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** : il n'existe pas ou peu de théories
- **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 :
  1. ***Principes de conception*** : notions importantes desquelles dépend la qualité d'une conception
  2. ***Règles de conception*** : ensemble de prescriptions de conception à respecter
  3. ***Patrons de conception*** : modèles de solutions à des problèmes récurrents

Artisans du logiciel

- **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

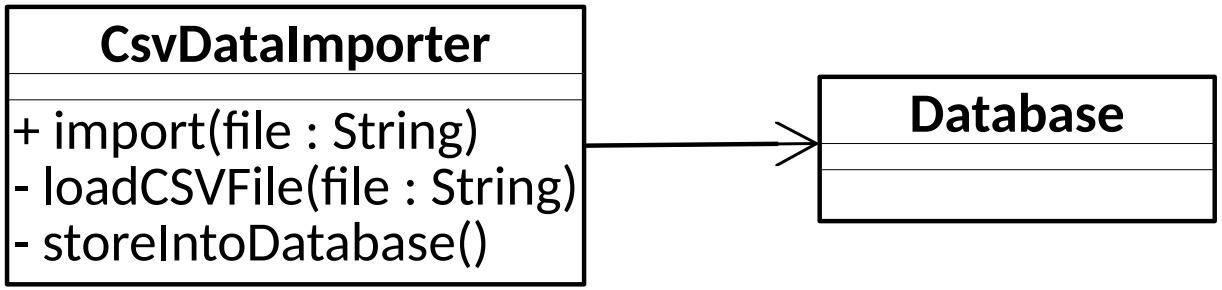
D ependency Inversion Principle
Principes d'inversion des dépendances

Principe 1. Responsabilité unique (Solid)

- **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
- Le but est évidemment d'augmenter la cohésion

Exemple

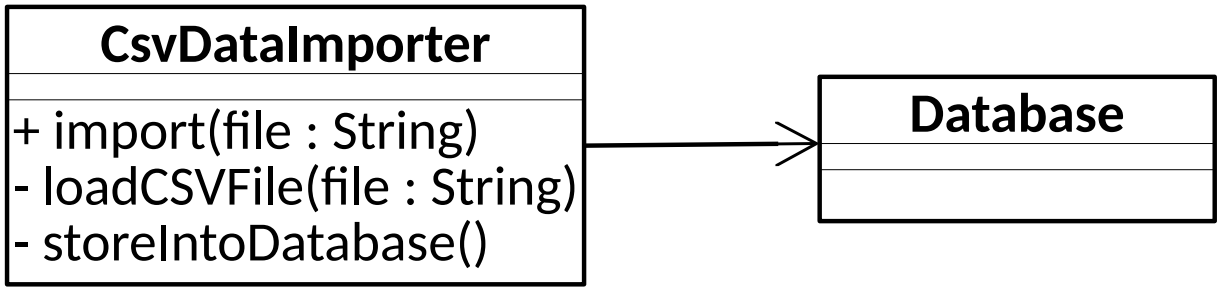
- Importer des données d'un fichier CSV dans une base de données



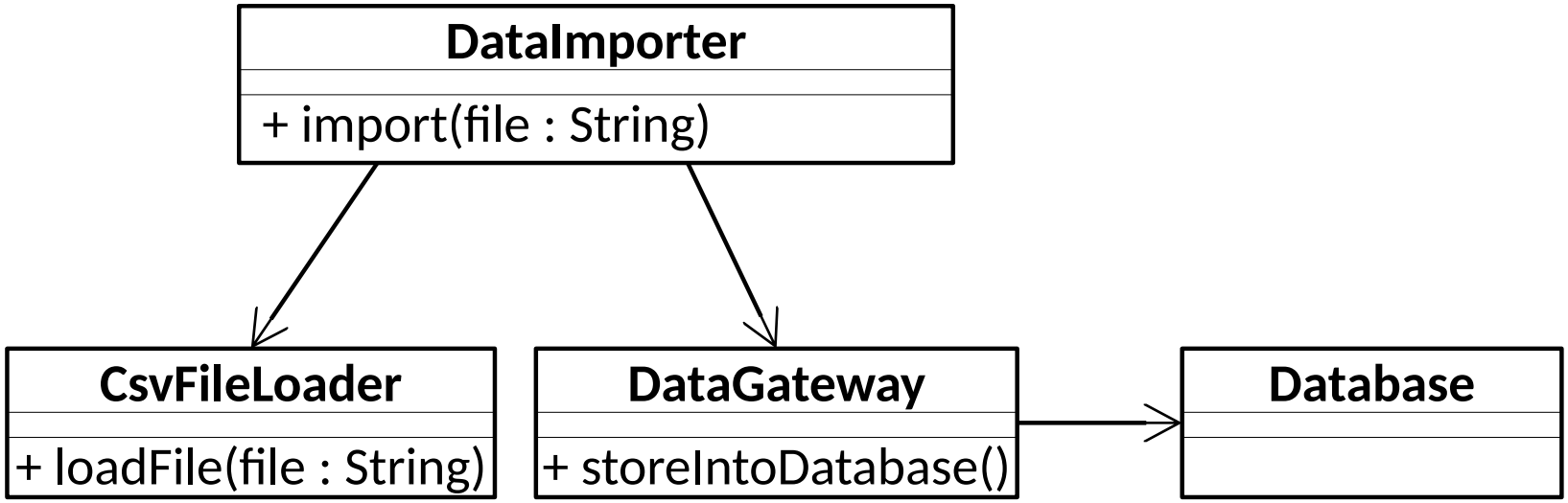
- Quel est le problème avec cette conception ?
  - Il y a 2 responsabilités donc 2 raisons de changer
    1) Le code pour lire le fichier CSV sous forme d'enregistrements
    2) Le code pour stocker les enregistrements dans la base de données

Exemple (refactoring)

- Comment augmenter la cohésion ?



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

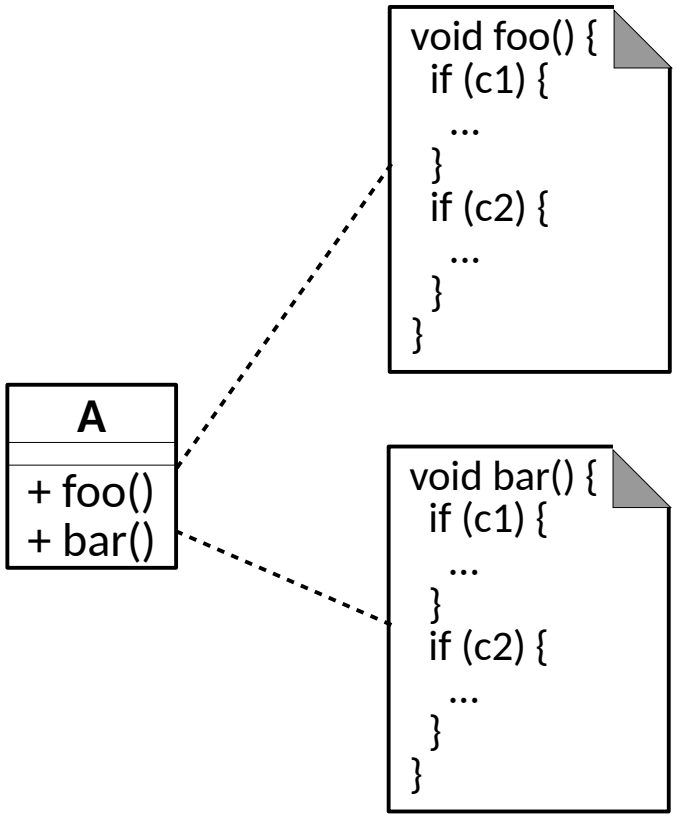


Principe 2. Ouverture / Fermeture (sOlid)

- **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

Exemple

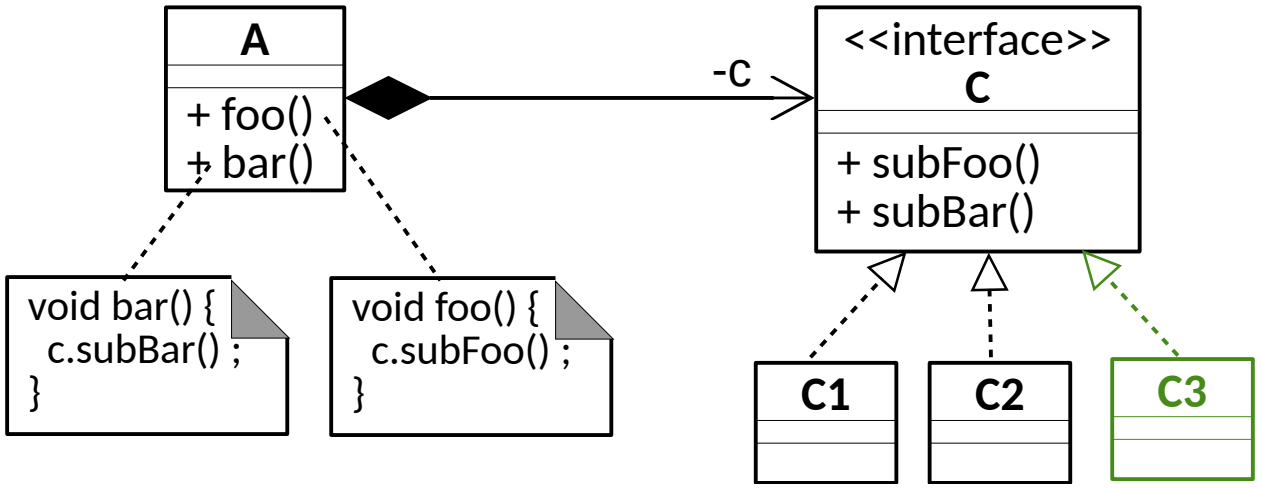
- Considérons la classe A avec deux méthodes qui dépendent des deux attributs c1 et c2



- Quelle est la faiblesse de cette conception ?
  - Si on veut ajouter une variation sur l'attribut c3, il faut modifier le code de foo() et bar() pour y intégrer la variation sur c3

Exemple (refactoring)

- Comment la rendre ouverte aux extensions et fermée aux modifications ?
  - Composition, héritage et polymorphisme

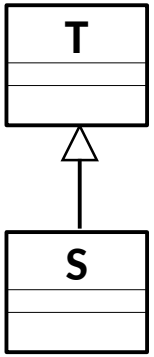


Corollaire : le principe de choix unique

- 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 (soLid)

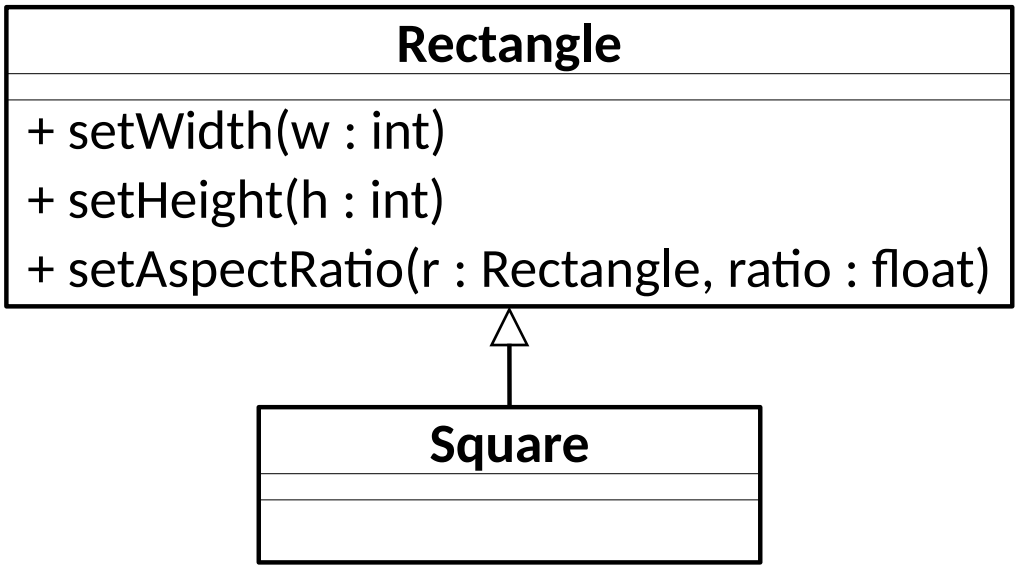
- **Les objets de classes dérivées doivent être substituables aux objets de la classe de base**



- 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

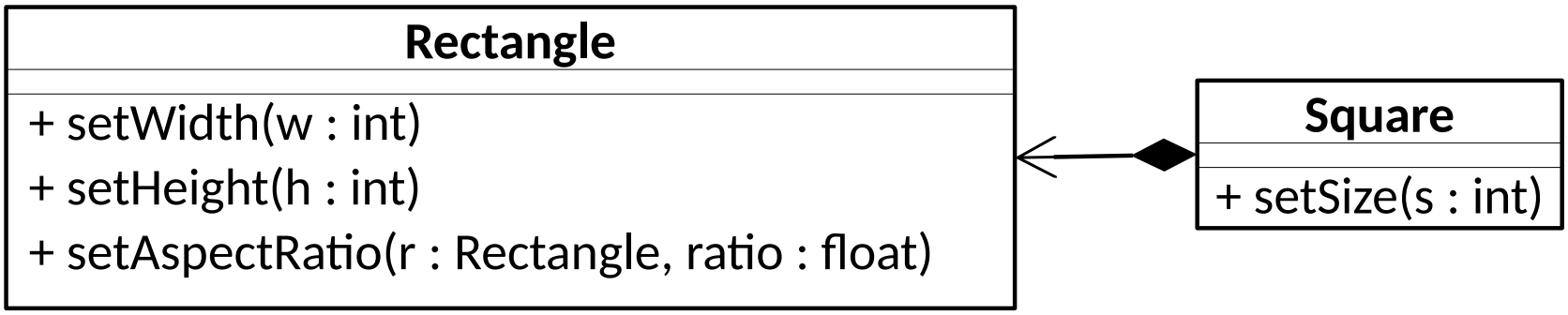
Exemple

- Soit la modélisation suivante : un carré est un rectangle particulier
  - Pourquoi cette modélisation est fausse ?
    - 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



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



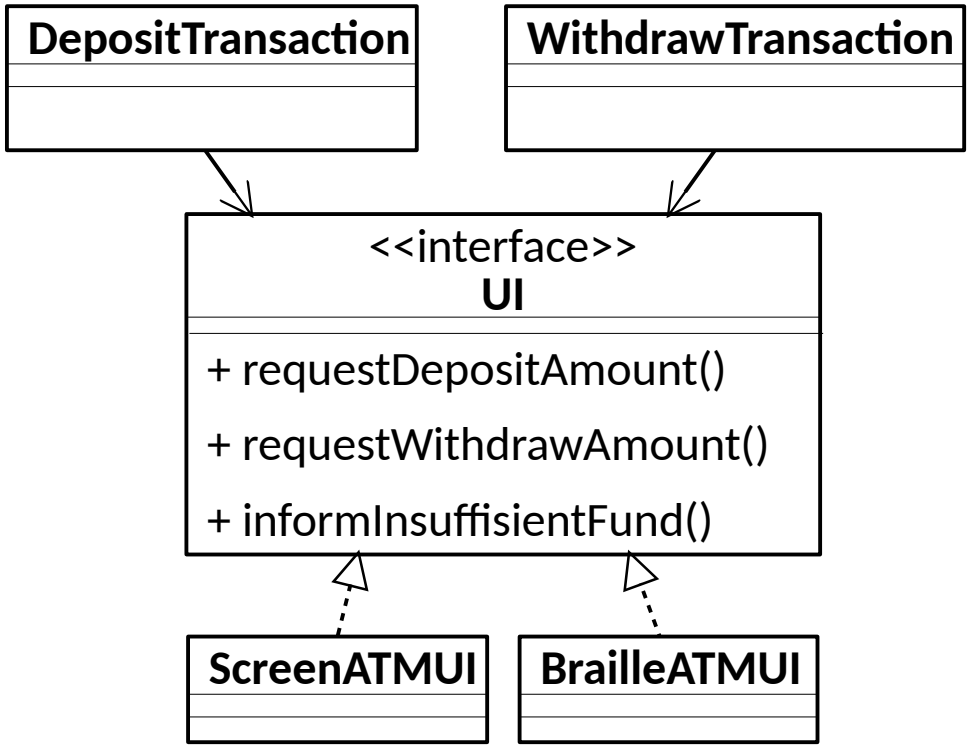
- Cette fois le carré n'est pas substituable au rectangle

Principe 4. Ségrégation d'interface (solId)

- **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

Exemple

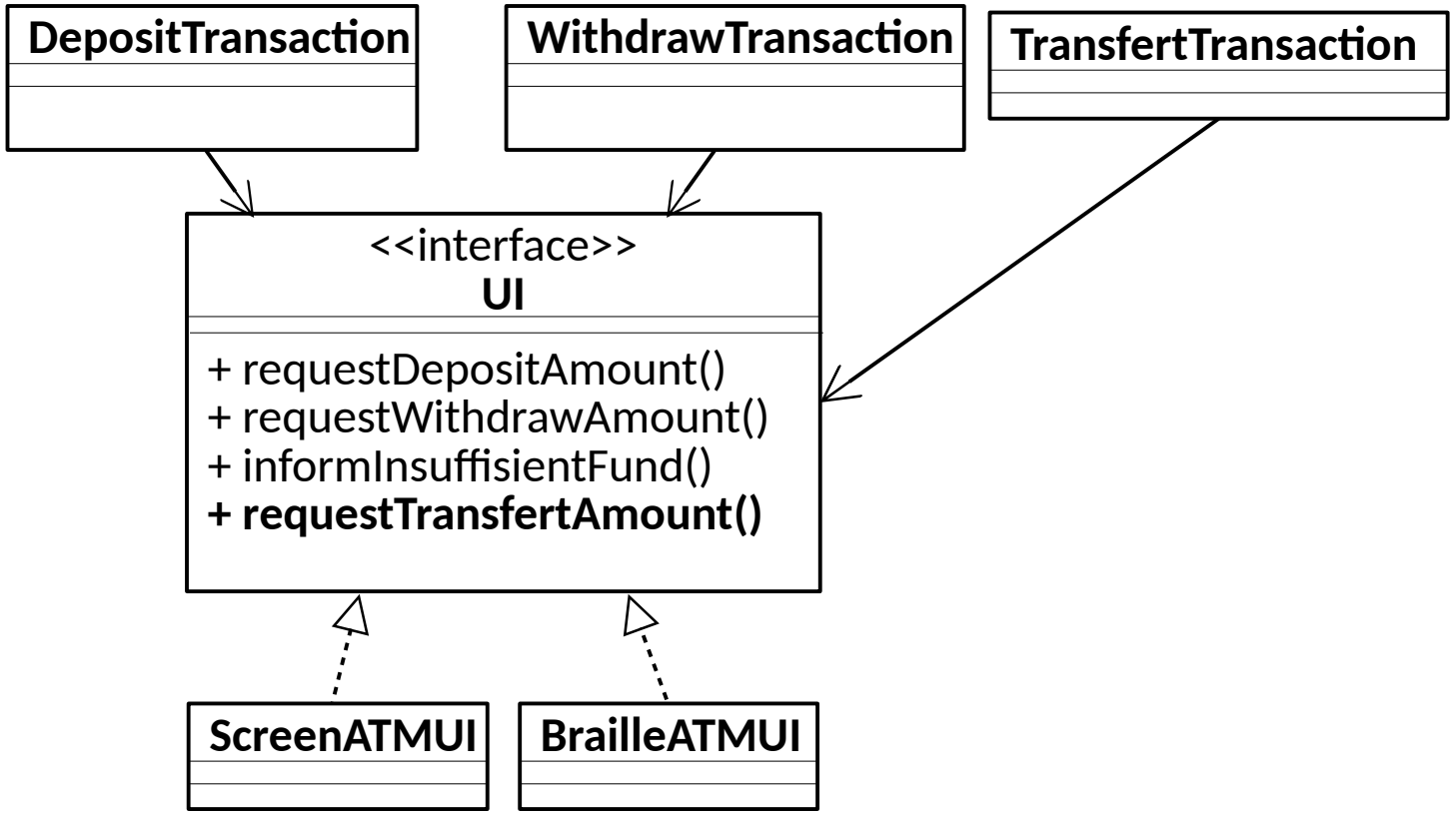
- Guichet Automatique Bancaire (GAB)
  - Toutes les transactions interagissent avec la même interface leur permettant de profiter des méthodes pour gérer l'affichage sur écran ou en braille



  - Quels sont les problèmes potentiels ?
    - La modification d'une méthode impacte toutes les classes dépendantes même si elles n'utilisent pas la méthode

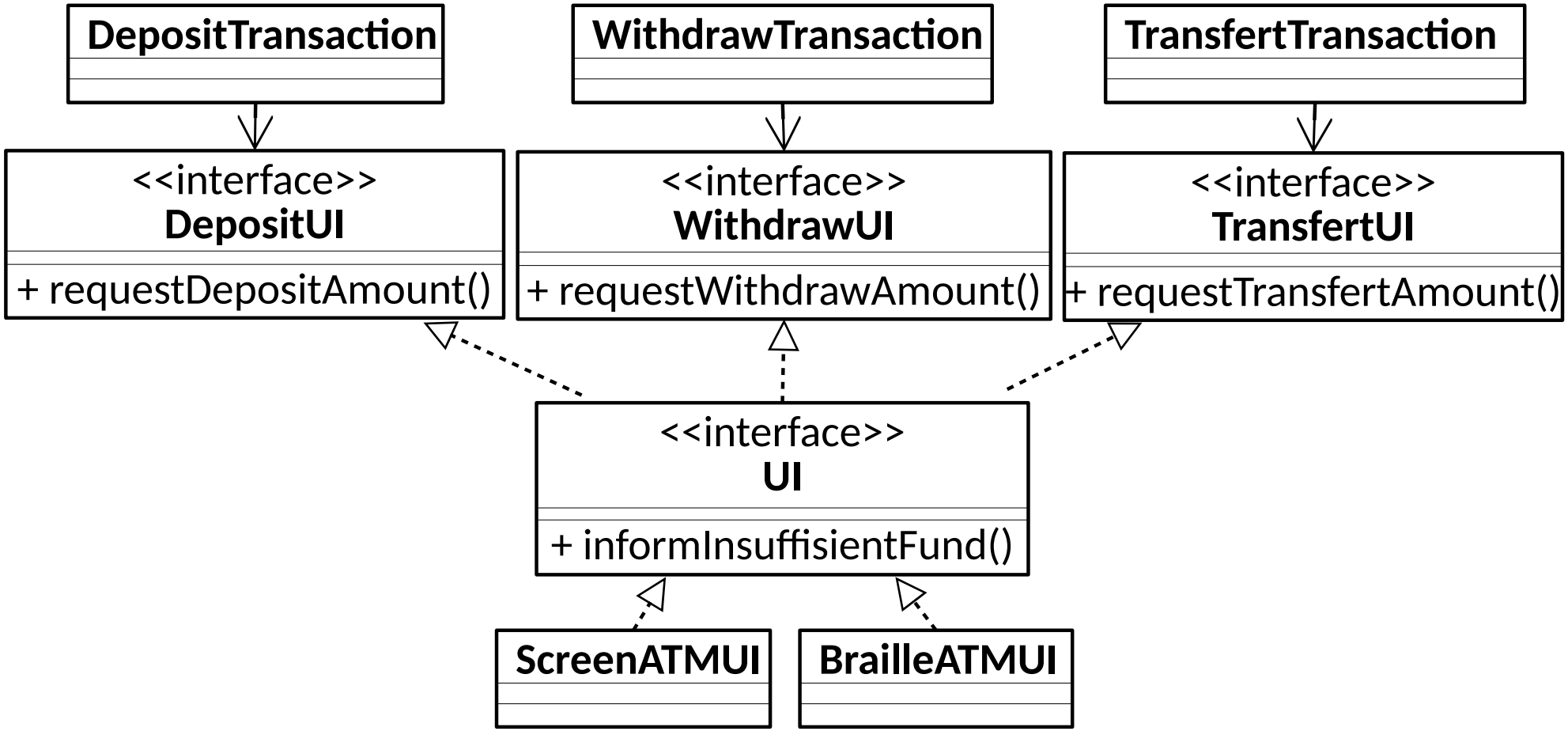
Exemple

- On veut ajouter une nouvelle fonctionnalité de transfert.
  - Les classes clients qui utilisent les interfaces DepositTransaction et WithdrawTransaction se trouvent impactées alors qu'elles n'utilisent pas la nouvelle 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

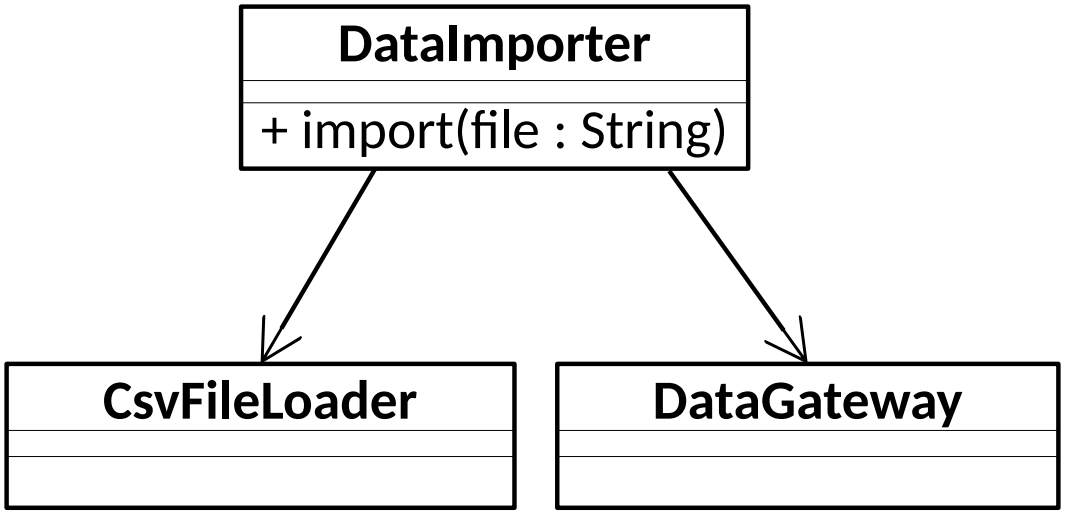


Principe 5. Inversion des dépendances (soliD)

- **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

Exemple

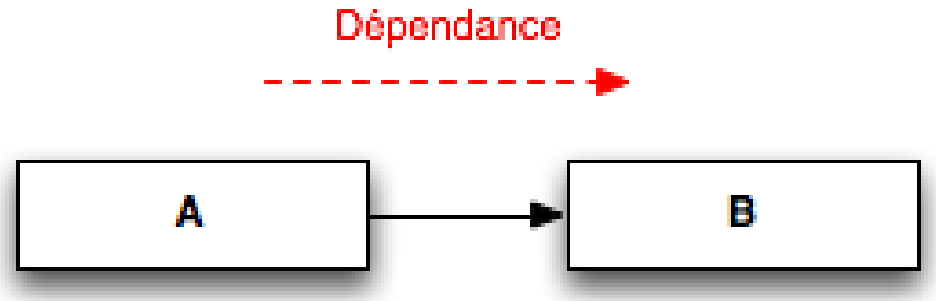
- La classe `DataImporter` est dépendante du chargeur de fichier et de la passerelle de stockage



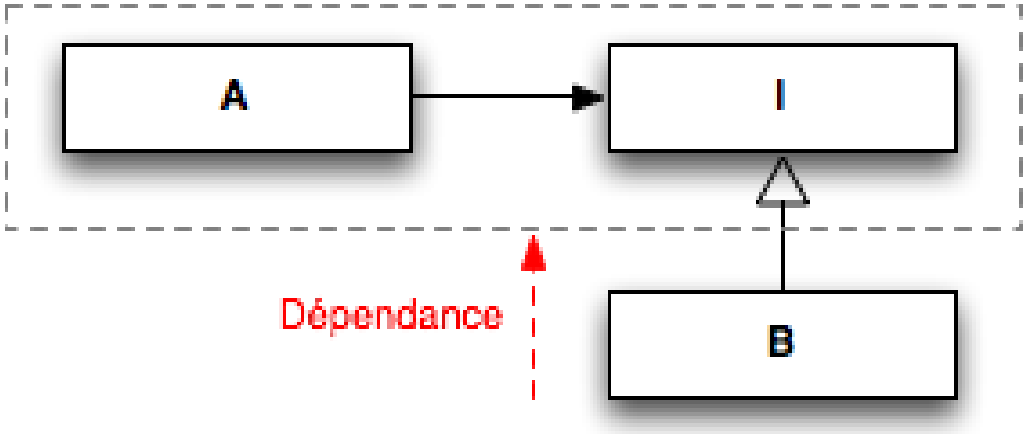
- Quel est le problème ?
  - On ne peut pas réutiliser la classe d'importation sans réutiliser le chargeur de fichier CVS et la passerelle de stockage des données

L'abstraction comme technique d'inversion des dépendances

- Relation conventionnelle

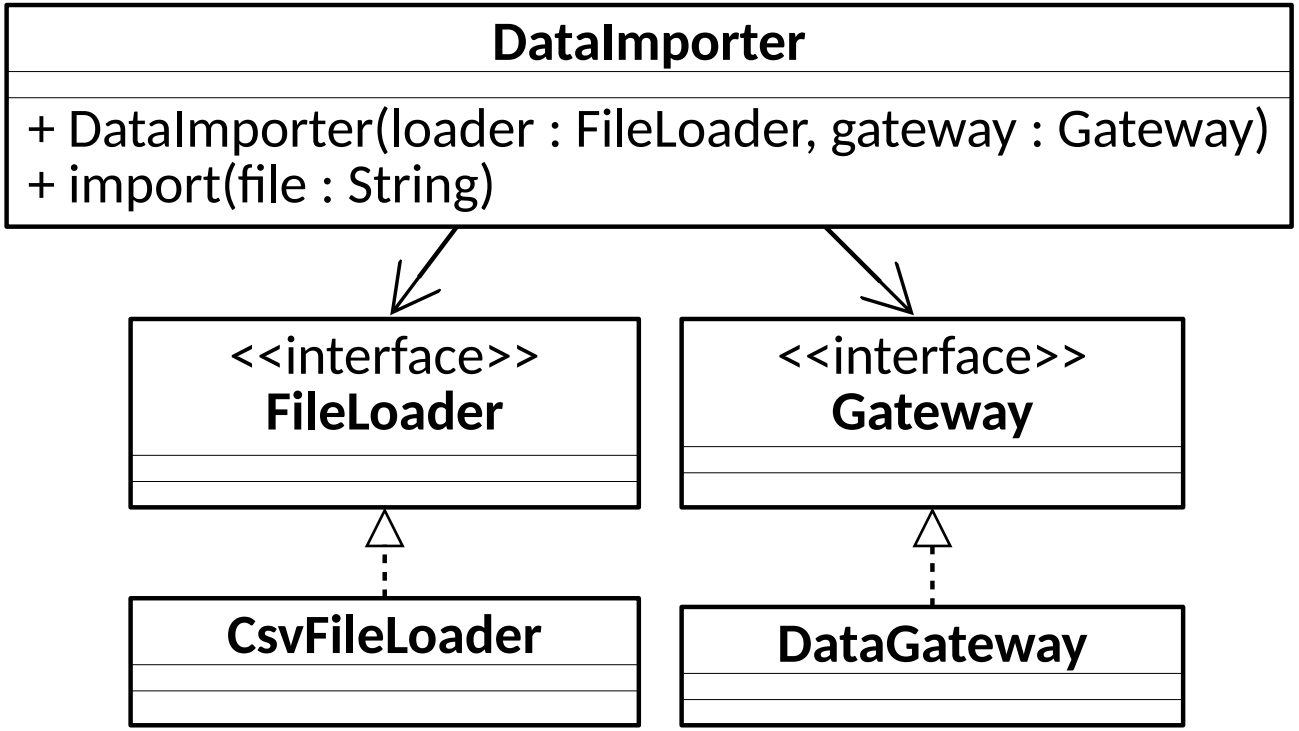


- Inversion de la dépendance par introduction d'une interface entre la classe A et la classe B



Exemple (refactoring)

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



II. Règles de conception

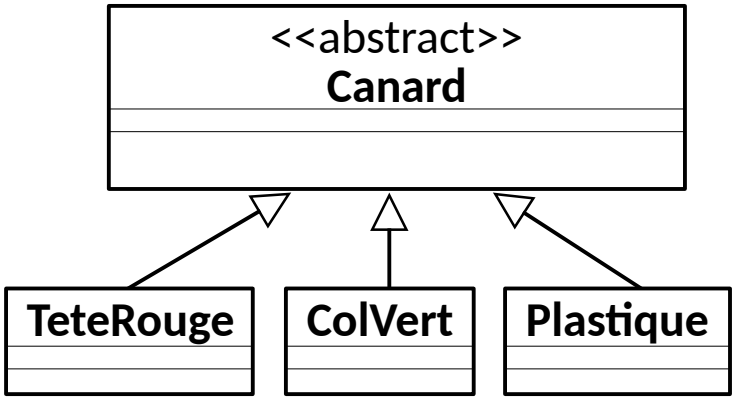
- 4 règles
  - Règle 1. Réduire l'accessibilité des membres de classe
  - Règle 2. Encapsuler ce qui varie
  - Règle 3. Programmer pour une interface, non pour une implémentation
  - Règle 4. Privilégier la composition à l'héritage

Règle 1. Réduire l'accessibilité des membres de classe

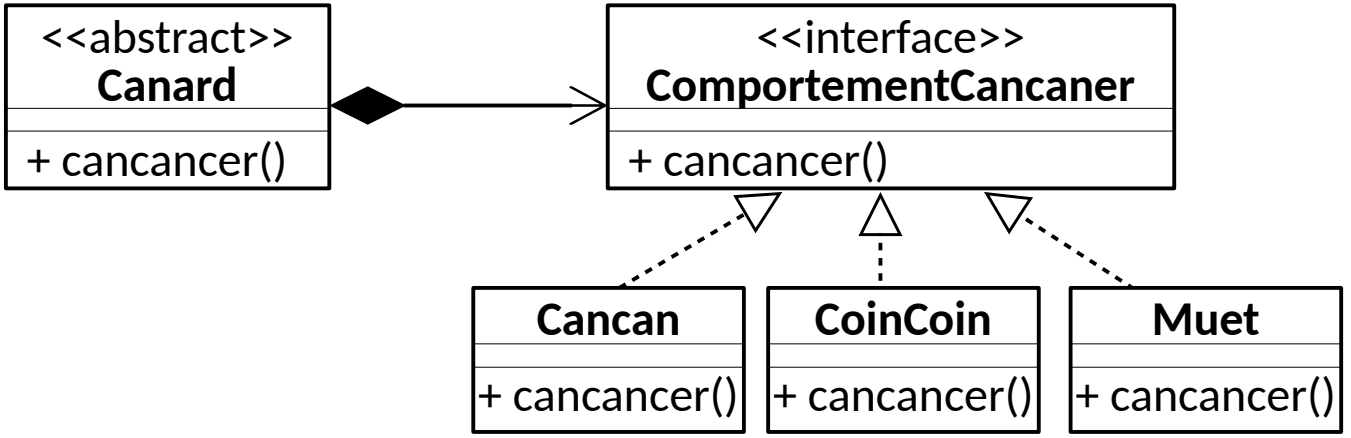
- **L'accès direct aux données membres d'une classe devrait être limité à la classe elle-même**
  - Éviter d'exposer les détails d'implémentation pour faciliter l'évolution future sans aucune conséquence sur la classe
- Solution : encapsulation
  - Faire des attributs privés
  - Réduire l'utilisation des accesseurs (*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

Règle 2. Encapsuler ce qui varie

- **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 :



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 implémentations sont difficiles à changer**
  - Programmer pour une implémentation :

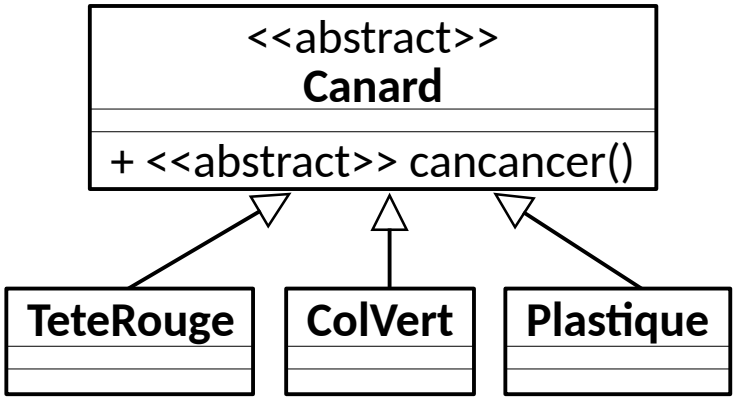
```
ColVert c = new Colvert();
c.cancane();
```

  - Programmer pour une interface :

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

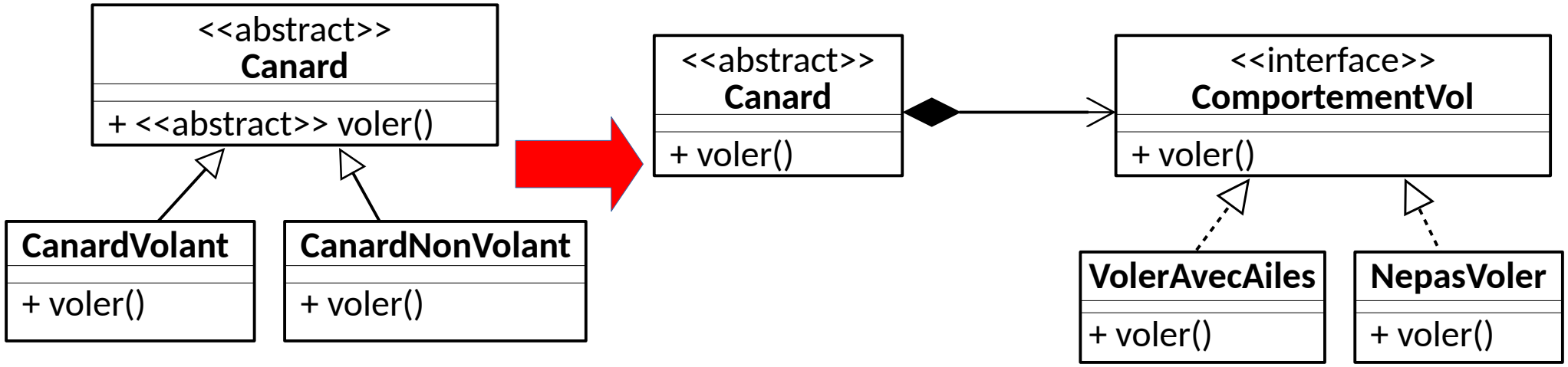
  - Ce qui permet des évolutions sans rien changer par ailleurs :

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



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 (*mécanisme de création de type boîte blanche*)
  - La composition est définie dynamiquement (*création type boîte noire*)

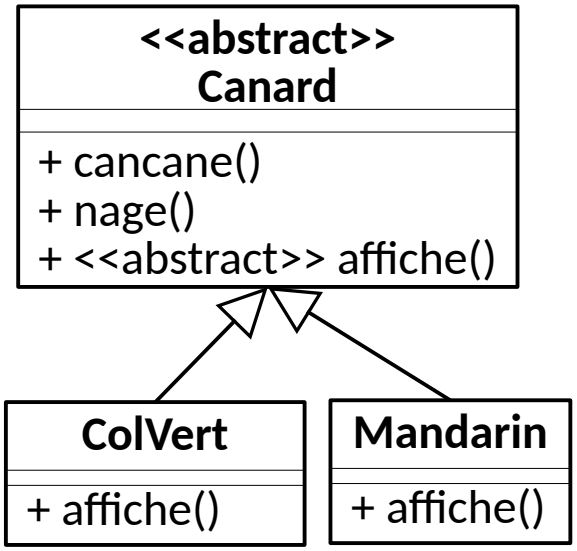


Exemple de l’application des principes et des règles pour la conception

- Un jeu de simulation d'une mare aux canards

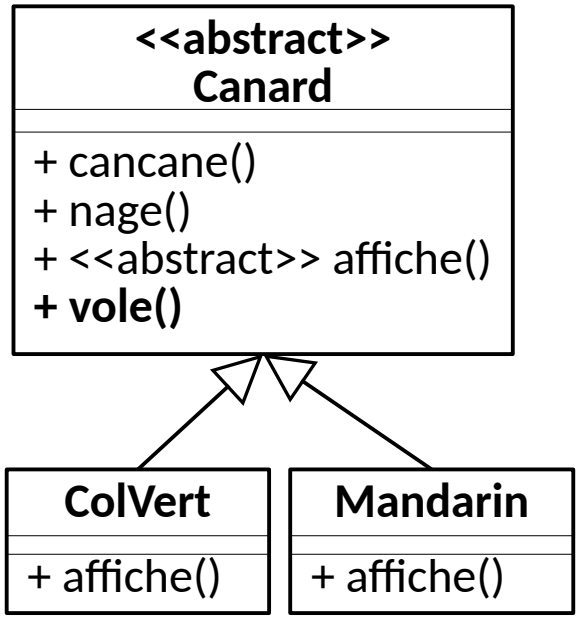
Jeu de simulation d'une mare aux canards

- Solution : représentation une large gamme de Canards nageant et cancanant grâce à une hiérarchie sur les concepts
  - L'affichage est spécifique de chaque espèce



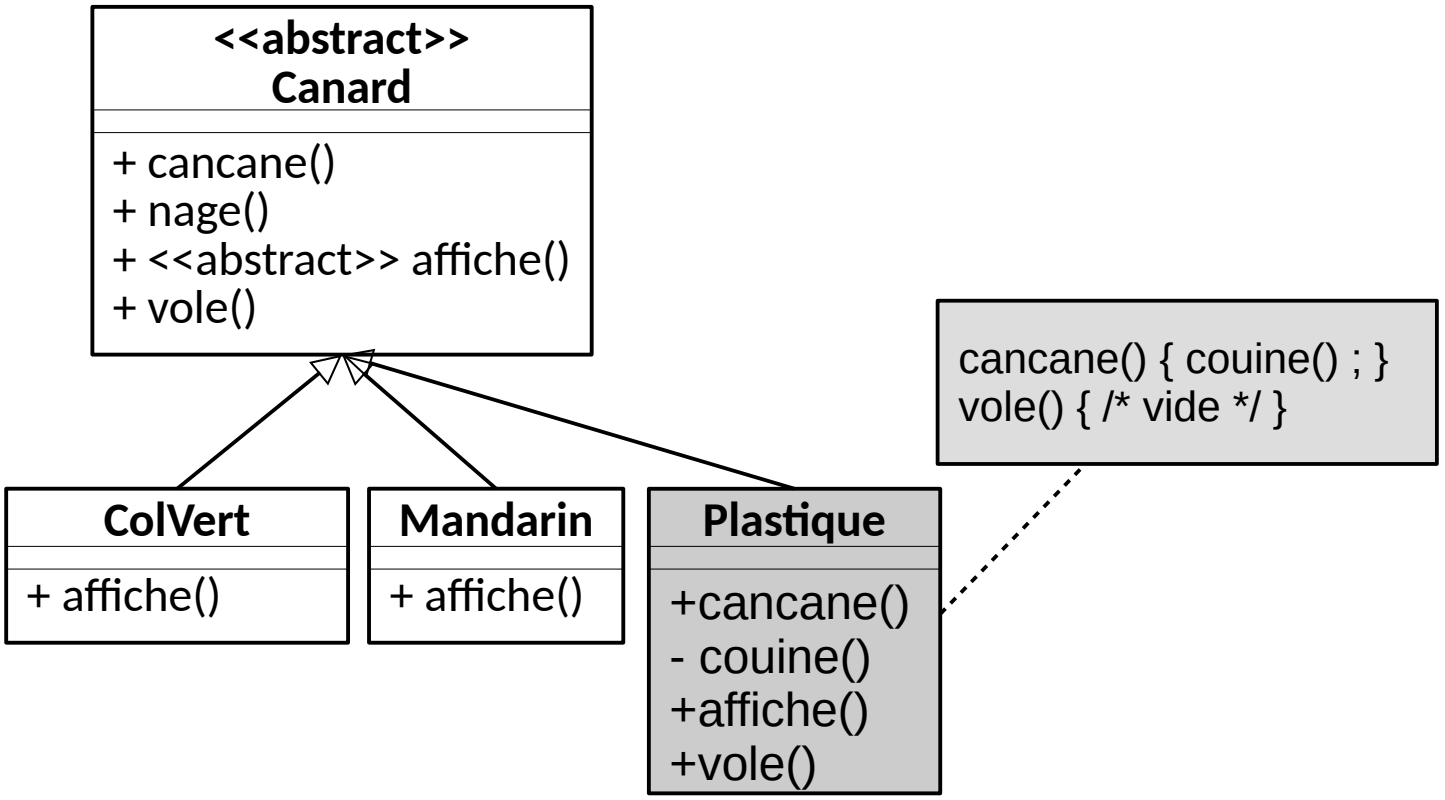
Jeu de simulation d'une mare aux canards

- Nouvelle fonctionnalité : voler



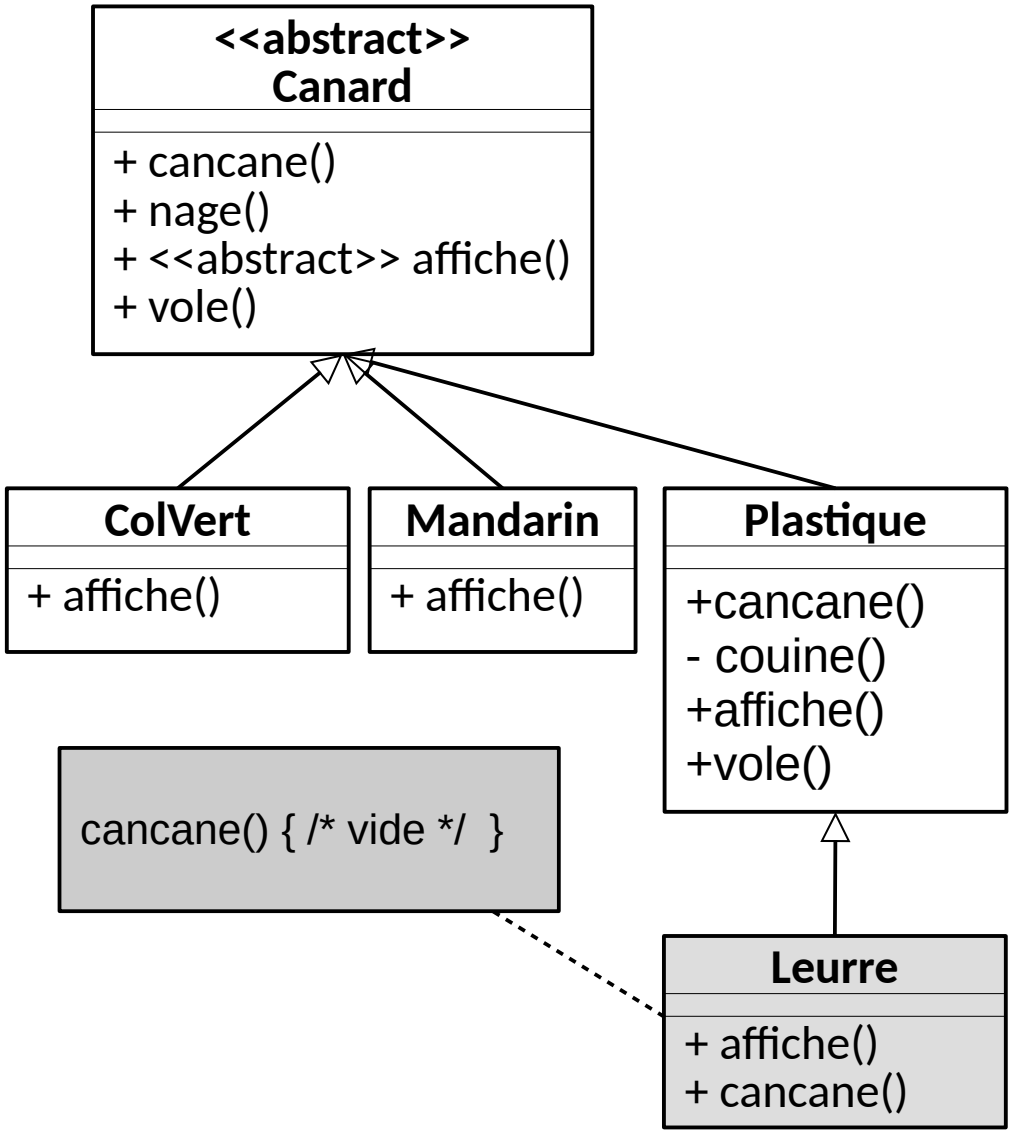
Jeu de simulation d'une mare aux canards

- Nouvelle fonctionnalité : canard en plastique



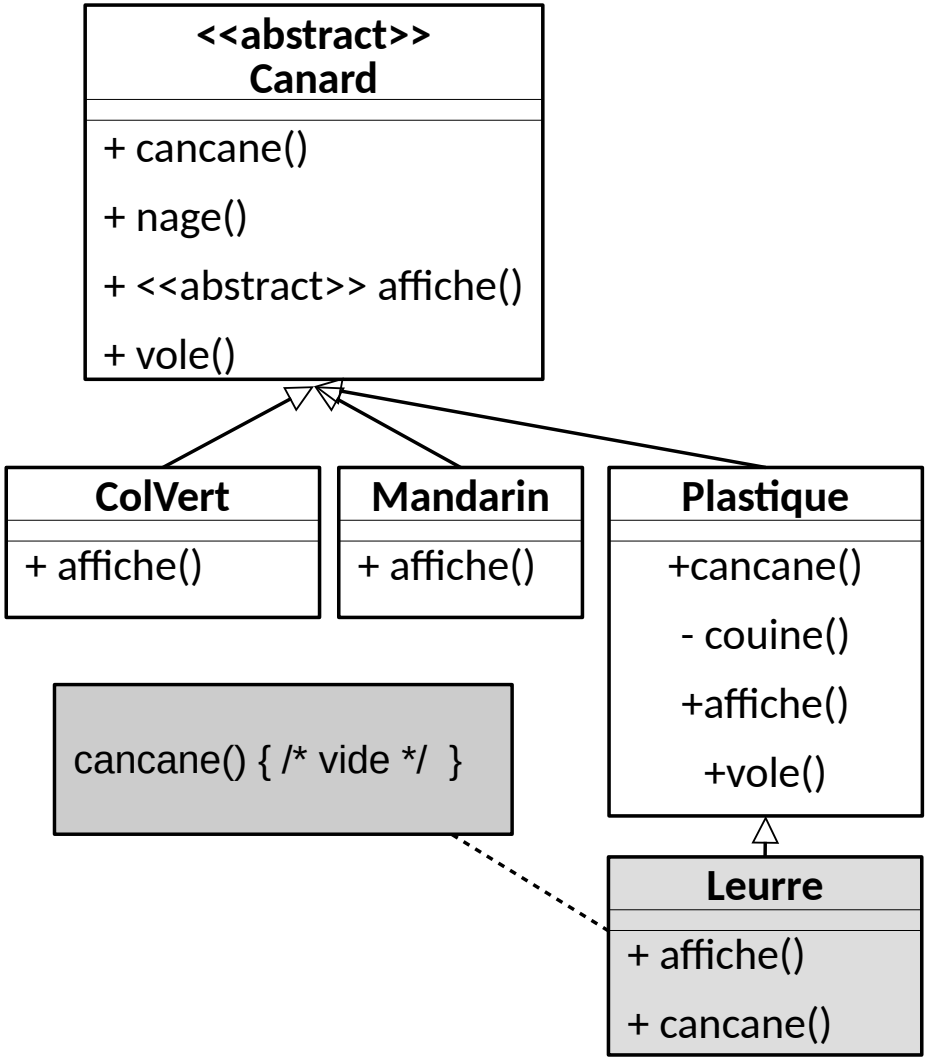
Jeu de simulation d'une mare aux canards

- Nouvelle fonctionnalité : canard leurre



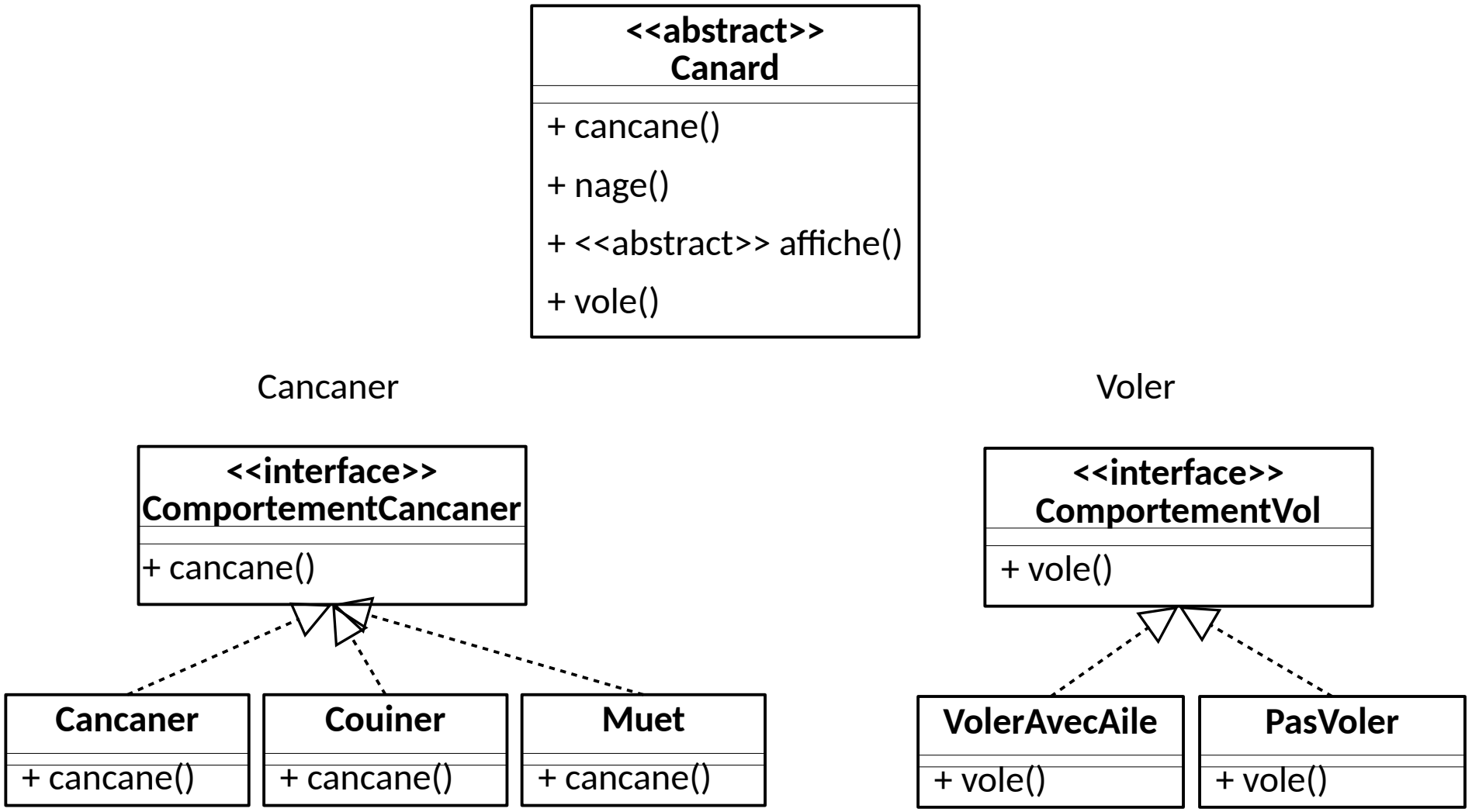
Analyse de la solution courante

- Sentez-vous le pourrissement ?
- Quels principes SOLID sont violés ?



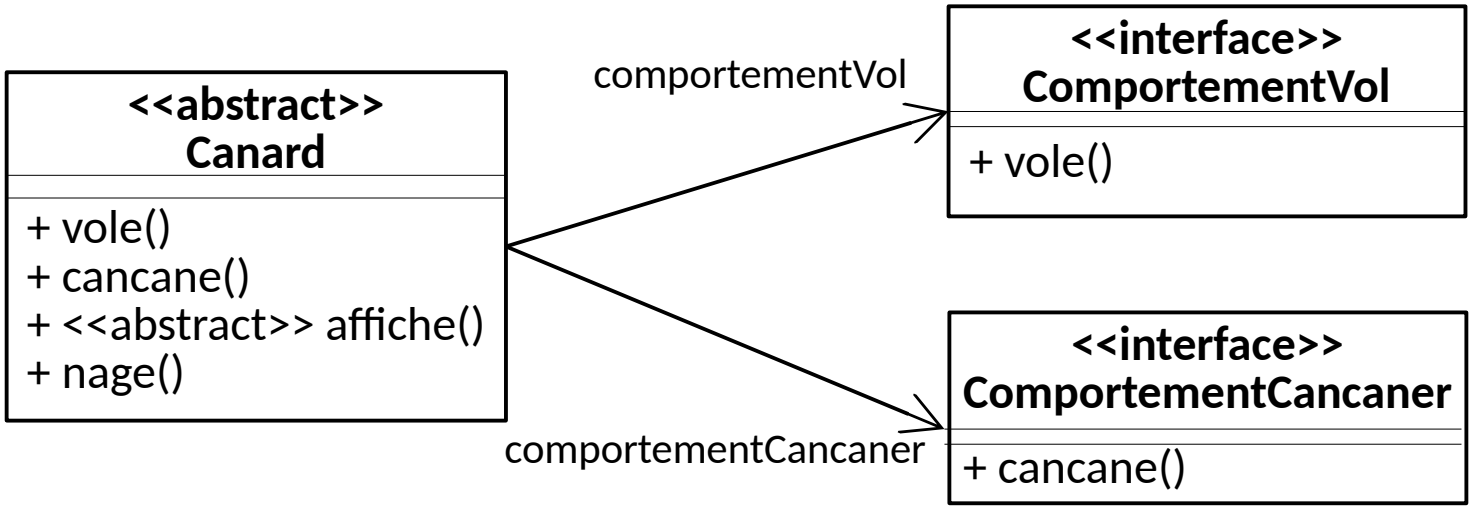
Jeu de simulation d'une mare aux canards

- Solution : appliquer les règles,2,3 et 4 : identifier ce qui varie → encapsuler chaque variation dans une hiérarchies à part



Jeu de simulation d'une mare aux canards

- Relier




```
public void vole() { _comportementVol.vole(); }
public void cancane() { _comportementCancaner.cancane(); }
```

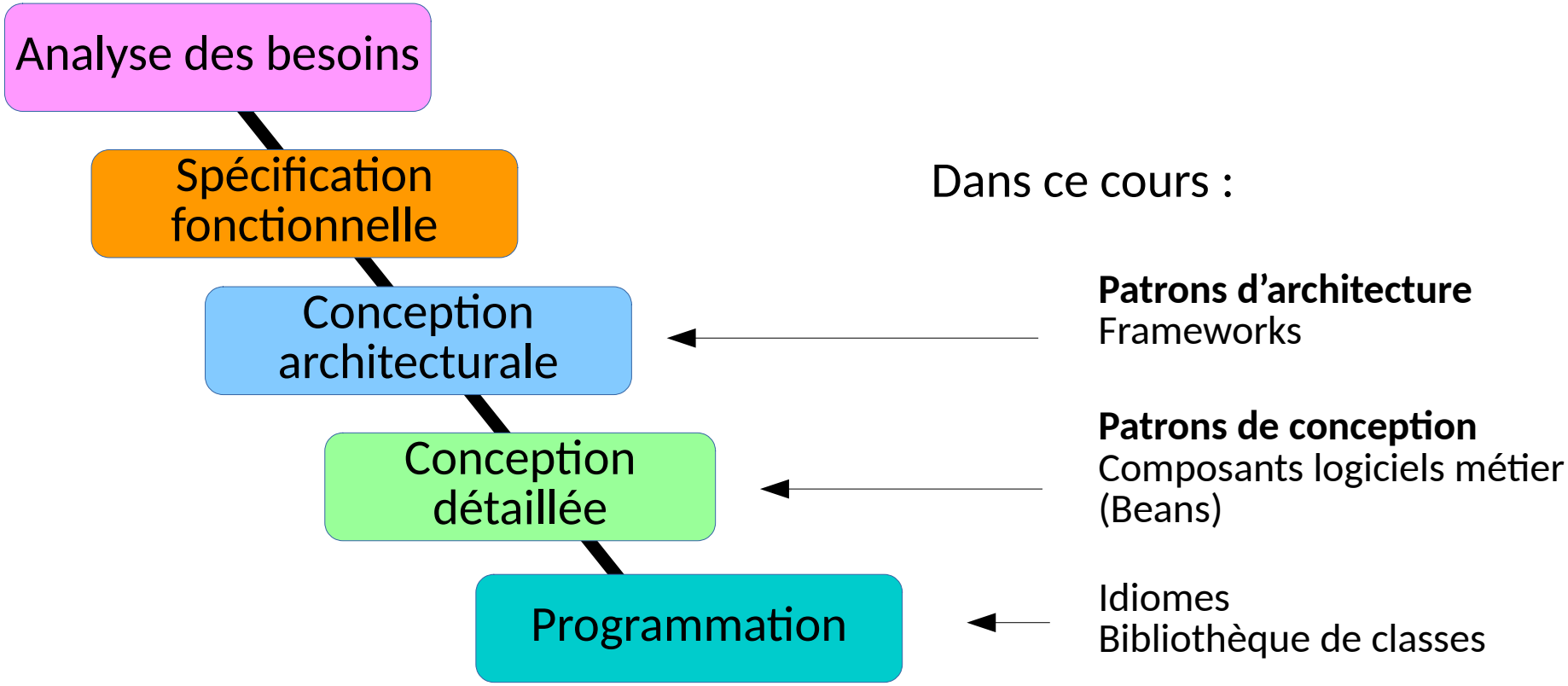
Jeu de simulation d'une mare aux canards

- Comment créer des canards colverts et des mallards ?

```
class Duck {
    private QuackBehavior _quackBehavior;
    private FlyBehavior _flyBehavior;
    protected Duck( QuackBehavior q, FlyBehavior f ) {
        _quackBehavior = q;
        _flyBehavior = f;
    }
    public void performFly() { _flyBehavior.fly(); }
    public void performQuack() { _quackBehavior.quack(); }
}
class MallardDuck extends Duck {
    public MallardDuck() {
        super(new Quack(), new FlyWithWings());
    }
}
class RubberDuck extends Duck {
    public RubberDuck() {
        super(new Squeak(), new FlyNoWay());
    }
}
```

III. Patrons de conception

- **Réutilisabilité**
  - Une grande partie de l'activité de développement de logiciels se fait 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

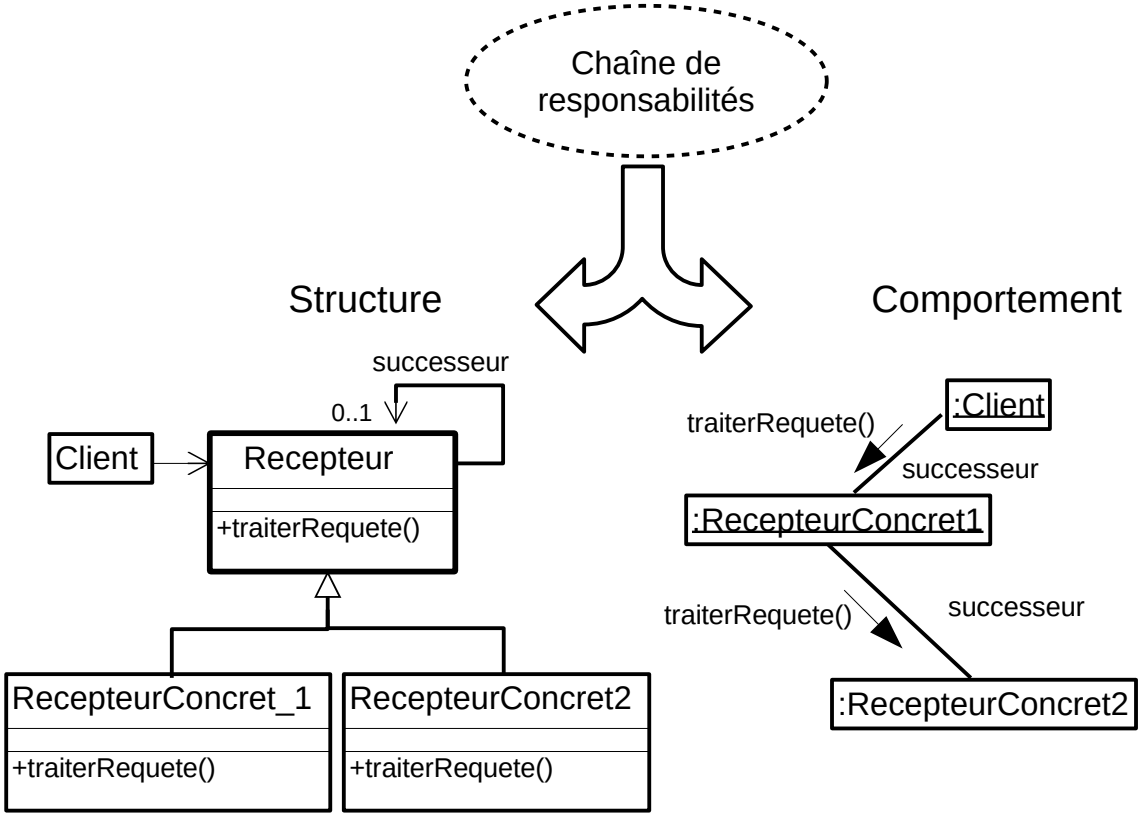
- Collection de classes et de fonctions
  - Souvent des classes concrètes
  - Classes connexes mais indépendantes
  - Sans comportement par défaut
  - Ne prescrit pas de méthode de conception spécifique
  - Spécifique d'un langage de programmation
- Exemples
  - Bibliothèques graphiques : X11, JavaFX, Qt
  - Bibliothèques d’utilitaires : 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
- Exemples
  - Frameworks : 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 *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 !