INF3055
2021-2022

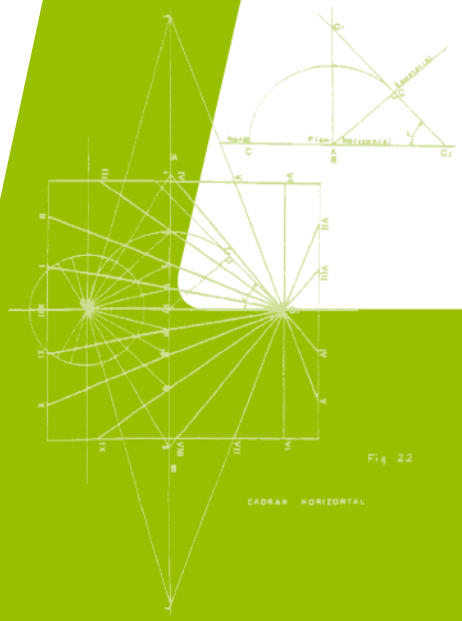
Université de Yaoundé I

INF3055 : Conception Orientée Objet

Bien concevoir

Principes SOLID

Octobre 2021



Valéry MONTHE

valery.monthe@facsciences-uy1.cm

Bureau R114, Bloc pédagogique 1

Plan du cours

1. **Pourquoi bien concevoir**
2. **Concepts de base**
3. **Les cinq principes SOLID**
    1. **S**ingle Responsibility Principle
    2. **O**pen-Closed Principle
    3. **L**iskov Substitution Principle
    4. **I**nterface Segregation Principle
    5. **D**ependency Inversion Principle

Pourquoi bien concevoir? est ce important?

- On conçoit une classe non pas pour ETRE, mais qui pourra **CHANGER**, être **REPARE**.
- Concevoir = définir un logiciel, qui est vivant
- Concevoir = formaliser des choses non définitives. Il faut toujours anticiper leur possible évolution.
- Les **objets** peuvent être **vu comme des briques rendant des services** aux autres objets et donc réutilisables.
- Les fonctions attendues sont conçues et réalisées par les interactions entres les objets
- La conception est donc une étape importante pour modéliser les éléments du monde réel et les transcrire en code

Difficulté de la conception

La conception reste difficile dans le développement logiciel, car :

- Les principes de base de la POO : abstraction, modularité, encapsulation, héritage, polymorphisme, composition, ne suffisent pas à guider dans la conception
- Les design patterns ne suffisent pas à former un tout cohérent pour la construction de designs complets.

Risques de dégénérescence de l'application

Lorsqu'une application est en PRODUCTION, les phénomènes suivantes sont observés pendant les activités de développement :

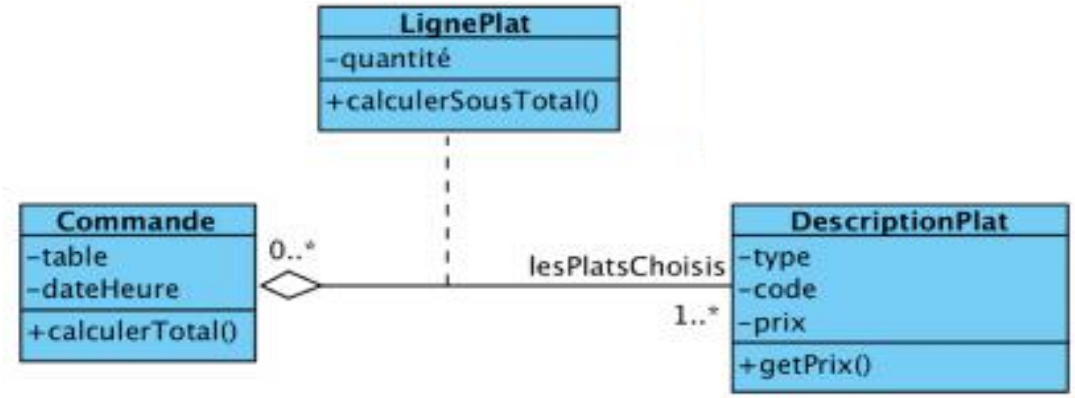
- La **rigidité** : chaque évolution risque d'impacter d'autres parties de l'application. Le coût de développement augmente et avec l'approche de la date de livraison, la qualité de code est négligée.
- La **fragilité** : modifier une partie du code entraîne des erreurs dans d'autres parties du logiciel qui devient peu robuste.
- **L'immobilité** / **non réutilisabilité** : il est difficile de retirer une partie du code pour la réutiliser ailleurs.

Source de problème : la gestion des dépendances

- Modifications de code inévitables avec l ’évolution des besoins
- La conception vise à amortir l’impact des dépendances et à aboutir aux qualités de :
  - **Robustesse**: les changement n’introduisent pas de régression;
  - **Extensibilité** : l’ajout de fonctionnalités doit être facile;
  - **Réutilisabilité** : possibilité de réutiliser certaines parties du système pour en construire d’autres.

Concepts de base

- Les **responsabilité** d'une classe :
  - Ce qu'elle **SAIT**
  - Ce qu'elle est capable de **FAIRE**
- Exemple : pour la classe **LignePlat** suivant
  - Elle **sait** : à quel objet **Commande** elle appartient, et quel objet **DescriptionPlat** elle comprend
  - Elle **sait** combien de plats elle comporte
  - Elle **peut calculer** son sous-total(prix*quantité de plats)



Concepts de base

- Le **contrat** = services rendus par une classe
  - Exprimé par les opérations de classe ou d'interface
  - **Stable**
  - Masque les détails de réalisation
- **L'implémentation**
  - Représente les classes concrètes
  - **Peut évoluer**
- Toujours chercher à **bien les dissocier**

Interface vs. Classe abstraite

- Similaires (méthodes abstraites) mais différentes (objectifs)
  - **Interface** : abstraction complète, héritage multiple d'interface, nouvelle implémentation possible à tout moment
  - **Classe Abstraite** : définition partielle possible, héritage simple qui permet de spécifier des comportements

NB : Depuis sa version 8, Java permet d'ajouter 2 types de méthodes dans une interface :

- Des **méthodes statiques**
- Des **méthodes par défaut**

Concepts de base : Cohésion / couplage

- La **Cohésion** = esprit de famille
  - Degré avec lequel les tâches d'un module sont fonctionnellement reliées entre elles
    - ✓Quel est sont objectif?
    - ✓Fait-il une ou plusieurs choses?
    - ✓Quelle est sa fonction au sein du système?
- Le **Couplage** = Dépendance
  - Force de l'interaction entre les modules d'un système
    - ✓Comment les modules travaillent ensemble?
    - ✓Qu'ont-ils besoin de savoir l'un de l'autre?
    - ✓Quand font-ils appel aux fonctionnalités de chacun?
    - ✓Exemple : une classe qui crée une instance d'une autre classe = couplage fort.

      -> Car ne peut pas être tester indépendamment de l'autre classe.

Concepts de base : Cohésion / couplage

- Une **faible cohésion** altère :
  - ✓La compréhension
  - ✓La réutilisabilité
  - ✓La maintenabilité
  - ✓Le code est fragile, car subit toute sorte de changement très fréquemment.
- Un **Fort couplage** :
  - ✓Maintenance difficile
  - ✓Lisibilité faible
  - Parfois volontaire : code rendu impénétrable pour protéger ses sources de rétro-ingénierie

Concepts de base : Cohésion / couplage

Forte cohésion : Quelques règles

- Regrouper les éléments en **forte relation**
- Regrouper les classes qui rendent des **services de même nature aux utilisateurs**
- Isoler les **classes stables** de celles qui risquent d'évoluer au cours du projet
- Isoler les classes **métiers** des classes **applicatives**

Faible couplage: Quelques règles

- Préférer un couplage avec des interfaces, pas des classes concrètes
- Ne pas ajouter plus de dépendance que nécessaire

Les principes SOLID

5 principes regroupés par Robert C. Martin

- **S**ingle Responsibility Principle
- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

SOLIDE = Single, Open, Liskov, Interface, Dependency

✓ Ces principes se renforcent entre eux mutuellement

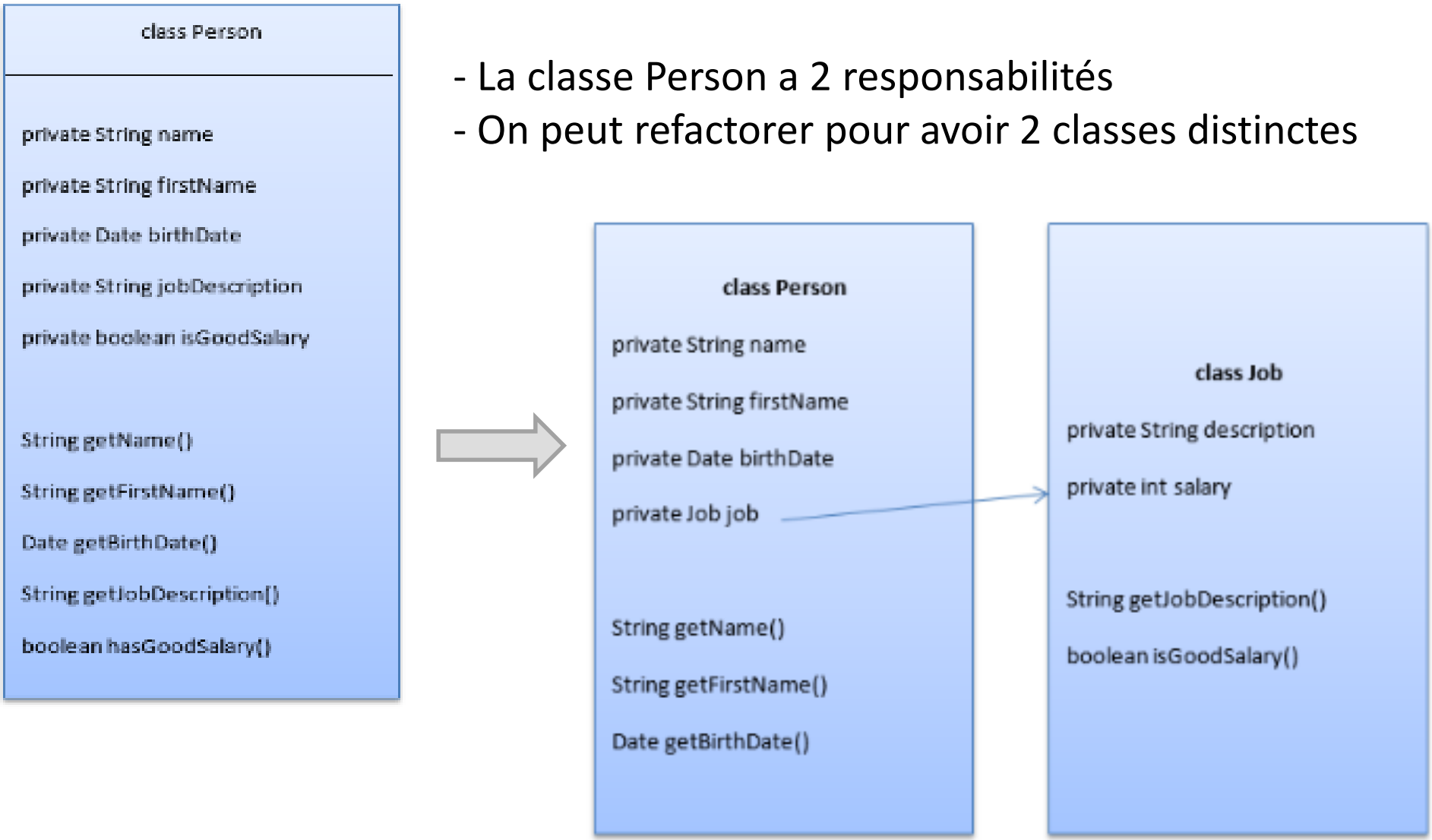
✓ C'est la base de tout code qui se veut : **claire, propre, facilement maintenable et facile à faire évoluer**

=> Le coût de changement reste toujours inférieur aux bénéfices apportés

SRP : Single Responsability Principle

- Un module(fonction,classe, paquet, etc.) ne devrait avoir qu'une seule raison de changer
- **Une seule responsabilité = une seule raison d'être modifiée**
- On a souvent tendance à donner trop de responsabilité à un objet :
  - ✓ Analyser les méthodes de la classe
  - ✓ Les regrouper pour constituer des ensembles homogènes : accès BD, API, etc
  - ✓ Affecter si possible les responsabilités correspondants aux informations que la classe possède.
- Pour savoir si une classe respecte le SRP, il faut dire :
  - ✓ « La classe X fait … » en étant le plus spécifique possible.
  - ✓ Si la phrase ci-dessus contient des ***et*** ou des ***ou***, alors la classe a plus d'une responsabilité.
  - ✓ Si elle contient des mots génériques comme ***gère*** ou ***objet*** (exemple: gère les utilisateurs, valide les objets), alors le SRP n'est pas respecté

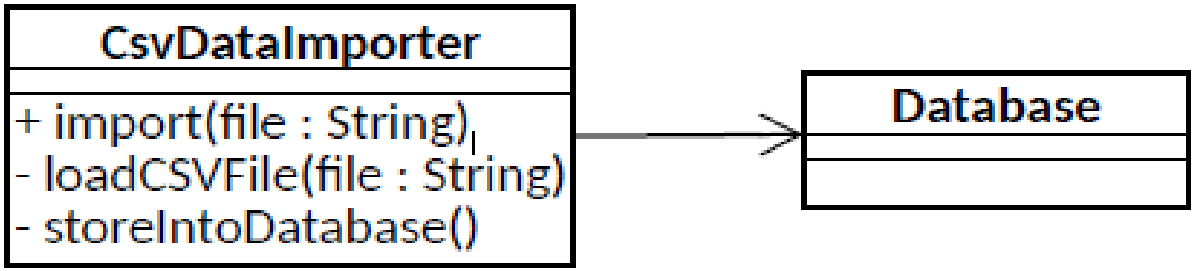
SRP : Exemple 1



- La classe Person a 2 responsabilités
- On peut refactorer pour avoir 2 classes distinctes

SRP : Exemple 2

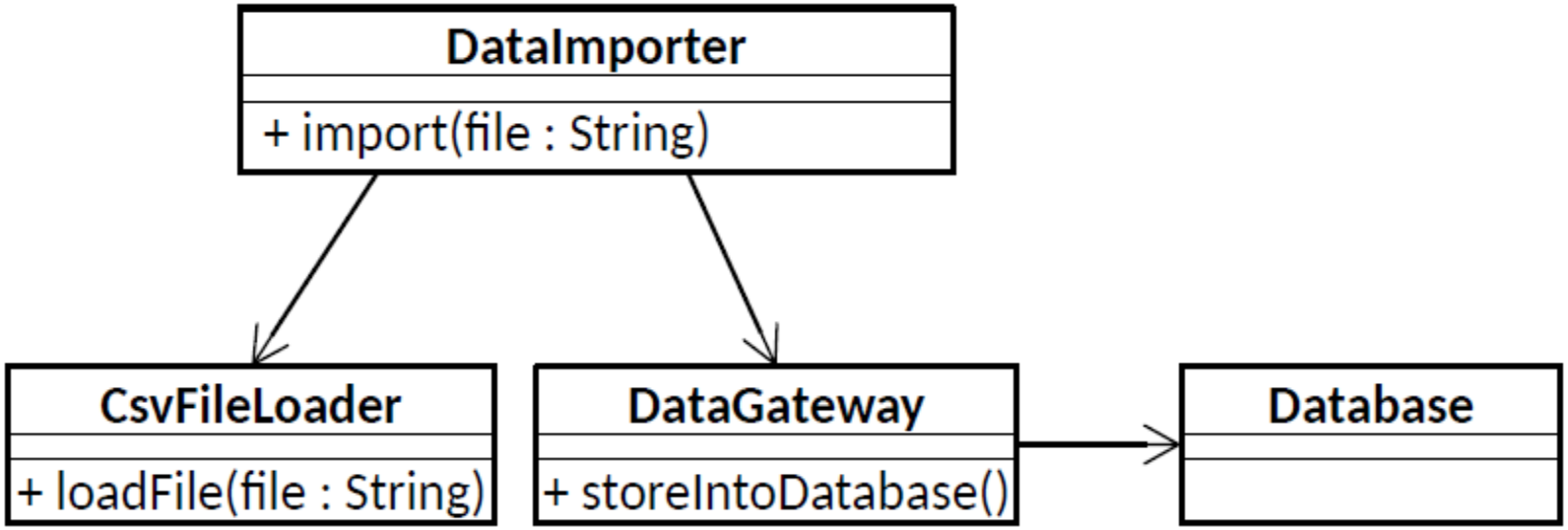
- Importer des données d'un fichier CSV dans une base de données



- Problème avec cette conception
  - Deux responsabilité de la classe CsvDataImporter
    - ✓ *Lire le fichier CSV sous forme d'enregistrements*
    - ✓ *Stocker les enregistrement dans la base de données*

SRP : Exemple 2 (refactoring)

- Une solution (refactoring)
  - Comment augmenter la cohésion?
    - ✓ Externaliser et séparer les responsabilités : chargeur de fichier et la passerelle de stockage



OCP : Open Close Principle

- La rigidité et la fragilité du code viennent de l'impact d'un changement d'une partie de l'application sur d'autres.
- Les entités logicielles (classes, packages, etc) doivent êtres :
  - ✓ **Ouvertes à l'extension** : on peut ajouter des fonctionnalités non prévues à la création
  - ✓ **Fermées à la modification** : les changements introduits ne modifient pas le code existant
- L'extensibilité se traduit par l'ajout de code uniquement
- Une fois le code produit, testé et livré en production, le seul moyen de modifier est d'étendre le code
  - ✓ Assure que le code existant ne sera pas altéré au risque d'entrainer des régressions.
- L'**abstraction** et le **polymorphisme** sont les moyens pour y parvenir

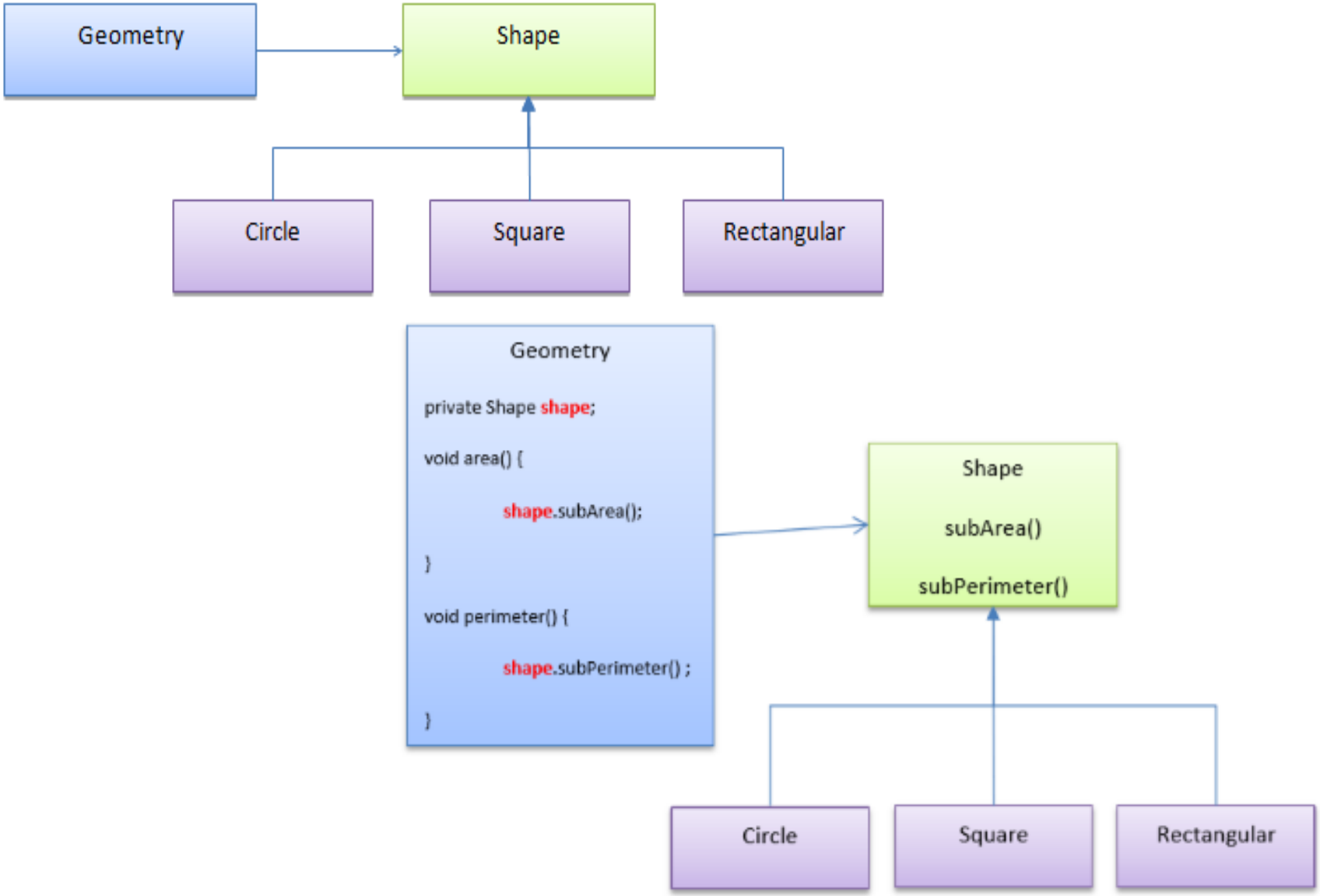
OCP : Exemple 1

```
1. class Geometry {
2.     void area(Shape shape) {
3.         if (shape instanceof Circle) {
4.             // calculate for circle
5.         }
6.         else if (shape instance of Square) {
7.             // calculate for square
8.         }
9.     }
10.
11.    void perimeter (Shape shape) {
12.        if (shape instanceof Circle) {
13.            // calculate for circle
14.        }
15.        else if (shape instanceof Square()) {
16.            // calculate for square
17.        }
18.    }
19. }
```

- la classe géométrie gère les deux formes Cercle et Carre.
- Si une nouvelle forme doit être ajoutée, il faut modifier Geometry

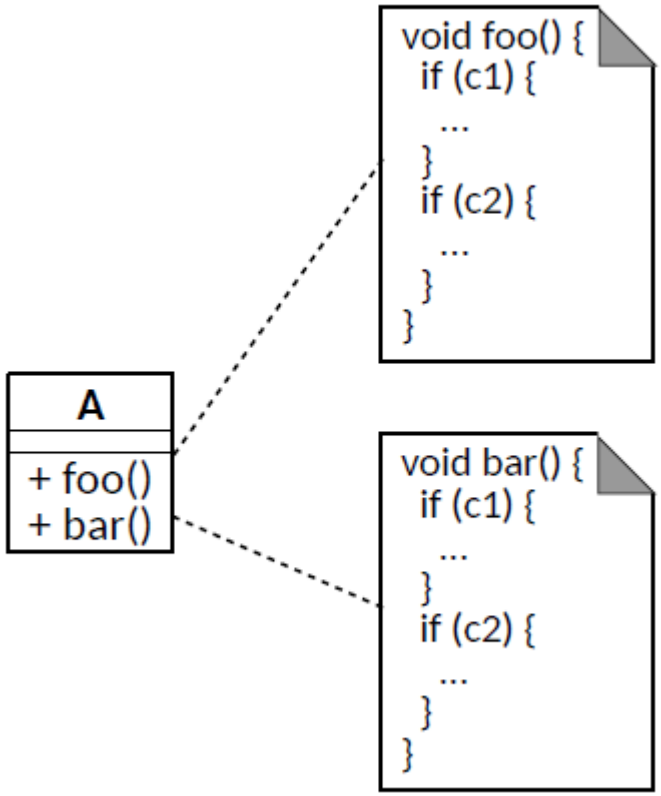
OCP : Exemple 1

- L' OCP est incontournable pour rendre le code flexible



OCP : Exemple 2

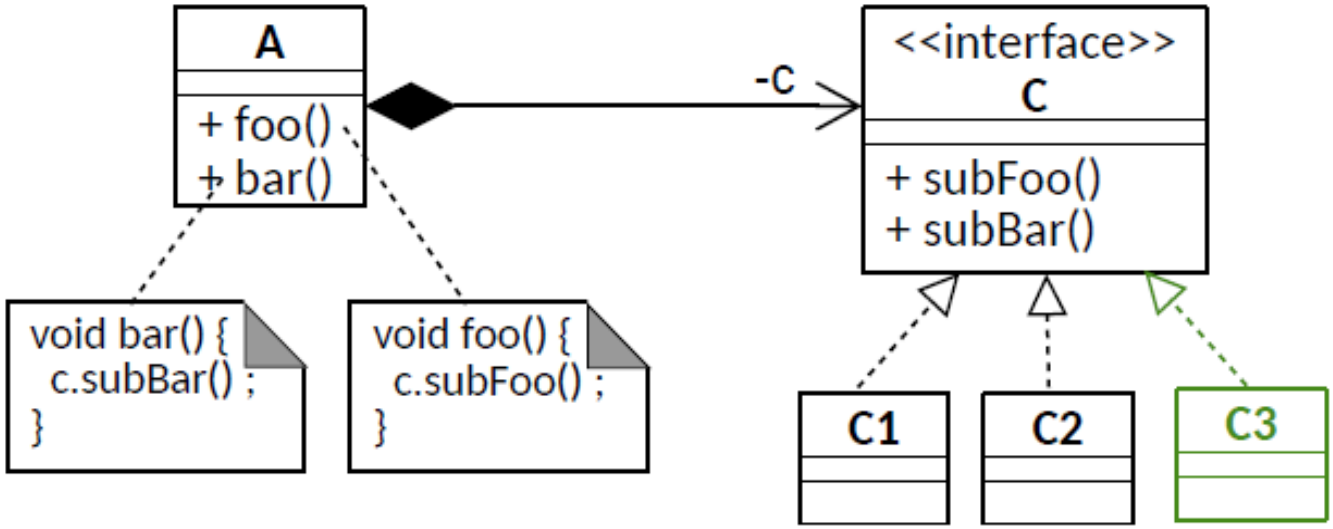
- Considérons la classe A avec deux méthodes foo() et bar()



- Quelle est la faiblesse de cette concfeption?

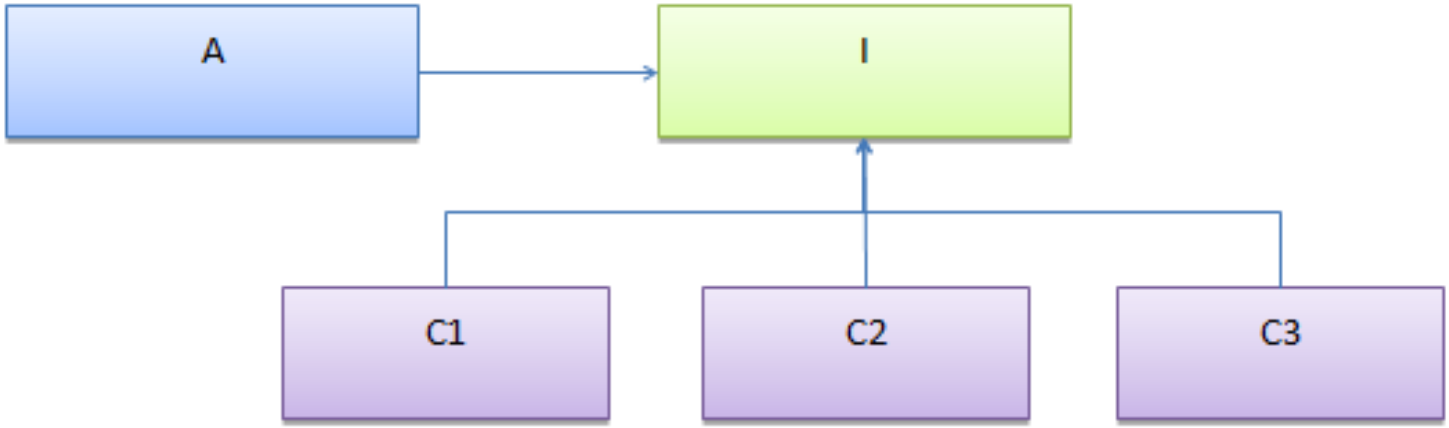
OCP : Exemple 2 (refactoring)

- Comment la rendre ouverte aux extensions et fermée aux modifications?
  - ✓ *Composition, abstraction et polymorphisme*

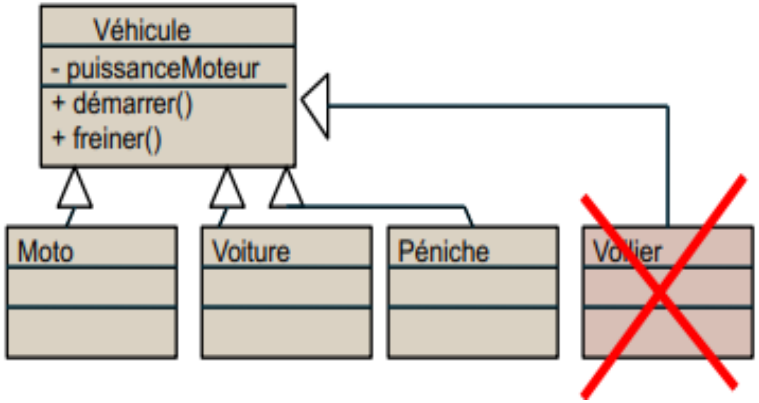


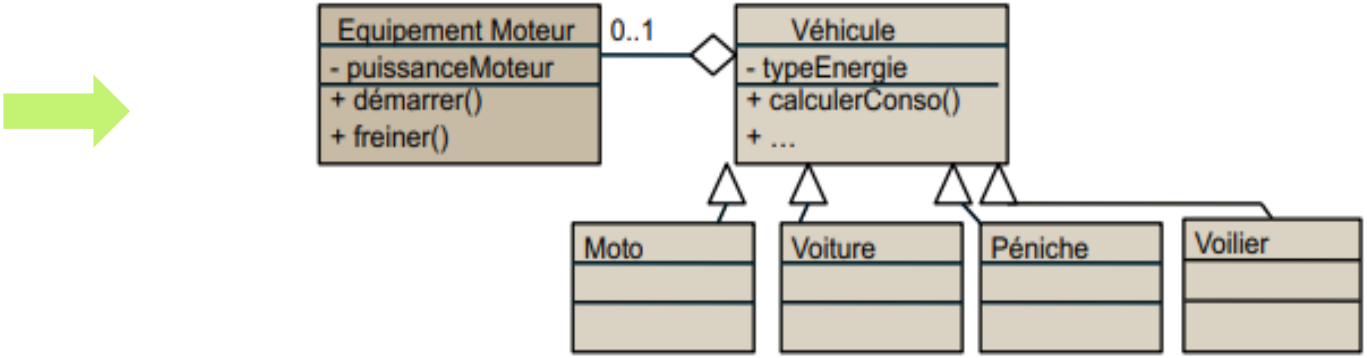
LSP: Liskov Substitution Principle

- On doit pouvoir placer la sous-classe partout dans le code où figure la classe parent
- Un sous-type S doit être substituable à son type de base T dans toute l'application où T est utilisé sans causer de comportement non désiré dans le programme.
- *Si B et C sont des implémentations de A, alors B et C doivent pouvoir être inter-changées sans affecter l'exécution du programme.*



LSP: Exemple 1





- Pour Voilier, il faudrait redéfinir démarrer() et freiner() en méthodes vides (qui ne font rien) !

LSP: Exemple 2

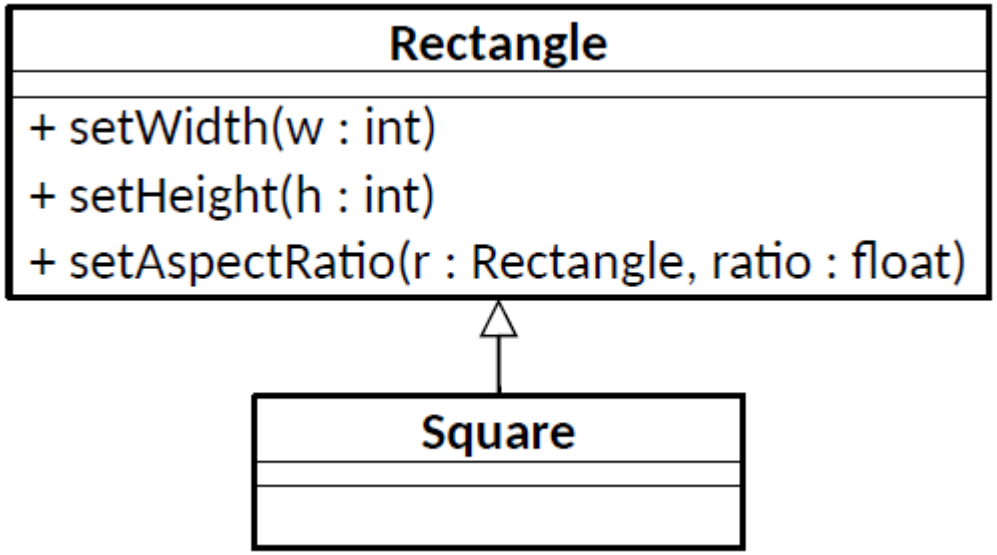
```java
public class User {
  private String emailAddress;

  public String getEmailAddress() {
    return this.emailAddress;
  }
}

public class AnonymousUser extends User {
  public String getEmailAddress() {
    throw new RuntimeException("Anonymous users don't have an email address.");
  }
}
```



```java
package com.ezoqc.blog.solid.lsp;

public abstract class User {

}

public class RegisteredUser extends User{
  private String emailAddress;

  public String getEmailAddress() {
    return this.emailAddress;
  }
}

public class AnonymousUser extends User {

}
```

LSP: Exemple 3

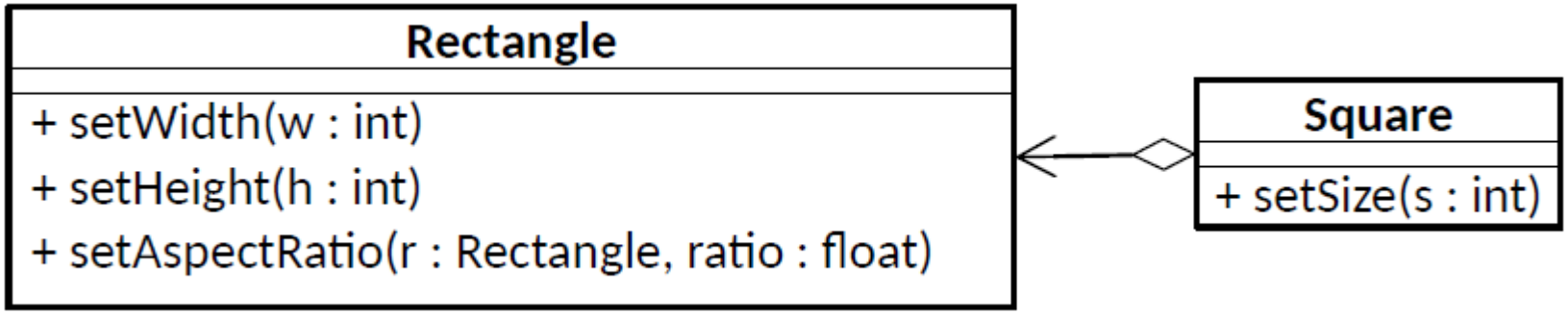
- Soit la modélisation suivante : un carré est un rectangle particulier



- Le carré ne respecte pas tout le contrat de Rectangle,
- **Par exmple** : après l'appel de ***setWith()*** on s'attend à ce que la largeur ait la nouvelle valeur et la hauteur conserve son ancienne valeur. Or ce n'est pas le cas pour le carré

LSP: Exemple 3 (refactoring)

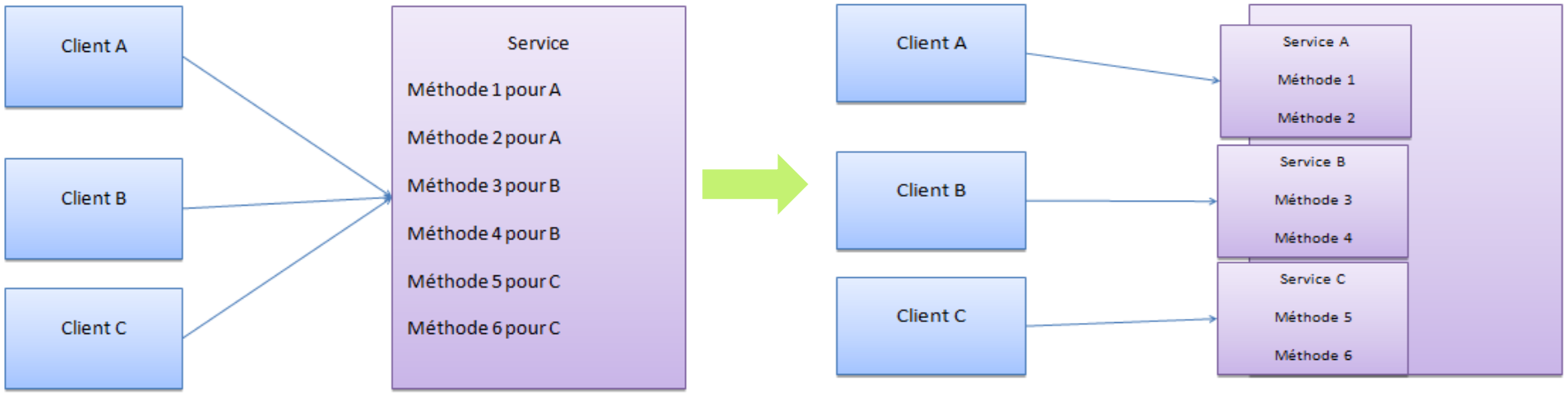
- Le carré n'hérite plus de rectangle
- Le carré utilise le rectangle par composition



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

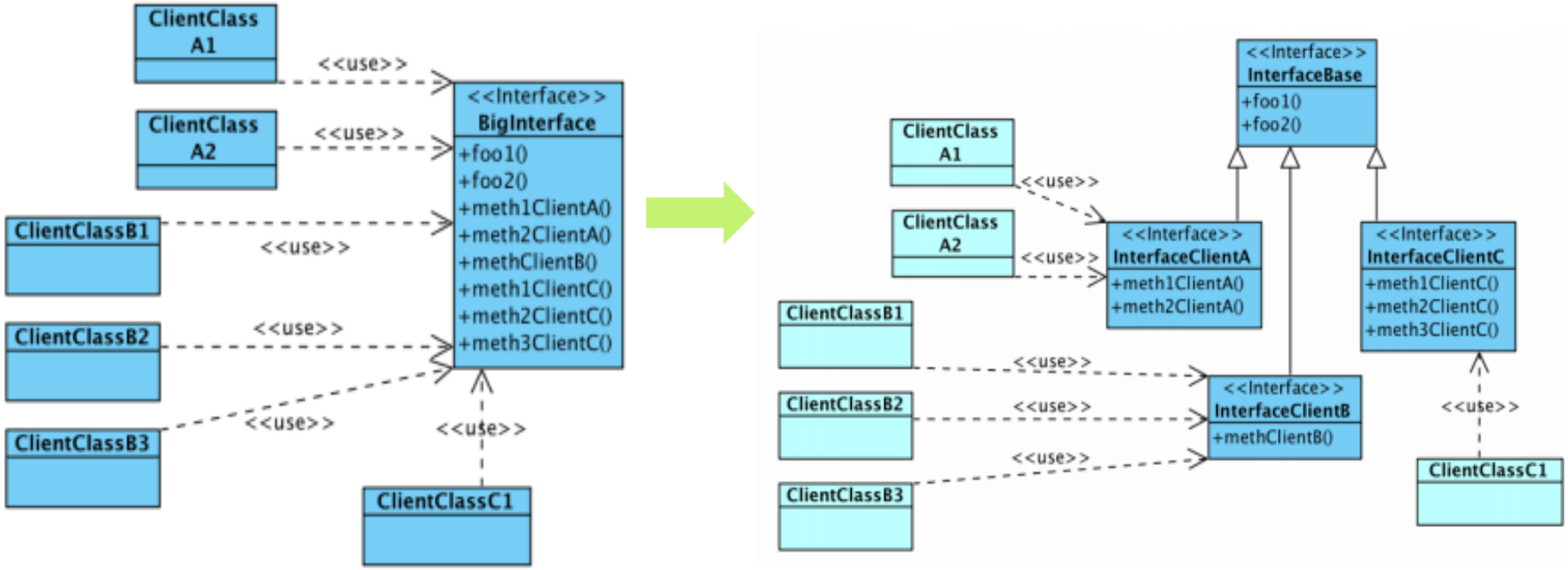
ISP: Interface Segregation Principle

- La dépendance d'une classe à une autre doit 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.
  - ✓ Le client ne doit voir que les services dont il a besoin
  - ✓ Toute classe client qui utilise une BigInterface a (sauf pour son concepteur) un comportement flou
  - ✓ Toute classe réalisant une interface doit implémenter chacune de ses fonctions



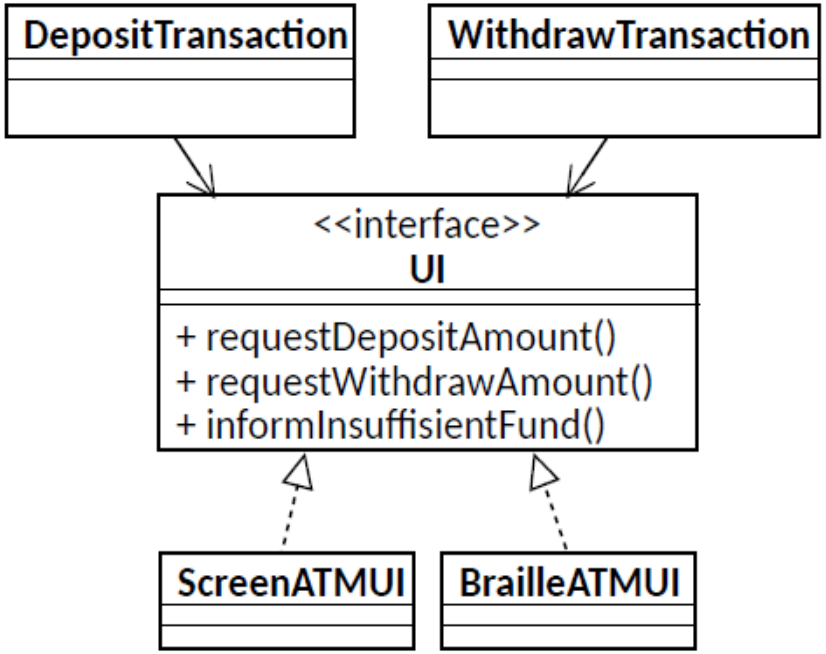
ISP: Exemple 1

- L'appelant ne devrait pas connaitre les méthodes qu'il n'a pas à utiliser



ISP: Exemple 2

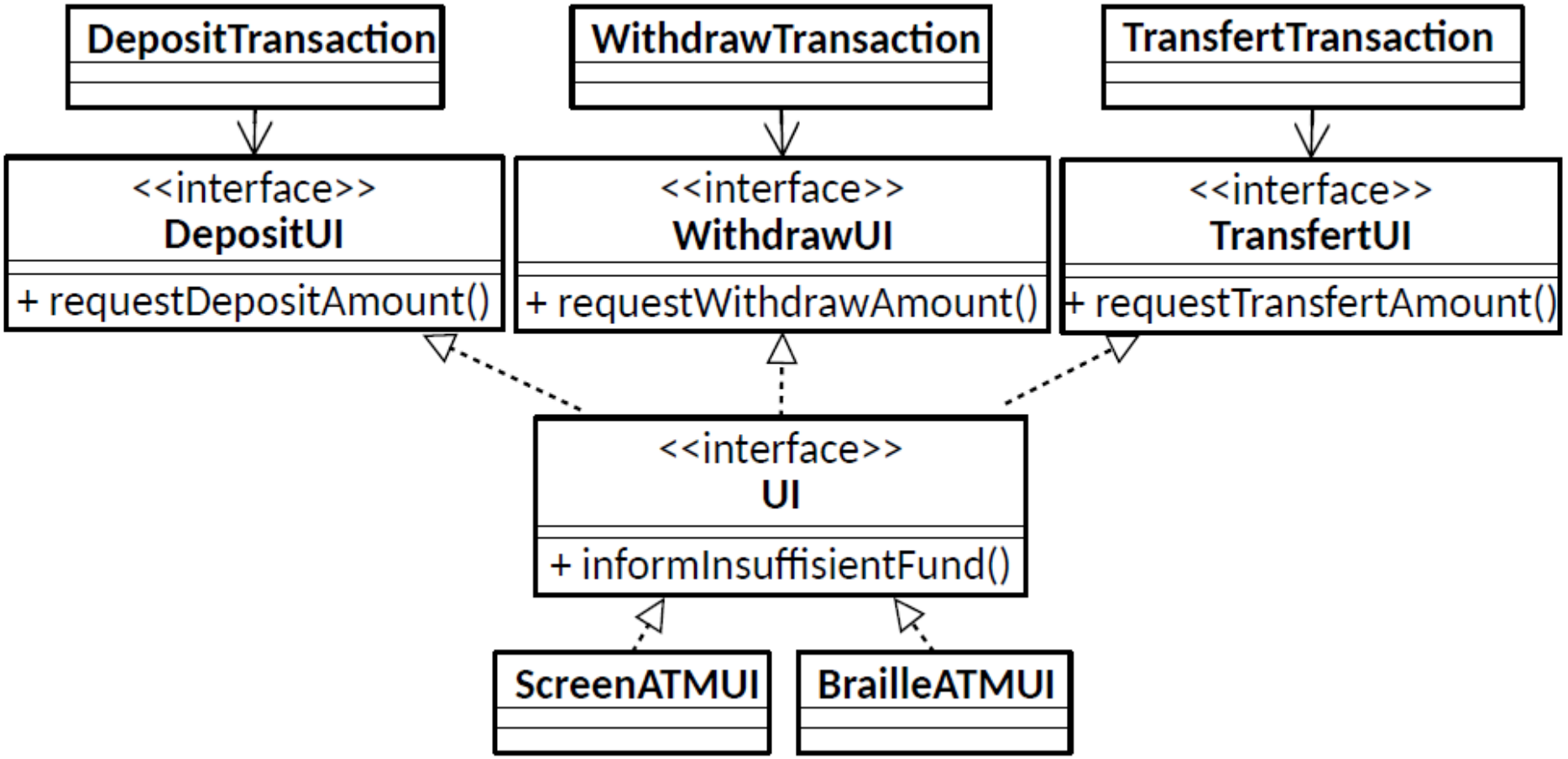
- Guichet Automatique Bancaire (GAB)



- Toutes les transactions interagissent avec la même interface leur permettant d'utiliser des services d'une interface écran ou braille
- **Problèmes potentiels** : la modification d'une méthode impacte toutes les classes dépendantes même si elles n'utilisent pas la méthodes

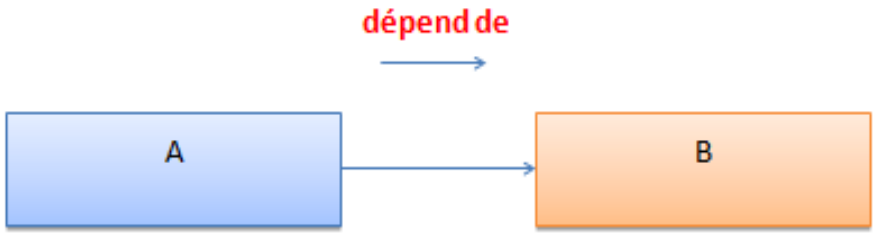
ISP: Exemple 2 (refactoring)

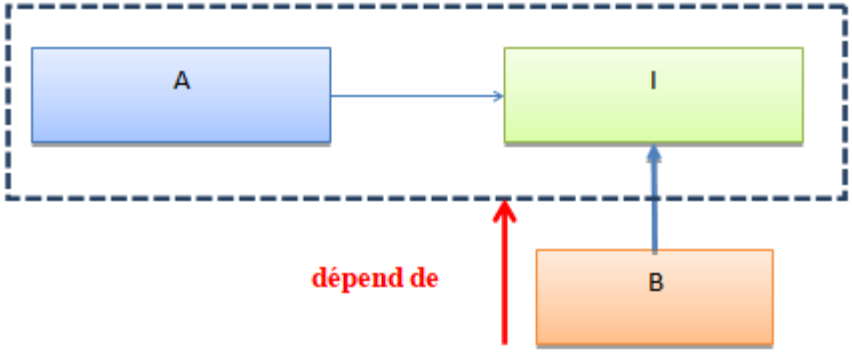
- **Une solution** : Ségrégation par héritage d'interfaces
  - Chaque client n'est lié qu'à une interface minimale sous-ensemble de l'interface intégrale construite par héritage.



DIP : Dependency Inversion Principle

- Les modules de haut niveau (aspect métier) ne doivent pas dépendre de modules de bas niveau (aspect implémentation)
- Les modules d'une application devraient dépendre d'abstractions
  - ✓ Les abstractions ne doivent pas dépendre de détails
  - ✓ Les détails doivent dépendre des abstractions
- Les dépendances d'une classe ne devraient pas être concrètes
  - ✓ Elle ne doit pas connaître l'implémentation de ses dépendances





DIP : Exemple 1

```java
class Logger {
  public void log(String msg) {
    System.out.println(msg);
  }
}

class SomeService {
  private Logger logger;

  public SomeService() {
    this.logger = new Logger();
  }

  public void someMethod() {
    this.logger.log("Hi!");
  }
}
```



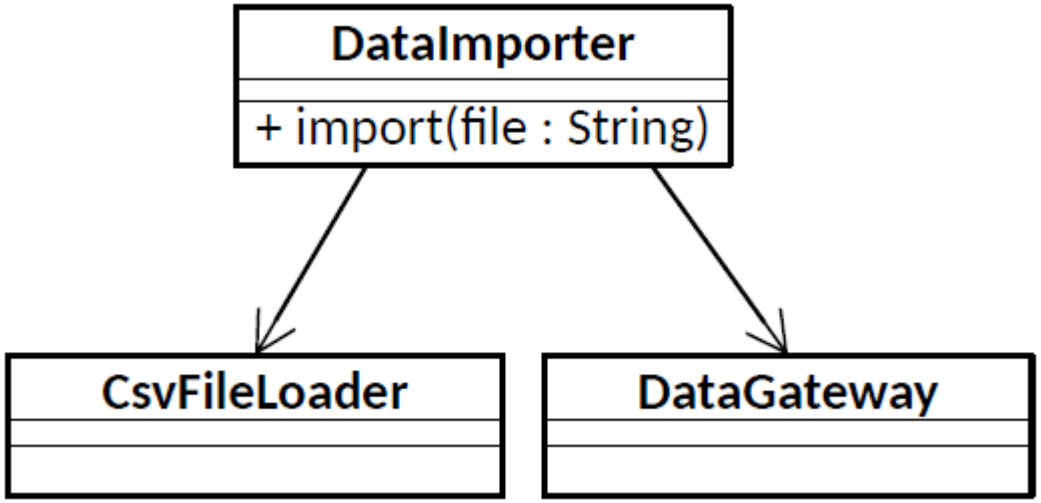
```java
class ConsoleLogger extends Logger {
  public void log(String msg) {
    System.out.println(msg);
  }
}

class SomeService {
  private Logger logger;

  public SomeService() {
    this.logger = new ConsoleLogger();
  }

  public void someMethod() {
    this.logger.log("Hi!");
  }
}
```



```java
interface Logger {
  void log(String msg);
}

class ConsoleLogger extends Logger {
  public void log(String msg) {
    System.out.println(msg);
  }
}

class SomeService {
  private Logger logger;

  public SomeService(Logger logger) {
    this.logger = logger;
  }

  public void someMethod() {
    this.logger.log("Hi!");
  }
}
```

DIP : Exemple 2

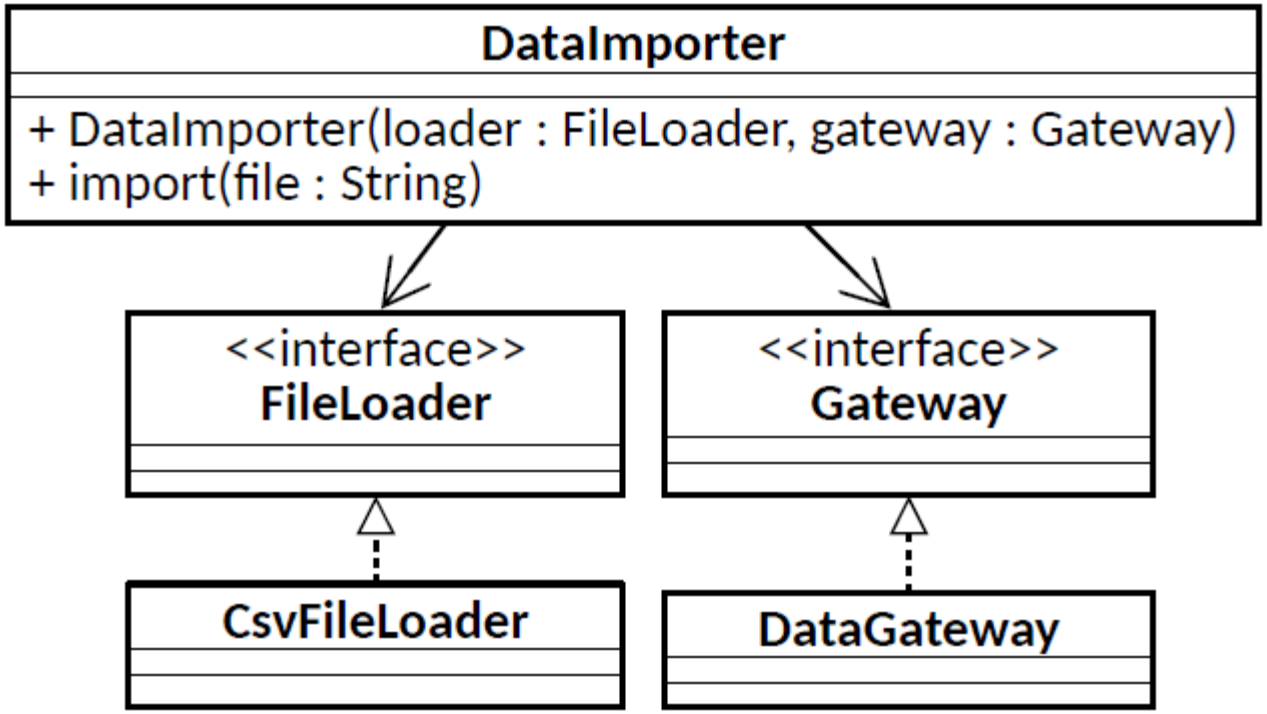
- Dans l'exemple ci-dessous, la classe DataImporter est dépendante du chargeur de fichier et la passerelle de sctockage dans la BD.



- **Problème** : on ne peut pas réutiliser la classe d'importation sans réutiliser le chargeur de fichier et la passerelle de stockage des données.

DIP : Exemple 2 (Refactoring)

- **Une solution** : inverser les dépendances avec les interfaces.



Bibliographie et ressources

Ouvrages recommandés

- Software Architecture in Practice, 3e édition, Len Bass, Paul Clements et Rick Kazman, Addison-Wesley, 2012.
- Architecture logicielle : Concevoir des applications simples, sûres et adaptable, 2e edition, Jacques Printz, Dunod.

Notes de cours

- Principes avancés de conception objet, Régis Clouard, ENSICAEN-GREYC
- Architecture logicielle, Université Joseph Fourier, Lydie du Bousquet