|  |  |  |
| --- | --- | --- |
| logo_IUT_Annecy_RVB | **R3.04 Qualité de développement**  **TD6 Partie 2 – Injection de dépendance** | *Une image contenant texte, clipart  Description générée automatiquement* |

**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 utilisé́ 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;

}

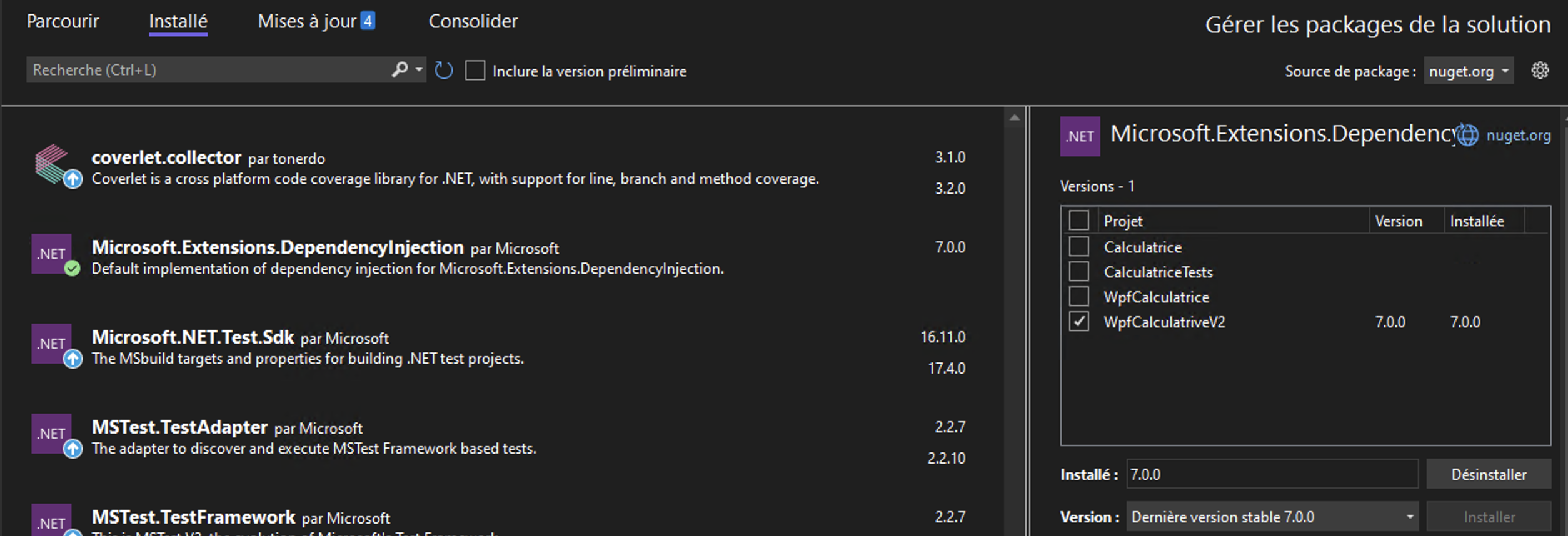
}

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 :

private 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 :

Une image contenant texte

Description générée automatiquement

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*](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.*

1. Travail à faire n°1

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