.

2. ***Travail à faire n°1***

Appliquer le patron DI à la classe `CalculAvance` (ajouter un service de type singleton au container de DI).

3. ***Travail à faire n°2 : application ComptesBancaires***

- Installer le package *Microsoft.Extensions.DependencyInjection* dans tous les projets.

- Appliquer le patron DI à la classe `DataAccess` (singleton) :
  - Supprimer le code du singleton (le constructeur devra être `public`).
  - Code à ajouter dans `ServiceCompte` pour créer les services (on pourrait aussi le faire dans une classe spécifique `DI`) :

```
public IDataAccess ObjDataAccess
{
      get
      {
          // Ici nous n'avons pas accès à App car il ne s'agit pas d'une application, mais d'une bibliothèque de classes
          // Accès direct aux services définis dans la classe
          return Services.GetService<IDataAccess>();
      }
}

public ServiceCompte()
{
      //Création des services utilisés par la classe
      ServiceCollection services = new ServiceCollection();
      services.AddSingleton<IDataAccess, DataAccess>();
      Services = services.BuildServiceProvider();
}
```

Intérêt d'utiliser la DI sur la classe `DataAccess` : lire **l'introduction** de l'article suivant :
https://nathanaelmarchand.developpez.com/tutoriels/dotnet/architecture-couches-decouplage-et-injection-dependances-avec-unity/
Remarque : Unity est un framework de DI aujourd'hui déprécié et remplacé par Microsoft.Extensions.DependencyInjection.

- Appliquer le patron DI à la classe `ServiceCompte` (transient) de l'application ComptesBancaires :
  - Créer les services dans `App.xaml.cs` de l'application WPF.
  - Si vous voyez l'erreur « Impossible d'utiliser la méthode… non générique avec des arguments de type », n'oubliez pas d'ajouter : *using Microsoft.Extensions.DependencyInjection*

Bilan :
Une interface permet donc :

- *Un code découplé de son implémentation, respectant les principes **SOLID**.*
- *Un code facilité pour **utiliser l'injection de dépendance**, qui permet d'encore plus de découpler son code.*
- *A tout architecte d'imposer des contrats et donc des méthodes à être présentes dans des classes.*

L'injection de dépendance correspond au D de SOLID.
SOLID :
Les S.O.L.I.D. sont des principes de développement énoncés et mis en forme par Michael Feathers et Robert C.Martin (Oncle Bob). Ils ont été créés au début des années 2000. L'intérêt de ces principes de développement est de donner une ligne conductrice afin que la création d'applications orientées objet soit la plus efficace, la plus cohérente et la plus maintenable possible. À la différence des modèles de développement « classiques », ils peuvent être appliqués à différents niveaux lors du développement. Cela va du niveau le plus atomique, c'est-à-dire le développement d'une procédure, au niveau le plus global qu'est le développement d'un module. Leur connaissance constitue donc un atout que tout bon développeur et architecte doit posséder.

S comme Single Responsibility
Chaque classe ne doit posséder qu'une seule responsabilité. Chaque objet ne doit remplir qu'une tâche pour que celle-ci ait toutes les chances d'être bien faite. Logiquement, il est plus simple d'effectuer la conception et le développement d'un objet si celui-ci n'a qu'un but unique. Si les développements suivent ce principe, l'application devient plus parcellisée et plus modulaire car ses composants sont moins liés entre eux. De plus, si une fonctionnalité est directement liée à un objet, sa maintenance est plus ciblée et donc plus efficace. Enfin, si l'identification d'un problème est plus simple alors la maintenance est moins risquée. Ce principe s'applique aux classes mais pourrait très bien s'appliquer à des modules ou à des services.

O comme Open Close Principle
Chaque partie d'un logiciel doit être ouverte aux extensions mais fermée aux modifications. L'idée derrière ce principe est que lorsqu'un objet est développé, testé et revu, alors il est préférable de pouvoir l'étendre sans en modifier son fonctionnement. Le développeur aura alors prévu un mécanisme pour rendre son comportement extensible. Il est possible de prévoir une classe qui peut être héritée ou un système de gestion modulaire. En lui fournissant cette capacité d'extension, le développeur assure un code principal fonctionnel mais aussi évolutif. On peut transposer ce principe en créant des logiciels extensibles sans avoir à retoucher à leur contenu existant. Si la partie existante n'est pas impactée, les tests qui la concernent restent valides. L'évolution de l'application se fait donc sur des bases stables.

L comme Liskov Substitution
Ce principe est plus abstrait. Il permet de valider une bonne utilisation de l'héritage : si une classe S dérive d'une classe T, alors un objet de type T peut être remplacé par un objet de type S sans altérer le fonctionnement du logiciel.
L'exemple le plus simple concerne la création des classes géométriques Carre et Rectangle. La question est la suivante : est-ce que le carré dérive du rectangle ou est-ce que le rectangle dérive du carré ?

Le premier cas part du postulat que le carré est logiquement un rectangle plus spécifique. La classe Carre dérive donc de la classe Rectangle.

```
public class RectangleV1
{
    public virtual float Longueur { get; set; }
    public virtual float Largeur { get; set; }

    public float Perimetre { get { return 2 * (Longueur + Largeur); } }
    public float Aire { get { return Longueur * Largeur; } }
}
```

Dans le cas de la classe **RectangleV1**, la longueur et la largeur sont gérées de manière indépendante.

```
public class CarreV1 : RectangleV1
{
    private float _Longueur;
    private float _Largeur;
    public override float Longueur
    {
        get
        {
            return _Longueur;
        }
        set
        {
            _Longueur = value;
            _Largeur = value;
        }
    }
    public override float Largeur
    {
        get
        {
            return _Largeur;
        }
        set
        {
            _Longueur = value;
            _Largeur = value;
        }
    }
}
```

Si le carré dérive du rectangle et afin que chaque instance de carré soit valide à tout moment, il faut redéfinir la longueur et la largeur pour que l'écriture de l'un impacte forcément l'autre. Un simple test démontre alors que cet héritage ne respecte pas le principe de substitution de Liskov.

```
[TestClass]
public class TestV1
{
    [TestMethod]
    public void SimpleSubstitution()
    {
        RectangleV1 r = new RectangleV1() { Largeur = 10, Longueur = 20 };
        Assert.IsTrue(r.Longueur != r.Largeur);

        //Après substitution
        CarreV1 c = new CarreV1() { Largeur = 10, Longueur = 20 };
        Assert.IsTrue(c.Longueur != c.Largeur);
    }
}
```

Dans ce cas, le test échoue car les longueurs des deux côtés du carré sont égales bien que l'affectation des longueurs et largeurs ait eu des valeurs différentes. Cela démontre que l'instance du rectangle ne peut être remplacée par une instance de carré sans impacter le fonctionnement du code.

Dans un second cas, à l'inverse, il est possible de dériver la classe **Rectangle** de la classe **Carre** en partant du principe que le rectangle est semblable à un carré à la différence qu'il a deux côtés qui peuvent être différents.

```
public class CarreV2
{
    public float LongueurCote { get; set; }

    public virtual float Perimetre { get { return 4 * (LongueurCote); } }
    public virtual float Aire { get { return LongueurCote * LongueurCote; } }
}
```

La classe **Carre** expose ses composantes mais offre la possibilité de redéfinir le calcul du périmètre et de l'aire.

```
public class RectangleV2 : CarreV2
{
    public float Largeur { get; set; }

    public override float Perimetre { get { return 2 * (LongueurCote + Largeur); } }
    public override float Aire { get { return LongueurCote * Largeur; } }
}
```

La classe **Rectangle** ajoute une propriété pour stocker sa largeur. Avec l'ajout de cette dimension, les calculs du périmètre et de l'aire sont différents. Ils sont donc redéfinis. Un second test démontre que le principe de substitution de Liskov n'est pas respecté.

```
[TestClass]
public class TestV2
{
    [TestMethod]
    public void SimpleSubstitution()
    {
        CarreV2 c = new CarreV2() { LongueurCote = 10};
        Assert.AreNotEqual(0, c.Aire);
        Assert.AreNotEqual(0, c.Perimetre);

        //Après substitution
        RectangleV2 r = new RectangleV2() { LongueurCote = 10 };
        Assert.AreNotEqual(0, r.Aire);
        Assert.AreNotEqual(0, r.Perimetre);
    }
}
```

Dans le second test, l'aire du rectangle est égale à 0 car sa seconde longueur n'est pas définie (et donc égale à 0). Le remplacement de la classe **CarreV2** par **RectangleV2** change donc le comportement de l'application.

Un troisième cas est discernable et permet de satisfaire la contrainte de cette règle en créant une classe qui est la source de l'héritage de la classe **Carre** mais aussi de la classe **Rectangle**. **GeometrieV3** ne possède que l'intersection des deux classes filles. Un carré et un rectangle ont tous deux une aire et un périmètre.

```
public abstract class GeometrieV3
{
    public abstract float Perimetre { get; }
    public abstract float Aire { get ;}
}
```

La classe précédente ne contient aucune indication quant au calcul des aires et périmètres. Elle ne porte que l'obligation pour la classe fille d'implémenter ces deux propriétés.

```
public class CarreV3 : GeometrieV3
{
    public float LongueurCote { get; set; }

    public override float Perimetre { get { return 4 * (LongueurCote); } }
    public override float Aire { get { return LongueurCote * LongueurCote; } }
}
```

```
public class RectangleV3 : GeometrieV3
{
    public float Longueur { get; set; }
    public float Largeur { get; set; }

    public override float Perimetre { get { return 2 * (Longueur + Largeur); } }
    public override float Aire { get { return Longueur * Largeur; } }
}
```

Les deux classes précédentes héritent de la classe **GeometrieV3** et incluent une méthode **get** pour les propriétés **Perimetre** et **Aire**.
Aucun test ne peut démontrer la validité de cette solution sachant que la classe **GeometrieV3** ne peut exister indépendamment de la classe **CarreV3** et **RectangleV3**. Il est alors valide, suivant le design de la solution, que la classe fille remplace la classe parente dans l'héritage.

I comme Interface Segregation
Un objet ne devrait pas être dépendant de méthodes qu'il n'utilise pas. Il est donc intéressant d'abstraire par la création d'interfaces, et il est d'autant plus intéressant que ces interfaces soient unitaires et minimales. Le consommateur de l'interface et le fournisseur de l'interface sont donc liés par un contrat qui est minimal et qui ne concerne qu'un seul type de service.

D comme Dependency Inversion
En prenant l'exemple d'un logiciel construit en différentes couches, ce principe dit qu'aucune couche ne devrait être dépendante des autres couches. Elles devraient toutes dépendre d'interfaces. Par ce biais elles deviendront agnostiques des comportements des autres et sont abstraites des particularités de chacune.
L'inversion de dépendance propose que la couche supérieure propose les interfaces que doivent respecter les modules. Ainsi, l'inversion se fait et les modules sont libres de leurs implémentations sachant qu'ils n'exposent que le contenu des interfaces connues par la couche supérieure.