Dependency inversion, inversion of control et dependency injection : ne confondons pas tout !

Dependency inversion, inversion of control et dependency injection, ces trois termes vous sont sans doute familiers. Pourtant, bon nombre de développeurs confondent ces trois notions. Bien qu'elles soient fortement liées, elles n'en restent pas moins indépendantes. Explications.

Dans cet article, nous allons essayer de distinguer les différences entre ces trois notions : Dependency Inversion, Inversion of Control et Dependency Injection. En tant que développeurs, vous avez forcément entendu au moins une fois l'un de ces termes. Etaient-ils correctement utilisés ? Cet article, qui s'appuiera sur différents exemples en C#, devrait vous éclairer.

> Dependency Inversion Principles

Comme leur nom l'indique, les Dependency Inversion Principles (DIP) sont avant tout des principes, c'est-à-dire des éléments sur lesquels un développeur va pouvoir se baser pour concevoir son application. Ces principes ont été introduits par Robert Cecil Martin, un consultant américain, également développeur impliqué dans les méthodes Agiles et Extreme Programming. En 1996, ce dernier inventait et décrivait les principes d'inversion de dépendances dans un papier pertinemment intitulé « The Dependency Inversion Principle ». Au sein de celui-ci, Robert C. Martin décrit dans un premier temps ce qui, selon lui, représente une mauvaise conception d'un logiciel. Il définit notamment la notion de rigidité : le fait que le changement d'une partie d'un logiciel affecte bien souvent de nombreuses autres parties. Sa solution est alors de concevoir un système au sein duquel chacun des modules serait entièrement interdépendant.

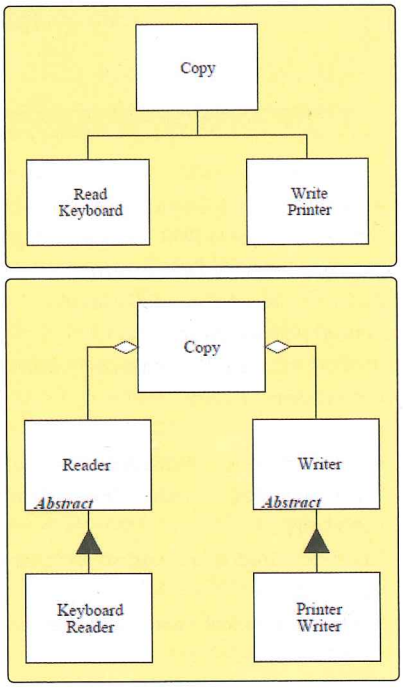
Robert C Martin illustre ses propos à l'aide d'un petit programme contenant un moyen de saisir du texte, un moyen d'écrire ce texte, le tout contrôlé par un module nommé Copy [Fig.1].

Les modules de bas niveau, ici le clavier et l'imprimante, sont totalement indépendants. Ce n'est pas le cas du module « Copy » qui, pour fonctionner, a absolument besoin des deux modules sous-jacents. Si le programme doit évoluer pour prendre en compte de nouveaux modules de bas niveau, alors le module Copy devra lui aussi forcément évoluer. En appliquant le principe d'inversion de dépendances, le programme pourrait évoluer de façon à ce que le module Copy ne repose plus directement sur les couches de bas niveau, mais sur des abstractions de ces dernières [Fig.2].

Désormais le module Copy s'appuie sur une couche d'abstraction « reader » d'un côté et « writer » de l'autre. Chacune de ces couches peut alors fournir un ensemble d'implémentations, au-delà du clavier et de l'imprimante, sans pour autant impacter le module de haut niveau Copy. Les principes d'inversion de dépendances sont ainsi définis par deux règles :

- Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Chacun doit se reposer sur une abstraction.
- Les abstractions ne doivent pas reposer sur des détails. Ce sont les détails qui doivent reposer sur des abstractions.

En appliquant ces deux principes, les dépendances habituelles d'un logiciel se trouvent alors inversées. Les modules de haut niveau ne se reposent plus sur ceux de bas niveau, mais sur des abstractions de ces derniers. Toutefois, il ne s'agit ici que de principes, c'est-à-dire de théorie.



> Inversion of Controls

L'inversion de contrôles (IoC) peut être vue comme l'implémentation pratique des principes d'inversion de dépendances. Le terme inversion de contrôles représente en fait un ensemble de pattern, donc indépendants des langages et des frameworks, permettant de découpler les différents modules d'une application. Concrètement, ces patterns vont nous permettre de modifier la façon dont fonctionne un module normalement, tout en en tirant des bénéfices. On distingue trois familles d'IoC : les inversions d'interface, les inversions de flow et les inversions de création. Les inversions d'interface représentent la mise en application directe de la règle DIP disant qu'un module de haut niveau ne doit pas dépendre d'un module de bas niveau. Prenons l'exemple d'une application 3-tiers permettant de gérer des produits. Ces produits peuvent être stockés soit au sein d'une base de données, soit au sein de fichiers XML. De base, la couche d'accès aux données pourrait contenir les classes suivantes.

```
public class DatabaseProduct
{
 private ProductsEntities _context;

 public DatabaseProduct()
 {
```

```
    _context = new ProductsEntities();
  }

  public void AddProduct(Product newProduct)
  {
    _context.Products.AddObject(newProduct);
    _context.SaveChanges();
  }

  public Product FindProduct(int productId) {}

  public void DeleteProduct(Product productToDelete) {}

  public void UpdateProduct(Product productToUpdate) {}
}

public class XmlProduct
{
  private XDocument _doc;

  public XmlProduct()
  {
    _doc = XDocument.Load(«products.xml»);
  }

  public void AddProductXML(Product newProduct) {}

  public Product FindProductXML(int productId) {}

  public void DeleteProductXML(Product productToDelete) {}

  public void UpdateProductXML(Product productToUpdate) {}
}
```

La couche logique ferait alors appel à l'une de ces classes de la façon suivante

```
public class ProductLogic
{
  private DatabaseProduct _database;

  public ProductLogic()
  {
    _database = new DatabaseProduct();
    Product product = _database.FindProduct(1);
  }
}
```

Ici, le module de haut niveau ProductLogic repose essentiellement sur la classe de bas niveau DatabaseProduct (ou XMLProduct). L'inversion d'interfaces va nous permettre d'inverser cette dépendance en rendant la couche de haut niveau indépendante des deux autres. Cela va passer par la création d'une interface, ici nommée IStoreProduct, que vont implémenter XmlProduct et DatabaseProduct.

```
public interface IStoreProduct
{
    void AddProduct(Product newProduct);
    void DeleteProduct(Product productToDelete);
    Product FindProduct(int productId);
    void UpdateProduct(Product productToUpdate);
}

public class XmlProduct : IStoreProduct
{
}
public class DatabaseProduct : IStoreProduct
{
}
```

Désormais, la classe ProductLogic ne dépendra plus directement des classes de bas niveaux, mais de leur abstraction, IStoreProduct.

```
public class ProductLogic
{
    private IStoreProduct _database;

    public ProductLogic()
    {
        _database = new DatabaseProduct();
    }
}
```

La seconde famille d'IoC est l'inversion de flow. Celle-ci est très peu connue, mais est pourtant utilisée par tous. Prenons l'exemple d'une application console permettant de gérer nos produits. Pour fonctionner, celle-ci va vous demander plusieurs informations sur le produit en question. Vous allez devoir renseigner un nom, un prix, puis enfin une description. En agissant ainsi, c'est l'application qui contrôle son propre flow. En utilisant une interface graphique au lieu d'une application console, ce n'est plus l'application qui va gérer la circulation des informations, mais l'utilisateur. En effet, ce dernier va alors pouvoir compléter un formulaire pour décrire son produit, puis, lorsqu'il aura fini, valider celui-ci, pour que l'application se charge du reste.

La troisième et dernière famille correspond à l'inversion de création. Cette famille contient elle-même un ensemble de pratiques applicables : le pattern factory, le pattern service locator, et les injections de dépendances. L'objectif de l'inversion de création est d'externaliser la création d'un contrôle, c'est-à-dire de faire en sorte que celle-ci ne dépende pas de la classe qui utilise ledit contrôle. Dans l'exemple utilisé pour l'inversion d'interface, les modules de logique et d'accès aux données ont été découplés. Toutefois, la classe ProductLogic s'occupe toujours de la création de l'implémentation d'IStoreProduct, elle en est donc toujours dépendante. L'inversion de création va nous permettre de résoudre ce problème.

Le pattern factory consiste en la création d'un objet indépendamment de son appelant. Reprenons l'exemple des produits et admettons maintenant que nous en ayons plusieurs types. Pour instancier l'un de ces produits, nous pourrions écrire ceci :

```
Product electronicProduct = new ElectronicProduct();
Product householdProduct = new HouseholdProduct();
```

En appliquant le pattern factory, ce ne serait plus l'appelant qui serait chargé de créer les différents produits, mais un module tiers. Nous pourrions ainsi obtenir ceci.

```
Product product = ProductFactory.GetProduct();
```

Dans ce cas, c'est la fabrique (factory) qui se charge de la création du produit. Désormais, c'est cette fabrique qui contiendra toute la logique permettant de créer les objets de type Product.
Le pattern service locator va lui aussi permettre de supprimer les dépendances entre les différents éléments d'une classe. Il va fonctionner plus ou moins de la même façon que le pattern factory, à la différence qu'il nous renverra une instance d'un objet en fonction d'un type passé. Généralement, les services locator contiennent un dictionnaire interne, permettant de mapper un type donné à une instance d'une classe. Ainsi, la récupération d'un produit via un service locator pourrait ressembler à ceci :

```
Product product = ProductServiceLocation.Retrieve<ElectronicProduct>();
```

En fonction du type passé, ici ElectronicProduct, le service locator nous renverra une instance d'un certain objet, ici un produit.

> Dependency injections

Les injections de dépendances sont une autre forme d'inversion de contrôle. Elles ont pour but, encore une fois, d'externaliser la création d'un objet, ceci en injectant l'instance d'un objet au sein du module devant utiliser cet objet. Cette injection peut se faire, via le constructeur, via un setter ou via une méthode.
Reprenons l'exemple d'inversion d'interface en utilisant cette fois une injection de dépendance par le constructeur.

```
public class ProductLogic
{
    private IStoreProduct _storeProduct;

    public ProductLogic(IStoreProduct storeProduct)
    {
        _storeProduct = storeProduct;
    }
}
```

Désormais, ce n'est plus la classe ProductLogic qui est chargée d'instancier l'objet _storeProduct, mais son appelant. Ce dernier a alors pour charge d'injecter l'implémentation d'IStoreProduct au sein du constructeur de ProductLogic. L'appelant fonctionnerait alors de la façon suivante.

```
DatabaseProduct database = new DatabaseProduct();
ProductLogic logic = new ProductLogic(database);
```

Cette méthode a plusieurs avantages. Ici, une valeur est attribuée à _storeProduct lors de la construction de ProductLogic et possède donc forcément une valeur lors de son utilisation. De plus, cela permet d'utiliser facilement la classe ProductLogic pour faire des tests, en passant en paramètre à cette dernière un objet n'accédant pas à la base de données (mocking). Son principal inconvénient provient de son avantage : une fois que l'injection est faite, il n'est plus possible de modifier ladite dépendance. L'objectif de l'injection par le setter est le même. Cette fois, ce n'est pas par le constructeur que l'injection va se faire, mais par une propriété.

```
public class ProductLogic
{
    private IStoreProduct _storeProduct;

    public ProductLogic()
    {}

    public IStoreProduct StoreProduct
    {
        get { return _storeProduct; }
        set
        {
            _storeProduct = value;
        }
    }
}
```

Son utilisation devient alors la suivante.

```
ProductLogic logic = new ProductLogic();
logic.StoreProduct = new DatabaseProduct();
```

Cette seconde façon de faire à l'avantage de laisser à l'appelant la possibilité de ne pas spécifier de paramètre pour le constructeur, rendant ainsi la classe plus modulable.
L'inconvénient cette fois, c'est que l'on n'est pas sûr que la dépendance existe lorsque l'on souhaite l'utiliser. En effet, rien n'empêche l'appelant d'utiliser une des méthodes de ProductLogic avant de passer une dépendance à celle-ci via le setter.
Enfin, l'injection par méthode. Celle-ci possède exactement les mêmes avantages et inconvénients que l'injection par setter. La façon d'exécuter cette injection est la même que la précédente, à la différence que, cette fois, ce n'est plus une propriété mais une méthode qui est utilisée.

> Conclusion

Vous l'avez compris, ces différentes pratiques nous permettent de découpler radicalement les différents modules d'une application. Les Dependency Inversion Principles correspondent aux idées de base, les Inversion of Controls aux implémentations pratiques de ces idées, et les dependency injections à un type d'inversion de contrôles. Attention toutefois à ne pas tomber dans l'excès en utilisant à tout va ces différentes notions.
Quelle que soit l'application que vous développiez, ces différentes techniques vont fatalement vous donner plus de travail. Inutile d'abstraire une classe d'accès aux données si vous savez pertinemment que le moyen actuel (une base de données par exemple) ne sera jamais remplacé.
Inutile également de réaliser une injection de dépendance si la dépendance en question est, au final, toujours la même. A vous, développeur, de vous adapter selon l'application.

Loïc **Rebours**
Etudiant en Master 2 à Supinfo Metz
Formateur .NET et Microsoft Student Partner
http://www.blog.loicrebours.fr