

Les patrons de conception (Design patterns)

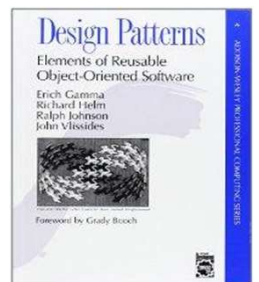
Dr. Ing. Jemal AHMED
Maître-Technologue à l'ISSET
ahmed.jemal.sfax.iset@gmail.com

Plan

- ▶ Historique & Motivation
- ▶ Le patron « Strategy »
- ▶ Le patron « Observer »
- ▶ Le patron « Decorator »
- ▶ Le patron « Factory »
- ▶ Le patron « Singleton »
- ▶ Le patron « Command »
- ▶ Le patron « Adapter »
- ▶ Le patron « Façade »
- ▶ Autres patrons

Historique

- ▶ Notion de « patron » d'abord apparue en architecture :
 - ▶ l'architecture des bâtiments
 - ▶ la conception des villes et de leur environnement
- ▶ L'architecte Christopher Alexander le définit comme suit:
 - ▶ «Chaque modèle [patron] décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cœur de la solution de ce problème, d'une façon telle que l'on peut réutiliser cette solution des millions de fois.» [Livre: *The Timeless Way of Building*, Oxford University Press 1979]
- ▶ Projeter la notion de patron à du logiciel : "**design pattern**"
 - ▶ premiers patrons à partir de 1987 (partie de la thèse de Erich Gamma)
 - ▶ puis Richard Helm, John Vlissides et Ralph Johnson («Gang of Four, GoF»)
 - ▶ premier catalogue en 1993 : *Elements of Reusable Object-Oriented Software*
- ▶ Vocabulaire:
 - ▶ modèles de conception= patrons de conception= motifs de conception= design patterns



Motivation

- ▶ Pourquoi définir des patrons de conception
 - ▶ Construire des systèmes plus extensibles, plus robustes au changement
 - ▶ Capitaliser l'*expérience collective des informaticiens*
 - ▶ Réutiliser les solutions qui ont fait leur preuve
 - ▶ Accélérer le “development process” en fournissant des paradigmes de développement prouvés et testés
 - ▶ Faciliter la communication entre les membres de l'équipe en utilisant des termes bien connus et bien assimilés
- ▶ Complémentaire avec les API
 - ▶ Une API propose des solutions directement utilisables
 - ▶ Un patron explique comment structurer son application avec une API
- ▶ Patron de conception dans le cycle de développement
 - ▶ Intervient en conception détaillée
 - ▶ Reste indépendant du langage d'implantation

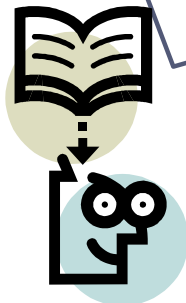
Définition

- ▶ Définition : "**Un patron de conception (design pattern) décrit une structure commune et répétitive de composants en interaction (la solution) qui résout un problème de conception dans un contexte particulier**"
- ▶ Au lieu de la réutilisation de code, on parle de la **réutilisation de l'expérience** avec les patrons
- ▶ Un bon patron de conception :
 - ▶ résout un problème
 - ▶ correspond à une solution éprouvée
 - ▶ favorise la réutilisabilité, l'extensibilité, etc.

Intérêt et utilisation des patrons de conception

- ▶ La meilleure manière d'utilisation des patrons de conception est de les mémoriser en tête, puis reconnaître leurs emplacements et les appliquer dans la conception des applications

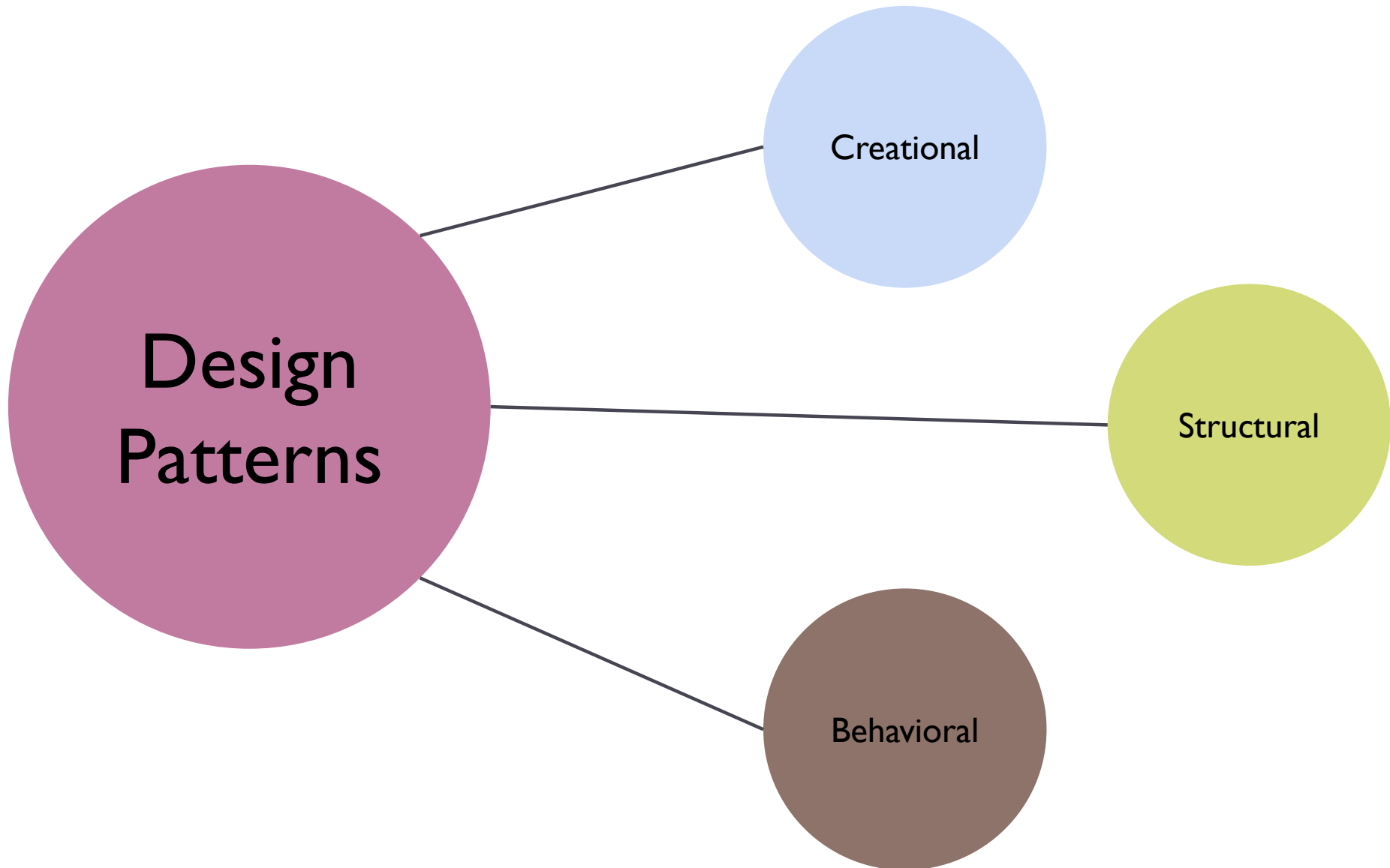
**Les concepts de l'orienté objet tels que
l'abstraction, l'héritage, et le polymorphisme ne
te rendent pas un bon concepteur!
Un concepteur pense à créer une conception
flexible qui est maintenable et fait face aux
changements**



Ce qu'il ne faut pas attendre des patrons de conception

- ▶ Une solution universelle prête à l'emploi
- ▶ Une bibliothèque de classes réutilisables
- ▶ L'automatisation totale de l'instanciation d'un patron de conception
- ▶ La disparition du facteur humain

Type des design patterns (1 / 4)



Type des design patterns (2/4)

▶ Creational Design Patterns

- ▶ Traiter avec les mécanismes de création des objets, essayant de créer les objets de façon adaptée à la situation
- ▶ La forme basique de la création d'objets peut mener à des soucis de design ou le rendre plus complexe. Les Creational design patterns résolvent ce problème en contrôlant la création des objets

Type des design patterns (3/4)

- ▶ **Structural Design Patterns**

- ▶ Ce sont des patterns qui facilitent le design en identifiant un moyen simple pour mettre en place la relation entre les entités

- ▶ **Behavioral Design Patterns**

- ▶ Ce sont des patterns qui sont concernés par les algorithmes et l'assignation des responsabilités entre les objets

Type des design patterns (4 / 4)

Creational Patterns

- Factory Method
- Abstract Factory
- Singleton
- Builder
- Prototype

Structural Patterns

- Adapter
- Decorator
- Façade
- Bridge
- Composite
- Flyweight
- Proxy

Behavioral Patterns

- Command
- Observer
- Strategy
- Chain of responsibility
- Interpreter
- Iterator
- Mediator
- Memento
- State
- Template Method
- Visitor

Utilisation des design patterns

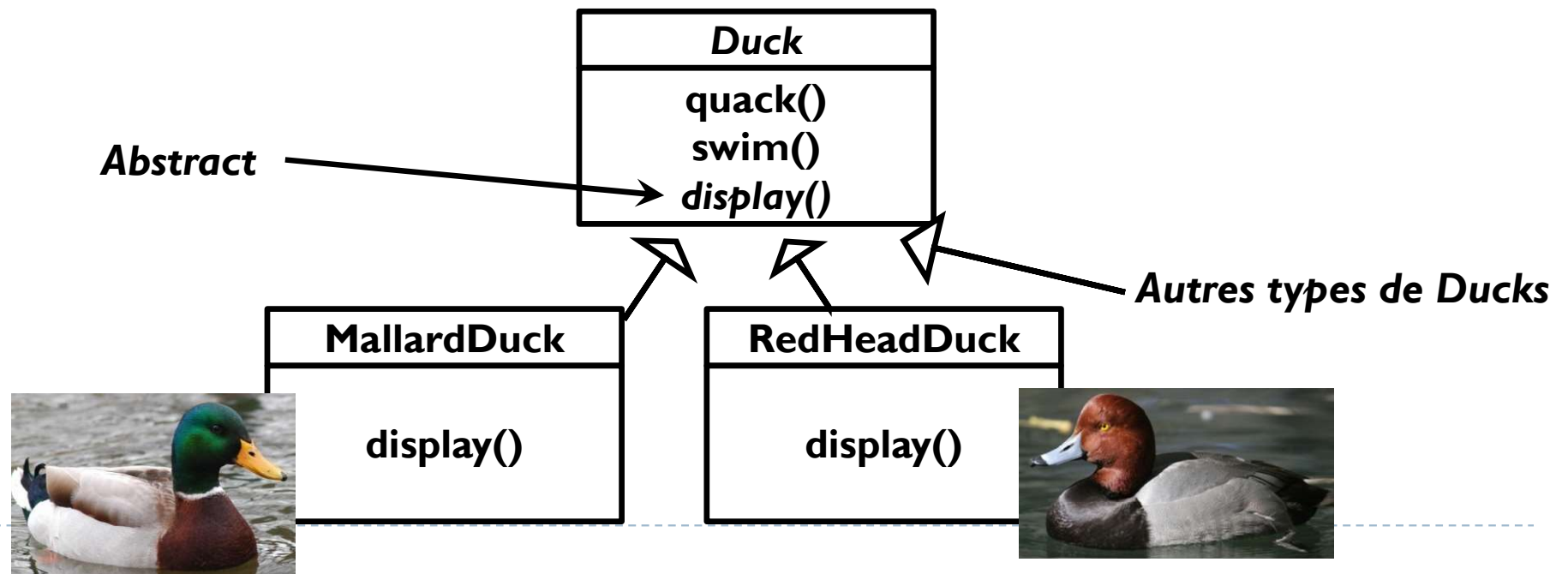
High	Medium High	Medium	Medium Low	Low
Abstract Factory	Adapter	Decorator	Builder	Flyweight
Factory Method	Command	Bridge	Chain of responsibility	Interpreter
Façade	Strategy	Prototype	Mediator	Memento
Observer	Singleton	State		Visitor
Iterator	Composite	Template Method		
	Proxy			

Le patron "Strategy"

SimUDuck:

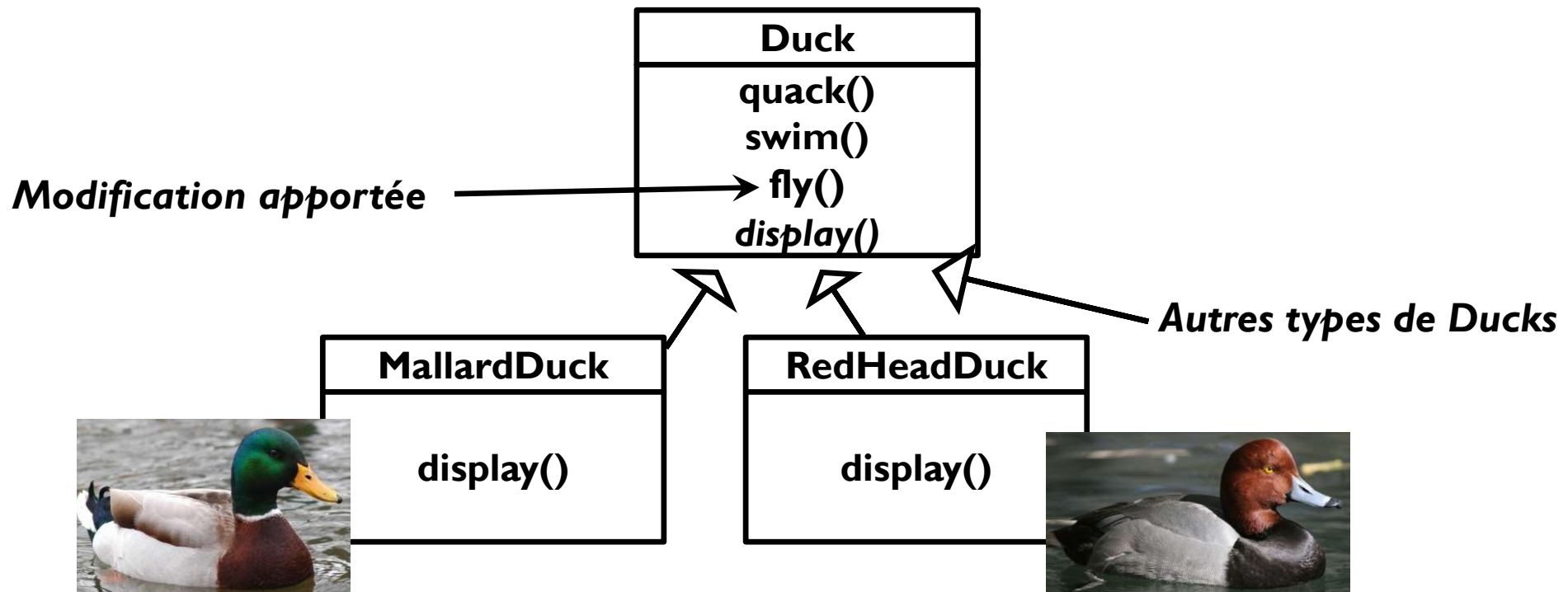
Conception (1/23)

- ▶ Objectif: développement d'un jeu de simulation d'un bassin pour les canards
- ▶ Besoin: nager, cancaner, afficher, etc..
 - ▶ Supporter une large variété de canards
- ▶ Conception: OO
 - ▶ Une supère classe Canard (Duck) dont tous les canards héritent



SimUDuck : Innovation (2/23)

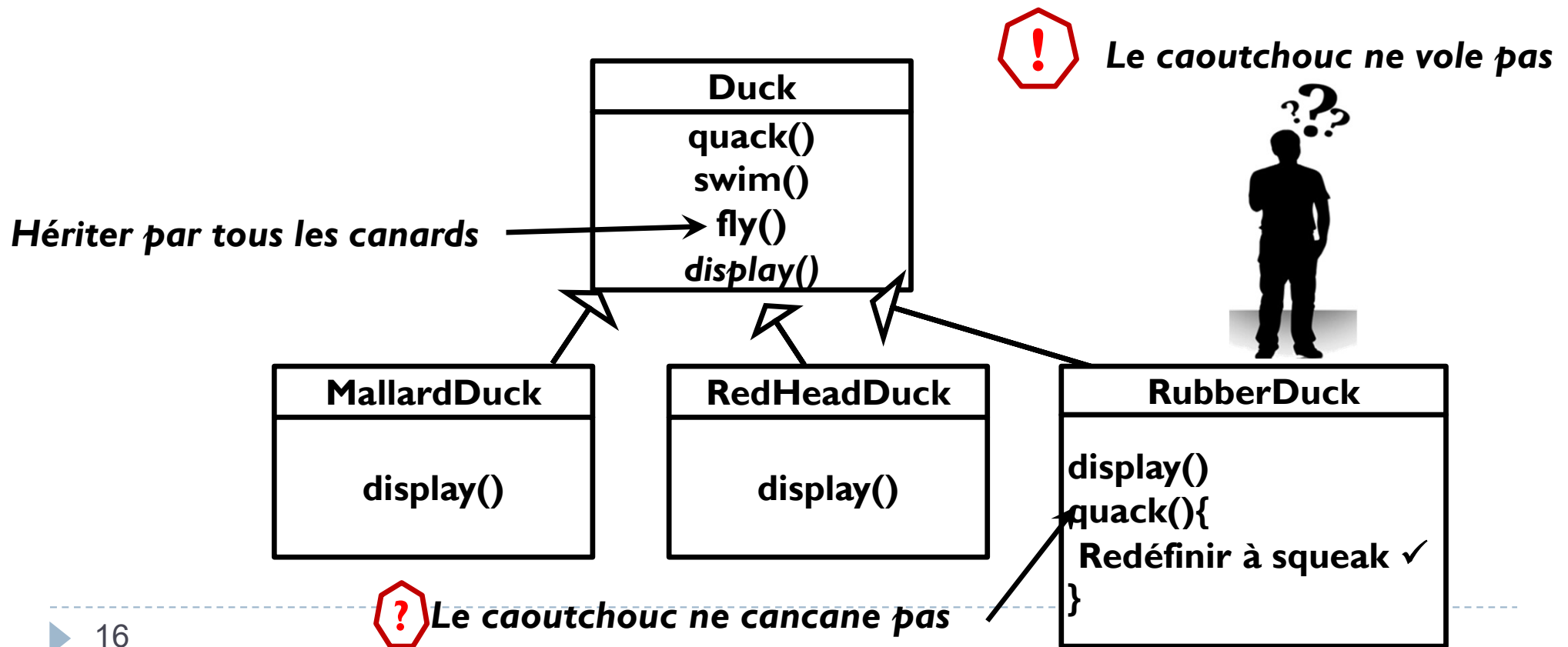
- ▶ Objectif: Innovation (pour impressionner et vendre +)
- ▶ Besoin: simuler le vol des canards!
- ▶ Conception: OO
 - ▶ Ajouter la méthode `fly()` à la supère classe



SimUDuck :

Problèmes (3/23)

- ▶ Besoin: Au moment de la démonstration du simulateur, on nous demande de simuler des canards en caoutchouc
- ▶ Conception: OO
 - ▶ Ajouter la classe RubberDuck qui hérite de la supère classe Duck



SimUDuck :

Constat (4/23)

- ▶ Problème 1: Le canard en caoutchouc ne cancanne pas!
- ▶ Solution : Redéfinir la méthode quack() à squeak()
(résolu)
- ▶ Problème 2: Le canard en caoutchouc ne vole pas!
Toutefois, il hérite la méthode fly() de la supère classe Duck!
- ▶ Constat:
 - ▶ Ce que nous avons cru une utilisation formidable de l'héritage dans le but de la réutilisation, s'est terminé mal au moment de la mise à jour!
 - ▶ Une mise à jour du code a causé un effet global sur l'application!

SimUDuck :

Solution?? (5/23)

- ▶ Problème 2: Le canard en caoutchouc ne vole pas! Toutefois, il hérite la méthode `fly()` de la supère classe `Duck`!
- ▶ Solution: Redéfinir la méthode `fly()` de `RubberDuck`

RubberDuck
<code>display()</code> <code>quack(){ squeak }</code> <code>fly(){</code> Redéfinir à rien ✓ <code>}</code>

- ▶ Question: est ce que c'est résolu pour tous types de canards?



SimUDuck :

Un autre canard (6/23)

- ▶ Nouveau type de canard: Canard en bois
- ▶ Problèmes levés:
 - ▶ Ce canard ne cancanne pas
 - ▶ Ce canard ne vole pas
- ▶ Solution: redéfinir (une autre fois) les méthodes quack() et fly()



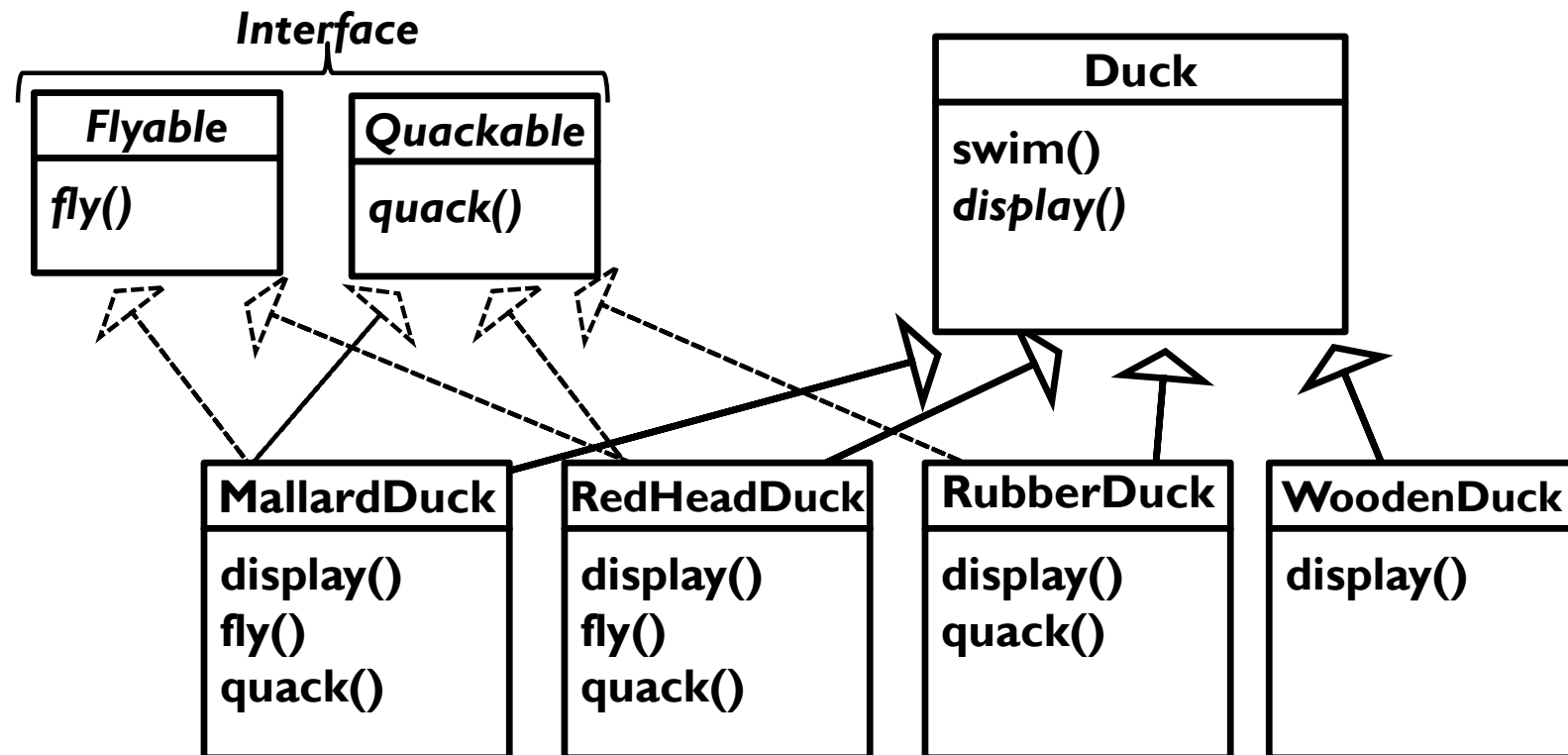
WoodenDuck
<pre>display() quack(){ Redéfinir à rien ✓ } fly(){ Redéfinir à rien ✓ }</pre>

- ▶ Inconvénients de l'utilisation de l'héritage
 - ▶ Il est difficile de connaître le comportement de tous les canards
 - ▶ Un changement non-intentionnel, affecte les autres canards

SimUDuck :

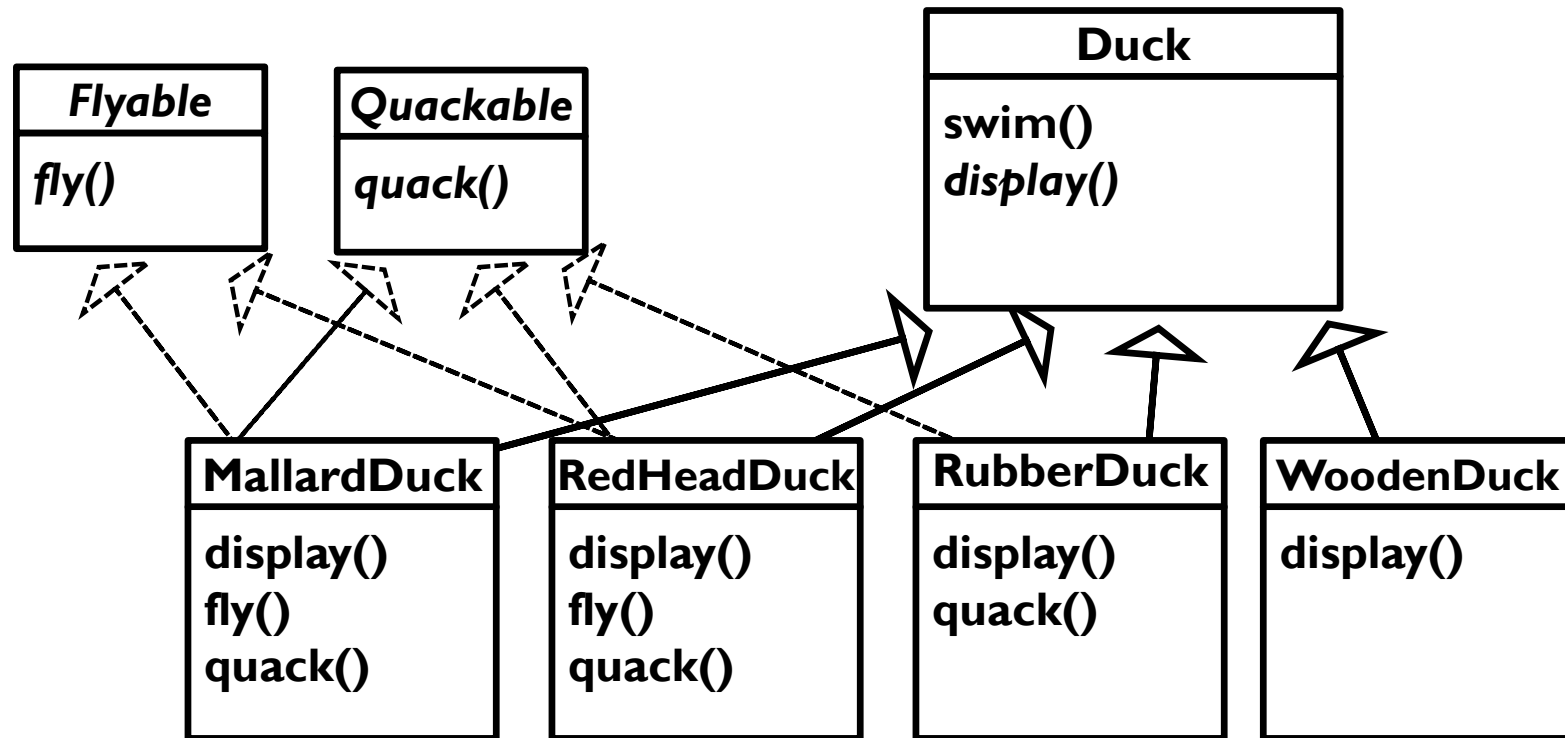
Des interfaces? (7/23)

- ▶ Hypothèse: On nous demande de mettre à jour SimUDuck tous les 6 mois: La spécification demeure changeante
 - ▶ Vérifier fly() et quack() pour chaque nouveau canard
 - ▶ Ré-écrire (si besoin) fly() et quack()
- ▶ Solution possible pour contourner le problème: les interfaces



SimUDuck :

Inconvénients (8/23)



- ▶ **Constat:**
 - ▶ **Duplication de code:** méthodes `fly()` et `quack()` dans les sous-classes
 - ▶ Autant d'interfaces tant qu'il y a un ensemble de canards ayant exclusivement un comportement commun (pondre: `lay()` pour les canards qui peuvent déposer un œuf)
- ▶ **Problème:** si on veut modifier/adapter légèrement la méthode `fly()`, il faut le faire pour toutes les classes des canards (10 classes, 100, ou +)

SimUDuck :

Moment de réflexion (9 / 23)

- ▶ Pas toutes les sous-classes qui ont besoin de voler (fly) ou de cancaner (quack)
 - ▶ L'héritage n'est pas la bonne solution
- ▶ Les interfaces Flyable et Quackable résolvent une partie du problème
 - ▶ Détruit complètement la réutilisation du code pour ces comportements
 - ▶ La maintenance et la mise à jour représentent un vrai calvaire
- ▶ Supposant qu'il existe plus qu'une façon de voler
 - ▶ Maintenance plus difficile...

SimUDuck :

Solution (10/23)

- ▶ Solution:

- ▶ Design pattern : solution ultime, cheval blanc, sauveur...



- ▶ Trouvons une solution avec l'"ancienne-mode" et ce en appliquant les bonnes principes de la conception OO
 - ▶ Concevoir une application, ou un besoin de modification/changement peut être appliqué avec le moindre possible d'impact sur le code existant

**Donner des raisons de changement de code
dans votre application**

► **Règle 1 :** Identifier les aspects variables de mon application et les séparer de ce qui reste invariant

- C'est la base de tous les patrons de conception
- Système plus flexible + peu de conséquences inattendues

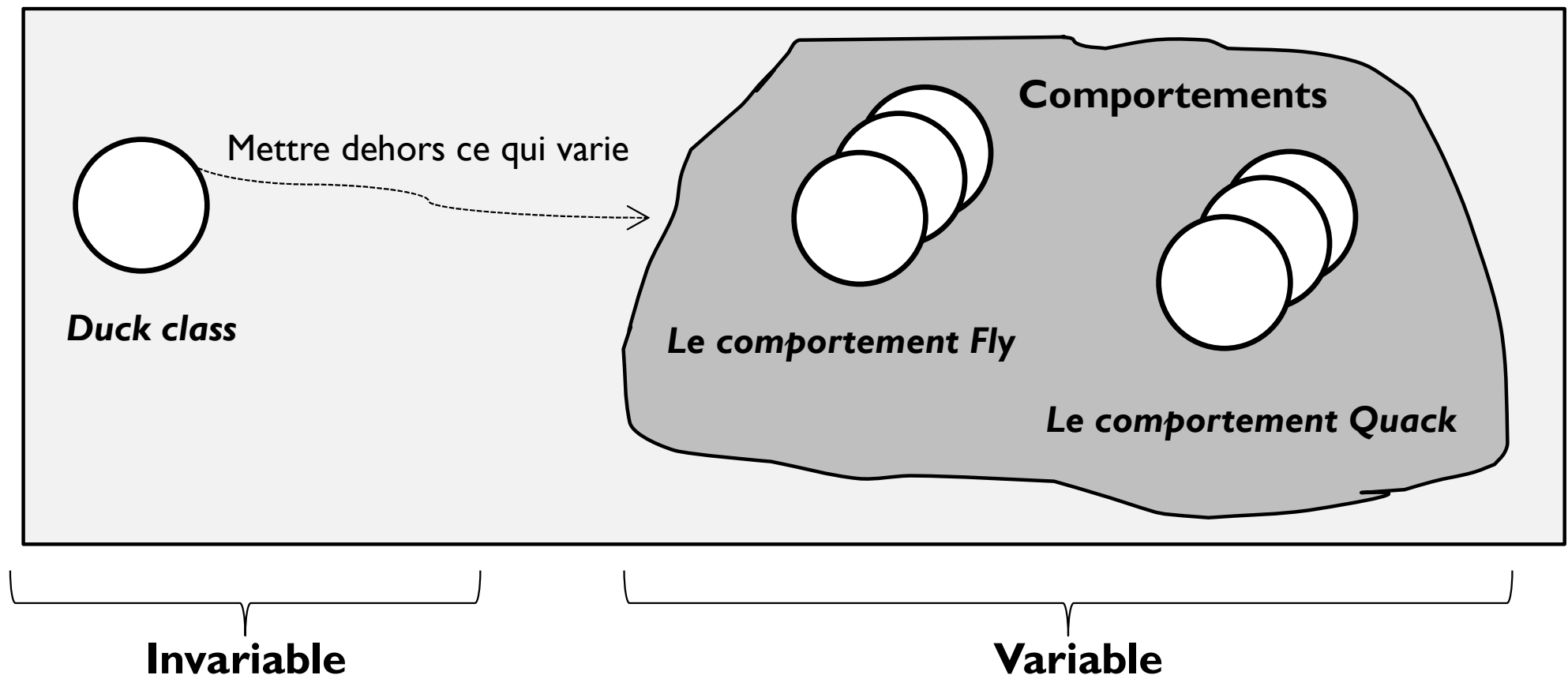
► Mise en œuvre

- Prendre la partie qui varie et l'encapsuler. De cette façon, un changement ultérieur affecte la partie variable, sans toucher à celle invariable

SimUDuck :

Séparation (12/23)

- ▶ La classe Duck est toujours la supère classe
- ▶ Les comportements fly() et quack() sont retirés, et mis dans une autre structure



SimUDuck :

Conception des comportements (13/23)

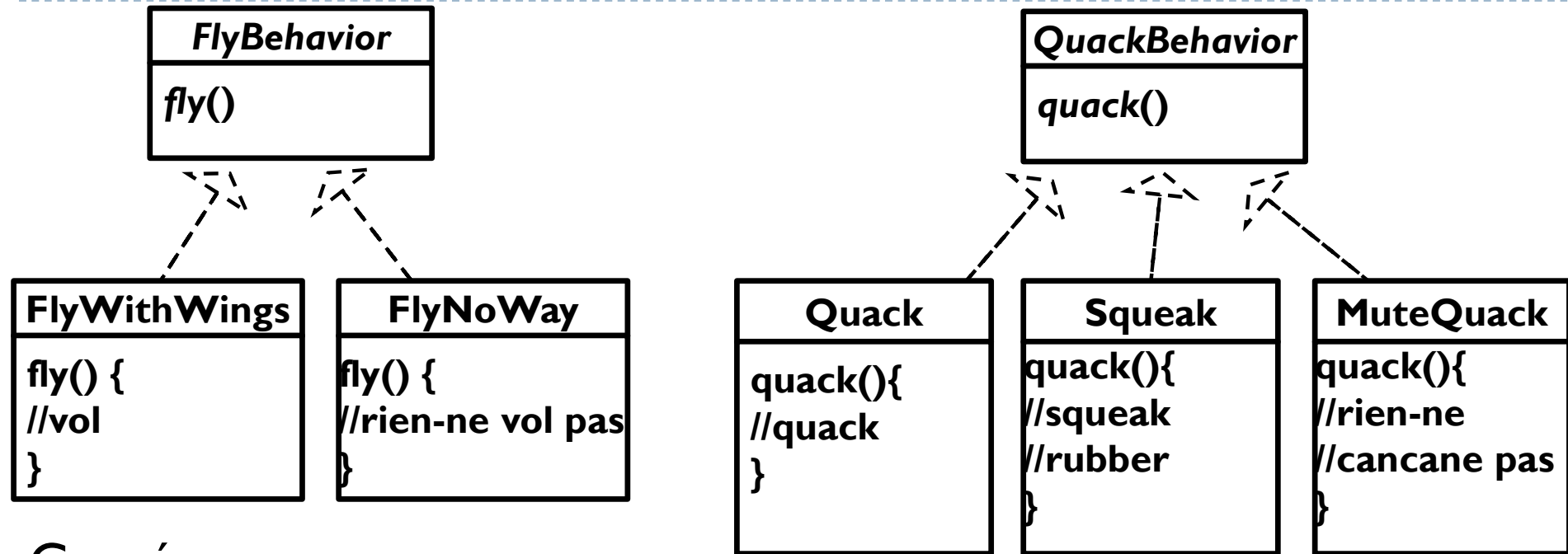
- ▶ Conception initiale: l'inflexibilité des comportements a engendré des troubles
- ▶ On veut affecter les comportements aux instances des Ducks tout en permettant:
 - ▶ La création d'une instance (MallardDuck),
 - ▶ L'initialisation avec un type de comportement (type de vol)
 - ▶ La possibilité de changer le type de vol dynamiquement (?)

▶ **Règle 2:** Programmer des interfaces, et non des implémentations (Program to Interface)

- ▶ Programmer pour les super-types!
- ▶ On utilise des interfaces pour représenter chaque comportement: **FlyBehavior** et **QuackBehavior**

SimUDuck :

Conception des comportements (14/23)



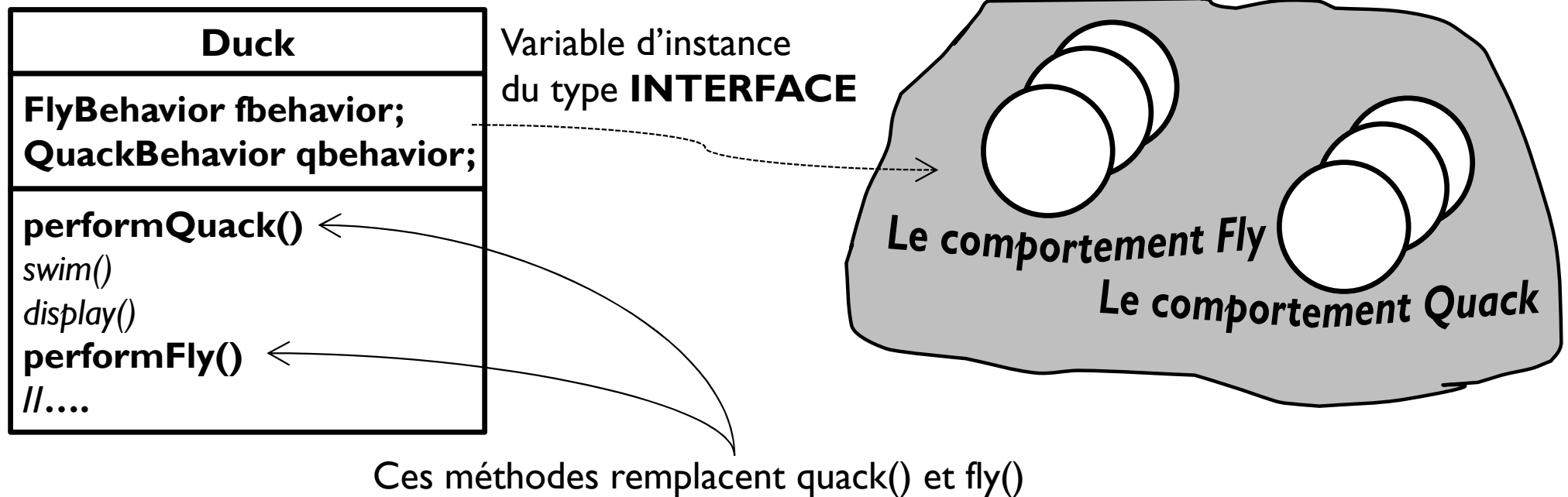
► Conséquences:

- On peut ajouter un nouveau comportement sans modifier ni le code des comportements existants, ni le code des classes des canards qui utilisent les comportements voler/cancaner
- Avec cette conception, d'autres objets peuvent réutiliser le comportement fly et quack, parce qu'ils ne sont plus cachés dans les classes canards.

SimUDuck :

Intégration des comportements(15/23)

- La supère classe Duck, dont hérite tous les canards



- La clé: le canard délègue les comportements fly et quack, au lieu d'utiliser les méthodes fly() et quack() définies dans la supère classe Duck.

SimUDuck :

Implémentation de la supère classe(16/23)

- La supère classe Duck, dont hérite tous les canards

```
public abstract class Duck{  
    protected QuackBehavior qbehavior;  
    protected FlyBehavior fbehavior;  
    //...
```

Chaque type de canard initialise ces attributs selon ses besoins. (par FlyWithWings pour le MallardDuck)

```
    public void performQuack() {  
        qbehavior.quack();  
    }  
    public void performFly() {  
        fbehavior.fly();  
    }  
  
    //..  
}
```

Grace au polymorphisme, la bonne méthode sera invoquée dans la sous-classe du type de canard. (Délégée à la classe gérant le comportement)

SimUDuck :

Implémentation d'un canard (17/23)



Cette classe inclut les méthodes réalisant le comportement fly et quack, par héritage (performQuack(), etc..)

```
public class MallardDuck extends Duck{  
    public MallardDuck () {  
        fbehavior = new FlyWithWings();  
        qbehavior = new Quack();  
    }  
    @Override  
    public void display() {  
        System.out.println("Je suis un canard Mallard");  
    }  
}
```

Initialisation des attributs déclarés dans la supère classe Duck

SimUDuck :

Tester le code du canard(18/23)

- ▶ Développer et compiler [Utiliser NetBeans/Eclipse]:
 - ▶ La classe abstraite Duck (Duck.java)
 - ▶ Le comportements: FlyBehavior.java, FlyWithWings.java et FlyNoWay.java,
 - ▶ Le comportement : QuackBehavior.java, Quack.java, Squeak.java et MuteQuack.java
 - ▶ Les classes MallardDuck.java et WoodenDuck.java
- ▶ Tester toutes les méthodes des canards créés dans un main: MallardDuckSim.java

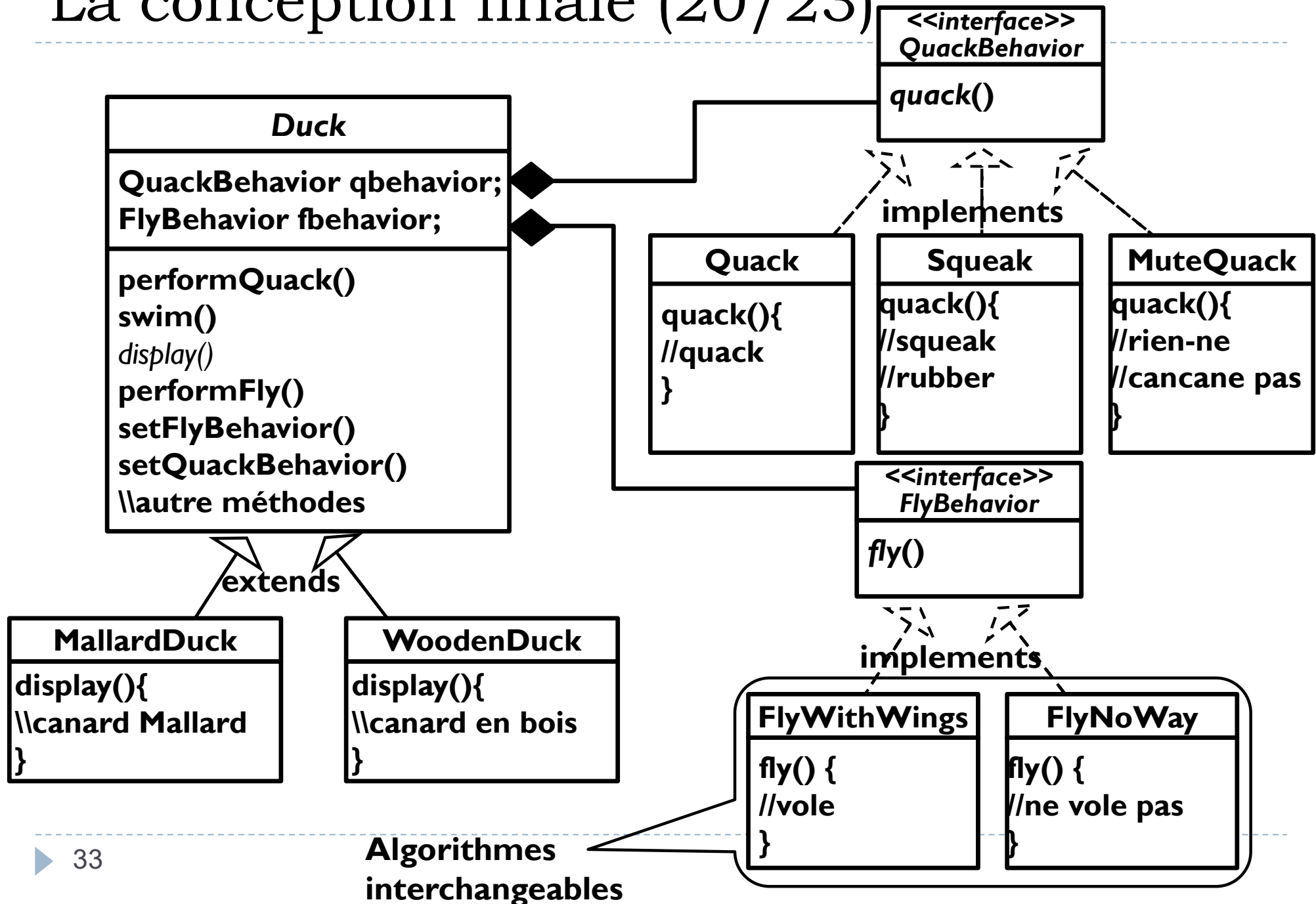
SimUDuck :

Le comportement dynamique (19/23)

- ▶ Changement dynamique de comportement
 - ▶ Ajouter les méthodes: `setFlyBehavior()`
 - ▶ Développer le canard `RedHeadDuck` (`RedHeadDuck.java`)
 - ▶ Implanter le nouveau comportement "vole-force-fusée" `FlyRocketPowered` (`FlyRocketPowered.java`)
 - ▶ Tester le nouveau canard dans un main `RedHeadSim.java`
 - ▶ Changer le comportement "voler" de *FlyWithWings* à *FlyRocketPowered*. Penser à utiliser le setter afin d'obtenir ces deux affichages: "Je peux voler" & "Je vole comme une fusée"
- ▶ Donner (et ne pas implémenter) les modifications à faire afin d'ajouter le comportement manger: `eat()`

SimUDuck :

La conception finale (20/23)



SimUDuck :

Composition / Héritage (21 / 23)

- ▶ Has-a: liaison intéressante
 - ▶ Chaque canard possède un FlyBehavior et QuackBehavior qui délègue flying et quacking
 - ▶ Composition (mettre les 2 classes ensemble)
 - ▶ Encapsuler une famille d'algorithmes dans leur propre ensemble de classes
 - ▶ Remplacer l'héritage pour favoriser le changement dynamique du comportement

▶ **Règle 3:** Favoriser la composition sur l'héritage

SimUDuck :

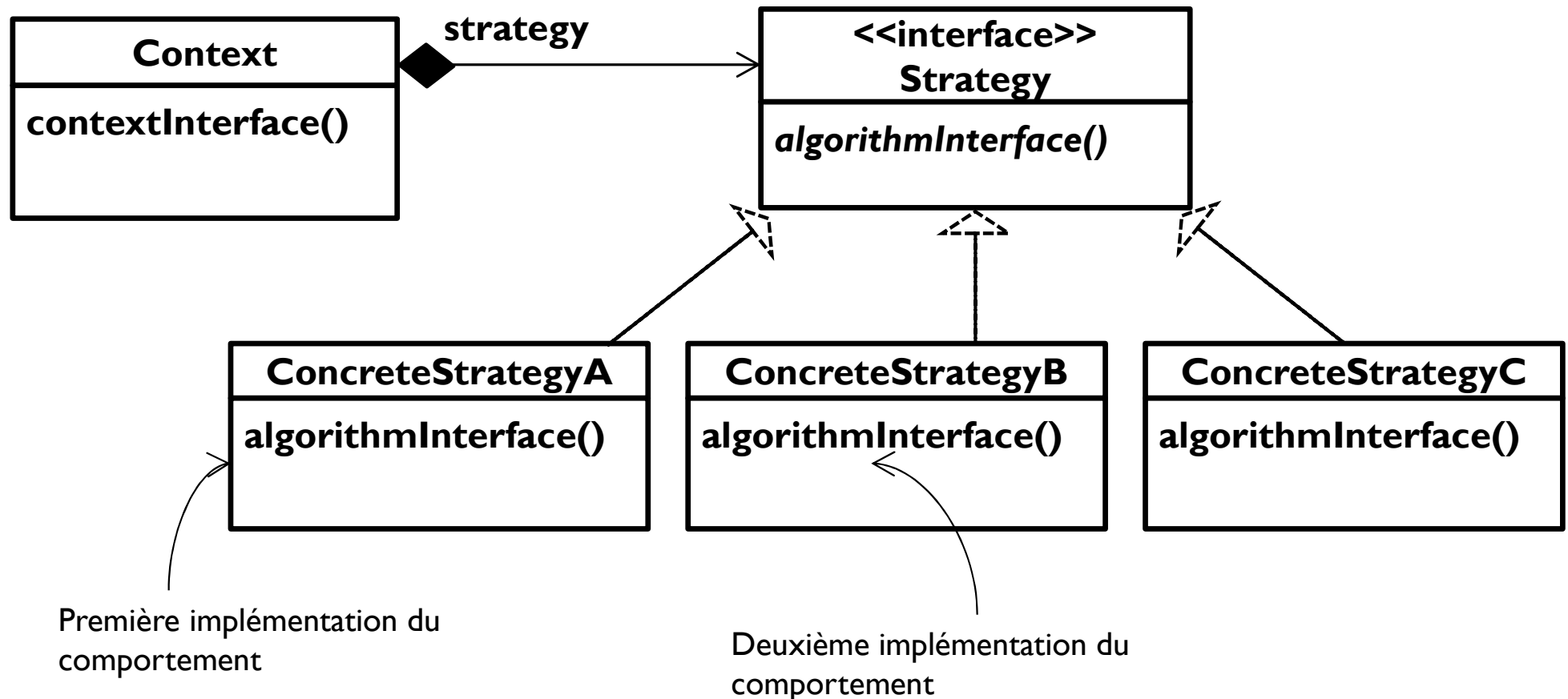
Notre premier patron (22/23)

- ▶ Notre premier patron: **STRATEGIE**

Le patron stratégie cherche principalement à séparer un objet de ses comportements/algorithmes en encapsulant ces derniers dans des classes à part.

Pour ce faire, on doit alors définir une famille de comportements ou d'algorithmes encapsulés et interchangeables.

Diagramme de classes du patron (23/23)



Récapitulatif (1 / 2)

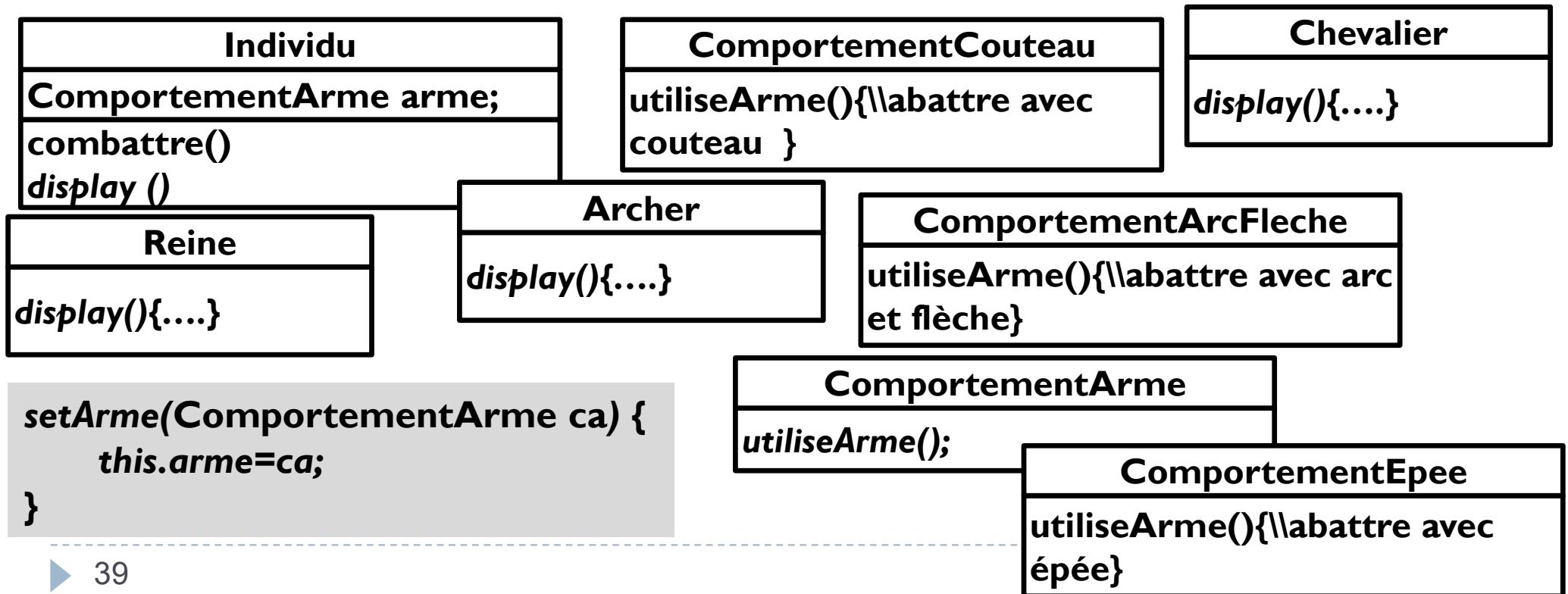
- ▶ Bases de l'OO:
 - ▶ Abstraction, Encapsulation, Polymorphisme & Héritage
- ▶ Principes de l'OO
 - ▶ Encapsuler ce qui varie
 - ▶ Favoriser la composition sur l'héritage
 - ▶ Programmer avec des interfaces et non des implémentations
- ▶ Patron de l'OO (stratégie)
 - ▶ Le patron stratégie définit une famille d'algorithmes, les encapsule, et les rend interchangeable. Ce patron laisse l'algorithme varier indépendamment du client qu'il l'utilise.

Récapitulatif (2/2)

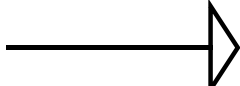
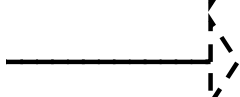
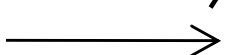
- ▶ Connaître les bases de l'OO ne fait pas de toi un bon concepteur
- ▶ Les meilleurs conception OO sont réutilisable, flexible et maintenable
- ▶ Les patrons nous guident à construire des systèmes avec les bonnes qualités de la conception OO
- ▶ Les patrons sont des expériences prouvées dans l'OO
- ▶ Les patrons ne donnent pas de code, mais plutôt des solutions générales pour des problèmes de conception
- ▶ Les patrons ne sont pas inventés, mais découverts
- ▶ La majorité des patrons et des principes adressent les problèmes de changement dans le logiciel
- ▶ On essaie toujours de prendre ce qui varie des systèmes et on l'encapsule
- ▶ Les patrons offrent un langage partagé qui peut maximiser la valeur de la communication avec les autres développeurs

Exercice (1/2)

- Ci-dessous, on donne l'ensemble de classes et interfaces d'un jeu d'action et d'aventure. Il y a des classes d'individus avec des classes pour les comportements d'armes que les individus peuvent utiliser. Chaque individu peut utiliser une seule arme à la fois, mais peut la changer à tout moment durant le jeu. La tâche demandée est d'ordonner le tout.



Exercice (2 / 2)

1. Arranger les classes
2. Identifier les classes, les classes abstraites des interfaces
3. Relier les entités pas des flèches ou:
 1.  représente **extends**
 2.  représente **implements**
 3.  représente **has-a**
4. Mettre la méthode `setArme()` dans la classe correspondante
5. Implémenter et tester cette conception dans un main.
Penser à changer dynamiquement le comportement de l'archer après avoir finir ces arcs.