Le principe d'inversion de dépendance

Adrien CAUBEL

3 janvier 2022

Table des matières

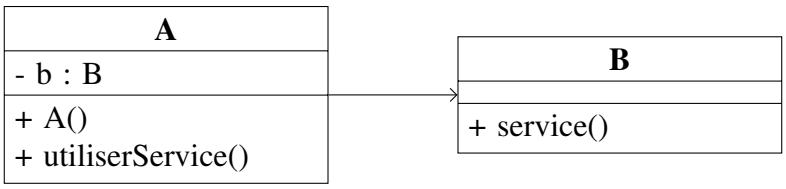
Introduction

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

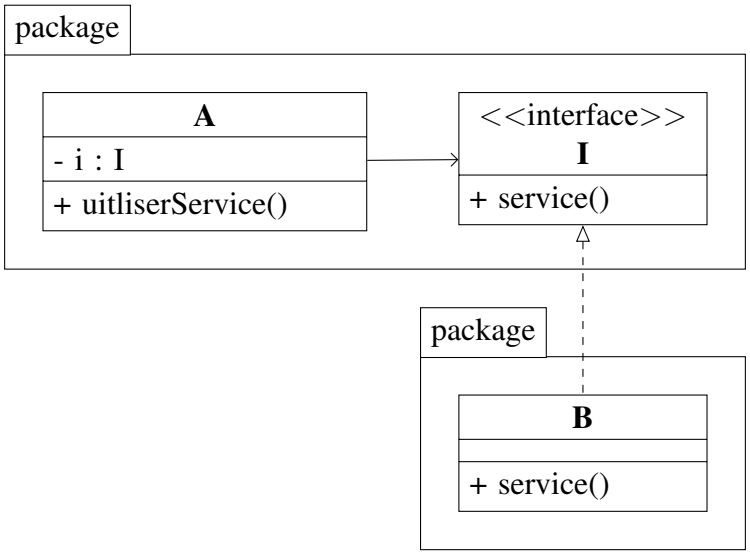
1. Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
2. Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

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 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.




```java
class A {
   private I i;

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

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

```java
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.

2.2 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.