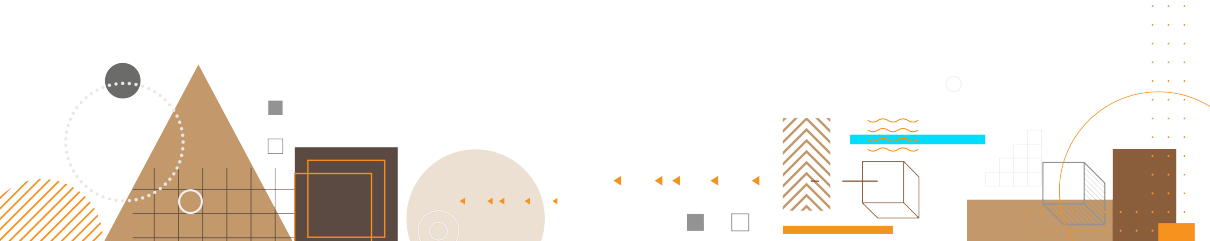
Une utilité des interfaces

Principes de bonne conception

Adrien CAUBEL

23 octobre 2022



Plan

Introduction

- Énumérer des problèmes et les résoudre
- Apporter un raisonnement *Pourquoi est-ce un problème ?* et *Comment le résoudre ?*
- Comprendre pourquoi des principes SOLID existent

Pourquoi modifier une classe ?

```
public class Employee {
    saveToDB() { }
    calculatePay() { }
    showOnWeb() { }
}
```

Impacts lorsqu'on modifie une classe

- Tous les acteurs sont impactés
- Modification pour la gestion des salaires :
  - L'administrateur de la base de données et l'intégrateur web ne sont pas intéressés par cette modification mais devront quand même faire une montée en version sur la nouvelle version de la classe `Employee`.
  - Introduction de bugs possiblent

Deux problèmes

- Compilation qui entraine une nouvelle livraison
- Nombre de responsabilités trop élevé

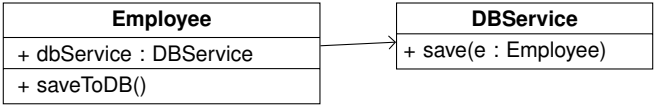
Réduire le nombre d'acteurs en déléguant les responsabilités

```
public class Employee {
    DBService dBService;
    saveToDb() {
        dBService.save(this)
    }
}
```

Pourquoi avons nous diminuer la responsabilité ?

Si l'administrateur de la base de données souhaite changer le mode de connexion à cette dernière, alors nous irons modifier le module `DBService` au lieu de `classe` Employee initialement

Il reste le problème de compilation



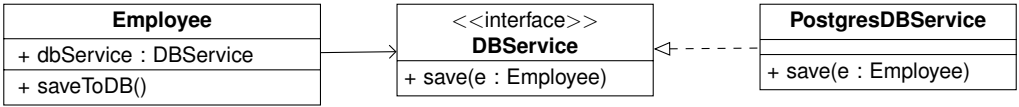
- Si `DBService` est recompilé
- Alors `Employee` devra être recompilé
- Donc tous les acteurs devront faire une montée en version

La solution

Les interfaces

Utiliser les interfaces pour inverser la dépendance

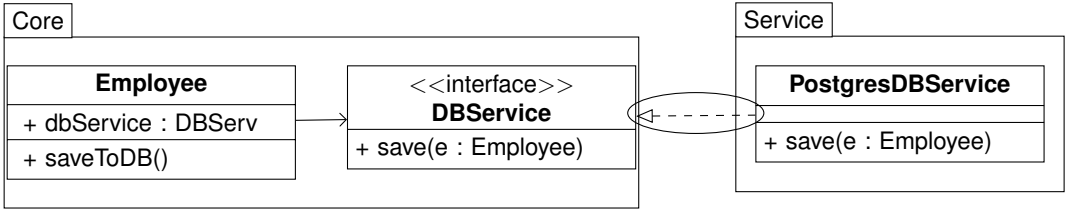
- Introduire une interface `DBService`.
- Créer l'implémentation `PostgresDBService`.
- Faire dépendre `Employee` de `DBService`.



Pourquoi Employee ne sera pas recompiler

- `Employee` dépend de `DBService`
- Les modifications ne sont pas sur `DBService` mais sur les implémentations
- Donc `Employee` n'a pas besoin d'être recompilé

Organiser notre application



Comment injecter la dépendance

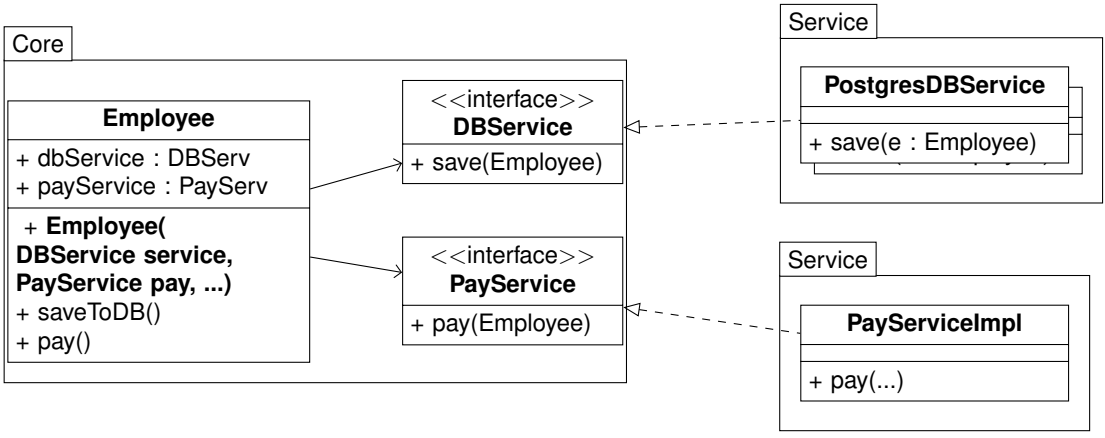
- Fournir une implémentation de `DBService`
- Passer l'implémentation en paramètre.

```
/* Constructeur de la classe Employee */
public Employee(DBService service) {
   dbService = service;
}
```

```
public static void main(String args[]) {
   DBService service = new PostgresDBService();

   /* On passe l'instance de DBService en paramètre
      du constructeur */
   Employee employee = new Employee(service);
}
```

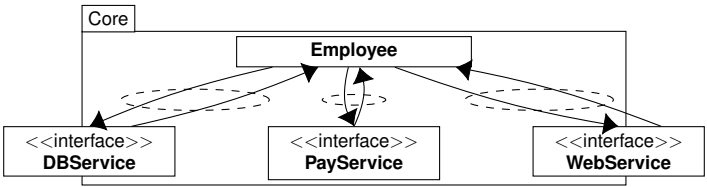
Comment injecter la dépendance



Une relation cyclique

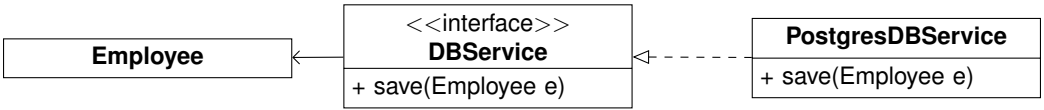
- La classe `Employee` à une dépendance vers l'interface `DBService`
- La méthode `saveToDB` de `DBService` référence `Employee`

```
public void saveToDB() {
    dbService.save(this) /* this référence Employee */
}
```



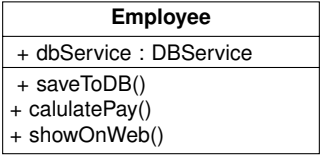
Qui est en tort ?

- `Employee` doit-il être au courant des services qui l'utilise ?



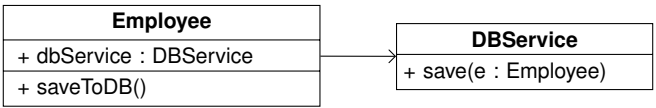
Conclusion

- Un problème de responsabilités trop nombreuses pour une seule classe
- La compilation entraine une montée en version



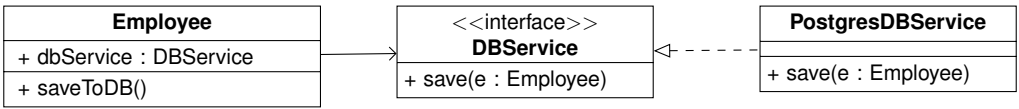
Conclusion

- Un problème de responsabilités trop nombreuses pour une seule classe
  - Créer des classes qui deviennent responsables des actions
- La compilation entraine une montée en version



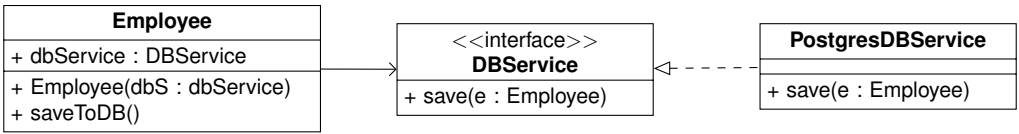
Conclusion

- Un problème de responsabilités trop nombreuses pour une seule classe
  - Créer des classes qui deviennent responsables des actions
- La compilation entraine une montée en version
  - Utiliser des interfaces



Conclusion

- Un problème de responsabilités trop nombreuses pour une seule classe
  - Créer des classes qui deviennent responsables des actions
- La compilation entraine une montée en version
  - Utiliser des interfaces
    - Inclure les implémentations via l'injection de dépendance



Conclusion

- Un problème de responsabilités trop nombreuses pour une seule classe
  - Créer des classes qui deviennent responsables des actions
- La compilation entraine une montée en version
  - Utiliser des interfaces
    - Inclure les implémentations via l'injection de dépendance