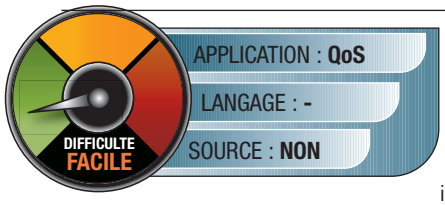
Un code de qualité est un code S.O.L.I.D.(E) !

Rigidité, fragilité, immobilité… Vous rencontrez ces obstacles dans votre code au quotidien ? Vous rêvez de développer un code robuste, extensible et réutilisable. Un mirage ? Non ! Les principes S.O.L.I.D. ! Ces principes vous sont expliqués et leur mise en application vous facilitera la vie.



Après avoir vu de nombreux projets, une évidence s'impose : les principes du développement objet sont souvent mal appliqués ou ignorés par beaucoup de développeurs, limitant l'évolutivité et la maintenance des applications. Bien que les bases de la programmation orientée objet soient simples, il y a souvent des problèmes dans leur application entraînant des bugs, une mauvaise structuration, un code non maintenable, une exécution longue, une compréhension difficile.

Les principes S.O.L.I.D., indépendants du langage, sont des principes permettant un meilleur design des classes dans la programmation orientée objet. Correctement appliqués, ces principes vont vous permettre de réduire les erreurs de conception et de structure amenant aux problèmes précédents.

S.O.L.I.D se compose de 5 principes :

- Single Responsibility Principle (Principe de la responsabilité unique)
- Open-Closed Principle (Principe Ouvert-Fermé)
- Liskov Substitution Principle (Principe de substitution de Liskov)
- Interface Segregation Principle (Principe de ségrégation des interfaces)
- Dependency Inversion Principle (Principe d'inversion de dépendance)

Le but est donc d'avoir un code de qualité grâce à un ensemble de bonnes pratiques. Vous trouverez dans cet article les lignes directives pour améliorer vos développements.

PRINCIPE DE LA RESPONSABILITÉ UNIQUE

Selon Robert Martin, plusieurs responsabilités au sein d'une même classe sont couplées. Ainsi, toute modification d'une responsabilité peut impacter l'ensemble de la classe. De ce couplage naît une architecture fragile dont les modifications peuvent entraîner des dysfonctionnements. En des termes plus simples, une classe doit gérer une responsabilité unique, une responsabilité étant un regroupement de fonctionnalités ayant un sens commun.

On a souvent tendance à donner plusieurs responsabilités à une classe. De fait, si on souhaite modifier le comportement d'une responsabilité au sein d'une classe, le comportement de l'ensemble des responsabilités de la classe risque d'être impacté.

Utiliser le principe de la responsabilité unique permet d'éviter les défauts de design, de bénéficier d'une meilleure lisibilité du code et d'une meilleure cohésion, d'abaisser le couplage et de garantir l'encapsulation de l'information. Ce principe semble simple, en tout cas sur le papier. Toutefois dans la vraie vie des projets, c'est un des principes les plus difficiles à respecter.

Afin de bien appliquer ce principe, il faut au préalable faire une première phase d'analyse en définissant l'ensemble des méthodes nécessaires à la réalisation des fonctionnalités attendues. A ce stade, nous avons une liste de méthodes non organisées entre elles. A partir de cette liste, on va regrouper les méthodes qui répondent à un besoin commun au sein d'une même classe. Il ne faut pas hésiter à créer autant de classes que nécessaire, même si certaines méthodes se retrouvent seules.

Pour faire évoluer le code, on respecte ce principe. Une nouvelle méthode correspondant à un besoin déjà existant sera rajoutée à la classe correspondante. Par contre, s'il s'agit d'un nouveau besoin on devra créer une nouvelle classe.

Il est bien entendu possible d'appliquer ce principe à du code existant. Prenons comme exemple une classe « Modem » qui regroupe des méthodes de connectivité ainsi que des méthodes d'envoi de données [Fig.1]. Ceci est une violation du principe de responsabilité unique. La solution dans cet exemple est de créer deux classes « Connection » et « DataChannel » regroupées au sein de la classe « Modem » [Fig.2].

Suite à cette optimisation, chaque classe contient une seule responsabilité. Ainsi, une modification sur chacune des classes n'impactera plus les autres classes.

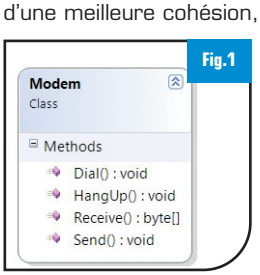
PRINCIPE OUVERT-FERMÉ

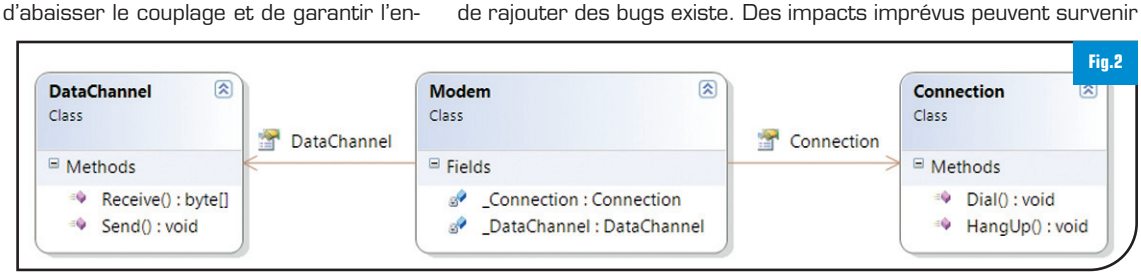
Selon Bertrand Meyer, les entités logicielles telles que les classes et les méthodes doivent être ouvertes pour l'extension, mais fermées à la modification.

C'est-à-dire que leur comportement peut être étendu ou bien complètement modifié si elles doivent remplir de nouveaux besoins (ouverte à l'extension) tout en s'assurant que leur code source ne peut pas être directement modifié, aucune modification n'est autorisée (fermé à la modification).

Après avoir lu cette définition, vous devez vous dire que cela semble contradictoire ! Comment une classe ou une méthode peut être à la fois ouverte pour extension et fermée à la modification ?!

Cela signifie simplement qu'une application doit être structurée de manière à pouvoir ajouter des fonctionnalités en touchant au minimum au code existant. À chaque modification de code, la possibilité de rajouter des bugs existe. Des impacts imprévus peuvent survenir





amenant à la modification du comportement de la classe et donc de l'application. L'implémentation devient fragile, difficilement maintenable et sujette aux régressions.
Prenons un exemple précis de non-respect du principe Ouvert-Fermé : afficher à l'écran deux formes graphiques, un carré et un cercle. L'implémentation comporte une seule classe nommée « Shape » avec une propriété « ShapeType » qui contient le type de formes graphiques voulues [Fig.3]. Le programme a une méthode « DrawAllShapes(…) » contenant le code nécessaire pour dessiner les différentes formes :

```
public void DrawAllShapes(List<Shape> shapeList)
{
  foreach (Shape obj in shapeList)
  {
    switch (obj.ShapeType)
    {
      case ShapeType.Square:
        DrawSquare(obj);
        break;

      case ShapeType.Circle:
        DrawCircle(obj);
        break;
    }
  }
}

public void DrawSquare(Shape obj) { /*Draw a square*/ }
public void DrawCircle(Shape obj) { /*Draw a circle*/}
```

Cette implémentation présente un défaut majeur. Si une nouvelle forme graphique doit être ajoutée, il est nécessaire de modifier la méthode « DrawAllShapes(…) ». Ceci alourdit le code, le rend illisible, sujet aux erreurs, difficile d'évolution, et il sera nécessaire de le re-tester dans son ensemble (surtout dans le cas où beaucoup de formes graphiques sont à gérer).
La solution est simple avec la programmation orientée objet : utiliser le principe de l'héritage.
Il suffit dans ce cas d'ajouter une classe abstraite qui contient la définition de la méthode « Draw() », mais qui ne contient pas d'implémentation. On crée ensuite une classe par forme graphique que l'on souhaite gérer. Les classes « Square » et « Circle » héritent des méthodes et des propriétés de la classe « Shape » [Fig.4]. Si maintenant on souhaite ajouter une forme graphique « Triangle », il suffit de créer une nouvelle classe qui hérite aussi de la classe « Shape » permettant d'éviter de faire des modifications dans le code existant.

```
public void DrawAllShapes(List<Shape> shapeList)
{
  foreach (Shape obj in shapeList)
  {
    obj.Draw();
  }
}
```

Le respect de ce principe permet donc de bénéficier des plus grands avantages de la programmation orientée objet, à savoir l'encapsulation, la réutilisation et la maintenabilité.

PRINCIPE DE SUBSTITUTION DE LISKOV

Selon Robert Martin, ce principe peut être défini de la manière suivante : les méthodes qui utilisent les objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir. En d'autres termes, une classe de base doit être substituable par ses classes dérivées sans que cela nécessite des modifications dans la méthode qui utilise ces classes. Ce principe est complémentaire au principe Ouvert-Fermé que nous venons d'expliquer.
Il existe deux types de violation du principe de Liskov, une plutôt conceptuelle et une plutôt fonctionnelle.
La première violation consiste à déterminer le type concret d'un objet pour faire le traitement adapté. Ceci est déterminé par l'utilisation de « cast » ou bien directement par vérification du type de l'objet. Cette violation est simple à détecter, car elle constitue également une violation du principe Ouvert-Fermé.
Si nous reprenons l'exemple utilisé dans le principe Ouvert-Fermé, une violation du principe de Liskov serait la suivante :

```
public void PrintShapeArea(Shape obj)
{
  double area = 0;
  if (obj is Square)
    area = GetSquareArea((Square)obj);
  else if (obj is Circle)
    area = GetCircleArea((Circle)obj);
  Console.WriteLine(string.Format(«Shape Area: {0}», area));
}

private double GetSquareArea(Square obj)
{
  return obj.SideLength * obj.SideLength;
}

private double GetCircleArea(Circle obj)
{
  return Math.PI * obj.Radius * obj.Radius;
}
```

Dans cet exemple, la méthode « PrintShapeArea(…) » nécessite la connaissance du type réel de chaque objet passé en paramètre

Fig.3

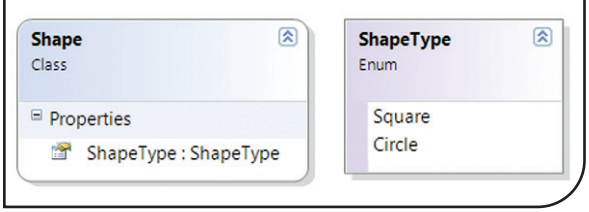


Fig.4

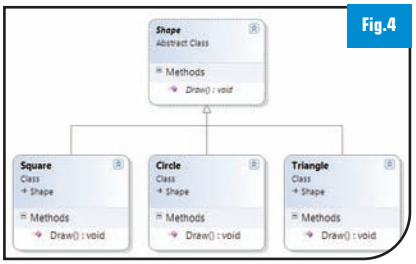
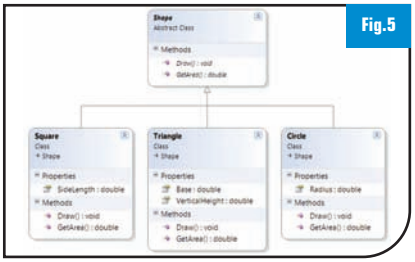


Fig.5



pour pouvoir appeler la fonction qui retourne la surface. Le traitement d'un nouveau type d'objet « Triangle » implique non seulement la modification de la méthode « PrintShapeArea(…) », mais aussi un renforcement du couplage entre la classe qui contient la méthode « PrintShapeArea(…) » et la classe « Triangle ». Pour éviter ce genre de violation, il faut donc minimiser l'utilisation des méthodes telles que « cast », « is », « as », « GetType », … qui permettent de travailler sur le type de l'objet.
Une bonne solution est d'appliquer un design qui respecte le principe Ouvert-Fermé, ce qui implique le rajout d'une méthode « GetArea() » à la classe de base « Shape » [Fig.5]. Tout nouveau type héritant de cette classe doit surcharger la nouvelle méthode avec le traitement adapté. La méthode « PrintShapeArea(…) » est maintenant en mesure de traiter tous les sous-types de manière transparente.

```
public void PrintShapeArea(Shape obj)
{
  Console.WriteLine(string.Format(«Shape Area: {0}», obj.GetArea()));
}
```

La deuxième violation de type fonctionnel est plus difficile à appréhender car elle n'est pas directement liée à une erreur de design, mais à une modification de comportement. Certains traitements nécessitent que les objets répondent à certaines spécifications et contraintes. Parfois, lorsque l'on dérive des objets qui ont des contraintes, il arrive qu'elles soient modifiées. Le résultat peut être différent de celui attendu, les objets se trouvant dans des états non-utilisables. À ce stade, on arrive à une difficulté de mise en œuvre des principes Ouvert-Fermé et de Liskov.
L'un pousse les développeurs à modifier les comportements dans les sous-types, alors que l'autre limite (voire interdit) ces mêmes modifications afin d'éviter les incohérences fonctionnelles. La solution, pour éviter les violations au principe de Liskov tout en respectant le principe Ouvert-Fermé, est l'utilisation du concept « Design By Contract » exposé par Bertrand Meyer. L'utilisation de ce concept dans les programmes consiste à établir des contraintes fonctionnelles dans l'utilisation des objets, ces contraintes définissant les conditions correctes d'utilisation d'un objet.
Il existe trois types de contrat :

- Les pré-conditions : tester l'état des objets avant le traitement (tester les paramètres)
- Les post-conditions : tester l'état des objets à la fin du traitement (tester les objets modifiés ou les résultats produits par le traitement)
- Les invariants : s'assurer que certains objets ne changent jamais pendant un traitement

Les modifications effectuées dans les sous-types sont donc encadrées par le biais de contrats « fonctionnels ». Cependant, il faut s'assurer que les fonctionnalités dans les sous-types respectent toujours les mêmes contraintes. Dans le cas où l'on souhaite utiliser le polymorphisme pour faire évoluer les fonctionnalités, les règles suivantes doivent être respectées :

- Aucune modification des invariants dans une classe dérivée
- Aucun renforcement des pré-conditions dans une classe dérivée
- Aucun affaiblissement des post-conditions dans une classe dérivée

Le principe de Liskov apparaît donc comme un complément du principe Ouvert-Fermé, il ajoute un aspect fonctionnel qui permet de s'assurer du bon fonctionnement des traitements.

PRINCIPE DE SÉGRÉGATION DES INTERFACES

Ce principe, également énoncé par Robert Martin, indique que les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas. Une entité informatique (classe, service,…) est cliente d'une interface si elle utilise une classe qui implémente cette interface. Appliquer ce principe est assez simple. La méthodologie employée est similaire à celle du principe de responsabilité unique. La création de classes qui possèdent plusieurs responsabilités est souvent nécessaire lorsqu'on développe des services. Dans ce cas, l'application du principe de ségrégation des interfaces permet de limiter le couplage entre les différentes classes.
Prenons le cas d'une classe « BusinessObject » qui implémente plusieurs groupes de fonctionnalités (plusieurs responsabilités) comme par exemple la création, la mise à jour, la validation, la suppression [Fig.6].

```
public class ValidationManager
{
  public void ValidateBusinessObjects(List<IBusinessObject>
businessObjects)
  {
    if (businessObjects != null)
    {
      foreach (IBusinessObject bo in businessObjects)
      {
        bo.Validate();
      }
    }
  }
}
public class PersistManager
{
  public void InsertAll(List<IBusinessObject> businessObjects)
  {
    if (businessObjects != null)
    {
      foreach (IBusinessObject bo in businessObjects)
      {
        bo.Insert();
      }
    }
  }
}
```

On note ici l'utilisation de l'interface « IBusinessObject » comme type à la place de la classe « BusinessObject ». On appelle cela « Interface-Based Programming ». Cette pratique permet de programmer en utilisant des interfaces à la place de classes concrètes. Tous les objets qui implémentent l'interface peuvent être utilisés au sein des méthodes. Il n'y a pas de limitation par un certain type de classe, mais l'exemple précédent présente encore des défauts de conception. Dans cet exemple, chacun des clients n'utilise qu'un aspect de l'interface :

- La classe « ValidationManager » n'utilise que les méthodes liées à la validation
- La classe « PersistManager » n'utilise que les méthodes liées à la persistance

Une modification des méthodes liées à la Validation implique une modification de toute l'interface « IBusinessObject » et donc indirectement de la classe « PersistManager » bien qu'elle n'utilise pas ces méthodes ! Il faut donc trouver une solution pour que les différents clients accèdent et dépendent seulement des méthodes qu'ils utilisent. La première modification à effectuer pour améliorer le design est le regroupement des méthodes communes dans des interfaces spécialisées (de la même manière que pour le principe de responsabilité unique). L'interface « IBusinessObject » est séparée en deux interfaces spécialisées : Persistance et Validation, et va hériter ces interfaces [Fig.7]. Il est maintenant possible d'utiliser aussi bien les fonctionnalités de Persistance que de Validation de façon séparée, ou bien d'utiliser les deux en même temps. Une modification dans une de ces interfaces n'a plus d'incidence dans l'autre. Elles évoluent de manière indépendante.

Bien évidemment, il faut maintenant s'assurer que ces interfaces spécialisées, « IPersistable » et « IValidatable », sont utilisées à la place de l'interface initiale « IBusinessObject ».

```
public class ValidationManager
{
  public void ValidateBusinessObjects(List<IValidatable> valida
tableObjects)
  {
    if (validatableObjects != null)
    {
      foreach (IValidatable vo in validatableObjects)
      {
        vo.Validate();
      }
    }

  }
}
public class PersistManager
{
  public void InsertAll(List<IPersistable> persistableObjects)
  {
    if (persistableObjects != null)
    {
      foreach (IPersistable po in persistableObjects)
      {
        po.Insert();
      }
    }
  }
}
```

Une interface globale peut conduire à un couplage non voulu entre plusieurs clients qui devraient être isolés. Le principe de ségrégation des interfaces indique donc que plusieurs interfaces clients spécifiques sont plus appropriées qu'une grande interface globale. Ce principe s'applique de la même manière aux classes abstraites, car elles peuvent être considérées comme des interfaces.

PRINCIPE D'INVERSION DE DÉPENDANCE

Dans de nombreuses applications, les modules de haut niveau sont directement liés aux modules de bas niveau. Les modules de haut niveau portent souvent la logique complexe, alors que les modules de bas niveau gèrent les opérations basiques et primaires. Ceci peut poser des problèmes de dépendance entre les modules.

Une application type qui présente des problèmes de dépendance forte peut être désignée ainsi : la Couche Présentation est dépendante de la Couche Métier qui est dépendante de la Couche Accès aux Données [Fig.8].

Le changement d'un module de bas niveau engendre la modification de ceux de haut niveau, les modules de hauts niveaux ne peuvent donc pas être réutilisés au sein d'autres contextes techniques.

Le principe d'inversion de dépendance, toujours énoncé par Robert Martin, indique que 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.

L'idée sous-jacente de ce concept est donc qu'une abstraction doit faire le lien entre les modules de haut et de bas niveau. Une abstraction est le plus souvent une interface, on peut également utiliser le concept « Factory Pattern ».

Ici la classe n'est dépendante d'aucune autre classe, les seules références à des objets sont typées via des interfaces :

```
public class ValidationManager
{
  private IValidatable _validatableObject;

  public ValidationManager(IValidatable validatableObject)
  {
  _validatableObject = validatableObject;
  }

  public void RunValidation()
  {
    _validatableObject.Validate();
  }
}
```

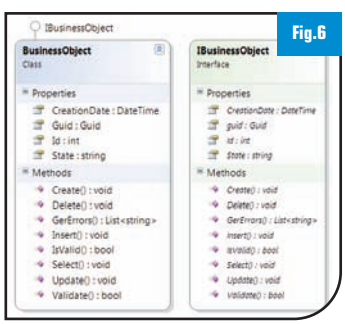


Fig.6

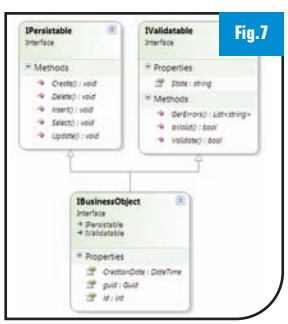


Fig.7

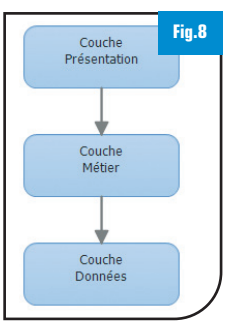


Fig.8

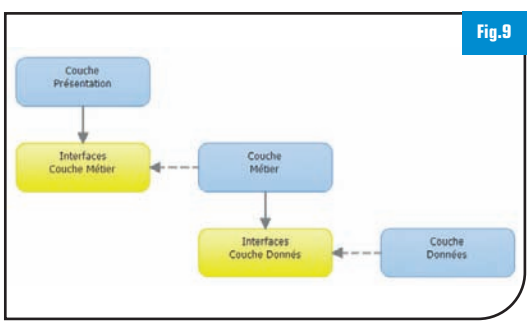


Fig.9

La classe « ValidationManagerFactory » se charge de résoudre les dépendances : elle associe un objet qui respecte l'interface « IValidatable » à un objet de type « ValidationManager ».

```
public class ValidationManagerFactory
{
  public ValidationManager GetValidationManager(IBusinessObject bo)
  {
    return new ValidationManager(bo);
  }
}
```

L'introduction d'interface permet donc d'inverser les dépendances et d'obtenir dans notre exemple des couches indépendantes [Fig.9].
Ce principe amène de la flexibilité dans les applications, les modules étant réutilisables. L'application de ce principe coupe donc totalement les liens pouvant exister entre deux modules mais complexifie le code en augmentant le niveau d'abstraction. Il faut donc savoir l'utiliser avec parcimonie.

CONCLUSION

S.O.L.I.D. est un groupe de principe qui va vous permettre d'aboutir à du code :

- compréhensible, car chaque classe sera spécialisée par responsabilité
- facilement maintenable, car le couplage entre les différentes entités est limité
- permettant l'intégration des évolutions de manière simple

Mais ces guides ne sont pas des règles figées dans leurs applications, elles doivent être mûrement réfléchies et adaptées à chaque contexte. L'intérêt de prendre en compte les principes S.O.L.I.D. dans les projets doit également être mis en relation avec la méthodologie de gestion de projet. Aujourd'hui les méthodes agiles comme SCRUM ou XP sont à la mode et remplacent de plus en plus les méthodologies classiques comme le modèle en cascade ou bien le modèle du cycle en V.
Les méthodes agiles reposent sur une multiplication de cycles ou d'itérations courtes dans le temps, elles favorisent la constante modification et la re-factorisation du design …ce qui est en fait très compatible avec S.O.L.I.D. !
Faire un projet en mode agile en respectant les principes S.O.L.I.D. évite l'apparition de problèmes liés au design. On se rend donc compte que ces principes de la programmation orientée objet sont plus que jamais d'actualité et aident à la réussite des projets Agile.

■ Jason **De Oliveira**, *Solutions Architect chez Winwise, 13 ans d'expérience dans le développement logiciel. Blog:* http://jasondeoliveira.com

■ Fathi **Bellahcene**, *Consultant/Formateur chez Winwise. 7 ans d'expérience dans le développement logiciel. Blog :* http://blogs.codes-sources.com/fathi