Principes SOLID

Ingéniérie des systèmes d'information

Paul Brunet

Introduction

- **Objectif :** produire des conceptions extensibles, maintenables et réutilisables
- **Problème :** la conception relève fortement de l'artisanat, cf. software craftsmanship
- **Solution :** il faut s'aider du savoir-faire.

> *« Le meilleur outil de conception pour le développement de logiciels est un esprit bien éduqué sur les principes de conception. Ce n'est pas UML ou toute autre technologie. »*
>
> Craig Larman

Le savoir-faire est formalisé sous forme de :

- **Principes de conception :** notions importantes desquelles dépend la qualité d'une conception
- **Règles de conception :** ensemble de prescriptions de conception à respecter
- **Patrons de conception :** modèles de solutions à des problèmes récurrents

Principes SOLID
Plan

1. Principes de conception

2. Règles de conception

3. Conclusion

Principes de conception

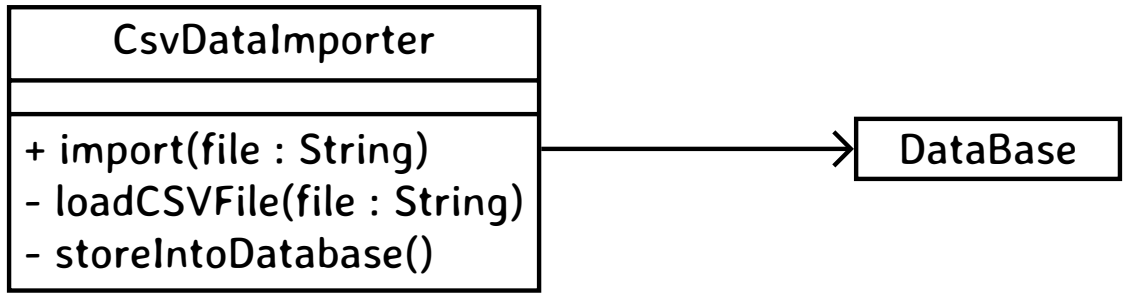
Ils sont connus sous l'acronyme SOLID :

- **S**ingle Responsibility Principle
  Principe de responsabilité unique
- **O**pen-Closed Principle
  Principe d'ouverture / fermeture
- **L**iskov Substitution Principle
  Principe de substitution de Liskov
- **I**nterface Segregation Principle
  Principe de ségrégation des interfaces
- **D**ependency Inversion Principle
  Principes d'inversion des dépendances

Responsabilité unique
principe S

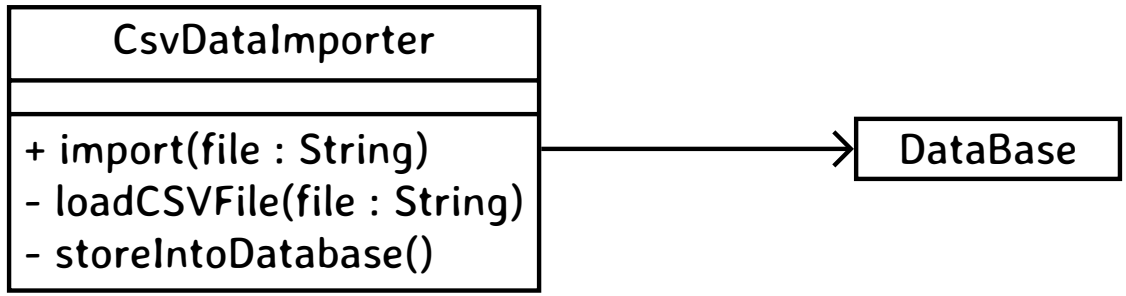
- Un module (fonction, classe, paquet, etc.) devrait n'avoir qu'une responsabilité unique.
- La responsabilité unique peut s'entendre comme une seule raison de changer
- Le but est évidemment d'augmenter la cohésion

Responsabilité unique
Exemple : Importer un CSV vers une BDD



Quel est le problème avec cette conception ?

Responsabilité unique
Exemple : Importer un CSV vers une BDD

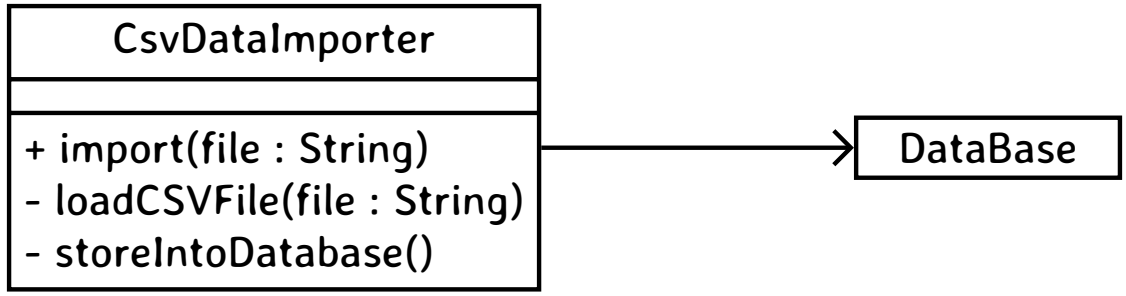


Quel est le problème avec cette conception ?

Il y a 2 responsabilités donc 2 raisons de changer

1) Le code pour lire le fichier CSV sous forme d'enregistrements
2) Le code pour stocker les enregistrements dans la base de données

Responsabilité unique
Exemple : Importer un CSV vers une BDD (refactoring)

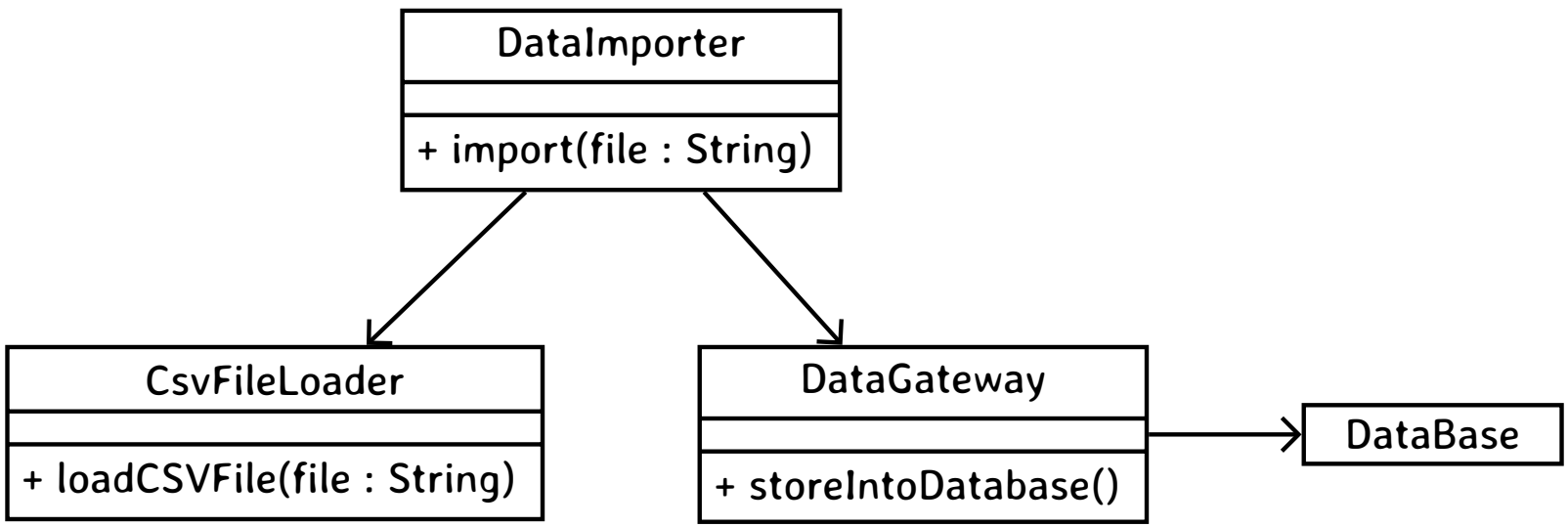


Comment augmenter la cohésion ?

Responsabilité unique
Exemple : Importer un CSV vers une BDD (refactoring)

Comment augmenter la cohésion ?

> Externaliser et séparer le code du chargeur de fichier et celui de la passerelle de stockage.

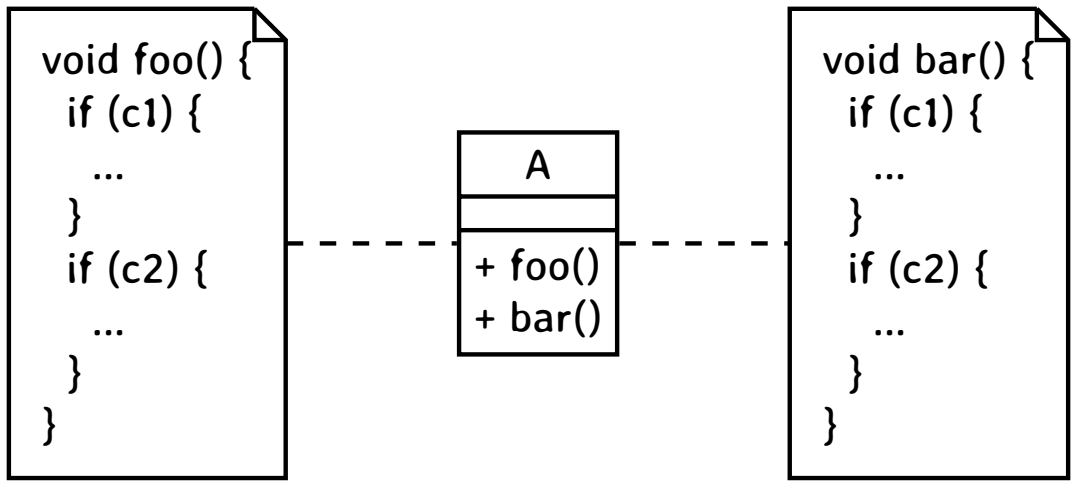


Ouverture/Fermeture
principe O

- Un module doit être ouvert aux extensions, mais fermé aux modifications.
- Nous devrions pouvoir ajouter une nouvelle fonctionnalité en créant du nouveau code et non en éditant du code existant.

Ouverture/Fermeture
Exemple

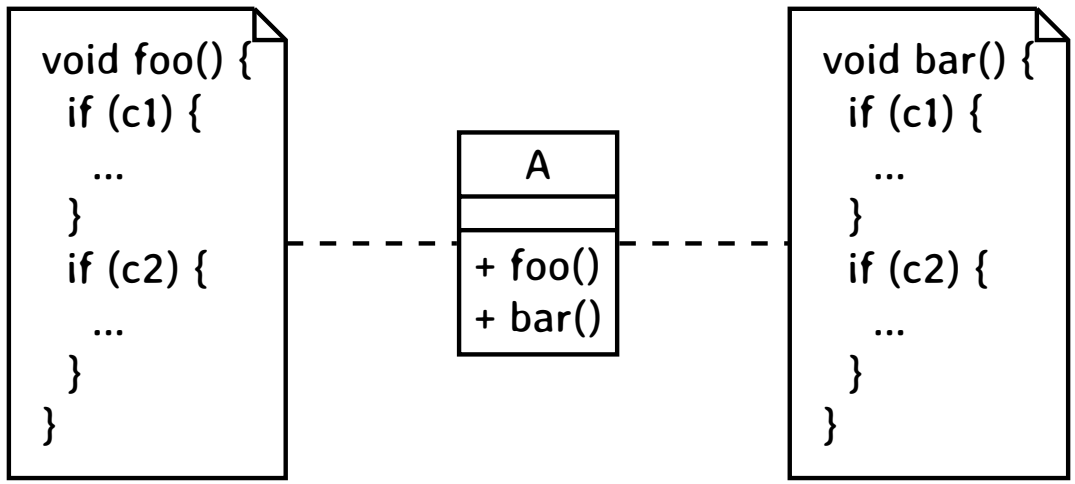
Considérons la classe A avec deux méthodes, qui ont chacunes deux comportements possibles, en fonction de conditions c1 et c2.



Quel est le problème avec cette conception ?

Ouverture/Fermeture
Exemple

Considérons la classe A avec deux méthodes, qui ont chacunes deux comportements possibles, en fonction de conditions c1 et c2.



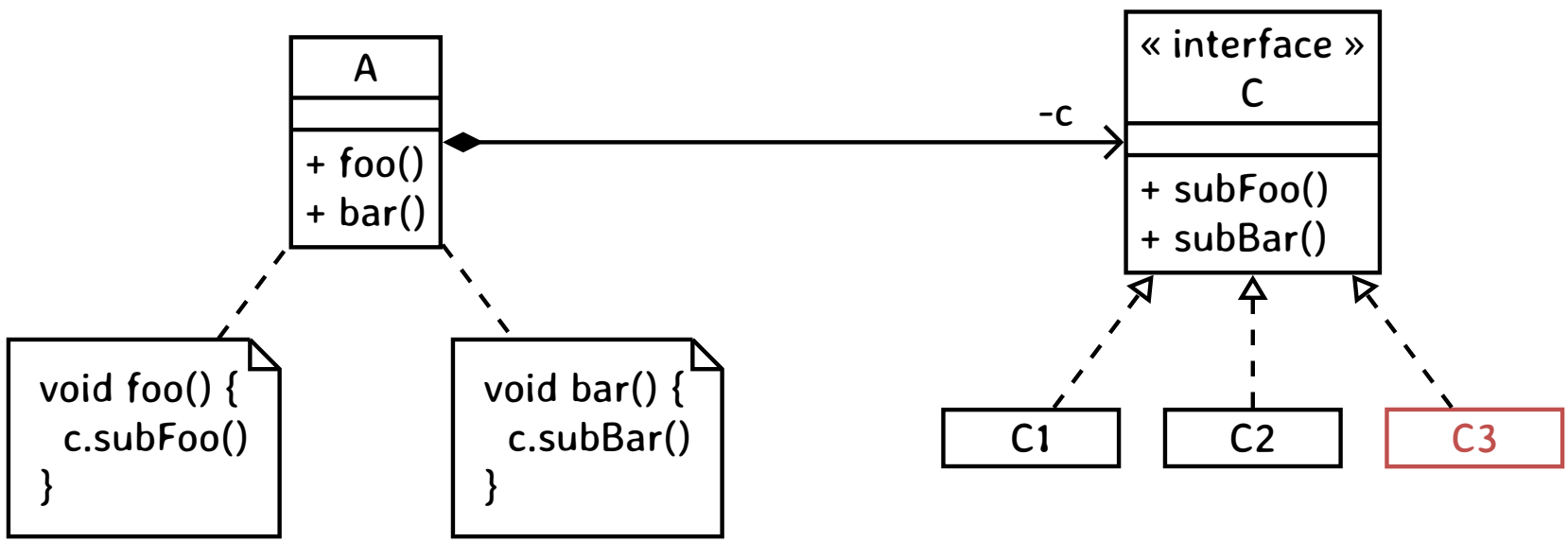
Quel est le problème avec cette conception ?

Si on veut ajouter une troisième variation, avec une condition c3, il faut éditer le code.

Ouverture/Fermeture
Exemple (refactoring)

Comment la rendre ouverte aux extensions et fermée aux modifications ?

Composition, héritage et polymorphisme

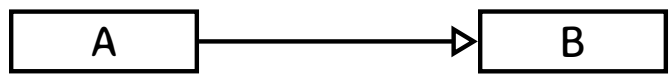


Ouverture/Fermeture
Corollaire : le principe de choix unique

- Aucun programme ne peut être ouvert à 100%.
  Par exemple, dans l'exemple précédent, il y a quelque part quelqu'un qui doit choisir entre les classes C1, C2 ou C3.
- Dans ce cas, une seule méthode ou une seule classe dans le système doit connaître l'ensemble des alternatives.
  cf. les patrons de conception Fabrique et Fabrique abstraite.

Substitution de Liskov
principe L

- Les objets de classes dérivées doivent être substituables aux objets de la classe de base.
- Ce principe pourrait être utilisé comme définition de la relation de généralisation.

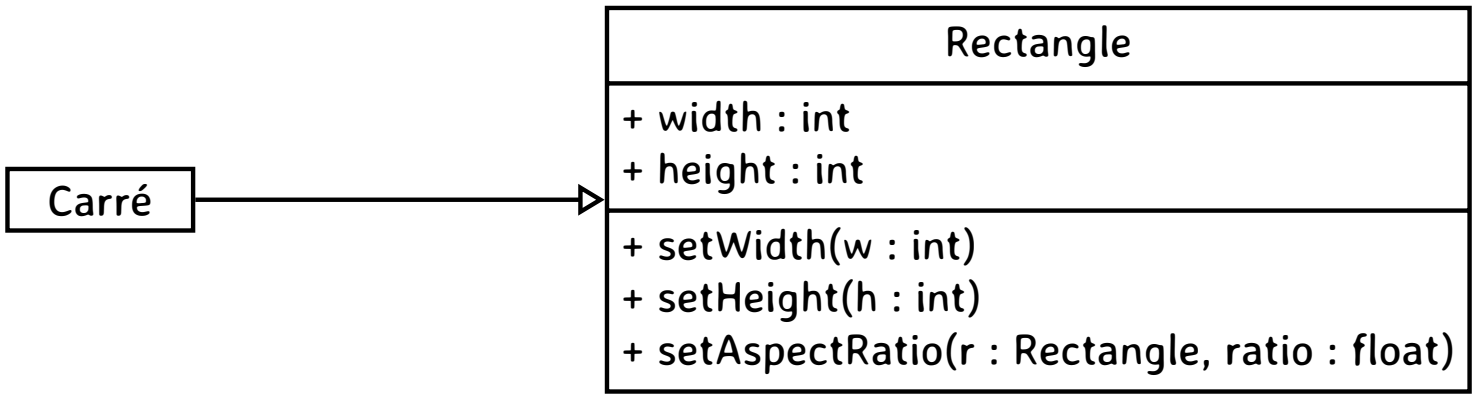


Péril d'une hiérarchie non substituable

Fragilité : le code créé pour une classe devient inapproprié pour une de ses sous-classes quelque part dans le logiciel. Les effets ne se révèlent qu'à l'exécution.

Substitution de Liskov
Exemple

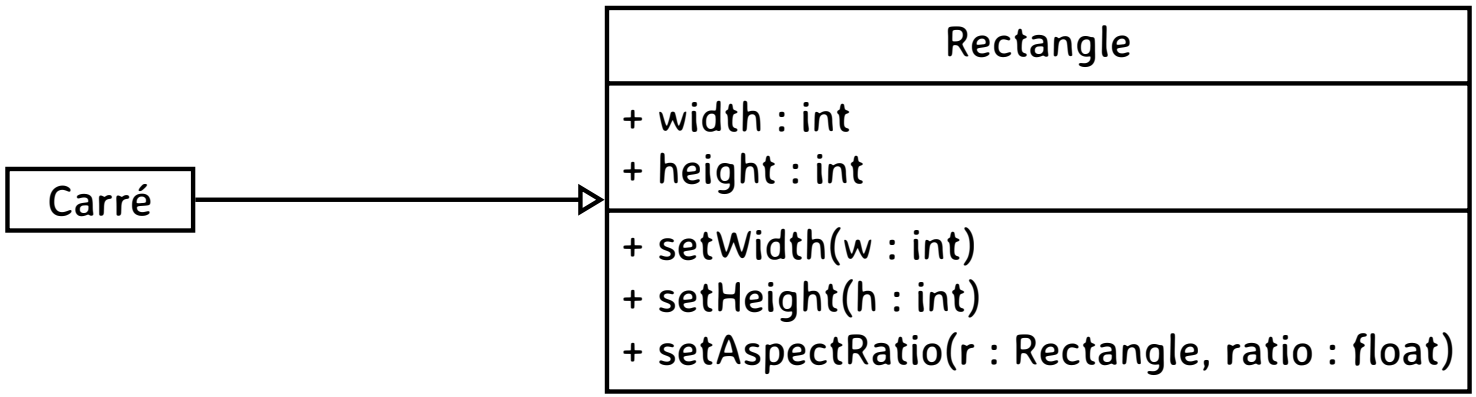
Soit la modélisation suivante : un carré est un rectangle particulier.



Quel est le problème avec cette conception ?

Substitution de Liskov
Exemple

Soit la modélisation suivante : un carré est un rectangle particulier.



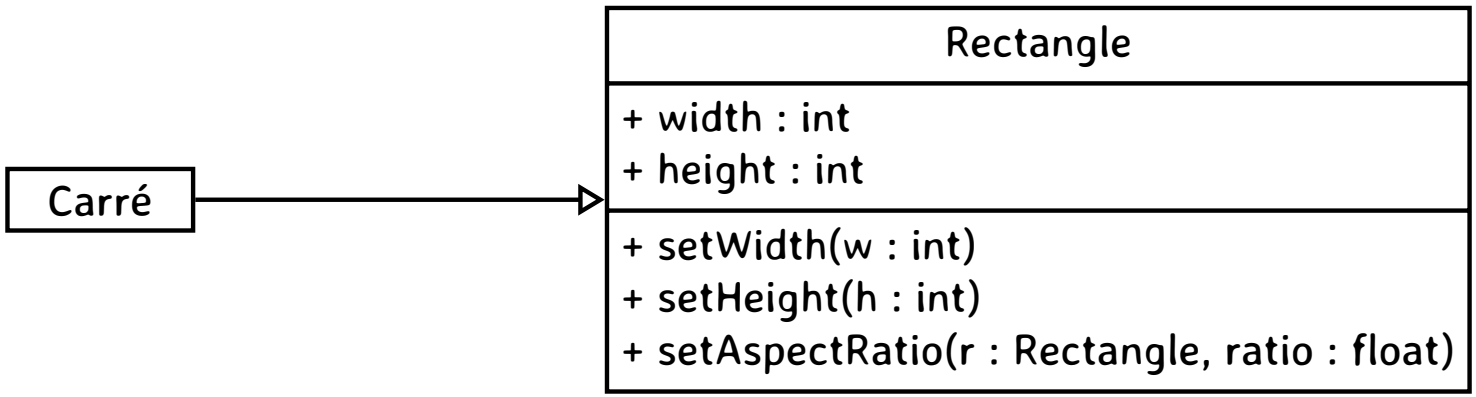
Quel est le problème avec cette conception ?

Le carré ne respecte pas tout le contrat de Rectangle

- ☞ La méthode setAspectRatio() (4/3, 16/9, A4, etc) n'a certainement pas le comportement attendu avec un carré
- ☞ Après setWidth() on s'attend à ce que la largeur ait la nouvelle valeur et la hauteur conserve son ancienne valeur. Ce n'est pas le cas pour le carré

Substitution de Liskov
Exemple

Soit la modélisation suivante : un carré est un rectangle particulier.



Quel est le problème avec cette conception ?

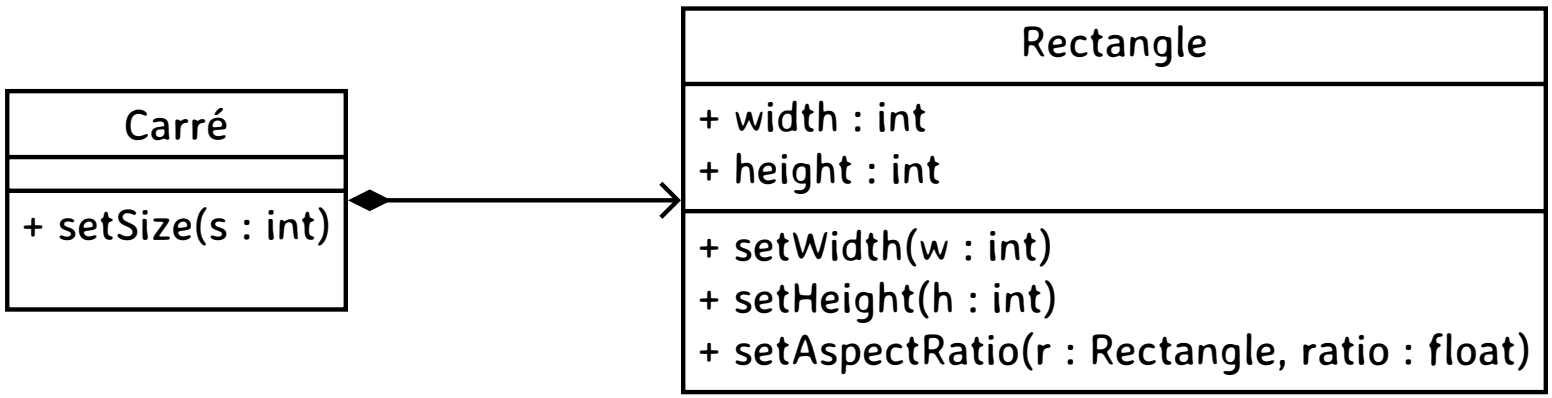
Le carré ne respecte pas tout le contrat de Rectangle

- ☞ La méthode setAspectRatio() (4/3, 16/9, A4, etc) n'a certainement pas le comportement attendu avec un carré
- ☞ Après setWidth() on s'attend à ce que la largeur ait la nouvelle valeur et la hauteur conserve son ancienne valeur. Ce n'est pas le cas pour le carré

Comment modifier la conception pour éviter le problème de substitution mais sans dupliquer le code commun ?

Substitution de Liskov
Exemple (refactoring)

Comment modifier la conception pour éviter le problème de substitution mais sans dupliquer le code commun ?



☞ Solution
- Le carré n'hérite plus de rectangle
- Le carré utilise le rectangle par composition

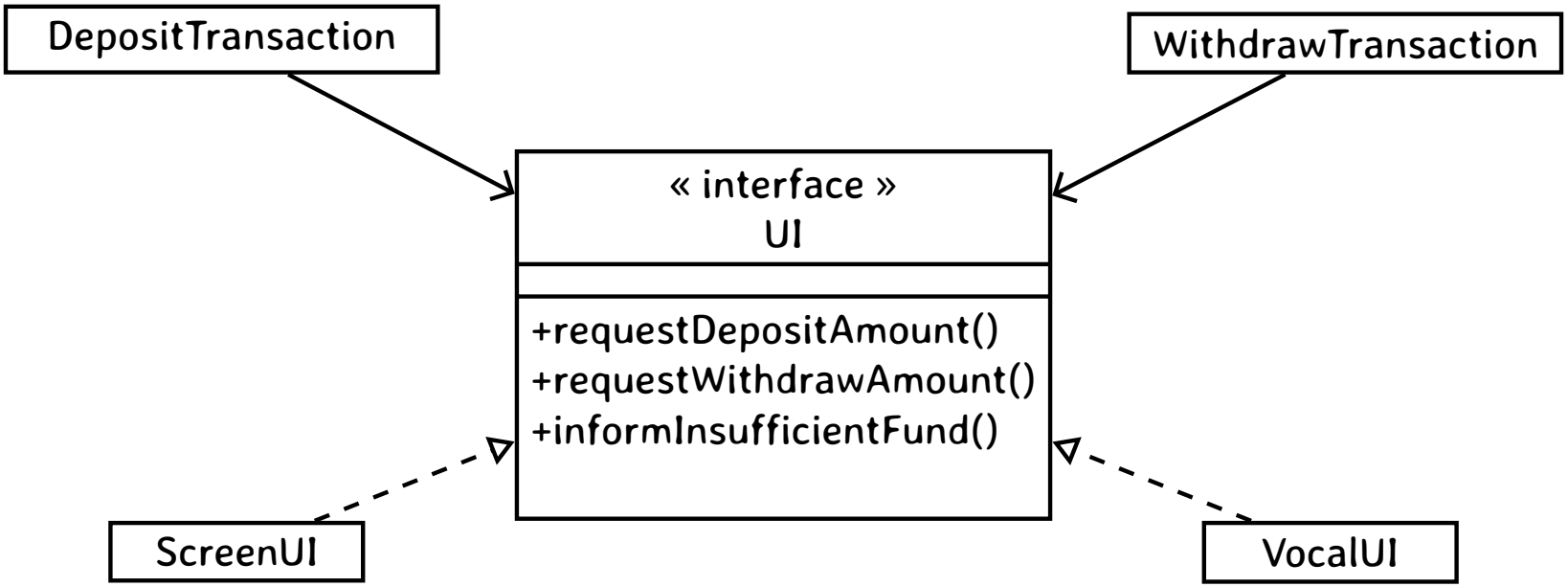
☞ Cette fois le carré n'est pas substituable au rectangle

Ségrégation d'interface
principe I

- La dépendance d'une classe à une autre devrait être restreinte à l'interface la plus petite possible
- Le client d'une classe ne doit pas être forcé de dépendre de méthodes qu'il n'utilise pas

Ségrégation d'interface
Exemple d'un GAB

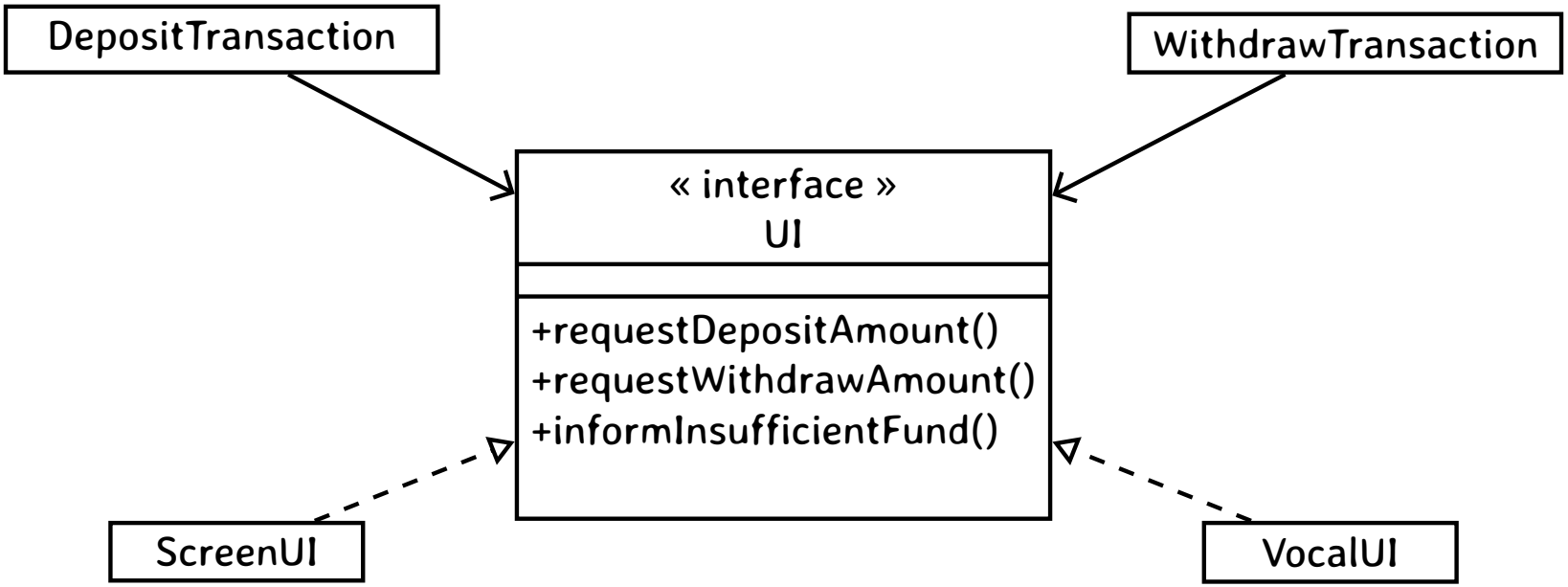
Toutes les transactions interagissent avec une même interface leur permettant de profiter d'une interface écran ou d'une interface vocale.



Quel est le problème avec cette conception ?

Ségrégation d'interface
Exemple d'un GAB

Toutes les transactions interagissent avec une même interface leur permettant de profiter d'une interface écran ou d'une interface vocale.

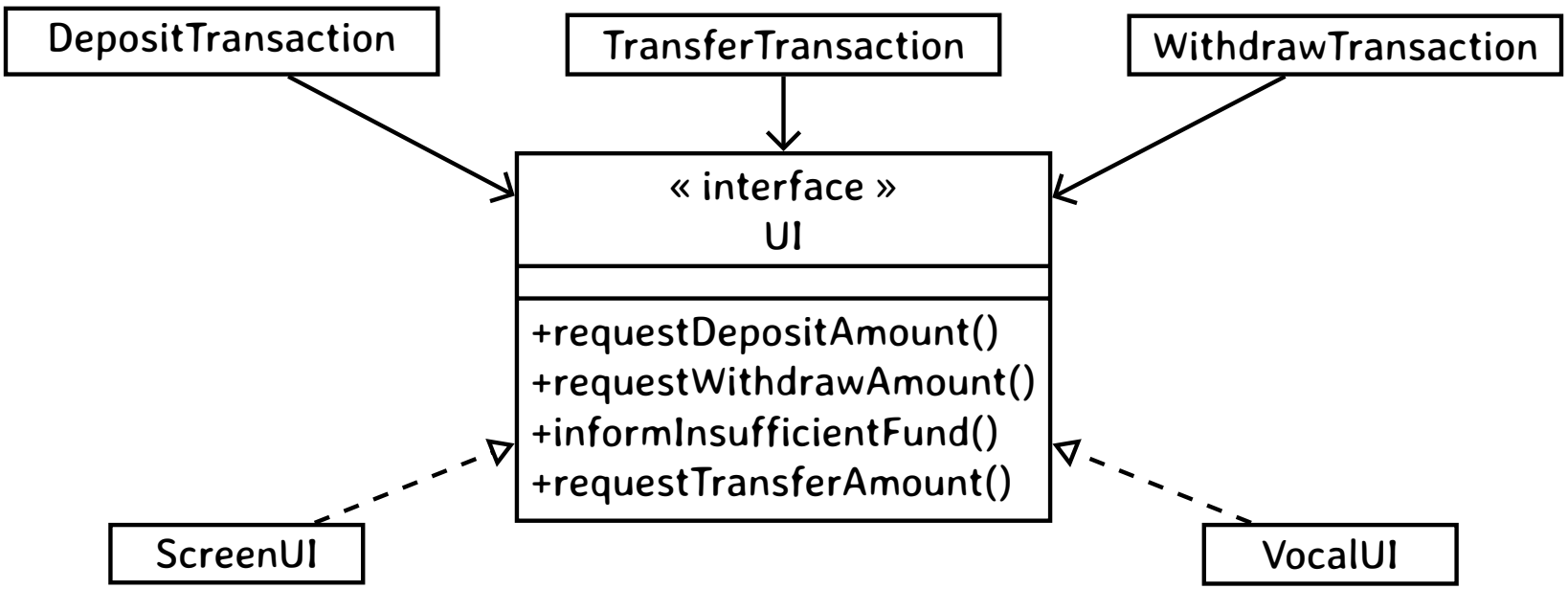


Quel est le problème avec cette conception ?

La modification d'une méthode impacte toutes les classes dépendantes même si elles n'utilisent pas la méthode.

Ségrégation d'interface
Exemple d'un GAB

Toutes les transactions interagissent avec une même interface leur permettant de profiter d'une interface écran ou d'une interface vocale.



Quel est le problème avec cette conception ?

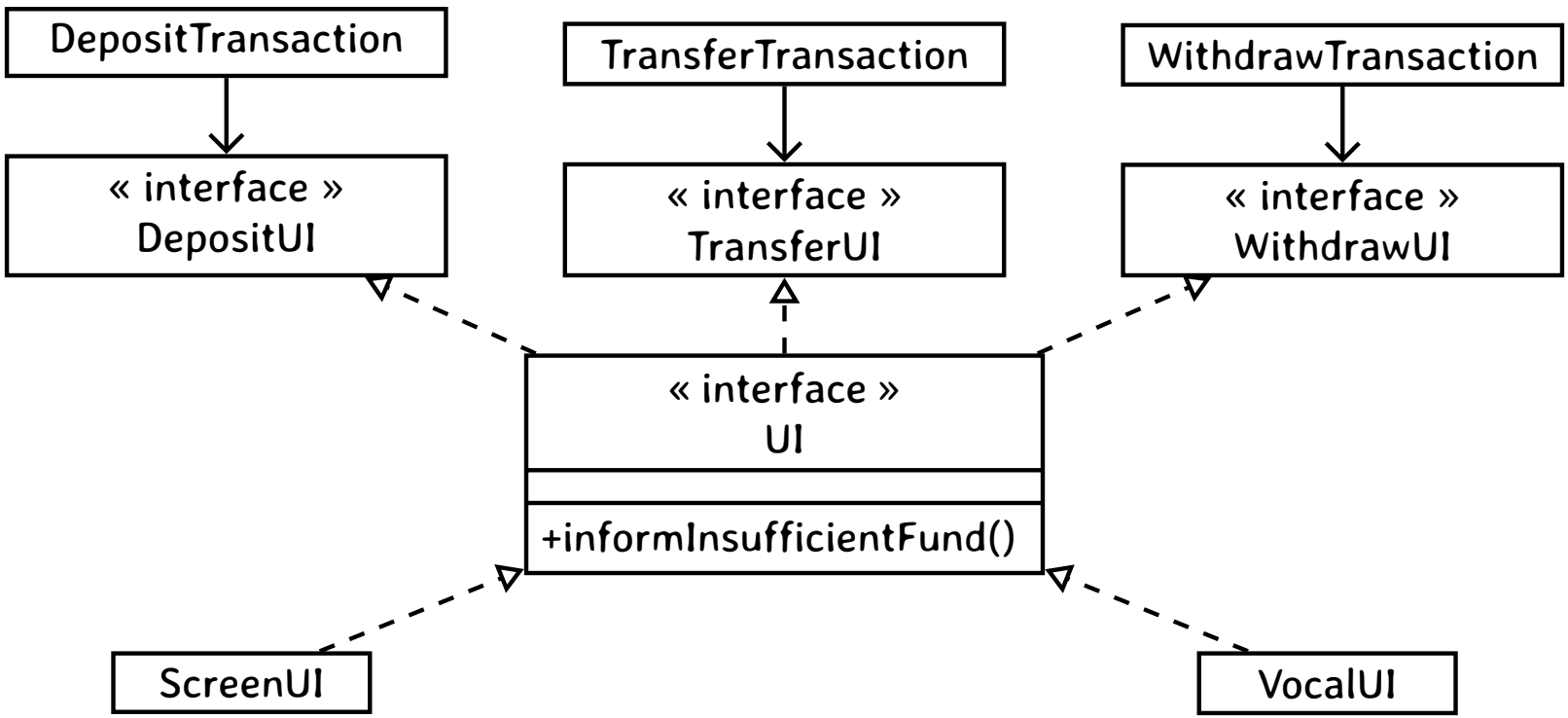
La modification d'une méthode impacte toutes les classes dépendantes même si elles n'utilisent pas la méthode.

Ségrégation d'interface
Exemple d'un GAB

Comment restructurer la conception pour n'avoir que des interfaces cohérentes ?

Ségrégation par héritage d'interfaces.

Chaque client n'est lié qu'à une interface minimale sous-ensemble de l'interface intégrale construite par héritage.



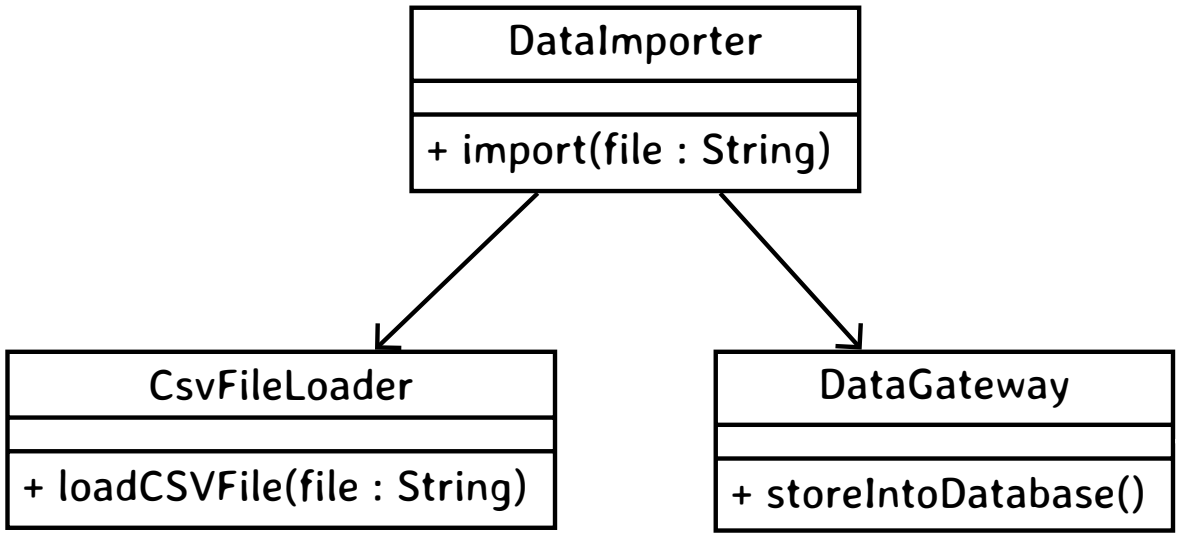
Inversion des dépendances
principe D

La relation de dépendance conventionnelle que les modules de haut niveau (aspect métier) ont par rapport aux modules de bas niveau (aspect implémentation), est inversée dans le but de rendre les premiers indépendants des seconds

- ☞ Les abstractions ne doivent pas dépendre de détails
- ☞ Les détails doivent dépendre des abstractions

Inversion des dépendances
Exemple : Importer un CSV vers une BDD

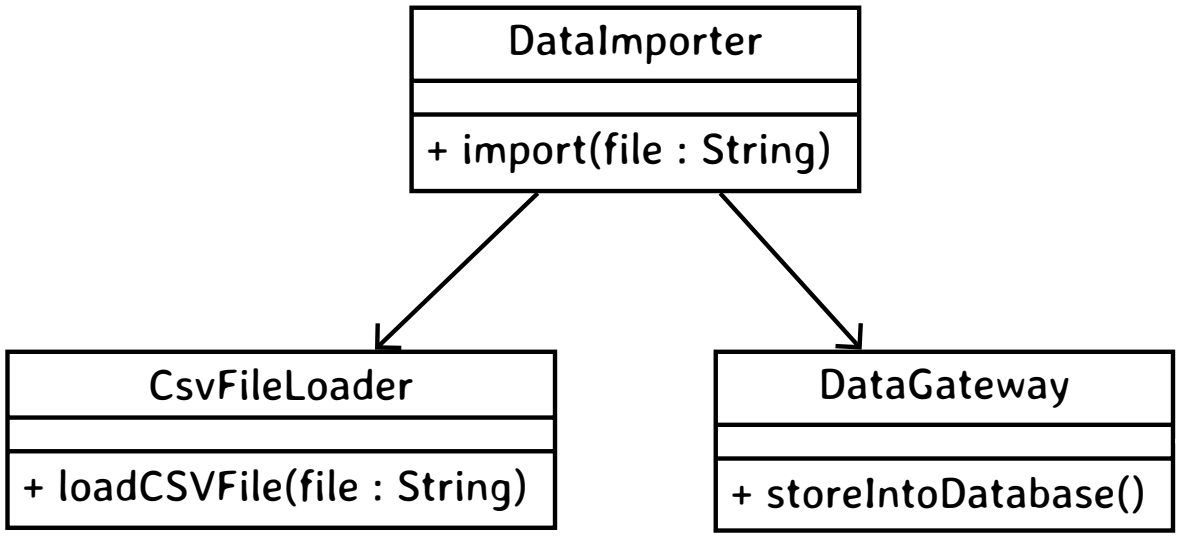
La classe `DataImporter` est dépendante du chargeur de fichier et de la passerelle de stockage.



Quel est le problème avec cette conception ?

Inversion des dépendances
Exemple : Importer un CSV vers une BDD

La classe `DataImporter` est dépendante du chargeur de fichier et de la passerelle de stockage.



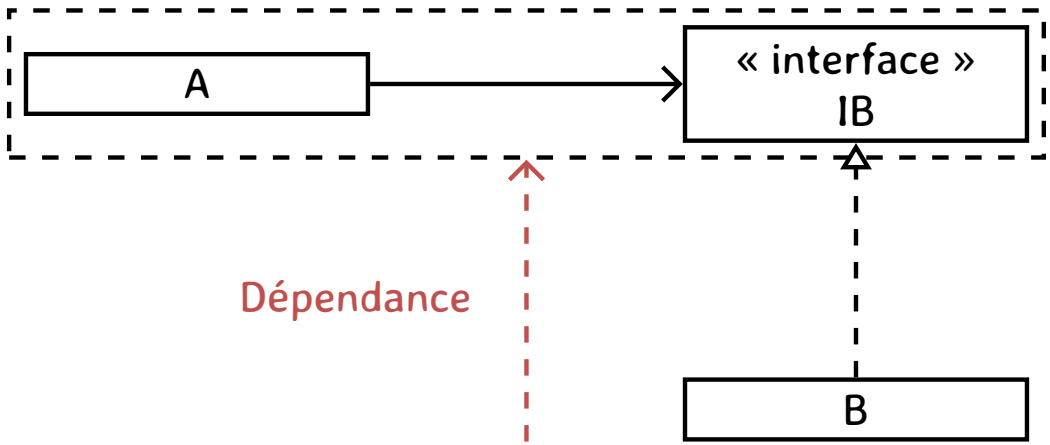
Quel est le problème avec cette conception ?

On ne peut pas réutiliser la classe d'importation sans réutiliser le chargeur de fichier CVS et la passerelle de stockage des données.

Inversion des dépendances
Abstraction

☞ Relation conventionnelle :

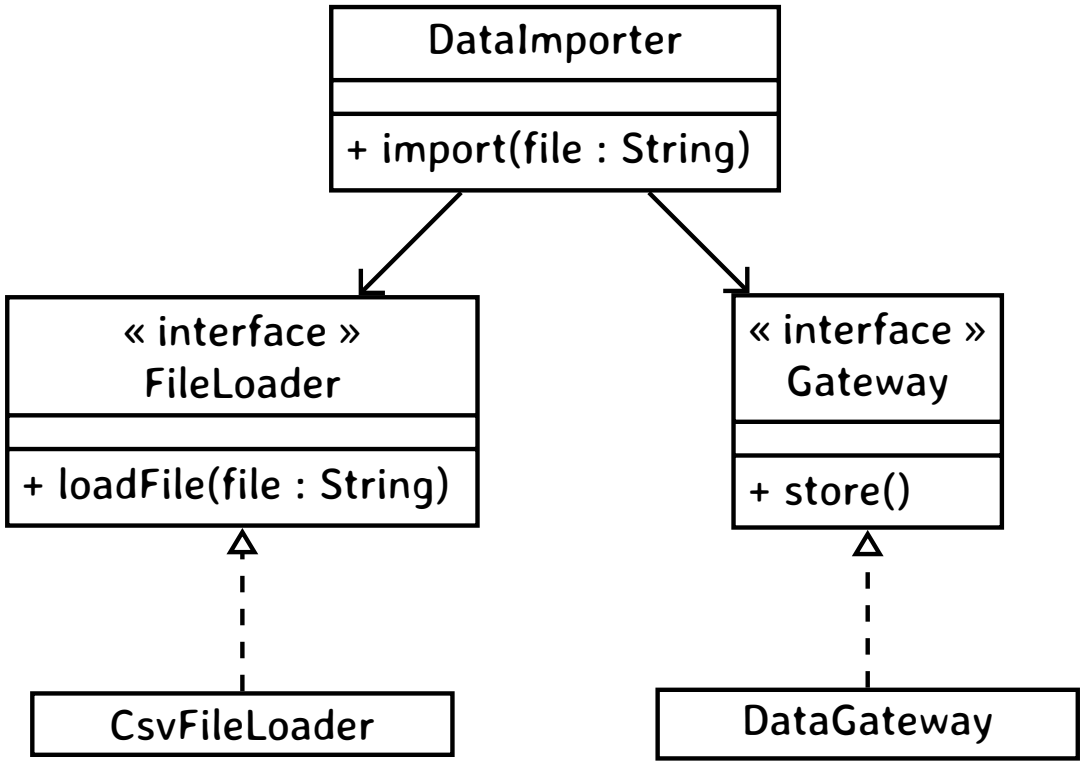
☞ Relation inversée :



Inversion des dépendances
Exemple : Importer un CSV vers une BDD (refactoring)

Comment rendre DataImporter indépendant des implémentations du chargeur du fichier CSV et de la passerelle ?

Inverser les dépendances avec des interfaces



Principes SOLID
Plan

1. Principes de conception

2. Règles de conception

3. Conclusion

Règles de conception

4 règles

- ☞ **Règle 1.** Réduire l'accessibilité des membres de classe
- ☞ **Règle 2.** Encapsuler ce qui varie
- ☞ **Règle 3.** Programmer pour une interface, non pour une implémentation
- ☞ **Règle 4.** Privilégier la composition à l'héritage

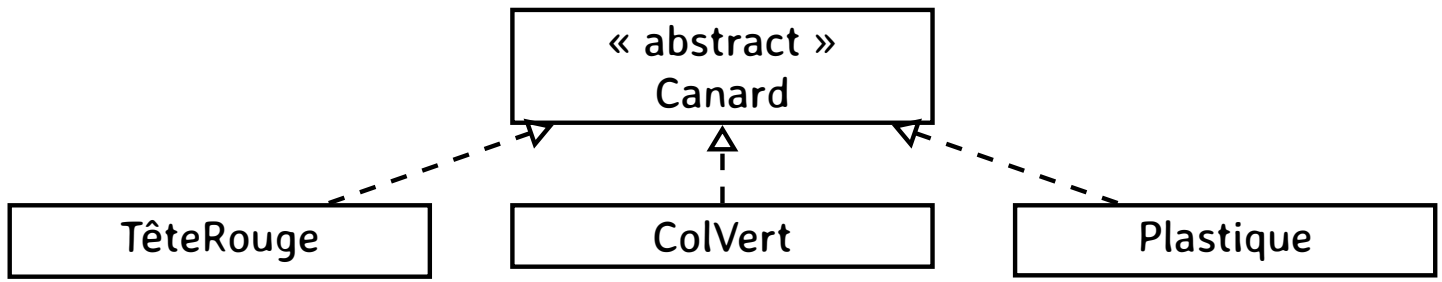
Réduire l'accessibilité des membres de classe
Règle 1.

- L'accès direct aux données membres d'une classe devrait être limité à la classe elle-même.
  Éviter d'exposer les détails d'implémentation pour faciliter l'évolution future sans aucune conséquence sur la classe.
- Solution : encapsulation
  - Faire des attributs privés
  - Réduire l'utilisation des accesseurs (getters) et mutateurs (setters) :
    - Leur nécessité est souvent révélatrice d'une mauvaise répartition des responsabilités
    - Leur utilisation suggère que l'objet est un fournisseur de données, alors qu'il faut considérer l'objet comme un fournisseur de services

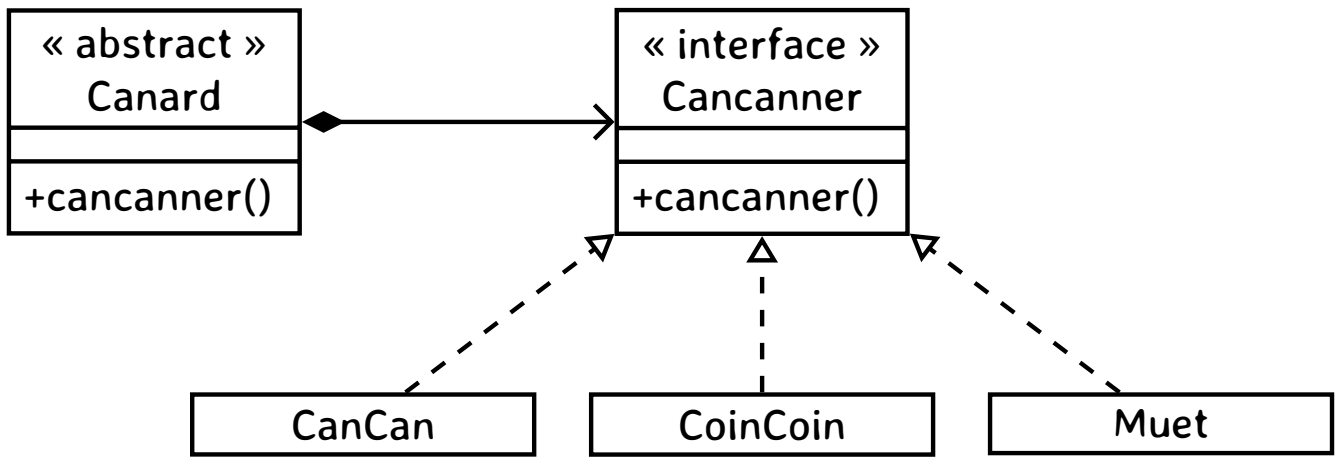
Encapsuler ce qui varie
Règle 2.

Identifier ce qui devrait être variable dans une conception puis encapsuler ce qui varie dans une hiérarchie spécifique.

☞ Variation sur un concept :



☞ Variation sur une méthode :



Programmer pour une interface, pas pour une implémentation
Règle 3.

Il faut programmer avec des supertypes (interfaces ou classes abstraites) au lieu d'instances. Les implémentations sont difficiles à changer.

☞ Programmer pour une implémentation :

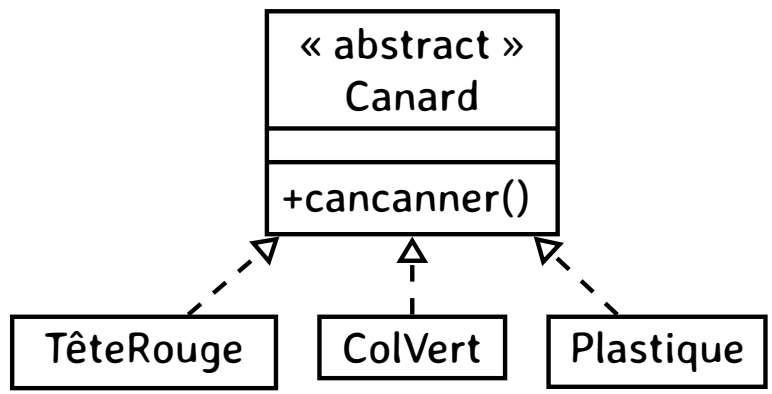
```
ColVert c = new ColVert();
c.cancane();
```

☞ Programmer pour une interface :

```
Canard c = new ColVert();
c.cancane();
```

☞ Ce qui permet des évolutions sans rien changer par ailleurs :

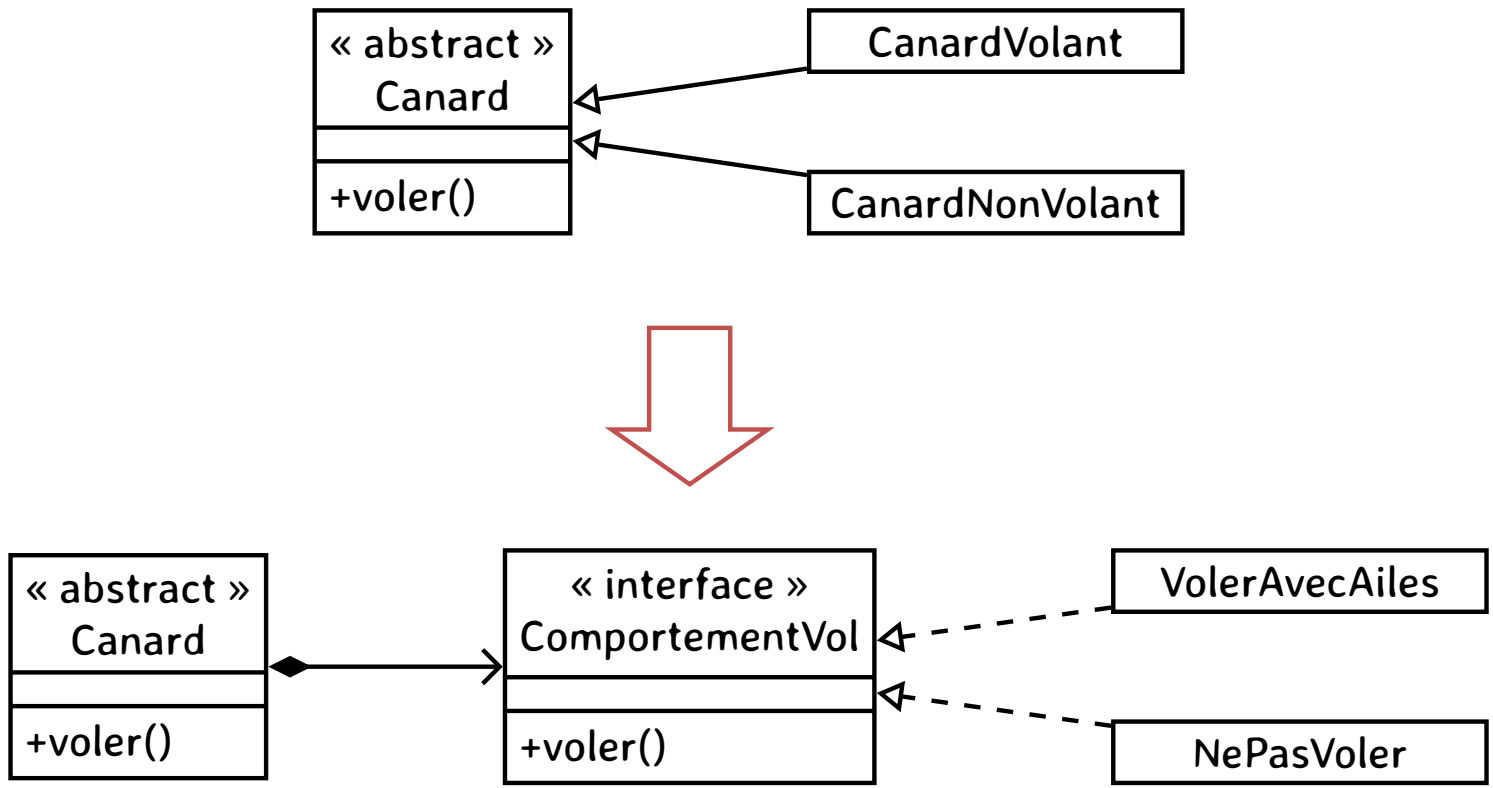
```
Canard c = getCanard();
c.cancane();
```



Privilégier la composition à l'héritage
Règle 4.

La conception est simplifiée par l'identification des comportements d'objets du système dans des interfaces séparées, au lieu de créer une relation hiérarchique pour répartir les comportements entre les classes métier par héritage.

- L'héritage rompt l'encapsulation (mécanisme de création de type boîte blanche)
- La composition est définie dynamiquement (création type boîte noire)



Principes SOLID
Plan

1. Principes de conception
2. Règles de conception

3. Conclusion

Conclusion

- Ces principes et ces règles ne fournissent pas de recettes miracles ou des lois absolues qui font de la conception un processus automatique.
  - Ne les appliquez pas pour toutes les conceptions.

    eg. le principe de responsabilité unique accroît le couplage
  - Appliquez les quand l'extension et la réutilisation sont des contraintes importantes, par exemple durant le processus de refonte de code.
- Cependant, ces principes et ces règles doivent être une préoccupation constante bien qu'ils ne doivent pas être appliqués systématiquement.

  Il faut trouver une raison de ne pas les appliquer !