Formation Clean code 2

Bad smells et refactoring

Introduction

- **Module consacré à la pratique du CleanCode**
  - Un petit livret vous est fourni
    - sur les mauvaises pratiques (dites bad smells)
    - les refactoring les plus utiles
    - Les recommandations CleanCode
  - Objectifs de cette journée :
    - Connaître les principales mauvaises pratiques de codage
    - Les identifier rapidement
    - Les solutions pour les corriger

Contenu de la formation

- Rappel du module précédent
- Présentation de la notion de bad smells
- Les principaux bad smells
- Comment smell, large methods, data clumps, primitive obsession, incomplete library, code spaghetti, middle man, message chains …
- Techniques de refactoring

Organisation de la journée

Matin	Après midi
Que sont les « code smell »	Code spaghetti
Bad smell	Change preventor

Organisation de la formation

- Journée 1 : Clean Code (nommage et codage)
- **Journée 2 : Bad smell et refactoring**
- Journée 3 : Design (patterns) d'applications, métriques, et patterns d'entreprise
- Journée 4 : TDD, Testabilité de son code
- Journée 5 : Audit sur une application réelle.

RAPPEL SESSIONS PRECEDENTES

Questions

- **Dette technique ? Origine ? Gain à diminuer ?**
- **Comment reconnaitre une méthode à refactorer ?**
- **Comment refactorer une méthode ?**

Rappel

Recommandation pour le nommage

- **Utilisez un nom qui révèle vos intentions**
- **Évitez la désinformations**
- **Distinguez vos variables correctement**
- **Utilisez des noms prononçables**
- **Utilisez des termes standard**

Rappel

Coder une fonction/méthode

- Petites méthodes
- Faites une seule chose
- 1 niveau d'abstraction par méthode
- Limitez le nombre de paramètre par méthodes
- Évitez les effets de bords
- Préférez les exceptions au code d'erreur
- Ne vous répétez pas

design pattern

- visitor
- Chain of responsability
- Stratégie

Métriques

- **Complexité cyclomatique**

- **Complexité NPath**

BAD SMELL

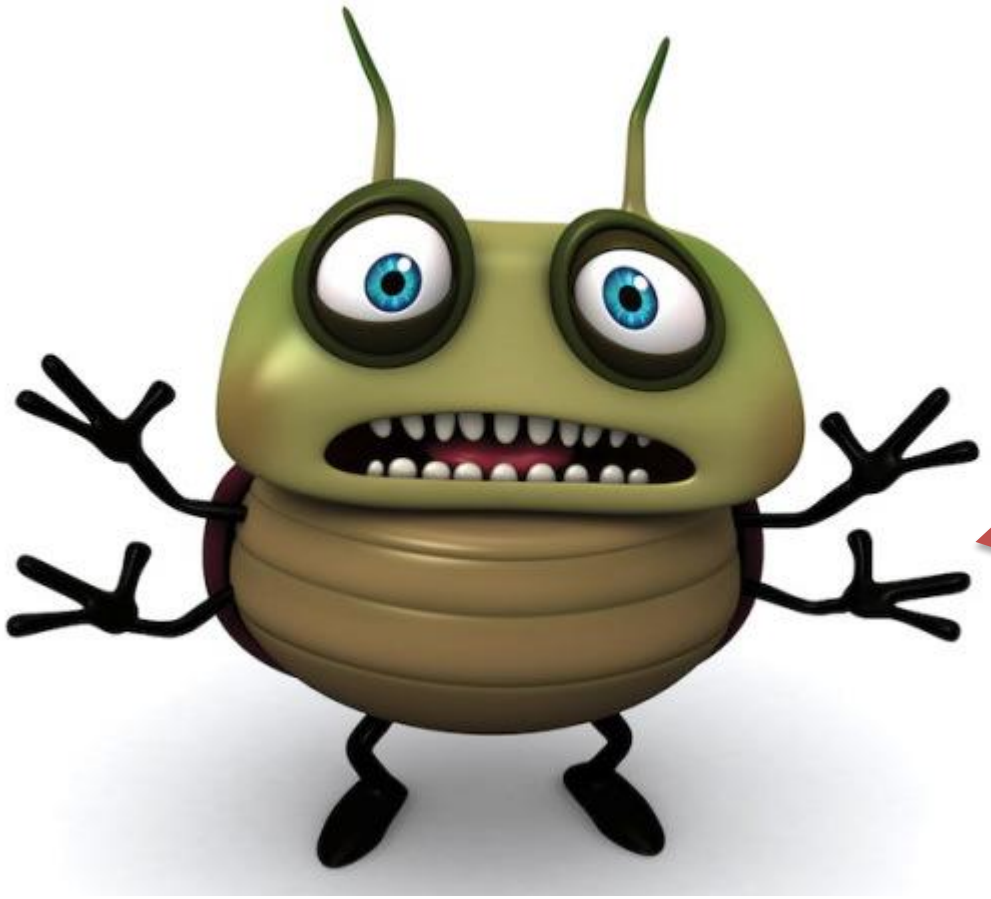
If it stinks, change it

tocea

Introduction

Il y a un passé pas si lointain

Introduction



Ceci est un bug totalitaire qui soumet les applications à ses désirs de plantage

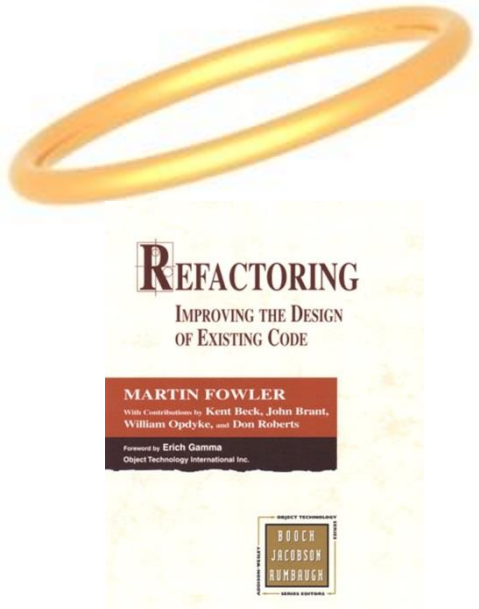
Le monde était soumis à la terrible dictature des bugs

Introduction

Une unité d'experts du domaine logiciels a eu pour mission secrète de les étudier

tocea

Introduction



Ils établirent une catégorisation de ces bugs, leur hiérarchie,

leurs impacts et les moyens de les éradiquer

Introduction

La guerre ne fut pas gagnée immédiatement mais

des progrès apparurent sur les différents fronts

Introduction

De nouvelles armes furent conçus pour la détection

Et l'élimination de ces bugs

Qui est Martin Fowler ?

- **Auteur, conférencier, informaticien et consultant britannique dans la conception de logiciels d'entreprise. Ses centres d'intérêt principaux sont la programmation orientée objet, la refactorisation (refactoring), les patrons de conception (design patterns), UML et les méthodes de programmation agile où il est un pionnier et une référence. (*wikipedia*)**
- **Principaux liens :**
  - http://www.refactoring.com
  - http://www.martinfowler.com

tocea

Bad smell : comment les identifier ?

- **Code smell : Un « mauvais » code caractéristique qui permet d'y associer un ensemble de solutions prédéfinies (appelée Refactoring)**
- **Dans le jargon :**
  - Code spaghetti
  - Code fourre-tout
  - Classe de la mort qui tue
  - Code que je redonne à mon voisin
  - Code non testable

Bad smell : Catégorisation

- Ces « bad smell » sont généralement répartis en 5 catégories
- → Problème de complexité
- → problème de design
- → problème de taille
- → code inutile
- → problème de couplage

BAD SMELL : COMMENT SMELL

Traquer et identifier les codes nauséabonds

Comments smell

« C'est comme le déodorant,

Cela masque les odeurs mais cela n'évite pas la douche »

- **Indicateur d'une méthode trop complexe**
- **Expliquer le « Pourquoi » et non le « Comment »**

Exemple : cas réel

```
… } else {
        // thierry le 05/02/2003 - gestion saisie destinataire etranger
        tmpLoc = getAdrVille();
        reponse = true; // a completer lors de la gestion de l'international
}
if (!reponse) {
        log.ecrire( " Localite inexistante ", "|" + getAdrPays() + "|" + getAdrCodepostal() + "|" +
        getAdrVille().toUpperCase().trim() +"|" + "(" + tmpLoc + ")", "I" );
} else {
 // localités étrangères A VOIR PLUS TARD
}
setAdrVille( tmpLoc );
```

Exemple : Corrections possibles ?

```
// This function will calculate the technical debt with the parameters,
// the parameters should be good according to some rules.
public  void myFunction1(parameters) {

                // It will init the calcul to get started.

                // It will calculate the approximative techical debt.

                // add some additional measure to the technical debt.

}
```

Refactorings possibles

- **Renommer la méthode**
  - public void calculateTechnicalDebt(parameters)
- **Extraire une ou plusieurs méthodes**

```
public double calculateTechDebt(parameters) {

        double techTebt = initializeTechDebtValue(parameters);
         techDebt += calculateApproximativeTechDebt(parameters);
        return calculateAGoodtechDebt(techDebt, parameters);

 }
```

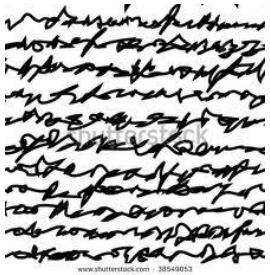
BAD SMELL : LONG METHODS

Combien d'écrans de VT 100 pour afficher cette méthode ?

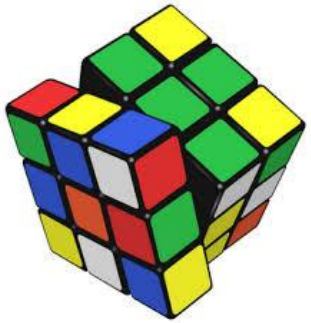
tocea

Long methods

- Lisibilité



- Complexité



- Trop de rôles

- Code dupliqué

Refactorings vu

- **Rappel des refactorings possibles vu :**
  - **Extraire des méthodes**
  - **Utiliser des « parameters Objects »**
  - **Utiliser le pattern stratégie**

Exercice

- D’autre refactoring possible ?

Autre réfactoring : Décomposer les conditions

```
if (date.before (SUMMER_START) || date.after(SUMMER_END)) {
        charge = quantity * _winterRate + _winterServiceCharge;
} else {
        charge = quantity * _summerRate;
}
```

```
if (notSummer(date)) {
        charge = winterCharge(quantity);
} else {
        charge = summerCharge (quantity);
}
```

Autre refactoring : ne pas décomposer les objets

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

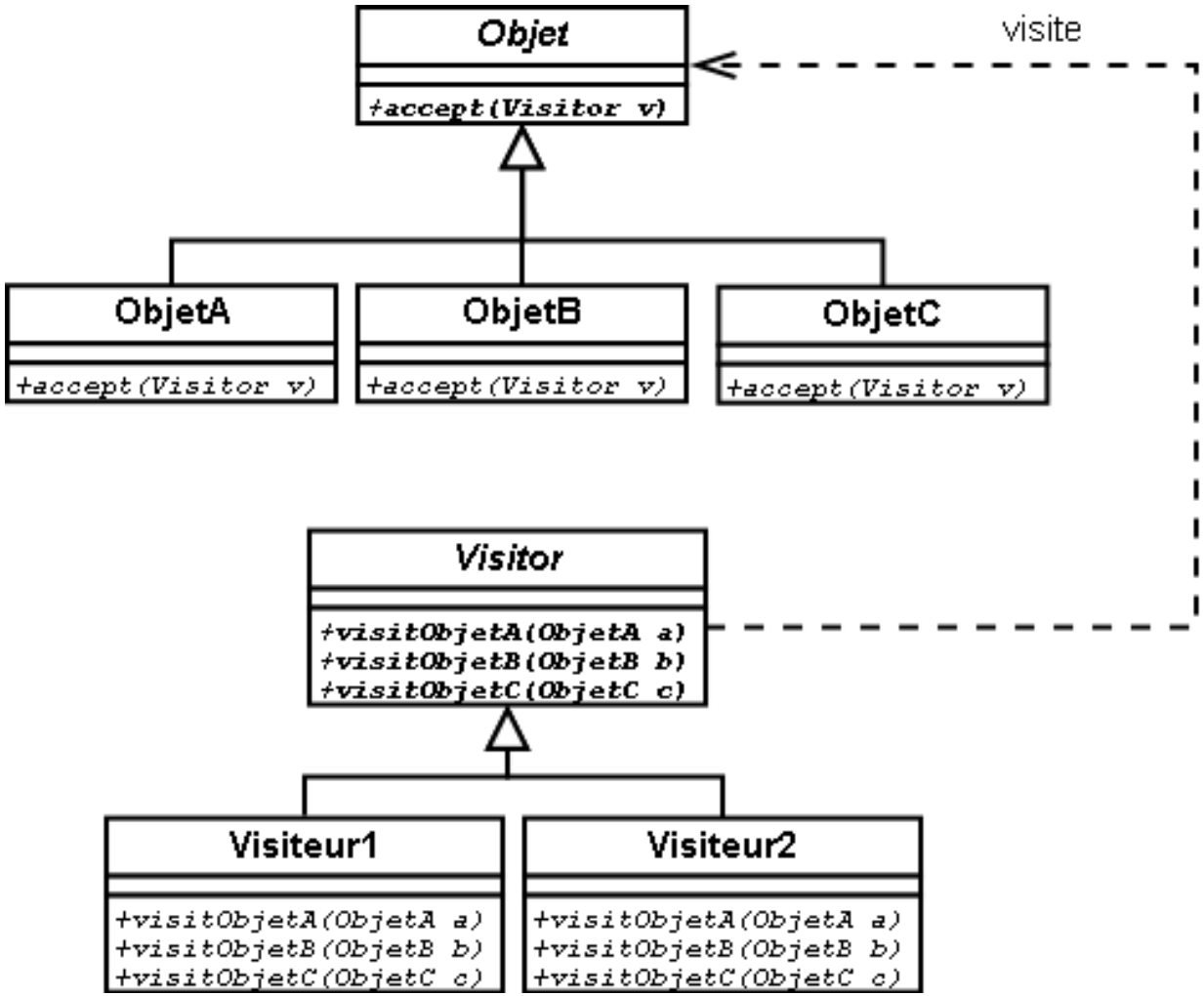
```
withinPlan = plan.withinRange(daysTempRange());
```

Autre refactoring : Utiliser le pattern visiteur

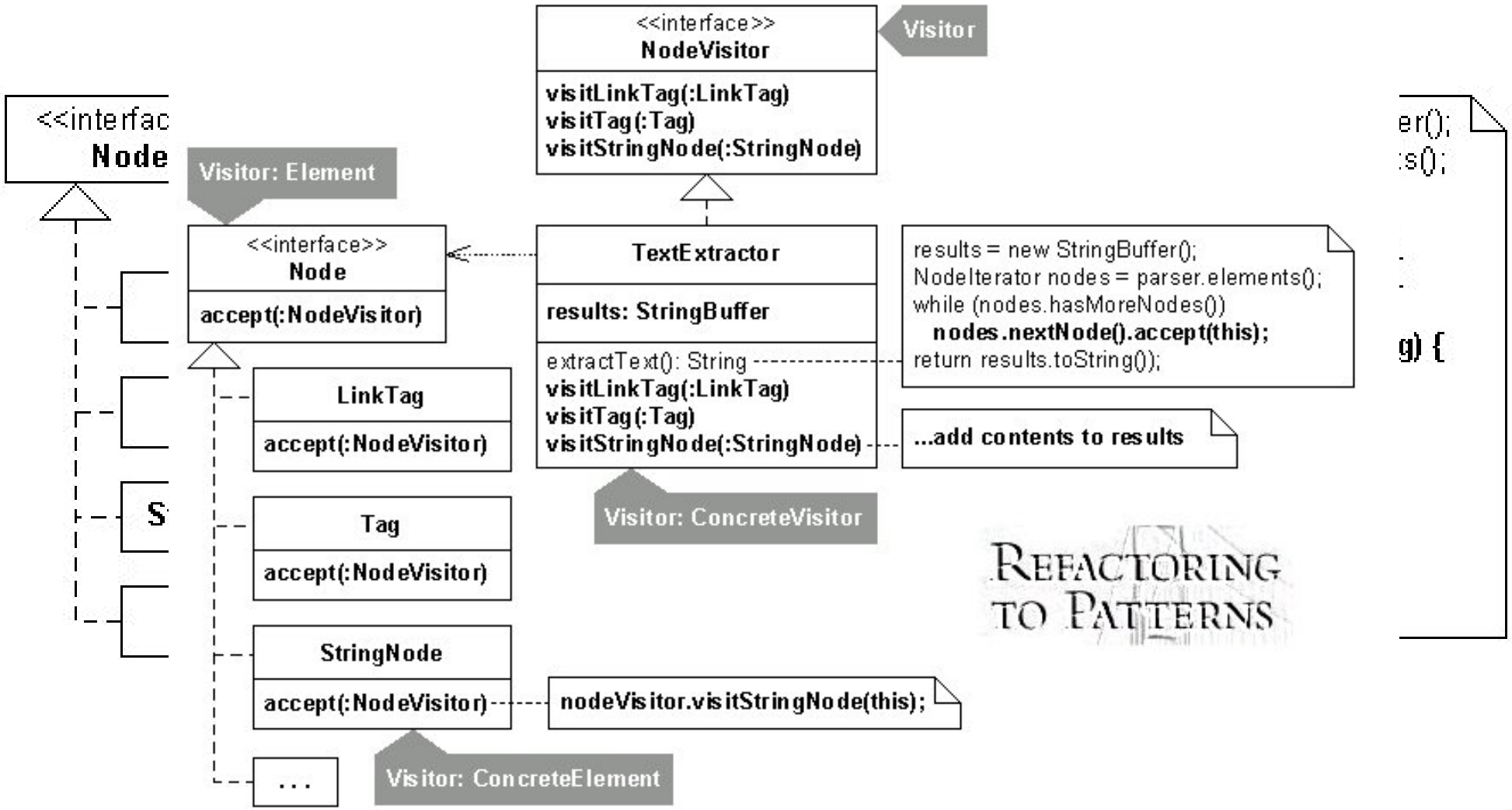
- Crée une sorte de gestion de plugin pour les méthodes
- De cette manière, les méthodes peuvent être ajoutées au runtime

tocea

Le pattern visiteur



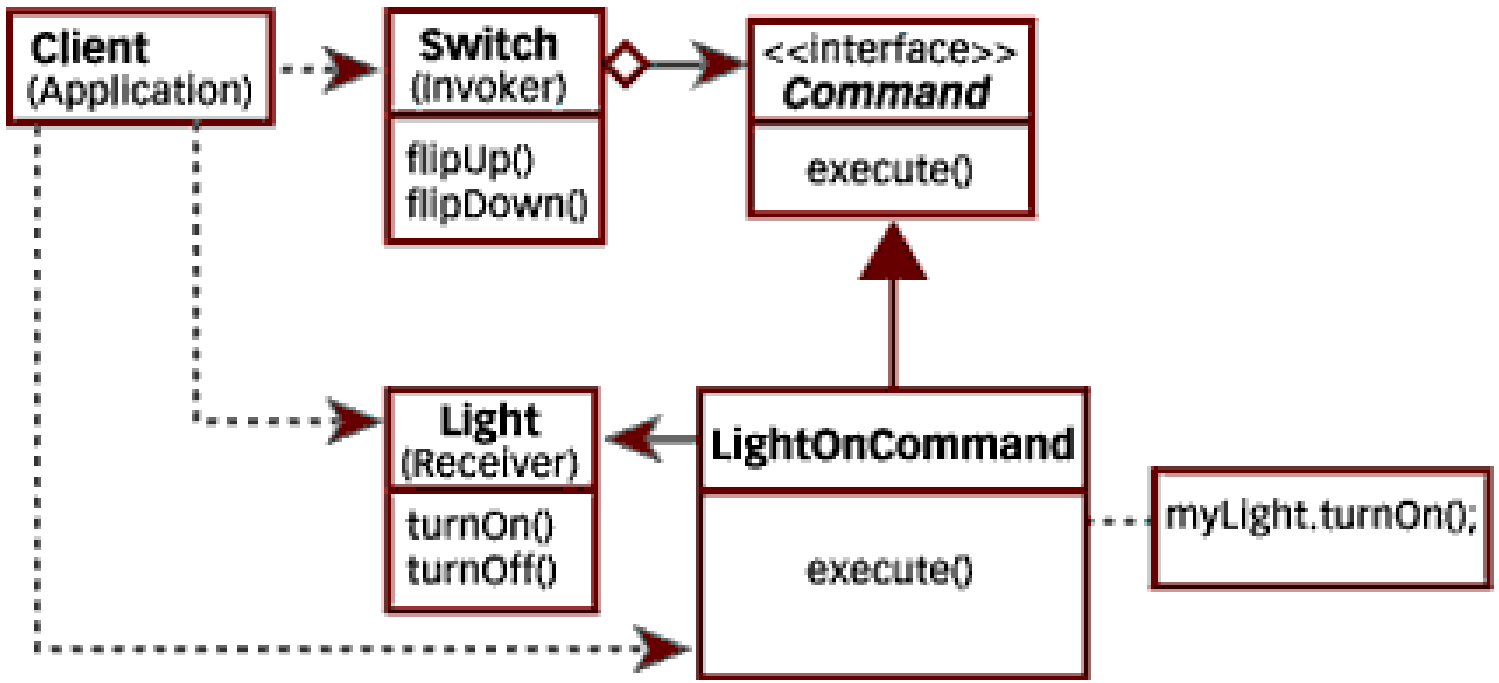
Refactorer avec le pattern visiteur



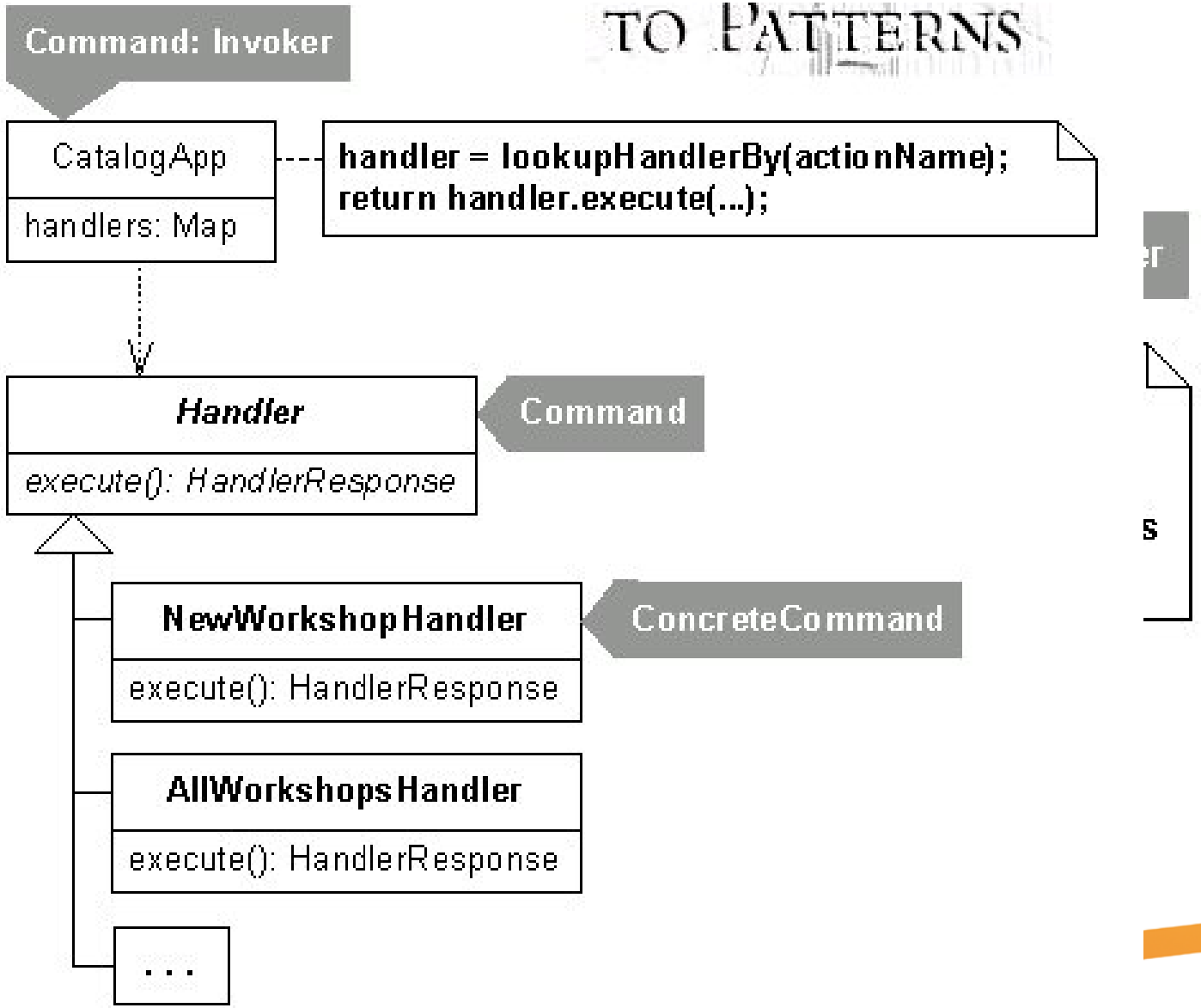
Autre réfactoring : Utiliser le pattern commande

- **Encapsule la requête d'une commande dans un objet**

Le pattern Commande



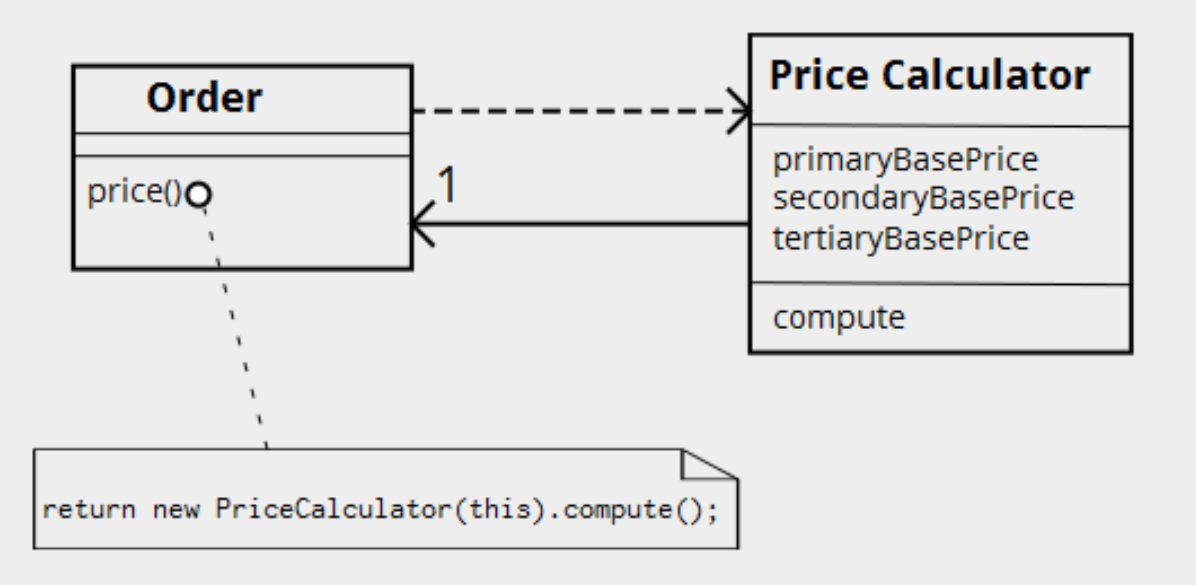
Refactorer avec le pattern command



Autres refactoring possibles

Utiliser les méthodes par des « methods Objects »

```
class Order {
        ..
        double price() {
                double primaryBasePrice;
                double secondaryBasePrice;
                double tertiaryBasePrice; // long computation;
                ...
        }
```



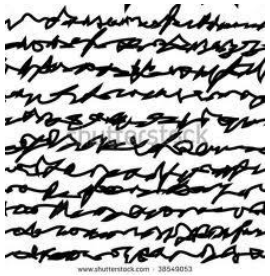
BAD SMELL : LARGES CLASSES

80% de l'application dans une classe

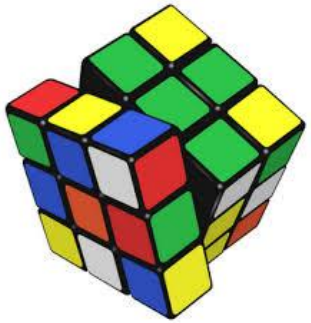
Une génération entière de développeurs sacrifiée à la tester

Larges classes

- **Lisibilité**



- **Complexité**



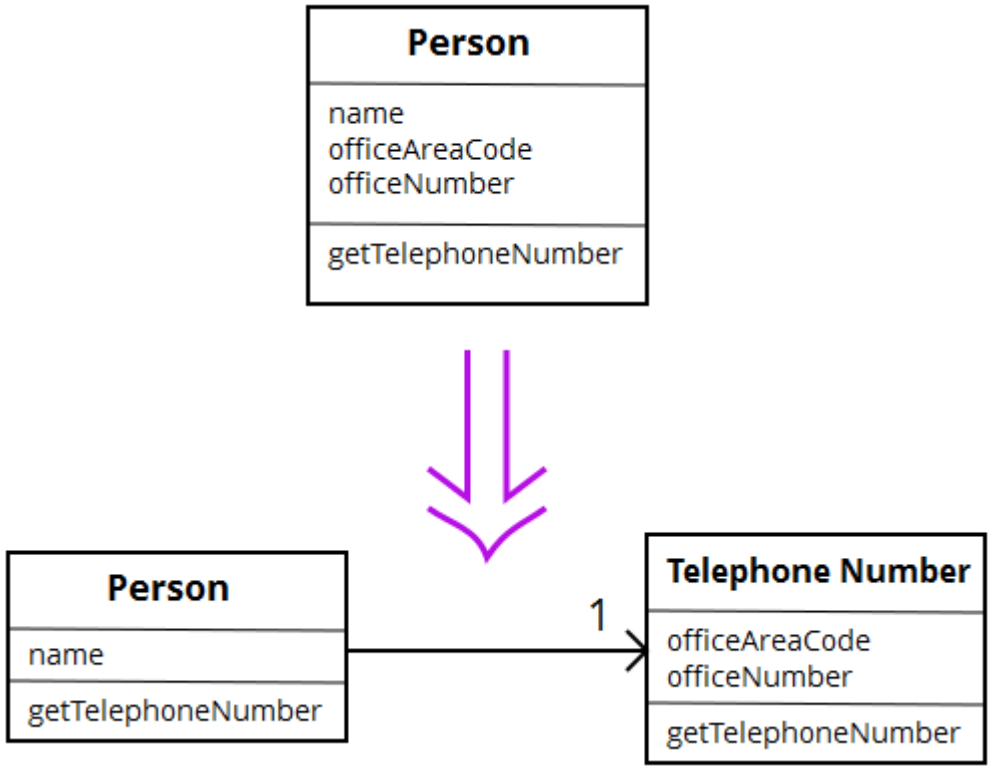
- **Trop de rôles**

Exercices

- **Quels sont les différents refactorings possibles ?**

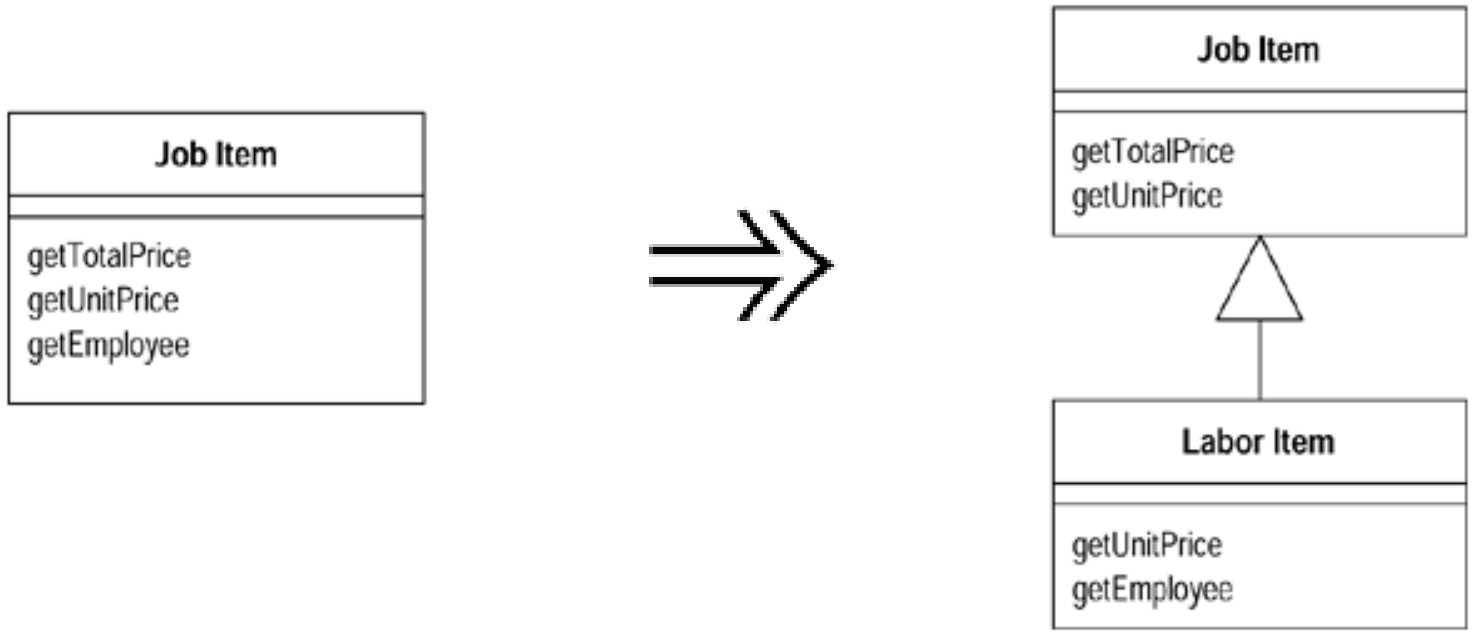
Refactorings possibles : extraire une classe

- **Créer une nouvelle classe et déplacer certains attributs et méthodes dans la nouvelle classes**

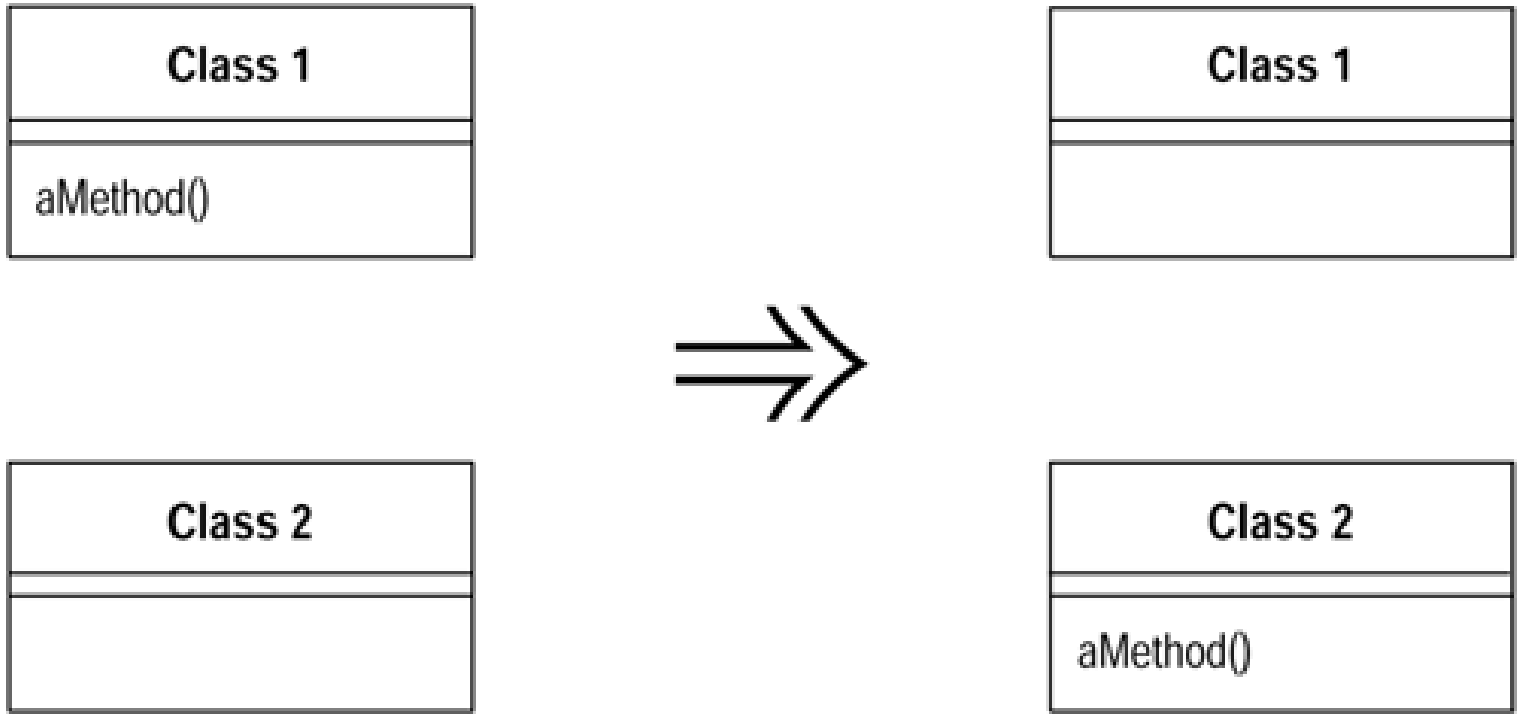


Refactorings possibles : extraire une classe interne

- **Créer une nouvelle classe interne et déplacer certains attributs et méthodes dans la nouvelle classes**
- **Cette classe interne n'est utilisée que par la classe englobante**



Refactorings possibles : Déplacement de méthodes



Autre refactoring

- Tout les refactoring de « long methods »

BAD SMELL : DATA CLUMP

Imprimer.salut(tous,les,arguments,de,l,univers)

Exemple

```
Public void dessinerPoint(int x, int y, int z)
```

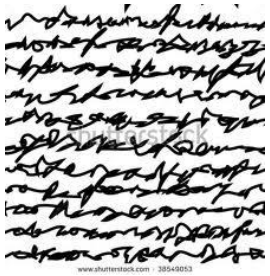
```
Public void dessinerTrait(int x1, int y1, int z1, int x2, int y2, int z2)
```

```
Public int calculateDistance(int x1, int y1, int z1, int x2, int y2, int z2)
```

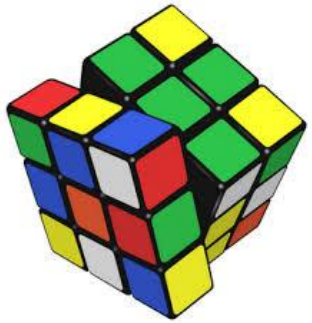
Refactoring ?

Impact

- **Lisibilité**

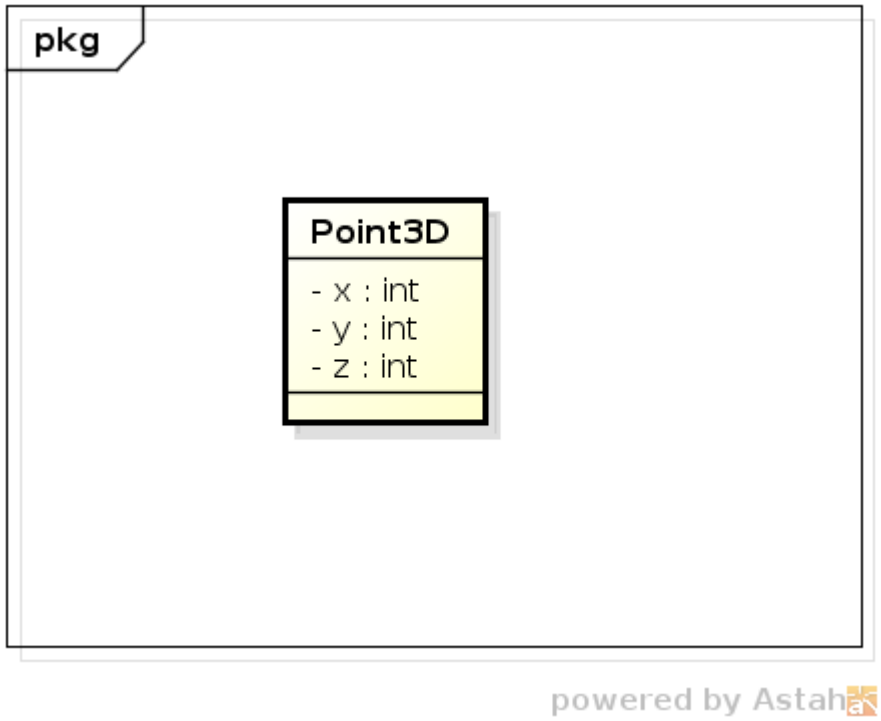


- **Complexité**



Refactoring

- Créer un objet donnée rassemblant ces informations




```
Public void dessinerPoint(Point3D p)
```

```
Public void dessinerTrait(Point3D p1, Point3D p2)
```

```
Public int calculateDistance(Point3D p1, Point3D p2)
```

BAD SMELL : DATA CLASS

I can do nothing

Principe

- **Classe ne contenant que des données**
- **Pas de métier**

- **Indique**
  - Classe exhibitionniste
  - Trop grande séparation donnée/métier

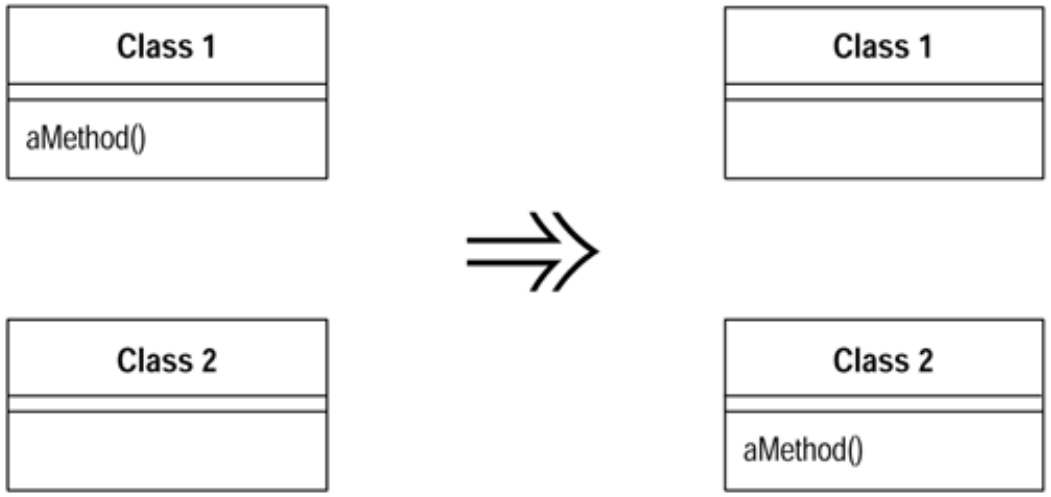
TP

Refactoring possible : encapsulation

- **Dans le cas d'attributs publique.**
- **→ encapsuler les attributs (getter/setters)**
- **→ encapsuler les collections**

Refactoring possible : Déplacer une méthode

- « Si je peux travailler, laisser moi travailler ! »
- Séparation donnée/métier non justifiée
- 1 méthode M correspond à un métier d'une classe A, mais fut écrite dans une classe B.
- → voir « feature envy »
- → déplacer cette méthode dans la « data class »



BAD SMELL : PRIMITIVE OBSESSION

Le régime Dukan des développeurs

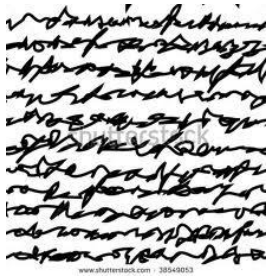
Principe

- **Utilisation de type simple du JDK à la place de structure appropriée**

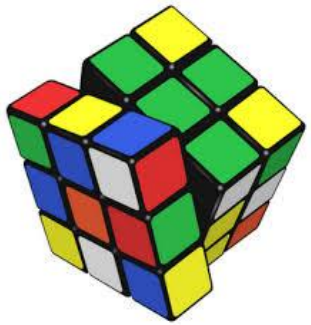
```
Map<String, Integer> citiesPopulations = new HashMap<String, Integer>();

…

citiesPopulations.put(« Paris », 35000000);
citiesPopulations.put(« L ondon », 35000000);
```

Impact

- Lisibilité



- Complexité



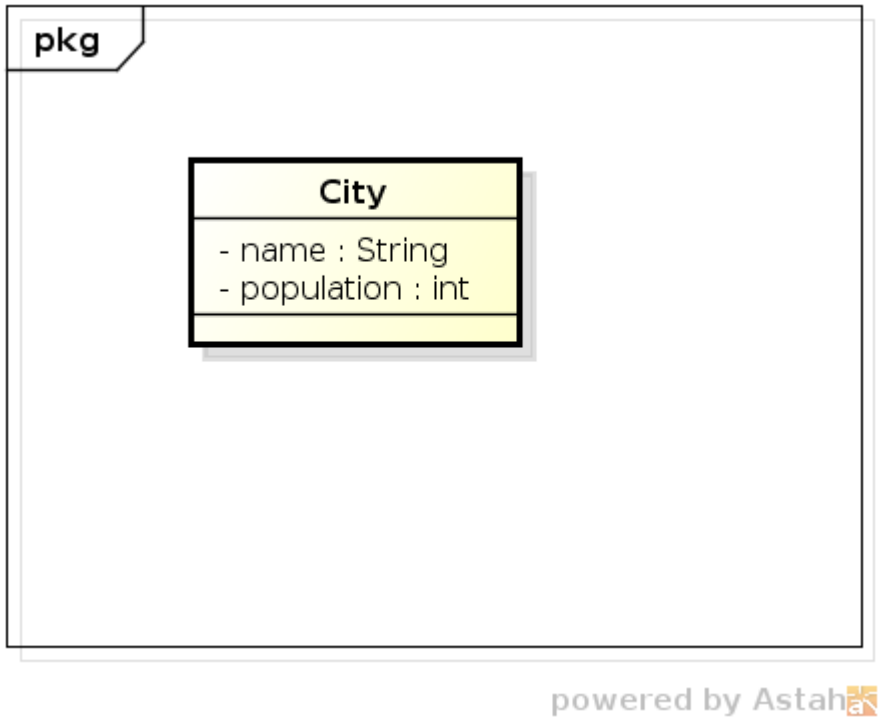
- Évolution

- Abstraction

tocea

Factoring possible

- **Créer une classe correspondant à cette structure**



BAD SMELL : INCOMPLETE LIBRARY

Il y a eu comme un trou dans le budget

Principe

Utiliser une bibliothèque

- Il manque une méthode

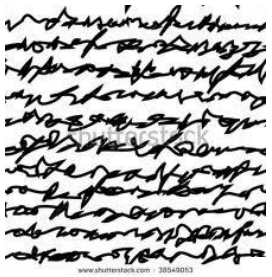
- → implémenter cette méthode
  - Ex : java.util.Date
  - Pour une date, avoir le jour suivant
  - Écrire localement la solution

```
Date newStart = new Date (previousEnd.getYear(), previousEnd.getMonth(),
previousEnd.getDate() + 1);
```

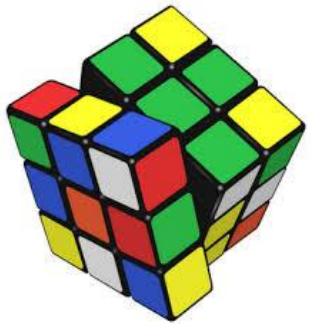
incomplete lib

Impact

- Lisibilité



- Complexité



- Code dupliqué

Exercice

```
Date newStart = new Date (previousEnd.getYear(), previousEnd.getMonth(),
previousEnd.getDate() + 1);
```

incomplete

lib

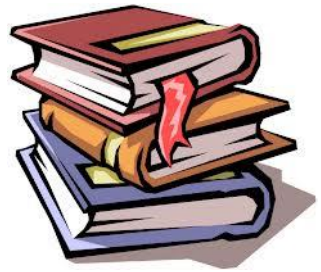
Refactoring Possible

- **Externaliser dans une méthode externe**

```
Date newStart = nextDay(previousEnd);
```

Projet

```
private static Date nextDay(Date arg) {
                return new Date (arg.getYear(),arg.getMonth(), arg.getDate() + 1);
 }
```

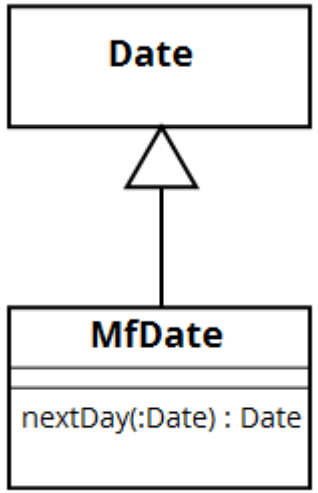


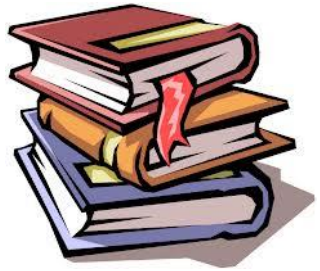
Bibliothèque interne
À l'entreprise

tocea

Refactoring Possible

- **Utiliser l'héritage**





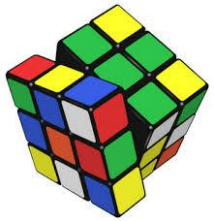
Bibliothèque interne
À l'entreprise

CODE SPAGHETTI

tocea

Code Spaghetti ?

- **Anti-patterns qui correspondent à des problèmes de couplages**
- **Dépendances trop fortes entre les méthodes et/ou classes**
- **Impact :**
  - Lisibilité
  - Complexité
  - Évolutivité



CODE SPAGHETTI : MESSAGE CHAINS

Si tu n’envoies pas cette cassette à au moins 3 de tes amis sous 72 heures, tu es mort.

Principe

- Utiliser une chaine d'appel

```
Person p = …;
Manager manager = p.getDepartment().getManager();
```

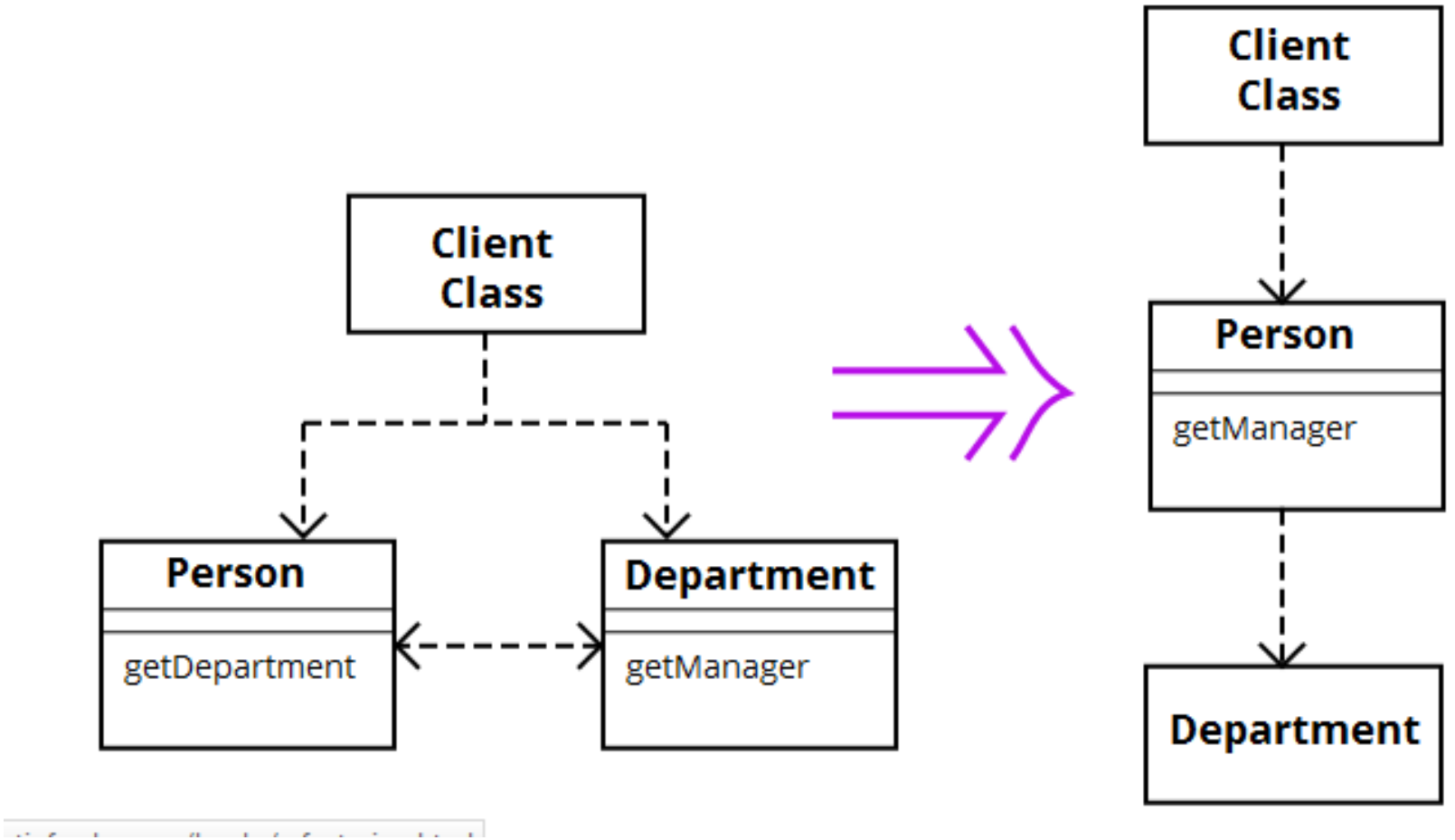
- Relation forte entre les classes
- Oblige l'utilisateur à connaitre la structure de l'application
- Code Complexe
- Peu évolutif

La loi de Demeter

- « Ne parle qu'a ton voisin »
- On n'a pas à connaitre la structure interne d'une application pour pouvoir l'utiliser

tocea

Refactoring possible : Cacher la délégation



Refactoring possible : Extraire dans une méthode

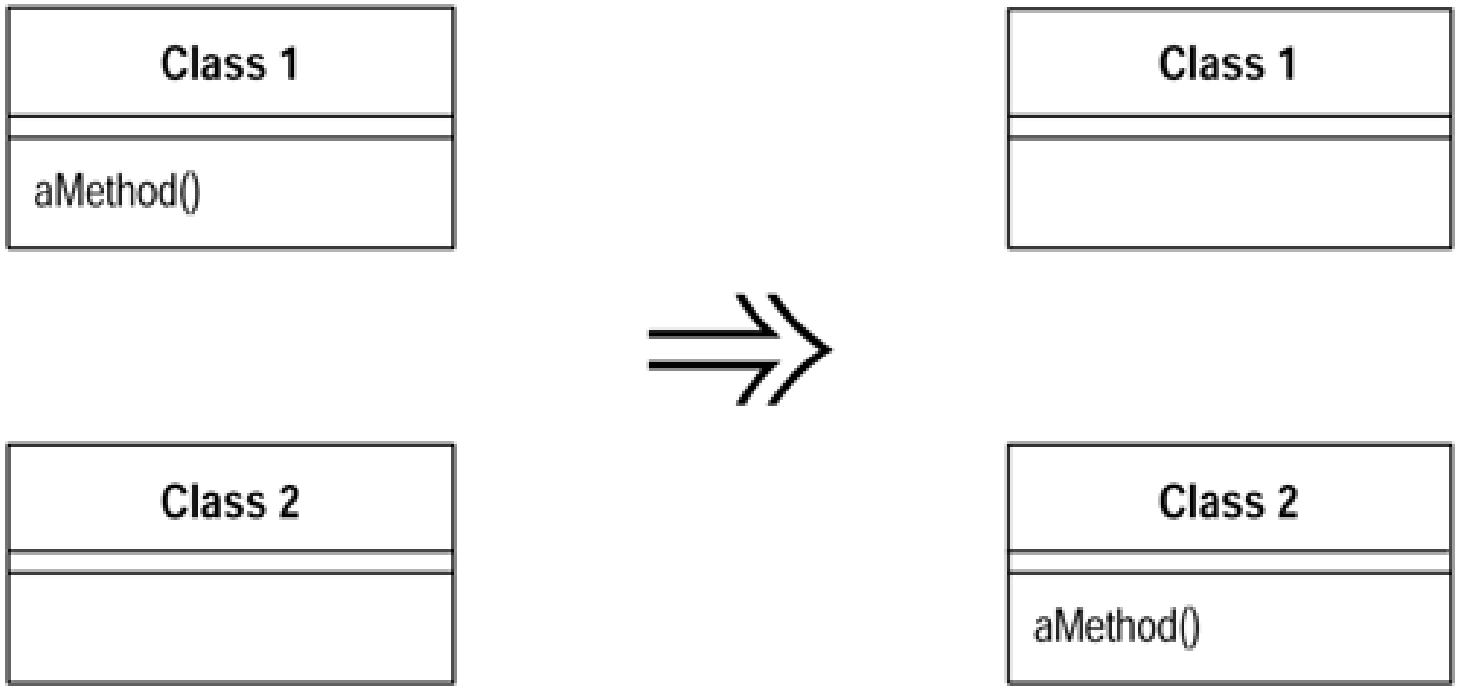
Extraire une partie de ce code complexe dans une méthode.

```
Public Departement getDepartement() {
        Person p = …;
        Manager manager = p.getDepartment().getManager();
}
```

```
getDepartment().getManager();
```

Refactoring possible : Déplacer la méthode

- **Déplacer la méthode dans une classe plus accessible**



CODE SPAGHETTI : LE MIDDLE MAN

L’intermédiaire entre votre argent et vous : le banquier

Principe

- VS Message chains
- Abus de la délégation
- « Si une classe se contente de déléguer tout son métier, alors pourquoi existe-t-elle ? »
- → Classe centrale
- → Vitale
- → Couplage très fort

Refactoring possible : Supprimer le middle man

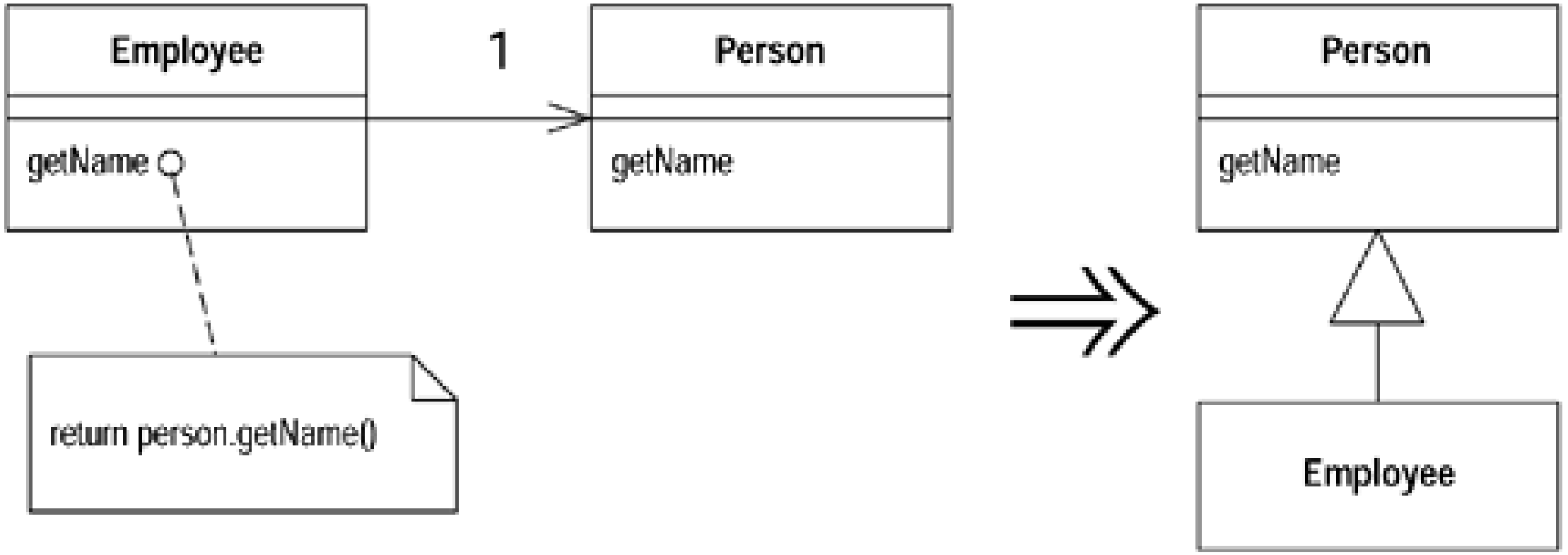
Refactoring possible : Inline methode

- VS extract methods
- Supprimer une méthode ce ces appels

```
int getRating() {
  return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
  return _numberOfLateDeliveries > 5;
}
```

```
int getRating() {
  return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Refactoring possible : utiliser l'héritage



CODE SPAGHETTI : FEATURE ENVY

D’où l’utilité d’avoir une société de nettoyage offshore

(Aussi appelée , Aides-toi et le ciel t’aidera)

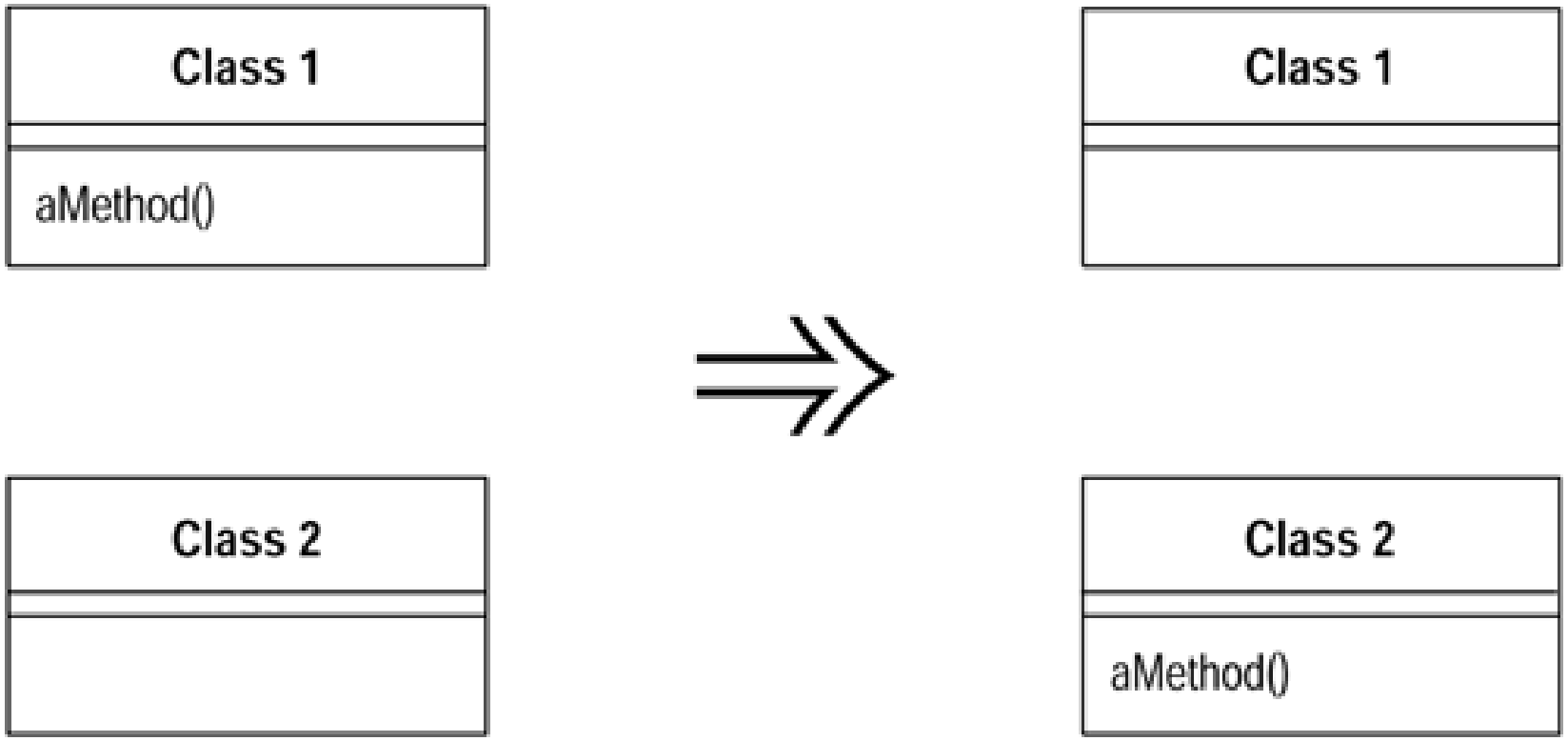
Principe

- Quand une méthode est plus intéressée par une autre classe que par sa propre classe

- Indique souvent que cette méthode n'est pas placée au bon endroit

TP

Refactroring possible : déplacer la méthode



Refactoring possible : Extraire une partie dans une méthode

- Parfois seule une partie de la méthode souffre de cet anti-pattern
- → Extraire ce morceau de code dans une méthode
- → Déplacer cette méthode dans la classe « enviée »

CHANGE PREVENTOR

Change preventor ?

- Anti-patterns qui empêchent une évolution simple de l'application
- Impact :
  - Complexité

  - Évolutivité

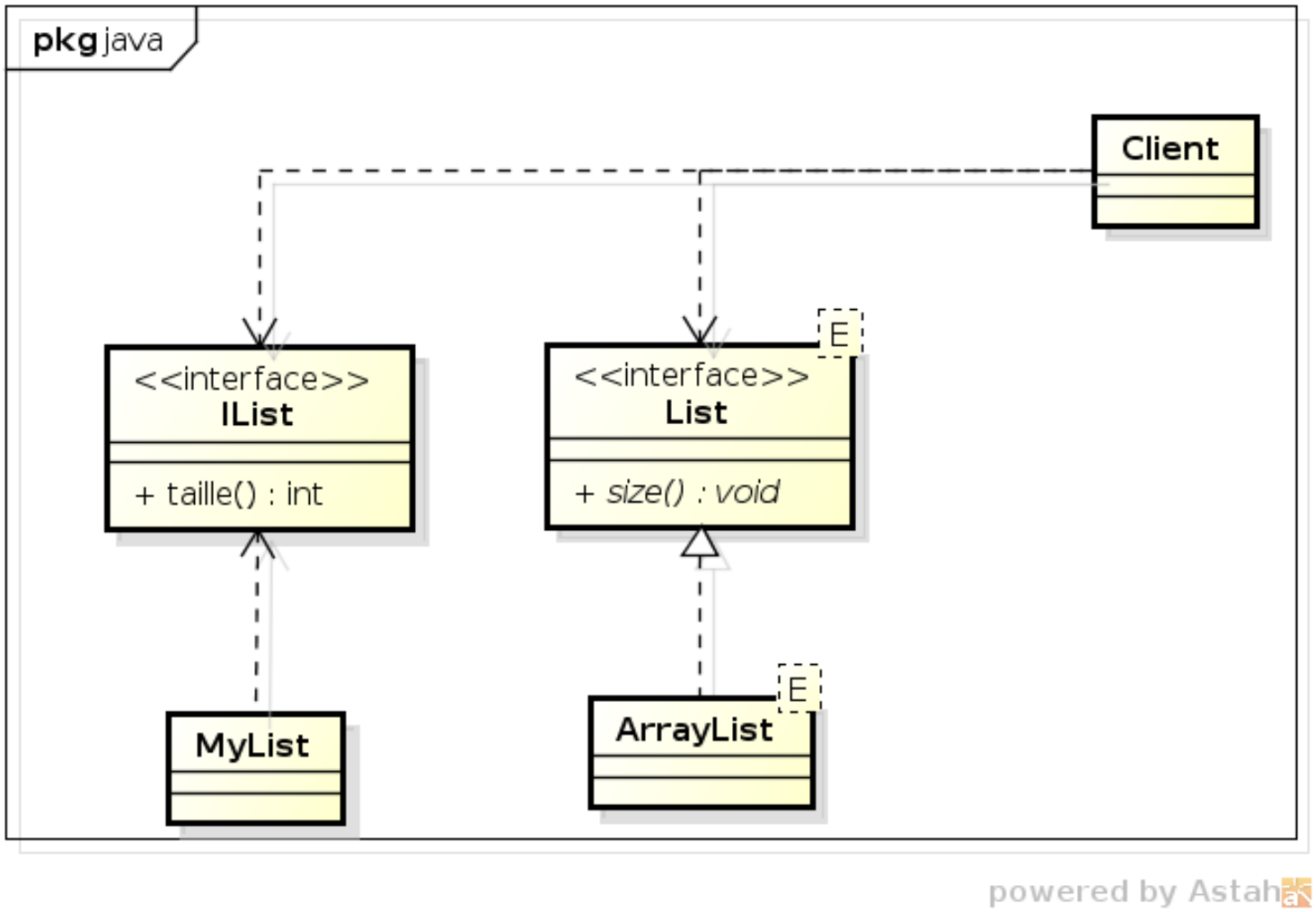
CHANGE PREVENTOR : ALTERNATIVE CLASSES WITH DIFFERENT INTERFACES

Le darwinisme appliqué au code de deux packages différents

Principe

- Quand 2 méthodes font globalement la même choses, mais ont des signatures et interfaces différentes
- Peut-on unifier leurs interface ?
- Cause d'un code dupliqué ?

Exemple


pkg java
Client
<<interface>>
IList
+ taille() : int
<<interface>>
List
E
+ size() : void
MyList
ArrayList
E
powered by Astah

TP

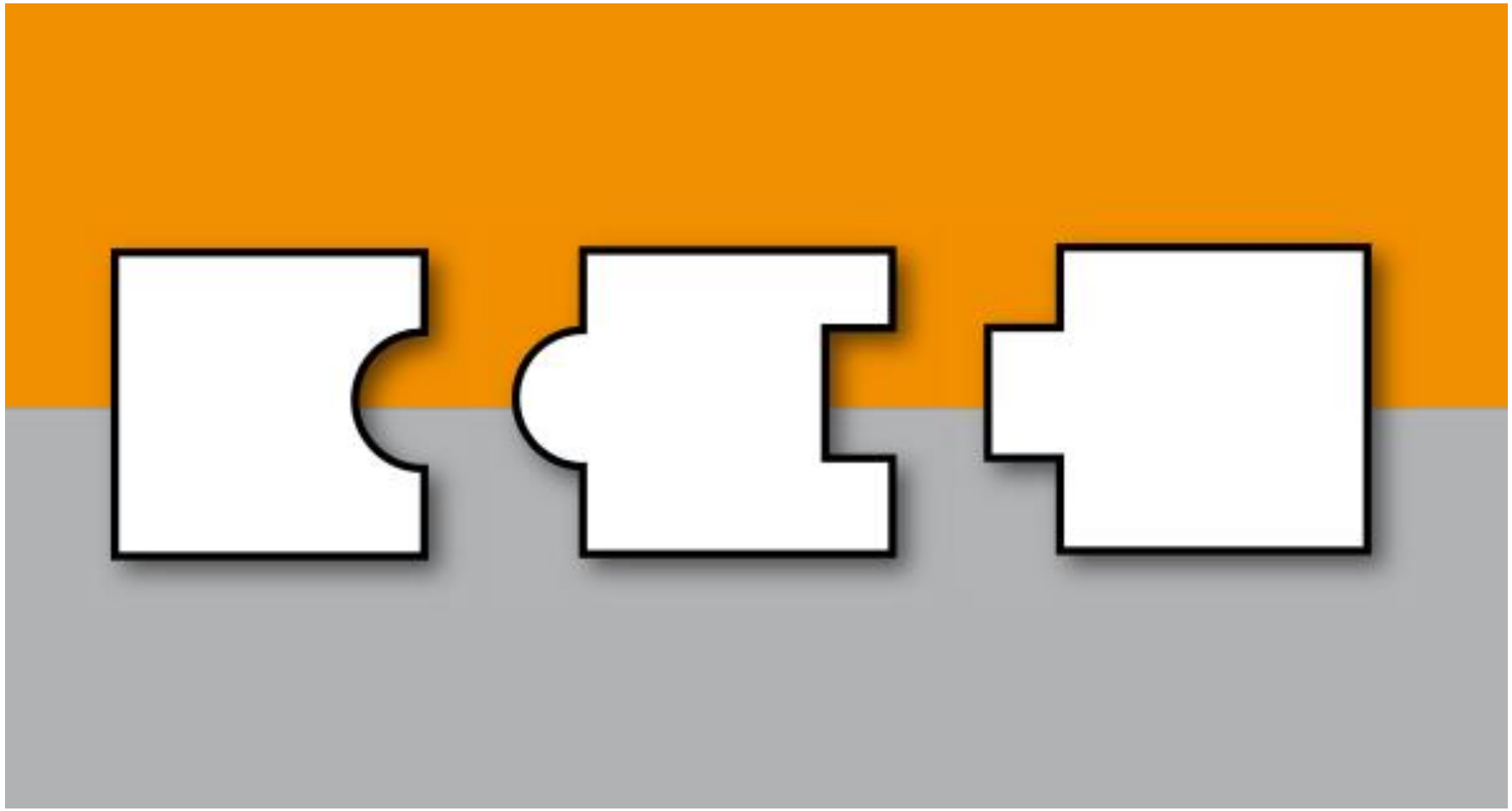
Refactoring possible : Supprimer l'alternative

- **Si 2 classes font globalement la même choses,**
- **Les 2 classes doivent-elles coexister ?**

Refactoring possible : Renommer la méthode

- **Renommer les méthodes de façon à obtenir les mêmes signatures**

Refactoring possible : Utiliser le pattern Adapter

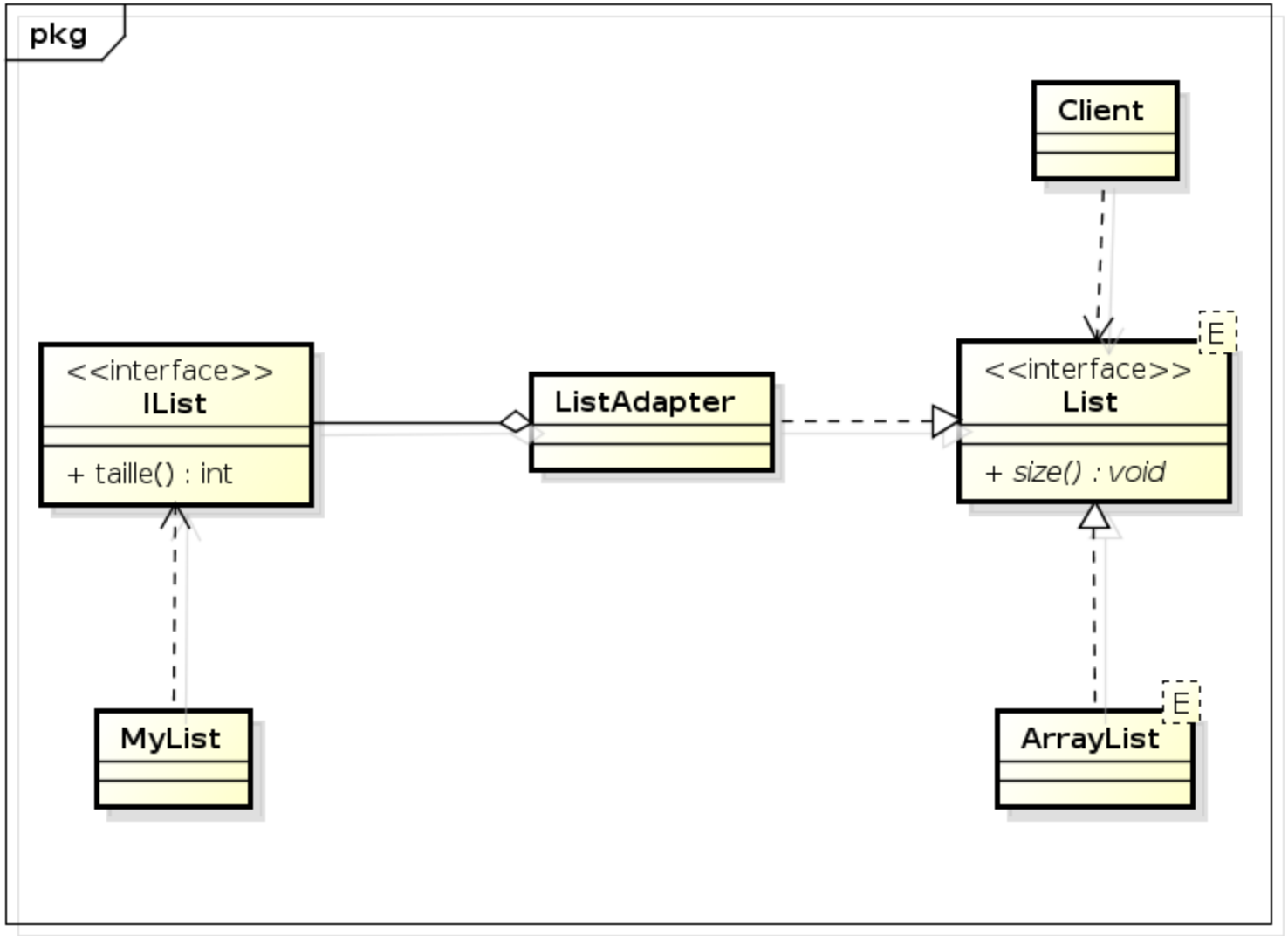


Le pattern adapter

- **Objectif : Il permet de convertir l'interface d'une classe en une autre interface que le client attend. L' Adaptateur fait fonctionner ensemble des classes qui n'auraient pas pu fonctionner sans lui, à cause d'une incompatibilité d'interfaces.**

- **Motivation : Vous voulez intégrer une classe que vous ne voulez/pouvez pas modifier.**

Pattern adapter : exemple



CHANGE PREVENTOR : CONDITIONAL COMPLEXITY

```
if (easyToRead()) rewriteAgain();
```

Exemple

```
public double capital() {
        if (expiry == null && maturity != null)
                return commitment * duration() * riskFactor();
        if (expiry != null && maturity == null) {
                 if (getUnusedPercentage() != 1.0)
                        return commitment * getUnusedPercentage() * duration() * riskFactor();
                else return (outstandingRiskAmount() * duration() * riskFactor()) +
                                                (unusedRiskAmount() * duration() * unusedRiskFactor());
        }
        return 0.0;
}
```

Principe

- Les blocs conditionnels sont souvent simple
- Mais ils vieillissent souvent mal
- ajouts de fonctionnalités → complexifient ces blocs

- → lisibilité

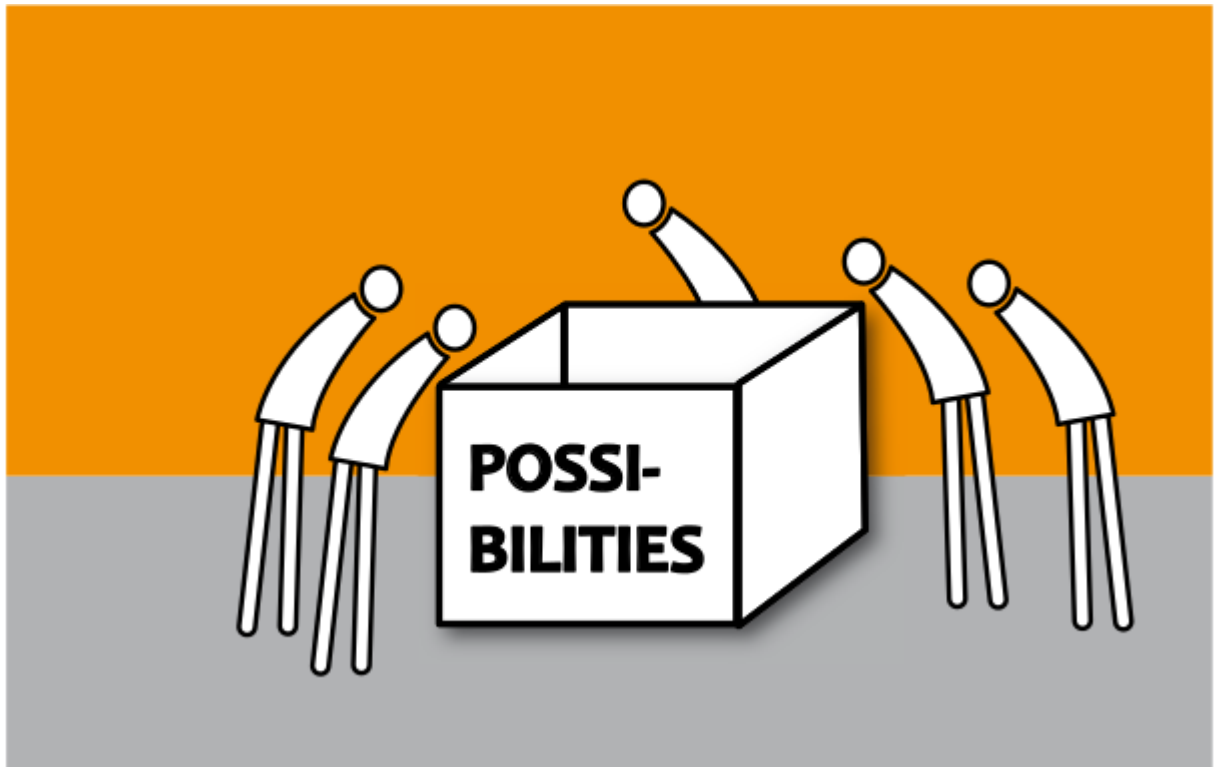
- → Évolution

Exercice

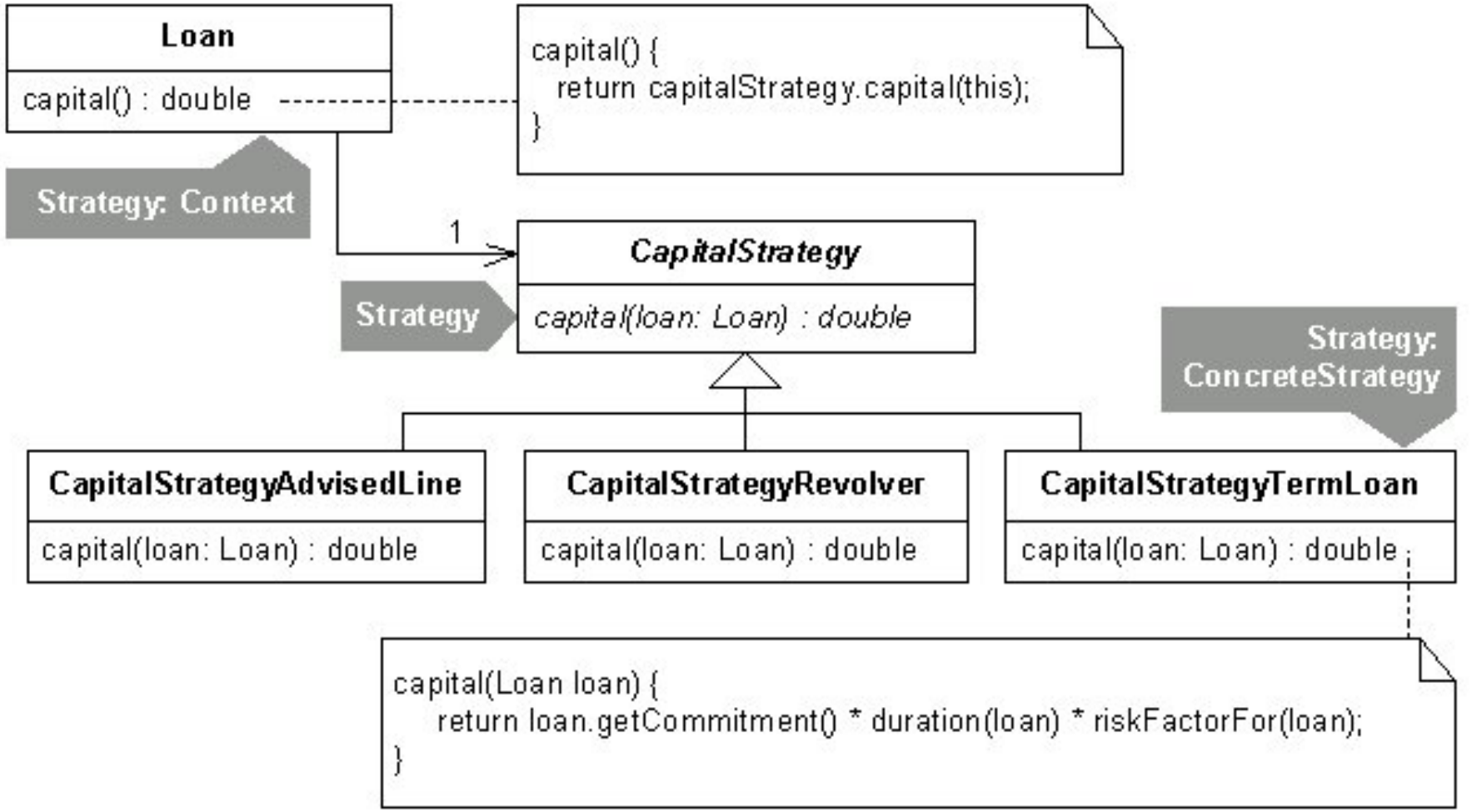
```
public double capital() {
        if (expiry == null && maturity != null)
                return commitment * duration() * riskFactor();
        if (expiry != null && maturity == null) {
                 if (getUnusedPercentage() != 1.0)
                         return commitment * getUnusedPercentage() * duration() * riskFactor();
                else return (outstandingRiskAmount() * duration() * riskFactor()) +
                                                 (unusedRiskAmount() * duration() * unusedRiskFactor());
        }
        return 0.0;
}
```

Refactoring possibles : Pattern stratégie

- **Quand quelque chose peut être fait de plusieurs façon : rendre ceux-ci interchangeable**

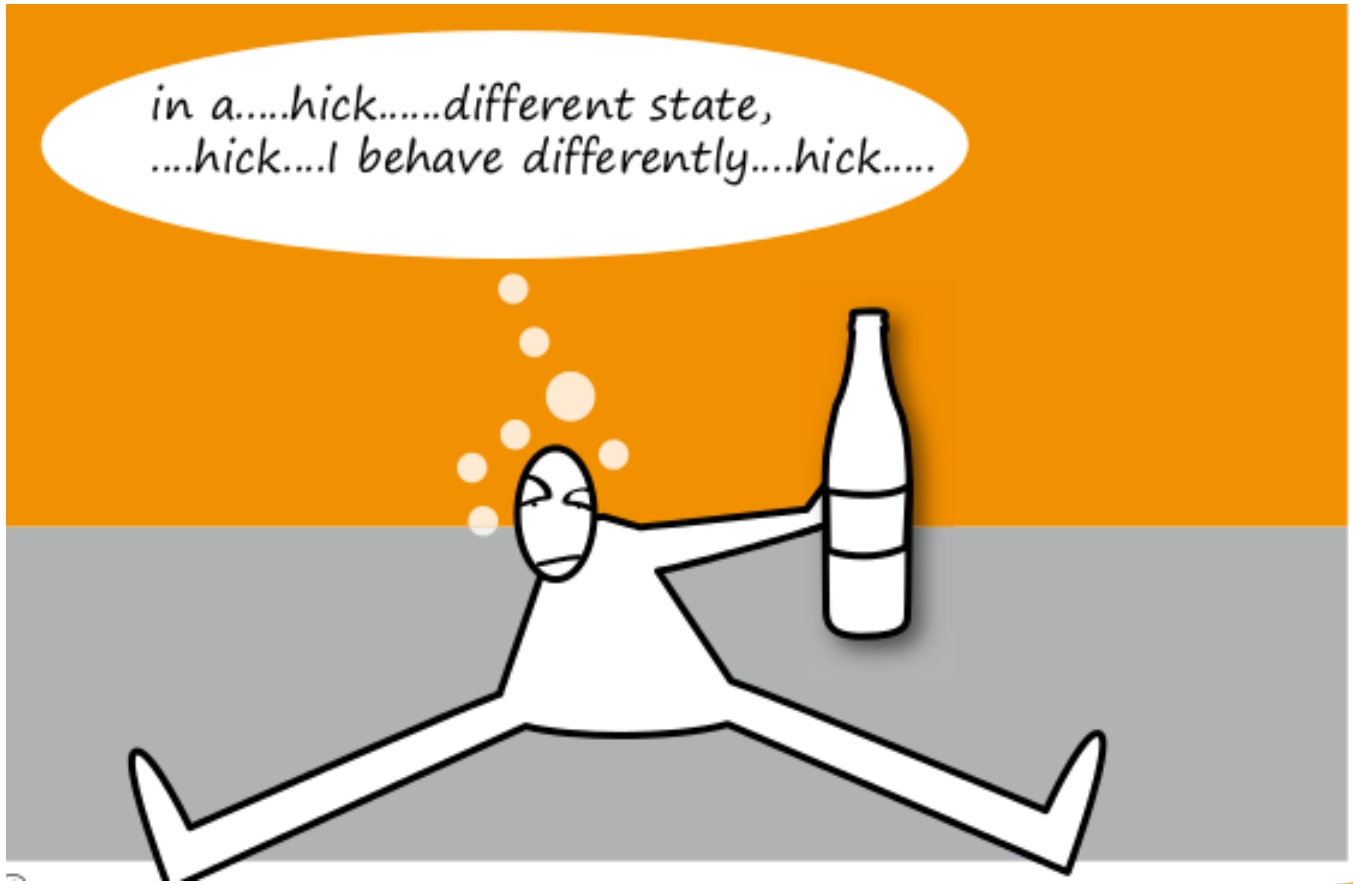


Pattern stratégie

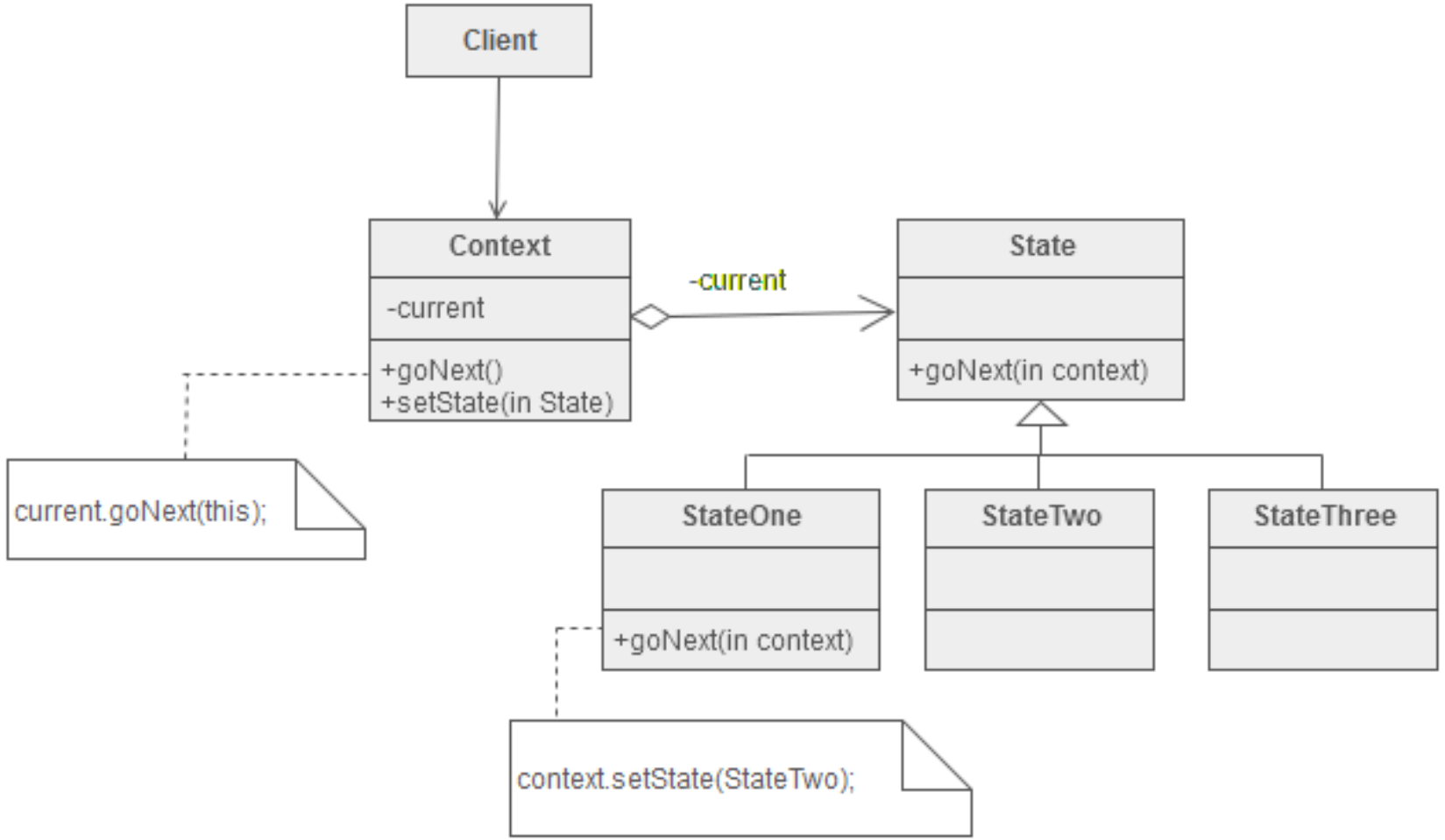


Refactoring possibles : Pattern état

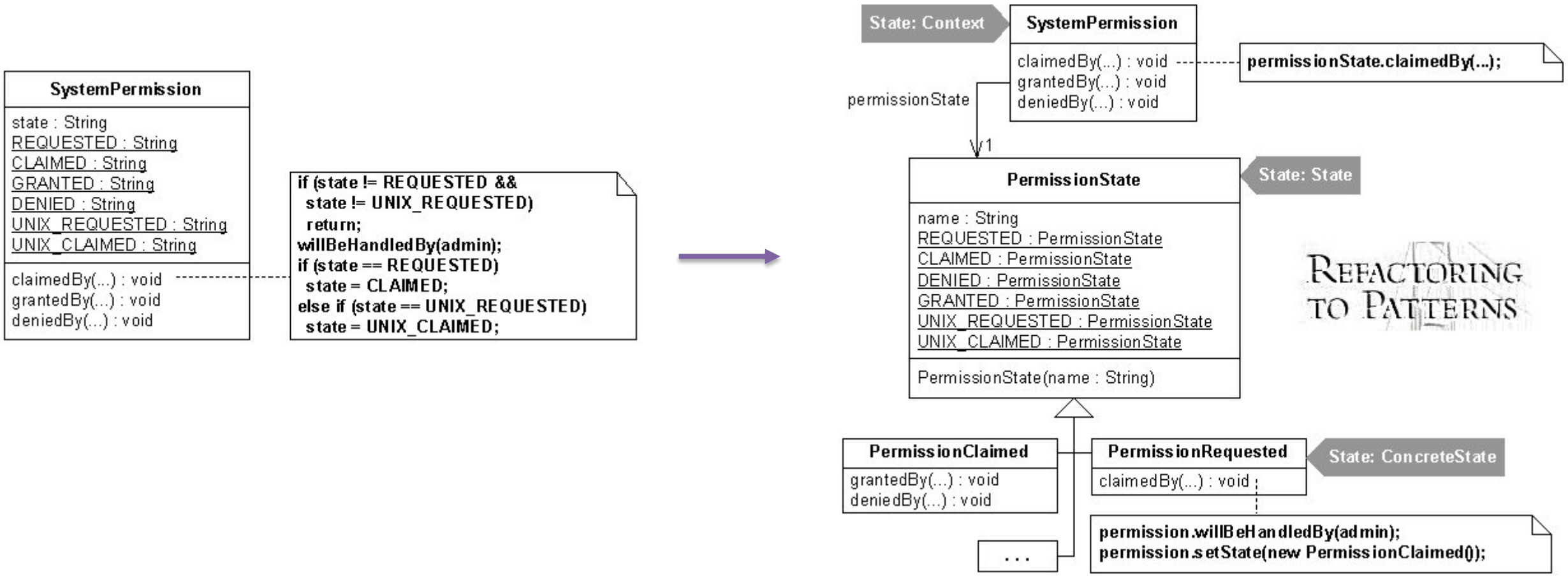
- **Laisser un objet présenter d'autre méthode après un changement interne de son état (comme si elle changeait sa catégorie)**



Pattern état



Pattern état



CHANGE PREVENTOR : INDECENT EXPOSURE

CodeRoulette.com / XCode.com

Principe

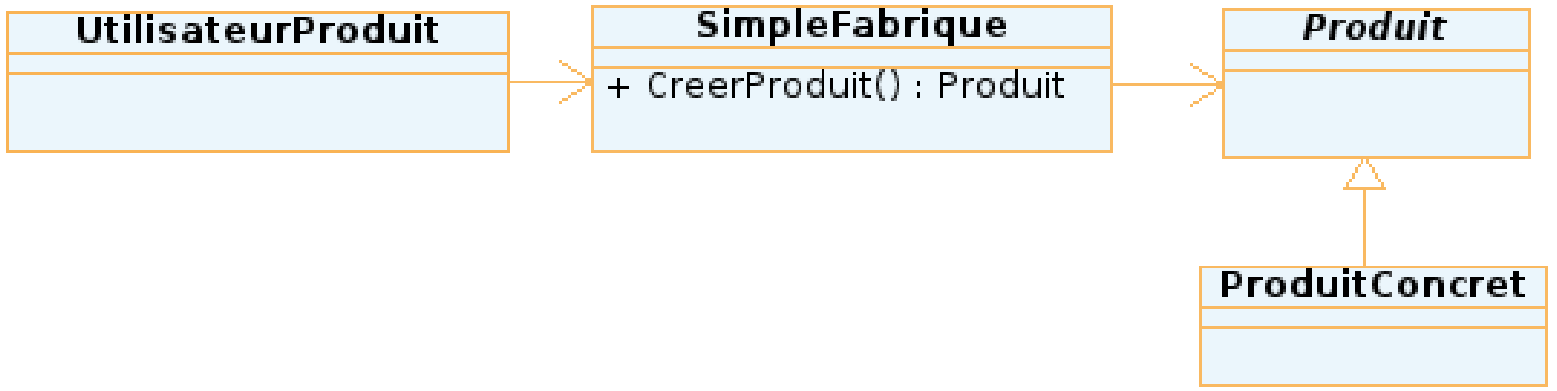
- **Quand une classe exhibe un peu trop sa structure interne**
- **S'oppose au principe d'encapsulation**
- **Pas d'abstraction**

- **Rend les changements difficiles**

  - Trop lié à l'implémentation de la classe

Refactoring possible

- **Utiliser des interfaces**
  - Mieux vaut être dépendant d'un métier que d'une implémentations (Abstraction)
- **Encapsuler la classes avec une Factory**
  - En utilisant le pattern factory l'utilisateur n'a plus connaissance de la classe d'implémentation utilisée.

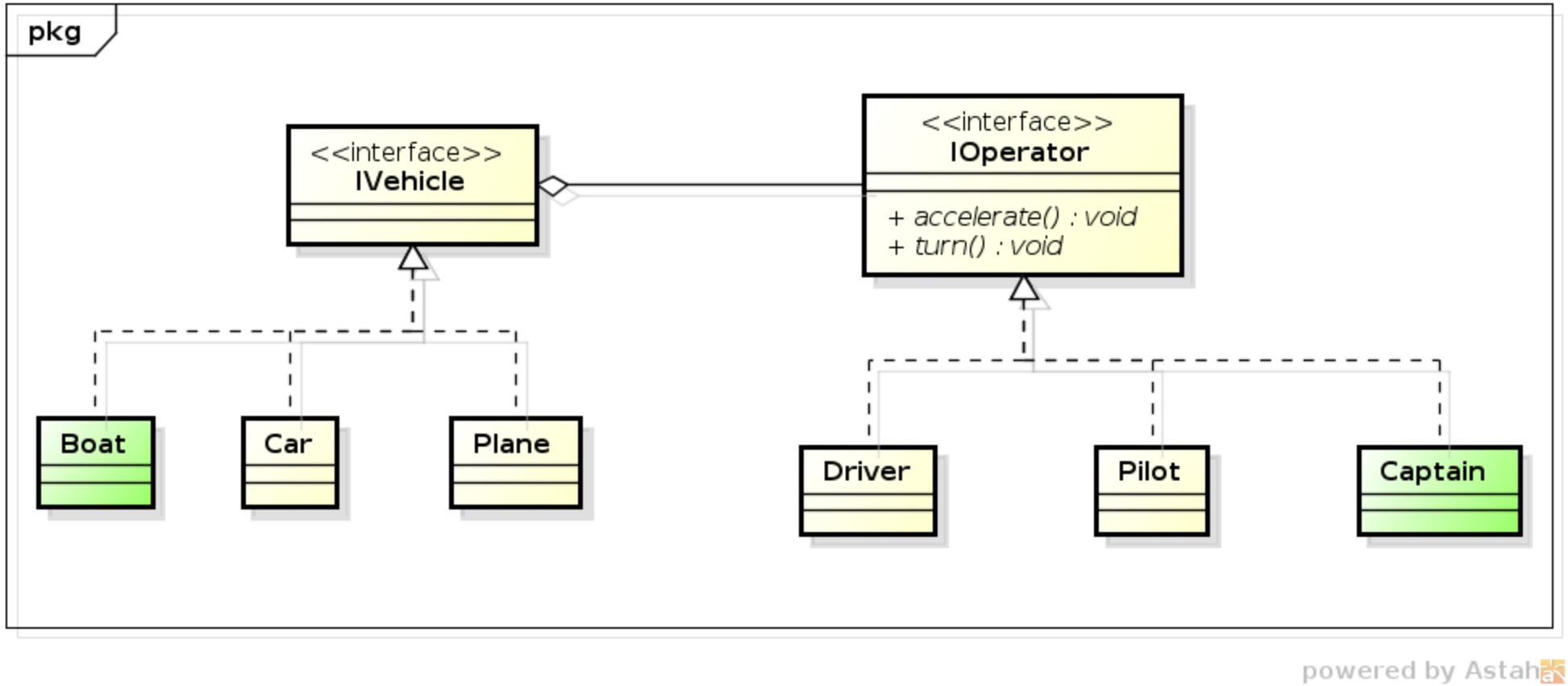


CHANGE PREVENTOR : PARALLEL INHERITANCE HIERARCHIES

J’ai quatre chefs mais zéro augmentation.

tocea

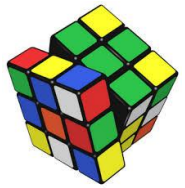
Exemple


pkg
<<interface>>
IVehicle
<<interface>>
IOperator
+ accelerate() : void
+ turn() : void
Boat
Car
Plane
Driver
Pilot
Captain
powered by Astah

Principe

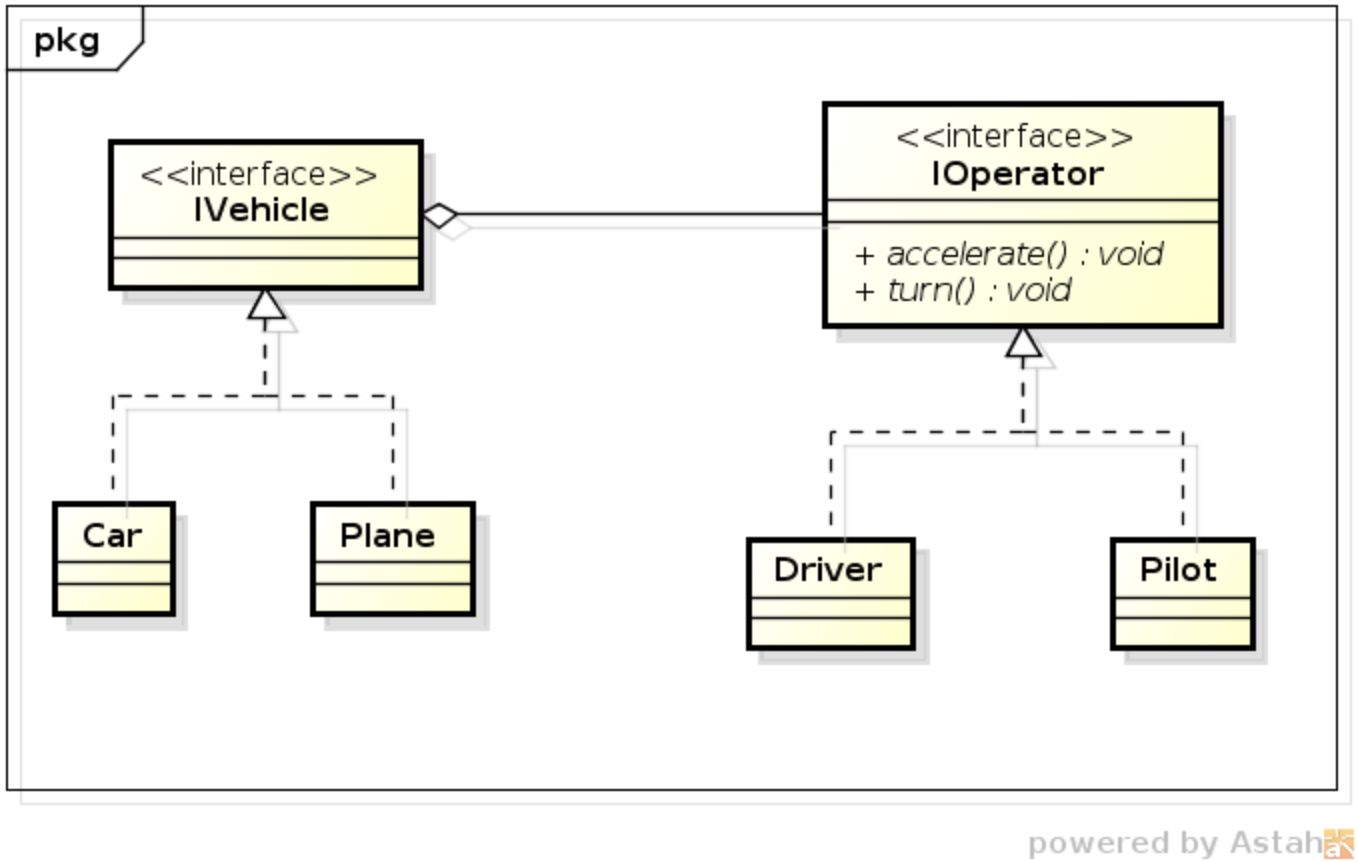
- **Quand 2 Hiérarchie de classes sont lié par une composition**
- **Et que l'ajout d'une classe dans une de ces hiérarchie implique la création d'une autre classe dans la deuxième hiérarchie**

- **Impact :**
  - Complexité



  - Évolution

Exercice


pkg
<<interface>>
IVehicle
<<interface>>
IOperator
+ accelerate() : void
+ turn() : void
Car
Plane
Driver
Pilot
powered by Astah

Refactoring possible

- Déplacer une méthode

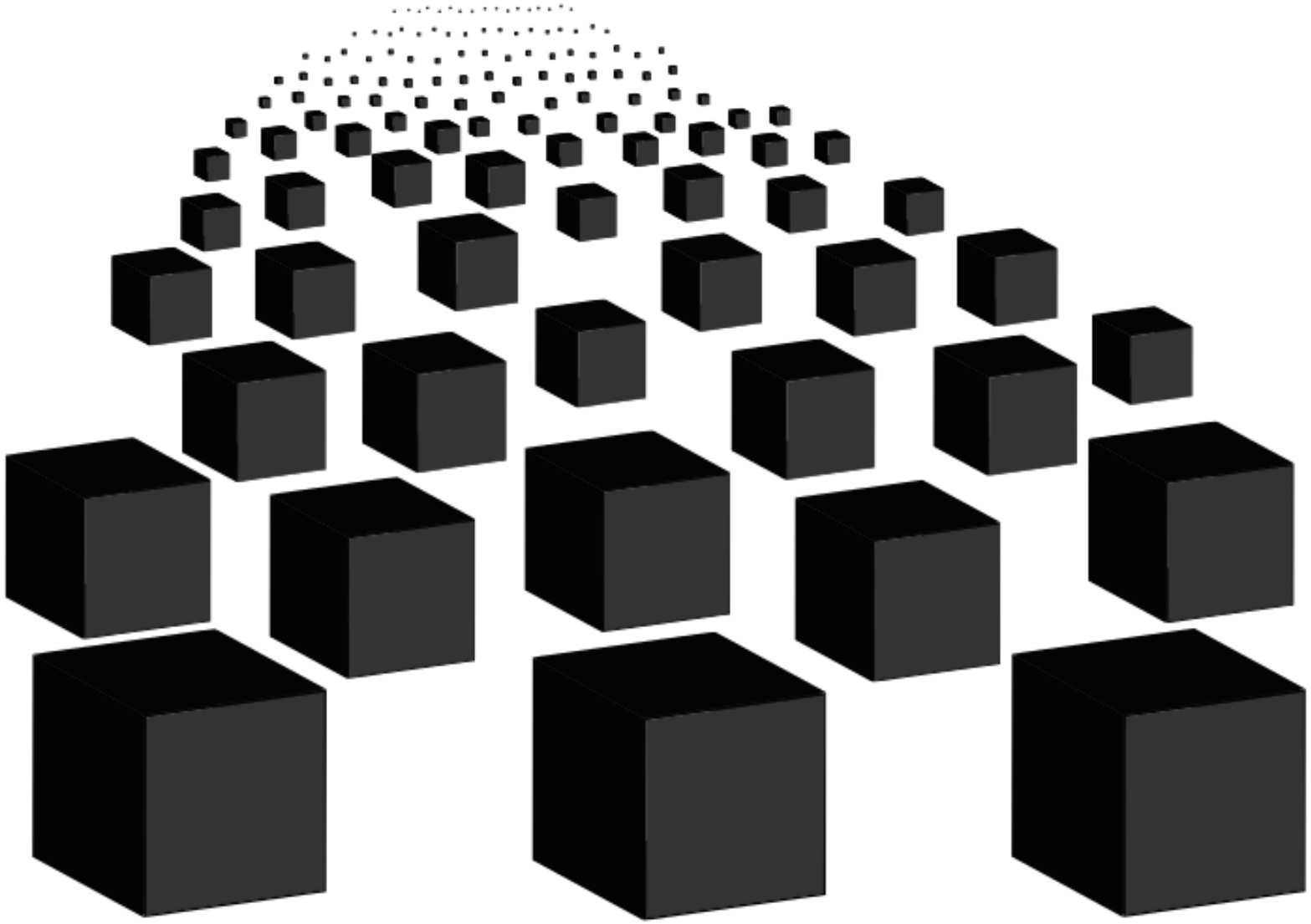
- Déplacer un attributs

CONCLUSION

Une constante dans le développement logiciel

CHANGE!

Idéal : coder avec des « boites noires » modulaires



Quelques principes d'OOP

- Couplage léger
- Réutiliser le code existant (≠ copier/coller)
- Encapsuler ce qui varie
- Principe d'unique responsabilité
- Préférez l'utilisation d'interfaces aux implémentations.
- Injections de dépendances
- Préférez la composition à l'héritage

Évitez le couplage fort

Cela pourrait vous attirer des ennuis…

Soyez attentif à « l'odeur » du code

Certains code « sentent »

- Code dupliqué
- Longue méthodes, classes
- Explosion combinatoire
- Conditionnelles complexes
- Classe « exhibitionnistes »
- Code sans test ou difficilement testable

tocea