Injection de dépendances en .NET, pattern, intérêt et outils

Par Philippe Vialatte

Date de publication : 2 juin 2009

Dernière mise à jour : 13 août 2009

Dans mon article précédent sur **les principes SOLID**, j'ai mentionné, brièvement, les outils disponibles pour faire de l'injection de dépendances.

Comme ces outils, ainsi que les patterns et principes qui s'y rattachent, semblent relativement peu connus (et encore moins utilisés) dans la communauté .NET francophone, cet article va s'attarder plus longuement sur les avantages de l'injection de dépendances, ainsi que sur les moyens de la mettre en oeuvre.

I - Introduction

Cet article va s'attarder sur les diverses techniques d'injection de dépendances

Au début de la rédaction de cet article, bien qu'ayant déjà une idée assez claire des différents framework, ainsi que des différents principes mis en oeuvre, je me suis rendu compte (à mes dépens) qu'une incertitude restait sur les différents termes gravitant autour de cette technique.

Pour clarifier, on distinguera les termes suivants :

1. Inversion de contrôle : l'inversion de contrôle (IoC) est un concept de haut niveau en développement orienté objet. L'inversion de contrôle 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'à l'appelant de la mécanique interne de l'appelé.
2. Inversion de dépendances : l'inversion de dépendances (DI) est un principe de développement orienté objet. L'inversion des dépendances est une des implémentations possibles de l'inversion de contrôle. J'ai déjà mentionné ce principe précédemment, mais pour éviter un (second !) lien vers le premier article, la définition simplifiée de ce principe est que, pour diminuer le couplage entre les classes, on va ajouter une interface entre chaque classe, de façon à ce qu'au lieu d'appeler une classe physique, l'appelant appelle une interface, ceci permettant d'ajouter un niveau d'abstraction supplémentaire entre l'appelant et l'appelé.
3. Injection de dépendances : l'injection de dépendances est une technique permettant de mettre en oeuvre l'inversion de dépendances.

On trouve régulièrement l'un des trois termes ci-dessus utilisés pour qualifier l'injection de dépendances.

En programmation orientée objet, de façon classique, un objet (classe ou module) va contenir un ensemble de dépendances sur d'autres objets, auxquels il va déporter tout ou partie de ses traitements. Le bon côté de la chose est que l'on évite que nos 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 notre objet appelant doit connaitre chacun des objets qu'il va utiliser avant de les instancier.

Par exemple, prenons le cas d'une classe d'accès aux données. L'exemple standard d'appel à une fonction située dans une classe d'accès aux données aura la forme suivante :

```
public class CustomerService {
 public Customer GetOneCustomer(int customerId){
    CustomerRepository _repository = new CustomerRepository();
    return new Customer(_repository.GetOneById(customerId));
 }
}
```

Ce code, au demeurant simple à comprendre, pose, dans le cadre de l'inversion de contrôle, un ensemble de problèmes. Premièrement, cette implémentation est fortement liée à la classe CustomerRepository, si cette classe (ou du moins son constructeur) change, GetOneCustomer devra changer aussi. Deuxièmement, dans cette implémentation, GetOneCustomer hérite d'une responsabilité superflue, à savoir la création du CustomerRepository.

Si l'on utilise l'injection de dépendances, la représentation finale de notre appel ressemblera plus à :

```
public class CustomerService {

 private CustomerRepository _repository;

 public Customer GetOneCustomer(int customerId){
    return new Customer(_repository.GetOneById(customerId));
 }
}
```

La différence, minime au demeurant, est que la résolution de la dépendance est déportée en dehors du code "métier". En d'autres termes, on va, dans un premier temps, définir un jeu d'interfaces de façon à ce que nos différents modules puissent communiquer par un contrat. Dans un second temps, on va "injecter" dans notre objet un autre objet répondant au contrat défini.

Utiliser l'inversion de contrôle offre les avantages suivants :

- Chaque système ne se concentre que sur sa ou ses tâches principales.
- Les différents systèmes ne font pas d'hypothèse sur le comportement des autres systèmes.
- Remplacer un système ne produit pas d'effets de bord sur les autres systèmes, tant que le contrat d'origine est respecté.
- Dans le cas d'une nouvelle version d'un composant (ou d'un composant alternatif, comme un changement de framework de log, par exemple), il est plus facile de changer le composant appelé.

Une utilisation que l'on rencontre régulièrement de l'injection de dépendances est l'injection d'un Mock ou d'un Stub dans le cas de tests unitaires en isolation.

II - Types d'injection possibles

De façon classique, on va distinguer deux types d'injection. Ces possibilités sont offertes par tous les langages objets. Ces types sont l'injection par une propriété (ou une méthod epour les langages n'en disposant pas), et l'injection par le constructeur.
D'autres possibilités sont disponibles, mais ces deux-ci sont celles que l'on rencontrera le plus souvent.

II-A - Injection par une propriété

II-A-1 - Mise en place

L'injection par une propriété est la plus simple à mettre en place. Pour ce faire, il va suffire de fournir une propriété supplémentaire pour chaque dépendance que l'on veut injecter.

Par exemple, dans le cas de notre exemple précédent, le code deviendra :

```
public class CustomerService {
 private CustomerRepository _repository;

 public CustomerRepository Repository {
  set{
   _repository = value;
  }
 }

 public Customer GetOneCustomer(int customerId){
    return new Customer(_repository.GetOneById(customerId));
 }
}
```

Ce code est directement utilisable, et la responsabilité de l'instanciation de la classe a, du coup, été enlevée de notre classe, le client de notre classe étant dorénavant responsable de la fourniture des dépendances adéquates.

II-A-2 - Avantages & Inconvénients

Le plus gros avantage de cette implémentation est qu'elle permet de résoudre certains cas compliqués, comme les dépendances circulaires.

Imaginons le cas d'un Service EmployeeService, avec une dépendance sur CompanyService. Si CompanyService a aussi une référence sur EmployeeService, on se retrouve dans le cas d'une dépendance circulaire, et le seul moyen d'injecter la référence sera d'utiliser une propriété.

Il est toutefois possible, dans l'exemple précédent, de n'utiliser l'injection de dépendances que sur l'un des deux objets. Dans ce cas, on n'utilisera l'injection par propriété que pour l'objet le moins utilisé.


Notez qu'il est fortement déconseillé d'avoir des dépendances cycliques entre les classes d'un projet.

Le plus gros inconvénient de cette méthode est qu'elle n'impose aucun contrat sur l'initialisation des dépendances. En effet, lorsque l'on va avoir, par exemple, une classe de gestion de portefeuille possédant des dépendances sur :

- Un service de gestion de sécurité
- Un repository permettant de stocker les données du portefeuille

Le code suivant permettra d'injecter les deux dépendances mentionnées :

```
public class PortfolioService{

  private ISecurityService _securityService;
  private IPortfolioRepository _portfolioRepository;

  public ISecurityService SecurityService{
    set{
      _securityService = values;
    }
  }

  public IPortfolioRepository PortfolioRepository{
    set{
      _portfolioRepository = values;
    }
  }

  public bool SavePortfolio(Portfolio pf, UserInfo user){
    if (_securityService.IsPortfolioWriteableByUser(user)){
       _portfolioRepository.Save(pf);
    }
  }
}
]
```

Par contre, rien n'empêche fondamentalement les clients du code de faire ceci :

```
PortfolioService tmpPortfilio = new PortfolioService();
tmpPortfilio.SavePortfolio(myPortfolio,currentUser);
// Va remonter une NullReferenceException
```

Il faut donc s'assurer systématiquement que toutes les dépendances soient correctement initialisées, ce qui ajoute au choix, des tests supplémentaires dans le code et/ou de la documentation sur ces fonctionnalités, et, de façon générale, du travail supplémentaire.

Un autre désavantage se situe au niveau des propriétés elles-même. En effet, les propriétés d'injection deviennent forcément une partie de l'API de l'objet, alors qu'elles ne sont que des méthodes de "plomberie".

II-B - Injection par le constructeur

II-B-1 - Mise en place

Le système d'injection le plus couramment utilisé est de passer la dépendance au constructeur. Dans notre exemple précédemment montré, cela donnera le code suivant :

```
public class CustomerService {
 private ICustomerRepository _repository;

 public CustomerService (ICustomerRepository repository) {
  _repository = repository;
 }

 public Customer GetOneCustomer(int customerId){
    return new Customer(_repository.GetOneById(customerId));
 }
}
```

II-B-2 - Avantages & Inconvenients

Le gros avantage de ce mode d'injection est que l'on est sûr qu'au moment où les fonctions sont appelées depuis la classe, toutes les dépendances passées par le constructeur ont été fournies à l'objet. De plus, le passage des dépendances par constructeur permet d'avoir une forme d'immutabilité des dépendances (on peut facilement passer les dépendances en lecture seule). Enfin, il est plus simple, dans ce cas, de voir quelles sont les dépendances nécessaires pour le module.

L'inconvenient de l'injection par le constructeur est qu'une fois que l'objet a été instancié, il n'est plus possible de modifier la dépendance injectée. En effet, comme l'injection se fait dans le constructeur, il n'est pas possible d'introduire une nouvelle dépendance dans l'objet sans ajouter un accesseur supplémentaire pour manipuler cette dépendance (et donc, retomber dans les problèmes liés à l'injection par propriété).

Il est aussi parfois plus difficile de maintenir des arborescences d'héritage profondes avec cette méthode.

II-C - En résumé

Ces deux types d'injection permettent de gérer la quasi-totalité des cas. Pour voir les autres possibilités d'injection, et pour approfondir le côté "scientifique" de l'injection de dépendances (je préfère garder l'article à un niveau lisible), un bon point de départ est l'article suivant, de Martin Fowler : **Inversion of Control Containers and the Dependency Injection pattern**

Il est même possible de mélanger les deux, en passant les dépendances "vitales" par le constructeur, et des dépendances "optionnelles" (même si, dans ce cas, ce ne sont plus vraiment des dépendances ;)) ou les cas particuliers par des propriétés.

L'inconvénient partagé par ces deux méthodes est que dans un cas comme dans l'autre, la responsabilité de l'injection de dépendances, déportée sur les appelants, va mécaniquement augmenter la quantité de code, et potentiellement augmenter la maintenance en dispersant la création des dépendances dans tous les appelants.

Pour contourner ce problème, on va se reposer sur des outils spécifiques, les conteneurs IoC.

III - Conteneurs IoC disponibles en .NET

Les conteneurs IoC sont des outils spécifiquement conçus pour faciliter l'injection de dépendances. Originellement, ces utilitaires viennent du monde Java, mais, au fil de ces dernières années, des conteneurs IoC propres au framework .NET sont apparus, issus du monde open source aussi bien que de Microsoft.

Pour montrer un exemple le plus proche de la réalité possible, on va partir du cas métier suivant :

On veut développer un gestionnaire de Clients, avec :

- une classe d'accès aux données ClientRepository, qui implémente l'interface IClientRepository
- une classe métier ClientService, qui implémente l'interface IClientService. ClientService possède une dépendance sur un objet de type IClientReposiory, qui va lui fournir des données à traiter
- un formulaire qui utilise une interface métier IClientService

Le code pour ces trois classes aura donc la forme suivante :

```
public class Form1{

    public void LoadCustomerData(int customerId){
        ICustomerRepository repository = new CustomerRepository();
        ICustomerService service = new CustomerService(repository);
        Customer monClient = service.GetOneCustomer(1);
        Messagebox.Show(monClient.Name);
    }

}

public interface ICustomerService {

 public DataTable GetOneCustomer(int customerId){
    ...
 }
}
public class CustomerService {
 private ICustomerRepository _repository;

 public CustomerService (ICustomerRepository repository) {
  _repository = repository;
 }

 public Customer GetOneCustomer(int customerId){
    return new Customer(_repository.GetOneById(customerId));
 }
}

public interface ICustomerRepository {

 public DataTable GetOneById(int customerId){
    ...
 }
}

public class CustomerRepository {

 public DataTable GetOneCustomer(int customerId){
    ...
 }
}
```

Comme on l'avait remarqué précédemment, la classe cliente récupère la responsabilité de l'instanciation des classes qu'elle doit injecter. On va voir comment se passer de ce code.

Je ne vais pas faire une présentation approfondie de chaque framework (sinon, ce ne serait plus un article, mais un livre, voire une bible), mais plutôt aborder rapidement leur fonctionnement et leurs différences.

III-A - Spring.NET

Spring.NET a l'avantage, sur les autres conteneurs, de la maturité. C'est en effet un portage en .NET de Spring en Java.

Il permet de faire tout ce qu'on peut vouloir demander à un framework IoC, mais souffre de son âge (pas de typage générique), toute la configuration et la récupération des dépendances est basée sur des chaines, ce qui peut entrainer des erreurs à l'exécution.

La philosophie de la partie IoC de Spring.NET est la suivante :

- Toute la configuration se fait dans le fichier de configuration de l'application (Web.Config ou App.Config)
- Le fichier de configuration ne mentionne pas le type d'interface implémenté, mais directement le type physique.
- La récupération des dépendances injectées se fait par l'objet ContextRegistry. Il est ensuite nécessaire de faire un cast de la classe retournée par l'objet ContextRegistry.

Dans notre exemple, le fichier de configuration aura le contenu suivant :

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    </sectionGroup>
  </configSections>
  <spring>
    <context>
      <resource uri="config://spring/objects"/>
    </context>
    <objects xmlns="http://www.springframework.NET">
      <object name="ProductService" type="ServiceLayer.ProductService, ServiceLayer"
              singleton="true">
        <constructor-arg index="0" ref="ProductRepository"/>
      </object>
      <object name="ProductRepository"
        type="RepositoryLayer.ProductRepository,RepositoryLayer"/>
    </objects>
  </spring>
</configuration>
```

La récupération des dépendances se fera ainsi :

```
IApplicationContext ctx = ContextRegistry.GetContext();
            var monClient = ((IProductService)ctx.GetObject("ProductService")).
                GetOneProductById(1);
```

Les seules raisons valides, à mon avis, d'utiliser Spring .NET pour de l'IoC sont :

- l'utilisation d'autres briques fournies par le framework Spring.NET (AOP, Messaging, interface avec NHibernate...)
- une équipe habituée à travailler avec Spring en Java
- un projet en .NET 1.1

Je ne travaille pas au quotidien avec Spring.NET, je suis fan des génériques et des lambdas, donc, ma vision des choses est probablement biaisée.
Je prierai donc les utilisateurs de Spring.NET de ne pas brûler ma voiture dans le cas d'une erreur dans cette partie. Si c'est le cas, signalez-le moi, je serais heureux d'adapter cet article.

Ce framework est disponible à l'adresse suivante : **http://www.springframework.NET/**

III-B - Unity

Unity est le conteneur d'IoC proposé par Microsoft. Quoique beaucoup plus récent que Spring, il est pratiquement aussi puissant... et aussi complexe. De plus, comme Unity a commencé à être développé alors que le framework 2.0 était déjà disponible, il intègre naturellement la résolution et la configuration des dépendances sous forme de génériques.

Dans le cas d'Unity, le travail de récupération des dépendances se fait en manipulant un IUnityContainer. Le framework Unity propose, à la base, une implémentation de l'interface IUnityContainer sous la forme de la classe UnityContainer.

Comme je le disais précédemment, on pourra déclarer les dépendances à résoudre de ces deux façons différentes :

Déclaration au format XML

```
<configuration>
 <configSections>

  <section name="unity" type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection, Microsoft.Practi
  </configSections>
  <unity>
    <typeAliases>
      <typeAlias alias="int" type="System.Int32, mscorlib" />

  <typeAlias alias="singleton" type="Microsoft.Practices.Unity.ContainerControlledLifetimeManager, Microsoft.Prac

  <typeAlias alias="IProductRepository" type="RepositoryLayer.IProductRepository,RepositoryLayer" />
      <typeAlias alias="IProductService" type="ServiceLayer.IProductService, ServiceLayer" />
    </typeAliases>

    <containers>
      <container>
        <types>
          <type type="IProductRepository" mapTo="RepositoryLayer.ProductRepository,RepositoryLayer">
            <lifetime type="singleton" />
          </type>
          <type type="IProductService" mapTo="ServiceLayer.ProductService, ServiceLayer">
            <lifetime type="singleton" />

  <typeConfig extensionType="Microsoft.Practices.Unity.Configuration.TypeInjectionElement, Microsoft.Practices.Un
              <constructor>
                <param name="repository" parameterType="IProductRepository"/>
              </constructor>
            </typeConfig>
          </type>
        </types>
      </container>
    </containers>
  </unity>
</configuration>
```

On notera que la configuration au format XML est encore plus verbeuse que la configuration XML des dépendances de Spring IoC. RegisterType, au contraire, est plus léger d'utilisation, et utilise une interface fluide, permettant de chainer les déclarations de résolution.

Déclaration en utilisant RegisterType

```
IUnityContainer container = new UnityContainer();
container.RegisterType<IProductRepository, ProductRepository>()
        .RegisterType<IProductService, ProductService>();
```

Utiliser le format précédent nécessite évidemment de penser à l'initialisation de container avant de l'utiliser. Typiquement, ce code d'initialisation aura lieu dans le Global.asax pour un site web, ou dans la méthode Main d'une application de bureau. Il faudra de plus permettre un accès direct au conteneur depuis les appelants, ce qui nécessitera souvent la mise en place d'une classe de résolution du conteneur.
Le gros avantage de ce type de déclaration est qu'elle permet d'avoir une résolution à la compilation, alors que la configuration au format XML ne sera résolue qu'à l'exécution, ce qui peut éventuellement entrainer des anomalies.

Une fois le conteneur initialisé, quelle que soit la méthode choisie, on récupère les dépendances de cette façon :

```
var monClient = container.Resolve<IProductService>().
    GetOneProductById(1);
```

Une étape supplémentaire est nécessaire dans le cas de la configuration au format XML. En effet, il faut récupérer les informations de configuration depuis le fichier XML, de la façon suivante :

```
var section = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");
section.Containers.Default.Configure(container);
```

Au final, l'avantage principal d'utiliser Unity est que c'est un framework Microsoft (l'utilisation d'outils open source est parfois découragée dans certaines sociétés), ce qui assure une certaine pérennité au framework, et un accès assez simple à la documentation (ce qui est souvent un point faible des autres frameworks). Côté défauts, la configuration XML est la plus lourde de tous les conteneurs présentés dans cet article, et les performances ne sont pas les meilleures.

Ce framework est disponible à l'adresse suivante : **http://www.codeplex.com/Wiki/View.aspx?ProjectName=unity**

III-C - Ninject

Ninject est un des derniers nés des frameworks d'IoC. Sa particularité est d'être beaucoup moins complexe que les autres. Ninject ne propose pas de configuration basée sur des fichiers XML, mais exclusivement sur une interface fluide.

La version actuellement disponible est la 1.0, qui fonctionne à partir du framework 2.0, la version suivante de Ninject (en beta pour le moment), abandonnera le support des framework 2.0 et 3.0 pour ne fonctionner qu'à partir du 3.5.

La philosophie de Ninject est légèrement différente de celle des frameworks précédents. En effet, là où les autres frameworks appellent un objet de contexte, Ninject travaille à partir d'un objet de type Kernel, dans lequel on va charger des modules, ces modules ayant la responsabilité de contenir les informations de résolution de type.

Par exemple, pour définir la résolution de IProductService en ProductService, et de IProductRepository en ProductRepository, on peut définir un module sous cette forme :

```
class ProductModule : StandardModule {
    public override void Load() {
        Bind<IProductService>().To<ProductService>();
        Bind<IProductRepository>().To<ProductRepository>();
```

```
    }
}
```

Et l'utiliser ainsi :

```
IKernel kernel = new StandardKernel(new ProductModule());

var monClient = kernel.Get<IProductService>().
    GetOneProductById(1);
```

Comme, pour une configuration simple (comme le cas présent), créer une classe supplémentaire peut être superflu, un objet InlineModule existe, qui permet de définir un module en lui passant les informations de résolution sous forme de lambda :

```
IKernel kernel = new StandardKernel(new InlineModule(
      module => module.Bind<IProductService>().To<ProductService>(),
      module => module.Bind<IProductRepository>().To<ProductRepository>()
      )
  );
```

En résumé, les forces principales de Ninject sont :

- sa petite taille, l'ensemble du framework ne pèse que 100 ko (80ko pour le moment pour la v2.0 beta)
- le dynamisme des contributeurs
- la richesse de son API (comme d'habitude, je ne suis pas un expert, mais elle me semble être la plus étendue de toutes).

Du coté des faiblesses, on pourra noter sa jeunesse, qui peut rendre difficile la recherche d'information sur des cas limites (à noter, le site de Ninject contient tout de même un tutoriel très complet), ainsi que l'absence de configuration par XML (ce qui peut être un problème lorsque l'on veut pouvoir changer l'implémentation résolue après une livraison).

L'un dans l'autre, c'est un framework avec lequel j'ai bien apprécié de jouer.

Ninject est disponible à l'adresse suivante : **http://ninject.org/**

III-D - StructureMap

Comme d'habitude, je préfère garder le meilleur pour la fin ;).

StructureMap est un conteneur open source, dont le développement est en cours depuis quatre ans (soit le plus vieux framework en dehors de Spring). Contrairement à Spring, il ne s'agit pas d'un portage d'un projet Java existant, mais bien d'un framework pensé et développé pour l'inversion de contrôle en .NET.

Les dernières versions de StructureMap incluent la notation lambda que l'on retrouve aussi dans Ninject, mais, contrairement à ce dernier, supporte également l'utilisation de fichiers de configuration, et même l'utilisation d'attributs de classe.

StructureMap travaille à partir d'une Factory, nommée ObjectFactory, qui va gérer à la fois la récuperation des objets et la configuration de résolution des dépendances. Il est à noter que par rapport à Unity (qui est le seul autre framework gérant les deux types d'initialisation), l'initialisation d'ObjectFactory dans le cadre d'une configuration basée sur XML se fait automatiquement.

Comme pour les trois précédents framework, on va voir comment configurer StructureMap selon les trois modes disponibles. On va commencer par voir la méthode XML. Pour cela, on va ajouter un fichier StructureMap.config a la racine du site web/au niveau de l'exe.

```
<?xml version="1.0" encoding="utf-8" ?>
<StructureMap MementoStyle="Attribute">
  <DefaultInstance
      PluginType="ServiceLayer.IProductService, ServiceLayer"
      PluggedType="ServiceLayer.ProductService, ServiceLayer" />
  <DefaultInstance
      PluginType="RepositoryLayer.IProductRepository,RepositoryLayer"
      PluggedType="RepositoryLayer.ProductRepository,RepositoryLayer" />
</StructureMap>
```

On remarquera que le fichier XML de StructureMap est le plus léger de tous ceux présentés dans ce tutoriel. On utilisera aussi un fichier XML pour la configuration basée sur les attributs, mais un fichier plus léger, qui sera le suivant :

```
<?xml version="1.0" encoding="utf-8" ?>
<StructureMap>
  <Assembly Name="StructureMapAttributs"/>
</StructureMap>
```

Ici, StructureMapAttributs est le nom de l'assembly sur laquelle on travaille. Si cette information n'est pas renseignée, le framework ne fonctionnera pas.

Il faut ensuite, dans ce cas, ajouter l'attribut [StructureMap.Pluggable("Default")] aux classes concrètes que l'on veut injecter, et [StructureMap.PluginFamily("Default")] aux interfaces, ce qui donne le code suivant :

```
    [StructureMap.PluginFamily("Default")]
    public interface IProductService {
        Product GetOneProductById(int id);
    }

    [StructureMap.Pluggable("Default")]
    public class ProductService : IProductService {

        private readonly IProductRepository _repository;

        public ProductService(IProductRepository repository) {
            _repository = repository;
        }

        public Product GetOneProductById(int id) {
            return new Product(_repository.GetOneProduct(id).Rows[0]);
        }
    }
}
```

Ce mode de configuration ne m'enchante personnellement pas. En effet, ce mode force chacun des objets qui sera concerné par l'IoC à avoir une référence sur StructureMap, ce qui me ferme la porte au cas où je voudrais changer de framework en cours de développement. De plus, ces informations sont disséminées dans l'application, au contraire du fichier XML, qui est centralisé.

On préférera en général le mode de configuration "fluide" aux deux précédents, pour les raisons mentionnées précédemment de résolution à la compilation. Dans le cas de StructureMap, cette configuration se fait ainsi :

```
ObjectFactory.Initialize(clientObject =>
       {
           clientObject.ForRequestedType<IProductService>().
               TheDefaultIsConcreteType<ProductService>();
           clientObject.ForRequestedType<IProductRepository>().
               TheDefaultIsConcreteType<ProductRepository>();
       });
```

Les gros avantages de StructureMap sont (par rapport a Ninject) sa versatilité et (par rapport aux autres) sa simplicité d'usage. Il a aussi l'avantage d'être de façon générale le plus rapide de tous les framework présentés.
J'ai plutôt du mal à trouver des mauvais côtés à StructureMap, mais je suis biaisé ;). À la limite on pourrait parler du risque, comme c'est un projet open-source, de le voir disparaitre dans la nature.

Ce framework est disponible à l'adresse suivante : **http://structuremap.sourceforge.NET/**

IV - Le framework CommonServiceLocator

On a vu qu'utiliser un framework d'IoC permet de réduire les dépendances entre les différents composants de notre application, pour en augmenter la maintenabilité. Or, si on pousse la logique à l'extrême, il nous restera toujours une dépendance, quoi que l'on fasse... sur le framework d'IoC. Autant, sur un projet "standard", cela peut ne pas être un problème, autant on risque de trouver des cas pour lesquels on veut pouvoir faciliter le passage d'un framework IOC à l'autre. Par exemple, si une bibliothèque ou un framework est développé par une entité A qui utilise StructureMap, une entité B qui voudra utiliser le framework héritera d'une dépendance sur StructureMap, même si elle utilise déjà Ninject (et l'exemple n'est pas anodin, car c'est de là que tout est parti, à savoir d'une dépendance de Fluent NHibernate sur StructureMap).

Le projet CommonServiceLocator, **disponible sur Codeplex**, est justement la réponse à cette problématique. Il permet en effet d'utiliser une interface de plus haut niveau, et d'agir comme un pattern Adapter pour rerouter les appels au Service Locator vers le framework d'IoC choisi.

L'interface du CommonServiceLocator est très simple (voire simpliste), et ne concerne que la récupération des objets, la configuration des résolutions de dépendance restant le travail du framework d'injection. La voici dans toute sa splendeur :

```
namespace Microsoft.Practices.ServiceLocation
{
    public interface IServiceLocator : IServiceProvider
    {
        object GetInstance(Type serviceType);
        object GetInstance(Type serviceType, string key);
        IEnumerable<object> GetAllInstances(Type serviceType);

        TService GetInstance<TService>();
        TService GetInstance<TService>(string key);
        IEnumerable<TService> GetAllInstances<TService>();
    }
}
```

La communauté a déjà contribué des adaptateurs spécifiques pour un certain nombre de conteneurs (**Castle Windsor**, Spring .NET, Unity, StructureMap, **Autofac** et **MEF**). Ces Adaptateurs permettent, dans le cas d'un framework, de laisser l'utilisateur choisir quel framework il veut utiliser, et de créer, par exemple, un locateur spécifique à StructureMap ou à Unity, et de l'utiliser comme locateur par défaut.

Pour cela, on va, dans un premier temps, créer un Locator spécifique (à StructureMap dans cet exemple, grâce à l'adaptateur StructureMapServiceLocator disponible sur le site du projet) :

```
    public static class StructureMapLocator {

        public static IServiceLocator CreateServiceLocator() {
            Registry registry = new Registry();

 registry.ForRequestedType<IProductRepository>().TheDefaultIsConcreteType<ProductRepository>();
            registry.ForRequestedType<IProductService>().TheDefaultIsConcreteType<ProductService>();
            IContainer container = new Container(registry);

            return new StructureMapServiceLocator(container);
```

```
        }
    }
```

Puis, on va, au niveau de notre code, déclarer l'utilisation de notre locateur comme ServiceLocator actuel, et appeler la fonction GetInstance du ServiceLocator pour effectuer la résolution de type :

```
var structureMapLoc = StructureMapLocator.CreateServiceLocator();
ServiceLocator.SetLocatorProvider(() => structureMapLoc);

var monClient = ServiceLocator.Current.GetInstance<IProductService>().
    GetOneProductById(1);
Console.WriteLine(monClient);
```

Le CommonServiceLocator est une couche d'indirection supplémentaire, mais permets, s'il est bien utilisé, de changer facilement de framework IoC si le besoin s'en fait sentir (comme, par exemple, une nouvelle version du framework A, dont les performances sont nettement supérieures au framework B).

V - Conclusion

Any problem in computer science can be solved with another layer of indirection. But that usually will create another problem
David Wheeler

J'espère que ce petit tour d'horizon vous aura permis d'apprendre au moins deux ou trois choses (moi en tout cas, j'en ai appris plein ;)).

Pour conclure, l'injection de dépendances est une technique très intéressante pour augmenter le degré d'indépendance des différents éléments manipulés durant un développement. Dans cet article, on a vu les grandes lignes de ce principe, et quelques frameworks permettant de le faciliter, mais ce n'est que le début du voyage. Je vous encourage donc à relever le "défi" d'incorporer dans votre prochain développement cette technique, et de voir où cela vous mène.

Un point à souligner tout de même. L'injection de dépendances n'est évidemment pas une panacée, et il faut retenir que, dans l'absolu, cette technique ne sert à rien si:

- on n'aura jamais besoin d'une autre implémentation.
- on n'aura jamais besoin d'une configuration différente
- on n'aura jamais besoin d'isoler un ou plusieurs composants (et pas de test unitaires)

Enfin, quels que soient les avantages de cette technique, la résolution est forcément légèrement plus coûteuse que la création d'un nouvel objet. Il peut donc être néfaste, dans des cas limites de créations de grandes quantités d'objets, d'utiliser l'injection de dépendances.

Toutes les sources utilisées dans cet article, ainsi que le projet (pour Visual Studio 2008) sont disponibles **ici**.

VI - Remerciements

Je remercie toute l'équipe Dotnet de developpez pour leurs relectures attentives du document. En particulier, merci à **tomlev** pour sa relecture.

VII - Contact

Si vous constatez une erreur dans le tutoriel, ou pour toute information, n'hésitez pas à me contacter par **mail**, ou sur le **forum**.