Le principe d'inversion de dépendance l'explication complète

Adrien CAUBEL

12 mars 2024

Table des matières

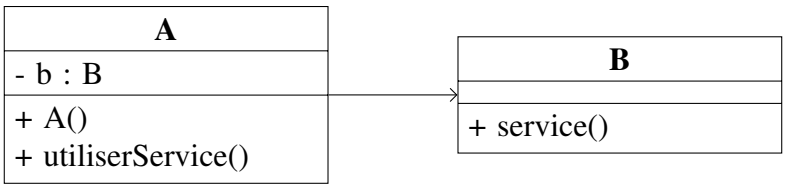
Introduction

Dans ce document nous revenons sur le principe d'inversion de dépendance qui est un fondamental à prendre en compte lors de la conception de votre application.

1 Réalisation d'une application

Notre étude de ce concept commencera par analyser une application basique afin d'évaluer ses faiblesses. Nous utiliserons un exemple abstrait, mais tout de même suffisant pour comprendre le problème.

1.1 Diagramme UML et implémentation




```
class A {
   private B b;

   public A(B b) { this.b = b; }

   public void utiliserService() {
      b.service();
   }
}
```

```
class B {
   public void service() {
      /* code */
   }
}
```

1.2 Le problème de cette conception

Cette conception présente un problème majeur.

— Si le composant B est modifié

— Alors nous sommes également obligés de compiler et déployer le composant A

— Or, nous ne souhaitons pas et n'avons pas à redéployer un composant en fonction de ses dépendances

Cela peut sembler anodin comme conséquence. Mais si vous travaillez sur un projet avec plusieurs milliers de classes avec un fort couplage entre elles. Lorsque vous modifiez une classe (`B`) alors toutes ses dépendances se voient recompilées (`A`, `C`, `D`, etc) alors que vous ne travaillez que sur un seul module (`A`).

2 Le principe d'inversion de dépendance

Pour respecter ce principe, deux assertions ont été définies

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

Avec ces deux affirmations, nous souhaitons que les interactions entre un module de haut niveau et un module de bas niveau passent forcément par une abstraction

2.1 Que signifie haut et bas niveau ?

— Les modules de haut niveau sont ceux qui ont des dépendances vers d'autres modules (de bas niveau)

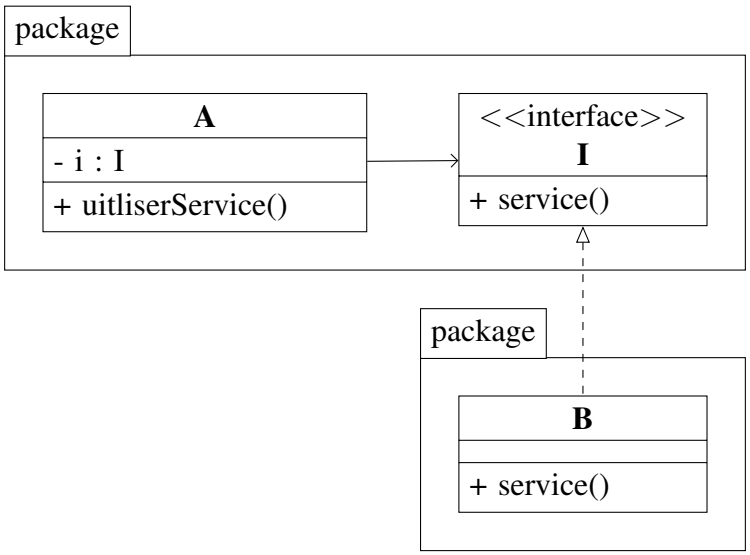
— Les modules de bas niveau sont les modules qui sont en dépendances

On peut voir ceci comme le module de haut niveau serait celui où l'action est invoquée et le module de bas niveau celui où l'action est exécutée.

2.2 Comment réaliser l'inversion de dépendance

Pour inverser les dépendances, nous devons utiliser des modules abstraits auxquels nos composants vont se référer. Les seuls moyens de référencer un autre composant sont les dépendances (solution inefficace), l'héritage et l'implémentation.

Dans notre cas, nous allons utiliser l'abstraction grâce aux interfaces.



```
class A {
   private I i;

   public A(I i) { this.i = i; }

   public void utiliserService() {
      i.service();
   }
}
```

```
class I {
   void service();
}

class B {
   public void service() {
      /* code */
   }
}
```

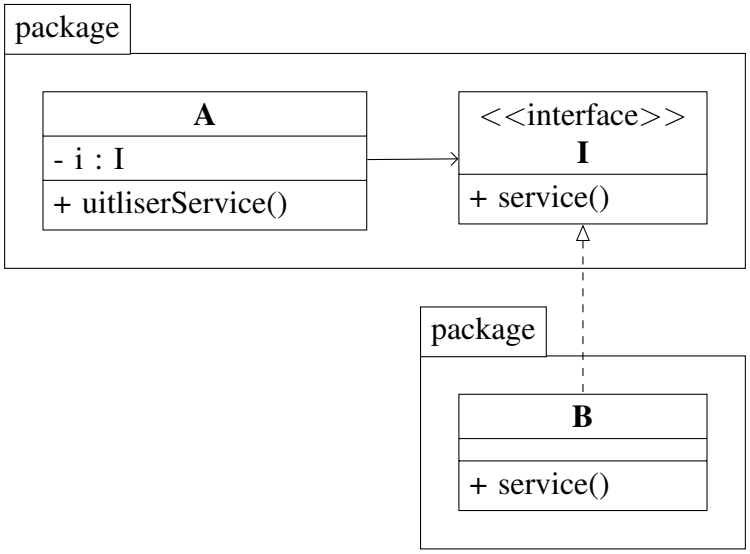
L'inversion de dépendance entre une classe `A` dépendant d'une classe `B` se réalise en ajoutant une interface `I` qui sera implémentée par `B`. Puis en réalisant une dépendance entre `A` et l'interface créée.

2.3 Les conséquences de l'inversion

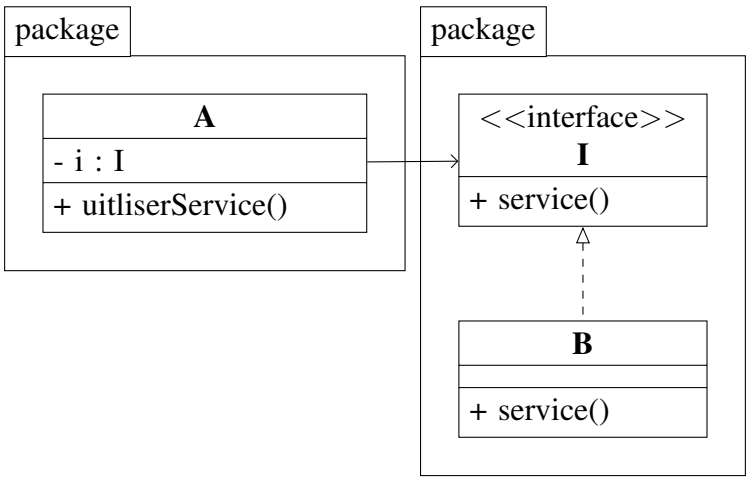
Nos composants peuvent maintenant être compilés et déployés indépendamment. En effet, si le corps des services de `B` évoluent nous n'aurons besoin de compiler et déployer uniquement cette classe. La seule obligation d'un déploiement complet est un changement dans l'interface. Mais, il faut considérer qu'une interface étant abstraite est moins volatile que ses implémentations concrètes.
Le composant contenant la logique métier n'est plus dépendant des autres composants annexes qui sont considérés comme des plug-ins.

3 Où positionner l'interface

Pour le moment nous avons toujours positionné l'interface dans le module de haut niveau.

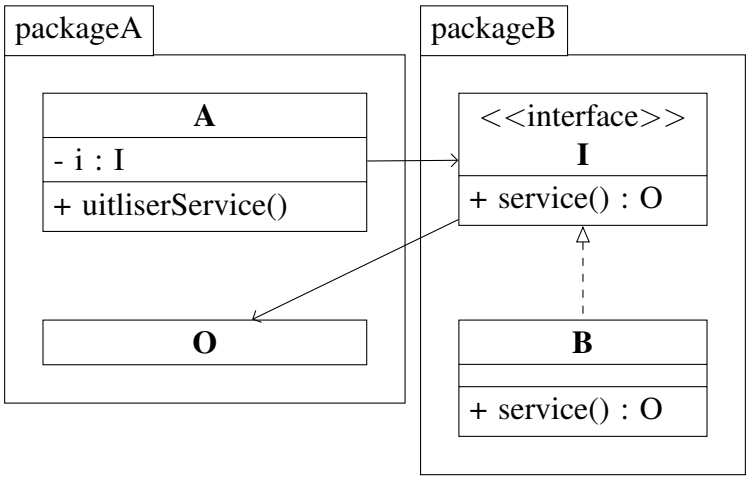


Mais qu'est-ce qui nous empêcherait de positionner l'interface dans le module de bas niveau ?



3.1 Relation cyclique

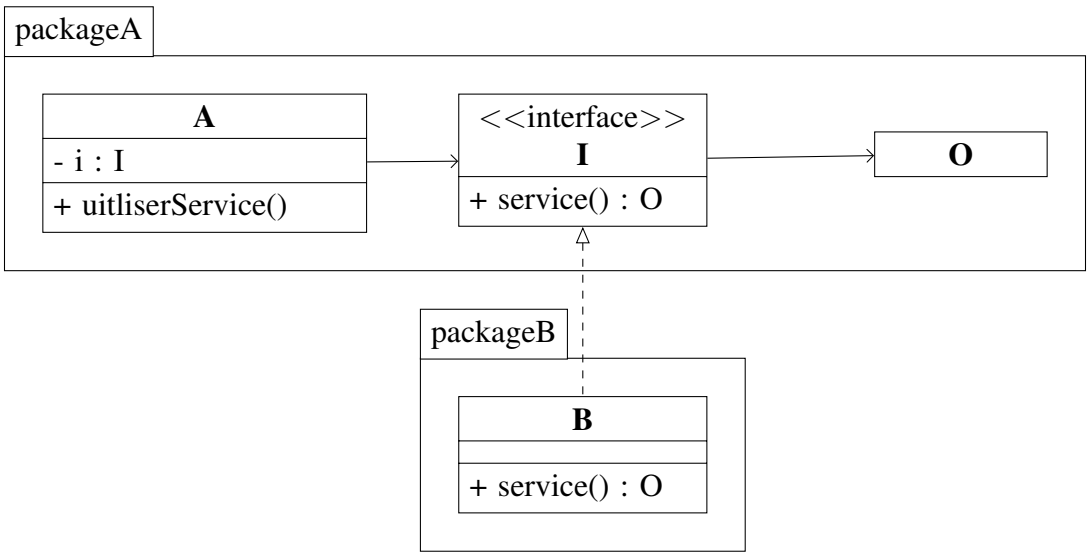
Supposons que la classe `B` retourne un objet `o` à la classe `A`. Nous avons donc l'organisation suivante (positionnement de l'interface dans le module de bas niveau).



Si nous prenons un peu de hauteur nous remarquons une relation cyclique entres nos deux packages. En effet

- le `packageA` dépend du `packageB` avec `A` utilise `I`.
- le `packageB` dépend du `packageA` car on retourne un objet de type `O` contenu dans `packageA`.

Pour éviter la relation cyclique, nous pouvons déplacer l'interface dans le `packageA`. Maintenant, nous avons seulement le `packageB` de bas niveau qui dépend du `packageA` de haut niveau.



3.2 Favoriser la stabilité

L'inversion de dépendance permet aux éléments les plus stables de ne pas dépendre pas d'éléments moins stables, qui sont plus sujet aux changement.
Dans l'exemple précédent, le `packageB` de bas niveau peut être facilement sujet aux changement. Or, nous ne souhaitons pas recompiler et redéployer le `packageA` régulièrement (i.g. à chaque changement).

Pour ce faire, nous devons déplacer l'interface `I` dans le module de haut niveau (i.g. `packageA`), par conséquent même si nous changeons d'implémentation de `B` en `B'` nous n'avons pas besoin de redéployer le `packageA` car il ne dépend pas de `packageB`.

3.3 Conclusion

C'est pour cette dernière raison principalement que nous placerons toujours l'interface dans le module le plus stable.

4 Quand utiliser l'inversion de dépendance

On utilise régulièrement l'inversion de dépendance, mais il faut faire attention à ne pas en faire un *non-sens*

4.1 Inversion dans des entités : non

On pourrait se demander si entre deux entités métiers nous avons besoin d'appliquer le principe d'inversion de dépendances.

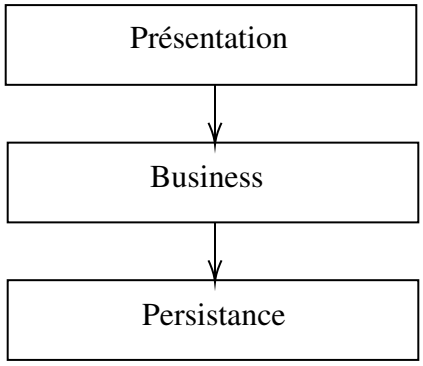
Si nous avons deux classes d'entités : `Dog` et `Food` où il y a une association entre `Dog` et `Food` (le chien peut manger de la nourriture), vous n'ajouterez pas d'interface entre les deux classes ; c'est un non-sens ; Parce que les deux classes se trouvent dans la même couche, la couche *Entité*.

4.2 Inversion entre couche : oui

L'inversion de dépendance est principalement utilisé pour découplé nos couches dans une *architecture par couches*.

Une architecture par couchons fonctionne comme suit :

1. La couche *Présentation* fait une requête à la couche *Business*
2. La couche *Business* est responsable de la logique métier de l'application (elle va faire des calculs qu'elle va retourner à la Vue). Et pour ce faire, elle a besoin de données qui sont gérées par la couche *Persistance*
3. La couche *Persistance* est responsable de l'accès aux données et de les renvoyer à la couche appelante



Entre chaque couche nous ajoutons une abstraction. Pour la couche persistance nous rajoutons dans le package *persistance* l'interface suivante.

```
// Persistance package
public interface IProductDAO {}

public class ProductDAO implements IProductDAO{
}
```

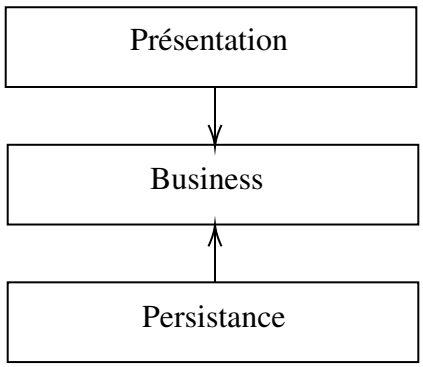
4.2.1 Où placer l'interface ?

Mais comme tout à l'heure nous devons nous demander où positionner l'interface.
Il est plus facile de modifier le type de persistance, c'est-à-dire la base de données ou la technologie permettant d'accéder à la même base de données, que la business logique. Par conséquent, c'est la couche *Business* qui est la plus stable. Nous devons donc positionner l'interface dans cette couche

```
// Business package
public interface IProductDAO {}

// Persistance package
public class ProductDAO implements IProductDAO{
}
```

Nous venons donc d'inverser la dépendance, c'est la couche *persistance* qui dépend maintenant de la couche *business*.



On remarque le même procédé pour la couche *présentation* qui dépend de la couche la plus stable *business*. Cela signifie que la couche *présentation* appelle la couche *business* au travers d'une interface.

```
// Présentation package
public class Controller {
  private IBusiness business; // utilisation de l'objet
}
```

```
// Business package
public interface IBusiness {}

public class Business implements IBusiness {
   private IProductDAO productDAO; // appel de la couche persistance
}
```

Nous venons donc de placer nos interfaces dans le module le plus stable *business*. Nous considérons *Business* comme devant être le plus stable car nous ne souhaitons pas que notre coeur applicatif soit impacté par des changements externes.

Business est un module de haut niveau :

- Il utilise la couche *persistance* et *présentation*
- mais pour faire en sorte qu'il ne dépende pas de *persistance* et de *présentation* nous avons inversé les dépendances via des interfaces placées dans *Business*