


Bonnes pratiques objet en .net : Introduction aux principes SOLID

Par Philippe Vialatte 

Date de publication : 21 octobre 2008

Dans cet article, je vais essayer de vous présenter les principes SOLID, tels que décrits dans le livre de Robert Martin,  **Agile Software Development, Principles, Patterns, and Practices**. On va essayer de voir l'intérêt de ces principes, et comment les appliquer, de façon (si possible) abordable par tout le monde.
Commentez cet article : [Commentez](#)

| | |
|---|----|
| I - Introduction..... | 3 |
| I---A - Cohésion..... | 3 |
| I---B - Couplage..... | 3 |
| I---C - Encapsulation..... | 3 |
| II - Responsabilité unique (SRP: Single Responsibility Principle)..... | 4 |
| II---A - Définition..... | 4 |
| II---B - Comment l'appliquer..... | 4 |
| II---B.1 - Analyse et regroupement des méthodes..... | 4 |
| II---B.2 - Analyse du code..... | 4 |
| II---C - Exemple..... | 4 |
| III - Ouvert/fermé (OCP: Open/closed Principle)..... | 7 |
| III---A - Définition..... | 7 |
| III---B - Comment l'appliquer..... | 7 |
| III---C - Exemple..... | 8 |
| IV - Substitution de Liskov (LSP: Liskov Substitution Principle)..... | 9 |
| IV---A - Définition..... | 9 |
| IV---B - Comment l'appliquer..... | 10 |
| IV---C - Exemple..... | 10 |
| V - Séparation des Interfaces (ISP: Interface Segregation Principle)..... | 11 |
| V---A - Définition..... | 11 |
| V---B - Comment l'appliquer..... | 12 |
| V---C - Exemple..... | 12 |
| VI - Inversion des dépendances (DIP: Dependency Inversion Principle)..... | 12 |
| VI---A - Définition..... | 12 |
| VI---B - Comment l'appliquer..... | 13 |
| VI---C - Exemple..... | 13 |
| VII - Conclusion..... | 14 |
| VIII - Remerciements..... | 15 |

I - Introduction

Aujourd'hui, une majorité des développeurs développe avec des langages orientés objet.

L'objet est partout, dans tous (ou presque) les langages, et tous les développeurs comprennent intimement ce qu'on entend par de la programmation orientée objet...ou pas.

Après avoir passé de nombreuses années à maintenir et développer du code, on se rend malheureusement compte que les principes du développement objet sont malheureusement soit ignorés, soit mal compris par de nombreux développeurs, ce qui rend assez souvent la maintenance des logiciels au mieux malaisée, au pire impossible.

SOLID est l'acronyme de cinq principes de base (**S**ingle Responsibility Principle, **O**pen/Closed Principle, **L**iskov Substitution Principle, **I**nterface Segregation Principle et **D**ependency Inversion Principle) que l'on peut appliquer au développement objet.

On verra dans cet article que ce sont avant tout des principes de **bon sens**. Aucun ne nécessite une connaissance approfondie d'un langage donné. Pour des raisons de goût, les exemples seront donnés en C#. Ces principes, lorsqu'ils sont compris et suivis, permettent d'améliorer la cohésion, de diminuer le couplage, et de favoriser l'encapsulation d'un programme orienté objet.

Avant d'attaquer les principes en eux-mêmes, on va brièvement revoir à quoi correspondent ces métriques.

I---A - Cohésion

La cohésion traduit à quel point les pièces d'un seul composant sont en relation les unes avec les autres. Un module est cohésif lorsqu'au haut niveau d'abstraction il ne fait qu'une seule et précise tâche. Plus un module est centré sur un seul but, plus il est cohésif.

I---B - Couplage

Le couplage est une métrique qui mesure l'interconnexion des modules. Deux modules sont dit couplés si une modification d'un de ces modules demande une modification dans l'autre.

I---C - Encapsulation

L'idée derrière l'encapsulation est d'intégrer à un objet tous les éléments nécessaires à son fonctionnement, que ce soit des fonctions ou des données.

Le corolaire est qu'un objet devrait (et non pas doit, comme c'est souvent expliqué) masquer la cuisine interne de la classe, pour exposer une interface propre, de façon à ce que ses clients puissent manipuler l'objet et ses données sans avoir à connaître le fonctionnement interne de l'objet.

Pour plus de clarté, on verra en quoi chaque principe va jouer sur ces métriques.



Comme l'indique le titre de cet article, c'est un article d'introduction. En conséquence, j'ai essayé de rester à un niveau abordable, ni trop théorique, ni trop pratique. Certaines formulations sont simplifiées, et les exemples restent volontairement à un niveau "scolaire".

II - Responsabilité unique (SRP: Single Responsibility Principle)

II---A - Définition

Ce principe est le plus vieux de ceux étudiés ici. Il se base sur les travaux de Tom DeMarco, en 1979, qui le qualifie de principe de cohésion. La définition que l'on va admettre est:

"Si une classe a plus d'une responsabilité, alors ces responsabilités deviennent couplées. Des modifications apportées à l'une des responsabilités peuvent porter atteinte ou inhiber la capacité de la classe de remplir les autres. Ce genre de couplage amène à des architectures fragiles qui dysfonctionnent de façon inattendues lorsqu'elles sont modifiées." -- Robert C. Martin

Le principe de responsabilité unique, réduit à sa plus simple expression, est qu'une classe donnée ne doit avoir qu'une seule responsabilité, et, par conséquent, qu'elle ne doit avoir qu'une seule raison de changer.

Les avantages de cette approche sont les suivants:

- Diminution de la complexité du code
- Augmentation de la lisibilité de la classe
- Meilleure encapsulation, et meilleure cohésion, les responsabilités étant regroupées

II---B - Comment l'appliquer

Bien que ce principe s'énonce assez facilement, c'est assez souvent le plus compliqué à mettre en oeuvre. En effet, on a souvent tendance à donner trop de responsabilités à un objet, et on a parfois du mal à identifier, sur un objet existant, les responsabilités qui lui échoient.

Une responsabilité peut être identifiée, dans un code existant, des façons suivantes:

II---B.1 - Analyse et regroupement des méthodes

Pour une classe de taille importante, il est souvent bénéfique de lister toutes les méthodes, et de regrouper celles dont le nom ou les actions semblent être de la même famille. Si plusieurs groupes apparaissent dans une classe, c'est un bon indicateur que la classe doit être reprise.

II---B.2 - Analyse du code

Une autre méthode est de regarder les dépendances externes de la classe.

La méthode appelle-t-elle directement la base de données ? Utilise-t-elle une API spécifique ? Certains membres sont-ils appelés uniquement par une fonction, ou par un sous-ensemble de fonctions ?

Si c'est le cas, ce sont peut-être des responsabilités annexes, dont il faut se débarrasser...

II---C - Exemple

Pour faire simple, on va prendre un mauvais exemple, que l'on va refactoriser.

On va se baser sur le scénario suivant : notre société veut concevoir un nouveau système de suivi de bugs. Pour cela, on va implémenter un système de suivi de Work Items. Après un premier jet, on va obtenir le code suivant:

```
public class WorkItem{
```

```
private string _id;
private string _name;

// accesseurs...pas la peine de s'appesantir...

public void Save(){
    SqlConnection
cnx = new SqlConnection(ConfigurationManager.ConnectionStrings["database"]);
    cnx.Open();

    SqlCommand cmd = new SqlCommand();
    cmd.Connection = cnx;
    cmd.CommandText = "INSERT INTO WorkItem (Id, Name) VALUES ('";
    cmd.CommandText += _id + "',' + _name + "')";
    cmd.ExecuteNonQuery();
    cnx.Close();
}

public void GetById(string id){
    SqlConnection
cnx = new SqlConnection(ConfigurationManager.ConnectionStrings["database"]);
    cnx.Open();

    SqlCommand cmd = new SqlCommand();
    cmd.Connection = cnx;
    cmd.CommandText = "SELECT Id, Name FROM WorkItem where Id = '" + id + "'";

    SqlDataReader dr = command.ExecuteReader();

    if (dr.Read()){
        _id = dr["id"].ToString();
        _name = dr["name"].ToString();
    }else{
        return null;
    }
}
}
```



*Ce fonctionnement correspond plus ou moins à un pattern d'architecture, le pattern **ActiveRecord**. Il n'est pas mauvais en soi, mais dans le cadre du principe SRP, pose quelques problèmes. A chacun de peser le pour et le contre de chaque approche...*

En termes de responsabilités, cette classe a les responsabilités

- de créer les objets
- de stocker les données de l'objet
- et de gérer la persistance des objets.

(On va passer sur le couplage fort entre la structure de la base et de l'objet, sur les risques d'injection, sur la répétition du code pour gérer les connexions, etc...)

Après refactorisation, on va obtenir trois objets, une Factory, un gestionnaire d'accès aux données, et un objet de transport de données.

```
public class WorkItem{

    private string _id;
    private string _name;
    // accesseurs...pas la peine de s'appesantir...

    public WorkItem(DataRow dr){

        _id = dr["id"].ToString();
        _name = dr["name"].ToString();
    }
}

public class WorkItemFactory{

    public void Save(WorkItem item){
        WorkItemDataAccess.Save(item.Id, item.Name);
    }
    public WorkItem GetWorkItemById(string id){
        DataRow dr = WorkItemDataAccess.GetById(id);
        if (dr == null){
            return null;
        }
        return new WorkItem(dr);
    }
}

public static class WorkItemDataAccess{

    public static void Save(string id, string name){
        /// requetes sql
    }
    public static DataRow GetById(string id){
        SqlConnection
        cnx = new SqlConnection(ConfigurationManager.ConnectionStrings["database"]);
        cnx.Open();

        SqlCommand cmd = new SqlCommand();
        cmd.Connection = cnx;
        cmd.CommandText = "SELECT Id, Name FROM WorkItem where Id = '" + id + "'";

        DataTable dt = new DataTable();
        using (SqlDataAdapter da = new SqlDataAdapter(cmd)) {
            da.Fill(dt);
        }

        return dt.Rows.Count == 0 ? null : dt.Rows[0];
    }
}
```

Suite a cette factorisation, les responsabilités de nos trois classes sont beaucoup plus évidentes, la classe d'accès aux données ne traite plus que des données, l'objet possède des méthodes pour manipuler ses propres données, et la factory a la responsabilité de faire travailler ensemble la classe d'accès aux données et l'objet...

La partie métier est un peu mince, mais on va essayer de l'étoffer plus tard.

Une notion à garder à l'esprit est qu'il ne faut pas aller trop loin dans la séparation des responsabilités, au risque de tomber dans un excès inverse et se retrouver avec un domaine anémique.

Un refactoring supplémentaire (et valide) pourrait faire intervenir un nouvel objet `CommonDataAccess`, qui serait appelé par les classes d'accès aux données

III - Ouvert/fermé (OCP: Open/closed Principle)

III---A - Définition

Le principe Ouvert/fermé est, tout comme la substitution de Liskov, issu d'un article datant d'une vingtaine d'années (1988). Sa formulation initiale est la suivante:

i *Les entités logicielles (classes, modules, fonctions, etc.) doivent être ouvertes pour l'extension, mais fermées à la modification. --Bertrand Meyer*

L'idée originale de Meyer était qu'en fait, une classe logicielle était un package de données, qui ne devait, une fois implémenté, ne plus être modifiée que pour corriger une erreur. Toute nouvelle fonctionnalité ne devrait, selon lui, pouvoir être rajouté qu'en ajoutant une nouvelle classe, laquelle pouvait éventuellement hériter de la classe d'origine. De nos jours, lorsque l'on parle de ce principe, le sens que l'on lui donne en général est celui repris dans un article de 1996 par Robert Martin. La définition étendue qui en est donnée est la suivante:

i *"Les modules qui se conforment au principe ouvert/fermé ont deux attributs principaux.
 1 - Ils sont "ouverts pour l'extension". Cela signifie que le comportement du module peut être étendu, que l'on peut faire se comporter ce module de façons nouvelles et différentes si les exigences de l'application sont modifiées, ou pour remplir les besoins d'une autre application.
 2 - Ils sont "Fermés à la modification". Le code source d'un tel module ne peut pas être modifié.
 Personne n'est autorisé à y apporter des modifications."
 --Robert C. Martin*

Le but de ce principe est donc de tendre, non plus vers des objets immuables, mais vers des objets auxquels les clients pourront ajouter de nouveaux comportements sans en modifier la mécanique interne.

La différence fondamentale entre les deux approches est que, dans un cas, on va utiliser un héritage d'implémentation, alors que dans l'autre, on va se baser sur des contrats.

Pour quelles raisons voudrait-on pouvoir mettre notre programme en conformité avec ce principe ?

- Plus de flexibilité par rapport aux évolutions
- Diminution du couplage

III---B - Comment l'appliquer

Deux possibilités nous sont données, la première étant de chercher à identifier une fonction ou une méthode dont le comportement est susceptible d'être fortement impacté par une modification ultérieure. Malheureusement, on va plus souvent qu'à notre tour être surpris par les demandes de nos clients. La seconde méthode, plus agile, est de conserver un design simple, et, lorsque l'on arrive aux limites de ce design, d'en changer...

On va donc, le plus souvent, commencer par le code le plus simple pouvant fonctionner, et, lorsque l'on rencontre une exception nous obligeant à modifier la classe pour l'étendre, s'assurer qu'une modification ultérieure de même type ne nous forcera pas à modifier de nouveau notre design. Evidemment, avec l'expérience, on est souvent plus à même de sentir quelles portions du programme sont plus à même d'être modifiées, mais c'est un pari que l'on fait.

Comme règles de bonne conduite, on peut essayer d'une part de ne pas dépendre du type d'un objet pour choisir un chemin de traitement. Cela se rapproche du principe LSP, que l'on va voir plus bas. D'autre part, on peut limiter l'héritage, en y préférant la composition.

Si vous vous intéressez un petit peu aux design patterns, vous devez certainement avoir déjà été exposé à ce principe...

En effet, un certain nombre de design patterns sont une mise en pratique de ce principe. Par exemple, le DP **Décorateur** permet de rajouter à un objet de nouvelles fonctionnalités sans modifier son code, par décoration, le DP **Visiteur** permet d'étendre les capacités de la collection parcourue sans modifier l'implémentation des objets manipulés, etc...

III---C - Exemple

On va reprendre notre exemple précédent de Work Item.

Après une itération, on va avoir une nouvelle demande de notre client (interne, mais client quand même), à savoir qu'il veut pouvoir gérer plusieurs types de work items. En effet, certains vont matérialiser des tâches pour le département informatique, d'autres, des tâches pour le département finance, marketing, ou gestion. Notre premier réflexe va être le suivant:

```
public class WorkItem{
    private string _id;
    private string _name;
}
public class ITWorkItem : WorkItem{
    public void ManageITWorkItem(){
        // execute des actions relatives au departement informatique
    }
}
public class FinanceWorkItem : WorkItem{
    public void ManageFinanceWorkItem(){
        // execute des actions relatives au departement finances
    }
}
public class MarketingWorkItem : WorkItem{
    public void ManageMarketingWorkItem(){
        // execute des actions relatives au departement marketing
    }
}
```

Tout cela semble marcher. Seulement, que se passe-t-il lorsque l'on veut traiter un ensemble de Work Items ?

```
public RunAllWorkItems(List<WorkItems> items){

    foreach(WorkItem item in items){

        if (item is ITWorkItem){
            item.ManageITWorkItem();
        }else if(item is FinanceWorkItem){
            item.ManageFinanceWorkItem();
        }else if(item is MarketingWorkItem){
            item.ManageMarketingWorkItem();
        }
    }
}
```

On se retrouve avec une implémentation fragile, qui va nécessiter, à chaque création d'un nouveau type de Work Item, la modification de la fonction RunAllWorkItems, et de toutes les fonctions qui se basent sur le type des Work Items.

Une solution conforme à OCP serait d'ajouter une interface `IWorkItem`, et de lui ajouter une fonction `ManageWorkItem`, `ManageWorkItem` devenant le contrat que chaque classe implémentant `IWorkItem` devra remplir.

```
public interface IWorkItem {
    void ManageWorkItem();
}

public class WorkItem {
    private string _id;
    private string _name;
    //on saute les accesseurs...
}

public class ITWorkItem : WorkItem, IWorkItem{
    public void ManageWorkItem(){
        // execute des actions relatives au departement informatique
    }
}

public class FinanceWorkItem : WorkItem, IWorkItem{
    public void ManageWorkItem(){
        // execute des actions relatives au departement finances
    }
}

public class MarketingWorkItem : WorkItem, IWorkItem{
    public void ManageWorkItem(){
        // execute des actions relatives au departement marketing
    }
}

.....
public RunAllWorkItems(List<IWorkItems> items){

    foreach(IWorkItem item in items){
        item.ManageWorkItem();
    }
}
```

De cette façon, on pourra ajouter de nouveaux types de comportement (de nouveaux Work Items) sans avoir à modifier la fonction `RunAllWorkItems`. On peut donc l'étendre sans avoir à la modifier.

IV - Substitution de Liskov (LSP: Liskov Substitution Principle)

IV---A - Définition

La substitution de Liskov, telle que définie par Barbara Liskov et Jeannette Wing, s'énonce ainsi :

Ce que l'on veut est vérifier la propriété de substitution suivante:



Si pour chaque objet o1 de type S il existe un objet o2 de type T tel que pour tout programme P défini en termes de T, le comportement de P est inchangé quand on substitue o1 à o2, alors S est un sous-type de T

A cette définition fonctionnelle légèrement alambiquée, je préfère, une fois de plus, la définition simplifiée donnée par Robert Martin:



Les sous-types doivent être remplaçables par leur type de base.

Plus simple, non ;) ?

La, je vais en voir un ou deux (ou plus) dire: "Oui, mais à partir du moment où ma classe S hérite de ma classe T", je dois pouvoir caster S en T et là ça va marcher...

Justement, non...

Le but de ce principe est exactement de pouvoir utiliser une méthode sans que cette méthode ait à connaître la hiérarchie des classes utilisées dans l'application, ce qui veut dire:

- pas de cast
- pas de as
- pas de is
- et surtout pas d'introspection/réflexion

En effet, si on vérifie le type des objets, on va violer non seulement LSP, mais aussi OCP, il suffit de voir l'exemple précédent de violation d'OCP, qui était une violation flagrante de LSP. L'inverse ne se vérifie pas, on peut tout à fait ne pas respecter OCP (l'exemple canonique étant de vouloir ajouter une routine de tri à nos objets, ce qui demande généralement de modifier les classes...), et rester en conformité avec LSP.

Ce principe apporte:

- Augmentation de l'encapsulation
- Diminution du couplage. En effet, LSP permet de contrôler le couplage entre les descendants d'une classe et les clients de cette classe.

IV---B - Comment l'appliquer

Pour détecter le non respect de ce principe, on va se poser la question de savoir si on peut, sans dommage, remplacer la classe en cours par une interface d'un niveau supérieur.

Le problème auquel on va se heurter est que, pour valider que le principe LSP est respecté, il faut le valider pour tous les clients de notre arborescence de classe. L'exemple canonique est le suivant: Si une classe Carré hérite de Rectangle, un client manipulant exclusivement des rectangles risque de vouloir affecter une largeur et une hauteur différentes à un carré. Soit le carré sera inconsistant (hauteur et largeur différentes), soit la fonction de surface sera inconsistante (le client attendant une surface de H*L, et recevant H*H ou L*L).

Ce principe nous invite à revoir la notion d'héritage, dans le sens où il définit la notion d'héritage comme celui du **comportement** qu'un client peut attendre d'un objet. Dans ce cas, on voit qu'en termes de développement objet, un carré n'est pas un rectangle, car leurs comportements diffèrent pour un client donné.

IV---C - Exemple

En dehors de l'utilisation d'un déterminant de type (is, as, réflexion), on va violer ce principe si un descendant modifie le comportement de la classe parente de façon à ce que le seul moyen, pour le programme, de fonctionner correctement, est de connaître le type concret de l'objet.

Pour continuer sur notre lancée, on va redéfinir la méthode ToString de nos Work Items.

Pour cela, on va définir, au niveau de notre classe WorkItem, une fonction ToString, qui va renvoyer l'id et le nom d'un work item.

```
public class WorkItem {
    private string _id;
    private string _name;
    //on saute les accesseurs...
    public override string ToString(){
        // renvoie
        return "id : " + _id + " - name : " + _name;
    }
}
```

Une violation du principe LSP serait d'avoir, au niveau des `FinanceWorkItem`, par exemple, un override de `ToString` dans ce gout-la:

```
public override string ToString(){
    throw new Exception("Pas de ToString, Merci !");
}
```

(Je sais, c'est moyennement utile, mais c'est pour l'exemple).

Inversement, à mon avis, LSP ne doit pas forcer les descendants d'une classe à conserver le même comportement métier.

Par exemple, si dans mes `FinanceWorkItem`, je fais:

```
public override string ToString(){
    return "id : " + _id + " name : " + _name;
}
```

Et qu'un de mes clients utilise une fonction de manipulation de chaîne de caractères pour récupérer le nom du `WorkItem`, comme :

```
string name = workItem.ToString().Split('-')[1].Substring(8);
```

La nouvelle version provoquera une exception chez le client, mais ne sera pas une violation de LSP, le comportement restant le même (on retourne une représentation de l'objet sous forme de chaîne), mais le contenu retourné change pour correspondre à une implémentation spécifique.

V - Séparation des Interfaces (ISP: Interface Segregation Principle)

V---A - Définition

Pour une fois, ce principe se passe d'une définition formelle, ce qui y ressemble le plus étant la définition suivante:



Les clients d'une entité logicielle ne doivent pas avoir à dépendre d'une interface qu'ils n'utilisent pas.

Le premier effet d'avoir une interface trop compliquée va se ressentir assez vite. En effet, toute classe implémentant une interface doit implémenter chacune de ses fonctions. On va donc très vite se retrouver avec une confusion sur le rôle des sous classes.

Dans le framework .net (on pourrait certainement trouver des exemples en Java et C++), de nombreuses classes se conforment à ce principe. Par exemple, la classe générique `List<T>` va implémenter toutes les interfaces suivantes:

- `IList<T>`
- `ICollection<T>`
- `IEnumerable<T>`
- `IList`
- `ICollection`
- `IEnumerable`

Les risques d'avoir des interfaces trop compliquées ne sont pas toujours évidents. En fait, ils reviennent au poids que l'on compte mettre sur les clients de l'interface. En effet, de façon générale, plus une interface est compliquée,

plus l'abstraction qu'elle présente est vaste, plus elle est susceptible d'évoluer avec le temps. Chaque évolution de l'interface va entraîner une livraison ou une modification de l'ensemble des clients.

En conséquence, si l'interface doit changer pour un client A qui utilise la fonction A, cela va aussi impacter le client B, qui n'utilise pas la fonction A, mais qui dépend de la même interface

Ce principe apporte principalement une diminution du couplage entre les classes (les classes ne dépendant plus les unes des autres). L'autre avantage d'ISP est que les clients augmentent en robustesse.

V---B - Comment l'appliquer

Appliquer ISP est assez simple dans la théorie. En fait, on va revenir à plus ou moins la même démarche que pour SRP. On va réunir les groupes "fonctionnels" des méthodes de la classe dans des Interfaces séparées. L'idée étant de favoriser le découpage de façon à ce que des clients se conformant à SRP n'aient pas à dépendre de plusieurs interfaces.

V---C - Exemple

Dans nos exemples de Work Items, on va devoir gérer des Work Items pour lesquels il existe une deadline. Nos Work Items dépendant tous de IWorkItem, on va directement ajouter Les informations de gestion de deadline au niveau de IWorkItem et de WorkItem.

Après un petit brainstorming, on fait une mise à jour de nos classes, et on obtient le code suivant:

```
public interface IWorkItem {
    ...
    bool IsDeadlineExceeded();
}
public class WorkItem {
    private DateTime _deadline;
    //on saute les accesseurs...
}
```

Jusqu'ici, tout va bien...Sauf que le marketing ne veut pas entendre parler de deadline pour ses items. On peut donc, soit renvoyer une information erronée, pour continuer à utiliser le IWorkItem courant, soit se conformer au principe ISP, et séparer notre interface en IWorkItem et IDeadLineDependent.

```
public interface IWorkItem {
    void ManageWorkItem();
    string ToString();
}
public interface IDeadLineDependent {
    bool IsDeadlineExceeded();
}
```

L'intérêt est que, si demain on a besoin d'une fonction ExtendDeadline dans IDeadLinedItem, cela n'impactera pas les WorkItems ne comportant pas de Deadline. Et si on ne le modifie pas, on n'introduit pas de bugs.

VI - Inversion des dépendances (DIP: Dependency Inversion Principle)

VI---A - Définition

Le principe d'inversion des dépendances est un peu un principe secondaire. En effet, il résulte d'une application stricte de deux autres principes, à savoir les principes OCP et LSP. Sa définition est la suivante:



*Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.*

Par module de haut niveau, on va entendre les modules contenant les fonctionnalités métier, les modules de bas niveau gérant la communication entre machines, les logs, la persistance. Si on change le mode de fonctionnement de la base (passage de Oracle à SQL Server), du réseau (changement de protocole), de système d'exploitation, les classes métiers ne doivent pas être impactées. Inversement, le fait de changer les règles de validation au niveau de la partie métier du framework ne doit pas demander une modification de la base de données (à la limite, modifier une fonction, mais ne pas changer les briques de base).

Ce principe apporte:

- Une nette diminution du couplage
- Une meilleure encapsulation, l'implémentation concrète pouvant éventuellement être choisie dynamiquement



*Ce principe est équivalent au principe d'Hollywood ("Ne nous appelez pas, nous vous appellerons"), qui est une forme plus générale d'inversion de contrôle.
Pour plus d'information sur ce principe, rendez-vous ici : [\(Wikipedia\) : Inversion de contrôle](#)*

VI---B - Comment l'appliquer

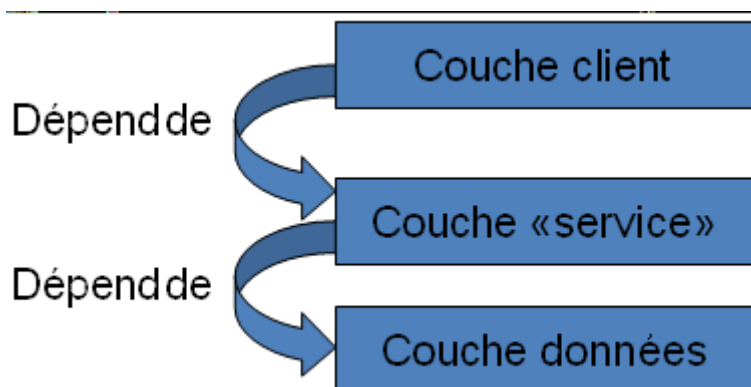
L'idée est que chaque point de contact entre deux modules soit matérialisé par une abstraction.

Par abstraction, on va entendre généralement, une interface, mais on peut aussi considérer qu'une classe (une factory, par exemple) représente une abstraction d'une classe de niveau plus élevé ou plus bas.

VI---C - Exemple

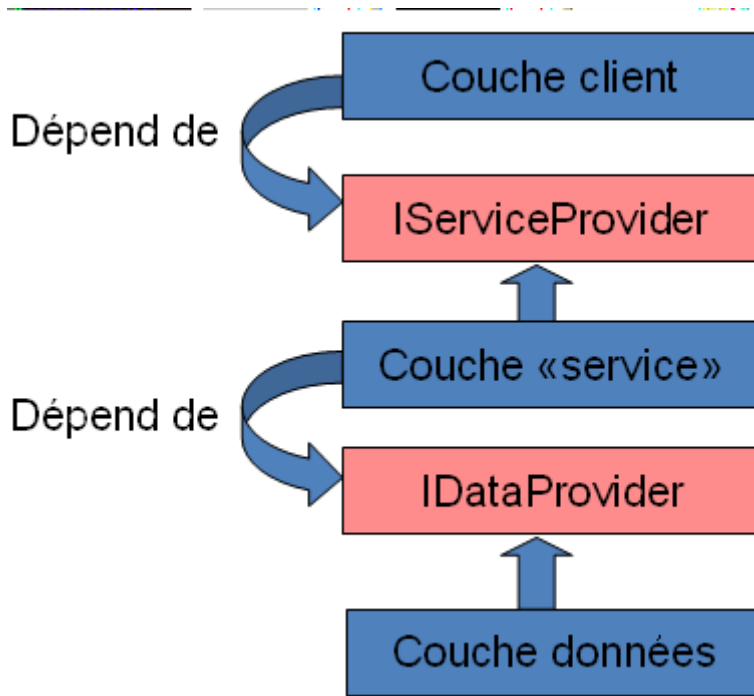
On va reprendre notre couche d'accès aux données, et la factory qui la manipule. Actuellement, les fonctions de chargement et de sauvegarde des objets métier possèdent une dépendance forte sur la couche d'accès aux données.

On a donc un diagramme de classe équivalent a:



Par conséquent, on va avoir, dans l'état, du mal à isoler nos fonctions de création d'objets de notre couche de données. Donc, on va avoir du mal à les tester.

Pour se conformer au principe DIP, on va modifier notre code de cette façon:



Pour aller encore plus près du principe DIP, on pourrait ajouter encore une couche d'abstraction supplémentaire, en retournant des interfaces aux couches hautes.

Une fois notre code factorisé, va se poser la question de comment affecter la bonne implémentation de nos interfaces au bon composant.

Une première approche est de passer à notre client une classe service choisie, ainsi qu'une référence sur un objet d'accès aux données. Cette approche va, en fait, déporter le choix de l'implémentation au niveau de la construction du client. Elle peut, à mon avis, être utilisée pour des projets de petite taille. Pour des projets plus conséquents, une seconde option est d'utiliser un configurateur externe, qui va, durant l'exécution de notre code, renvoyer la bonne implémentation concrète de notre abstraction. Un certain nombre d'outils d'injection de dépendance sont disponibles et matures, je vous renvoie à Google pour plus d'information sur ces outils. Vous trouverez, par exemple, [StructureMap](#), [Castle Windsor](#), ou, chez Microsoft, [Unity](#).

L'idéal est de chercher à tendre vers ce principe, pas forcément de l'appliquer à la lettre, ce qui peut se montrer contre-productif.

En effet, pour se conformer à 100% à ce principe, il faudrait que:

- aucune variable ne référence une classe concrète
- aucune classe ne dérive d'une classe concrète
- aucune classe ne réécrit une méthode d'une de ses classes de base.

VII - Conclusion

Ces principes peuvent s'énoncer clairement, les utiliser demande de les conserver à l'esprit durant chaque session de développement, et le coût initial d'introduction peut être décourageant pour certains projets.

Pour cette raison, ils sont souvent utilisés conjointement avec une méthodologie Agile, qu'elle soit XP, Scrum ou autre, supportée en tout cas par une panoplie de tests unitaires. En conclusion, le choix de se conformer ou non à ces principes doit donc se faire en fonction du contexte du projet, et de l'importance que l'on donne à la qualité et à l'évolutivité du programme développé. Il est néanmoins toujours utile d'en avoir au moins entendu parler :).



*Pour un approfondissement avec des gens beaucoup plus intelligents que moi, je vous invite à vous rendre sur les forums de la communauté  **Alt.Net**, où j'ai, pour la première fois, entendu parler des principes SOLID et de lire le livre de  **Robert Martin**.*

VIII - Remerciements

Merci à **tomlev** pour sa première relecture (sans laquelle l'article serait sûrement moins intéressant), et à **Skalp** pour ses corrections