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

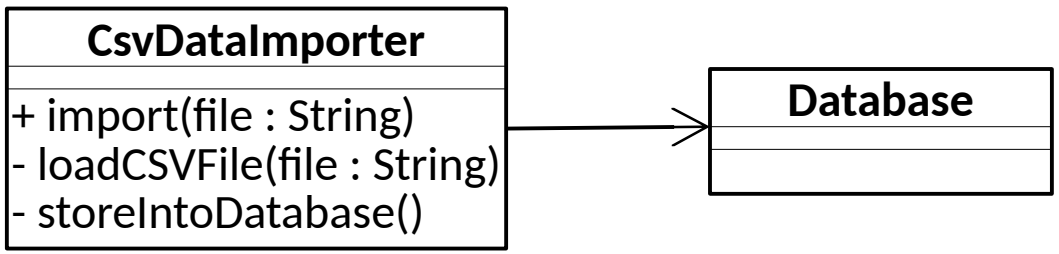
D ependency Inversion Principle

Principe 1. Responsabilité unique ([S]OLID)

- **Un module (fonction, classe, paquet, etc.) devrait n'avoir qu'une responsabilité unique**
  - La responsabilité unique s'entend comme une seule raison de changer
- **Objectif**
  - Augmenter la cohésion

Exemple

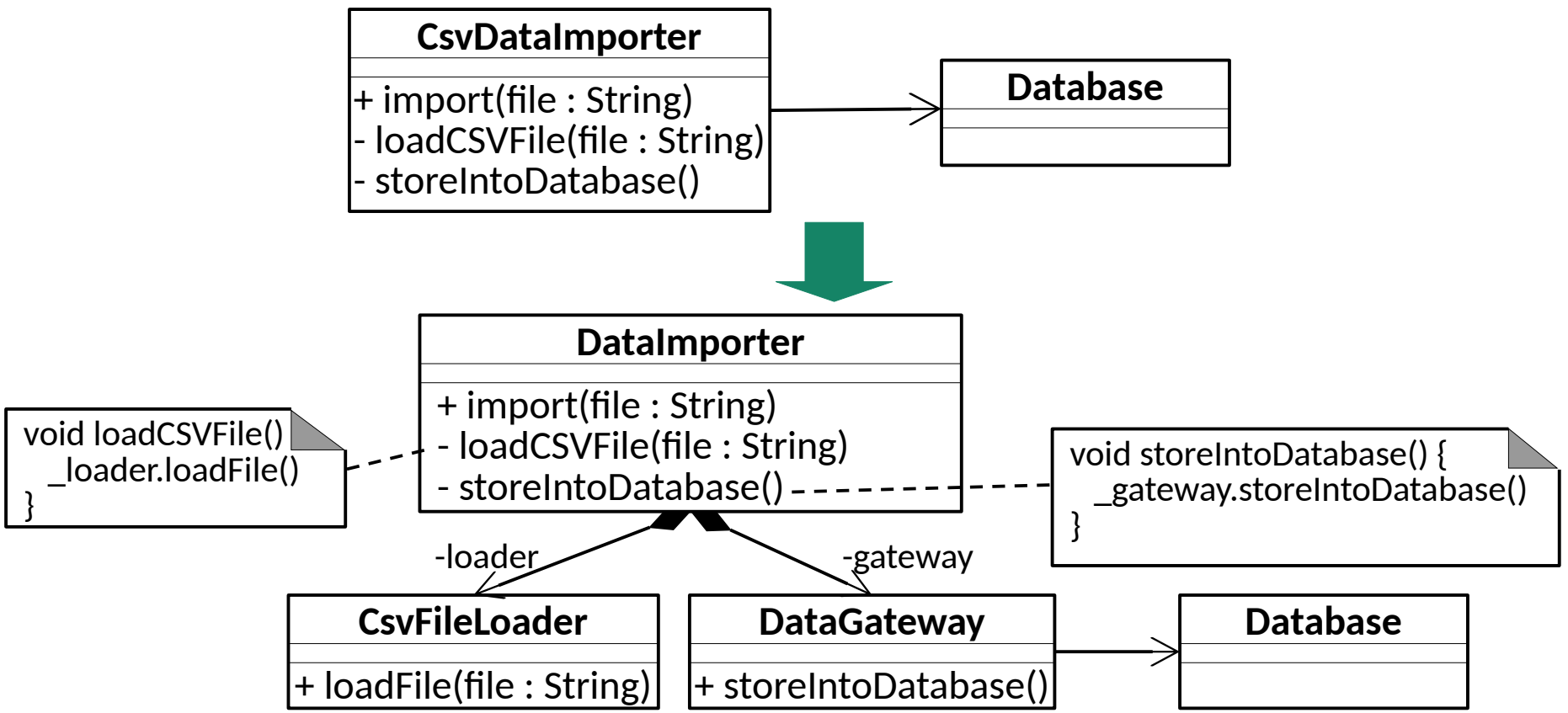
- Importer des données d'un fichier CSV dans une base de données



- Quel est le problème avec cette conception ?
  - Bien qu'il n'y ait qu'un seul service, il y a 2 raisons de changer donc 2 responsabilités :
    1) Le code pour lire le fichier CSV sous forme d'enregistrements
    2) Le code pour stocker les enregistrements dans la base de données
- **Remarque** : le nombre de responsabilité n'est pas lié au nombre de méthodes publiques

Exemple (refactoring)

- Comment augmenter la cohésion ?
  - Modularité et composition
  - Externaliser le code du chargeur de fichier et celui de la passerelle de stockage

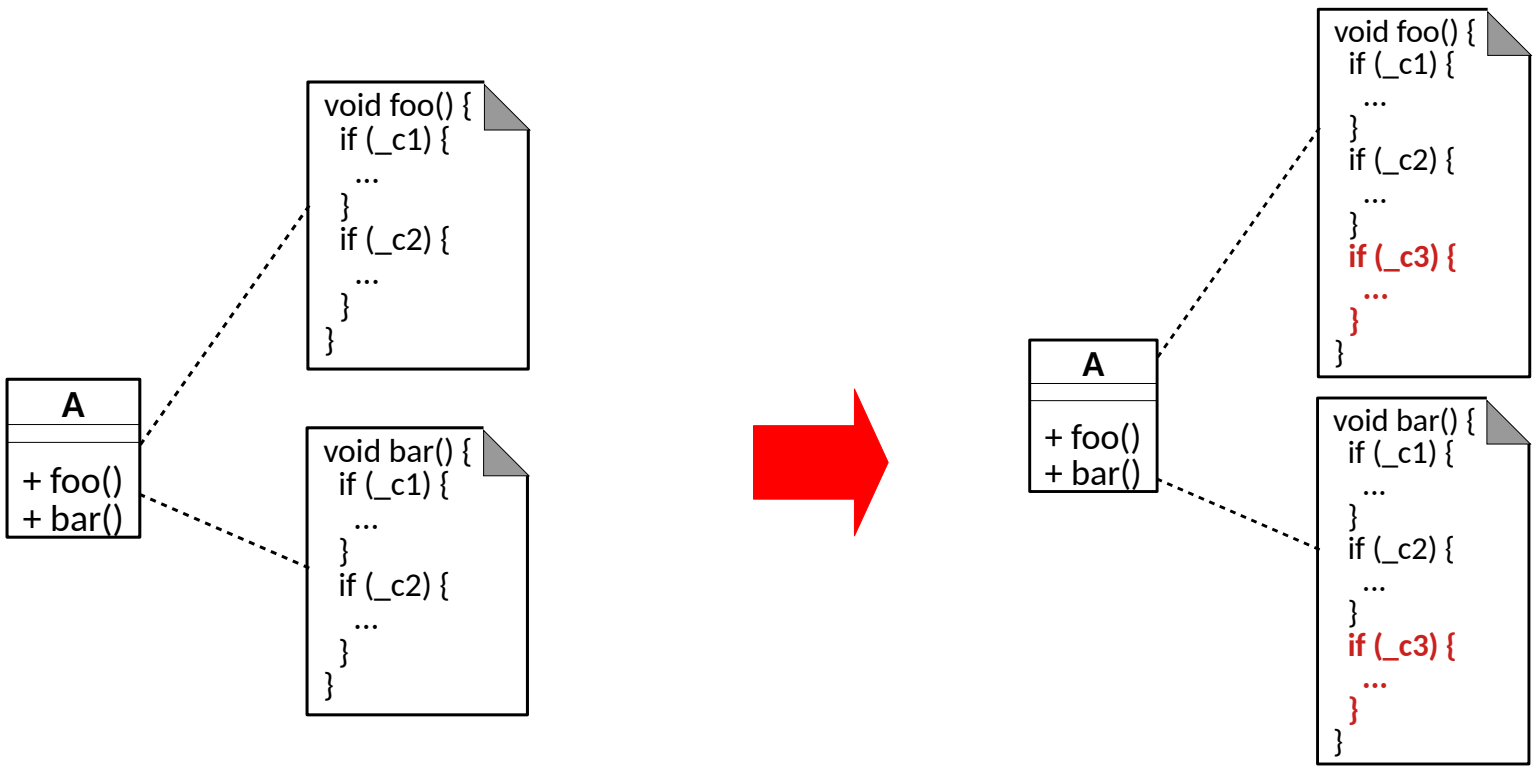


Principe 2. Ouverture / Fermeture (S[O]LID)

- **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
- **Objectif**
  - Éviter la régression logicielle

Exemple

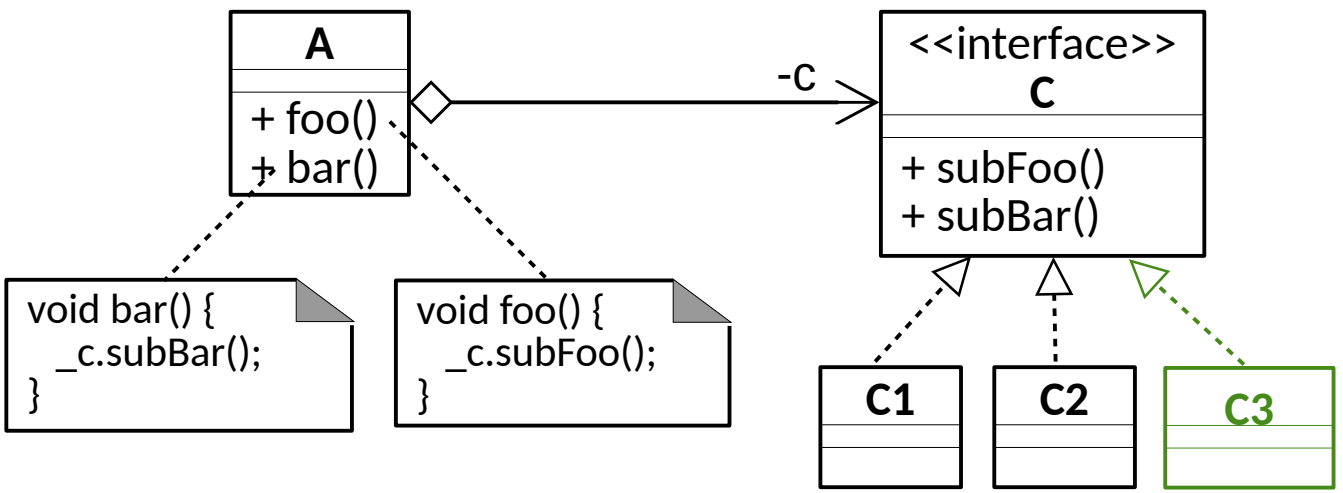
- Considérons la classe A avec 2 méthodes qui dépendent des 2 attributs c1 et c2



- Quelle est la faiblesse de cette conception si on doit ajouter une variation sur un attribut c3 ?
  - Il faut modifier le code de foo() et bar() pour y intégrer la variation sur c3

Exemple (refactoring)

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



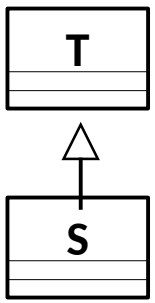
  - Création : `A a = new A(new C1());`
    ou `A a = new A(new C3());`

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 comment associer l'instance de A avec une instance de 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 (SO[L]ID)

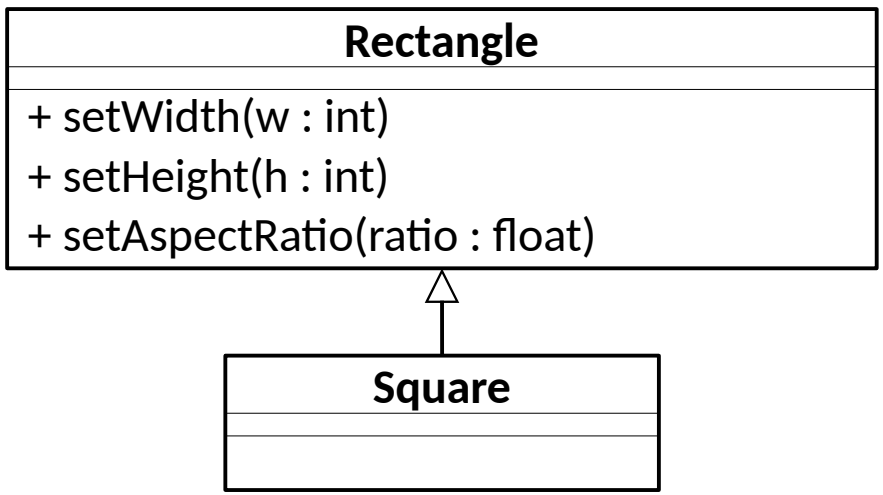
- **Les objets de classes dérivées doivent être substituables aux objets de la classe de base**



- **Objectif**
  - Restreindre l'utilisation de l'héritage pour éviter une régression logicielle lors de l'ajout d'un nouveau sous-type dans une hiérarchie

Exemple

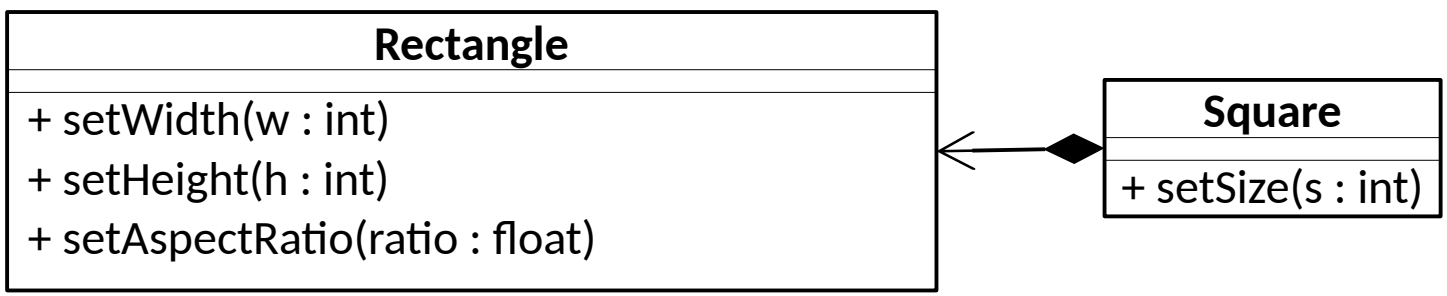
- Soit la modélisation suivante : un carré est un rectangle particulier



- Pourquoi cette modélisation est fausse ?
  - Le carré ne respecte pas l'intégralité du contrat de *Rectangle :*
    - Après `setWidth()` on ne s'attend pas à ce que la hauteur change aussi.
    - La méthode `setAspectRatio()` (*4/3*, *16/9*, *A4*, *etc*) n'a pas de sens pour le carré

Exemple (refactoring)

- Comment modifier la conception pour éviter le problème de substitution sans dupliquer le code réellement commun ?
- Solution : composition



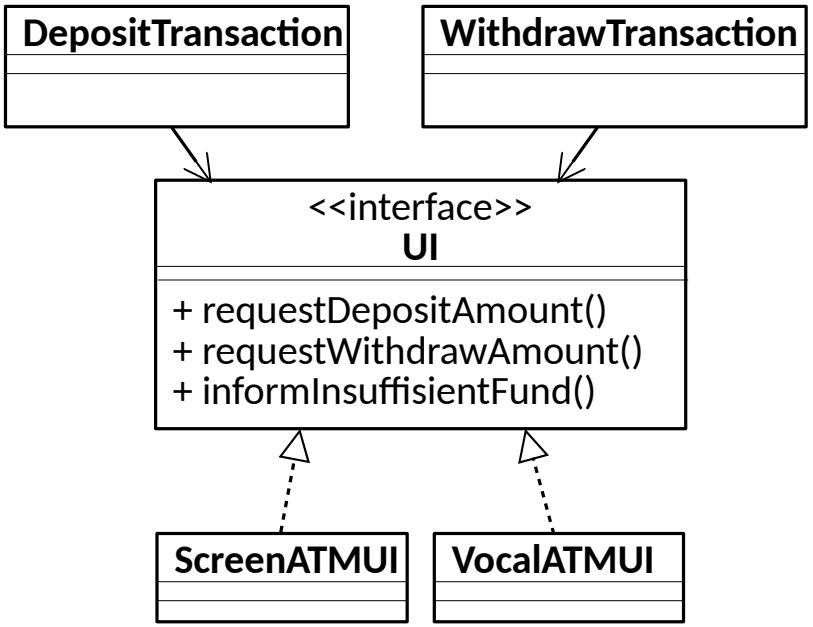
  - Cette fois, le carré n'est pas substituable au rectangle
  - Le code de rectangle est partagé avec carré

Principe 4. Ségrégation d'interface (SOL[I]D)

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

Exemple

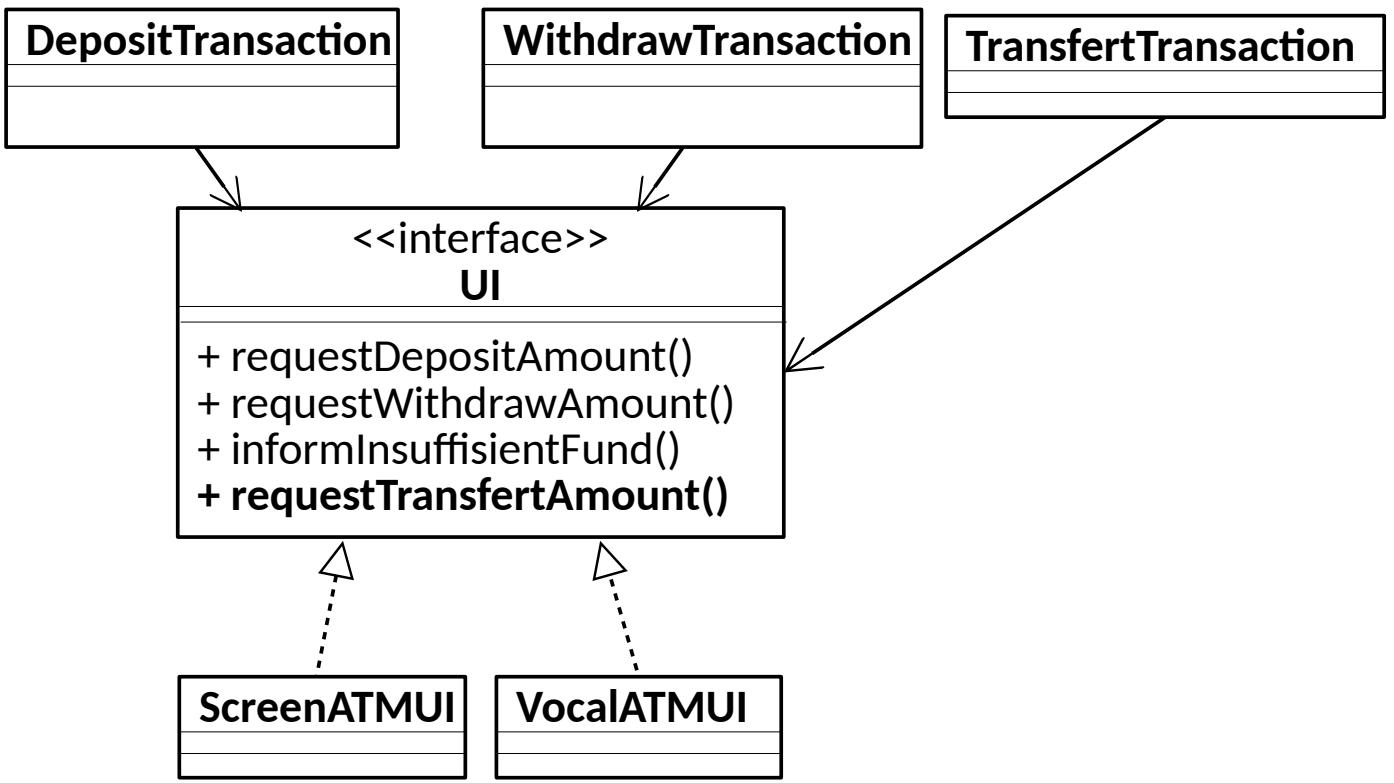
- Guichet Automatique Bancaire (GAB)
  - Toutes les transactions interagissent avec la même interface leur permettant de profiter des méthodes d'affichage sur écran ou de synthèse vocale



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

Exemple

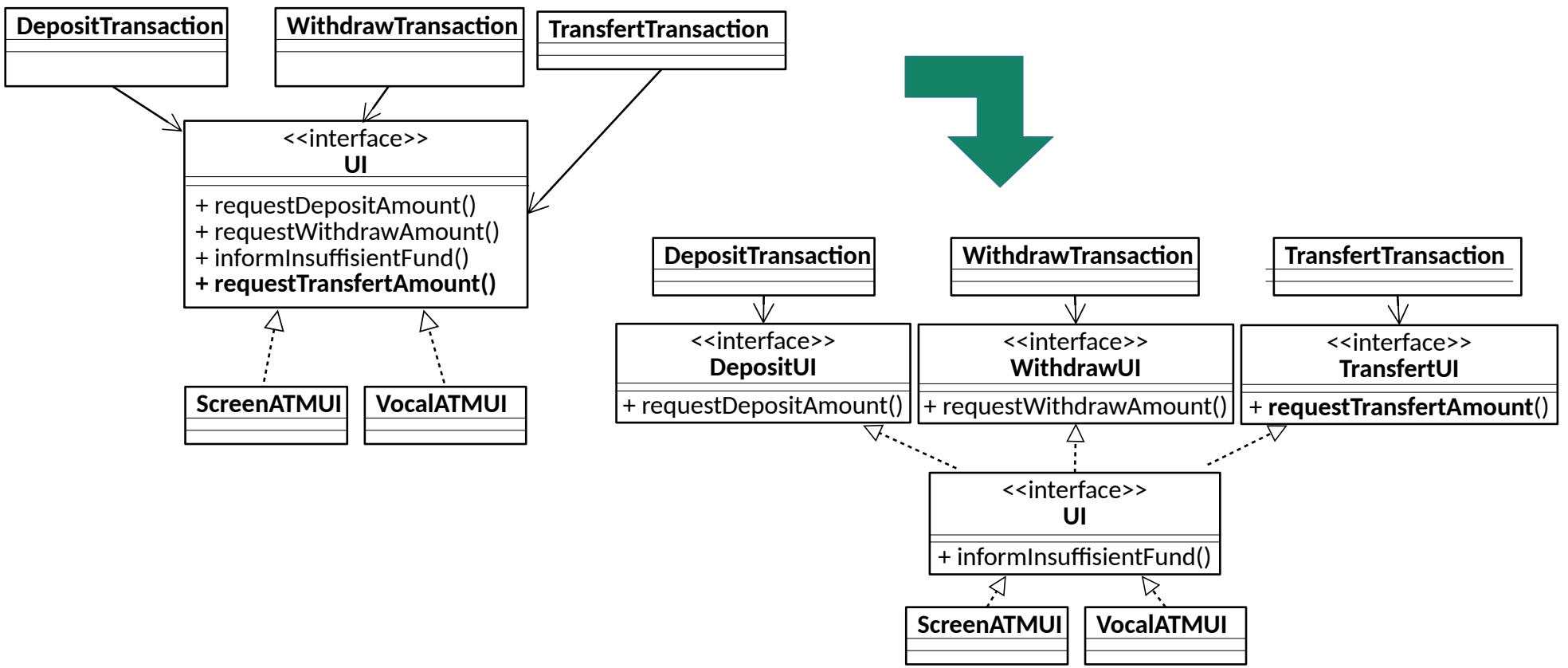
- On veut ajouter une nouvelle fonctionnalité de transfert



- **Problème**
  - Les classes `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 ?

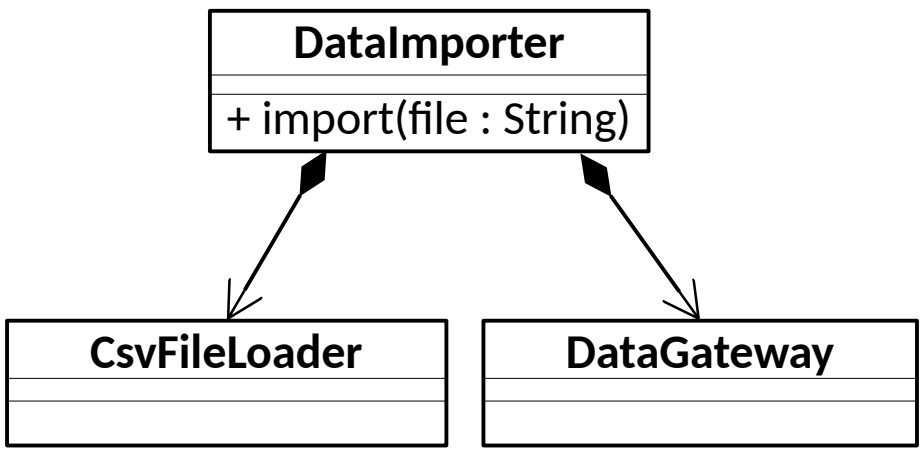


Principe 5. Inversion des dépendances (SOLI[D])

- **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
- **Objectifs**
  - Éviter que les changements dans les modules de bas niveau plus versatiles n'impactent les modules de haut niveau plus stables
  - Augmenter la réutilisabilité des modules de haut niveau

Exemple

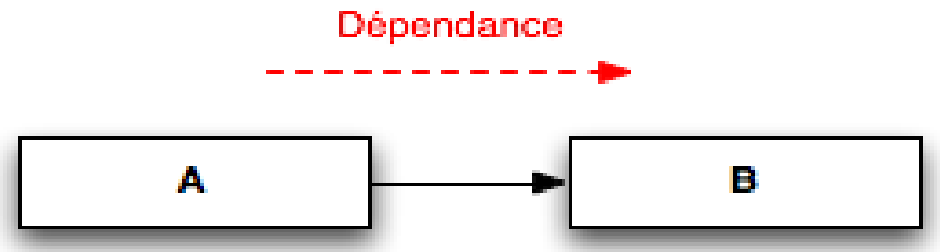
- La classe `DataImporter` est dépendante du chargeur de fichier et de la passerelle de stockage



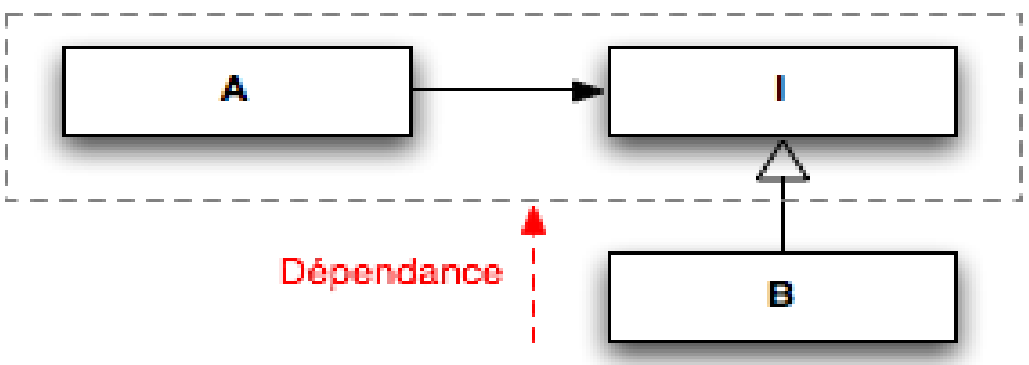
- Quel est le problème ?
  - On ne peut pas réutiliser la classe d'importation sans réutiliser le chargeur de fichier CSV et la passerelle de stockage des données

L'abstraction comme technique d'inversion des dépendances

- Relation conventionnelle

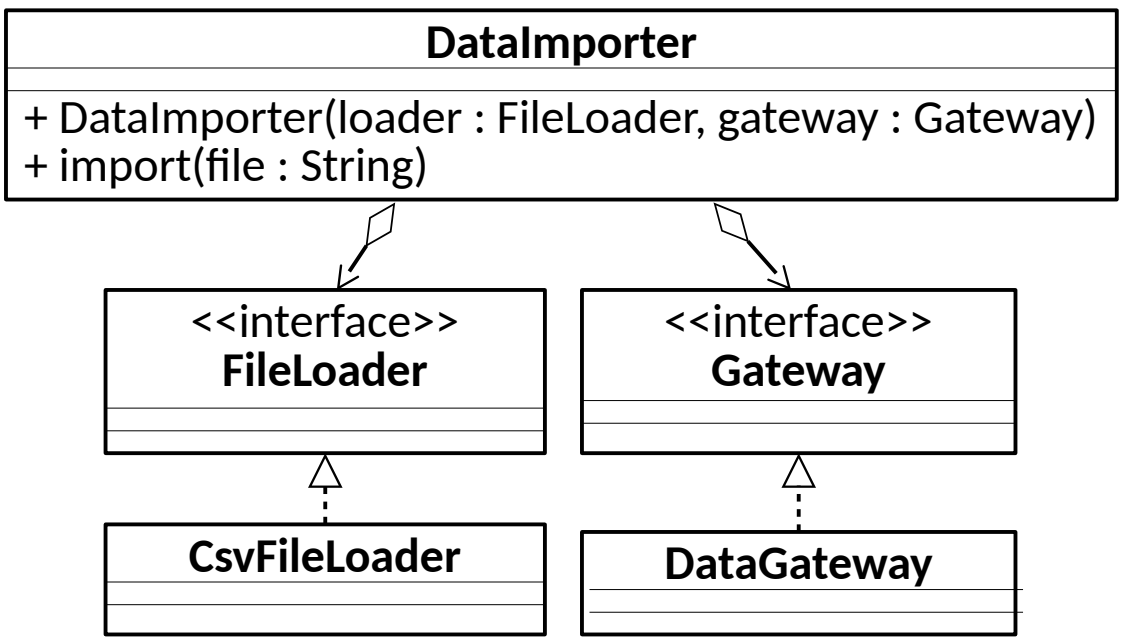


- Inversion de la dépendance par introduction d'une interface entre la classe A et la classe B



Exemple (refactoring)

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



II. Règles de conception

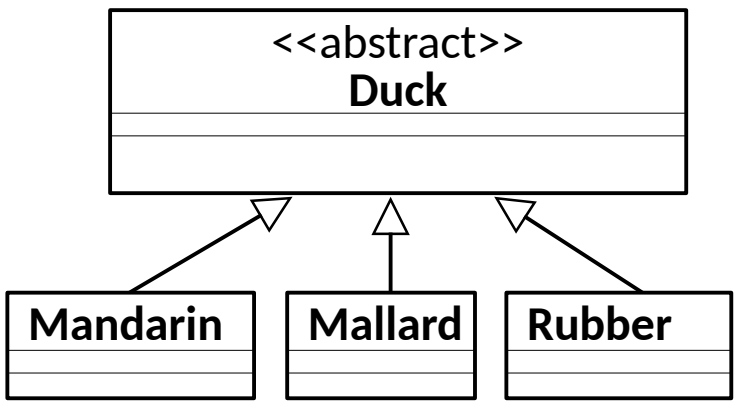
- 4 règles
  - Règle 1. Réduire l'accessibilité des membres de classe
  - Règle 2. Encapsuler ce qui varie
  - Règle 3. Programmer pour une interface, non pour une implémentation
  - Règle 4. Privilégier la composition à l'héritage

Règle 1. Réduire l'accessibilité des membres de classe

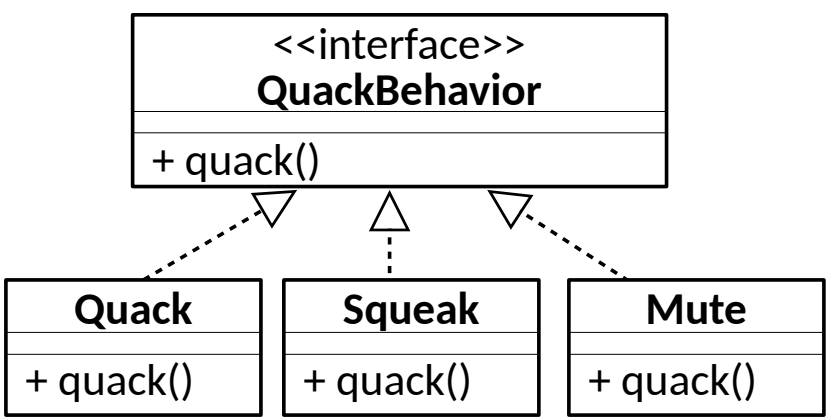
- **L'accès direct aux données membres d'une classe devrait être limité à la classe elle-même**
  - Éviter d'exposer les détails d'implémentation pour faciliter l'évolution future sans aucune conséquence sur les classes liées
- Solution : encapsulation
  - Faire des attributs privés
  - Réduire l'utilisation 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 est variable dans une conception puis encapsuler la variation 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 :

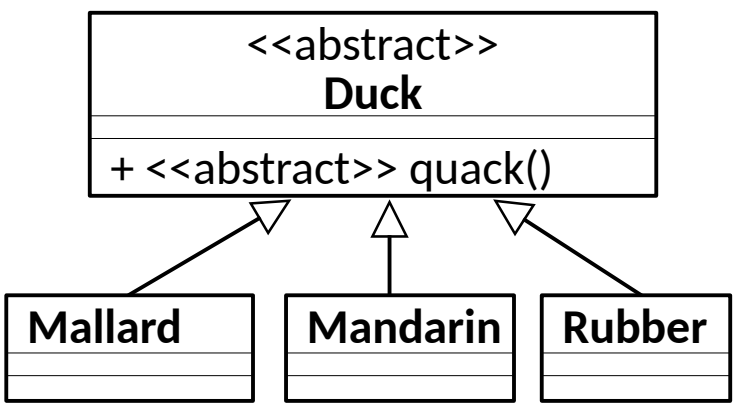
```
Mallard d = new Mallard();
d.quack();
```

  - Programmer pour une interface :

```
Duck d = new Mallard();
d.quack();
```

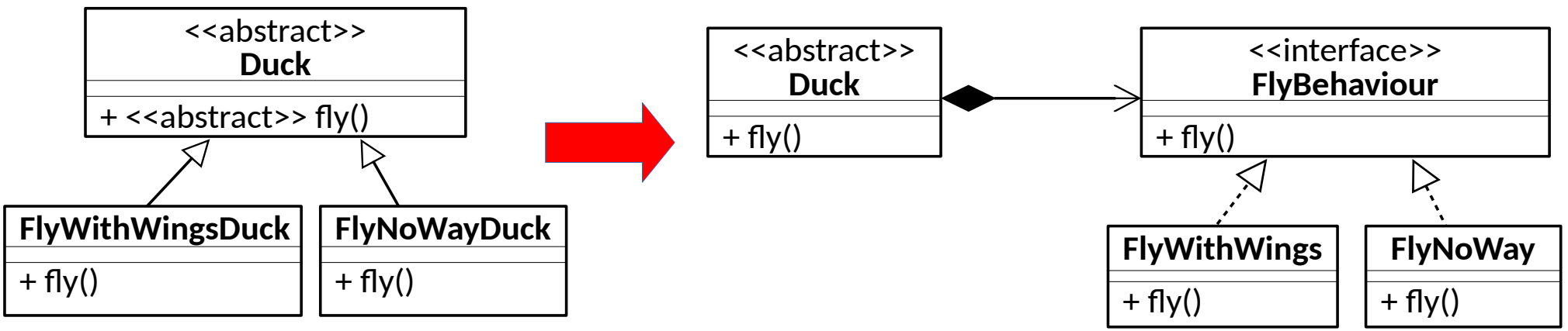
Ce qui permet des évolutions sans rien changer par ailleurs :

```
Duck d = new Mandarin();
d.quack();
```



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 de type boîte noire*)



Règle de Coad pour l’héritage

- Il faut 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 d'une sous-classe n'a jamais besoin de devenir l'objet d'une autre
  - 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, ce qui est contraire au principe de Liskov.

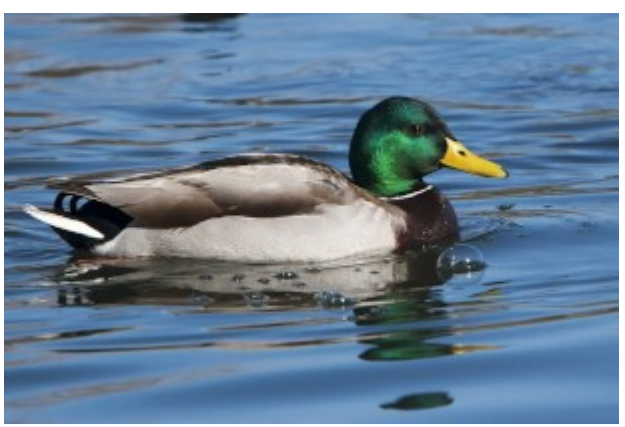
Exemple de l’application des principes et des règles pour la conception

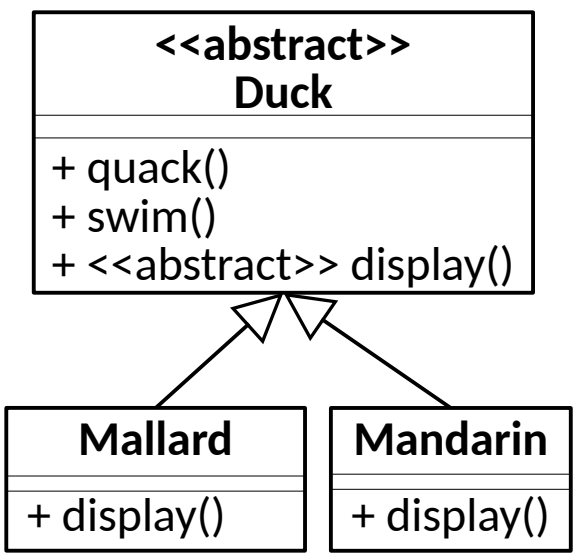
Étude de cas

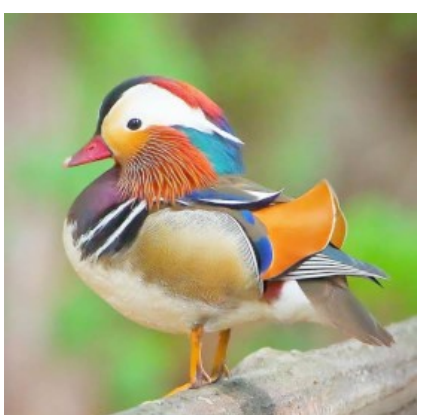
- Un jeu de simulation d'une mare aux canards

Jeu de simulation d'une mare aux canards

- **Besoin initial**
  - Modélisation d'une large gamme de Canards nageant et cancanant
    - Seul l'affichage est spécifique de chaque espèce
- **Solution**

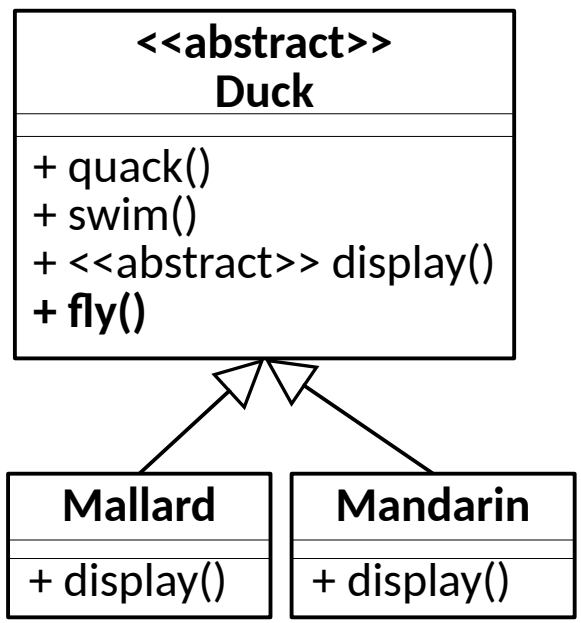






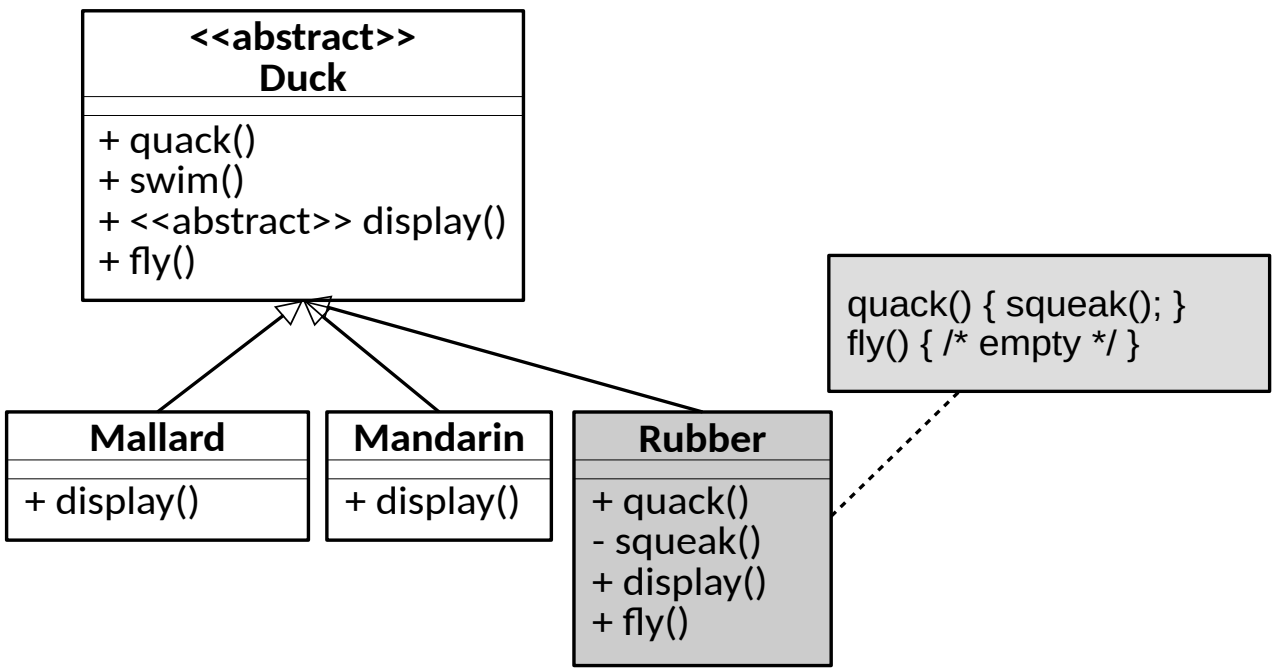
Jeu de simulation d'une mare aux canards

- **Nouveau besoin**
  - voler
- **Solution**
  - Ajout d'une méthode concrète `fly()` dans la classe abstraite qui est commune à tous les canards



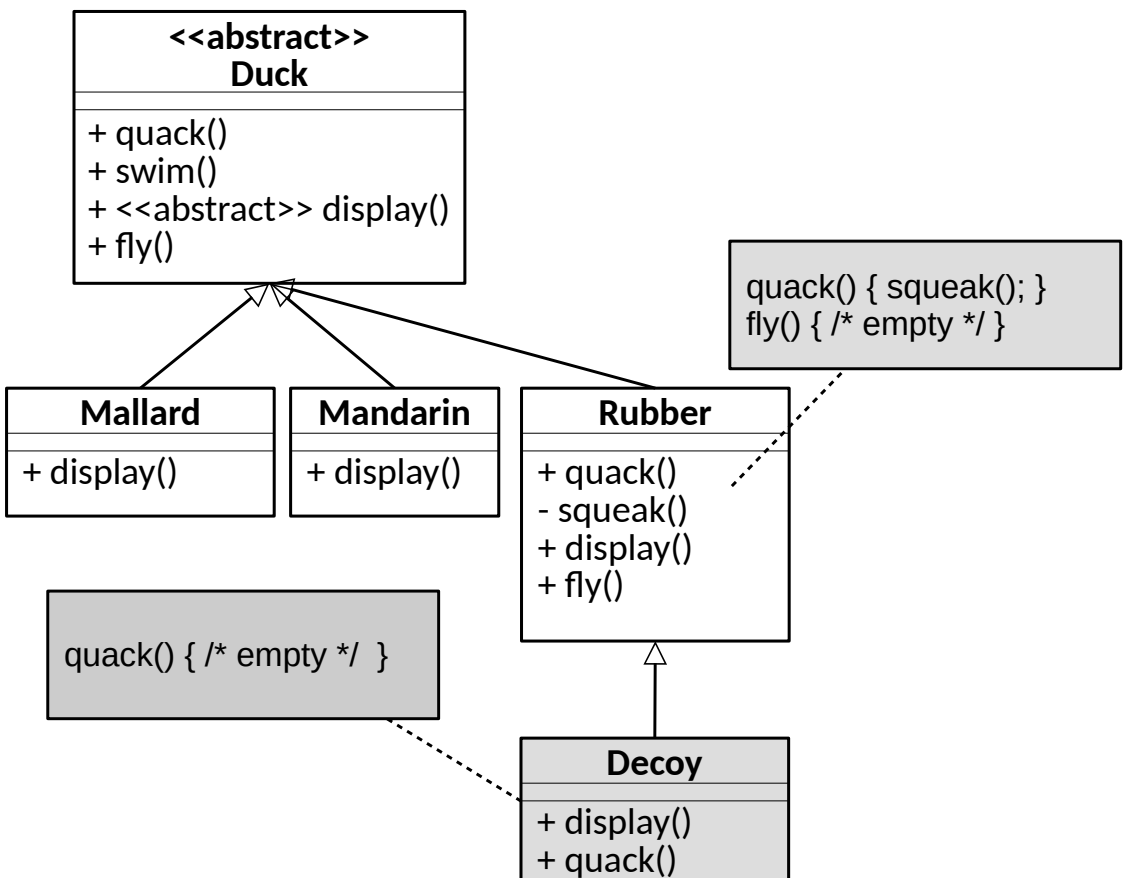
Jeu de simulation d'une mare aux canards

- **Nouveau besoin**
  - Canard en plastique : ne vole pas et ne cancane pas mais couine
- **Solution**



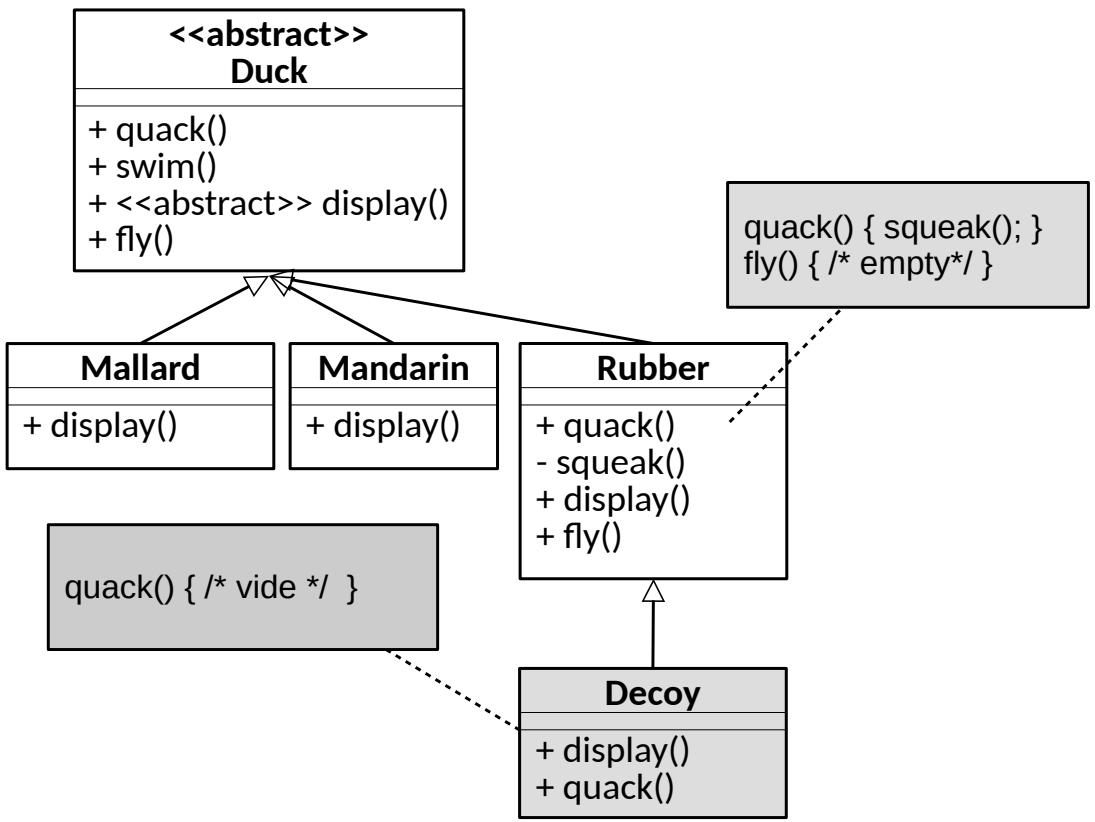
Jeu de simulation d'une mare aux canards

- **Nouveau besoin**
  - Canard leurre : ne vole pas et est muet
- **Solution**



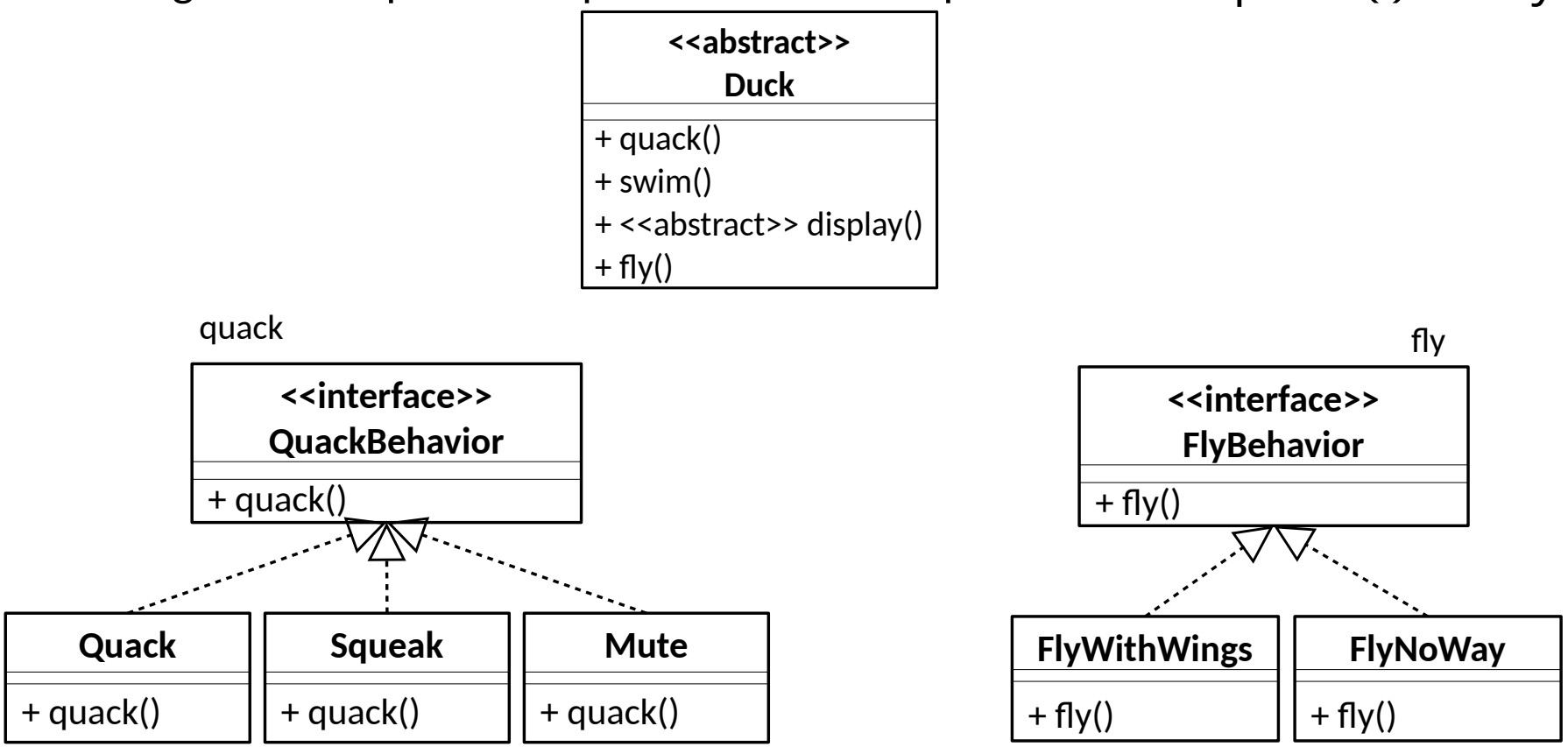
Analyse de la solution courante

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



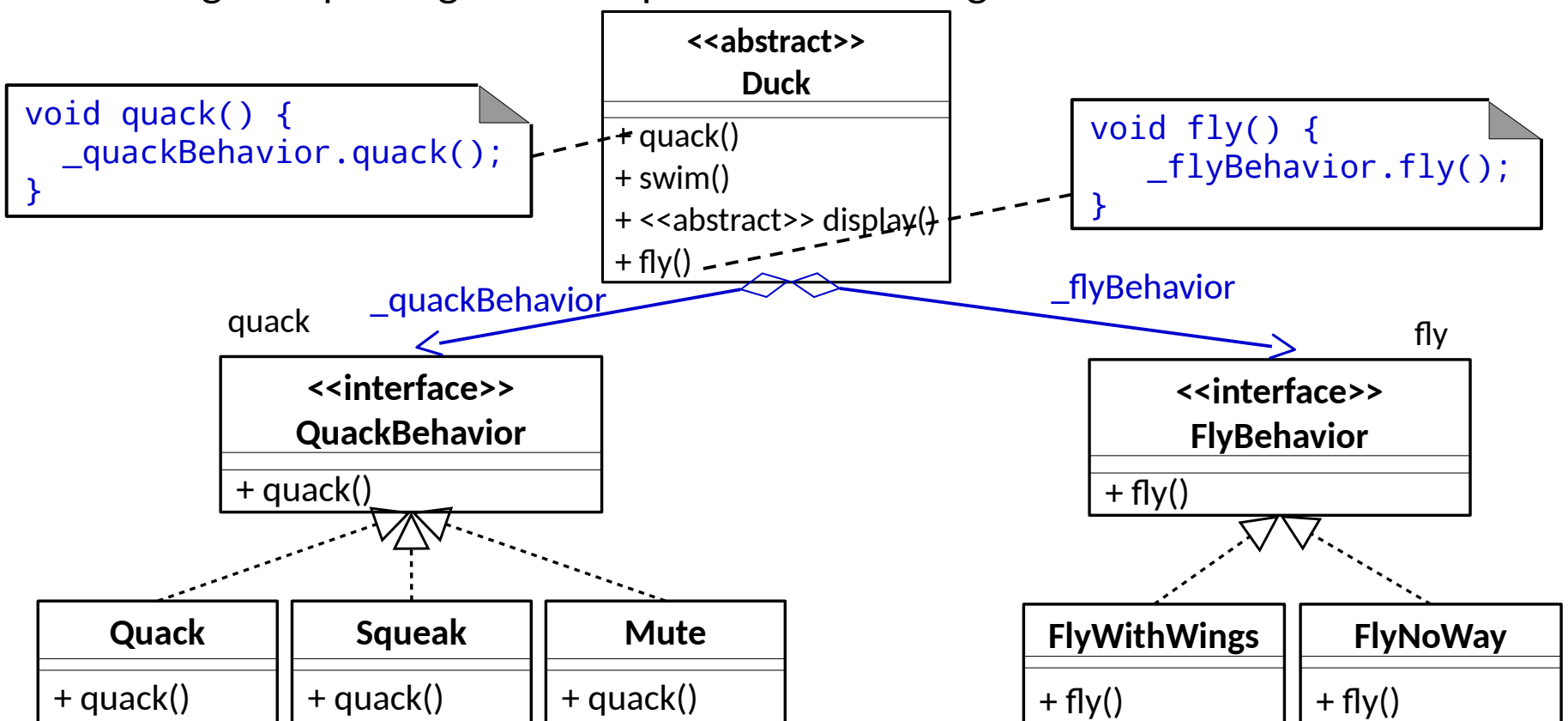
Jeu de simulation d'une mare aux canards

- **Solution**
  - Refondre en appliquant les **règles 2 et 4**
    - Règle 2 : encapsuler ce qui varie : ici le comportement de `quack()` et `fly()`



Jeu de simulation d'une mare aux canards

- **Solution (suite)**
  - Refondre en appliquant les **règles 2 et 4**
    - Règle 4 : privilégier la composition à l'héritage



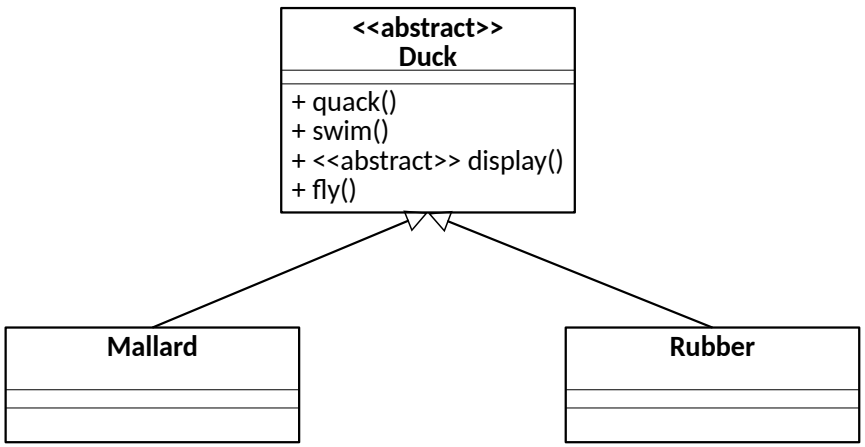
Jeu de simulation d'une mare aux canards

- Code de la classe abstraite Duck

```
abstract class Duck {
    private final QuackBehavior _quackBehavior;
    private final FlyBehavior _flyBehavior;
    public Duck( QuackBehavior q, FlyBehavior f ) {
        _quackBehavior = q;
        _flyBehavior = f;
    }
    public void fly() { _flyBehavior.fly(); }
    public void quack() { _quackBehavior.quack(); }
}
```

Jeu de simulation d'une mare aux canards

- Comment créer des canards mallards ou des canards en plastique ?




```
class Mallard extends Duck {
    public Mallard() {
        super(new Quack(), new FlyWithWings());
    }
}
class Rubber extends Duck {
    public Rubber() {
        super(new Squeak(), new FlyNoWay());
    }
}
```

Analyse de la solution

- La conception est-elle SOLID ?

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 là où l'évolution et la réutilisation sont des contraintes importantes
- 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 !