ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

02
Chapitre

Principes avancés de conception objet

2I1AC3 : Génie logiciel et Patrons de conception

Régis Clouard, ENSICAEN - GREYC

« Ce n'est pas l'espèce la plus puissante qui survit, mais celle qui s'adapte le mieux au changement. »
Charles Darwin

Introduction

- **Objectif** : produire des conceptions extensibles, maintenables et réutilisables.
- **Problème** : la conception relève de l'art, cf. artisan du logiciel.
- **Solution** : il faut s'aider du savoir-faire formalisé sous la forme :
  1. ***Principes de conception*** : base sur laquelle repose l'organisation de la conception et qui en régit le fonctionnement
  2. ***Règles de conception*** : un ensemble de prescriptions de conception à respecter
  3. ***Patrons de conception*** : des modèles de solutions à des problèmes récurrents
- **Références**

Martin Fowler

Barbara Liskov
(prix Turing 2008)

Robert Martin
(Uncle Bob)

Bertrand Meyer

I. 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

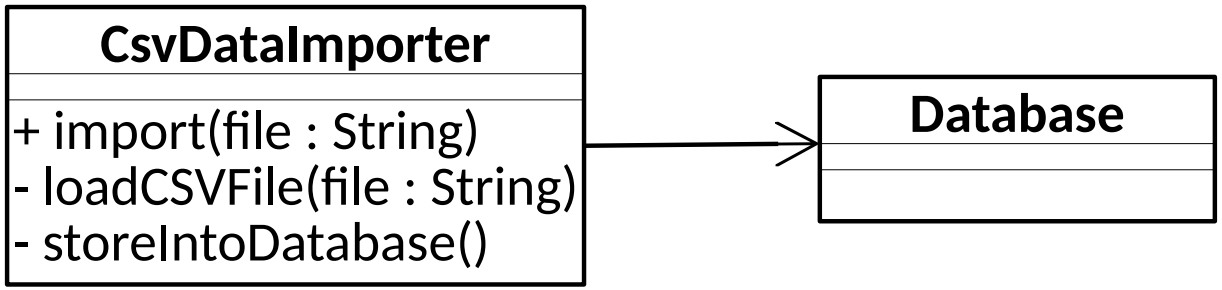
Principe 1. Responsabilité unique

- **Un module (fonction, classe, paquet, etc.) devrait n'avoir qu'une responsabilité unique**.
  - La responsabilité unique doit s'entendre comme une seule raison de changer.
  - Donc, la phrase précédente peut être reformulée en :

    Un module ne devrait jamais avoir plus d'une raison de changer.
- Le but est évidemment d'augmenter la cohésion.
- Péril majeur d'un module à responsabilités multiples.
  - **Immobilité** : Il est impossible de ne réutiliser qu'une partie du module sans réutiliser le module entier.

Exemple

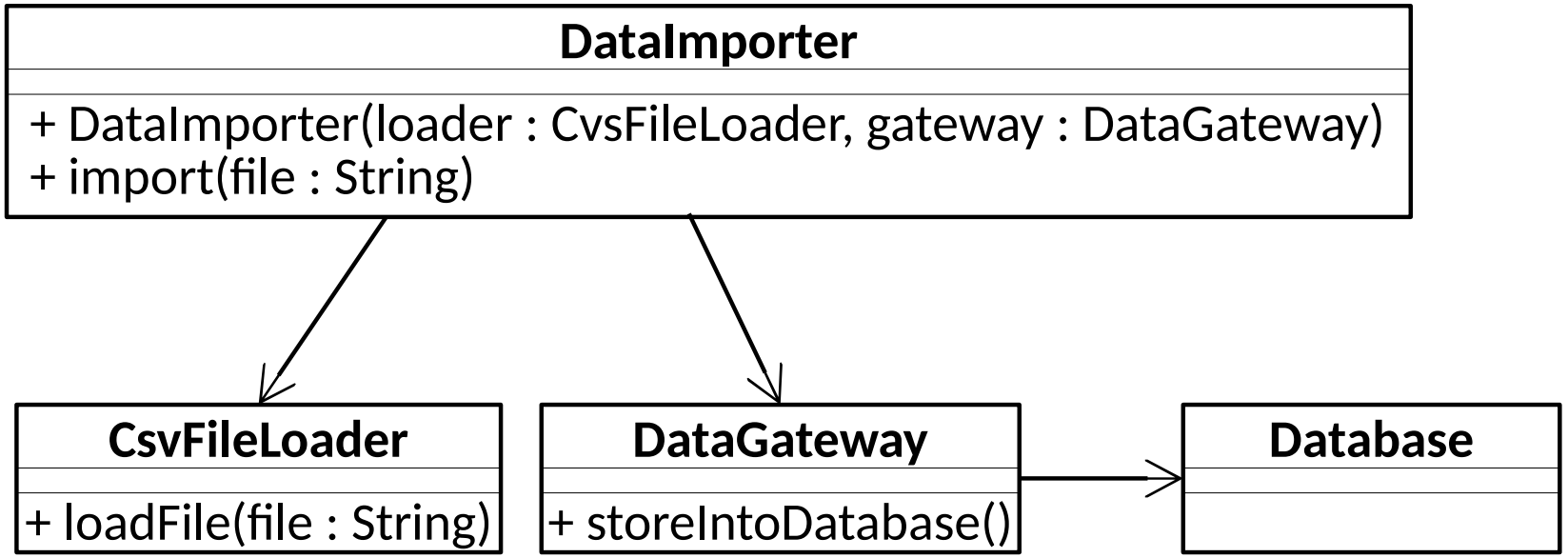
- Importer des données d'un fichier CSV dans une base de données.



- Quel est le problème avec cette conception ?
  - Il y a 2 responsabilités donc 2 raisons de changer.
    1) Lire le fichier CSV sous forme d'enregistrements.
    2) Stocker les enregistrements dans la base de données.

Exemple (refactoring)

- Comment augmenter la cohésion ?
  - Séparer le chargeur de fichier et la passerelle de stockage.



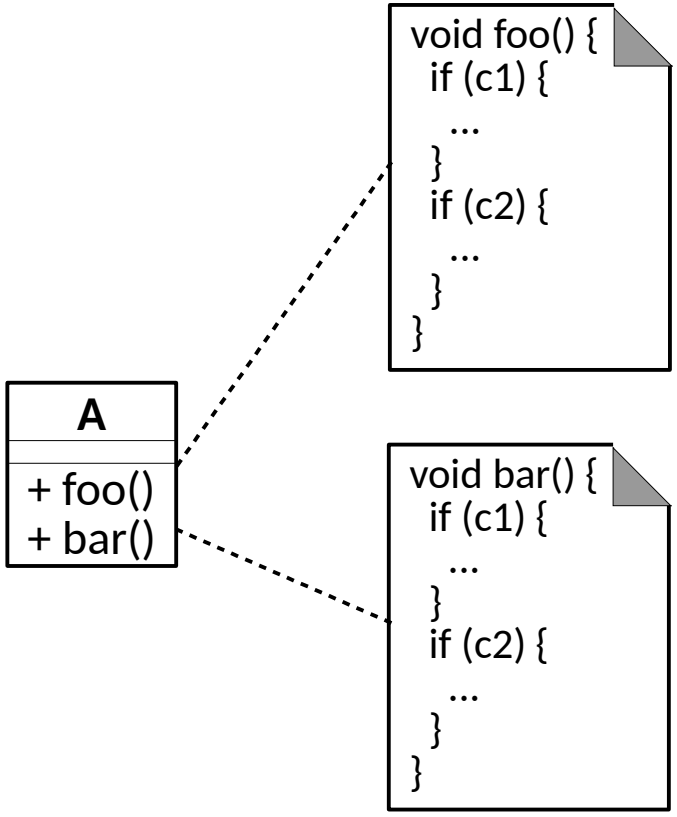
  - On se retrouve avec 3 petites classes :
    - facile à changer,
    - facile à étendre,
    - facile à tester.

Principe 2. Ouverture / Fermeture

- **Un module doit être ouvert aux extensions, mais fermé aux modifications**.
  - Nous devons ajouter une nouvelle fonctionnalité en ajoutant du nouveau code et non en éditant du code existant.
- Péril majeur d'un code fermé.
  - **Fragilité** : la modification de code existant entraîne une régression en introduisant de nouvelles erreurs dans les parties préexistantes.

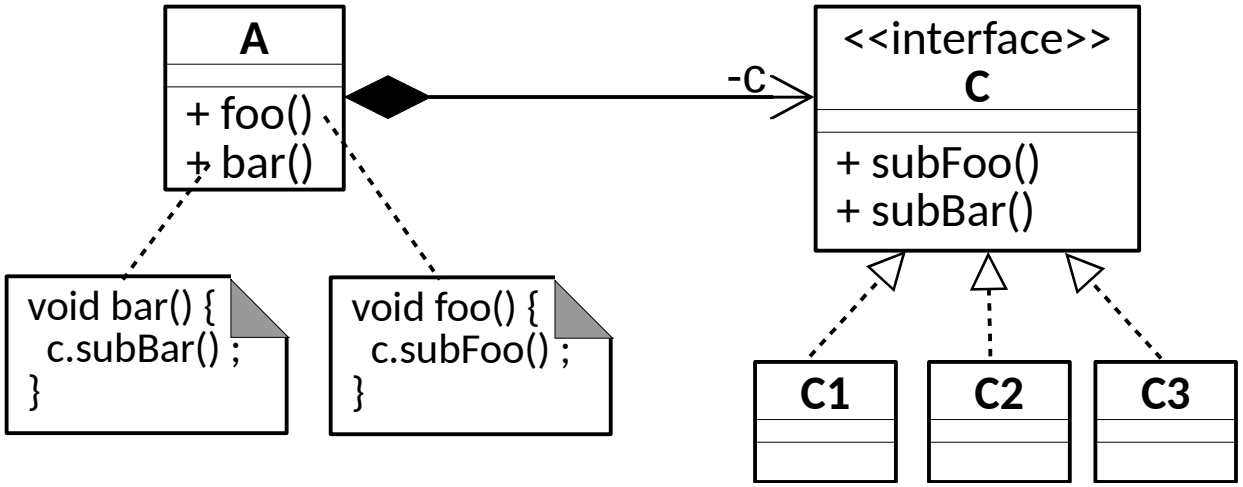
Exemple

- Considérons la classe `A` avec deux méthodes qui dépendent des deux attributs `c1` et `c2`.
  - Quelle est la faiblesse de cette conception ?
    - Elle est fermée à l'introduction d'une nouvelle alternative pour l'attribut `c3`. L'ajout nécessite d'éditer le code des méthodes `foo()` et `bar()`.



Exemple (refactoring)

- Comment la rendre ouverte aux extensions et fermée au modifications ?
  - Abstraction et polymorphisme.



Corollaire : le principe de choix unique

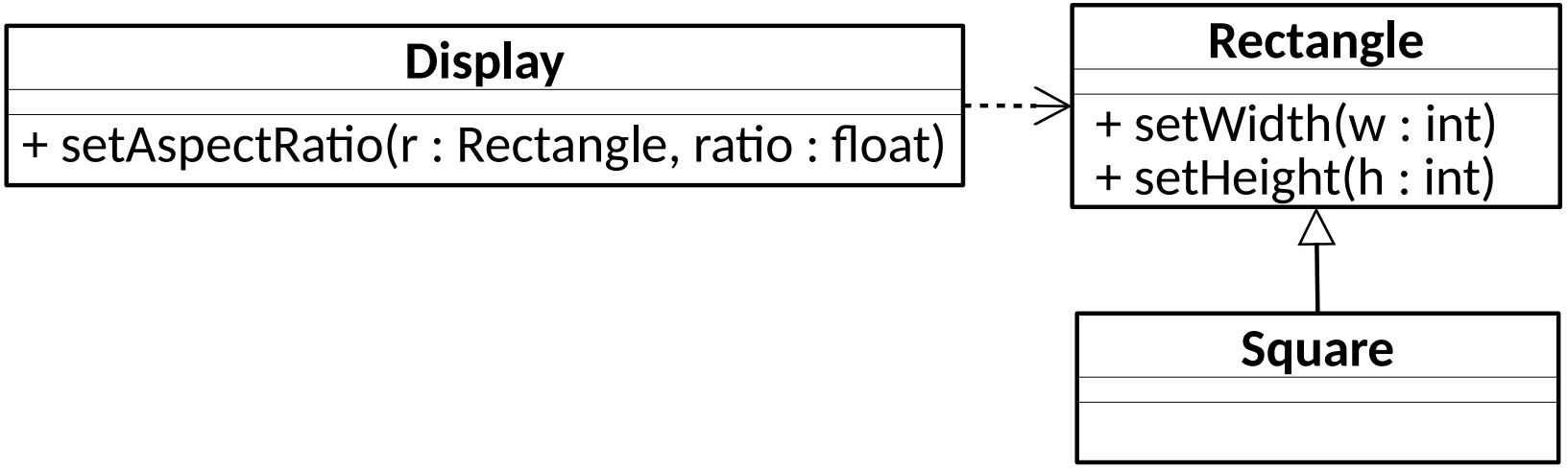
- Aucun programme ne peut être ouvert à 100 %.
  - Par exemple, dans l'exemple précédent, il y a quelque part quelqu'un qui doit choisir entre les classes `C1`, `C2` ou `C3`.
- Dans ce cas, une seule méthode ou une seule classe dans le système doit connaître l'ensemble des alternatives.
  - cf. les patrons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

Principe 3. Substitution de Liskov

- **Les objets de classes dérivées doivent être substituables aux objets de la classe de base.**
  - Les objets de la classe dérivée doivent se comporter d'une manière compatible avec les promesses faites dans le contrat de la classe de base.
    - Si S est un sous-type de T, alors tous les objets de T doivent être substituables par tout objet de S.
    - Par exemple, lorsque T est utilisé comme paramètre d'une fonction, il doit être substituable par un objet de S.
  - Cela revient à dire que la classe de base est une interface exportée par toutes les sous-classes.
- 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.

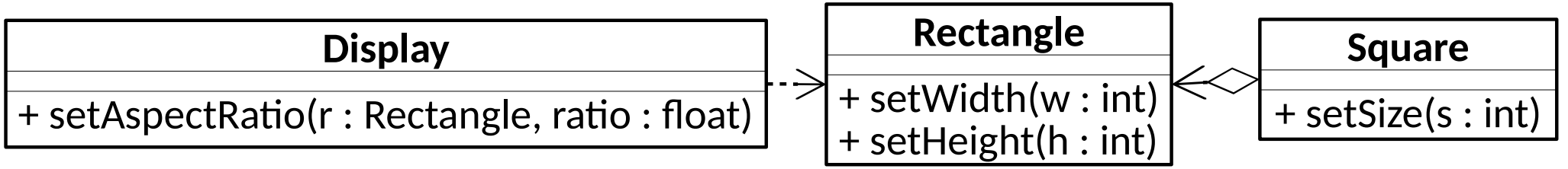
Exemple

- Soit la modélisation suivante que l'on trouve comme exemple d'héritage dans tous les manuels de conception orientée objet : un carré est un rectangle particulier.
  - Pourquoi cette modélisation est fausse ?
    - Le carré ne respecte pas tout le contrat de *Rectangle*.
      - Par exemple, la méthode `setAspectRatio()` (*4/3*, *16/9*, *A4*, *etc*) n'a certainement pas le comportement attendu avec un carré.
      - Par exemple, 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 la carré.



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.



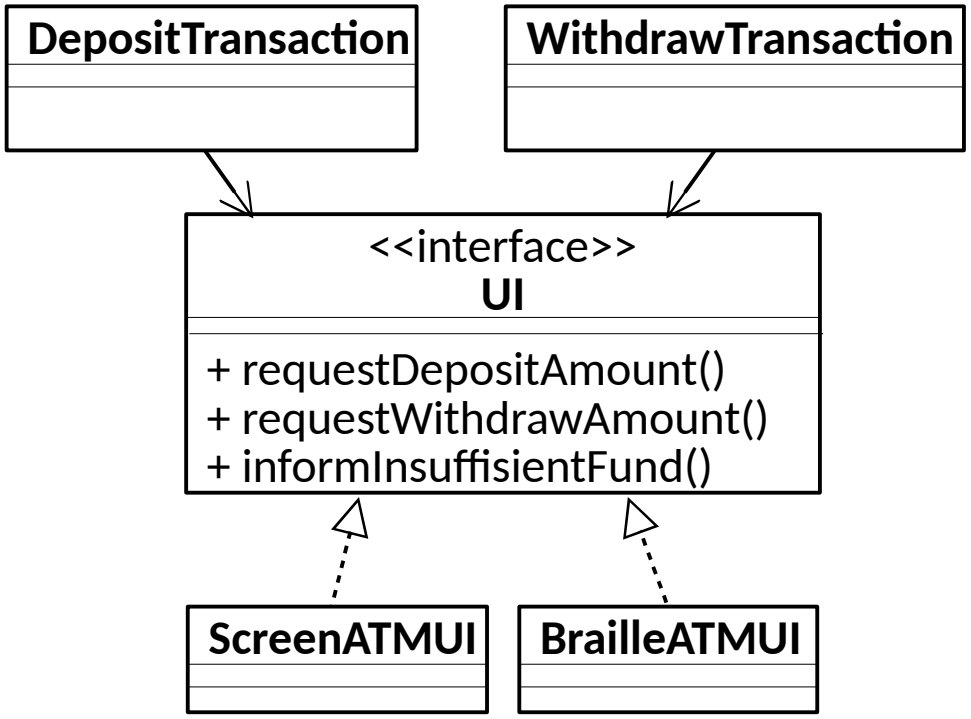
- Cette fois le carré n'est pas substituable au rectangle.

Principe 4. Ségrégation d'interface

- **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.
- Péril majeur d'interfaces complexes :
  - **Rigidité :** Le client se trouve affecté par des changements dans des méthodes qu'il n'utilise pas.

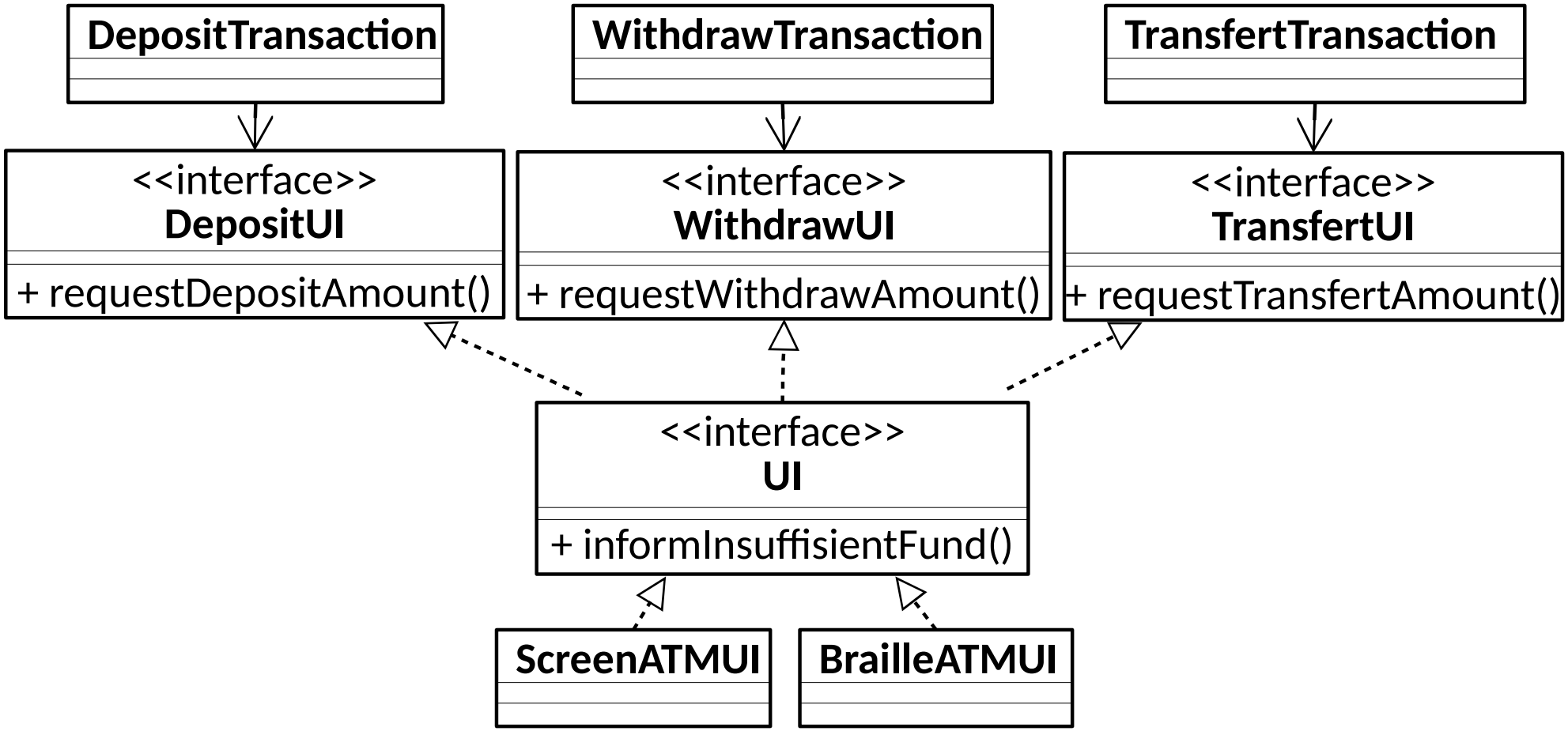
Exemple

- Guichet Automatique Bancaire (GAB)
  - Toutes les transactions partagent la même interface.
  - Quels sont les problèmes potentiels ?
    - La modification d'une méthode impacte toutes les classes dépendantes même si elles n'utilisent pas la méthode.



Exemple (refactoring)

- 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.

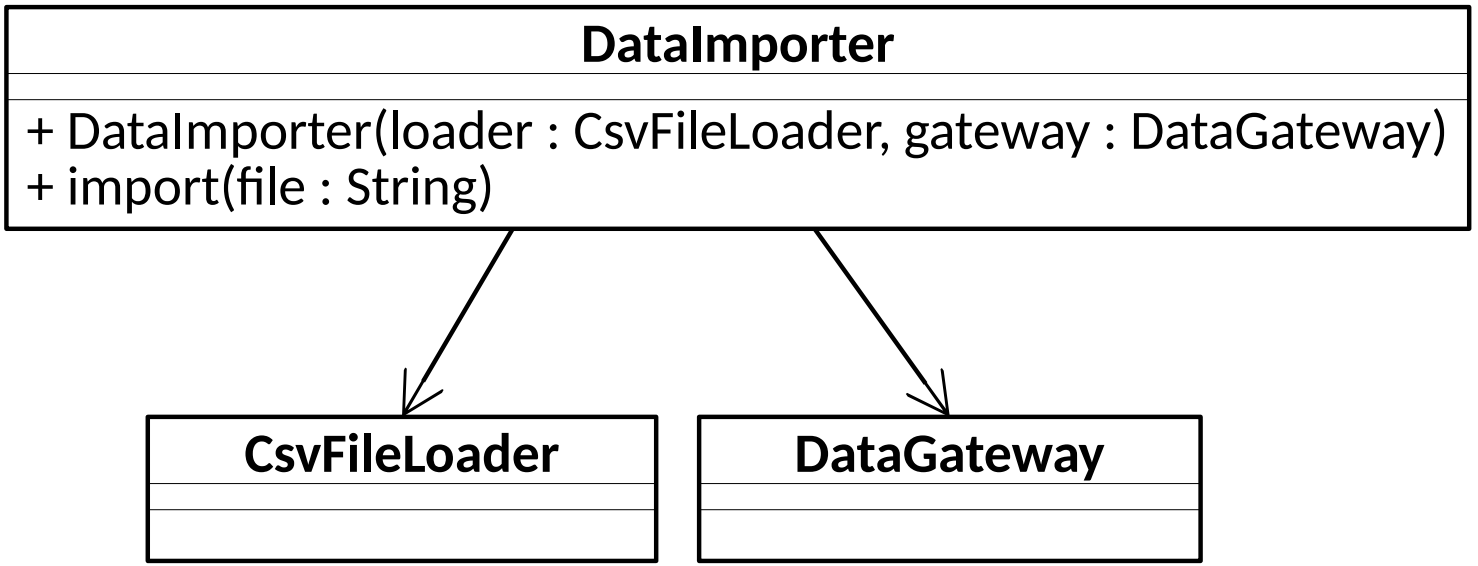


Principe 5. Inversion des dépendances

- **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.
- Périls majeurs de dépendances conventionnelles
  - **Rigidité** : Les modules fonctionnels sont impactés par les changements des modules techniques.
  - **Immobilité** : Les modules génériques ne peuvent pas être réutilisés parce qu'ils sont liés à une implémentation particulière.

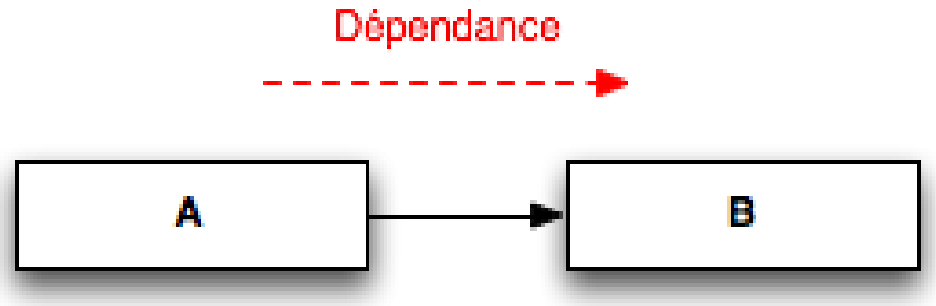
Exemple

- La classe `DataImporter` est dépendante du chargeur de fichier et de la passerelle de stockage.
  - Quel est le problème ?
    - On ne peut pas réutiliser la classe d'importation sans réutiliser le chargeur de fichier CVS et la passerelle de stockage des données.

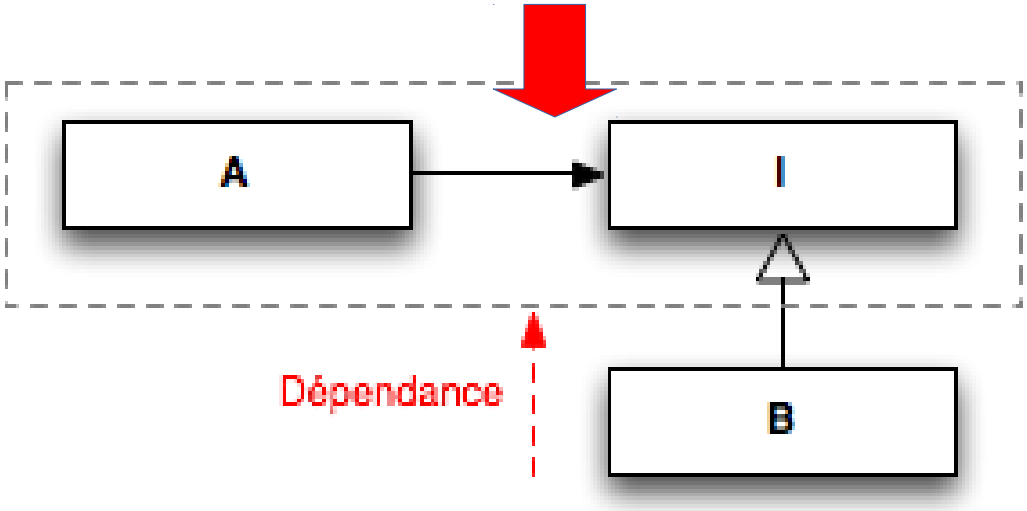


L'abstraction comme technique d'inversion des dépendances

- Relation conventionnelle

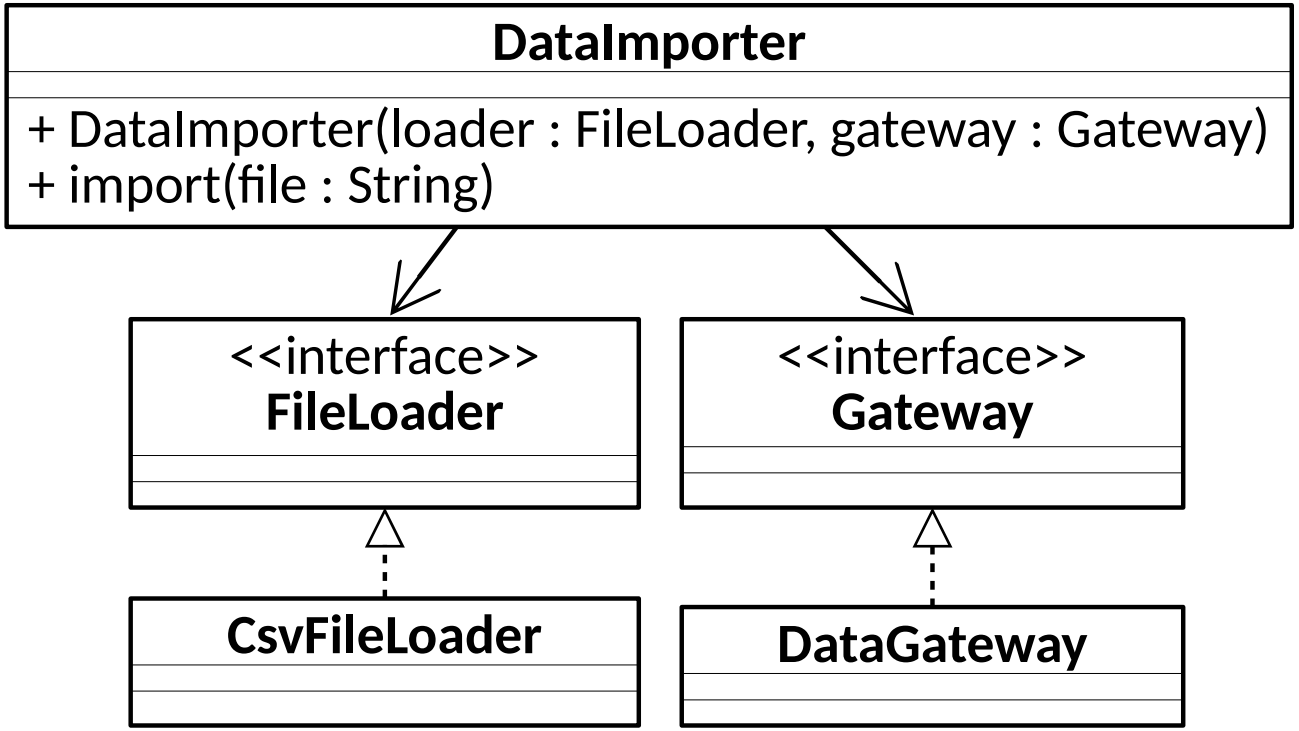


- Inversion de la dépendance : Les modules de bas niveau doit être conformes aux interfaces définies par des modules de haut niveau.



Exemple (refactoring)

- Comment rendre `DataImporter` indépendant des implémentations du chargeur du fichier CSV et de la passerelle ?
  - Inverser les dépendances avec des interfaces.



Loi de Déméter

- **Un objet A ne doit pas utiliser un objet B pour accéder à un troisième objet C pour requérir ses services**.
  - Faire cela signifierait que A a une connaissance plus grande que nécessaire de la structure interne de B (ie, que B est composé de C).
  - Au lieu de cela, B pourrait être modifié si nécessaire pour que A puisse faire la requête directement à B, et B propagera la requête au composant ou sous-composant approprié. La classe B a la connaissance de C.
- Avantage
  - Accroître la maintenabilité et la robustesse du logiciel (Basili et al. 1996).
- Désavantage
  - Requière un grand nombre de petites méthodes « wrapper » dans B pour propager les appels des méthodes de C.

II. Règles de conception

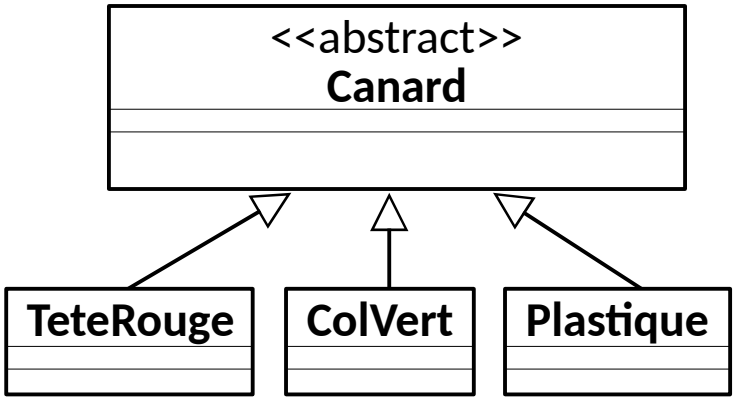
- 4 règles
  - Règle 1. Réduire l'accessibilité des membres de classe.
  - Règle 2. Encapsuler ce qui varie.
  - Règle 3. Programmer pour une interface et non pour une implémentation.
  - Règle 4. Privilégier la composition à l'héritage.

Règle 1. Réduire l'accessibilité des membres de classe

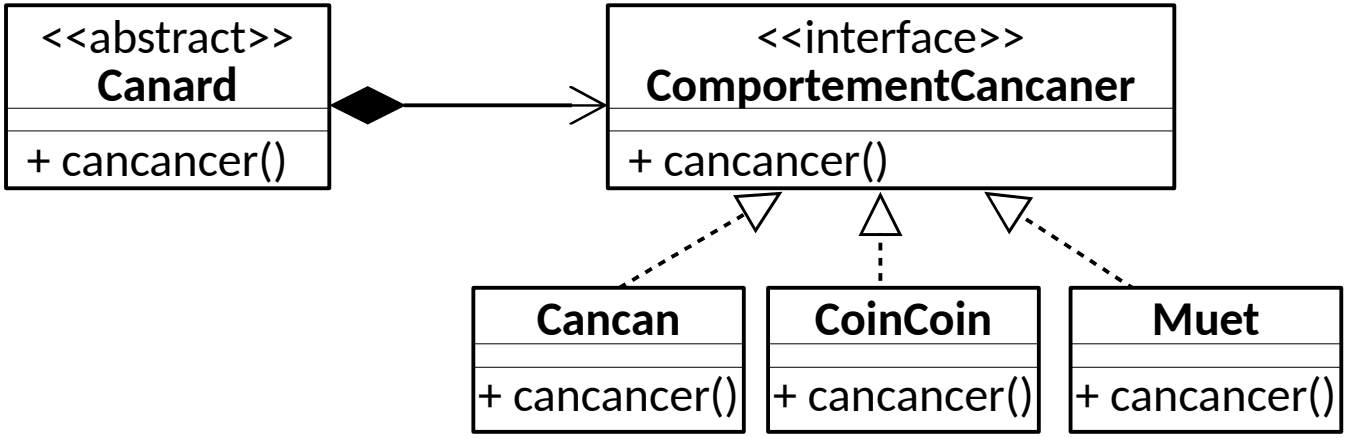
- **L'accès direct aux données membres d'une classe devrait être limité à la classe elle-même.**
  - Éviter d'exposer les détails d'implémentation pour faciliter l'évolution future sans aucune conséquence sur la classe.
- Solution : encapsulation.
  - Faire des attributs privés.
  - Réduire l'utilisation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

Règle 2. Encapsuler ce qui varie

- **Identifier ce qui devrait être variable dans une conception puis encapsuler ce qui varie dans une hiérarchie propre.**
  - Variation sur un concept :



  - Variation sur une méthode :



  - Remarque : il faut mettre une classe « abstraite » à la base de la hiérarchie.

Règle 3. Programmer pour une interface et non pour une implémentation

- **Il faut programmer avec des supertypes (interfaces ou classes abstraites) au lieu d'instances. Les supertypes sont faciles à modifier alors que leurs implémentations sont difficiles à changer.**
  - Programmer pour une implémentation :

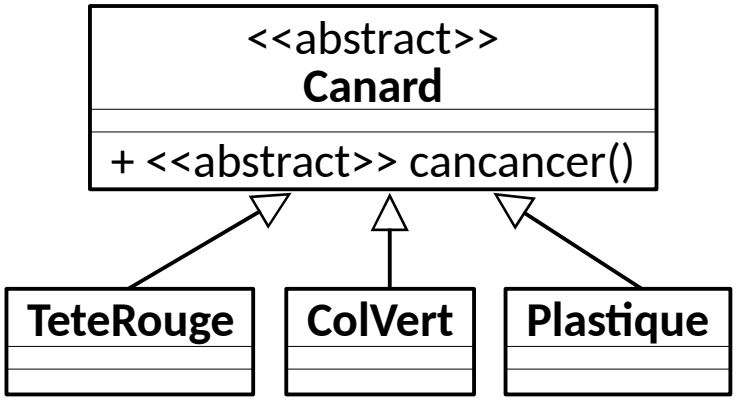
```
ColVert c = new Colvert();
c.cancane();
```

  - Programmer pour une interface :

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

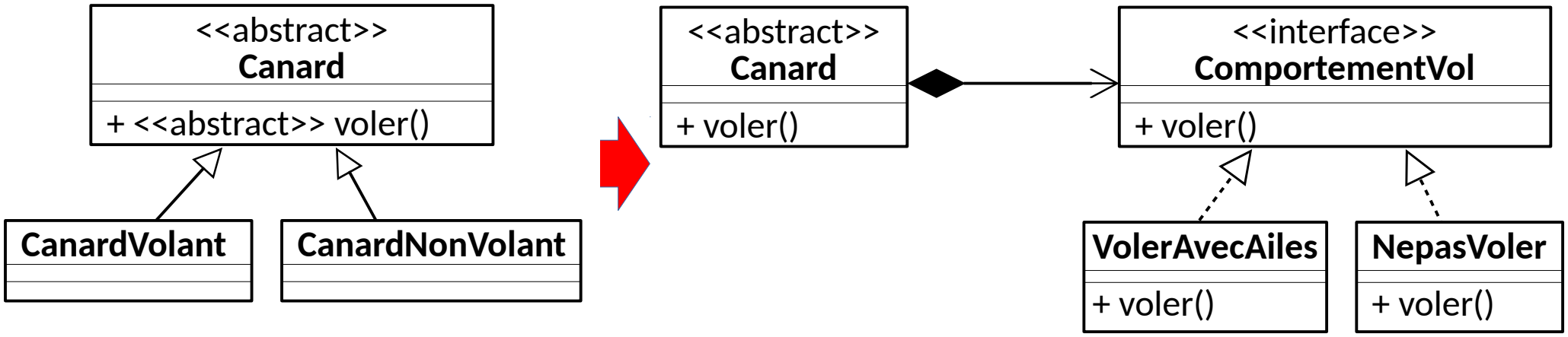
  - Ce qui permet des évolutions sans rien changer par ailleurs :

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



Règle 4. Privilégier la composition à l'héritage

- **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 (*création boîte blanche*).
  - La composition est définie dynamiquement (*création boîte noire*).

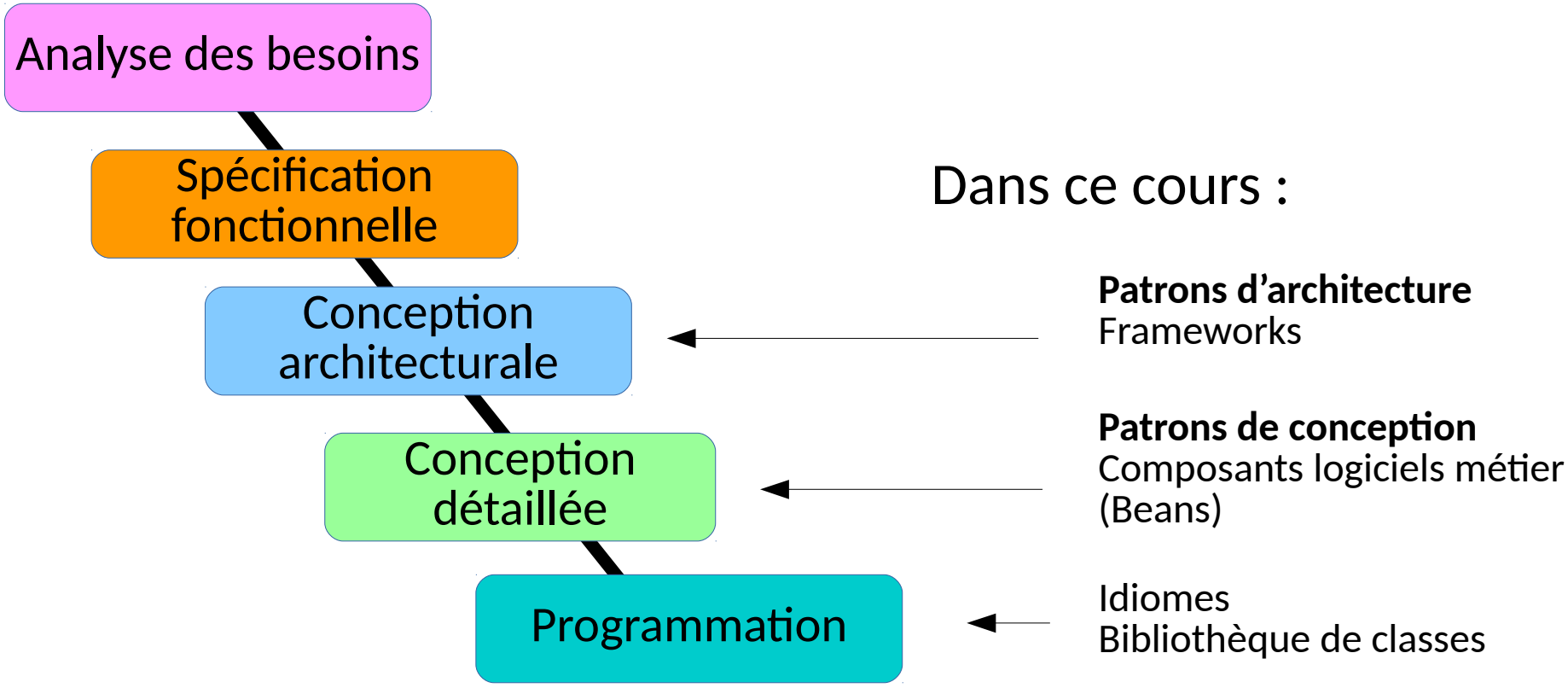


Les règles de Peter Coad

- Utiliser l'héritage quand tous les critères suivants sont satisfaits :
  - La sous-classe représente un « type spécial » de la super-classe et non un « rôle joué » par la super-classe.
  - Une instance de la sous-classe ne doit jamais devenir un objet d'une autre sous-classe.
  - La sous-classe étend plutôt que redéfinie ou annule les responsabilités de la super-classe.
  - La classe de base n'est pas une classe utilitaire qui détient des fonctionnalités qui sont simplement réutilisées dans la sous-classe.

III. Patrons de conception

- **Réutilisabilité**
  - Une grande partie de l'activité de développement de logiciels se fait à partir à partir de savoir-faire récurrents.
  - Nous procédons par recopie, imitation et réutilisation de solutions qui ont fait leurs preuves d'efficacité.
  - La réutilisation concerne tous les niveaux du développement.



Idiome

- Construction récurrente dans un langage de programmation particulier.
  - Non transposable dans un autre langage de programmation.
  - Les idiomes sont décrits dans les manuels de programmation avancée.
- Exemple : parcours d'une chaîne de caractères
  - En C

```
void handleCString( const char * s) {
   for (;*s ;) {  // fin de la chaîne avec '\0'
      function(*s++);
   }
}
```

  - En Java

```
void handleJavaString( String s ) {
   s.chars().forEach(c -> function(c)); // stream
}
```

Bibliothèque de classes

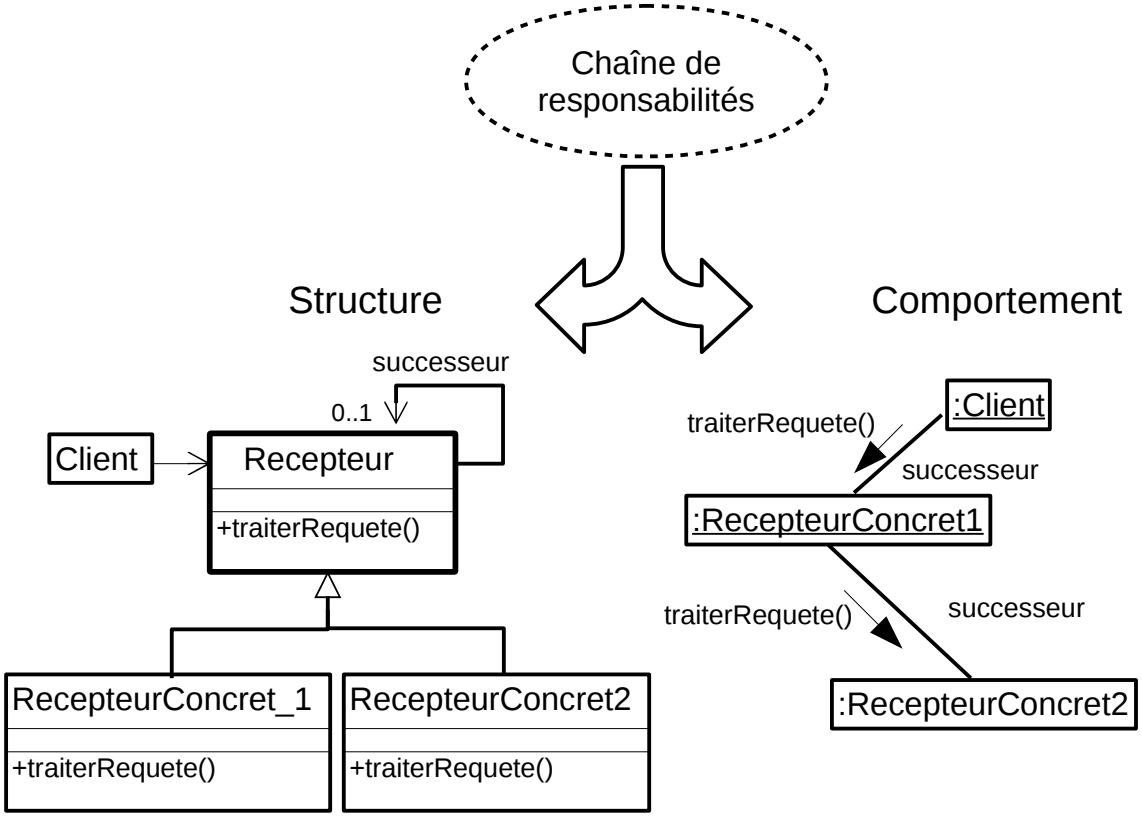
- Collection de classes et de fonctions.
  - Souvent des classes concrètes.
  - Classes connexes mais indépendantes.
  - Sans comportement par défaut.
  - Ne prescrit pas de méthode de conception spécifique.
  - Spécifique d'un langage de programmation.
- Exemples
  - X11, JavaFX, Qt
  - C++ STL, Boost
  - Java SE, Guava

Framework

- Ensemble de classes qui composent un modèle d'application.
  - Classes abstraites et concrètes.
  - Définies pour être utilisées ensemble.
  - Fournissant un comportement par défaut.
- Framework boîte blanche
  - Basé sur l'héritage.
  - Prêt à l'emploi par dérivation de classe.
- Framework boîte noire
  - Basé sur la composition.
  - Prêt à l'emploi par composition des classes entre elles.
- Exemple
  - Java EE (Jakarta EE), .NET, Spring Boot, Struts.

Patron (*Pattern*)

- Une solution conceptuelle pour un problème récurrent.
  - Composants logiques décrits indépendamment de tout langage de programmation.
  - Les patrons sont représentés par des modèles … et des commentaires !
  - Les patrons sont nommés pour être communiqués.



Anti-Patron (Anti-Pattern)

- Un patron qui peut être couramment employé, mais qui est inefficace voire néfaste en pratique.
  - Représente une leçon apprise.
- Deux catégories :
  - Solution d'un problème qui conduit à un échec.
  - Comment se sortir d'une mauvaise situation et continuer à partir de là vers une bonne solution.

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 *refactoring*.
- 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 !