

# DUT-Info/S3/M3105 (CPOA)

## Table des Matières

Avant-propos .....	4
Ce que <b>n'est pas</b> ce livre : .....	4
Pourquoi lire ce livre ? .....	4
Comment lire ce livre ? .....	4
Conventions typographiques et symboles récurrents .....	4
Remerciements .....	5
Matériel à disposition avec ce livre .....	5
Style "télégraphique" de ce livre .....	6
1. Introduction .....	6
1.1. Concepts objets .....	6
1.2. Objectifs de la conception objet .....	8
1.3. Bonnes pratiques et patrons .....	9
1.4. Organisation du cours .....	9
2. Rappels sur des éléments Java importants .....	10
2.1. Importance du typage .....	10
2.2. Importance de la visibilité .....	12
2.3. Retour sur les Membres <b>static</b> .....	14
2.4. Utilité générale des <b>enum</b> .....	16
3. Design patterns .....	18
3.1. Introduction : importance des patrons .....	18
3.2. Un peu d'histoire .....	27
3.3. Exemples de bons principes .....	28
3.4. Les patrons : comment ça marche ? .....	38
4. Le patron Fabrique .....	39
4.1. Principes de conception .....	39
4.2. Premier exemple d'utilisation de patron .....	40
4.3. Autre exemple concret .....	40
4.4. Un autre exemple concret .....	42
4.5. Mais c'est pas fini! .....	43
4.6. Fabrique abstraite .....	47
5. Etat .....	47
5.1. Implémentation intuitive .....	48
5.2. Erreur d'implémentations .....	48
5.3. Une meilleure implémentation .....	49
5.4. Illustration .....	49
5.5. Le patron Etat .....	50

6. Observateur .....	51
6.1. Motivation .....	52
6.2. Définition .....	52
6.3. Application .....	53
6.4. <i>Observer</i> en Java .....	53
6.5. Une implémentation du MVC : les JTable java .....	54
7. Adaptateur .....	56
7.1. Le problème .....	57
7.2. Exemple concret : le retour des canards .....	57
7.3. Le patron Adaptateur .....	59
8. Le patron Visiteur .....	60
8.1. Le problème .....	60
8.2. Illustration .....	61
8.3. Le patron Visiteur .....	66
8.4. Avantages/Inconvénients .....	67
8.5. Exemples d'utilisation .....	68
8.6. Exemple concret d'utilisation en Java .....	68
9. Proxy .....	72
9.1. Le problème .....	72
9.2. Le patron <i>Proxy</i> .....	72
9.3. Utilisations .....	73
9.4. Exemple concret : RMI .....	73
10. Itérateur .....	74
10.1. Le problème .....	74
10.2. Le patron Itérateur .....	74
10.3. Exemple concret .....	75
11. Le patron Composite .....	76
11.1. Le problème .....	76
11.2. Le patron Composite .....	76
11.3. Exemple concret .....	77
11.4. un "Anti" exemple .....	78
12. Pour aller plus loin avec les patrons... .....	79
12.1. Partagez votre vocabulaire .....	79
12.2. Ne foncez pas tête baissée .....	79
12.3. Les autres types de patrons .....	80
12.4. Les anti-patrons .....	80
12.5. Tous les patrons qu'on a pas vu .....	81
12.6. Injection de dépendances .....	81
Glossaire et définition .....	81
Patrons de création .....	81
Patrons comportementaux .....	83

Patrons structurels .....	86
Références .....	88
Exercices corrigés .....	88
Appendix A: CPOA - Support TD 1 .....	88
A.1. Rappel du cours .....	89
A.2. L'application SuperCanard .....	89
Pour aller plus loin .....	101
Appendix B: CPOA - Support TD 2 .....	103
B.1. Rappel du cours .....	103
B.2. La fabrique de chocolat .....	103
B.3. Singleton .....	112
B.4. Le singleton pour le jeu d'aventure .....	113
Pour aller plus loin .....	114
Appendix C: CPOA - Support TD 3 .....	115
C.1. Rappel du cours .....	116
C.2. La pizzeria O'Reilly .....	116
C.3. On y est presque... .....	119
C.4. Le patron Fabrique (simple) .....	123
Pour aller plus loin .....	123
Appendix D: CPOA - Support TD 4 .....	129
D.1. Rappel du cours .....	129
D.2. Différences entre dépendance, association, composition, agrégation .....	129
D.3. Patrons .....	130
D.4. Diagrammes de séquences .....	133
D.5. Machines à état .....	137
Pour aller plus loin .....	139
Appendix E: CPOA - Support TD 5 .....	140
E.1. Rappel du cours .....	141
E.2. Motivation .....	141
E.3. Le patron Observer .....	141
E.4. Implémentation .....	144
Pour aller plus loin .....	156
Appendix F: CPOA - Support TP 4 .....	156
F.1. Le but : "Objet-ivez" votre code .....	157
F.2. Etude et reprise de l'existant .....	157
F.3. "Objet-iver" les fonctions .....	159
F.4. Abstraire le problème .....	163
F.5. Pour aller plus loin : complétons et encore plus d'abstraction .....	166
About....	168

# Avant-propos

## Ce que n'est pas ce livre :

Ce livre **n'est pas** un livre :

- de référence (pour cela, même s'il est en anglais, nous conseillons [[Meyer88](#)])
- sur les *design patterns* (pour cela, même s'il est en anglais, nous conseillons [[GoF](#)])



Nous considérons également que vous avez un bagage minimum en programmation objet. Notamment que vous avez suivi les modules de programmation et de conception (M2103 et M2104) si vous êtes en DUT Informatique.

## Pourquoi lire ce livre ?

Parce que vous êtes en DUT Informatique et que vous souhaitez préparer le module consacré aux *design patterns* (CPOA ou M3105 dans le jargon du PPN2013); parce que vous aimez les livres reliés et que lire le support sur le site du livre vous barbe; parce que vous vous intéressez aux *design patterns* et que vous souhaitez pratiquer de manière ludique.

## Comment lire ce livre ?

Ce document a été réalisé de manière à être lu de préférence dans sa version électronique, ce qui permet de naviguer entre les références et les renvois interactivement, de consulter directement les documents référencés par une URL, etc.



Si vous lisez la version papier de ce document, ces liens clickables ne vous servent à rien, mais n'hésitez pas à en consulter la version [électronique](#)!

## Conventions typographiques et symboles récurrents

Nous avons utilisé un certain nombre de conventions personnelles pour rendre ce document le plus agréable à lire et le plus utile possible, grâce notamment à la puissance d'[AsciiDoc](#) :

- des mises en formes particulières (e.g., [NomDeClasse](#) pour un élément de modèle),
- des références bibliographiques (comme la bible [[Meyer88](#)]), présentées en fin de document (cf. [Références](#)),
- les termes anglais (souvent incontournables) sont repérés en *italique*, non pas pour indiquer qu'il s'agit d'un mot anglais, mais pour indiquer au lecteur que nous employons volontairement ces termes (e.g., *design pattern*),
- un certain nombre de symboles viennent identifier les notes :



Ceci est une simple note, un point remarquable.



Attention, piège ou erreur à éviter.



Ceci est un point important.



Ceci est un conseil ou une bonne pratique.

## Remerciements

Merci tout d'abord à ma famille pour sa patience vis-à-vis de ce métier qui ne se termine pas quand on quitte le bureau.

Merci aux contributeurs et aux collègues qui ont, par leurs conseils, leurs discussions, leurs remarques, ont permis d'enrichir ce support.

Et un merci tout particulier à celui qui m'a tant apporté à l'IUT et qui nous a quitté trop tôt, Jean-Michel Inglebert.

## Matériel à disposition avec ce livre

Vous trouverez sur le dépôt lié à ce livre, link:<http://bit.ly/jmb-cpoa> :

- une version web avec liens clickables

Table of Contents

1. Introduction
- 1.2. Objectifs de conception objet
- 1.3. Bonnes pratiques et patrons
- 1.4. Organisation du cours
- 1.5. Evaluation et notation
- 1.6. A propos des auteurs et personnes importantes
- 2.1. Importance de l'objet
- 2.2. Importance de la lisibilité
- 2.3. Retour sur les Membres static
- 2.4. Utilité générale des enum
3. Concepts objets et applications
- 3.1. Les scripts
- 3.2. mœurs
- 3.3. arr
- 3.4. Mœurs
- 3.5. lv
- 3.6. Grâde
4. Des patrons patterns
- 4.1. importance : importance des patrons
- 4.2. Un peu d'histoire
- 4.3. Exemples de bons principes
- 4.4. GRASP
- 4.5. Les patrons : comment ça marche
5. Le patron Fabrique
- 5.1. Principes de conception
- 5.2. Premier exemple d'utilisation du patron
- 5.3. Autre exemple concret

### DUT-Info/S3/M3105 (CPOA)

Jean-Michel Bruel - [jbruel@gmail.com](mailto:jbruel@gmail.com) - Version 1.6, 2017-01-08

#### 1. Introduction

Ce cours porte sur la Conception et Programmation Objet Avancée.

##### 1.1. Concepts objets

Vous allez apprendre (cf. M2103 et M2104) un certain nombre de concepts objets :

- Abstraction
- Encapsulation
- Héritage
- Polymorphisme

##### 1.1.1. Abstraction

Définition (restrictive) :

**“Une classe est une abstraction des caractéristiques communes d'un ensemble d'objets.**

##### 1.1.2. Encapsulation

Définition (restrictive) :

**“Dans la description d'un objet, le but de l'encapsulation est de masquer les attributs et les méthodes, c'est à dire, la manière dont est réalisée le comportement de l'objet.**

#### 1.2. Objectifs de conception objet

Pour répondre aux problèmes ci-dessus, on va s'engager à élaborer des objectivités et éviter les qualités négatives ci-dessous :

- Incohérence : le changement n'introduit pas de régressions.
- Difficulté : il est facile d'apprendre de nouvelles fonctionnalités.
- Complexité : il est possible de résoudre toutes sortes de cas de figure pour concevoir d'autres applications.

#### 2.3. Bonnes pratiques et patrons

Pour répondre aux problèmes ci-dessus, on va s'engager à élaborer des objectivités et éviter les qualités négatives ci-dessous :

- Incohérence : le changement n'introduit pas de régressions.
- Difficulté : il est facile d'apprendre de nouvelles fonctionnalités.
- Complexité : il est possible de résoudre toutes sortes de cas de figure pour concevoir d'autres applications.

#### 2.4. Organisation du cours

Appel à réflexion : 1 cours, 1 TD et 2 TP par semaine. Pendant 4 semaines.

La première semaine est consacrée au pétrole des patrons de conception, en partant d'un exercice.

Les 3 semaines suivantes sont consacrées à l'étude de certains patrons classiques. Vise en pratique sur des exercices en TD.

The screenshot shows a PDF document titled "DUT-Info/S3/M3105 (CPOA)" with a page number of 4. The document contains the table of contents from the previous page, followed by the introduction section. The introduction discusses the purpose of the course, the concepts of objects, and the goals of object-oriented design. It also mentions the use of patterns and provides a brief overview of the course organization.

- la version PDF téléchargeable

- les exercices de TD, TP (version élève et profs) avec les corrections

**Version corrigée**

Cette version comporte des indications pour les réponses aux exercices.

PreReq	1. Je sais programmer en <b>Java</b> . 2. J'ai conscience qu'il faut réfléchir avant de se lancer dans le codage. 3. Je maîtrise les concepts objet de base (héritage, polymorphisme, ...).
ObjetID	Comprendre ce qu'est une <b>conception</b> .
Durée	1 TD et 2 TP de 1,5h (sur 2 semaines différentes).

## 1. L'application SuperCanard



Les exercices de ce TD sont tirés de l'excellent livre "Tête la première : Design Pattern". Bert Bates, Eric Freeman, Elisabeth Freeman, Kathy Sierra. Editions O'Reilly, 2005.

### 1.1. Application existante

Soit l'application (un jeu de simulation de mare aux canards) **SuperCanard** dont le modèle est décrit ci-dessous :

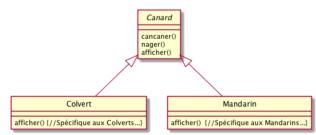


Figure 1. Extrait d'une application existante (source plantUML [ici](#))

- le support de cours pour les enseignants (slides HTML)

### Illustration (suite)

Etape 4 (enfin, retour sur l'étape 2) : une solution possible...

```

public class EtatSansPiece implements Etat {
    Distributeur distributeur; // référence au distributeur qu'on gère

    public EtatSansPiece(Distributeur distributeur) {
        this.distributeur = distributeur;
    }

    public void insererPiece() {
        System.out.println("Vous avez inséré une pièce");
        distributeur.setEstat(distributeur.getEstatAPiece());
    }
    ...
}
  
```

# Style "télégraphique" de ce livre

Vous pourrez être surpris du style "télégraphique" de la suite de ce livre (presque à l'opposé du livre de référence [Meyer]). Il faut réaliser qu'il ne s'agit que d'un support de cours, sensé être accompagné des exercices réalisés en travaux dirigés (que vous trouverez en [Annexes](#)), des travaux pratiques sur machine (disponibles [ici](#)), et des cours d'amphi qui vont avec (eux aussi bientôt disponibles).

# 1. Introduction

Ce cours porte sur la **Conception et Programmation Objet Avancée**. Il part des principaux concepts objets, considérés comme déjà manipulés, et aborde les principes importants, les bonnes pratiques et plus spécifiquement les patrons de conception (*design patterns*).

## 1.1. Concepts objets

Vous avez appris (cf. M2103 et M2104) un certain nombre de **concept objets** :

- Abstraction

- Encapsulation
- Héritage
- Polymorphisme

Rappelons rapidement leurs définitions.

### 1.1.1. Abstraction

Définition (restrictive) :

Une classe est une **abstraction** des caractéristiques communes d'un ensemble d'objets.

### 1.1.2. Encapsulation

Définition (restrictive) :

Dans la description d'un objet, le but de l'**encapsulation** est de masquer les attributs et les méthodes, c'est-à-dire, la manière dont est réalisé le comportement de l'objet.

### 1.1.3. Héritage

Définition (simpliste) :

L'héritage est la transmission de caractéristiques à ses descendants.

La classe qui hérite dispose des méthodes et attributs de niveau **public** et **protected** de sa classe mère.

### 1.1.4. Polymorphisme

Le nom de polymorphisme vient du grec :

qui peut prendre plusieurs formes



L'héritage concerne les classes, le polymorphisme concerne les objets.

On distingue généralement trois types de polymorphisme :

- Le polymorphisme **ad hoc** (également surcharge ou en anglais overloading)
- Le polymorphisme **paramétrique** (également générativité ou en anglais template)
- Le polymorphisme **d'héritage** (également redéfinition, spécialisation ou en anglais overriding)

## Polymorphisme ad hoc

- Appelé aussi **surcharge**.
- Permet d'avoir des fonctions de même nom dans des classes sans aucun rapport entre elles.
- Permet de définir des opérateurs d'utilisation différente en fonction des paramètres.

```
int method(int,int);
int method(int);
int method(float,float);
```

## Le polymorphisme paramétrique

Appelé aussi **généricité**. En voici un exemple en Java :

```
interface Iterator<E> {
    boolean hasNext();
    E next();
}

public <T> static void copy(Collection<? extends T> source, Collection<? super T> dest) {
    for (T t : source) {
        dest.add(t);
    }
}
```

## Le polymorphisme d'héritage

- Appelé aussi **spécialisation** (ou redéfinition).
- Lié à la redéfinition des méthodes héritées.

# 1.2. Objectifs de la conception objet

On essaye d'éviter trois problèmes principaux du développement :

### La rigidité

Anticiper les évolutions susceptibles d'impacter l'application.

### La fragilité

Eviter les erreurs provoquées par la modification d'une partie du code.

### L'immobilité

Rendre moins difficile l'extraction d'une partie du code.

## 1.3. Bonnes pratiques et patrons

Pour répondre aux problèmes ci-dessus, on va s'attaquer à diminuer les **dépendances** et éviter l'"effet spaghetti".

Les qualités recherchées sont :

- Robustesse : les changements n'introduisent pas de régressions.
- Extensibilité : il est facile d'ajouter de nouvelles fonctionnalités.
- Réutilisabilité : il est possible de réutiliser certaines parties de code pour construire d'autres applications.

Nous allons apprendre des **bonnes pratiques** :

- Identifier les aspects qui varient et les séparer des aspects constants
- Programmer une interface, non une implémentation
- Préférer la composition à l'héritage
- Les classes doivent être ouvertes à l'extension, mais fermées à la modification
- Dépendez d'abstractions. Ne dépendez pas de classes concrètes (inversion des dépendances)
- Ne parlez pas aux inconnus

L'étape suivante consiste à apprendre les bonnes solutions de conception, ce qu'on appelle les **patrons de conception** (ou *design patterns* en anglais).

## 1.4. Organisation du cours

Nous indiquons ici l'organisation du cours associé à ce livre, tel qu'il est pratiqué à l'[IUT de Blagnac](#). Le module est réalisé sur 8 semaines (sur les 16 que dure le semestre), donc à un rythme plus soutenu, mais qui ne pose aucun problème aux étudiants. À l'[IUT de Blagnac](#), ce module (M3105) fait suite à 8 semaines de Méthodologie de la Production d'Application (MPA - M3301).



Rappel du rythme : 1 cours, 1 TD et 2 TPs par semaine. Pendant 8 semaines.

- La première semaine est consacrée aux principes généraux des patrons de conception, en partant d'un exemple (cours en fin de semaine).
- Les 5 ou 6 suivantes sont consacrées à l'étude de certains patrons classiques. Mise en pratique sur des exercices en TP.



Le cours est inversé par rapport aux habitudes : Conception et étude d'un ou plusieurs patrons semaine N; mise en oeuvre en TP semaine N+1; puis cours en amphi (détails, discussions) en semaine N+2.

- Les 2 ou 3 dernières semaines, les étudiants sont en mode projet pour faire du *refactoring* d'applications réelles (conception aidée en TD sur les modèles UML™, mise en oeuvre en TP).

Voici une proposition de déroulement des semaines :

## Semaine 1

SuperCanard, le grand classique, *Strategy*

## Semaine 2

- *Singleton*

## Semaine 3

Patrons *Le patron Fabrique, Proxy, Etat*

## Semaine 4

- *Observateur*
  - version intuitive (2 interfaces)
  - version Java (classe *Observable*)

## Semaine 5

L'exemple de Meyer : menus en objet

## Semaine 6

- Patrons Décorateur, Façade, Visiteur
- MVC avec l'exemple *JTable* de Java
- Patrons Chaîne de responsabilité (juste en cours)

## Semaines 7 et 8

- Quelques idées de projet final :
  - Refactorer un code généré par *Umpire*.
  - Refactorer le code de MPA (mais pas le leur, celui d'un autre groupe)

# 2. Rappels sur des éléments Java importants

Ces rappels ont été introduits dans mon cours au fur et à mesure des constats de manques et de difficultés des étudiants pour aborder correctement la programmation des patrons.

## 2.1. Importance du typage

### 2.1.1. Différents types de typage

Le fait d'attribuer un type (une classe) à une variable (un objet) peut se faire de plusieurs façons :

- statique
- dynamique
- *duck typing*

## 2.1.2. Typage statique

On parle de **typage statique** quand la majorité des vérifications de type sont effectuées au moment de la **compilation**.

*Exemple de typage statique*

```
int i = 0; // cette déclaration indique explicitement que  
           // la variable i est de type entier
```

## 2.1.3. Typage dynamique

Le **typage dynamique** consiste à laisser l'ordinateur réaliser l'opération de typage *à la volée*, **lors de l'exécution du code**.

*Exemple de typage dynamique*

```
/**  
 * @author André Peninou  
 */  
public class Type {  
    void m() {  
        System.out.println ("Type");  
    }  
}  
public class SousType extends Type {  
    void m() {  
        System.out.println ("SousType");  
    }  
    void autreM(){  
        System.out.println ("Spécifique SousType");  
    }  
}  
...  
    Type a = new Type();  
    a.m(); // "Type"  
  
    a = new SousType();  
    a.m(); // "SousType"  
    // Statique : a est un Type (à la compil)  
    // Dynamique : a est un SousType au runtime.  
  
    // D'où :  
    a = new SousType();  
    a.autreM();  
    // NOK car type statique == A => autreM() n'existe pas à la compilation  
...
```

## 2.1.4. Duck typing

Style de **typage dynamique** où la **sémantique** d'un objet (c'est-à-dire son type) est déterminée par l'ensemble de ses **méthodes** et de ses **attributs**, et non par un type défini et nommé explicitement par le programmeur.

L'origine de cette expression est liée à cette citation :



Si je vois un animal qui vole comme un canard, cancane comme un canard, et nage comme un canard, alors j'appelle cet oiseau un canard.

— James Whitcomb Riley

Exemple de duck typing en Ruby

```
def calcule(a, b, c)
    return a*b+c
end

$a = calcule(6, 3, 2)
$b = calcule('6', 3, ', the number of the beast')

puts $a.to_s
puts $b.to_s
```

Ce qui donne :

20

666, the number of the beast



Pour aller plus loin : [http://fr.wikipedia.org/wiki/Duck\\_typing](http://fr.wikipedia.org/wiki/Duck_typing)

## 2.2. Importance de la visibilité

Dès que l'on commence à avoir une application conséquente, l'organisation en *package* devient obligatoire. Revenons donc sur les questions de **visibilité** des propriétés et méthodes, qui seront importants dans la plupart des aspects de ce module.

Si un champ d'une classe A :

- est *private*, il est accessible uniquement depuis sa propre classe ;
- à la visibilité *package* (visibilité par défaut, pas de mot-clef), il est accessible de partout dans le paquetage de A mais de nulle part ailleurs ;
- est *protected*, il est accessible de partout dans le paquetage de A et, si A est publique, grossièrement dans les classes héritant de A dans d'autres paquetages ;

- est *public*, il est accessible de partout dans le paquetage de A et, si A est publique, de partout ailleurs.



Ci-dessus, les niveaux de visibilité sont rangés par visibilité croissante.

```
package UN;
public class A {
    protected String attrprotected;
    String attrfriend; // friend
}
```

Si on définit une deuxième classe dans le même package :

```
package UN;
class B {
    ...
{
    A a = new A ();
    a.attrprotected// OK : même si bizarre
    a.attrfriend // OK : visible package
}
}

package UN;
class C extends A {
    ...
{
    this.attrprotected// OK : normal
    this.attrfriend // OK : visible package
}
}
```

```

package DEUX;
class B {
    ...
    {
        A a = new A ();
        a.attrprotected// NON OK : normal
        a.attrfriend // NON OK : normal, proche de "private"
    }
}

class C extends A {
    ...
    {
        this.attrprotected// OK : normal car protected et héritage
        this.attrfriend // NON OK : normal, proche de "private"
    }
}

```

À la question **private** ou **protected** ? Quel est le mieux pour les attributs ?

- C'est une question de **style de programmation !**
- Puristes (cf. [Meyer]) ⇒ **private**
- Parfois utile : cf. *Strategy*, évite les getters/setters



Il n'y a pas de visibilité par défaut en **UML™**.

## 2.3. Retour sur les Membres **static**

```

class VariableDemo
{
    static int count=0;
    public void increment()
    {
        count++;
    }
    public static void main(String args[])
    {
        VariableDemo obj1=new VariableDemo();
        VariableDemo obj2=new VariableDemo();
        obj1.increment();
        obj2.increment();
        System.out.println("Obj1: count is="+obj1.count);
        System.out.println("Obj2: count is="+obj2.count);
    }
}

```

Output:

```
Obj1: count is=2
Obj2: count is=2
```

### 2.3.1. Membres **static** (suite)

Comment ça marche :

- Les variables **static** sont initialisées au chargement de la classe.
- Les variables **static** d'une classe sont initialisées avant que la moindre instance ne soit créée.
- Les variables **static** sont initialisées avant que la moindre méthode **static** ne s'exécute.

### 2.3.2. Méthodes **static**

```
import java.lang.Math;

class Another {
    public static void main(String[] args) {
        int result;

        result = Math.min(10, 20); //calling static method min by writing class name

        System.out.println(result);
        System.out.println(Math.max(100, 200));
    }
}
```

### 2.3.3. Méthodes **static** et appel aux méthodes non-statiques

```
public class Main {
    public static void main(String[] args) {
        Main p = new Main();
        k();
    }

    protected Main() {
        System.out.print("1234");
    }

    protected void k() {
    }
}
```

À l'exécution :

```
Main p = new Main(); // => prints 1234
k()           // => raises error
```

## Static method cannot call non-static methods

Bien sûr que si, sauf qu'il faut que cette dernière **porte sur une instance** de la classe.

## Constructors are kind of a method with no return type.

En fait il vaudrait mieux les considérer comme une sorte de méthode statique. En effet elle ne requièrent pas de porter sur un objet!



cf. la discussion <http://stackoverflow.com/questions/10513633/static-method-access-to-non-static-constructor>

## 2.4. Utilité générale des enum

### 2.4.1. Modélisation

Le type enumération est souvent utilisé en modélisation :

*Exemple de classe Enumeration en UML*

```
hide circle
hide methods
class VisibilityKind <<enumeration>> {
    public
    private
    protected
    package
}
```

### 2.4.2. Propriétés

```
public enum Civilite {
    MADAME, MONSIEUR
}
```

- Chaque élément d'une énumération est un objet à part entière
- Les objets enum héritent de `java.lang.Enum`
- On peut compléter les comportements des objets en ajoutant des méthodes

### 2.4.3. Méthodes de base

- `toString()`

```
System.out.println(Civilite.MADAME); //MADAME
```

- `valueOf()`

```
Civilite civilite = Civilite.valueOf("MONSIEUR") ;
```

- `values()`

```
Civilite[] civilites = Civilite.values() ;
```

- `ordinal()`

```
Civilite civilite = Civilite.MONSIEUR ;
System.out.println("Civilite : " + civilite + " [" + civilite.ordinal() + "]") ;
// Civilite : MONSIEUR [1]
```



Le 1er numéro d'ordre est 0.

- `compareTo()`

```
System.out.println(Civilite.MADAME.compareTo(Civilite.MONSIEUR)) ;
// -1
```

### 2.4.4. Exemple plus complexe

```
public enum Langage {  
    //Objets directement construits  
    JAVA("Langage JAVA", "Eclipse"),  
    C ("Langage C", "Code Block"),  
    CPlus ("Langage C++", "Visual studio"),  
    PHP ("Langage PHP", "PS Pad");  
  
    private String name = "";  
    private String editor = "";  
  
    //Constructeur  
    Langage(String name, String editor){  
        this.name = name;  
        this.editor = editor;  
    }  
  
    public void getEditor(){  
        System.out.println("Editeur : " + editor);  
    }  
  
    public String toString(){  
        return name;  
    }  
  
    public static void main(String args[]){  
        Langage l1 = Langage.JAVA;  
        Langage l2 = Langage.PHP;  
  
        l1.getEditor();  
        l2.getEditor();  
    }  
}
```

## 3. Design patterns

Ce chapitre présente les principes généraux des patrons de conception et les suivants détaillent quelques patrons importants à connaître en sortant d'un DUT Informatique. Néanmoins, si vous voulez les apprendre correctement à partir de ce livre, nous vous déconseillons de lire cette partie avant d'avoir réalisé les TDs/TPs qui leurs sont associés (cf. [Annexes](#)).



### 3.1. Introduction : importance des patrons

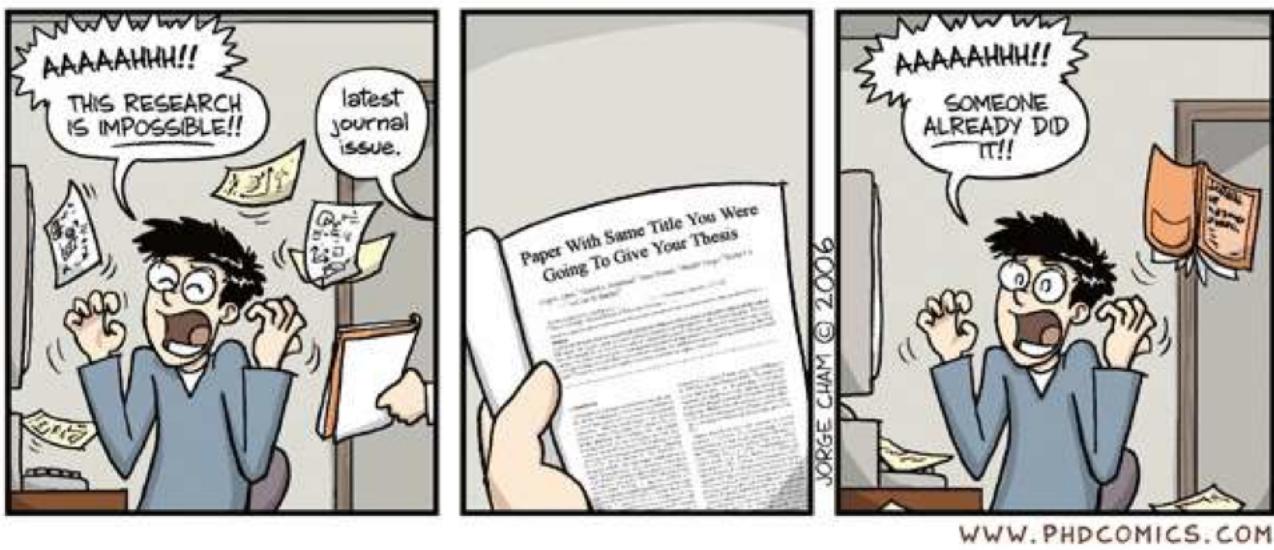


Figure 1. Les patrons : des réponses éprouvées à des problèmes récurrents

Science is what we understand well enough to explain to a computer. Art is everything else we do.

— Donald Knuth

Nous allons illustrer les principes généraux au travers d'un premier exemple : le patron *Strategy*.

### 3.1.1. *Strategy*



Pensez à d'abord réaliser le [TD en annexe](#).

#### Principes de conception

Les bons principes de conception qui sont mis en oeuvre dans ce patron sont les suivants.

##### *Principe de conception*



Identifiez les aspects de votre code qui varient et séparez-les de ceux qui demeurent constant.

##### *Principe de conception*



Programmer une interface, non une implémentation.

##### *Principe de conception*



Préférez la composition à l'héritage.

#### Définition du patron

## Design pattern : Stratégie (Strategy)

**Stratégie** définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Il permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

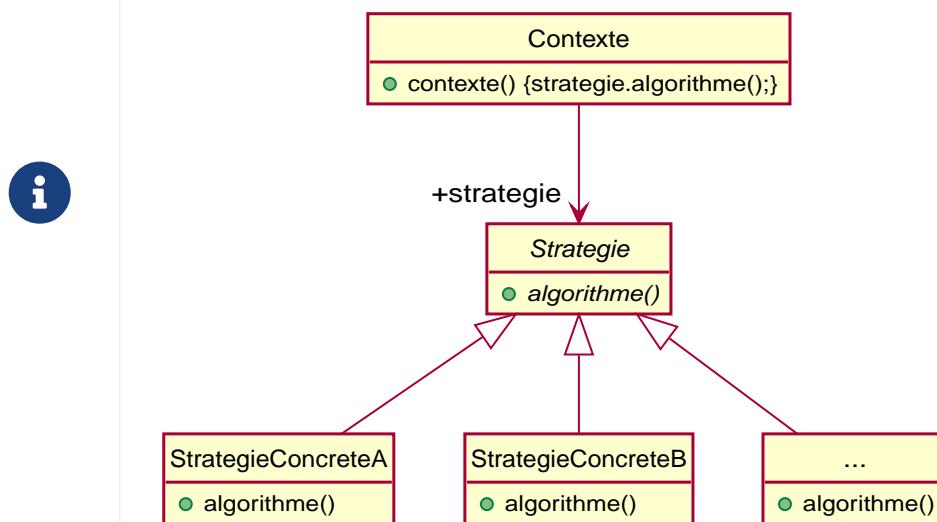


Figure 2. Modèle UML du patron Strategy

## Premier exemple d'utilisation

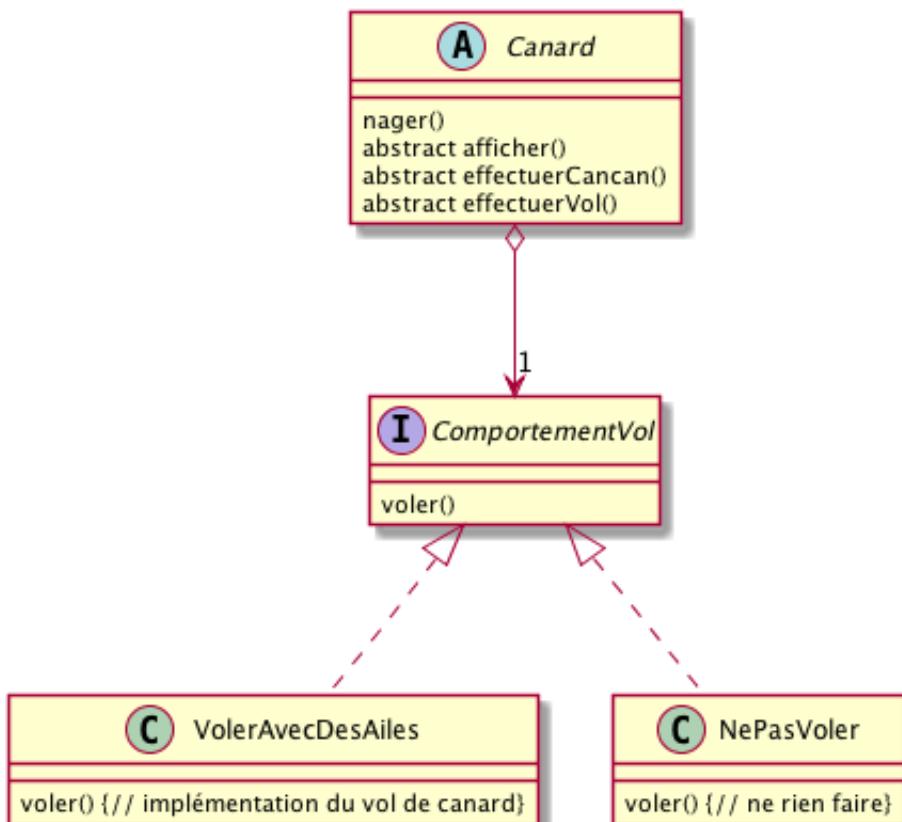


Figure 3. Premier exemple d'utilisation de patron (1er comportement)

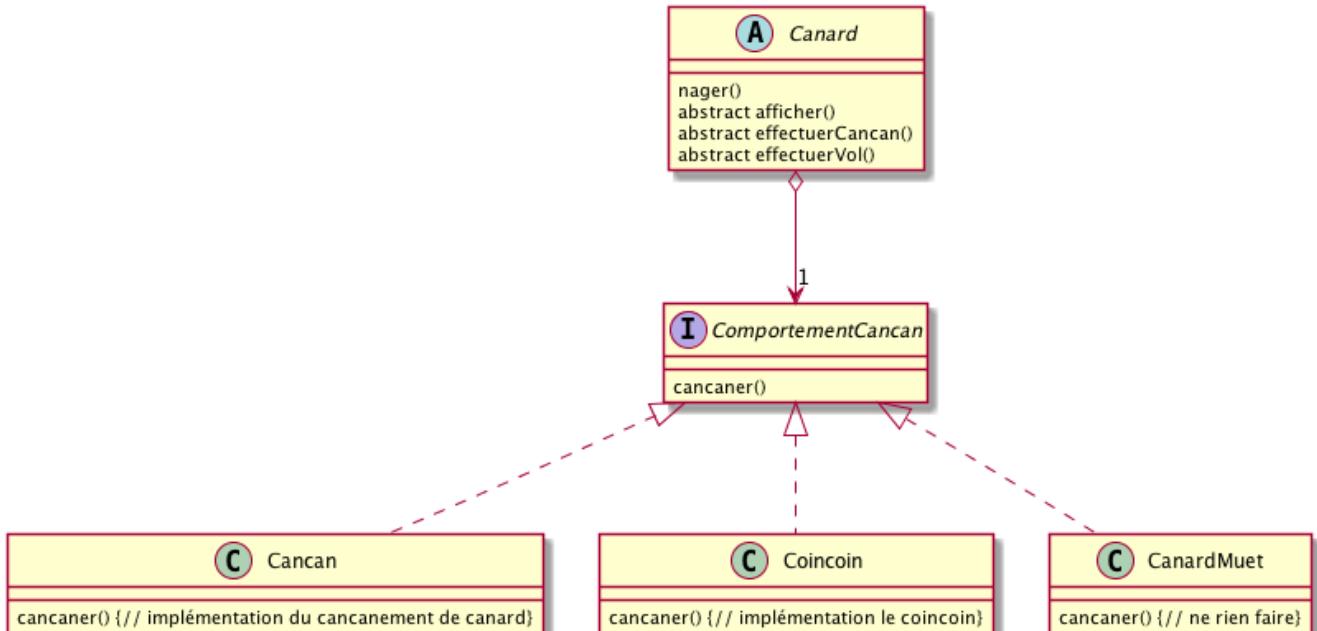


Figure 4. Premier exemple d'utilisation de patron (2ème comportement)

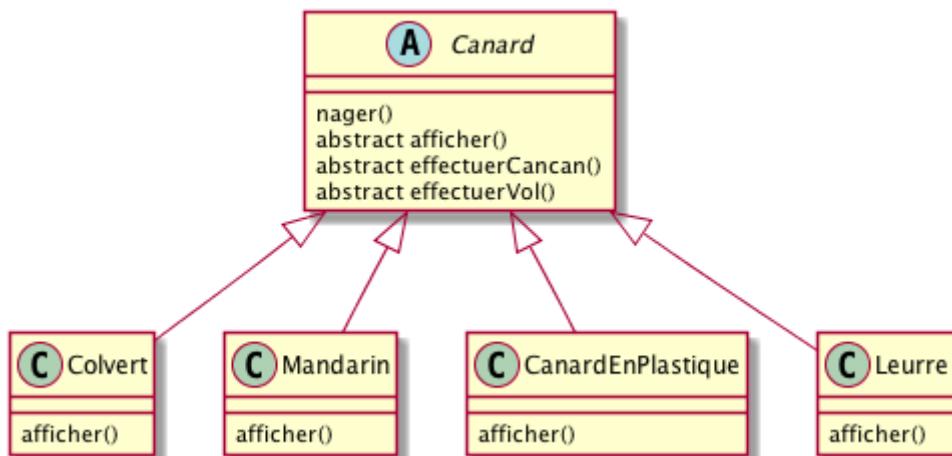


Figure 5. Premier exemple d'utilisation de patron (hiérarchie des classes)



### Question

Pourquoi n'a-t'on pas utilisé *Strategy* pour `afficher()` ou `nager()`?

## Autre exemple concret



L'exemple qui suit est tiré de [ce cours](#).

### Le problème

Vous avez une classe `FileWriter` qui a pour rôle d'écrire dans un fichier ainsi qu'une classe `DBWriter`. Dans un premier temps, ces classes ne contiennent qu'une méthode `write()` qui n'écrira que le texte passé en paramètre.

Au fil du temps, vous vous rendez compte que c'est dommage qu'elles ne fassent que ça et vous aimerez bien qu'elles puissent écrire en différents formats (HTML, XML, etc.) : les classes doivent donc formater puis écrire.

## La solution

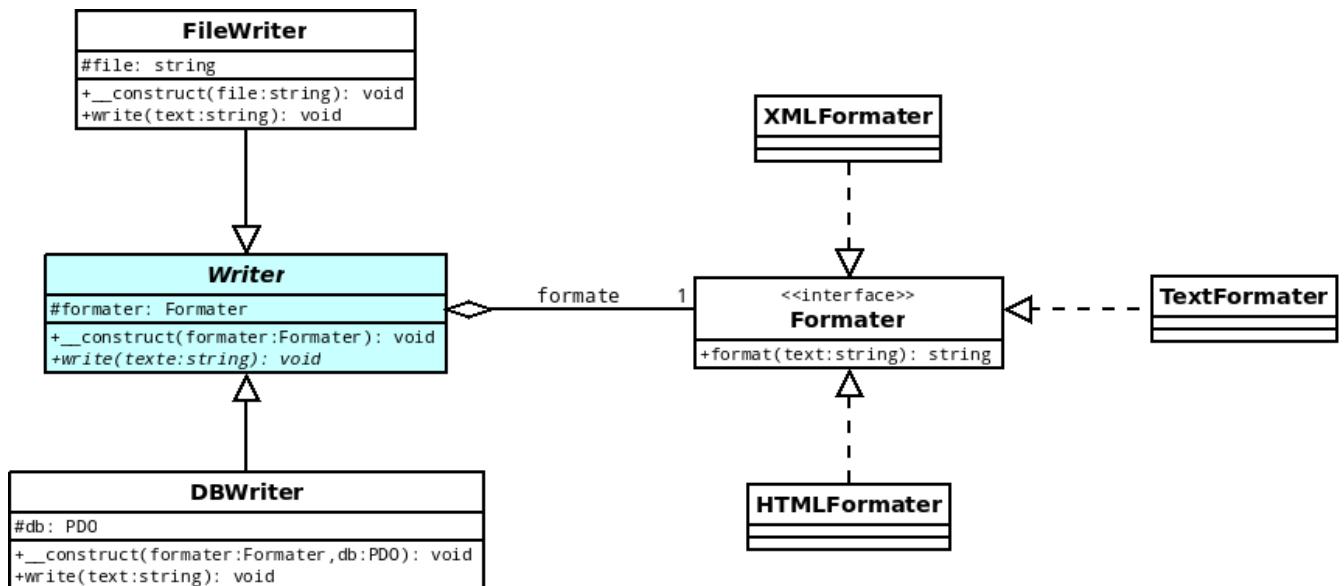


Figure 6. Application du pattern *Strategy* (source)

L'interface en PHP (code source [ici](#))



```
<?php
interface Formater
{
    public function format($text);
}
?>
```

La classe abstraite `Writer` (code source [ici](#))



```
<?php
abstract class Writer
{
    // Attribut contenant l'instance du formateur que l'on veut utiliser.
    protected $formater;

    abstract public function write($text);

    // Nous voulons une instance d'une classe implémentant Formater en
    // paramètre.
    public function __construct(Formater $formater)
    {
        $this->formater = $formater;
    }
}
```

La classe **FileWriter** (code source [ici](#))

```
<?php
class FileWriter extends Writer
{
    // Attribut stockant le chemin du fichier.
    protected $file;

    public function __construct(Formater $formater, $file)
    {
        parent::__construct($formater);
        $this->file = $file;
    }

    public function write($text)
    {
        $f = fopen($this->file, 'w');
        fwrite($f, $this->formater->format($text));
        fclose($f);
    }
}
?>
```



La classe **DBWriter** (code source [ici](#))

```
<?php
class DBWriter extends Writer
{
    protected $db;

    public function __construct(Formater $formater, PDO $db)
    {
        parent::__construct($formater);
        $this->db = $db;
    }

    public function write ($text)
    {
        $q = $this->db->prepare('INSERT INTO lorem_ipsum SET text =
:text');
        $q->bindValue(':text', $this->formater->format($text));
        $q->execute();
    }
}
?>
```



Enfin, nous avons nos trois formateurs. L'un ne fait rien de particulier (**TextFormater**), et les deux autres formatent le texte en deux langages différents (**HTMLFormater** et **XMLFormater**).

*La classe TextFormater (code source [ici](#))*

```
<?php
class TextFormater implements Formater
{
    public function format($text)
    {
        return 'Date : ' . time() . "\n" . 'Texte : ' . $text;
    }
}
?>
```

*La classe HTMLFormater (code source [ici](#))*



```
<?php
class HTMLFormater implements Formater
{
    public function format($text)
    {
        return '<p>Date : ' . time() . '<br />' . "\n". 'Texte : ' . $text .
'</p>';
    }
}
?>
```

*La classe XMLFormater (code source [ici](#))*

```
<?php
class XMLFormater implements Formater
{
    public function format($text)
    {
        return '<?xml version="1.0" encoding="ISO-8859-1"?>' . "\n".
'<message>' . "\n".
"\t". '<date>' . time() . '</date>' . "\n".
"\t". '<texte>' . $text . '</texte>' . "\n".
'</message>';
    }
}
?>
```

## D'autres exemples

- La fonction standard `sort()` de python

```
>>> sorted("This is a test string from Andrew".split(), key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

- Stratégie de cryptage en fonction de la taille d'un fichier

```

File file = getFile();
Cipher c = CipherFactory.getCipher( file.size() );
c.performAction();

// implementations:
interface Cipher {
    public void performAction();
}

class InMemoryCipherStrategy implements Cipher {
    public void performAction() {
        // load in byte[] ....
    }
}

class SwapToDiskCipher implements Cipher {
    public void performAction() {
        // swap partial results to file.
    }
}

```



Plus de détails [ici](#)

### 3.1.2. (non) Réutilisation



Les patrons **ne sont pas réutilisables!**

Il faut implémenter la solution qu'il représente à chaque fois.

Exception : certains font l'objet d'une librairie (comme *Observer* de Java).

Par exemple le patron Singleton existe dans la bibliothèque standard du langage en Ruby. C'est un *mixin* qu'il suffit d'inclure dans la classe qui doit être un singleton.

```

class Klass
  include Singleton
  # ...
end

a,b = Klass.instance, Klass.instance

a == b
# => true

Klass.new
# => NoMethodError - new is private ...

```

### 3.1.3. Association ou composition

On trouve deux modèles UML™ :

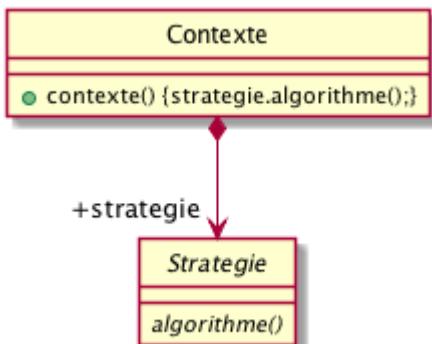


Figure 7. Strategy et composition

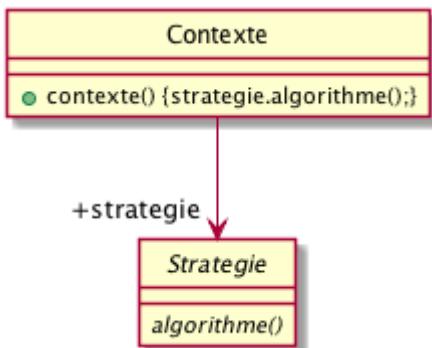


Figure 8. Strategy et association

Et donc deux implémentations :

*Composition ⇒ le composé encapsule les composants*

```
public class Colvert extends Canard {  
  
    protected Colvert() {  
        this(new VolerAvecDesAiles(), new Cancan());  
    }  
  
    ...  
    c1 = new Colvert();
```

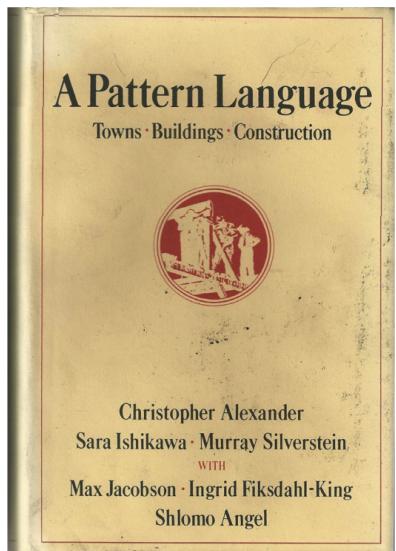
*Association ⇒ le composant existe "en dehors"*

```
...  
vol = new VolerAvecDesAiles();  
cri = new Cancan();  
c1 = new Colvert(vol,cri);  
...
```

## 3.2. Un peu d'histoire

Voici un bref point sur les moments clefs qui ont permis de définir les patrons de conception.

1977



1987

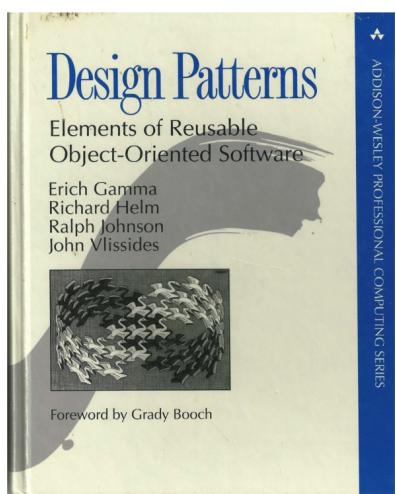
Beck et Cunningham : patterns pour des interfaces utilisateurs

1988

Meyer : livre sur l'orienté objet (langage [Eiffel](#)), devenu la bible pour beaucoup de programmeurs (cf. [\[Meyer88\]](#))

1990-1995

Gamma, Helm, Johnson et Vlissides : LE livre de référence (cf. [\[GoF\]](#))



Les auteurs de ce livre sont connus comme les **Gof** pour « *Gang of Four* ».

2003

Martin : principes SOLID (cf. [\[Martin03\]](#))

2004

Craig Larman décrit des modèles de conception : les Patterns GRASP (cf. [\[Larman05\]](#))

### 3.3. Exemples de bons principes

Nous ne pouvons être exhaustifs sur la liste des bons principes objets ni sur de longues explications sur chacun d'eux (une fois encore nous renvoyons le lecteur à [\[Meyer\]](#) pour les fondements), mais nous examinons dans ce qui suit quelques exemples clefs.

**SOLID:**

- *Single Responsibility Principle*
- *Open-Closed Principle*
- *Liskov Substitution Principle*
- *Interface Segregation Principle*
- *Dependency Inversion Principle*

#### 3.3.1. Single Responsibility Principle



Figure 9. Single Responsibility Principle (source [\[SOLID\]](#))

Responsabilité => Sujet à changement

Autrement dit, limitez le plus possible ce que doit réaliser une classe. Mieux vaut combiner plusieurs classes qui chacune fait bien ce qu'elle doit faire.

### 3.3.2. Open-Closed Principle



Figure 10. Open-Closed Principle (source [\[SOLID\]](#))

Ouvert à l'extension mais fermé à la modification

Ainsi, une fois écrite et testée, une classe ne devrait être modifiée que pour être corrigée! Toute modification devrait être possible par extension.

### 3.3.3. Liskov Substitution Principle

Barbara Liskov, pionnière en informatique et plus précisément en OO, a donné son nom à un principe important et bien connu : le principe default: substitution de Liskov. Elle a reçu l'équivalent du prix Nobel d'Informatique (le *Turing Award*) en 2009.

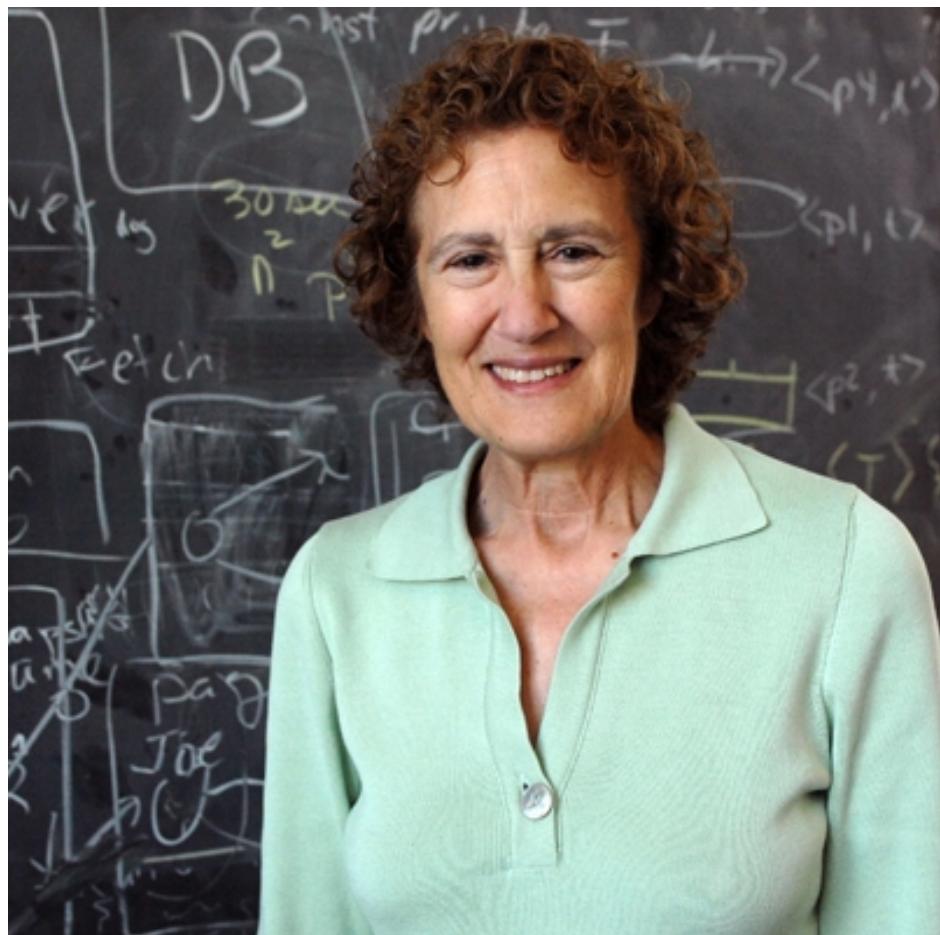


Figure 11. Barbara Liskov reçoit le Turing Award

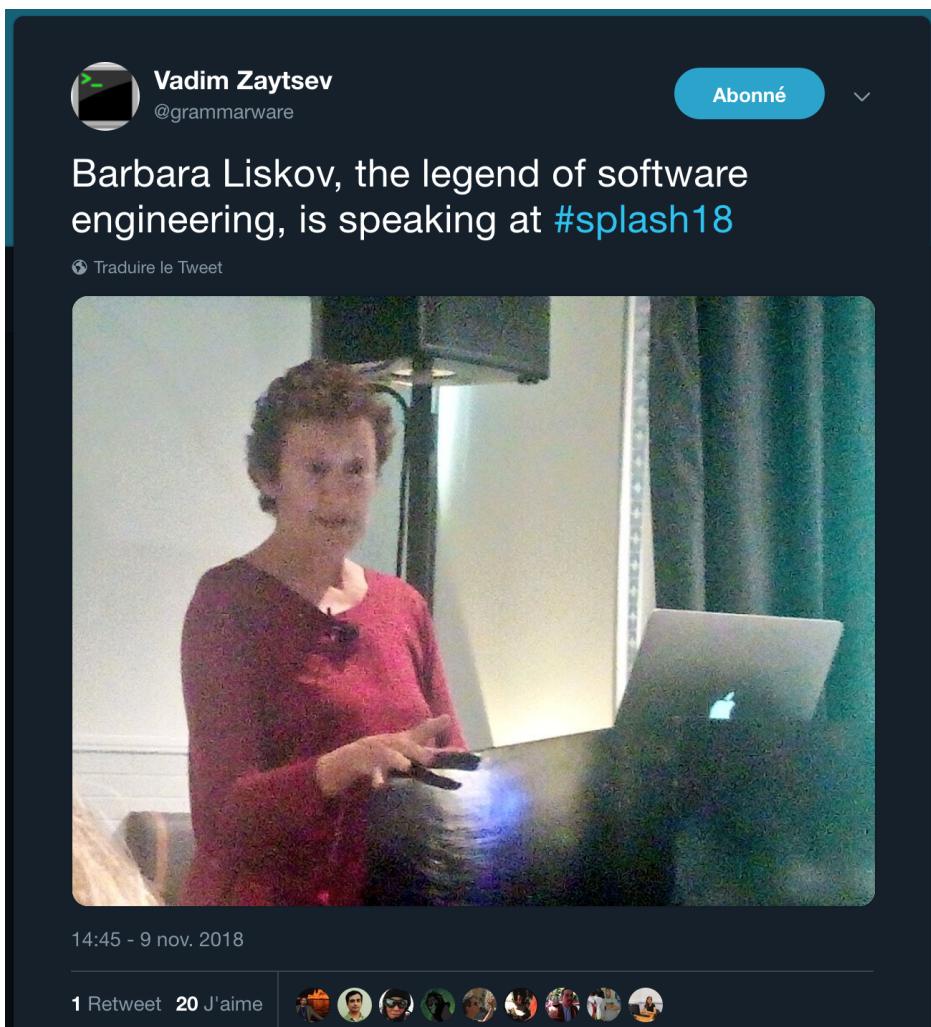


Figure 12. Barbara Liskov est toujours active!

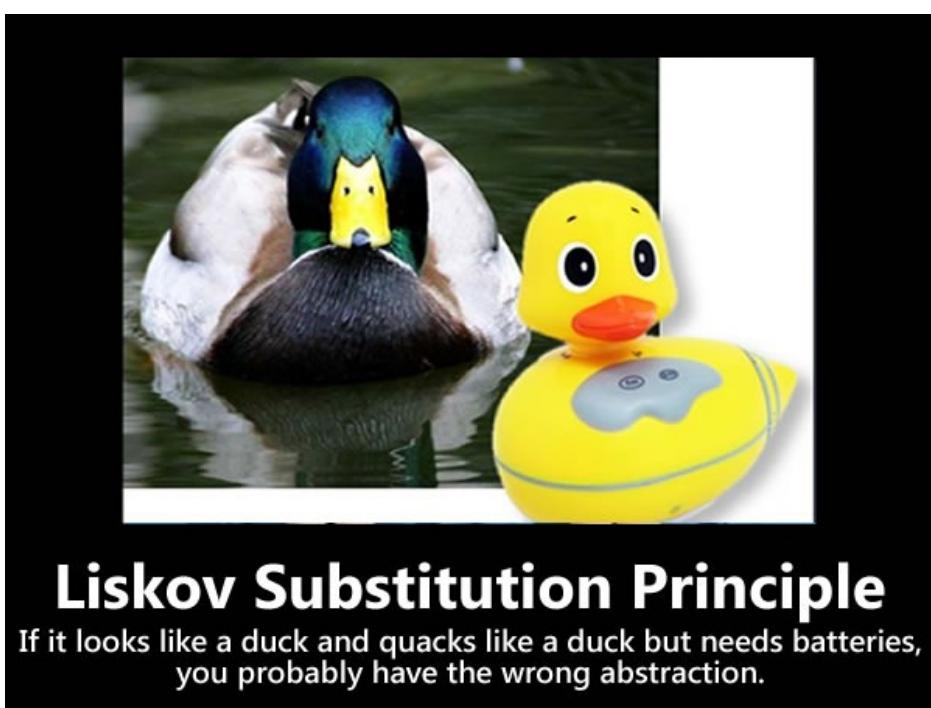


Figure 13. Liskov Substitution Principle (source [SOLID])

Une classe doit pouvoir être remplacée par une instance d'un de ses sous-types, sans modifier la cohérence du programme

Un carré est un rectangle particulier.

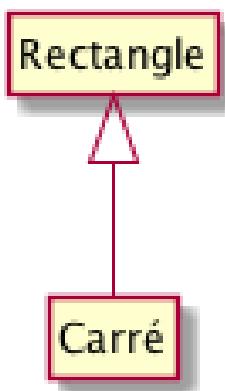


Figure 14. Exemple classique de violation du principe de substitution de Liskov



*Question*

Peut-on toujours substituer un **Carré** à la place d'un **Rectangle** ?

Examinons avec un exemple concret pourquoi la réponse est non.

Réponse ([Rectangle.java](#))

```
class Rectangle
{
    protected int m_width;
    protected int m_height;

    public void setWidth(int width){
        m_width = width;
    }

    public void setHeight(int height){
        m_height = height;
    }

    public int getWidth(){
        return m_width;
    }

    public int getHeight(){
        return m_height;
    }

    public int getArea(){
        return m_width * m_height;
    }
}
```

Réponse ([Square.java](#))

```
// Violation of Likov's Substitution Principle
class Square extends Rectangle
{
    public void setWidth(int width){
        m_width = width;
        m_height = width;
    }

    public void setHeight(int height){
        m_width = height;
        m_height = height;
    }

}
```

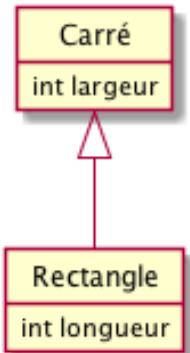
Réponse ([Square.java](#) - suite)

```
class LspTest
{
    private static Rectangle getNewRectangle()
    {
        // it can be an object returned by some factory ...
        return new Square();
    }

    public static void main (String args[])
    {
        Rectangle r = LspTest.getNewRectangle();

        r.setWidth(5);
        r.setHeight(10);
        // User knows that r is a rectangle.
        // It assumes that he's able to set the width and height as for the base class

        System.out.println(r.getArea());
        // Now she's surprised to see that the area is 100 instead of 50.
    }
}
```



*Un rectangle est un carré avec une dimension de plus...*

Figure 15. Et si on essaye l'inverse

Et bien non, ça ne fonctionne pas plus dans l'autre sens.

### Réponse (`Rectangle.java`)

```
class LspTest
{
    private static Square getNewSquare()
    {
        // it can be an object returned by some factory ...
        return new Rectangle();
    }

    public static void main (String args[])
    {
        Square s = LspTest.getNewSquare();

        s.setWidth(5);
        // User knows that r is a rectangle.
        // It assumes that he's able to set the width and height as for the base class

        System.out.println(s.getArea());
        // Now she's surprised to see that the area is 0 instead of 25.
    }
}
```

### 3.3.4. Interface Segregation Principle

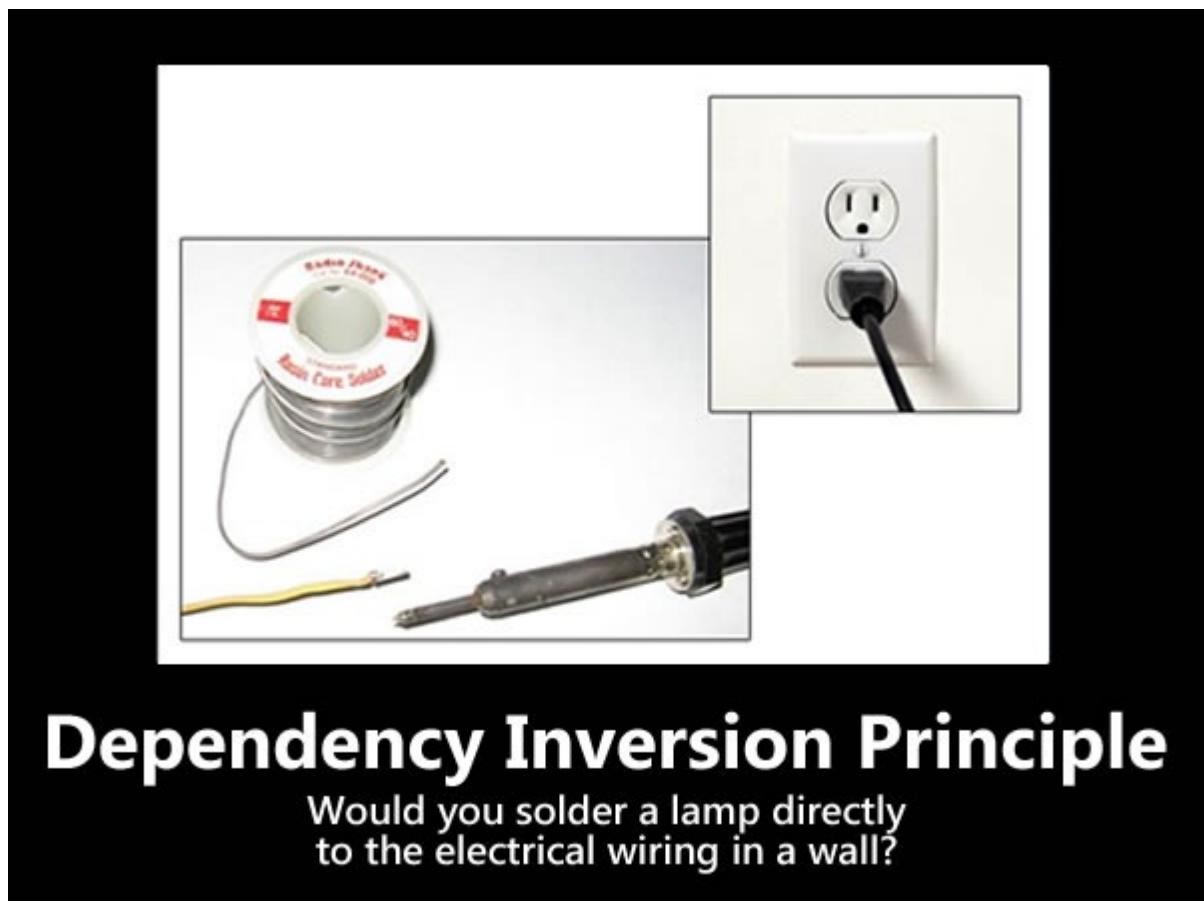


Figure 16. Interface Segregation Principle (source [\[SOLID\]](#))

Préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale

C'est un peu le même principe que la *Single Responsibility* des classes, mais appliqué aux interfaces.

### 3.3.5. Dependency Inversion Principle



## Dependency Inversion Principle

Would you solder a lamp directly  
to the electrical wiring in a wall?

Figure 17. Interface Segregation Principle (source [\[SOLID\]](#))

Il faut dépendre des abstractions, pas des implémentations

Ce principe indique :

- Les modules de haut niveau (abstraits) ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
- Les abstractions ne doivent pas dépendre des détails d'implémentation. C'est l'inverse : les détails doivent dépendre des abstractions.



Ainsi ce principe va à l'encontre de l'intuition classique.

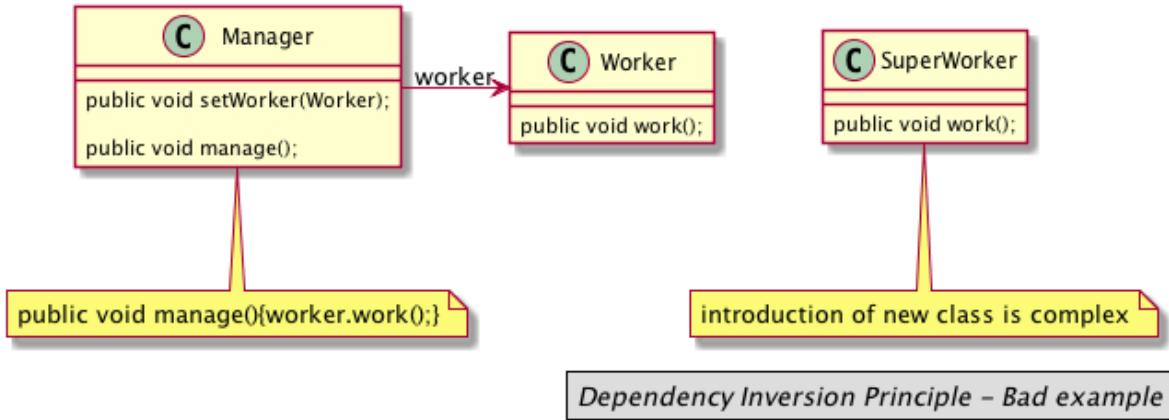


Figure 18. Exemple de code violant le principe d'inversion des dépendances

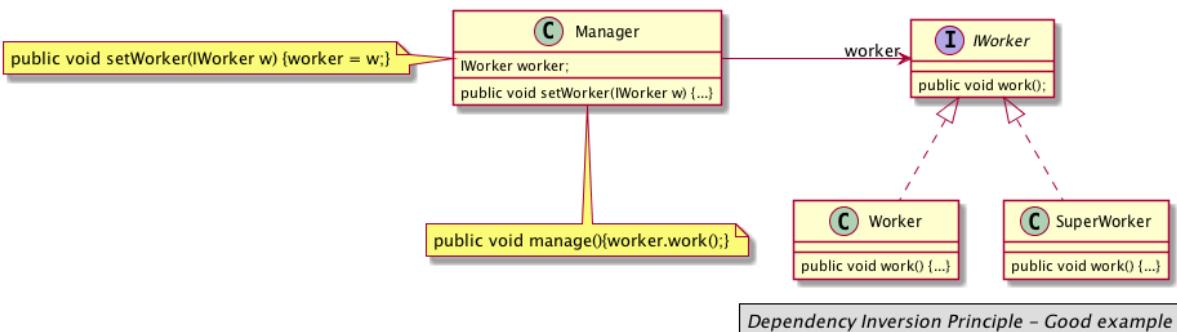


Figure 19. Exemple de code ne violant plus le principe d'inversion des dépendances

### 3.3.6. SOLID et patrons



#### QUESTION

Lesquels des 5 principes SOLID s'appliquent bien à *Strategy* ?

Quelques éléments de réponses :

#### *Single Responsibility Principle*

Bof

#### *Open-Closed Principle*

⇒ Oui : extension (du comportement) sans toucher au code!

#### *Liskov Substitution Principle*

Non

#### *Interface Segregation Principle*

Oui, mais pas spécifiquement

#### *Dependency Inversion Principle*

⇒ Oui : les algos dépendent des mêmes abstractions que les données (les interfaces)

### 3.3.7. GRASP

The critical design tool for software development is a **mind well educated in design principles**. It is not the UML or any other technology.

— Craig Larman, 2005

Il s'agit d'un ensemble de patrons, plutôt orientés conception (UML). Nous en aborderons certains au travers des exemples de ce module (cf. [\[Larman05\]](#)).



Notez que les principes SOLID ne s'appliquent pas qu'à la programmation objet.

Pour une discussion sur leur application avec React (language fonctionnel), cf.

<https://dev.to/shadid12/can-you-apply-solid-principles-to-your-react-applications-46il>.

## 3.4. Les patrons : comment ça marche ?

### 3.4.1. Intérêt

- Réponses éprouvées à des problèmes récurrents
- Vocabulaire commun

Exemple de phrase qu'on entend dans un *open space* de programmeurs (et qui justifie qu'à défaut de les connaître tous, il faut savoir rapidement se documenter et les comprendre) :

T'as qu'à utiliser une *factory*!

Pour chaque patron étudié dans ce livre nous aurons les mêmes éléments de définition.

### 3.4.2. Patrons abordés

Voici la liste des patrons abordés dans ce cours.

- [Singleton](#)
- [Stratégie](#)
- [Le patron Fabrique \(\*factory\*\)](#)
- [Observateur](#)
- [Itérateur](#)
- [Le patron Composite](#)
- [Etat](#)
- [Le patron Visiteur](#)
- [Proxy](#)
- [Adaptateur](#)

### 3.4.3. Patrons non abordés

Voici une liste (non-exhaustive) des patrons **non abordés** dans ce cours (pour donner une idée du chemin qu'il vous reste à parcourir).

- Décorateur
- Commande
- Façade
- Patron de méthode
- Chaînes de responsabilité
- Prototype
- Memento
- Médiateur
- Interprète
- Poids-mouche
- Monteur
- Pont

## 4. Le patron Fabrique

### 4.1. Principes de conception

*Design pattern : Fabrique (Factory)*

Fabrique (simple) définit une interface pour la création d'un objet, mais en laissant à des sous-classes le choix des classes à instancier (voir aussi [Fabrique abstraite](#)).

i

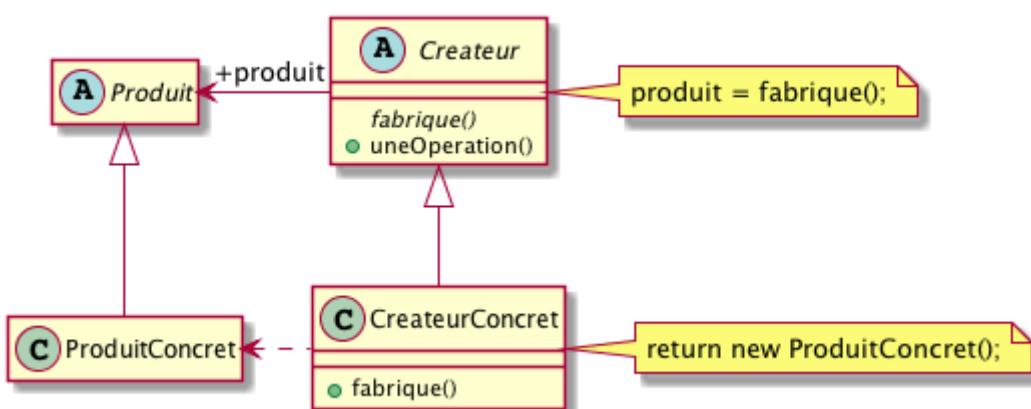


Figure 20. Modèle UML du patron Fabrique

## 4.2. Premier exemple d'utilisation de patron

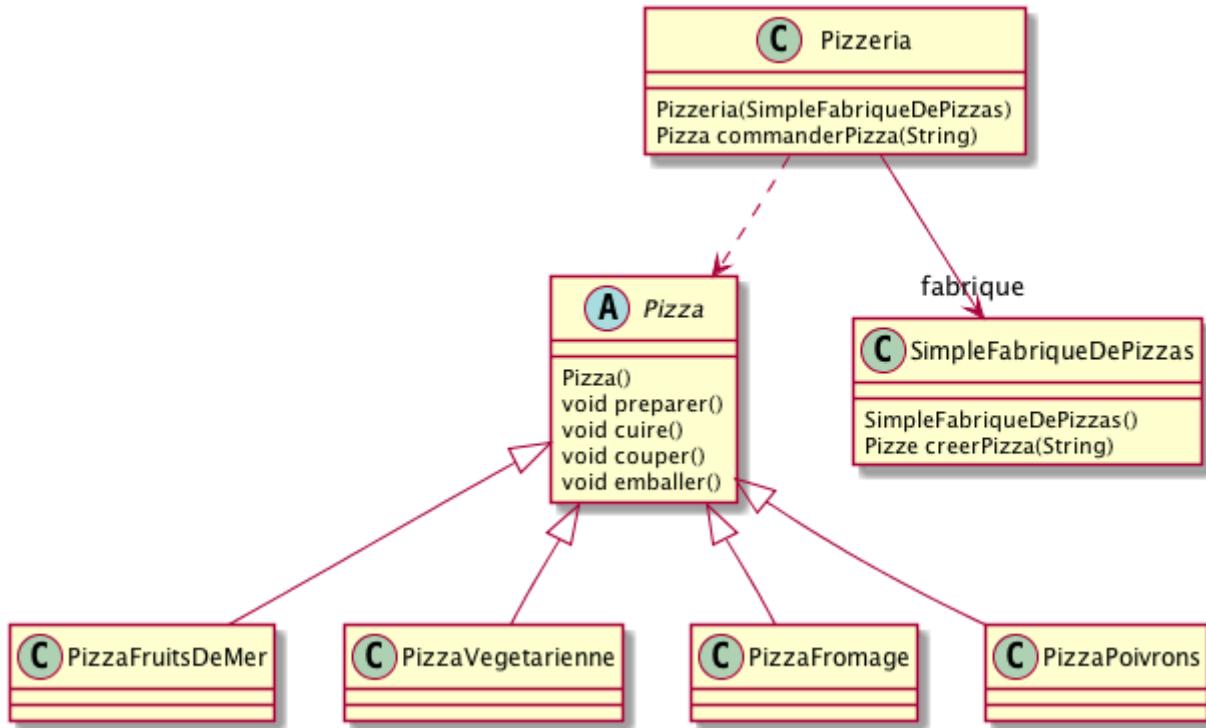


Figure 21. 1er exemple d'utilisation de Fabrique

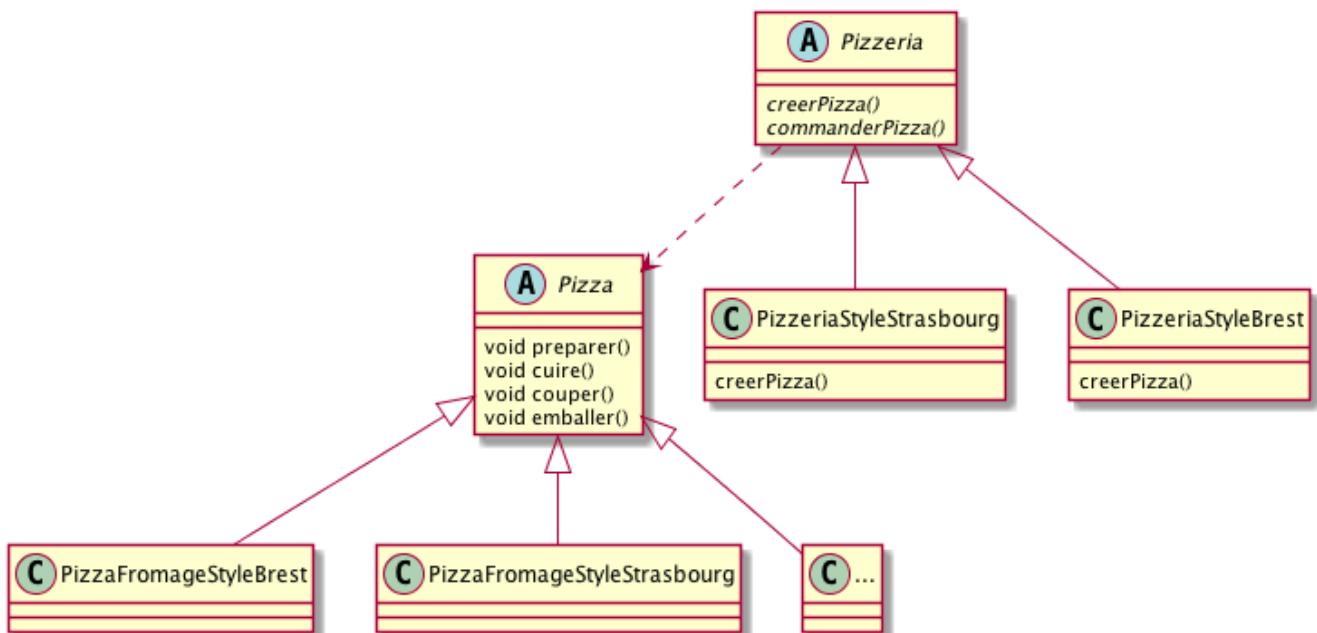


Figure 22. Exemple complet d'utilisation de Fabrique

## 4.3. Autre exemple concret

Factory en PHP (source [ici](#))

```
<?php
class DBFactory
{
    public static function load($sgbdr)
    {
        $classe = 'SGBDR_' . $sgbdr;

        if (file_exists($chemin = $classe . '.class.php'))
        {
            require $chemin;
            return new $classe;
        }
        else
        {
            throw new RuntimeException('La classe <strong>' . $classe . '</strong> n\'a pu
être trouvée !');
        }
    }
}
?>
```

Factory en PHP (source [ici](#))

```
<?php
try
{
    $mysql = DBFactory::load('MySQL');
}
catch (RuntimeException $e)
{
    echo $e->getMessage();
}
?>
```

Factory en java (source : votre module AA!)

```
public class XmlExprParser {  
  
    public static Expression fromFile(String file) throws ... {  
        SAXParserFactory spf = SAXParserFactory.newInstance();  
        spf.setValidating(true);  
        SAXParser sp = spf.newSAXParser();  
        ExprHandler ep = new ExprHandler();  
        sp.parse(file, ep);  
        return ep.getResult();  
    }  
  
}
```

## 4.4. Un autre exemple concret

Factory en Java (source [ici](#))

```
MazeGame ordinaryGame = new OrdinaryMazeGame();  
MazeGame magicGame = new MagicMazeGame();
```

```
public abstract class MazeGame {  
    private final List<Room> rooms = new ArrayList<>();  
  
    public MazeGame() {  
        Room room1 = makeRoom();  
        Room room2 = makeRoom();  
        room1.connect(room2);  
        rooms.add(room1);  
        rooms.add(room2);  
    }  
  
    abstract protected Room makeRoom();  
}
```

```

public abstract class Room {
    abstract void connect(Room room);
}

public class MagicRoom extends Room {
    public void connect(Room room) {}
}

public class OrdinaryRoom extends Room {
    public void connect(Room room) {}
}

```

```

public class MagicMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new MagicRoom();
    }
}

public class OrdinaryMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new OrdinaryRoom();
    }
}

```

## 4.5. Mais c'est pas fini!

Reprenons nos pizzas vues en TD

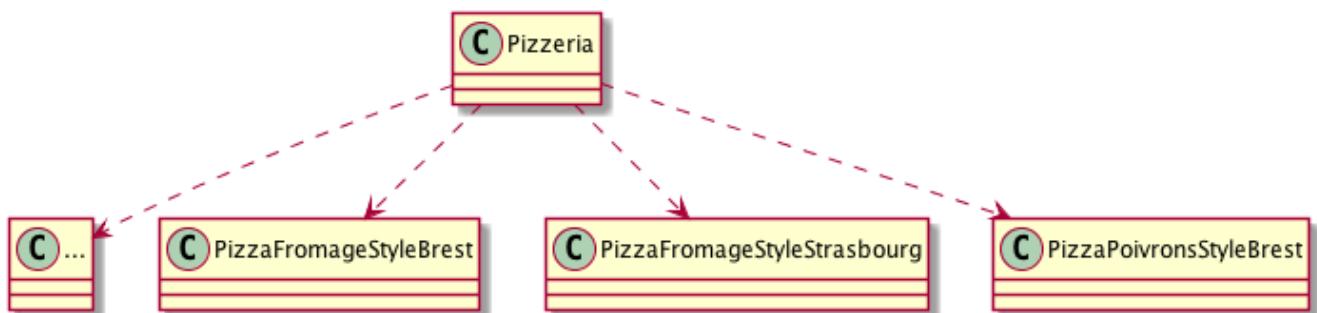


Figure 23. Une pizzeria dépendante

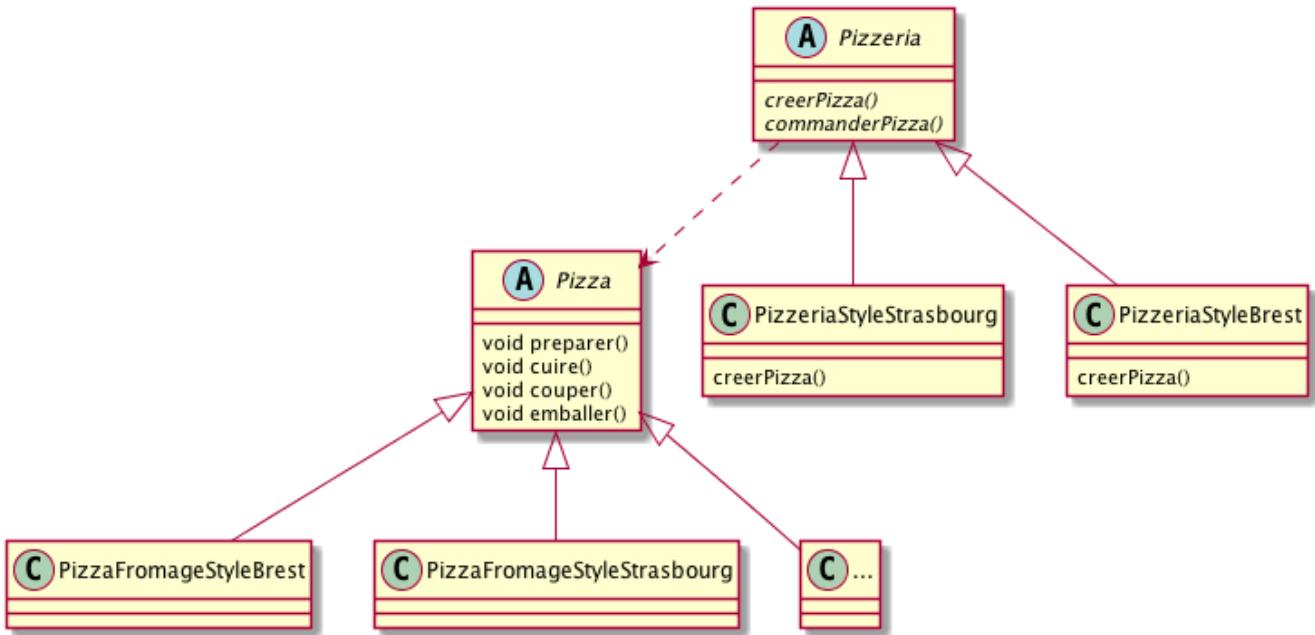


Figure 24. Une pizzeria indépendante

- Aucune variable ne doit contenir une référence à une classe concrète.
- Aucune classe ne doit dériver d'une classe concrète.
- Aucune classe ne doit redéfinir une méthode implémentée dans une classe de base.

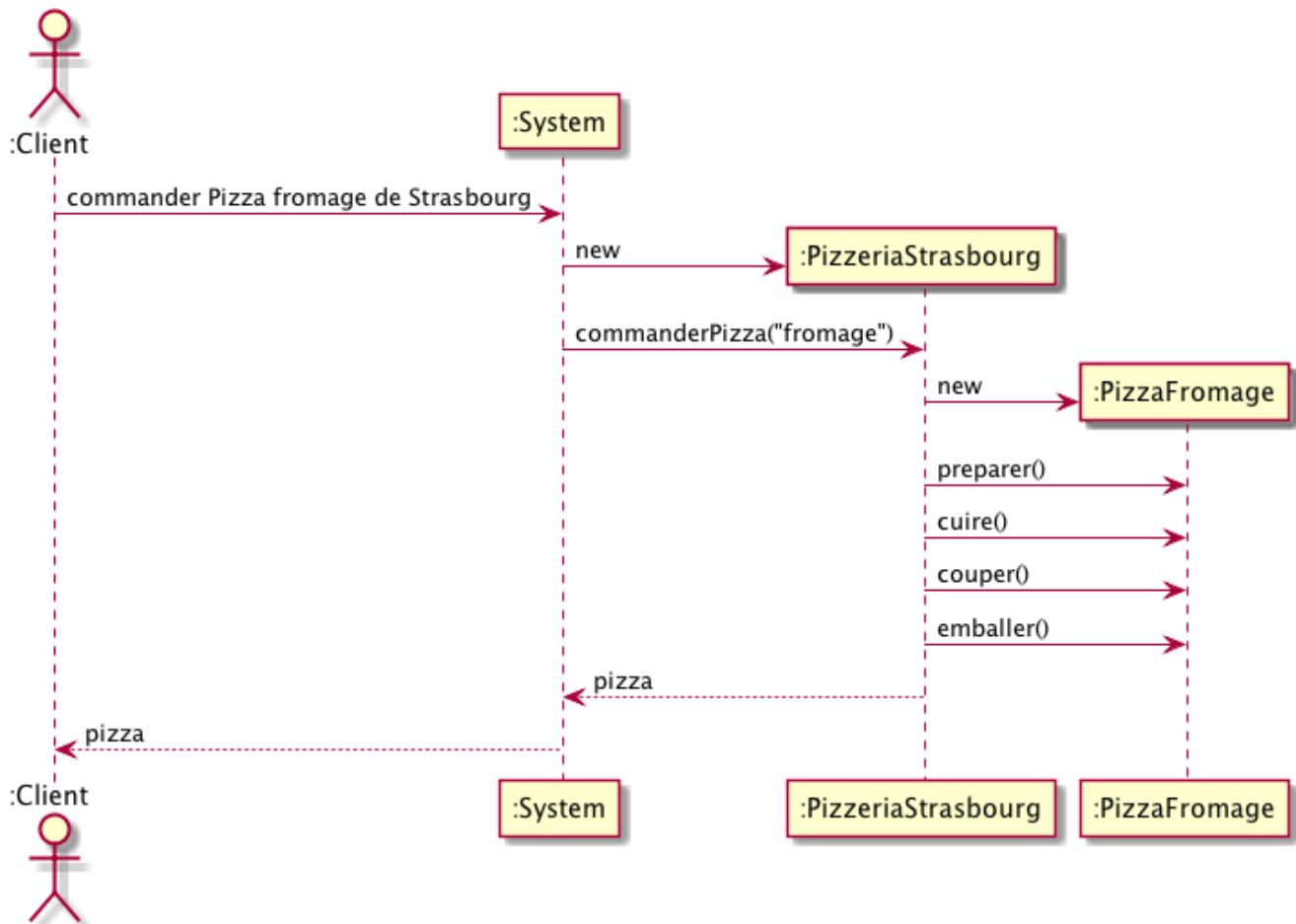


Figure 25. Une pizzeria avec Fabrique

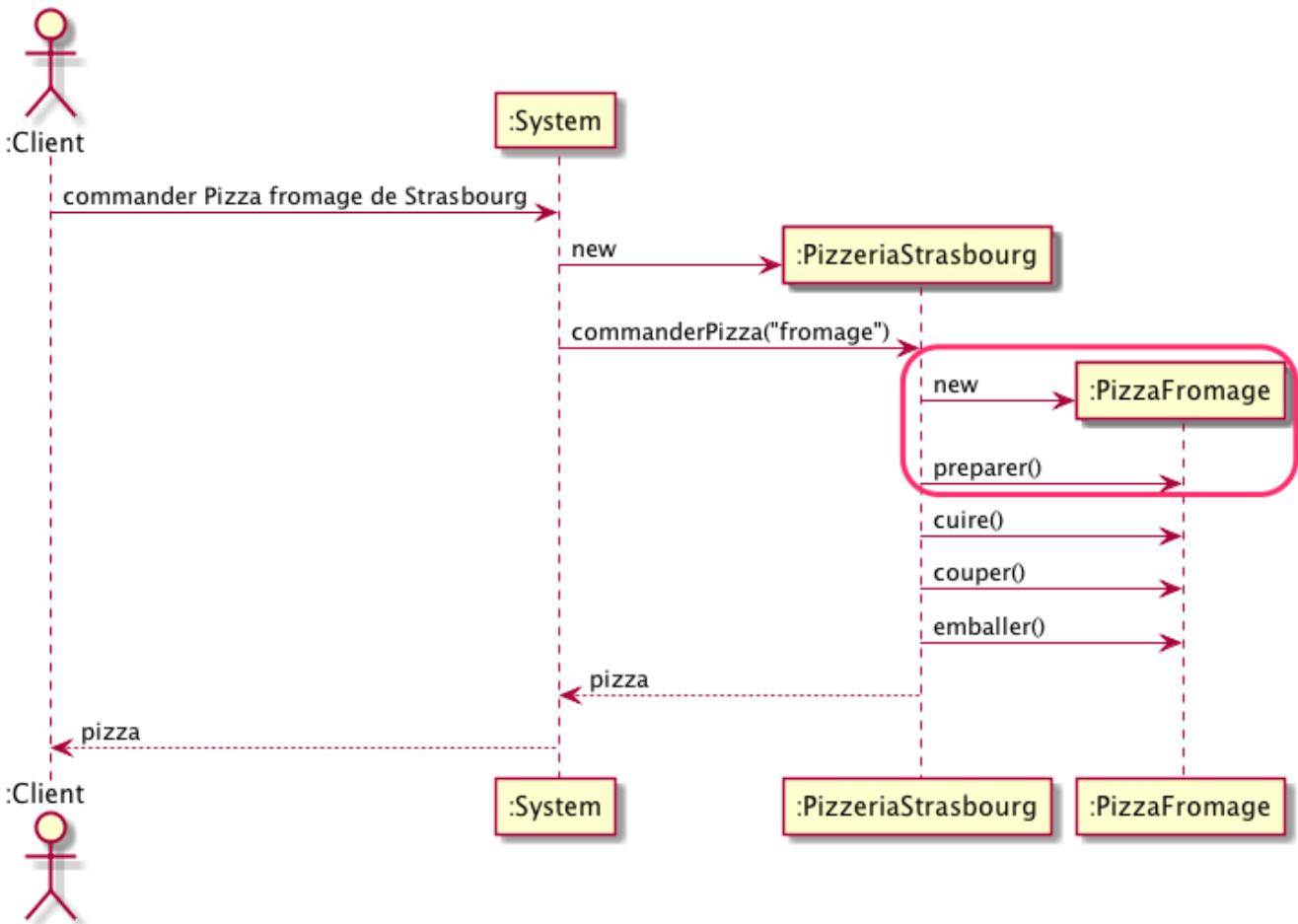


Figure 26. Problème de la dépendance des ingrédients



Figure 27. Des cartes adaptées (source [Freeman04])

```
public interface FabriqueIngredientsPizza {  
    public Pate creerPate();  
    public Sauce creerSauce();  
    public Fromage creerFromage();  
    public Legumes[] creerLegumes();  
    public Poivrons creerPoivrons();  
    public Moules creerMoules();  
}
```

```
public class FabriqueIngredientsPizzaBrest implements FabriqueIngredientsPizza {  
    public Pate creerPate() {  
        return new PateFine();  
    }  
    public Sauce creerSauce() {  
        return new SauceMarinara();  
    }  
    ...  
}
```

```
public class FabriqueIngredientsPizzaStrasbourg implements FabriqueIngredientsPizza {  
    public Pate creerPate() {  
        return new PateEpaisse();  
    }  
    public Sauce creerSauce() {  
        return new SauceTomateCerise();  
    }  
    ...  
}
```

```
public class PizzaFromage extends Pizza {  
    FabriqueIngredientsPizza fabriqueIngredients;  
  
    public PizzaFromage(FabriqueIngredientsPizza fabriqueIngredients) {  
        this.fabriqueIngredients = fabriqueIngredients;  
    }  
    void preparer() {  
        System.out.println("Préparation de " + nom);  
        pate = fabriqueIngredients.creerPate();  
        sauce = fabriqueIngredients.creerSauce();  
        fromage = fabriqueIngredients.creerFromage();  
    }  
}
```

## 4.6. Fabrique abstraite

Nous sommes arrivé à une version du patron Fabrique appelée **Fabrique Abstraite** :

**Fabrique** (abstraite) fournit une interface pour la création de familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes (voir aussi [Fabrique](#)).

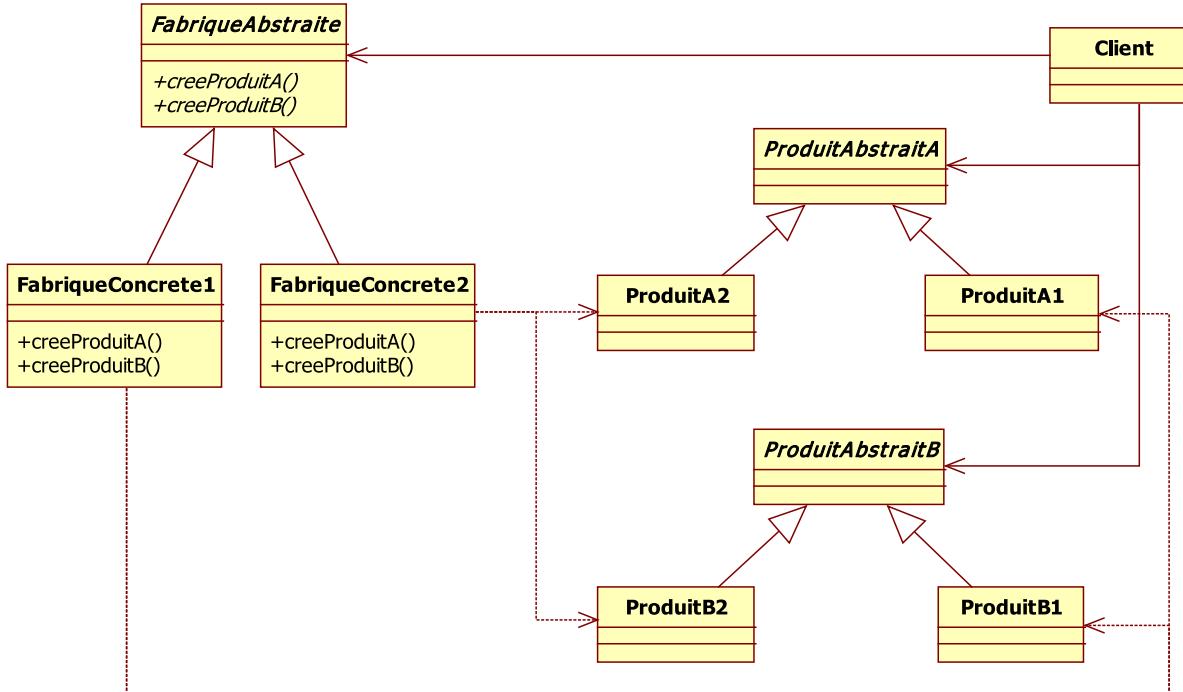


Figure 28. Modèle UML du patron Fabrique Abstraite

## 5. Etat

Soit la machine à état suivante :

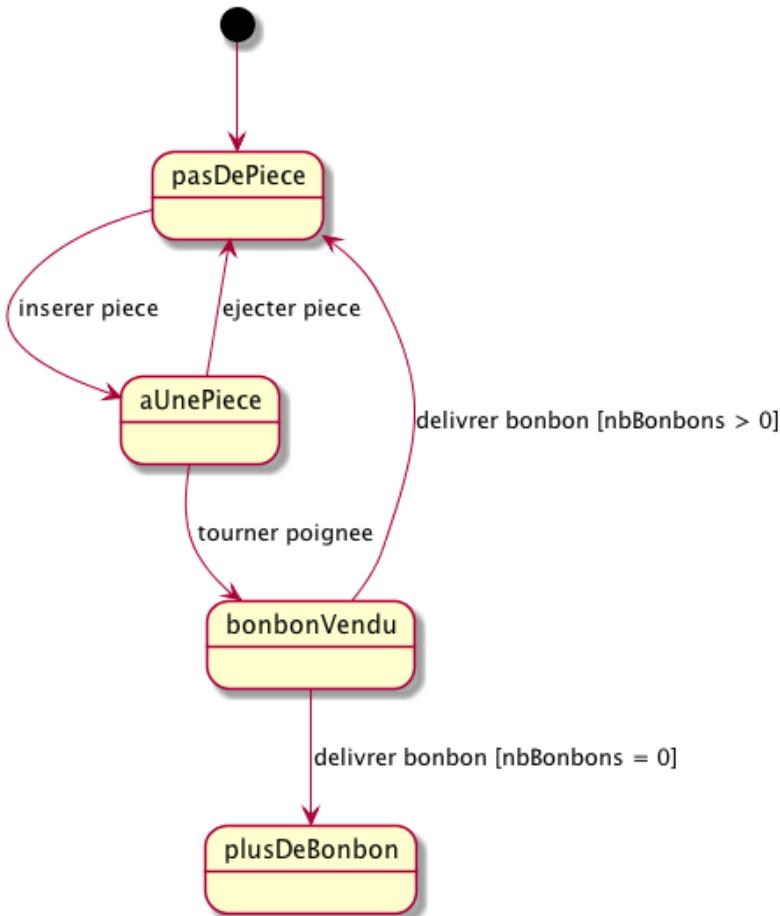


Figure 29. Machine à état d'un distributeur simple

## 5.1. Implémentation intuitive

*Implémentation sans switch case*

```

public void insererPiece() {
    if (etat == A_PIECE) {
        System.out.println("Vous ne pouvez plus insérer de pièces");
    } else if (etat == EPUISE) {
        System.out.println("Vous ne pouvez pas insérer de pièce, nous sommes en rupture de stock");
    } else if (etat == VENDU) {
        System.out.println("Veuillez patienter, le bonbon va tomber");
    } else if (etat == SANS_PIECE) {
        etat = A_PIECE;
        System.out.println("Vous avez inséré une pièce");
    }
}

```

## 5.2. Erreur d'implémentations

- Ce code n'adhère pas au principe Ouvert-Fermé.
- Cette conception n'est pas orientée objet.

- Les transitions ne sont pas explicites. Elles sont enfouies au milieu d'un tas d'instructions conditionnelles.
- Nous n'avons pas encapsulé ce qui varie.
- Les ajouts ultérieurs sont susceptibles de provoquer des bugs dans le code.

## 5.3. Une meilleure implémentation

1. Définir une nouvelle interface **Etat** qui contiendra une méthode pour chaque action
2. Implémenter une classe pour chaque **Etat**. Elles seront responsable du comportement.
3. Se débarrasser de toutes les instructions conditionnelles et les remplacer par une délégation à la classe adéquate.

## 5.4. Illustration

Etape 1 : les états comme implémentations d'une interface

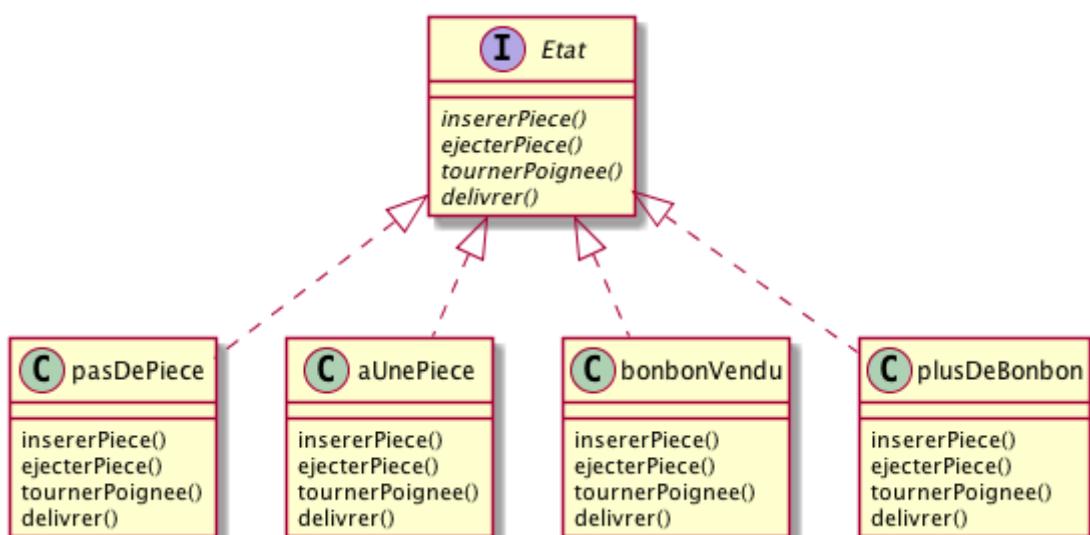


Figure 30. Implémentation des états

Etape 2 : implémentation des méthodes de l'interface

```

public class EtatSansPiece implements Etat {

    // Va falloir remplir ici...

    public void insererPiece() {
        System.out.println("Vous avez inséré une pièce");
        // changer d'état si besoin

    }
    ...
}
  
```

### Etape 3 : utilisation

```
public class Distributeur {  
  
    Etat etat = new EtatSansPiece(); // état initial  
    ...  
    public void insererPiece() {  
        etat.insererPiece(); // on délègue à l'état le soin de réagir  
    }  
    ...  
}
```

### Etape 4 (enfin, retour sur l'étape 2) : une solution possible...

```
public class EtatSansPiece implements Etat {  
    Distributeur distributeur; // référence au distributeur qu'on gère  
  
    public EtatSansPiece(Distributeur distributeur) {  
        this.distributeur = distributeur;  
    }  
  
    public void insererPiece() {  
        System.out.println("Vous avez inséré une pièce");  
        distributeur.setEtat(distributeur.getEtatAPiece());  
    }  
    ...  
}
```

## 5.5. Le patron Etat

**Etat** permet à un objet de modifier son comportement, quand son état interne change. Tout se passe comme si l'objet changeait de classe.

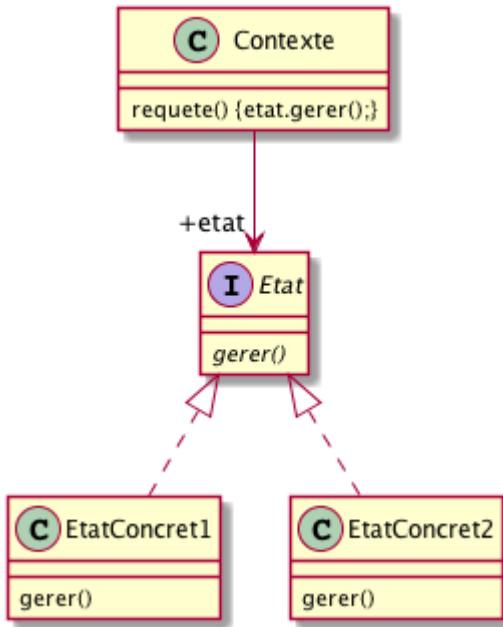


Figure 31. Modèle UML du patron Etat



### QUESTION

Que pensez-vous de notre solution précédente par rapport à ce diagramme UML?

L'état possède une référence vers le contexte (**Distributeur** dans notre exemple).

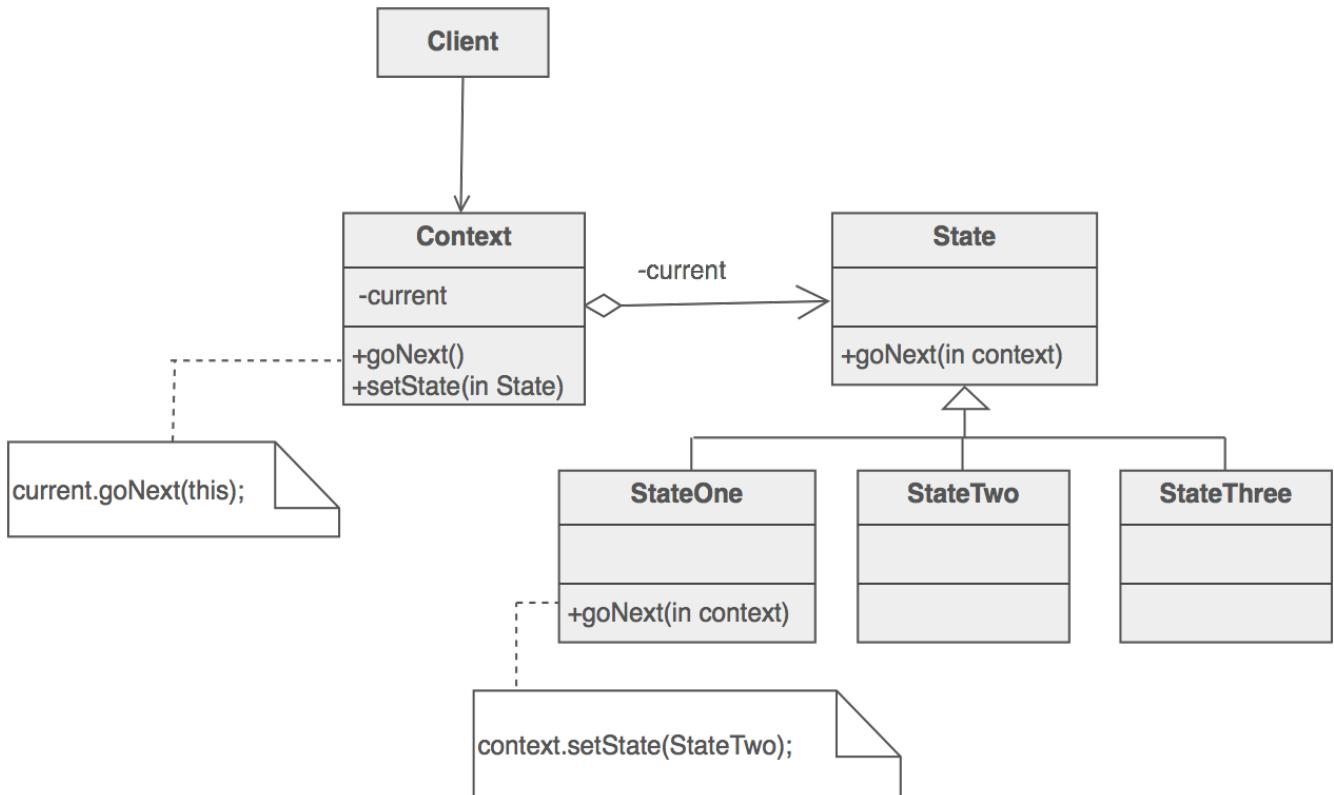


Figure 32. Une autre implémentation (source : [https://sourcemaking.com/design\\_patterns/state](https://sourcemaking.com/design_patterns/state))

## 6. Observateur



## 6.1. Motivation

Comment s'assurer que quand une donnée est modifiée, tous ceux qui l'utilisent en soient informés et réagissent en conséquence. Il s'agit d'une situation très souvent rencontrée en informatique.

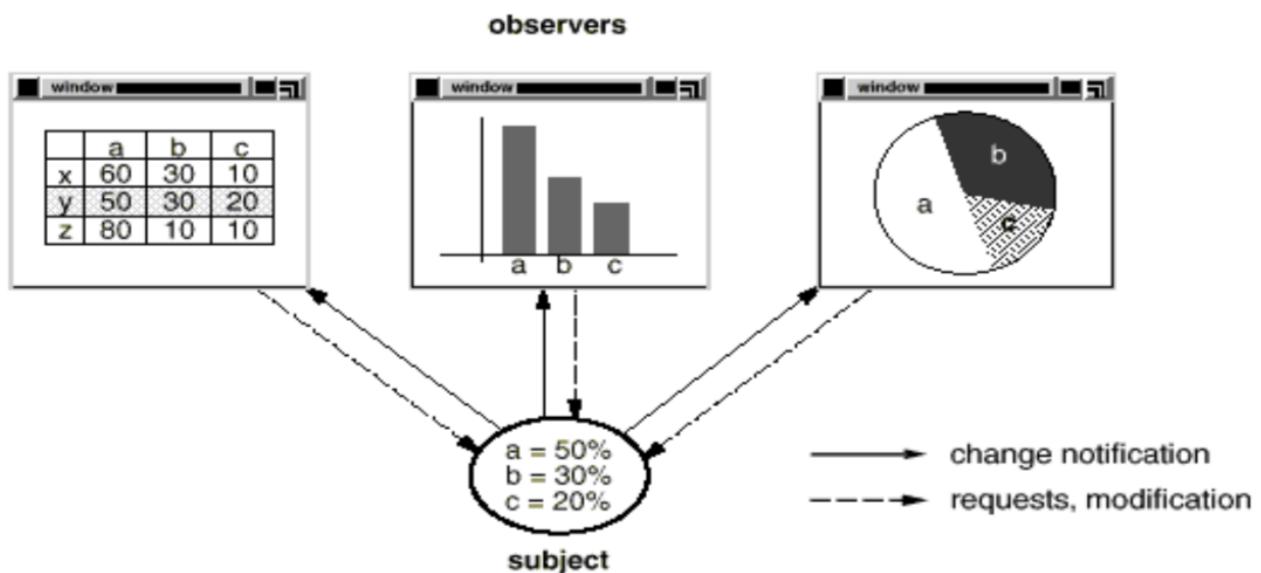


Figure 33. L'illustration classique du patron Observer (source [Meyer])

## 6.2. Définition

**Observateur** définit une relation entre objets de type un-à-plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

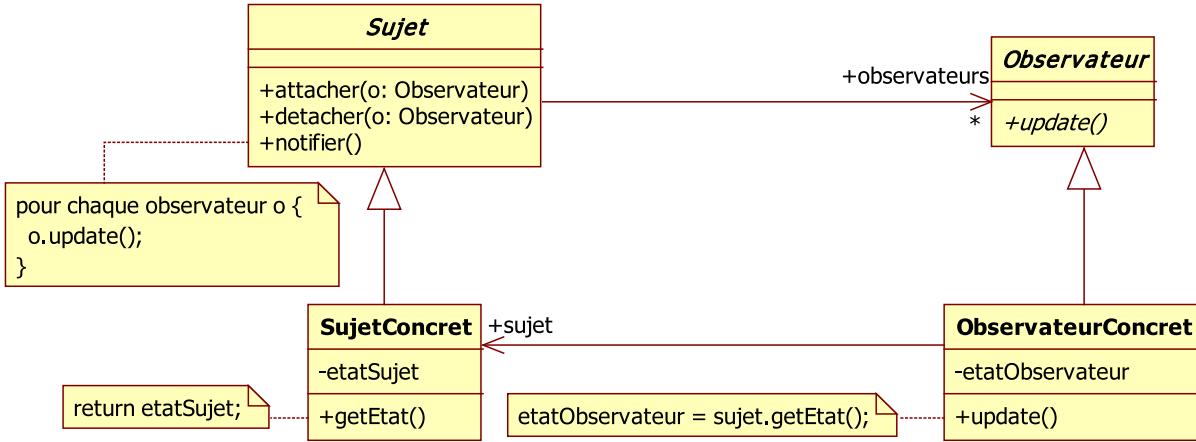


Figure 34. Modèle UML du patron Observer

## 6.3. Application

Le patron *Observer* est utilisable dans de nombreuses situations :

- Quand un concept a deux aspects, l'un dépendant de l'autre. Encapsuler ces aspects dans des objets séparés permet de les utiliser et les laisser évoluer de manière indépendante.
- Dès que le changement d'un objet entraîne le changement de plusieurs autres.
- Dès qu'un objet doit en notifier un certain nombre d'autres sans les connaître.

## 6.4. Observer en Java

Java fournit des classes *Observable/Observer* pour le patron *Observer*. La classe `java.util.Observable` est la classe de base pour les sujets. Ainsi, toute classe qui veut être observée étant cette classe dont voici les caractéristiques :

- fournit des méthodes pour ajouter/enlever des observateurs
- fournit des méthodes pour notifier les observateurs
- une sous-classe concrète doit seulement s'occuper de notifier à chaque méthode modifiant l'état des objets (*mutators*)
- utilise un vecteur stockant les références des observateurs

L'interface `java.util.Observer` correspond aux observateurs qui doivent implémenter cette interface.

### 6.4.1. La classe `java.util.Observable`

Voici la liste des méthodes de `java.util.Observable` :

```
public Observable()
public synchronized void addObserver(Observer o)
protected synchronized void setChanged()
public synchronized void deleteObserver(Observer o)
protected synchronized void clearChanged()
public synchronized boolean hasChanged()
public void notifyObservers(Object arg)
public void notifyObservers()
```

#### 6.4.2. L'interface `java.util.Observer`

*java.util.Observer*

```
/**
 * This method is called whenever the observed object is changed. An
 * application calls an observable object's notifyObservers method to have all
 * the object's observers notified of the change.
 *
 * Parameters:
 * o - the observable object
 * arg - an argument passed to the notifyObservers method
 */
public abstract void update(Observable o, Object arg)
```

### 6.5. Une implémentation du MVC : les `JTable` java

Examinons un exemple classique d'utilisation d'*Observer* : les **JTable** de **Java**.

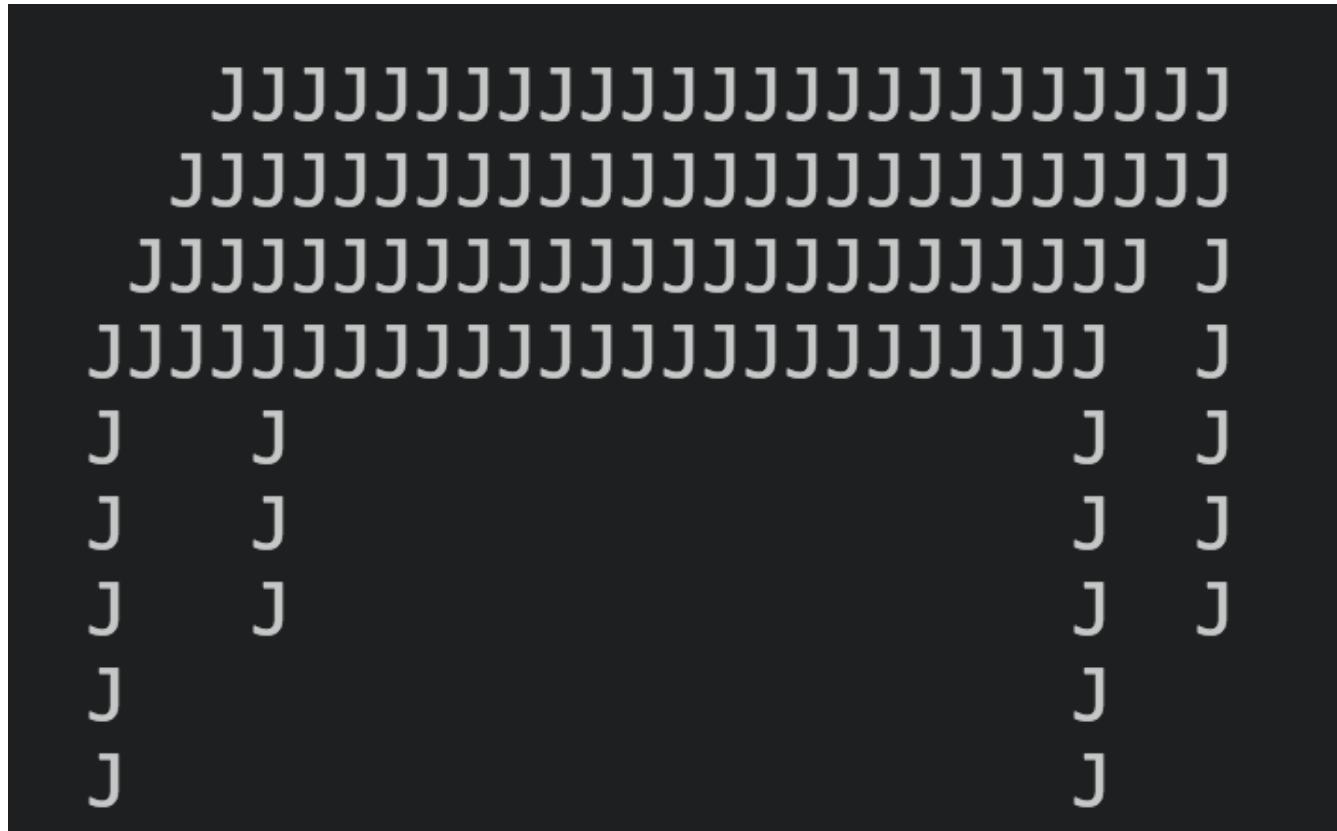


Figure 35. Ceci n'est pas une JTable (Crédit Dessin Justine Bruel, ma fille adorée)

### 6.5.1. Le principe

Avec une [JTable](#), on sépare la visualisation de la donnée elle-même.

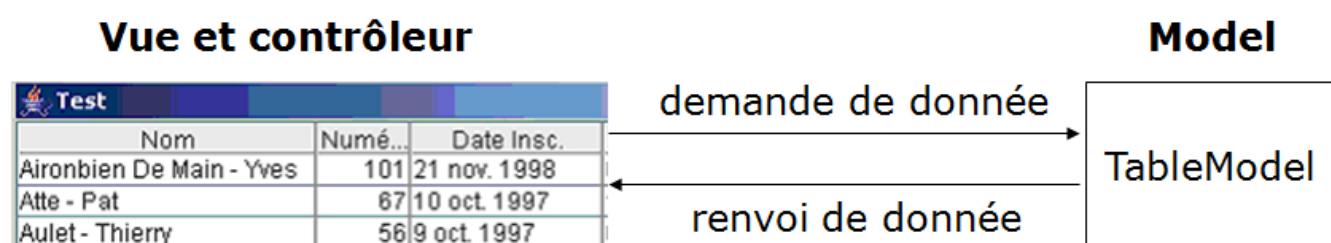


Figure 36. Principe de JTable

### 6.5.2. L'architecture

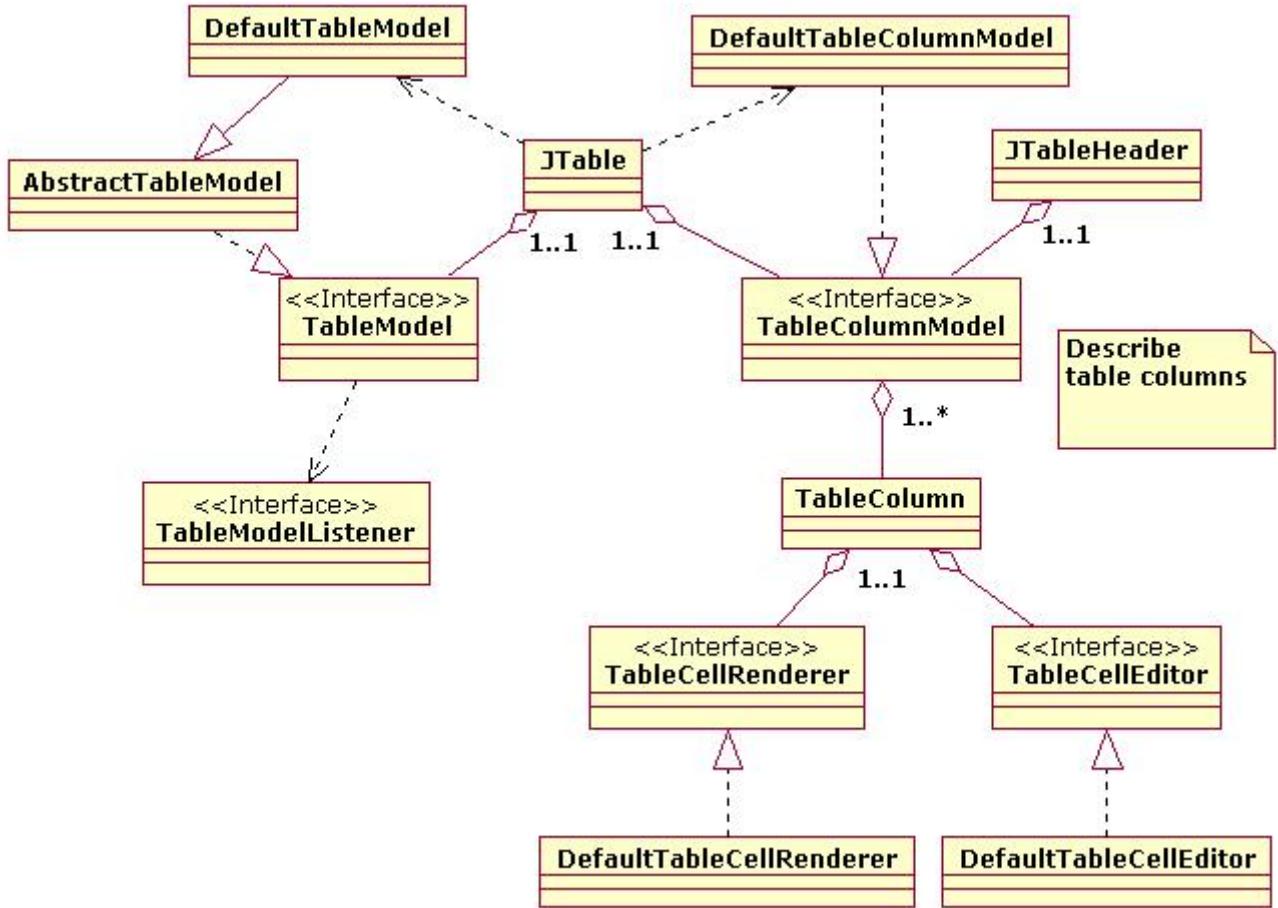


Figure 37. Architecture de `JTable`

## 7. Adaptateur



## 7.1. Le problème

On veut pouvoir :

- utiliser une classe existante, mais dont **l'interface ne coïncide pas** avec celle escomptée.
- créer une classe réutilisable qui collabore avec des classes sans relations avec elle et encore inconnues, c'est-à-dire avec des classes qui n'auront **pas nécessairement des interfaces compatibles**.
- vous avez besoin d'utiliser plusieurs sous-classes existantes, mais l'**adaptation de leur interface** par dérivation de chacune d'entre elles est impraticable. Un adaptateur objet peut adapter l'interface de sa classe parente.



Ce dernier cas ne concerne que le cas "adaptateur d'objet"

## 7.2. Exemple concret : le retour des canards

- L'existant :

```
public interface Canard {  
    public void cancaner();  
    public void voler();  
}  
  
public class Colvert implements Canard {  
    public void cancaner() {  
        System.out.println("Coincoin");  
    }  
    public void voler() {  
        System.out.println("Je vole");  
    }  
}
```

- Le "presque canard" :

```

public interface Dindon {
    public void glouglouter();
    public void voler();
}

public class DindonSauvage implements Dindon {
    public void glouglouter() {
        System.out.println("Glouglou");
    }
    public void voler() {
        System.out.println("Je ne vole pas loin");
    }
}

```

Vous êtes à court d'objets **Canard** et vous aimeriez utiliser des objets **Dindon** à la place!

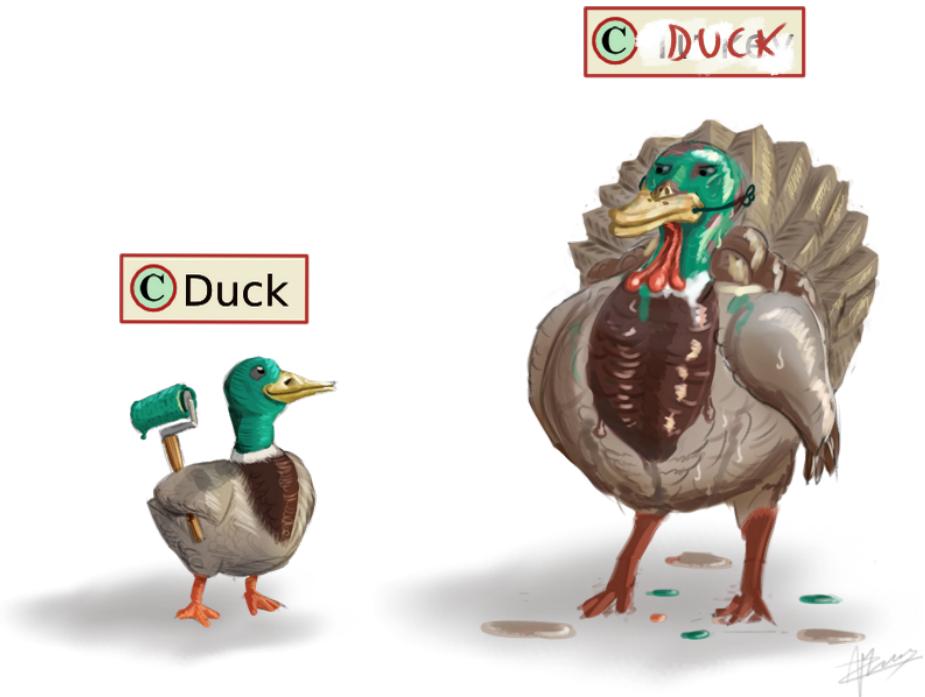


Figure 38. Un "presqueCanard" (Crédit Dessin C. Aribaud, 2A 2016-2017)

```

public class AdaptateurDindon implements Canard {
    Dindon dindon;

    ...

    public void cancaner() {
        dindon.gloogloouter();
    }

    public void voler() {
        // Adaptation du vol
        for(int i=0; i < 5; i++) {
            dindon.voler();
        }
    }
}

```

## 7.3. Le patron Adaptateur

**Adaptateur (Adapter)** permet de convertir l'interface d'une classe en une autre conformément à l'attente du client. L'Adaptateur permet à des classes de collaborer, alors qu'elles n'auraient pas pu le faire du fait d'interfaces incompatibles.

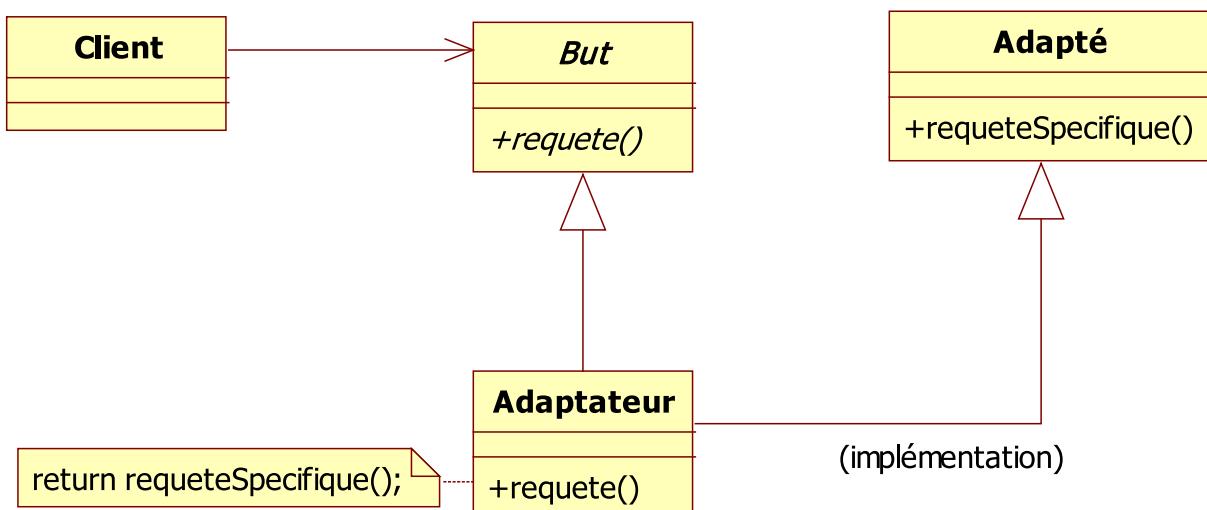


Figure 39. Modèle UML du patron Adaptateur

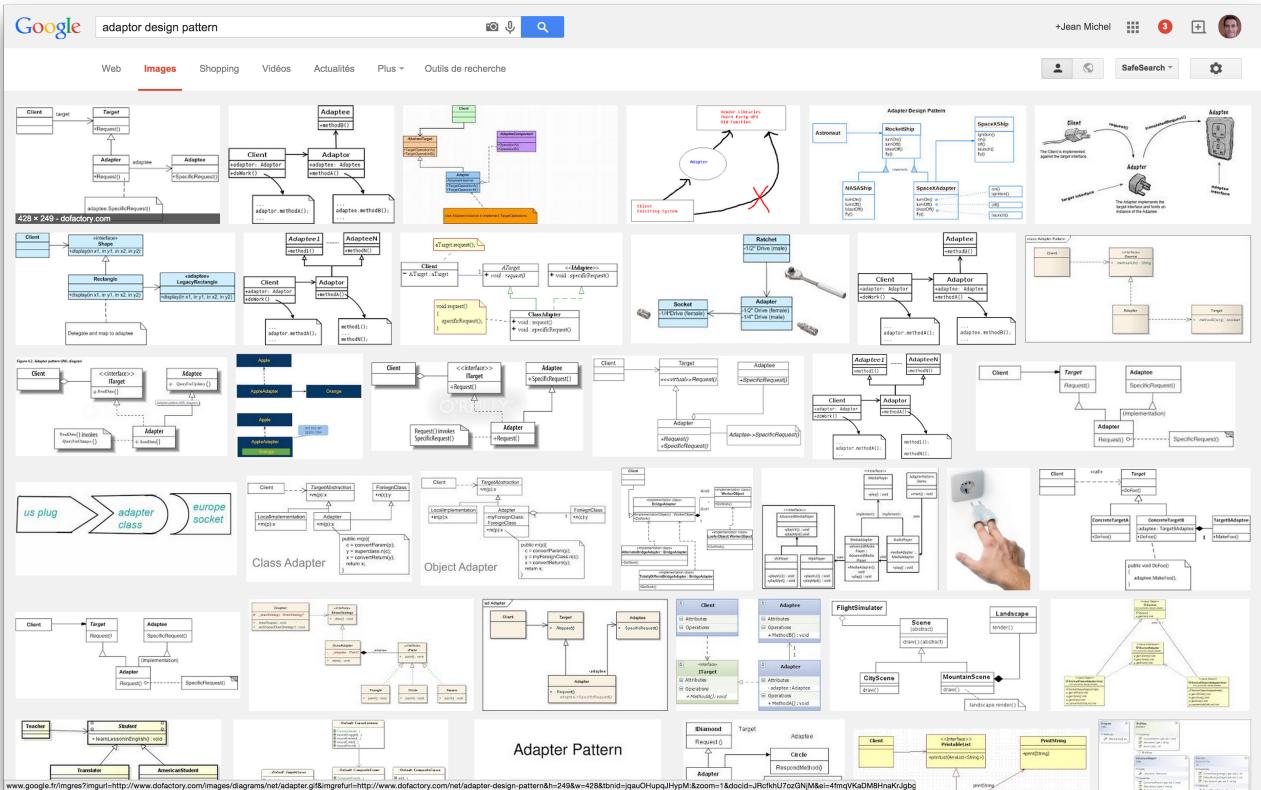


Figure 40. Adaptateur sur Google

## 8. Le patron Visiteur



### 8.1. Le problème

Quelques situations à problème :

- Une **structure** d'objets contient beaucoup de classes différentes d'interfaces distinctes, et vous désirez réaliser des opérations sur ces objets qui dépendent de leurs classes concrètes.
- Il s'agit d'effectuer plusieurs opérations distinctes et sans relation entre elles, sur les objets d'une structure, et ceci en **évitant de polluer leurs classes** avec ces opérations.

- Les classes qui définissent la structure objet changent rarement, mais on doit souvent **définir de nouvelles opérations** sur cette structure.

## 8.2. Illustration

Adapté d'un exemple tiré de [http://www.tutorialspoint.com/design\\_pattern/visitor\\_pattern.htm](http://www.tutorialspoint.com/design_pattern/visitor_pattern.htm)

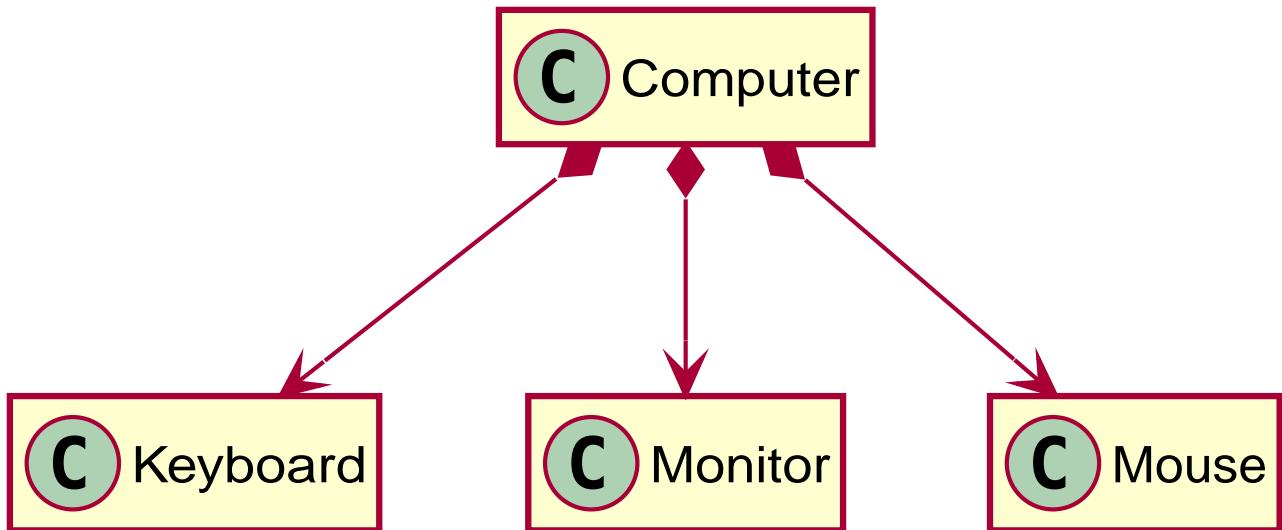


Figure 41. Une structure classique (sans patron)

### 8.2.1. Step 1

Définir une interface pour représenter les éléments de la structure.

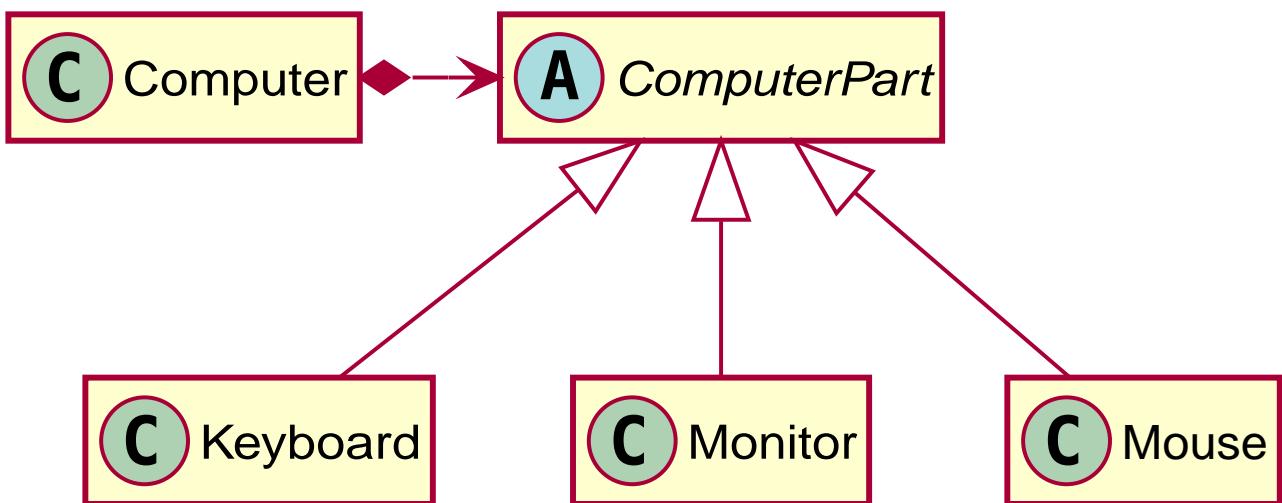


Figure 42. Implémentation d'une hiérarchie

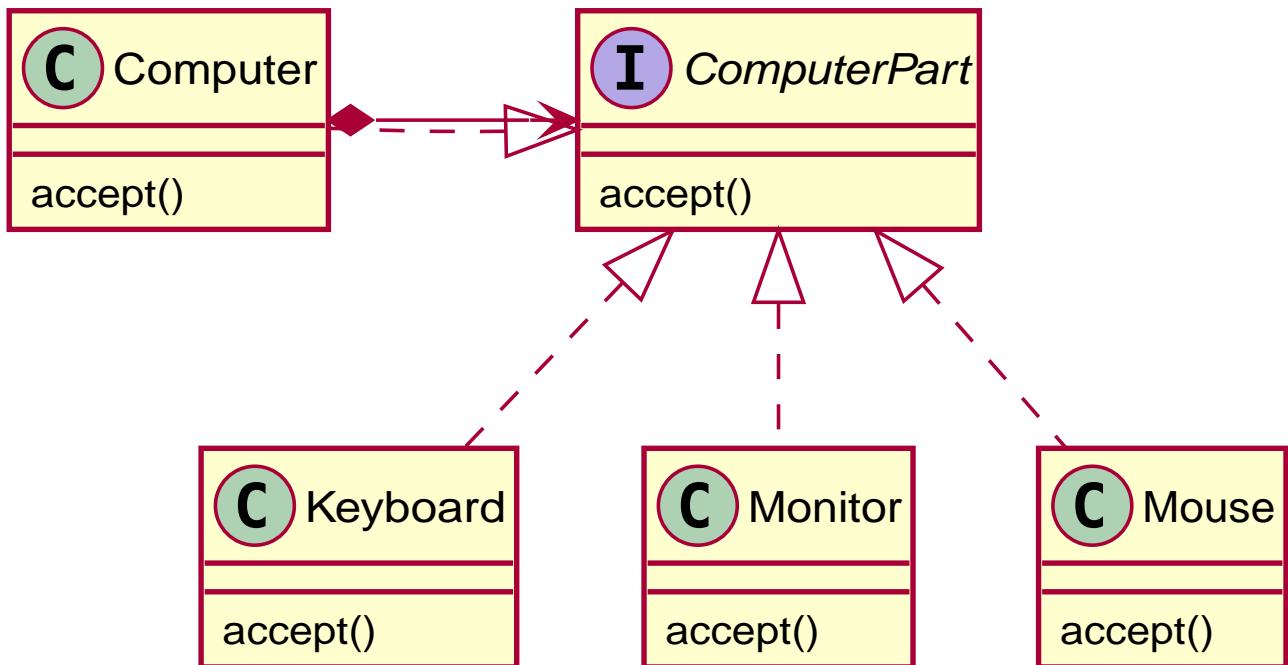


Figure 43. Utilisation d'une interface

*ComputerPart.java (extrait)*

```

public interface ComputerPart {
    ...
}
  
```

### 8.2.2. Step 2

Anticiper l'utilisation du visiteur.

*ComputerPartVisitor.java (extrait)*

```

public interface ComputerPartVisitor {
    ...
}
  
```

*ComputerPart.java*

```

public interface ComputerPart {
    public void accept(ComputerPartVisitor computerPartVisitor);
}
  
```

### 8.2.3. Step 3

Créer les classes concrètes qui implémentent l'interface.

### *Keyboard.java*

```
public class Keyboard implements ComputerPart {  
  
    @Override  
    public void accept(ComputerPartVisitor computerPartVisitor) {  
        computerPartVisitor.visit(this);  
    }  
}
```

...

### *Computer.java*

```
public class Computer implements ComputerPart {  
  
    ComputerPart[] parts;  
  
    public Computer(){  
        parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};  
    }  
  
    @Override  
    public void accept(ComputerPartVisitor computerPartVisitor) {  
        for (int i = 0; i < parts.length; i++) {  
            parts[i].accept(computerPartVisitor);  
        }  
        computerPartVisitor.visit(this);  
    }  
}
```

## 8.2.4. Step 4

Définir l'interface pour représenter le visiteur.

### *ComputerPartVisitor.java*

```
public interface ComputerPartVisitor {  
    public void visit(Computer computer);  
    public void visit(Mouse mouse);  
    public void visit(Keyboard keyboard);  
    public void visit(Monitor monitor);  
}
```

## 8.2.5. Step 5

Créer des visiteurs concrets.

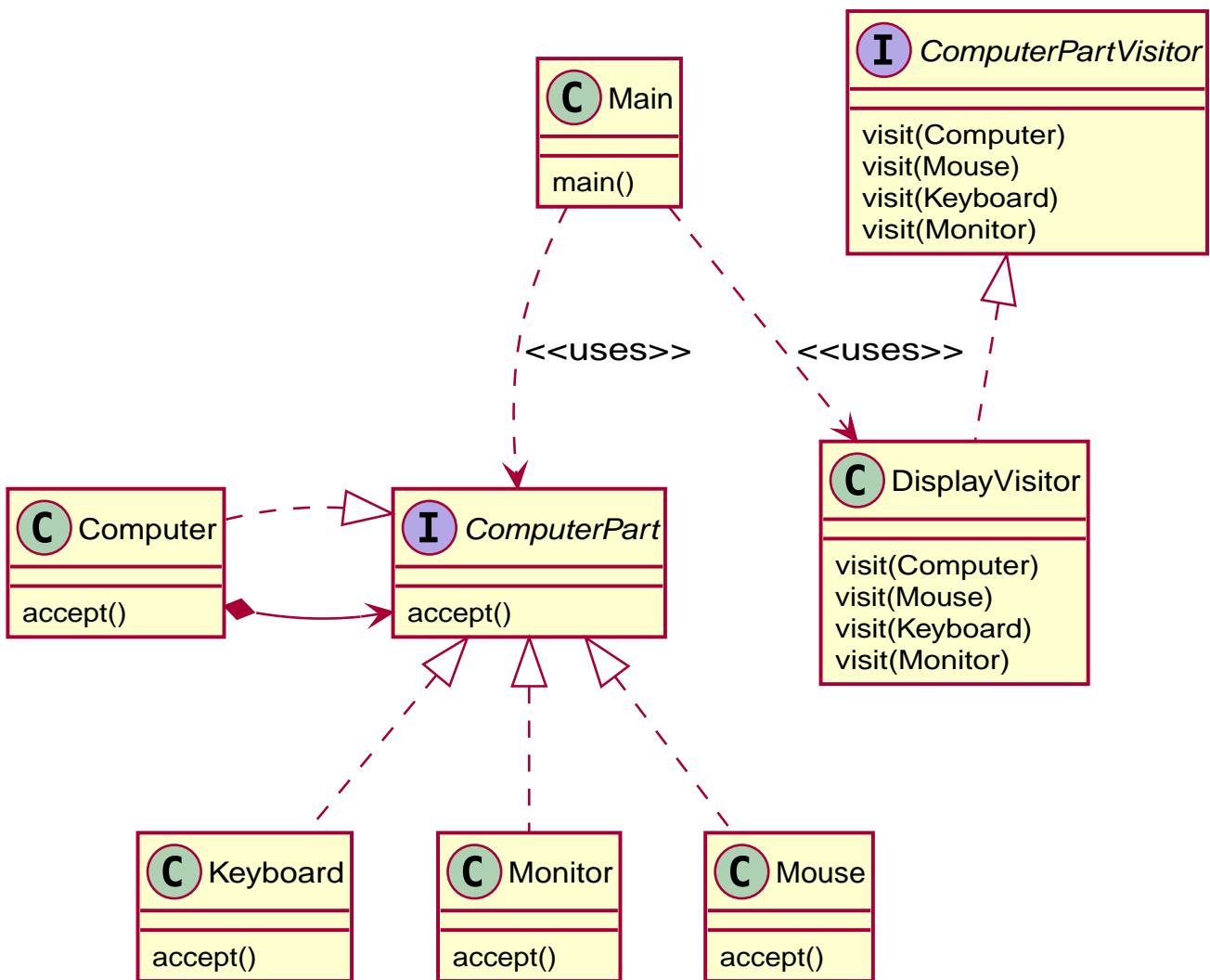


Figure 44. Visiteur : l'interface

### *DisplayVisitor.java*

```
public class DisplayVisitor implements ComputerPartVisitor {

    @Override
    public void visit(Computer computer) {
        System.out.println("Displaying Computer.");
    }

    @Override
    public void visit(Mouse mouse) {
        System.out.println("Displaying Mouse.");
    }

    @Override
    public void visit(Keyboard keyboard) {
        System.out.println("Displaying Keyboard.");
    }

    @Override
    public void visit(Monitor monitor) {
        System.out.println("Displaying Monitor.");
    }
}
```

### **8.2.6. Step 6**

Utiliser le visiteur *DisplayVisitor*.

### *VisitorPatternDemo.java*

```
public class VisitorPatternDemo {
    public static void main(String[] args) {

        ComputerPart computer = new Computer();
        computer.accept(new DisplayVisitor());
    }
}
```

### **8.2.7. Step 7 (final)**

Verify the output.

```
Displaying Mouse.
Displaying Keyboard.
Displaying Monitor.
Displaying Computer.
```

## 8.3. Le patron Visiteur

Visiteur (*Visitor*) permet la représentation d'une opération applicable aux éléments d'une structure d'objet.

Il définit une nouvelle opération, **sans qu'il soit nécessaire de modifier la classe** des éléments sur lesquels elle agit.

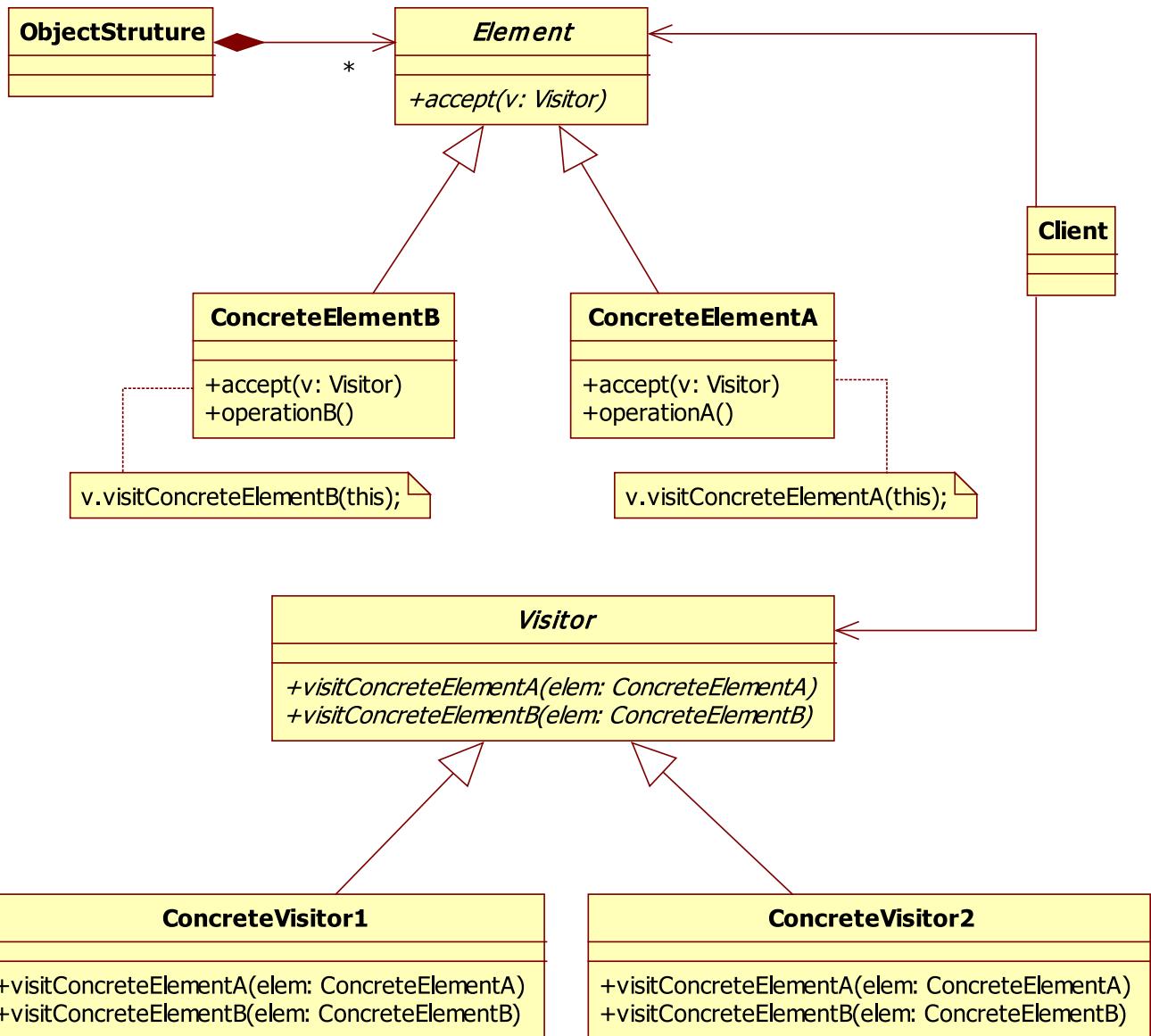


Figure 45. Patron Visiteur : structure

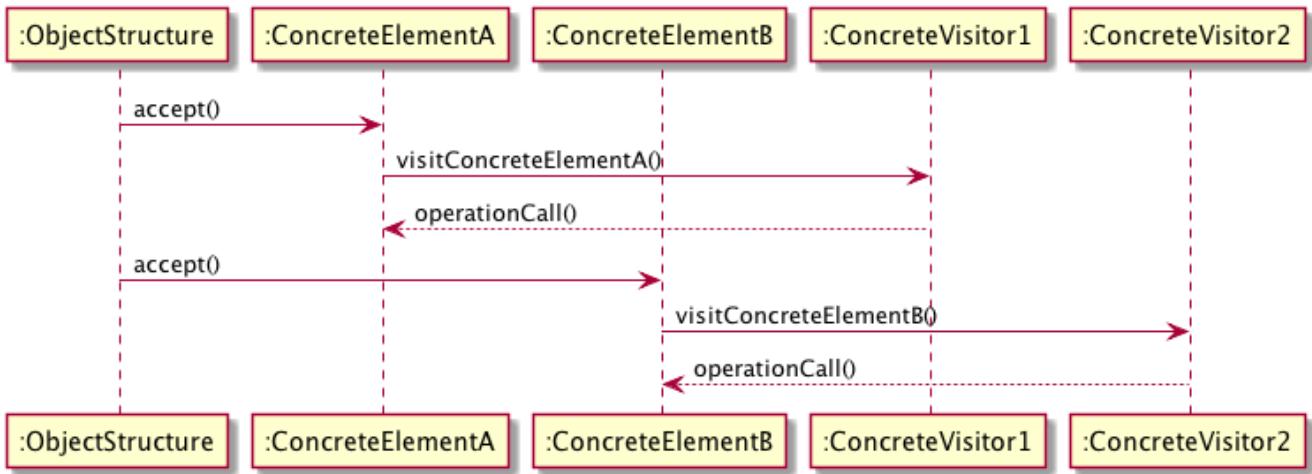


Figure 46. Patron Visiteur : comportement

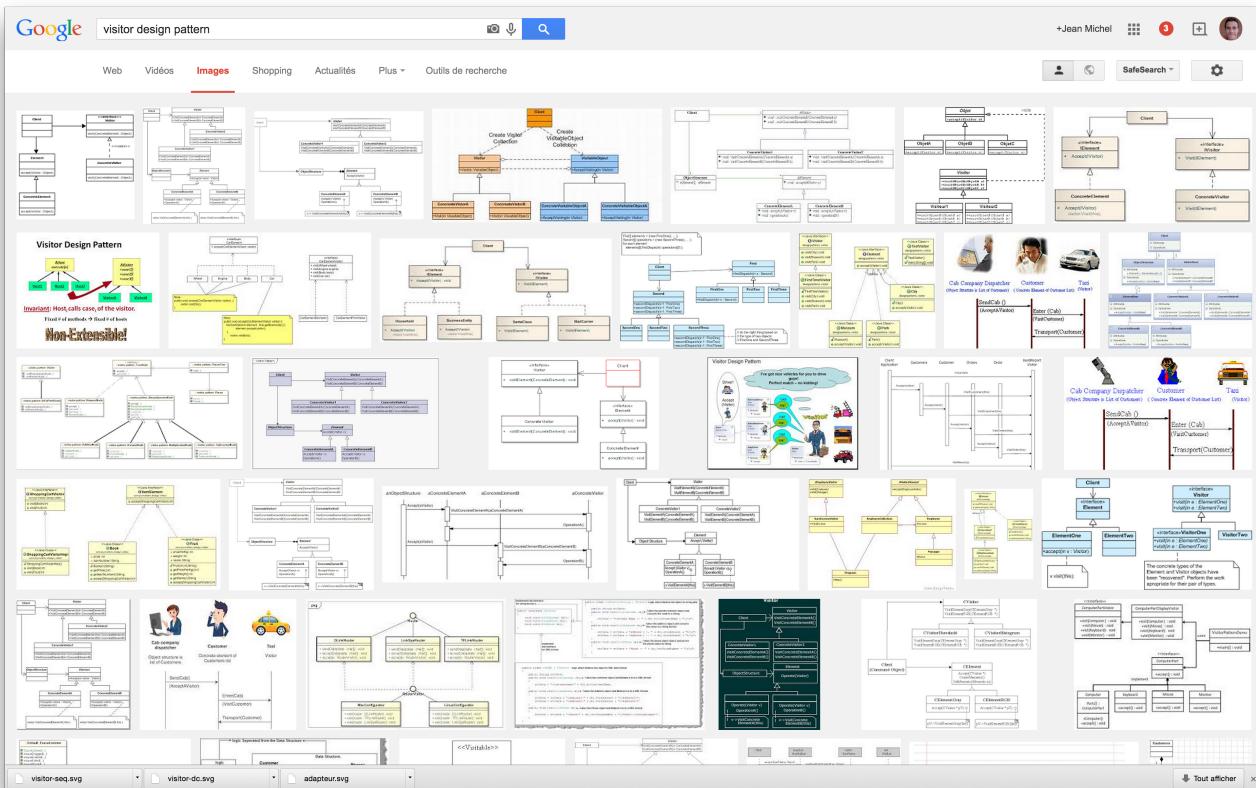


Figure 47. Visiteur sur Google

## 8.4. Avantages/Inconvénients

Avantages :

- Permet d'ajouter des opérations à la structure d'un Composite **sans modifier la structure elle-même**.
- L'ajout de nouvelles opérations** est relativement facile.
- Le code des opérations exécutées par le Visiteur est **centralisé**.

Inconvénients :

- L'encapsulation des classes du Composite est brisé.e.
- Comme une fonction de navigation est impliquée, les modifications de la structure du Composite sont plus difficiles.

## 8.5. Exemples d'utilisation

- calcul sur un ensemble structuré d'éléments
- génération de rapports ou de code
- ...

## 8.6. Exemple concret d'utilisation en Java



Exemple tiré de [ce site](#).

*ItemElement.java*

```
public interface ItemElement {  
    public int accept(ShoppingCartVisitor visitor);  
}
```

*Book.java*

```
public class Book implements ItemElement {  
  
    private int price;  
    private String isbnNumber;  
  
    public Book(int cost, String isbn){  
        this.price=cost;  
        this.isbnNumber=isbn;  
    }  
  
    public int getPrice() {  
        return price;  
    }  
  
    public String getIsbnNumber() {  
        return isbnNumber;  
    }  
  
    @Override  
    public int accept(ShoppingCartVisitor visitor) {  
        return visitor.visit(this);  
    }  
}
```

*Fruit.java*

```
public class Fruit implements ItemElement {  
  
    private int pricePerKg;  
    private int weight;  
    private String name;  
  
    public Fruit(int priceKg, int wt, String nm){  
        this.pricePerKg=priceKg;  
        this.weight=wt;  
        this.name = nm;  
    }  
  
    public int getPricePerKg() {  
        return pricePerKg;  
    }  
  
    public int getWeight() {  
        return weight;  
    }  
  
    public String getName(){  
        return this.name;  
    }  
  
    @Override  
    public int accept(ShoppingCartVisitor visitor) {  
        return visitor.visit(this);  
    }  
}
```

*ShoppingCartVisitor.java*

```
public interface ShoppingCartVisitor {  
  
    int visit(Book book);  
    int visit(Fruit fruit);  
}
```

### *ShoppingCartVisitorImpl.java*

```
public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {  
  
    @Override  
    public int visit(Book book) {  
        int cost=0;  
        //apply 5$ discount if book price is greater than 50  
        if(book.getPrice() > 50){  
            cost = book.getPrice()-5;  
        } else cost = book.getPrice();  
        System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost =" +cost);  
        return cost;  
    }  
  
    @Override  
    public int visit(Fruit fruit) {  
        int cost = fruit.getPricePerKg()*fruit.getWeight();  
        System.out.println(fruit.getName() + " cost = " +cost);  
        return cost;  
    }  
}
```

### *ShoppingCartClient.java*

```
public class ShoppingCartClient {  
  
    public static void main(String[] args) {  
        ItemElement[] items = new ItemElement[]{new Book(20, "1234"), new Book(100,  
        "5678"),  
        new Fruit(10, 2, "Banana"), new Fruit(5, 5, "Apple")};  
  
        int total = calculatePrice(items);  
        System.out.println("Total Cost = " +total);  
    }  
  
    private static int calculatePrice(ItemElement[] items) {  
        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();  
        int sum=0;  
        for(ItemElement item : items){  
            sum = sum + item.accept(visitor);  
        }  
        return sum;  
    }  
}
```

```

Book ISBN::1234 cost =20
Book ISBN::5678 cost =95
Banana cost = 20
Apple cost = 25
Total Cost = 160

```

## 9. Proxy

### 9.1. Le problème

On a besoin de références à un objet, qui soient plus **créatives** et plus **sophistiquées** qu'un simple pointeur.

### 9.2. Le patron *Proxy*

**Procuration (Proxy)** fournit à un tiers un mandataire ou un remplaçant, pour contrôler l'accès à cet objet.

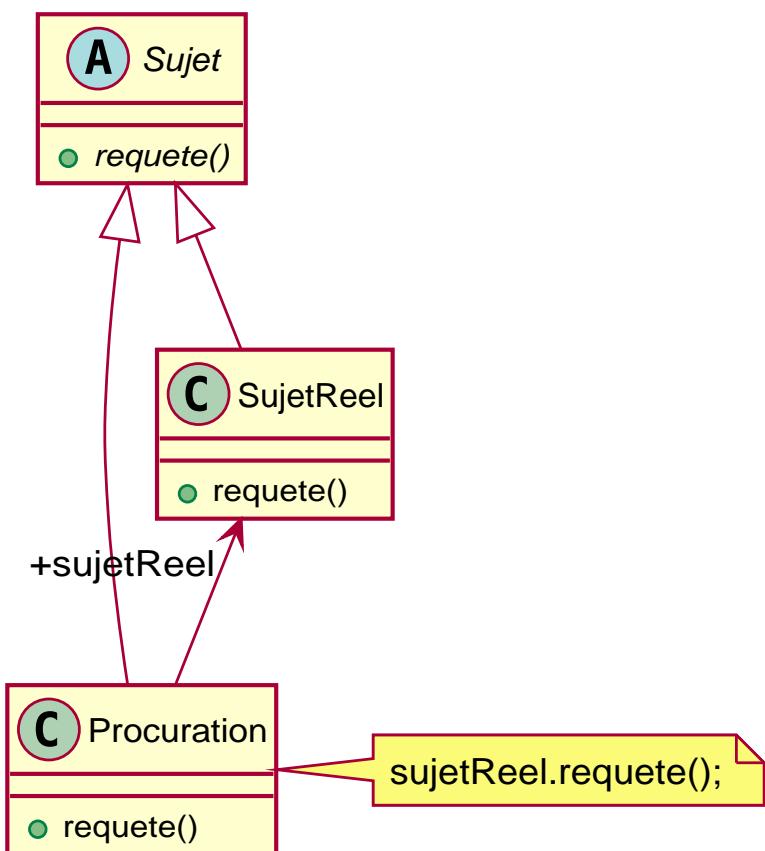


Figure 48. Modèle UML du patron *Proxy*

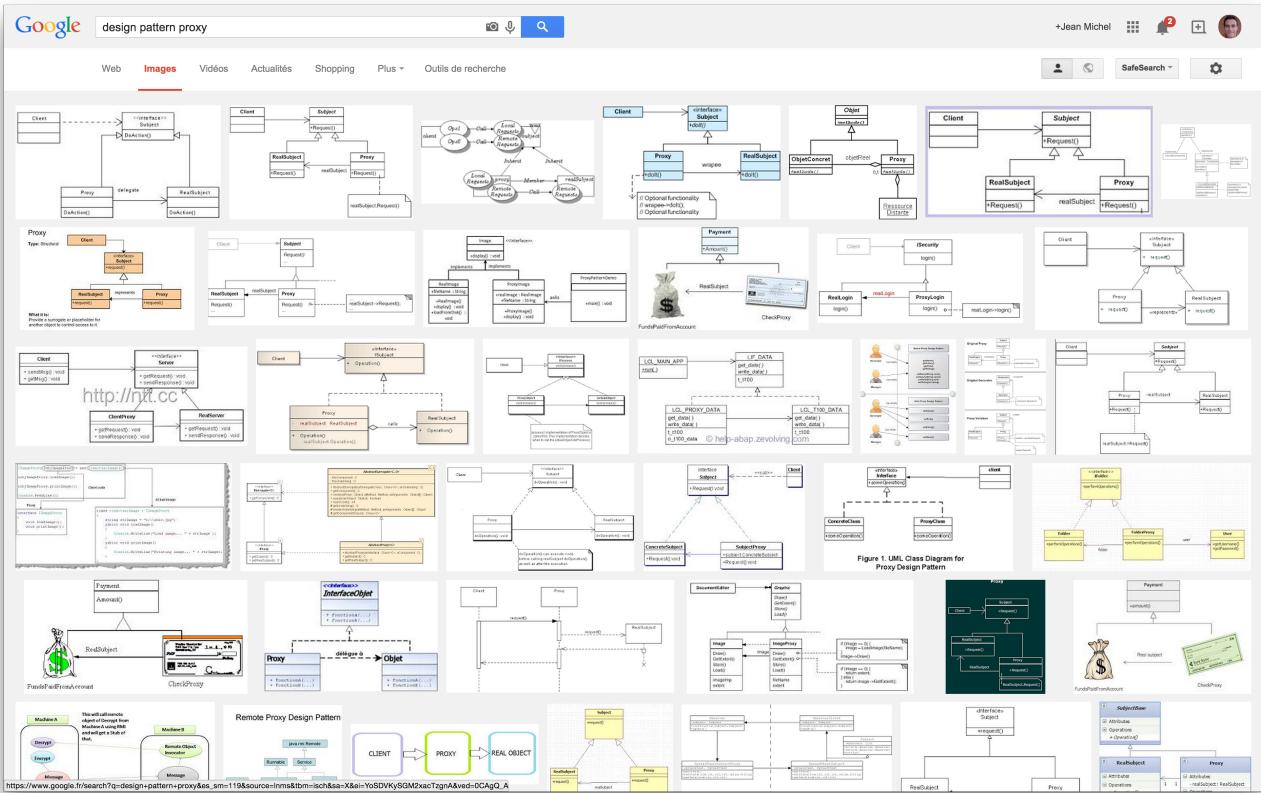


Figure 49. Proxy sur Google

## 9.3. Utilisations

- Une *procuration à distance* fournit un représentant local d'un objet situé dans un espace adresse différent.
- Une *procuration virtuelle* crée des objets lourds à la demande.
- Une *procuration de protection* contrôle l'accès à l'objet original. Les procurations de protection sont utiles quand les objets doivent satisfaire différents droits d'accès.
- Une *référence intelligente* est le remplaçant d'un pointeur brut, qui réalise des opérations supplémentaires, lors de l'accès à l'objet. Quelques utilisations typiques sont :
  - décompte du nombre des références faites à un objet réel, de sorte que celui-ci puisse être libéré automatiquement, dès qu'il n'y a plus de références ;
  - charger en mémoire un objet persistant quand il est référencé pour la première fois ;
  - vérifier, avant d'y accéder, que l'objet réel est verrouillé, pour être sûr qu'aucun autre objet ne pourra le changer.

## 9.4. Exemple concret : RMI

Remote Method Invocation est une méthode d'accès à un service à distance.

```

import java.rmi.*;
public interface MonService extends Remote {
    public String direBonjour() throws RemoteException;
}

```

```

import java.rmi.*;
import java.rmi.server.*;

public class MonServiceImpl extends UnicastRemoteObject implements MonService {
    public String direBonjour() {
        return "Le serveur dit 'Bonjour'";
    }
    public MonServiceImpl() throws RemoteException {}
    public static void main (String[] args) {
        try {
            MonService service = new MonServiceImpl();
            Naming.rebind("BonjourDistant", service);
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

```

MonService service =
    (MonService) Naming.lookup("rmi://127.0.0.1/BonjourDistant");
...
service.direBonjour();

```

## 10. Itérateur

### 10.1. Le problème

On veut pouvoir :

- pour accéder au contenu d'un objet d'un agrégat sans en révéler la représentation interne ;
- pour gérer simultanément plusieurs parcours dans des agrégats d'objets ;
- pour offrir une interface uniforme pour les parcours au travers de diverses structures agrégats (c'est-à-dire, pour permettre l'itération polymorphe).

### 10.2. Le patron Itérateur

**Itérateur (Iterator)** fournit un moyen d'accès séquentiel aux éléments d'un agrégat d'objets, sans mettre à découvert la représentation interne de celui-ci.

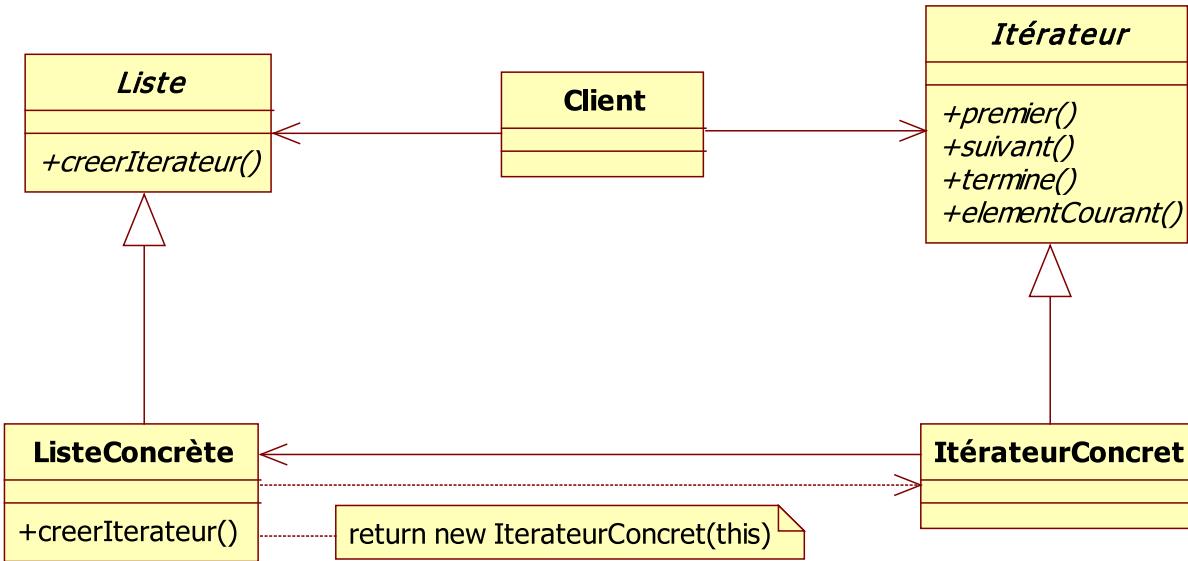


Figure 50. Modèle UML du patron Itérateur

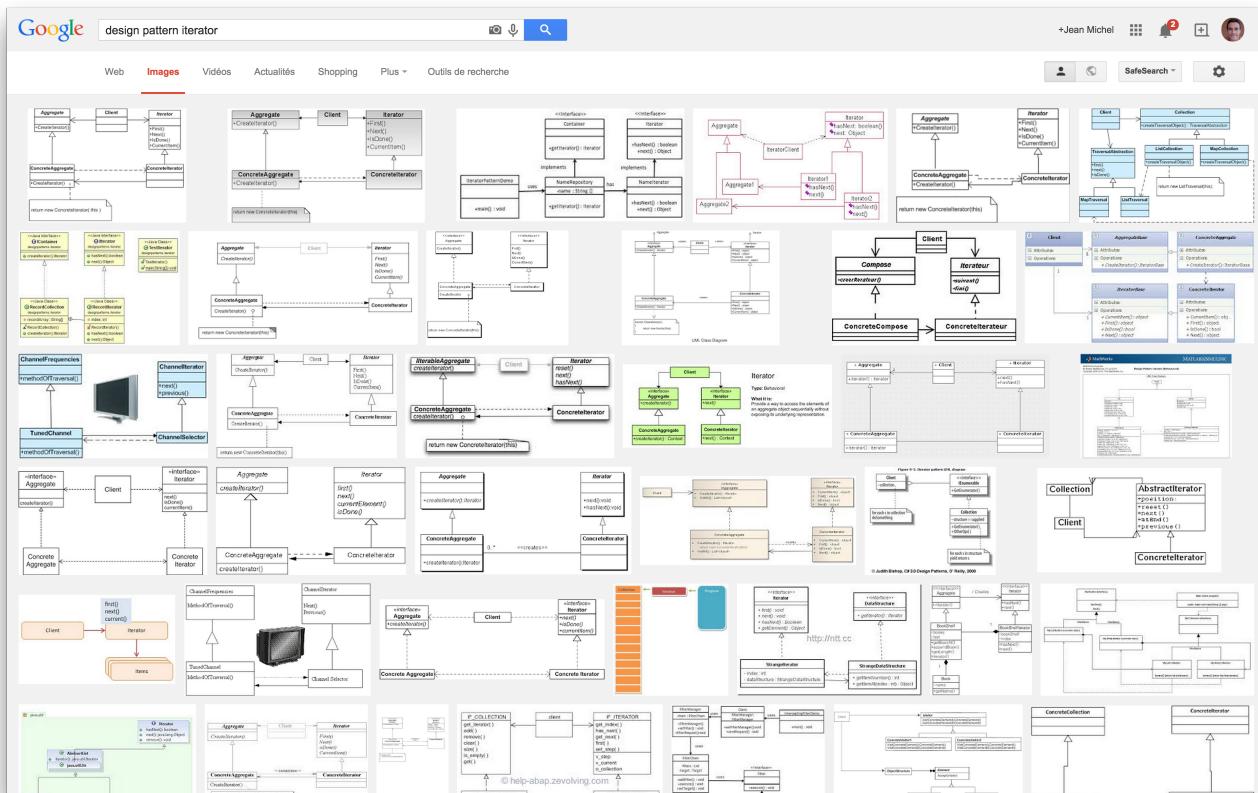


Figure 51. Iterateur sur Google

## 10.3. Exemple concret

Voici un exemple en Ruby :

```
# Saluer tout le monde
def say_hi
  if @names.nil?
    puts "..."
  elsif @names.respond_to?("each")
    # @names est une liste de noms : traitons-les uns par uns
    @names.each do |name|
      puts "Hello #{name}!"
    end
  else
    puts "Hello #{@names}!"
  end
end
```

## 11. Le patron Composite

### 11.1. Le problème

On veut pouvoir :

- représenter des hiérarchies de l'individu.
- que le client n'ait pas à se préoccuper de la différence entre "combinaisons d'objets" et "objets individuels". Les clients pourront traiter de façon uniforme tous les objets de la structure composite.

### 11.2. Le patron Composite

**Composite** permet de composer des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Permet au client de traiter d'une façon unique les objets et les combinaisons d'objets.

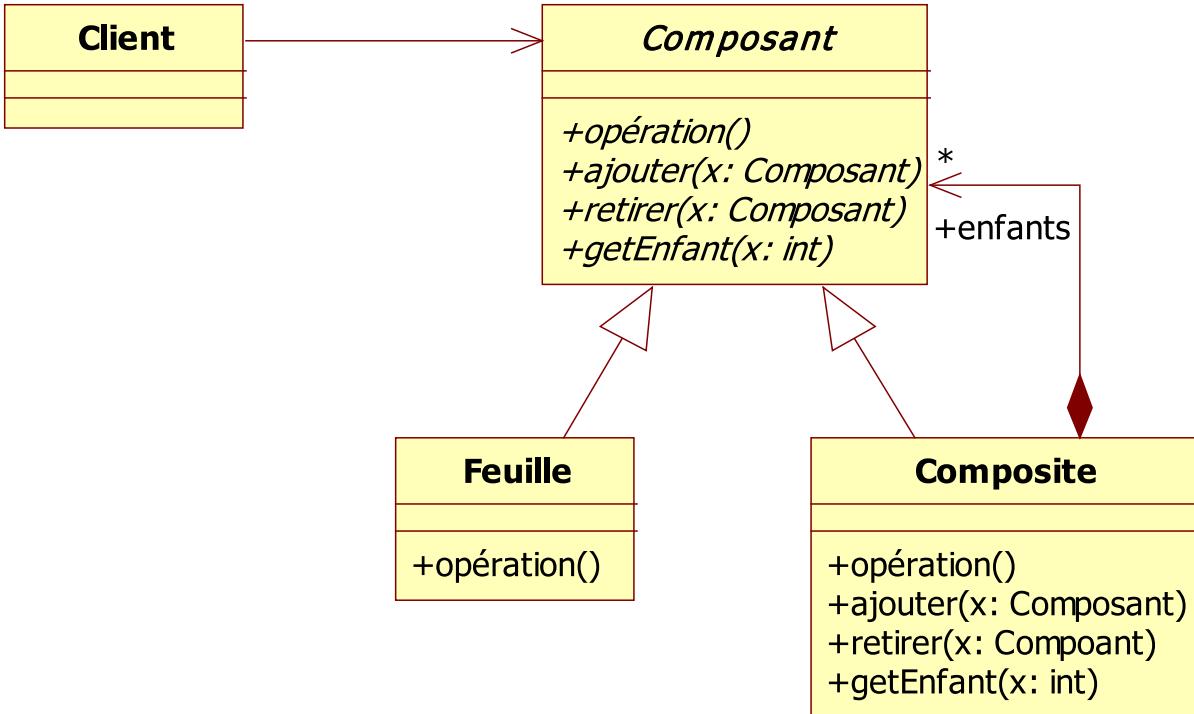


Figure 52. Modèle UML du patron Composite

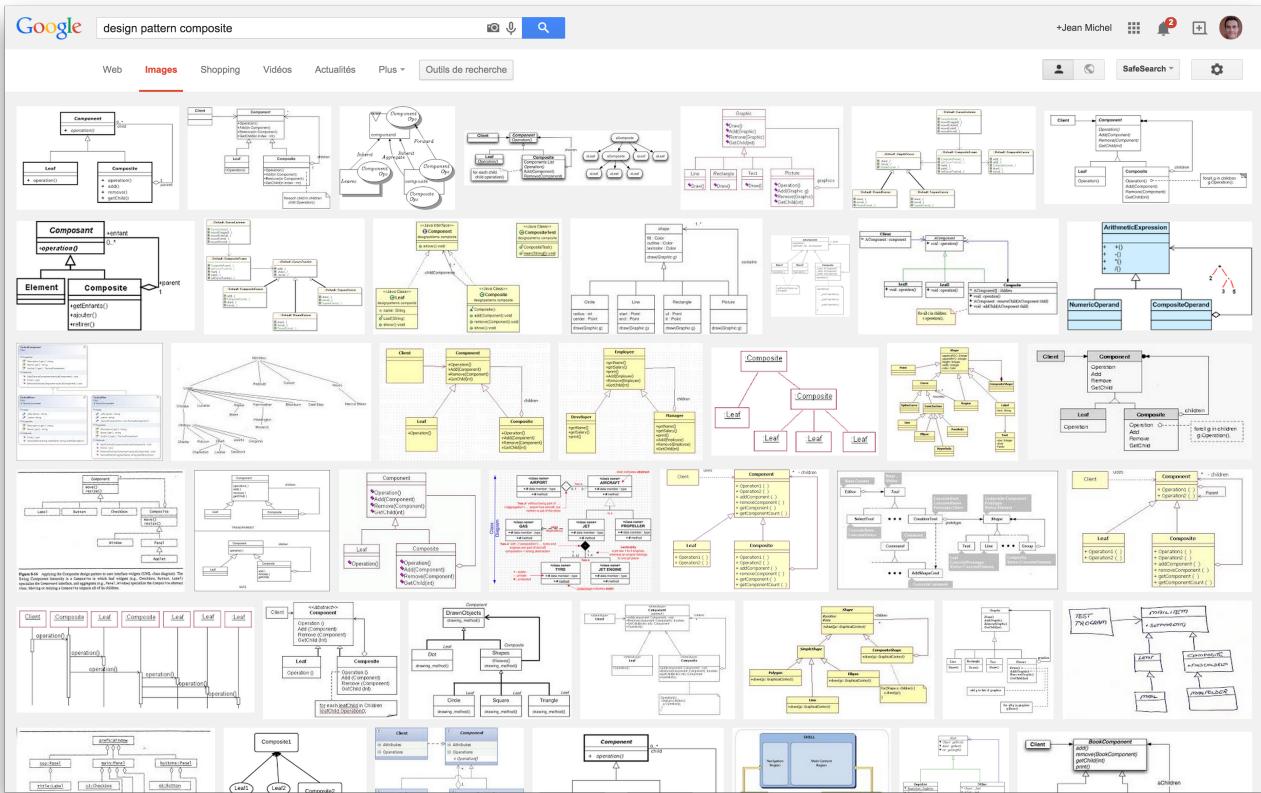


Figure 53. Composite sur Google

## 11.3. Exemple concret

```

import java.util.ArrayList;

interface Graphic {
    public void print();
}

class CompositeGraphic implements Graphic {

    private ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();

    public void print() {
        for (Graphic graphic : mChildGraphics) {
            graphic.print();
        }
    }

    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }

    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
}

```

## 11.4. un "Anti" exemple

Que pensez-vous de cette définition de Composite ?

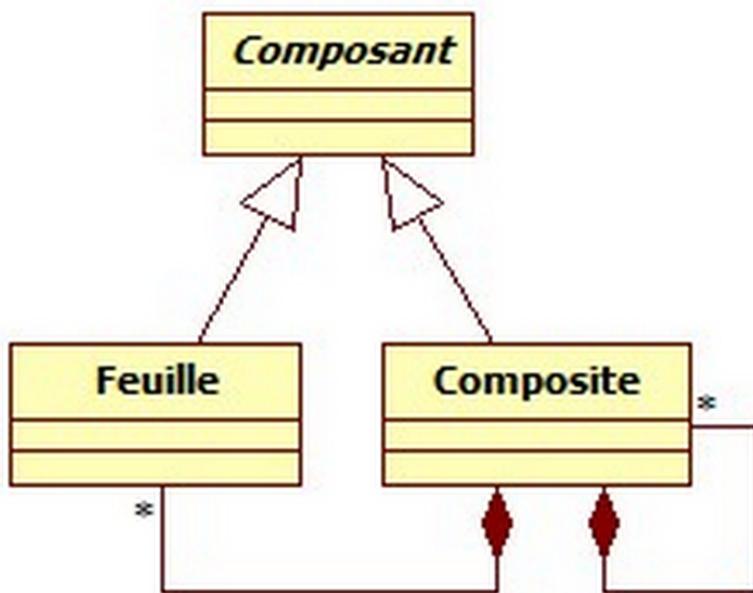


Figure 54. Patron abîmé composite



On appelle ces modèles des "Patrons abîmés" (*anti-patterns*).

# 12. Pour aller plus loin avec les patrons...



Figure 55. Crédit photo : <http://blogs.radiocanut.org/>

## 12.1. Partagez votre vocabulaire

- Dans les réunions de conception (pas nécessairement avec le client)
- Avec les autres développeurs
- Dans la documentation de votre architecture
- Dans les commentaires du code et les conventions de nommage
- Dans les groupes/blogs de développeurs
- (pas pendant les exams!)

## 12.2. Ne foncez pas tête baissée

Quelques conseils :

- Les patterns sont des **outils, non des règles.**  
⇒ Rien n'empêche de les modifier et de les adapter à votre problème.
- Ne visez l'extensibilité **que si la question se pose réellement** dans la pratique, pas si elle est uniquement hypothétique.
- Ne vous emballiez pas et **recherchez la simplicité.**  
⇒ Si vous trouvez une solution plus simple que l'emploi d'un pattern, n'hésitez pas !

- éliminez ce qui n'est pas vraiment nécessaire.

⇒ N'ayez pas peur de **supprimer un design pattern inutile** de votre conception.

## 12.3. Les autres types de patrons

Il n'y a pas que les 3 types de patrons que l'on a vu :

- De création
- Structurels
- Comportementaux

Il y a par exemple :

- Les patrons d'architecture
- Les patrons d'application
- Les patrons de domaine
- Les patrons de processus
- Les patrons d'organisation
- Les patrons de conception d'interfaces utilisateur

## 12.4. Les anti-patrons



- Des solutions souvent **appliquées à tort** à des problèmes récurrents
- Décrit comment partir d'un problème pour arriver à une **mauvaise solution**
- Vous dit **pourquoi** une mauvaise solution est attrayante

- Suggère d'autres patrons applicables pouvant fournir de meilleures solutions

## 12.5. Tous les patrons qu'on a pas vu

Il y en a beaucoup :

- Chaîne de responsabilité
- Commande
- Décorateur
- Façade
- Interprète
- Médiateur
- Memento
- Monteur
- Patron de méthode
- Poids-mouche
- Pont
- Prototype

## 12.6. Injection de dépendances

Nous n'avons pas le temps de traiter ce point, mais cf. par exemple <https://blog.angularindepth.com/why-do-we-have-dependency-injection-in-web-development-f8815e593b38>.

# Glossaire et définition



Ces définitions reprennent les définitions vues dans ce livre, en les organisant par grands types de patrons de conception : de création, comportementaux et structurels.

## Patrons de création

### Singleton

**Singleton** garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.

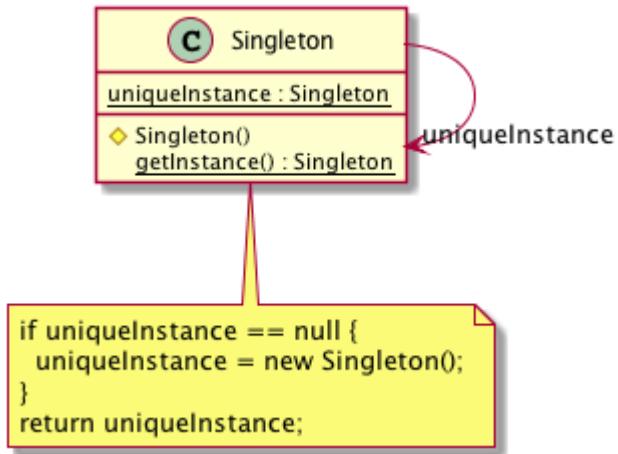


Figure 56. Modèle UML du patron Singleton

## Fabrique

**Fabrique** (simple) définit une interface pour la création d'un objet, mais en laissant à des sous-classes le choix des classes à instancier (voir aussi [Fabrique abstraite](#)).

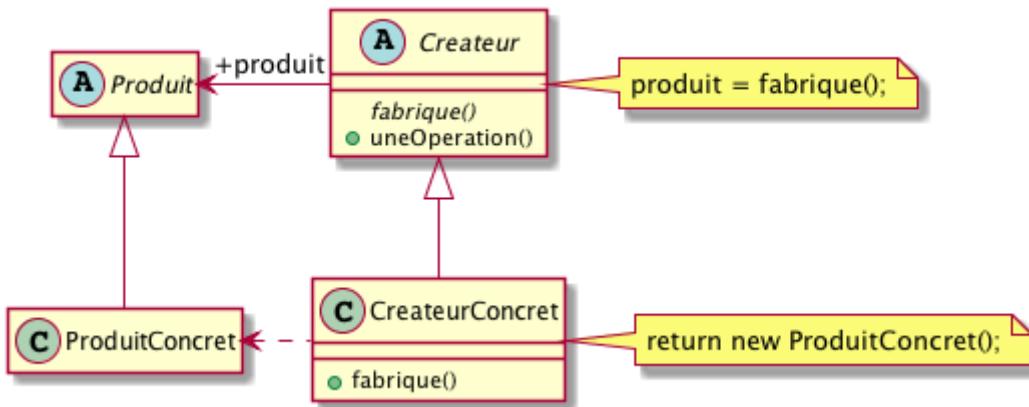


Figure 57. Modèle UML du patron Fabrique

## Fabrique abstraite

**Fabrique** (abstraite) fournit une interface pour la création de familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes (voir aussi [Fabrique](#)).

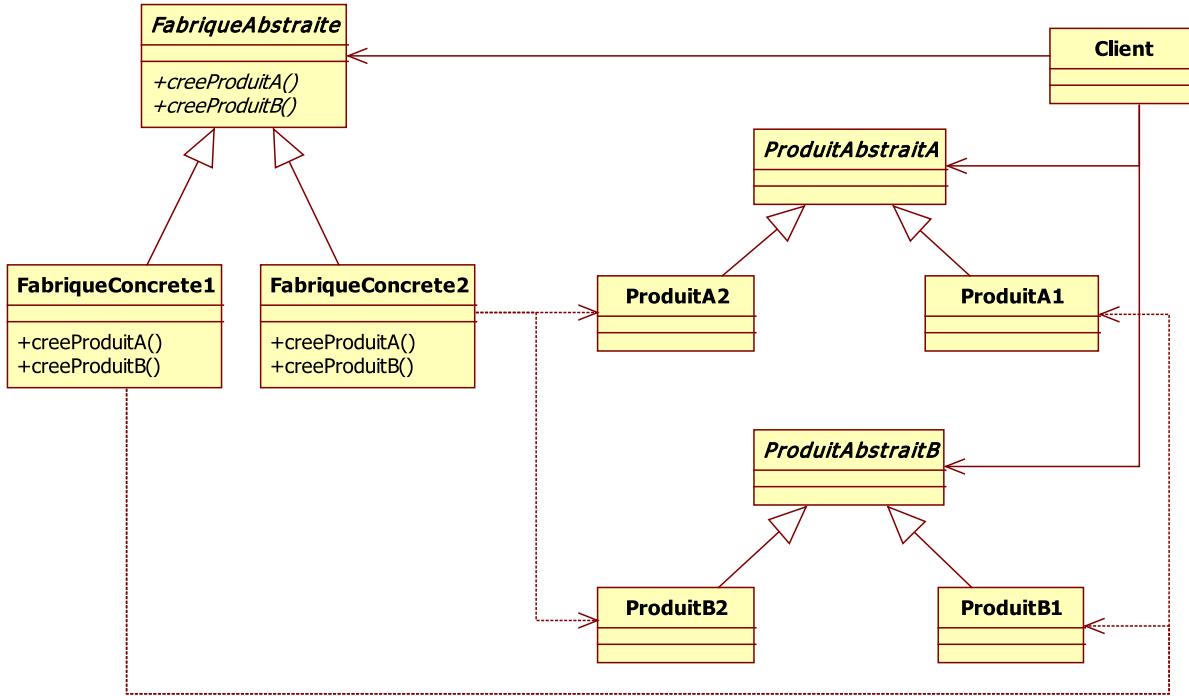


Figure 58. Modèle UML du patron Fabrique Abstraite

## Patrons comportementaux

### État (State<sup>uk</sup>)

Etat permet à un objet de modifier son comportement, quand son état interne change. Tout se passe comme si l'objet changeait de classe.

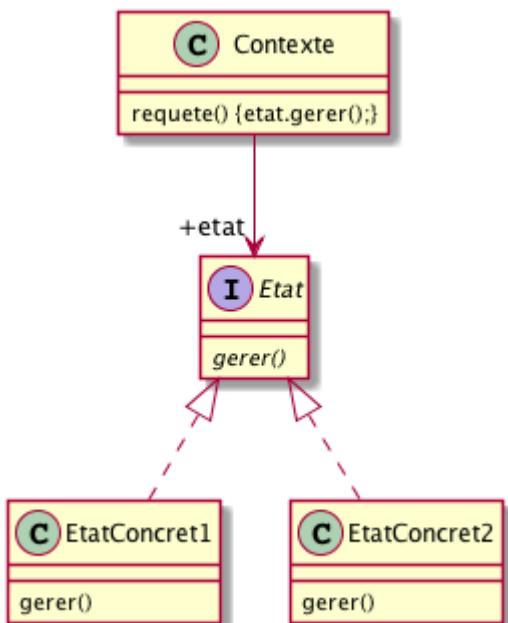


Figure 59. Modèle UML du patron Etat

### Itérateur (Iterator<sup>uk</sup>)

Itérateur (**Iterator**) fournit un moyen d'accès séquentiel aux éléments d'un agrégat d'objets, sans mettre à découvert la représentation interne de celui-ci.

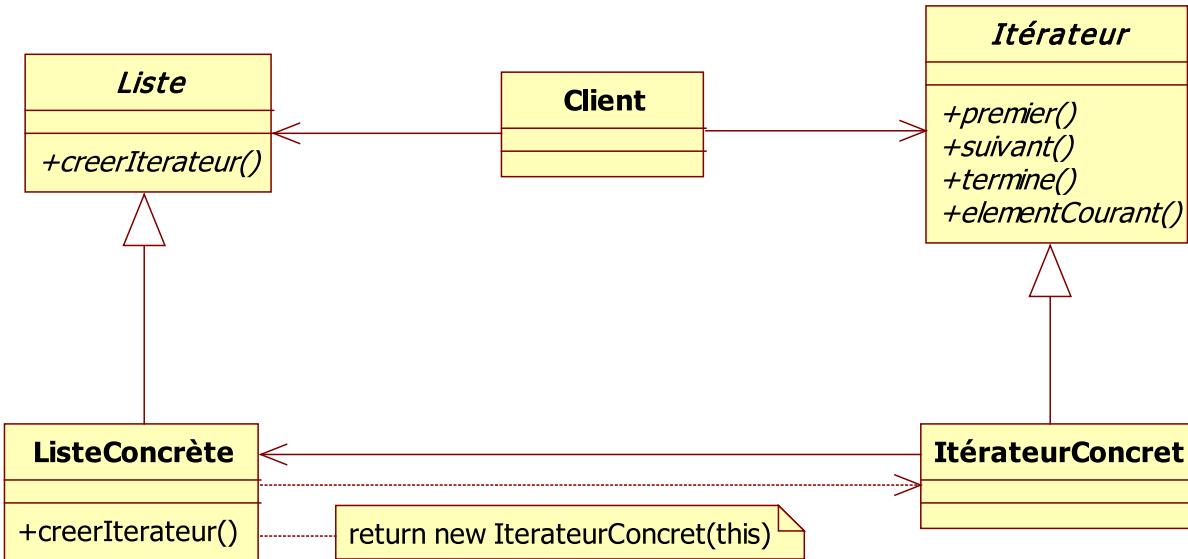


Figure 60. Modèle UML du patron Itérateur

### Observateur (Observer<sup>uk</sup>)

**Observateur** définit une relation entre objets de type un-à-plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

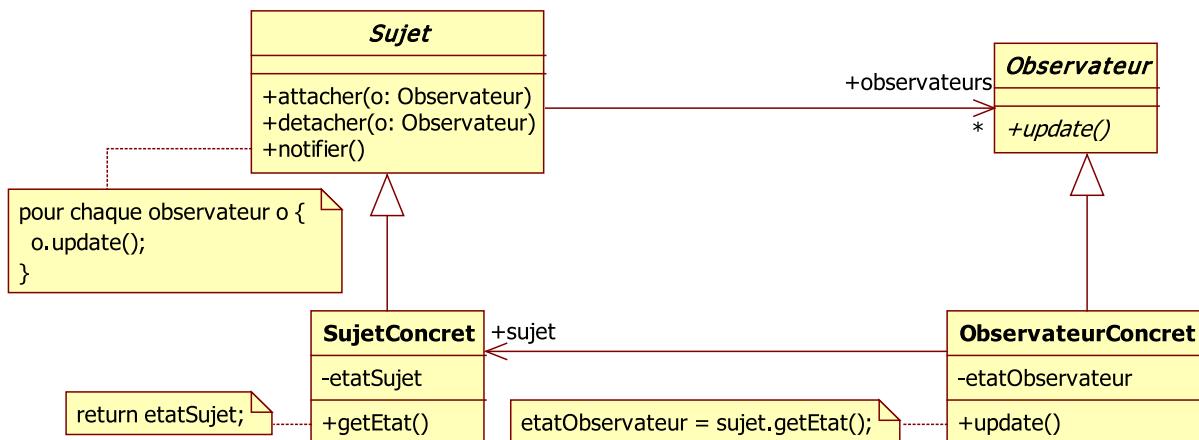


Figure 61. Modèle UML du patron Observer

### Stratégie (Strategy<sup>uk</sup>)

**Stratégie** définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Il permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

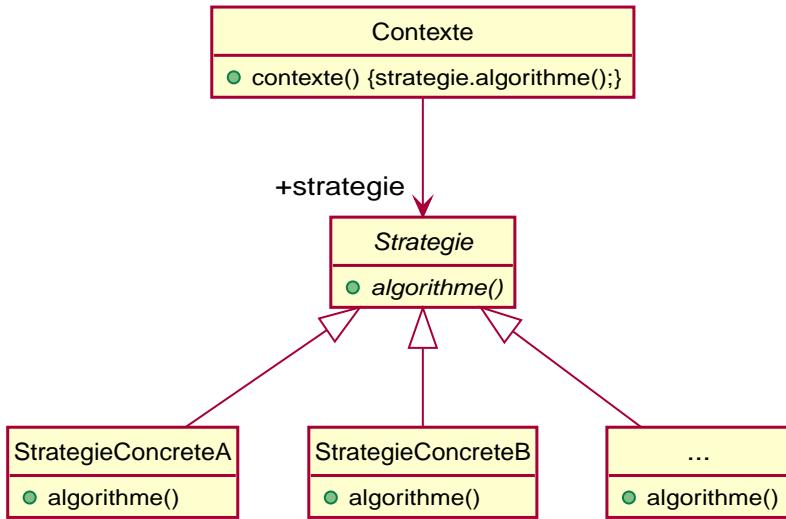


Figure 62. Modèle UML du patron Strategy

### Visiteur (Visitor<sup>uk</sup>)

**Visiteur (Visitor)** permet la représentation d'une opération applicable aux éléments d'une structure d'objet. Il définit une nouvelle opération, sans qu'il soit nécessaire de modifier la classe des éléments sur lesquels elle agit.

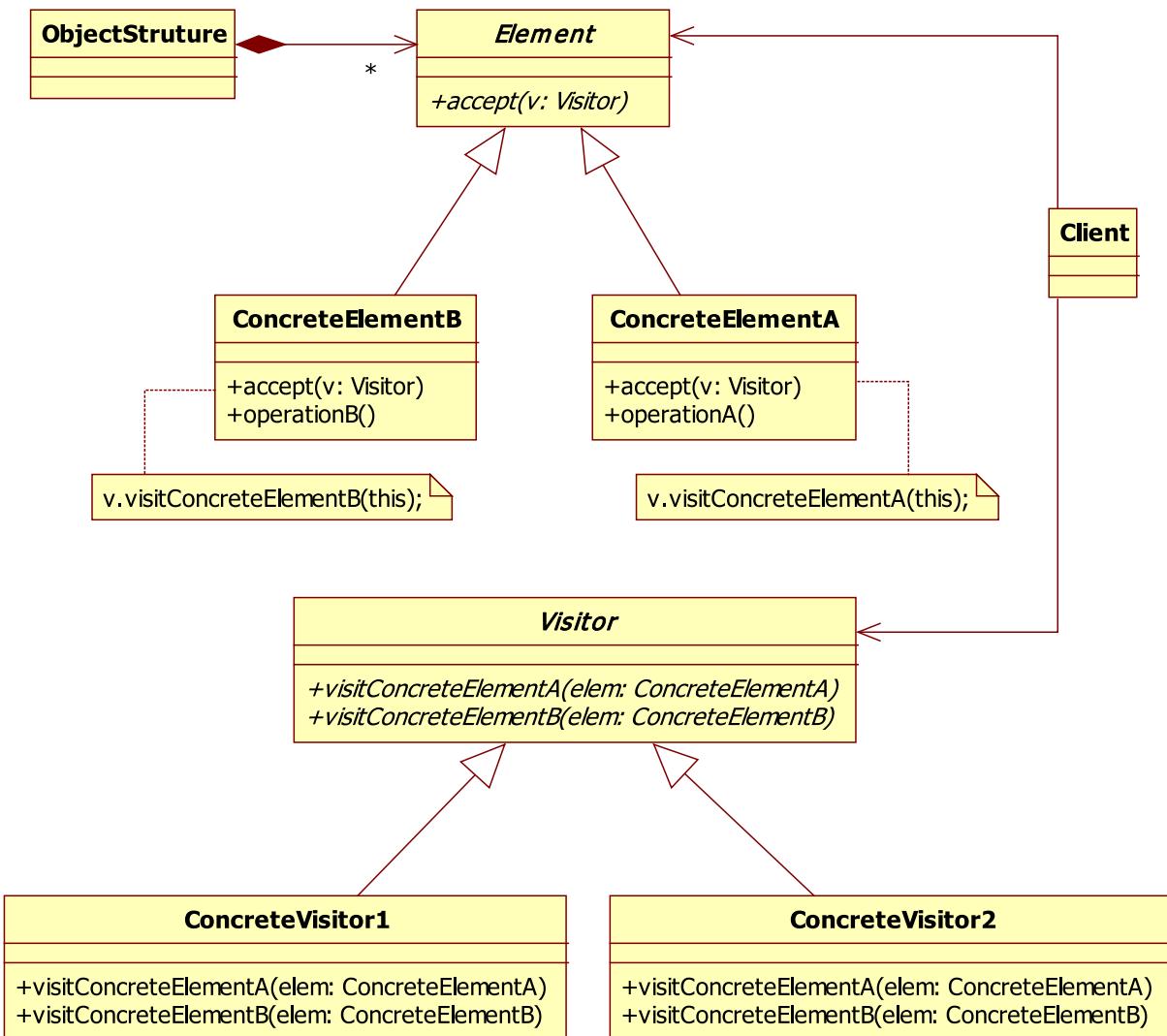


Figure 63. Patron Visiteur : structure

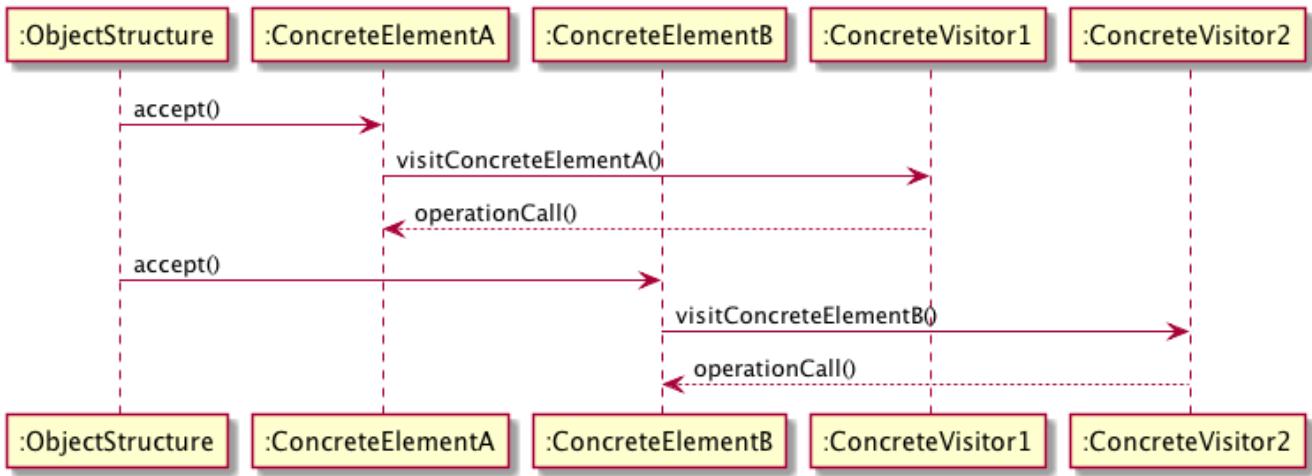


Figure 64. Patron Visiteur : comportement

## Patrons structurels

### Adaptateur (*Adaptor*<sup>uk</sup>)

**Adaptateur (*Adaptor*)** permet de convertir l'interface d'une classe en une autre conformément à l'attente du client. L'Adaptateur permet à des classes de collaborer, alors qu'elles n'auraient pas pu le faire du fait d'interfaces incompatibles.

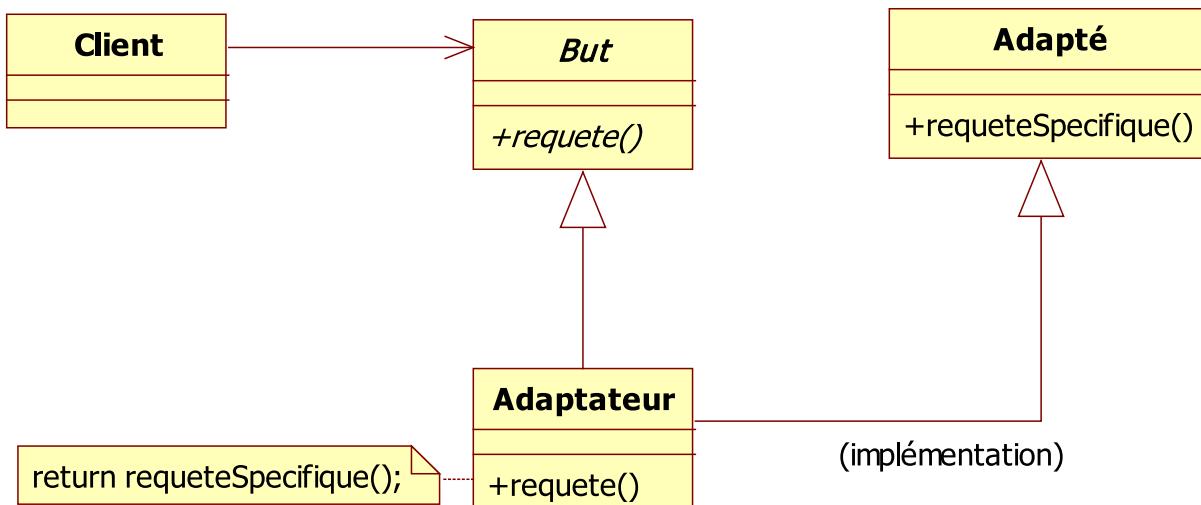


Figure 65. Modèle UML du patron Adaptateur

### Composite

**Composite** permet de composer des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Permet au client de traiter d'une façon unique les objets et les combinaisons d'objets.

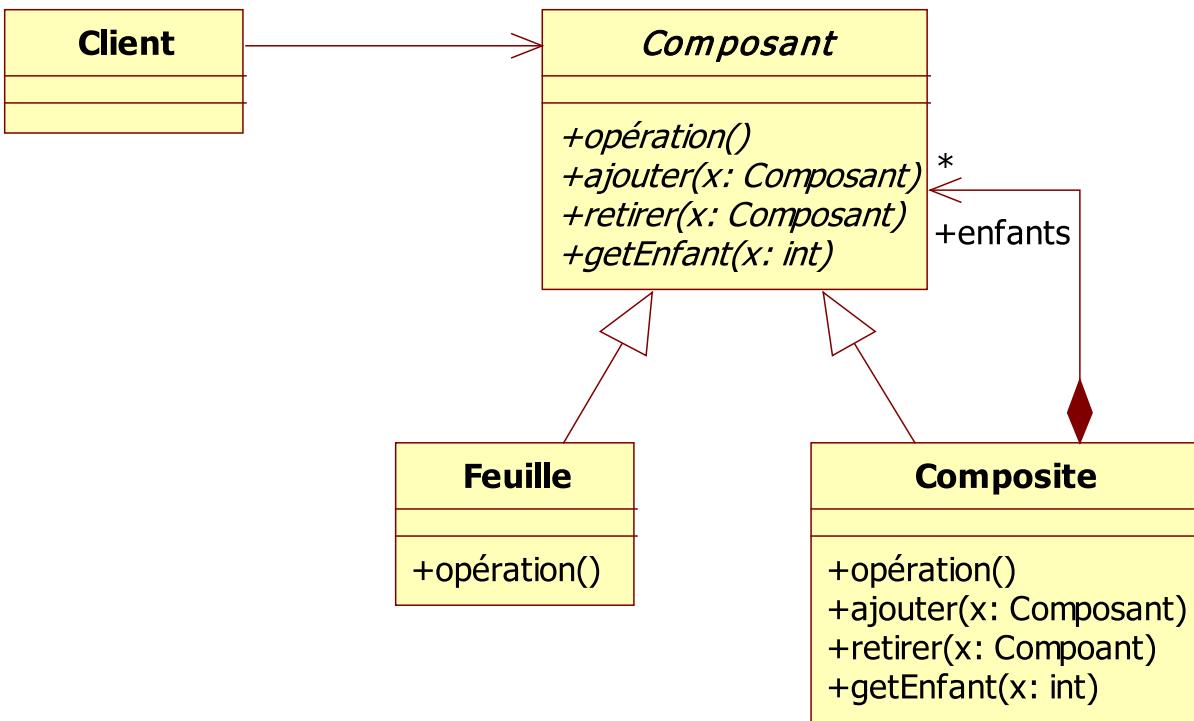


Figure 66. Modèle UML du patron Composite

### Procuration (Proxy<sup>uk</sup>)

**Procuration (Proxy)** fournit à un tiers un mandataire ou un remplaçant, pour contrôler l'accès à cet objet.

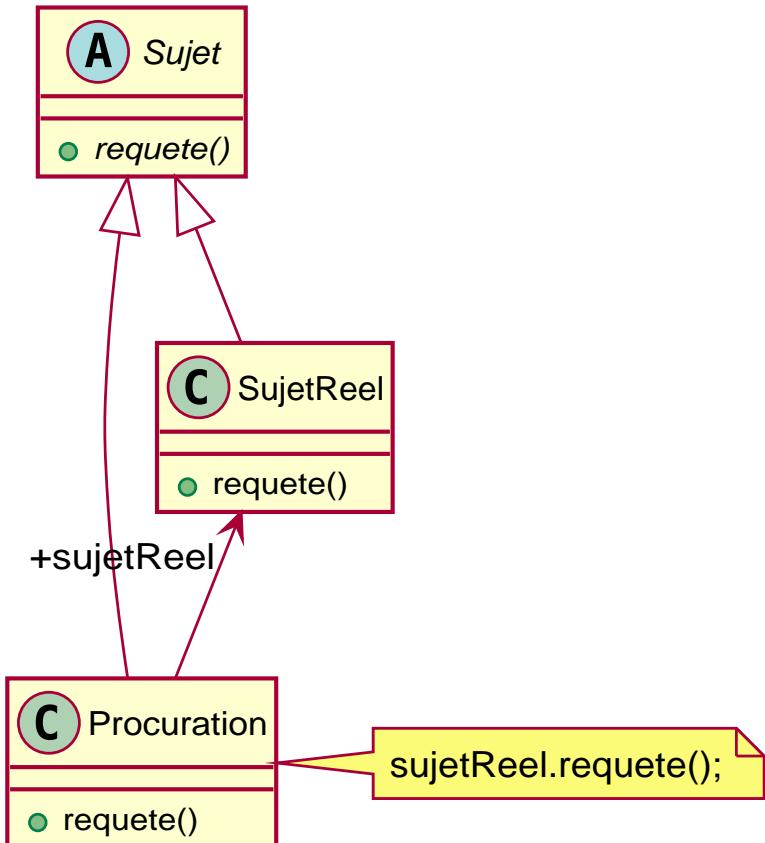


Figure 67. Modèle UML du patron Proxy

# Références

- [Cysboy] Apprenez à programmer en Java. Par [cysboy](#). Disponible [ici](#) (le 2022-01-11).
- [Freeman04] Design Patteren - Head First. Bert Bates, Eric Freeman, Elisabeth Freeman, Kathy Sierra. O'Reilly, 09/2004.
- [Freeman05] Tête la première : Design Pattern. Bert Bates, Eric Freeman, Elisabeth Freeman, Kathy Sierra. Editions O'Reilly. 2005.
- [GOF] Design Patterns: Elements of reusable object oriented software. 1994.
- GOPROD - De bonnes pratiques au service de la conception orientée objets. Disponible [ici](#) (le 2022-01-11).
- [Larman05] Larman, Craig. Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd ed.). Prentice Hall. 2005. ISBN 0-13-148906-2.
- [Martin03] “Principles Of OOD”, Robert C. Martin (“Uncle BOB”), <http://butunclebob.com>.
- [Meyer88] Meyer, Bertrand. Object-Oriented Software Construction. Prentice Hall. 1988. ISBN 0-13-629049-3.
- [SOLID] <https://blogs.msdn.microsoft.com/cdndevs/2009/07/15/the-solid-principles-explained-with-motivational-posters/>

## Exercices corrigés

Les exercices qui suivent servent à apprêhender les patrons de conception et les situations où on en a besoin. Ils sont très largement inspirés de l'excellente série *Head First* de la société O'Reilly, et plus précisément [Freeman04] qui n'est malheureusement plus édité en français.

Les enseignants qui souhaitent utiliser la version PDF des énoncés (sans les corrections) les trouveront sur le site du cours : link:<http://bit.ly/jmb-cpoa>.

Pour ce qui concerne les TP, il aurait été trop long de les mettre dans ce livre, et ils ont véritablement vocation à être lus et utilisés sur machine. Nous en donnons donc un à titre d'exemple en fin d'annexes, mais là aussi, vous pourrez utiliser ceux disponibles sur link:<http://bit.ly/jmb-cpoa>.

## Appendix A: CPOA - Support TD 1



Version corrigée



Cette version comporte des indications pour les réponses aux exercices.

PreReq	1. Je sais programmer en <a href="#">Java</a> . 2. J'ai conscience qu'il faut réfléchir avant de se lancer dans le codage. 3. Je maîtrise les concepts objet de base (héritage, polymorphisme, ...).
ObjTD	Comprendre ce qu'est une <b>conception</b> .
Durée	<b>1 TD et 2 TP de 1,5h</b> (sur 2 semaines différentes).

## A.1. Rappel du cours

Ce TD étant le premier de l'année, aucune référence au cours de cette année n'est requise. Le cours du semestre dernier (M2104) est considéré comme acquis!



Mais pour les prochains, n'hésitez pas à (re)lire régulièrement le [site du cours](#).

## A.2. L'application SuperCanard



Les exercices de ce TD sont tirés de l'excellent livre "Tête la première : Design Pattern". Bert Bates, Eric Freeman, Elisabeth Freeman, Kathy Sierra. Editions O'Reilly. 2005.

### A.2.1. Application existante

Soit l'application (un jeu de simulation de mare aux canards) **SuperCanard** dont le modèle est décrit ci-dessous :

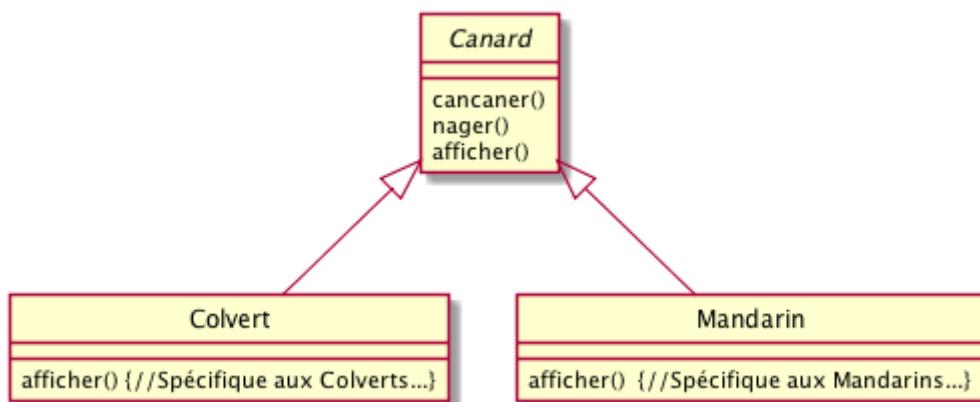


Figure 68. Extrait d'une application existante (source plantUML [ici](#))



De nombreuses autres classes héritent de **Canard**.

Voici un exemple de code :

### Première version de Canard.java

```
abstract public class Canard {  
  
    public void cancaner() {  
        System.out.println("Je cancane comme un Canard!");  
    }  
  
    public void nager() {  
        System.out.println("Je nage comme un Canard!");  
    }  
  
    abstract public void afficher();  
}
```

### Première version de Colvert.java

```
public class Colvert extends Canard {  
  
    public void afficher() {  
        System.out.println("Je suis un Colvert");  
    }  
  
}
```

## A.2.2. Modification/Amélioration

Votre hiérarchie vous demande maintenant d'améliorer l'application pour être plus proche de la réalité.

Vous décidez d'ajouter une méthode **voler()** à vos canards :

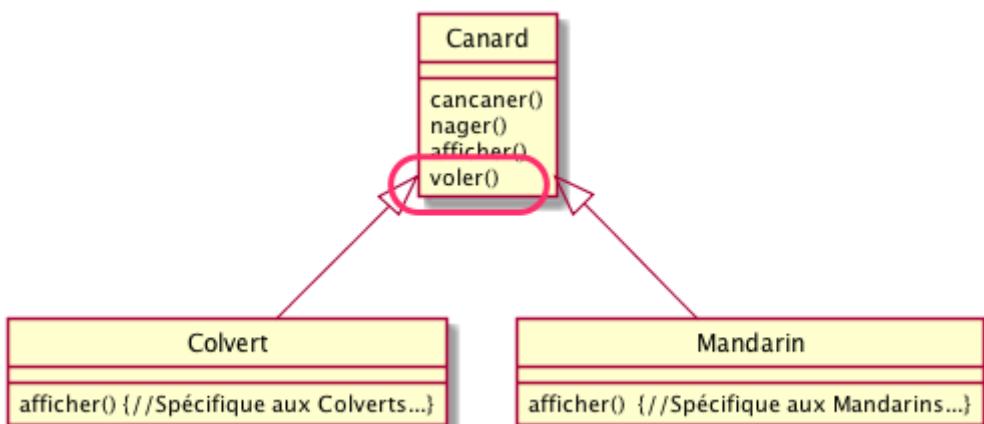


Figure 69. Nouvelle fonctionnalité

```
abstract public class Canard {  
  
    public void cancaner() {  
        System.out.println("Je cancane comme un Canard!");  
    }  
  
    public void nager() {  
        System.out.println("Je nage comme un Canard!");  
    }  
  
    abstract public void afficher();  
  
    public void voler() {  
        System.out.println("Je vole comme un Canard!");  
    };  
}
```

### A.2.3. Catastrophe!

La hiérarchie vous appelle en urgence : des canards en plastiques se mettent à voler dans l'application! En plus, certains canards malades, qui ne devraient pas voler, volent!



Vous avez oublié que certains canards ne volaient pas!



#### QUESTION

Complétez la phrase suivante : L'**héritage** c'est super pour faire de la ..... mais c'est plus problématique pour les aspects .....



#### Solution

L'**héritage** c'est super pour faire de la **réutilisation** mais c'est plus problématique pour les aspects **maintenance** (ou **évolution**).

### A.2.4. Solution 1 : redéfinition de méthodes

Vous songez à une première solution simple : redéfinir la méthode **voler()** dans les canards qui ne volent pas.

## QUESTION

Complétez le code java suivant pour réaliser cette solution :

```
public class CanardEnPlastique extends Canard {  
  
    @Override  
    public void afficher() {  
        System.out.println("Je suis un CanardEnPlastique!");  
    }  
  
}
```



## Solution

```
public class CanardEnPlastique extends Canard {  
  
    @Override  
    public void cancaner() {  
        System.out.println("Je couine puisque je suis en plastique!");  
    }  
  
    @Override  
    public void nager() {  
        System.out.println("Je flotte comme du plastique!");  
    }  
  
    @Override  
    public void voler() {  
        System.out.println("Je ne vole pas, je suis en plastique!!!!");  
    }  
  
    @Override  
    public void afficher() {  
        System.out.println("Je suis un CanardEnPlastique!");  
    }  
  
}
```



## QUESTION

Dans la liste ci-après, quels sont les inconvénients à utiliser l'héritage pour définir le comportement de **Canard** ? (Plusieurs choix possibles.) :



- Le même code est dupliqué (réécrit) entre les sous-classes.
- Les changements de méthodes de comportements au moment de l'exécution sont difficiles.
- Nous ne pouvons pas avoir de canards qui dansent.
- Il est difficile de connaître tous les comportements des canards
- Les canards ne peuvent pas voler et cancaner en même temps.
- Les modifications peuvent affecter involontairement d'autres canards.

## Solution



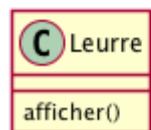
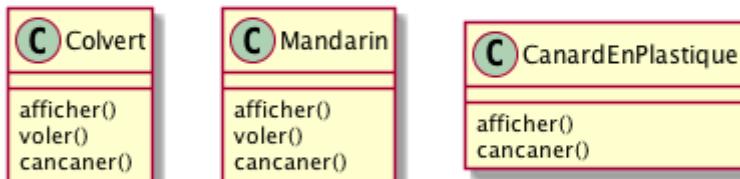
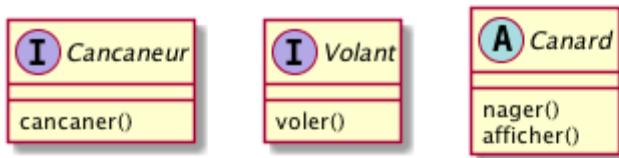
- Le même code est dupliqué (réécrit) entre les sous-classes.
- Les changements de méthodes de comportements au moment de l'exécution sont difficiles.
- Nous ne pouvons pas avoir de canards qui dansent.
- Il est difficile de connaître tous les comportements des canards
- Les canards ne peuvent pas voler et cancaner en même temps.
- Les modifications peuvent affecter involontairement d'autres canards.

### A.2.5. Solution 2 : utilisation des interfaces

Vous songez à utiliser les interfaces pour améliorer le code.

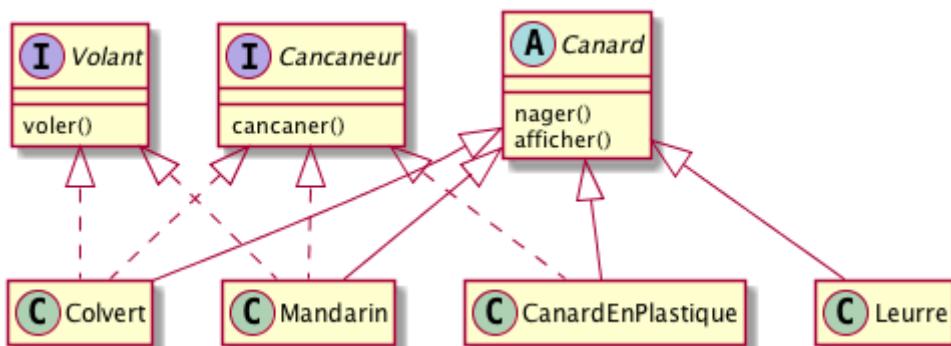
## QUESTION

1. Sur le diagramme suivant indiquez les relations d'héritage (`extends` de java) et d'implémentation (`implements` de java) :



2. Que pensez-vous de la conception obtenue ?

*Solution*



## A.2.6. Solution 3 : isoler ce qui varie

Vous êtes confrontés au même problème que dans le module MPA de ce début d'année : LE CHANGEMENT!

Nous allons donc appliquer un bon Principe de conception :

*Principe de conception*



Identifiez les aspects de votre code qui varient et séparez-les de ceux qui demeurent constants.



## QUESTION

Quels sont les deux principales choses qui varient dans votre code?



### Solution

`voler()` et `cancaner()` : des comportements.

## Implémentation des comportements

Commençons par implémenter les comportements de manière séparée. Pour cela nous rappelons un bon principe que vous avez déjà utilisé :



### Principe de conception

Programmer une interface, non une implémentation.

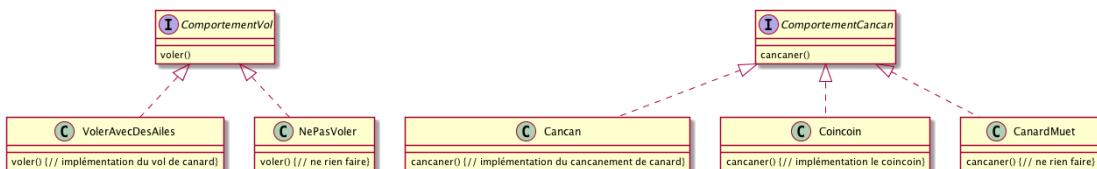


## QUESTION

En appliquant le principe ci-dessus, proposez une conception (diagramme de classe uniquement) avec les classes et/ou interfaces (à vous de juger) suivantes : `ComportementVol`, `VolerAvecDesAiles`, `NePasVoler`.

### Solution

Nous avons ajouté à la solution la version pour `ComportementCancan`.



Question : quelle est la différence avec une classe abstraite au lieu d'une interface pour les comportements?

## Intégration du comportement

Il nous faut maintenant relier les classes de canards à leur comportement.



## QUESTION

1. Ajouter à la classe `Canard` deux attributs pour référencer les comportements.
2. Enlevez les méthodes devenues inutiles.
3. Remplacez-les (donnez les implémentations) par les méthodes `effectuerVol()` et `effectuerCancan()` (qui utilisent les attributs précédents).
4. Modifiez les constructeurs de `Colvert` (par exemple) pour indiquer comment vos attributs sont initialisés.
5. Ajouter à la classe `Colvert` la méthode `setMalade()` et `setGueri()` qui permettent au run-time de modifier le comportement de vol.



### *Solution*

#### *Nouvelle version de [Canard.java](#)*

```
/**  
 * @author bruel  
 * @navassoc -- comportementVol ComportementVol  
 * @navassoc -- comportementCancan ComportementCancan  
 */  
abstract public class Canard {  
  
    /**  
     * @overrideAssoc  
     */  
    protected ComportementVol comportementVol;  
    /**  
     * @overrideAssoc  
     */  
    protected ComportementCancan comportementCancan;  
  
    public final void effectuerCancan() {  
        comportementCancan.cancaner();  
    }  
  
    public void nager() {  
        System.out.println("Je nage comme un Canard!");  
    }  
  
    abstract public void afficher();  
  
    //  
    public final void effectuerVol() {  
        comportementVol.volter();  
    }  
}
```

#### *Nouvelle version de*

#### *[Colvert.java](#)*

```

public class Colvert extends Canard {

    public Colvert() {
        comportementCancan = new Cancan();
        comportementVol = new VolerAvecDesAiles();
    }
    @Override
    public void afficher() {
        System.out.println("Je suis un Colvert");
    }

}

```

Pour la dernière question :

```

public void setMalade () { this.comportementVol = new NePasVoler(); }
public void setGueri () { this.comportementVol = new
VolerAvecDesAiles(); }

```

## Résumé et mise en oeuvre

Il est temps maintenant de prendre du recul et d'expérimenter les avantages de notre nouvelle conception.

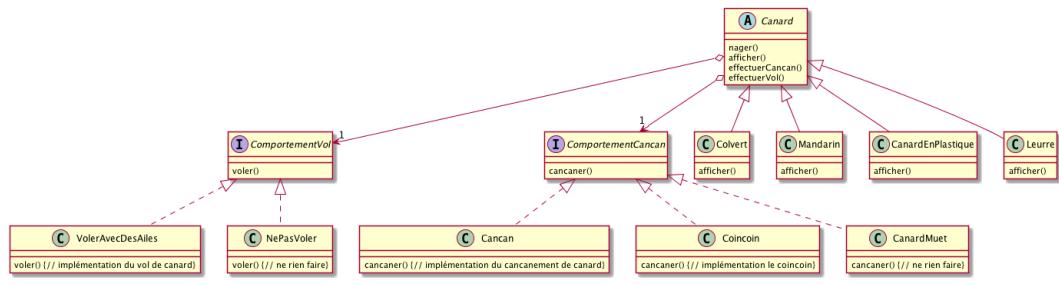
### QUESTION

- 1. Réalisez le diagramme de classe complet de l'application.
- 2. Que feriez-vous pour ajouter la propulsion à réaction à l'application?
- 3. Voyez-vous une classe qui pourrait utiliser le comportement de **Cancan** et qui n'est pas un **Canard**?



## Solution

- Réalisez le diagramme de classe complet de l'application.



- Que feriez-vous pour ajouter la propulsion à réaction à l'application?

Tout simplement créer une classe `VolAReaction` qui implémente l'interface `ComportementVol`.

- Voyez-vous une classe qui pourrait utiliser le comportement de `Cancan` et qui n'est pas un `Canard`?

Par exemple un appeau!

### A.2.7. Votre premier *Design Pattern*

#### La pattern Stratégie

En fait vous venez de mettre en oeuvre votre premier *Design Pattern* : le patron **Strategy** (**Stratégie** en français).

*Design pattern : Stratégie (Strategy)*

**Stratégie** définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Il permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

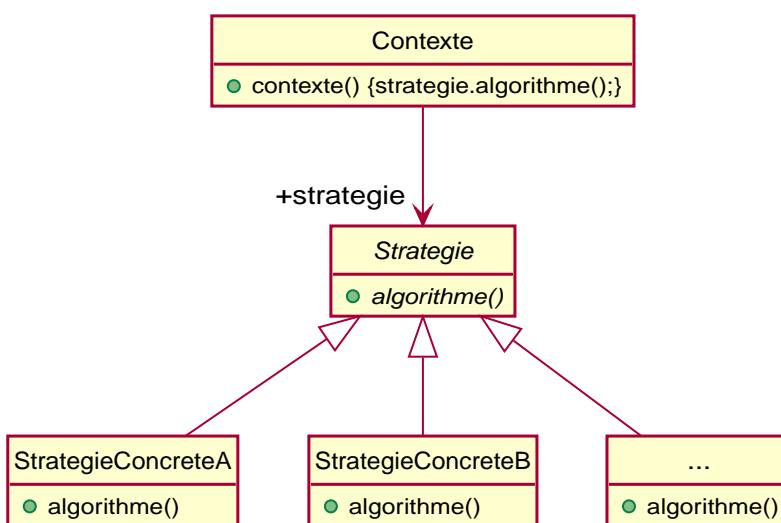


Figure 70. Modèle UML du patron *Strategy*

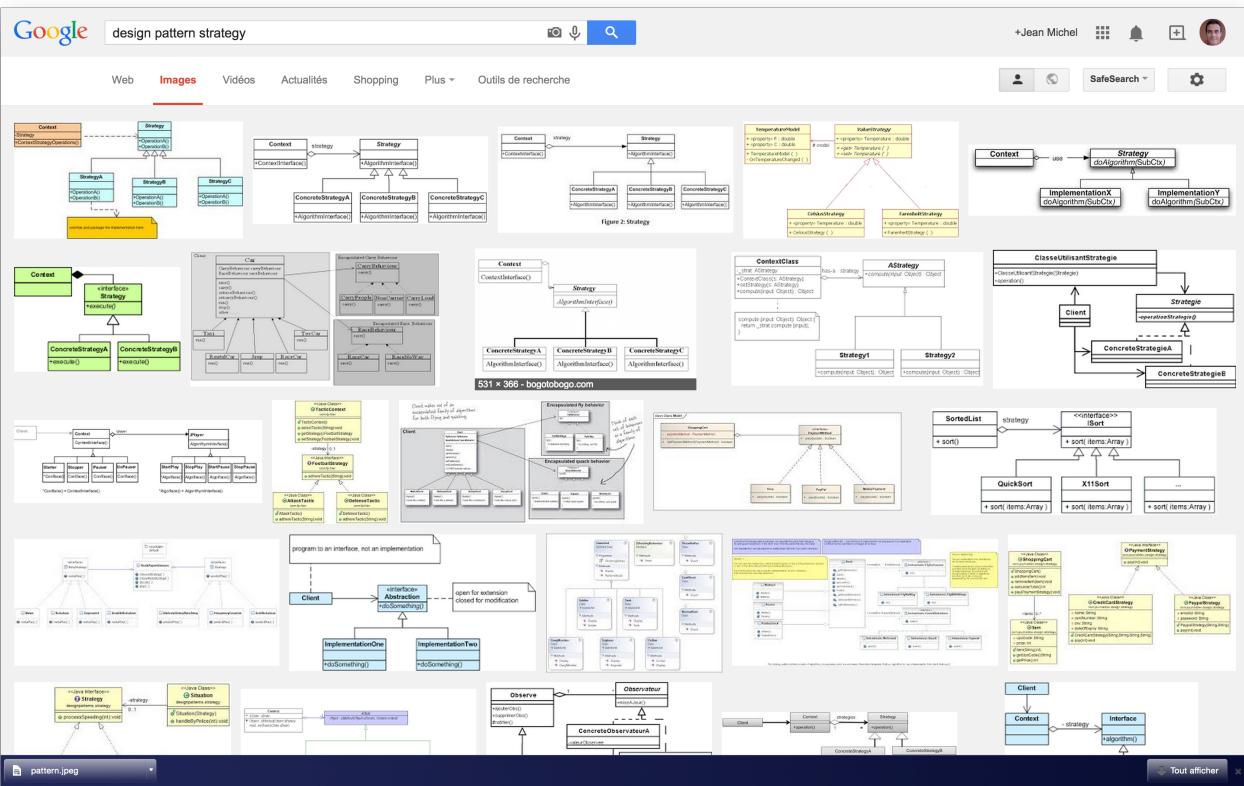


Figure 71. Quelques exemples de description du patron Strategy

## Mise en oeuvre et révision

On vous demande de reprendre un jeu d'aventure dont seul le modèle ci-dessous est fourni.

Personnage

Reine

Roi

ComportementPoignard

ComportementArc

ComportementEpee



ComportementArme

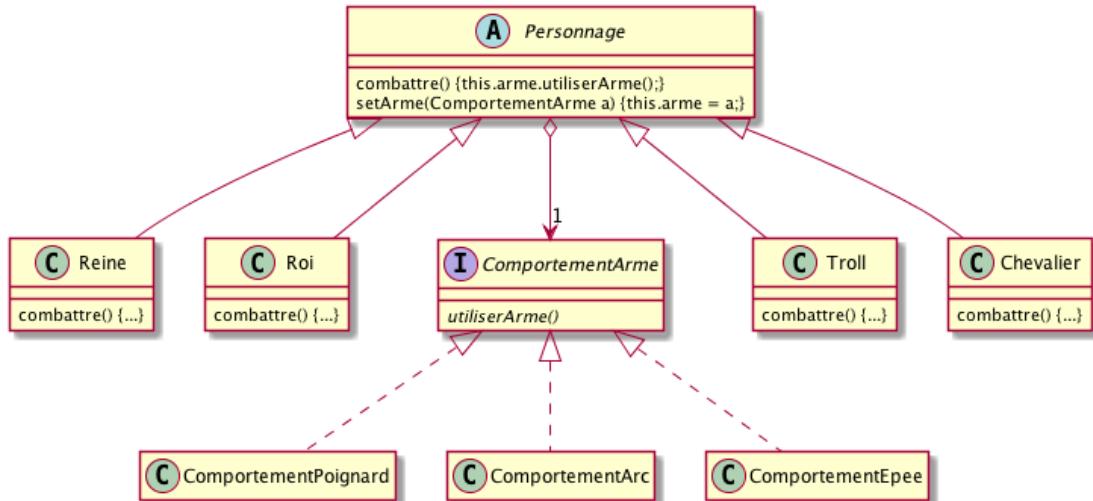
Troll

Chevalier

1. Réorganiser les classes
2. Identifier les classes abstraites, les interfaces et les classes ordinaires.
3. Tracer les liens entre les classes ("est un", implémentation, "a un")
4. Placer la méthode `setArme()` ci-dessous :

```
setArme(ComportementArme a) {  
    this.arme = a;  
}
```

Solution



Certains étudiants font même du zèle quand ils sont motivés :

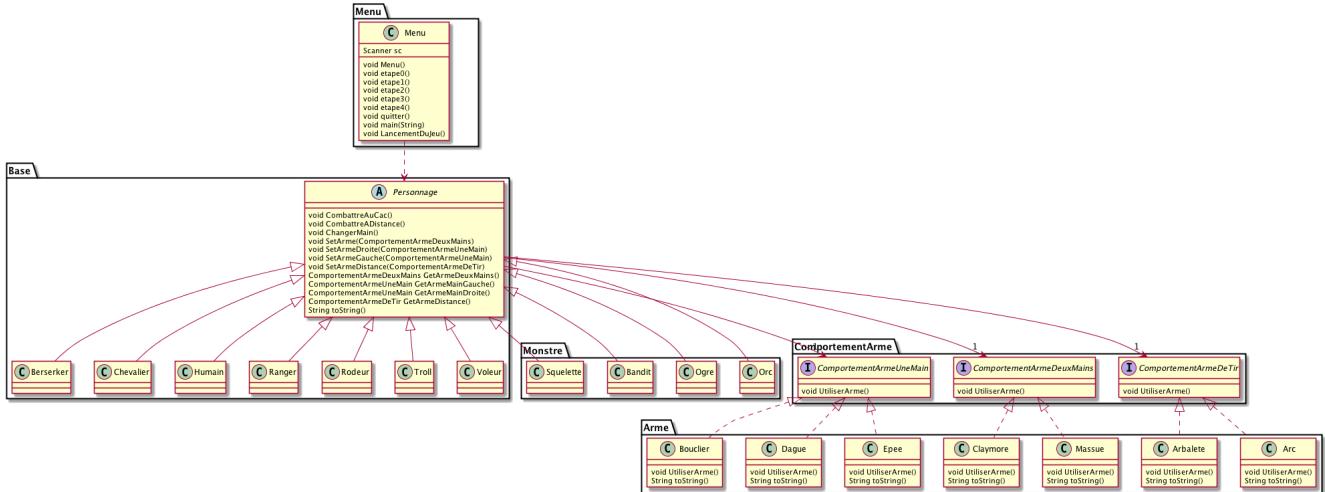


Figure 72. Le modèle de Sylvain Frappez (2015)

## Pour aller plus loin

Nous avons utilisé sans le nommer un troisième bon principe :



### *Principe de conception*

Préférez la composition à l'héritage.



### QUESTION

Quelle différence entre notre conception finale et une implémentation du type :

```
abstract public class Canard implements ComportementVol {...}
```



### QUESTION

Comment testeriez-vous la mise en oeuvre du patron [Strategy](#)?



### *Exemple de solution possible*

```
require "minitest/autorun"
MODEL_NAME = "model.uml"

module MiniTest
  class Unit
    class TestCase
      #Define new assertion
      def assert_contains(string_to_test, substring_to_verify)
        assert_match( string_to_test, substring_to_verify)
      end
      def assert_not_contains(string_to_test, substring_to_verify)
        assert_not_match( string_to_test, substring_to_verify)
      end
    end
  end
end
```

```

end
MiniTest::Unit.after_tests { p @assertions }

class TestGeneratedModel < MiniTest::Unit::TestCase
  #----- General tests about plantUML
  def test_generated_model_exists
    print assert_equal(true, File.exists?(MODEL_NAME))
  end

  def test_generated_model_is_plantuml
    assert_equal(true,
      File.readlines(MODEL_NAME).grep(/@startuml/).any?
      assert_equal(true, File.readlines(MODEL_NAME).grep(/@enduml/).any?))
  end

  def test_generated_model_exists
    assert_equal(true, File.exists?(MODEL_NAME))
  end

  #----- Specific tests about expected content
  def test_class_Canard_is_abstract
    assert_equal(true, File.readlines(MODEL_NAME).grep(/abstract
Canard/).any?)
  end

  def test_class_Canard_has_ComportementCancan_behavior
    assert_contains(/Canard\s+--> .* ComportementCancan/, 
File.readlines(MODEL_NAME).join)
  end

  def test_class_Canard_has_ComportementVol_behavior
    assert_contains(/Canard\s+--> .* ComportementVol/, 
File.readlines(MODEL_NAME).join)
  end

  def test_ComportementCancan_is_an_Interface
    assert_equal(true,
      File.readlines(MODEL_NAME).grep(/interface\s+ComportementCancan/).any?)
  end

  def test_ComportementVol_is_an_Interface
    assert_equal(true,
      File.readlines(MODEL_NAME).grep(/interface\s+ComportementVol/).any?)
  end

  def test_ComportementCancan_Interface_has_concrete_implementation
    assert_equal(true,
      File.readlines(MODEL_NAME).grep(/ComportementCancan\s+<\|\.\.\./).any?)
  end

  def test_ComportementVol_Interface_has_concrete_implementation

```

```

    assert_equal(true,
File.readlines(MODEL_NAME).grep(/ComportementVol\s+<\|\.\.\./).any?)
end

end

```



N'hésitez pas à consulter un autre exemple, orienté "jeux de rôle", [ici \(p. 116\)](#).

## Appendix B: CPOA - Support TD 2



*Version corrigée*



Cette version comporte des indications pour les réponses aux exercices.

PreReq	<ol style="list-style-type: none"> <li>1. Je sais programmer en <a href="#">Java</a>.</li> <li>2. J'ai conscience qu'il faut réfléchir avant de se lancer dans le codage.</li> <li>3. Je maîtrise les concepts objet de base (héritage, polymorphisme, ...).</li> <li>4. J'ai compris ce qu'est un patron et j'ai grand soif d'en apprendre d'autres que <i>Strategy</i></li> </ol>
ObjTD	Aborder le patron <b>singleton</b>
Durée	<b>1</b> TD et <b>2</b> TP de 1,5h (sur 2 semaines).

### B.1. Rappel du cours



N'hésitez pas à (re)lire régulièrement le [Support de Cours](#).

### B.2. La fabrique de chocolat

Vous participez au développement d'un simulateur de fabriques de chocolat modernes dont des bouilleurs sont assistés par ordinateur.

La tâche du bouilleur consiste à contenir un mélange de chocolat et de lait, à le porter à ébullition puis à le transmettre à la phase suivante où il est transformé en plaquettes de chocolat.

#### B.2.1. Problème initial

Voici la classe contrôleur du bouilleur industriel de *Bonchoco, SA*.

```
/**  
 * @author bruel (taken from Design Pattern - Head First, O'Reilly, 09/2004)  
 */  
public class BouilleurChocolat {  
    private boolean vide;  
    private boolean bouilli;  
  
    public BouilleurChocolat() {  
        vide = true;  
        bouilli = false;  
    }  
  
    public void remplir() {  
        if (estVide()) {  
            vide = false;  
            bouilli = false;  
            // remplir le bouilleur du mélange lait/chocolat  
        }  
    }  
  
    public void vider() {  
        if (!estVide() && estBouilli()) {  
            // vider le mélange  
            vide = true;  
        }  
    }  
  
    public void bouillir() {  
        if (!estVide() && !estBouilli()) {  
            // porter le contenu à ébullition  
            bouilli = true;  
        }  
    }  
  
    public boolean estVide() { return vide; }  
  
    public boolean estBouilli() { return bouilli; }  
}
```



#### QUESTION

1. À quoi servent les attributs **vide** et **bouilli**?



*Solution*



Si vous étudiez le code, vous constatez qu'ils ont essayé très soigneusement d'éviter les catastrophes, par exemple de vider deux mille litres de mélange qui n'a pas bouilli, de remplir un bouilleur déjà plein ou de faire bouillir un bouilleur vide !

Vous faites un cauchemar horrible (quoique) où vous vous noyez dans du chocolat. Vous vous réveillez en sursaut avec une crainte terrible.

### QUESTION



1. Que pourrait-il se passer avec plusieurs instances de contrôleurs (pour un seul et même bouilleur)?
2. De quoi faudrait-il s'assurer pour éviter ce problème?
3. Trouvez des exemples de situations où il est important de n'avoir qu'une seule instance d'une classe donnée.



*Solution*



1. Que l'un remplisse alors que l'autre n'a pas vidé par exemple.
2. S'assurer de n'avoir qu'une seule instance de ce contrôleur.
3. Quelques exemples :
  - accès unique à une base de données (on vient de le voir)
  - objet "parent" d'une interface
  - ...

### B.2.2. Amélioration 1

Vous vous souvenez des premiers exercices [Java](#) sur les variables de classe et vous proposez d'utiliser un compteur d'instance pour solutionner le problème.

## QUESTION

Vous essayez de modifier le constructeur pour qu'il ne fonctionne que si le compteur d'instance est à 0. Qu'est-ce qui ne va pas dans le code suivant :

```
public class BouilleurCptChocolat {  
    private boolean vide;  
    private boolean bouilli;  
    private static int nbInstance = 0;  
  
    public BouilleurCptChocolat() {  
        vide = true;  
        bouilli = false;  
        if (nbInstance == 0) {  
            nbInstance = 1;  
            return this;  
        }  
        else {  
            return null;  
        }  
    }  
}
```



Pas de return dans un constructeur.

### B.2.3. Amélioration 2

Vous changez de stratégie car vous vous souvenez avoir déjà vu ce type de code :

*Idée!*

```
public class MaClasse {  
    private MaClasse() {...}  
}
```

## QUESTION

1. Est-ce autorisé de rendre privé le constructeur?
2. Comment créer une instance dans ces conditions? N'a-t'on pas tout simplement une classe inutilisable?
3. Complétez le code suivant de façon à résoudre le problème :

```
public class BouilleurChocolat {  
    private boolean vide;  
    private boolean bouilli;  
    ...  
    ...  
  
    BouilleurChocolat() {  
        ...  
        ...  
    }  
  
    ...  
    ...  
    ...  
    ...  
  
    public void remplir() {  
        if (estVide()) {  
            vide = false;  
            bouilli = false;  
            // remplir le bouilleur du mélange lait/chocolat  
        }  
        // reste du code de BouilleurChocolat...  
    }  
}
```



4. Donnez un exemple d'utilisation de cette classe.

1. Oui!
  2. En implémentant une fonction qui s'en charge.

## *Extrait de la solution*

```
public class BouilleurSafeChocolat {  
    private boolean vide;  
    private boolean bouilli;  
    private static BouilleurSafeChocolat uniqueInstance;  
  
    private BouilleurSafeChocolat() {  
        vide = true;  
        bouilli = false;  
    }  
  
    public static final BouilleurSafeChocolat getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new BouilleurSafeChocolat();  
        }  
        return uniqueInstance;  
    }  
}
```



#### B.2.4. C'est pas fini!

Vos cauchemars continuent! Mais cette fois ils sont en anglais! Vous voyez un grand gaillard irlandais vous menacer (en fait vous confondez *threat* et *thread*...).

## QUESTION

1. En quoi les *threads* peuvent-ils poser des problèmes dans votre solution?
  2. Recopiez sur des bouts de feuilles les fragments de code ci-dessous en les plaçant dans les colonnes du tableau suivant pour mettre en évidence le problème en reconstituant un enchaînement erroné possible avec deux threads. :



Bloc 1

```
public static BouilleurChocolat getInstance() {
```

Bloc 2

```
if (uniqueInstance == null) {
```

Bloc 3

```
    uniqueInstance = new BouilleurSafeChocolat();
```

Bloc 4

```
}
```

Bloc 5

```
return uniqueInstance;
```

Bloc 6

```
}
```

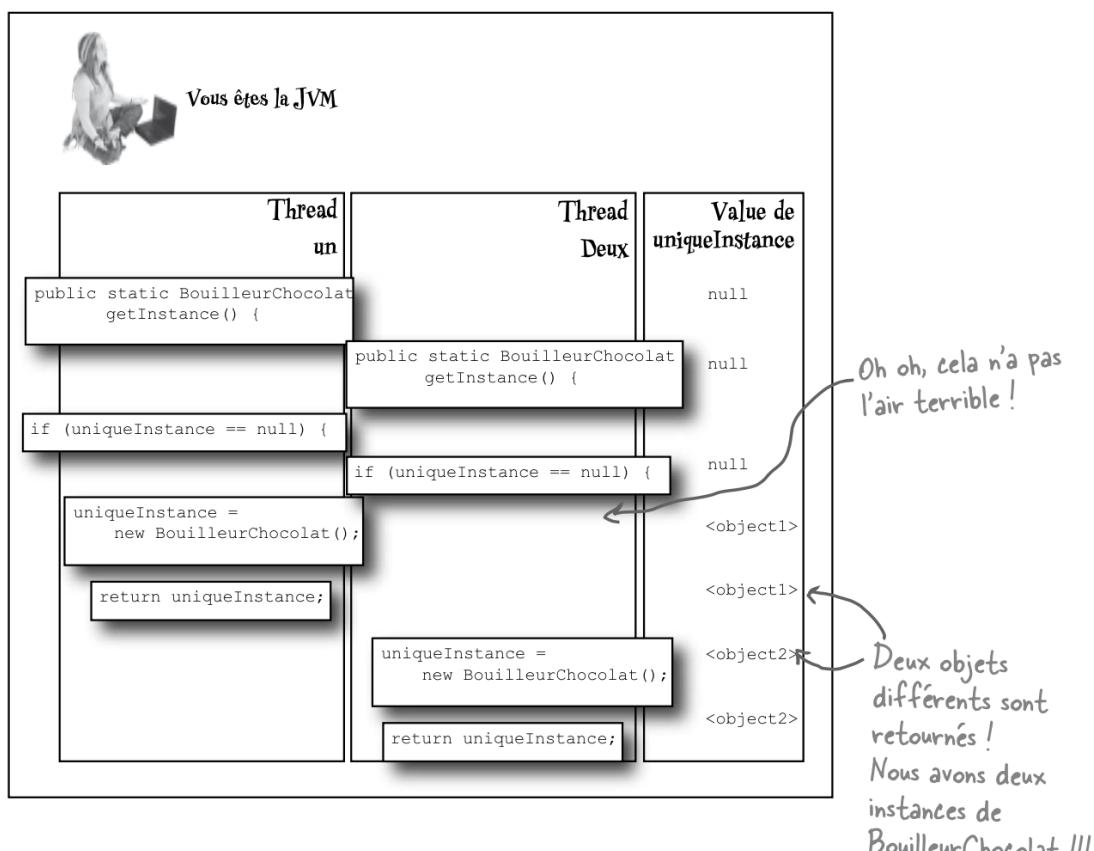


Figure 73. Solution (source [Freeman05])

```

public class BouilleurSafeChocolat {
    private boolean vide;
    private boolean bouilli;
    private static BouilleurSafeChocolat uniqueInstance;

    private BouilleurSafeChocolat() {
        vide = true;
        bouilli = false;
    }

    public static final BouilleurSafeChocolat getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new BouilleurSafeChocolat();
        }
        return uniqueInstance;
    }
}

```

Explications :

1. Thread 1 appelle `getInstance()` et détermine que `uniqueInstance` est `null` en ligne 12
2. Thread 1 entre dans le bloc `if` puis est préempté par le thread 2 avant l'exécution de la ligne 13
3. Thread 2 appelle `getInstance()` et détermine que `uniqueInstance` est `null` en ligne 12
4. Thread 2 entre dans le bloc `if`, crée un nouveau `BouilleurSafeChocolat` et assigne ce nouvel objet à la variable `uniqueInstance` en ligne 13
5. Thread 2 retourne la référence au `BouilleurSafeChocolat` en ligne 15
6. Thread 2 est préempté par le Thread 1
7. Thread 1 reprend où il s'était arrêté et exécute la ligne 13 créant alors une autre instance de `BouilleurSafeChocolat`
8. Thread 1 retourne cette nouvelle instance en ligne 15

### B.2.5. Solution au multithreading

Vous vous souvenez heureusement de vos cours de début d'année sur les *threads* :



#### QUESTION

1. Proposez une solution simple à ce problème.

Il suffit de faire de `getInstance()` une méthode **synchronisée** :

```
public class BouilleurSafeChocolat {  
    private boolean vide;  
    private boolean bouilli;  
    private static BouilleurSafeChocolat uniqueInstance;  
  
    private BouilleurSafeChocolat() {  
        vide = true;  
        bouilli = false;  
    }  
  
    public static synchronized BouilleurSafeChocolat getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new BouilleurSafeChocolat();  
        }  
        return uniqueInstance;  
    }  
}
```



## B.2.6. Problème de la solution!!

### QUESTION



1. Combien de fois le mécanisme mis en place va-t'il être utile ?
  2. Que pensez-vous alors de cette solution ?
  3. Proposez une solution où l'instance est créé au démarrage plutôt qu'à la demande.
- 
1. Une seule fois, lors du 1er passage dans la méthode!!
  2. C'est bien trop consomateur en ressource! En pratique, il y a des copies de blocs de mémoire, ce qui prend du temps.
  3. Voici un exemple :

*Création de l'instance unique au démarrage*



```
public class Singleton {  
    private static final Singleton uniqueInstance = new Singleton();  
    private Singleton() {}  
    public static Singleton getInstance() { return uniqueInstance; }  
}
```

En adoptant cette approche, nous nous reposons sur la JVM pour créer l'unique instance du Singleton quand la classe est chargée. La JVM garantit que l'instance sera créée avant qu'un thread quelconque n'accède à la variable statique `uniqueInstance`.



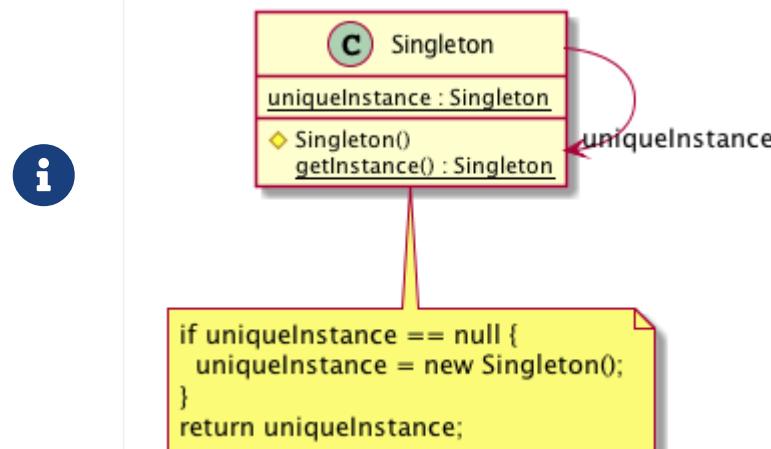
Il peut y avoir des situations où le coût de la synchronisation est inférieur au coût de créer dès le départ une instance (par exemple gourmande en mémoire).

### B.3. Singleton

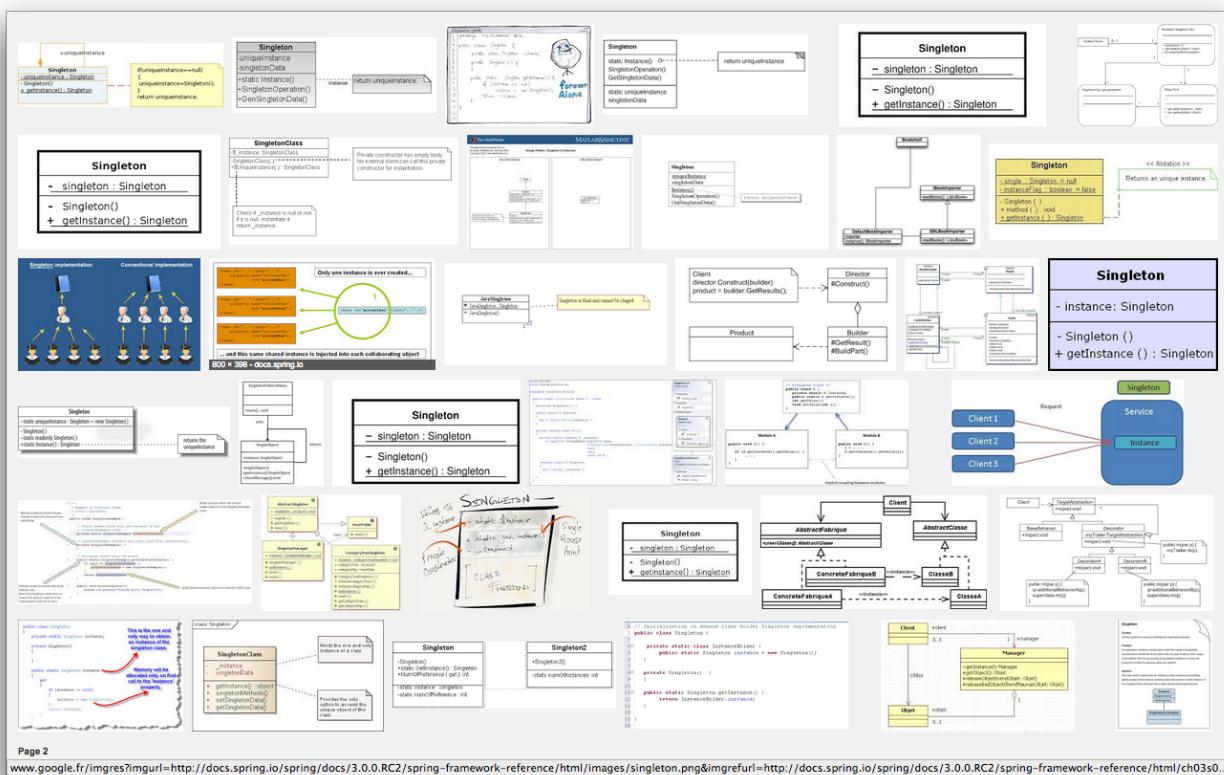
Félicitations, vous venez de mettre en oeuvre votre deuxième patron, le **Singleton**.

## *Design pattern : Singleton*

**Singleton** garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.



*Figure 74. Modèle UML du patron Singleton*



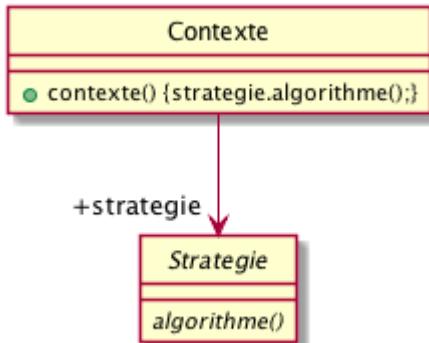
*Figure 75. Quelques exemples de description du patron Singleton*

## B.4. Le singleton pour le jeu d'aventure

### B.4.1. Combiner plusieurs patrons?

Peut-on combiner les deux derniers patrons vus en TD (*Strategy* et *Singleton*)? En effet, les comportements sont portés par des objets pour l'aspect algorithme, mais il n'y a pas de raison de ne pas les partager entre tous les objets qui "utilisent" ce comportement?!

Dans la plupart des cas ces deux patrons ne vont **pas du tout ensemble**. Cette stratégie n'est recommandée que dans un cas bien précis d'utilisation de *Strategy* : celui où les comportements sont simples et "statiques" (pas de consommation de ressources par exemple) et où l'on utilise une association :



Avec une implémentation du type :

```
...
vol = new VolerAvecDesAiles();
cri = new Cancan();
c1 = new Colvert(vol,cri);
...
```

### B.4.2. Et si on améliorait le jeu d'aventure avec Singleton?



#### QUESTION

1. Faites en sorte que les instances d'objet affectées à chaque comportement d'un **Personnage** soient uniques pour chaque comportement distinct.
2. Pourquoi ne devrait-on pas utiliser `getInstance()` dans le cas d'une composition (dans le constructeur du composé) ?

1. Extrait de solution (disponible sur le GitHub pour les profs)

*Extrait de ComportementEpee.java*

```
public class ComportementEpee implements ComportementArme {  
  
    private final static ComportementEpee uniqueInstance = new  
    ComportementEpee();  
  
    public final static ComportementEpee getInstance() {  
        return uniqueInstance;  
    }  
}
```

Qu'on pourra éventuellement enrichir pour être complet avec :



```
private ComportementEpee(){  
  
}
```

*Extrait de Chevalier.java*

```
public class Chevalier extends Personnage {  
  
    protected Chevalier(String nom) {  
        this(nom, ComportementEpee.getInstance(),  
        ComportementACheval.getInstance());  
    }  
}
```



Il faudra alors changer les appels comme `compAdequat = new ComportementEpee();` en `compAdequat = ComportementEpee.getInstance();`

2. Car dans une composition les objets possèdent les instances de leur comportement. Elles sont donc uniques et leurs instances ne doivent pas être transmises, pour être sûr que la destruction du composite détruit les composés.



On voit que ce n'est pas toujours évident de combiner les patrons entre eux.

## Pour aller plus loin



### QUESTION

Quelle est la différence entre un singleton et une variable globale?

Quelques éléments de solution :



- En Java les variables globales sont des références statiques à des objets.
- Problème déjà vu de l'instanciation à la demande vs. au démarrage.



### QUESTION

Comment testeriez-vous la mise en oeuvre du patron [Singleton](#)?



Exemples de test :

- Tentative d'instanciation depuis l'extérieur de la classe
- Tentative de construction de deux objets de type Singleton



### QUESTION

Il existe une autre façon de gérer le problème du multithreading. Cherchez sur Internet les articles sur le "verrouillage à double vérification" (qui ne fonctionne que depuis Java 1.5).



N'hésitez pas à consulter les liens suivants :

- <http://thecodersbreakfast.net/index.php?post/2008/02/25/26-de-la-bonne-implementation-du-singleton-en-java>
- <http://christophej.developpez.com/tutoriel/java/singleton/multithread/>

## Appendix C: CPOA - Support TD 3



*Version corrigée*

Cette version comporte des indications pour les réponses aux exercices.

PreReq	<ol style="list-style-type: none"><li>1. Je sais programmer en <a href="#">Java</a>.</li><li>2. J'ai conscience qu'il faut réfléchir avant de se lancer dans le codage.</li><li>3. Je maîtrise les concepts objet de base (héritage, polymorphisme, ...).</li><li>4. J'ai compris ce qu'est un patron et j'ai grand soif d'en apprendre d'autres que <i>Strategy</i> et <i>Singleton</i></li></ol>
ObjTD	Aborder le patron <b>fabrique</b> .
Durée	<b>1</b> TD et <b>2</b> TP de 1,5h (sur 2 semaines).

## C.1. Rappel du cours



N'hésitez pas à (re)lire régulièrement le [Support de Cours](#).

## C.2. La pizzeria O'Reilly

Vous êtes embauché dans une pizzeria pour faire ... de l'informatique (il y en a bien qui font leur PTUT pour une boulangerie...)!

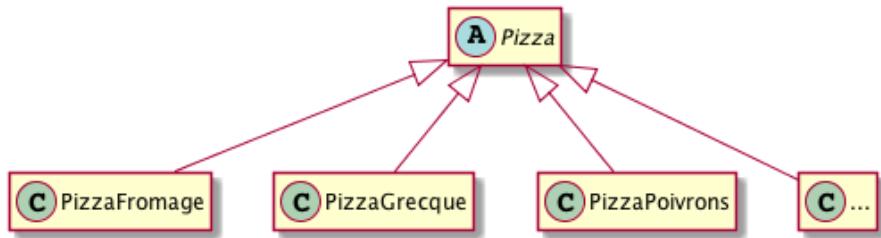
Le stagiaire de l'an dernier qui avait travaillé sur le code est parti avec la caisse (de Chianti). Vous n'avez à votre disposition que :

1. Le code de départ suivant :

*Code de Pizzeria.java*

```
/**  
 * @author bruel (from O'Reilly Head-First series)  
 */  
public class Pizzeria {  
  
    public Pizza commanderPizza(String type) {  
  
        Pizza pizza;  
  
        if (type.equals("fromage")) {  
            pizza = new PizzaFromage();  
        } else if (type.equals("grecque")) {  
            pizza = new PizzaGrecque();  
        } else {  
            pizza = new PizzaPoivrons();  
        }  
  
        pizza.preparer();  
        pizza.cuire();  
        pizza.couper();  
        pizza.emballer();  
  
        return pizza;  
    }  
}
```

2. L'ébauche de diagramme de classe des pizzas suivant :



3. Le bout de code de test suivant :

```

Pizzeria boutiqueBrest = new Pizzeria();
boutiqueBrest.commanderPizza("fromage");
...
Pizzeria boutiqueStrasbourg = new Pizzeria();
boutiqueStrasbourg.commanderPizza("grecque");
  
```

### QUESTION

1. Identifiez ce qui varie dans ce code (si la pression du marché fait ajouter des pizzas à la carte ou si une pizza n'a plus de succès et doit disparaître, etc.).
2. Isolez dans une classe **SimpleFabriqueDePizzas** ce code.
3. Réalisez le diagramme de classe obtenu.
4. Quel est l'avantage de procéder ainsi ? Ne transfère-t-on pas simplement le problème à un autre objet ?



Bien sûr vous héritez de cet horrible "if then else" et dans votre implémentation en TP vous remplacerez ce code avantageusement par un "switch case" et utiliserez une **enum** comme link:[vu en cours](#).



### Solution



1. Le **if** à remplacer par un **pizza = fabrique.creerPizza(type);** et on ajoute le code suivant dans la classe :

```

SimpleFabriqueDePizzas fabrique;

public Pizzeria(SimpleFabriqueDePizzas fabrique) {
    this.fabrique = fabrique;
}
  
```



Faire remarquer que la pizza n'est plus la propriété de la pizzeria, mais de la fabrique!

2. Code de **SimpleFabriqueDePizzas**

```

/*
 * @author bruel
 * @depend - new - Pizza
 */
public class SimpleFabriqueDePizzas {

    public Pizza creerPizza(String type) {
        Pizza pizza;

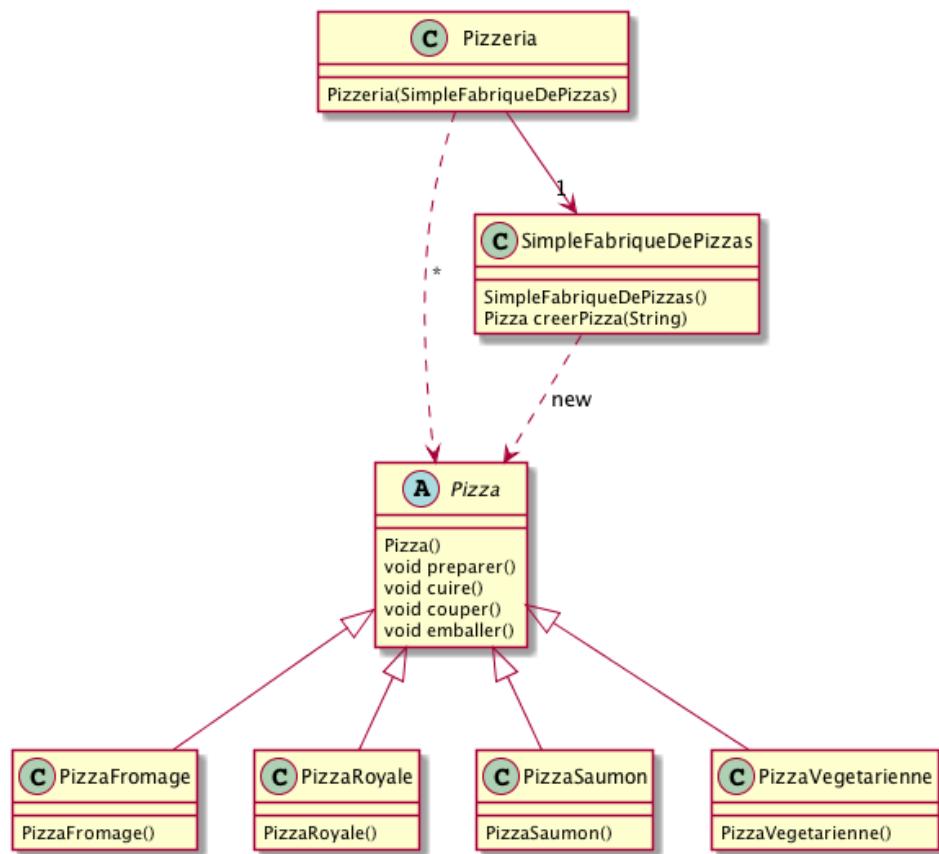
        if (type.equals("fromage")) {
            pizza = new PizzaFromage();
        } else if (type.equals("saumon")) {
            pizza = new PizzaSaumon();
        } else if (type.equals("royale")) {
            pizza = new PizzaRoyale();
        } else {
            pizza = new PizzaVegetarienne();
        }

        return pizza;
    }

}

```

### 3. Diagramme de classe :



### 4. En encapsulant la création des pizzas dans une seule classe, nous n'avons plus

qu'un seul endroit auquel apporter des modifications quand l'implémentation change.

## C.3. On y est presque...

Nous sommes arrivés à une situation propre, qui s'apparente à un patron de conception. Mais avant d'en arriver à la définition du patron lui-même, nous allons améliorer un peu les choses.

### C.3.1. Succès des pizzerias O'Reilly : les franchises

Plusieurs villes veulent ouvrir des pizzerias comme la vôtre. Votre patron, très content de vos programmes souhaite imposer à toutes les futures pizzerias d'utiliser vos codes.

Le problème : les pizzas au fromage de Strasbourg sont différentes des pizzas aux fromages de Corse!



#### QUESTION

Proposez une solution où **SimpleFabriqueDePizzas** serait une classe abstraite.

#### Solution



Simplement ajouter abstract, créer plusieurs sous-classes et avoir une utilisation du style :



```
SimpleFabriqueDePizzas fabriqueBrest = new FabriqueDePizzasBrest();
Pizzeria boutiqueBrest = new Pizzeria(fabriqueBrest);
boutiqueBrest.commander("Végétarienne");
...
SimpleFabriqueDePizzas fabriqueStrasbourg = new
FabriqueDePizzasStrasbourg();
Pizzeria boutiqueStrasbourg = new Pizzeria(fabriqueStrasbourg);
boutiqueStrasbourg.commander("Végétarienne");
```

### C.3.2. La dérive : chacun travaille comme il l'entend!

Les pizzerias utilisent bien vos fabriques mais ont changé leurs procédures : certaines ne coupent pas les pizzas, changent les temps de cuissons, et les pizzerias O'Reilly perdent leur identité. Il nous faut donc **restructurer** les pizzerias.

Un consultant italien payé fort cher (heureusement en pizzas!) propose de revenir à la structure suivante :

```

public abstract class Pizzeria {
    public final Pizza commanderPizza(String type) {
        Pizza pizza;

        pizza = creerPizza(type);
        pizza.preparer();
        pizza.cuire();
        pizza.couper();
        pizza.emballer();

        return pizza;
    }

    .....
    Pizza creerPizza(String type);
}

```



### QUESTION

Quelles sont les différences avec notre conception actuelle?



*Solution*



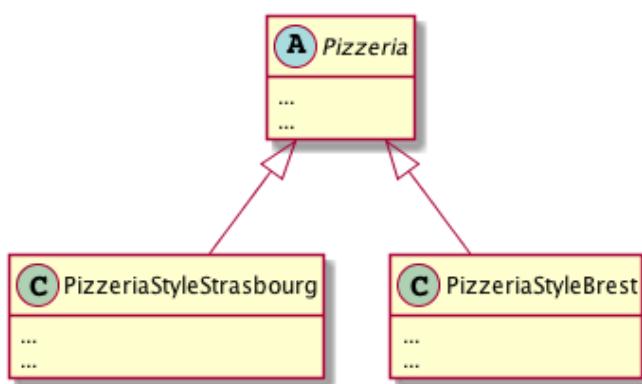
- Pizzeria est maintenant abstraite (vous allez voir pourquoi ci-dessous).
- Maintenant, `creerPizza()` est de nouveau un appel à une méthode de Pizzeria et non à un objet fabrique.
- Notre "méthode de fabrication" est maintenant abstraite dans `Pizzeria`.
- Et nous avons transféré la fonctionnalité de notre objet fabrique à cette méthode.

### C.3.3. Laisser les sous-classes décider

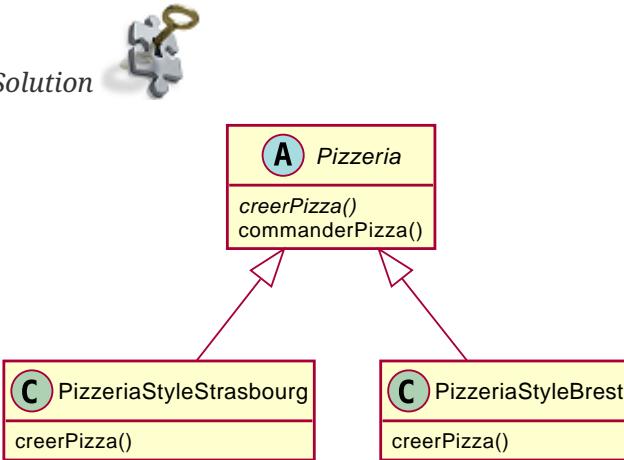


### QUESTION

Dans le schéma suivant, placez les méthodes au bon endroit de façon à ce que les procédures soient respectées tout en ayant des pizzas à variantes "régionales".



Solution



Chaque sous-classe redéfinit la méthode `creerPizza()`, tandis que toutes les sous-classes utilisent la méthode `commanderPizza()` définie dans `Pizzeria`.



Voici un exemple de `Pizzeria` concrète :

```
public class PizzeriaBrest extends Pizzeria {
    Pizza creerPizza(String item) {
        if (choix.equals("fromage")) {
            return new PizzaFromageStyleBrest();
        } else if (choix.equals("vegetarienne")) {
            return new PizzaVegetarienneStyleBrest();
        } else if (choix.equals("fruitsDeMer")) {
            return new PizzaFruitsDeMerStyleBrest();
        } else if (choix.equals("poivrons")) {
            return new PizzaPoivronsStyleBrest();
        } else
            return null;
    }
}
```

### C.3.4. Déclarer une méthode de fabrique

Rien qu'en apportant une ou deux transformations à `Pizzeria`, nous sommes passés d'un objet gérant l'instanciation de nos classes concrètes à un ensemble de sous-classes qui assument maintenant cette responsabilité.



#### QUESTION

Quelle est la déclaration exacte de la méthode `creerPizza()` de la classe `Pizzeria` ?

Solution

```
protected abstract Pizza creerPizza(String type);
```



- **protected abstract** : Comme une méthode de fabrication est abstraite, on compte sur les sous-classes pour gérer la création des objets.
- **Pizza** : Une méthode de fabrication retourne un Produit qu'on utilise généralement dans les méthodes définies dans la superclasse.
- **creerPizza**: Une méthode de fabrique isole le client (le code de la superclasse, tel `commanderPizza()`) : elle lui évite de devoir connaître la sorte de Produit concret qui est réellement créée.

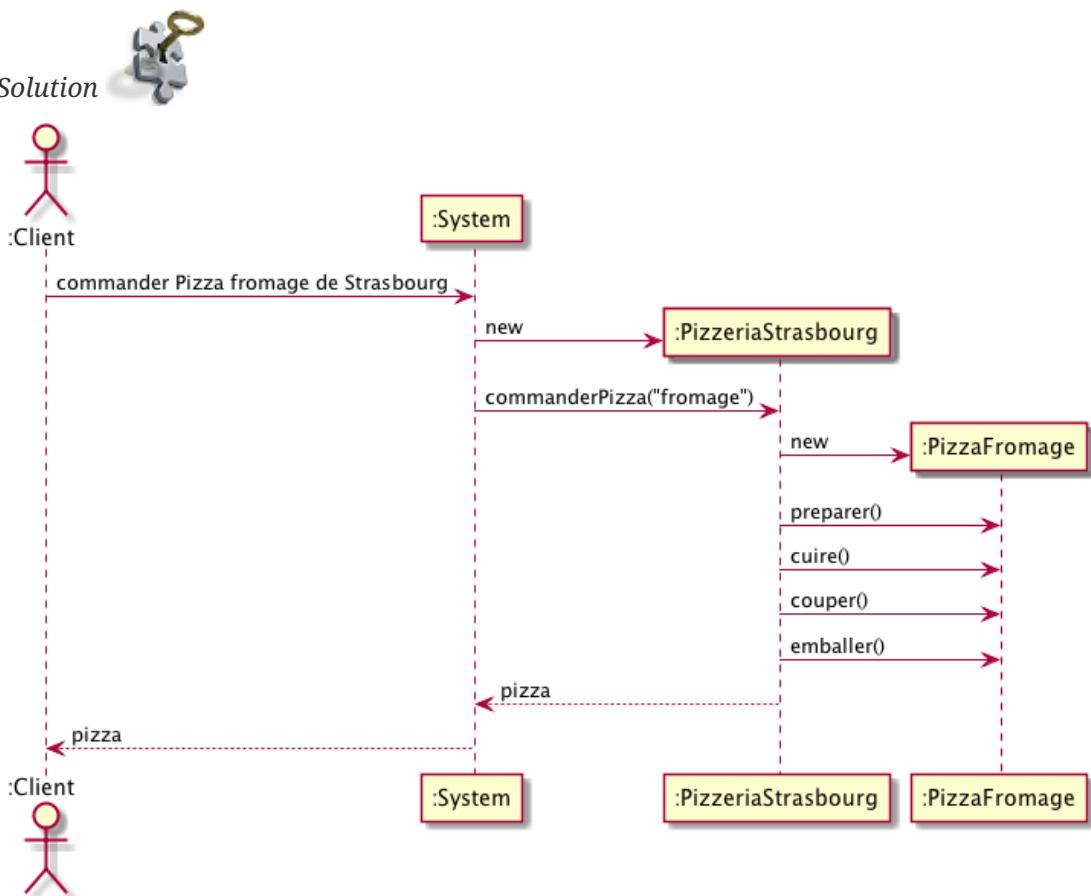
### C.3.5. Récapitulons



#### QUESTION

Donnez le diagramme de séquence d'une "commande de pizza au fromage de type Strasbourg".

Solution



Vous implémenterez les classes manquantes en TP.

## C.4. Le patron Fabrique (simple)

Nous y sommes, vous venez de décortiquer le patron Fabrique Simple

### Design pattern : Fabrique (simple)

**Fabrique** (simple) définit une interface pour la création d'un objet, mais en laissant à des sous-classes le choix des classes à instancier (voir aussi [Fabrique abstraite](#)).

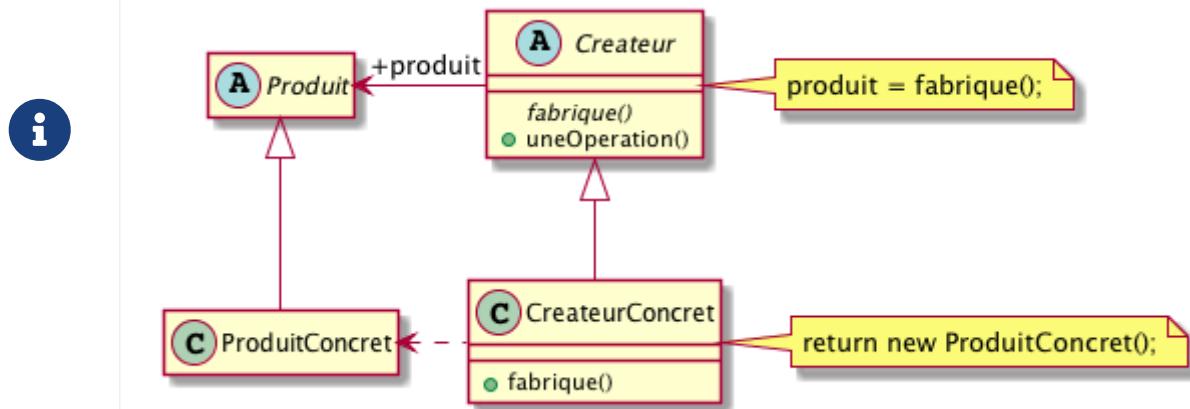


Figure 76. Modèle UML du patron Fabrique

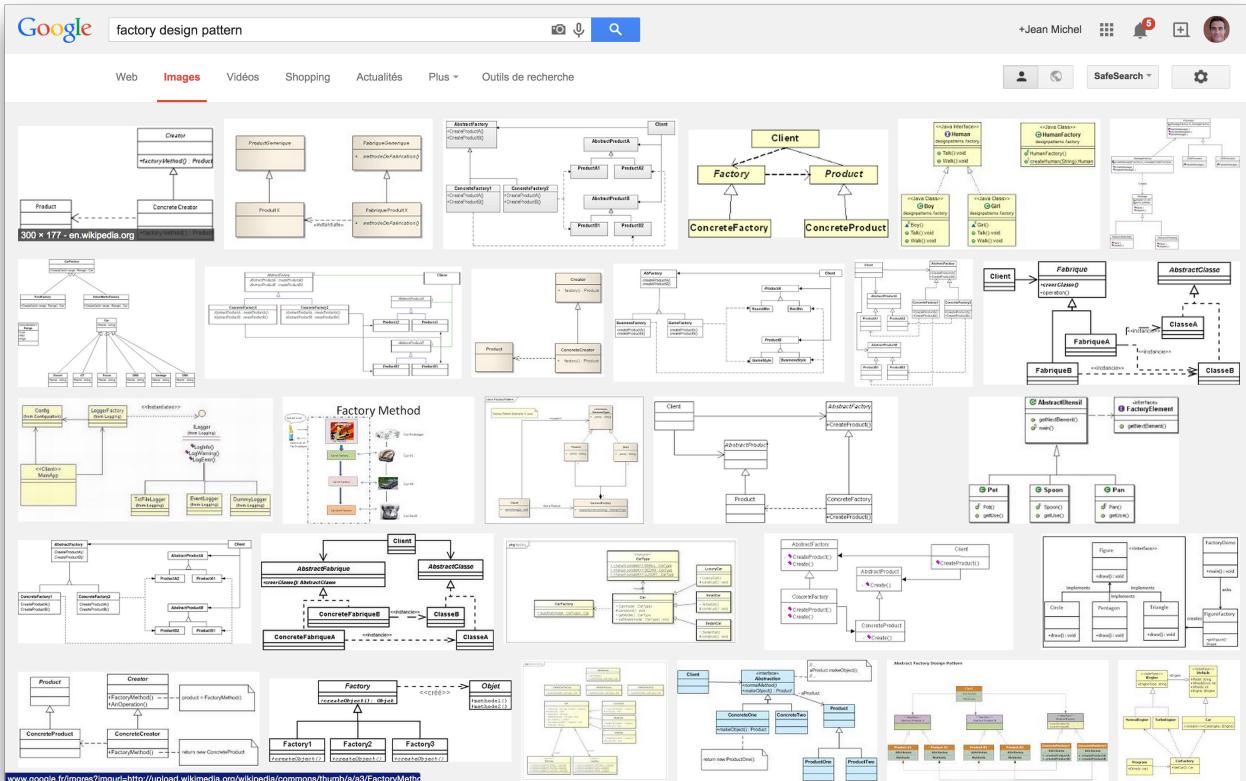


Figure 77. Quelques exemples de description du patron Fabrique

Pour aller plus loin

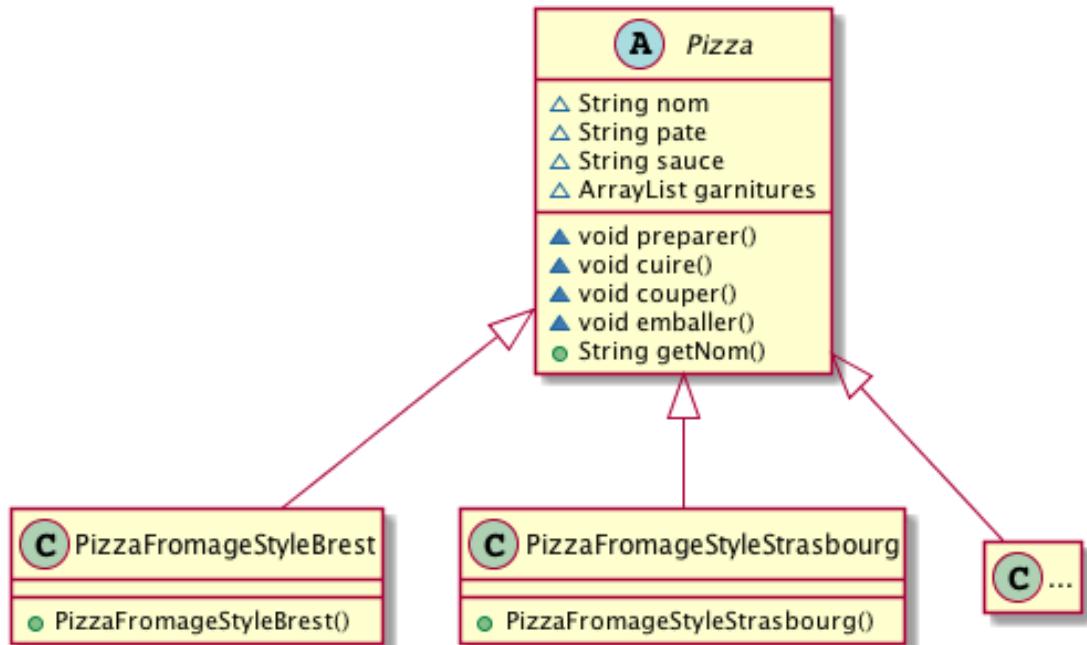
## Et les pizzas dans tout ça !?

### QUESTION



Proposez un diagramme de classe UML pour les pizzas (classes, attributs et méthodes).

Solution



Exemple de code pour **Pizza**:



```
public abstract class Pizza {  
    private String nom;  
    private String pate;  
    private String sauce;  
    private ArrayList <Ingredient> garnitures = new ArrayList<>();  
  
    public void preparer() {  
        System.out.println("Préparation de " + nom);  
        System.out.println("Étalage de la pâte...");  
        System.out.println("Ajout de la sauce...");  
        System.out.println("Ajout des garnitures: ");  
        for (int i = 0; i < garnitures.size(); i++) {  
            System.out.println(" " + garnitures.get(i));  
        }  
    }  
    public void cuire() {  
        System.out.println("Cuisson 25 minutes à 180°");  
    }  
    public void couper() {  
        System.out.println("Découpage en parts triangulaires");  
    }  
    public void emballer() {  
        System.out.println("Emballage dans une boîte officielle");  
    }  
    public String getNom() {  
        return nom;  
    }  
}
```

## Et sans patron, ça donne quoi ?

Un stagiaire de 2013 (les patrons n'étaient pas au programme du PPN!) a réalisé le programme suivant :

```

public class PizzeriaDependante {
    public Pizza creerPizza(String style, String type) {
        Pizza pizza = null;

        if (style.equals("Brest")) {
            if (type.equals("fromage")) {
                pizza = new PizzaFromageStyleBrest();
            } else if (type.equals("vegetarienne")) {
                pizza = new PizzaVegetarienneStyleBrest();
            } else if (type.equals("fruitsDeMer")) {
                pizza = new PizzaFruitsDeMerStyleBrest();
            } else if (type.equals("poivrons")) {
                pizza = new PizzaPoivronsStyleBrest();
            }
        } else if (style.equals("Strasbourg")) {
            if (type.equals("fromage")) {
                pizza = new PizzaFromageStyleStrasbourg();
            } else if (type.equals("vegetarienne")) {
                pizza = new PizzaVegetarienneStyleStrasbourg();
            } else if (type.equals("fruitsDeMer")) {
                pizza = new PizzaFruitsDeMerStyleStrasbourg();
            } else if (type.equals("poivrons")) {
                pizza = new PizzaPoivronsStyleStrasbourg();
            }
        } else {
            System.out.println("Erreur : type de pizza invalide");
            return null;
        }
        pizza.preparer(); pizza.cuire(); pizza.couper(); pizza.emballer();
        return pizza;
    }
}

```

### QUESTION



1. Faites le compte du nombre de classes concrètes dont cette classe dépend.
2. Et si vous ajoutez des pizzas de style Marseille à cette Pizzeria ?

*Solution*



1. 8
2. 12



À comparer aux 2 (fabrique et pizza) des Pizzéries avec Fabrique

## Problème du main de test du jeu d'aventure

Vous avez sûrement dans votre `main` de l'application de jeu d'aventure une partie du code ressemblant à ceci :

*Adaptation des comportements à la situation*

```
if (choix.equals("Epee")) {  
    perso.setArme(new ComportementEpee());  
}  
else if (choix.equals("Arc")) {  
    perso.setArme(new ComportementArc());  
else if ...  
...  
}
```

Ce code est peu adaptatif et va souffrir des évolutions, par exemple :

- changement de la liste des armes possibles
- rajouter des `if then else` à chaque nouvelle arme
- suppression de certaines armes
- ...

### QUESTION

1. Isoler ce code dans une classe `SimpleFabriqueArme` qui possèdera une méthode `creerComportementArme(String type)` qui retourne le comportement adapté en fonction du paramètre reçu.
2. Donnez le diagramme de classe `UML™` de la nouvelle organisation.
3. Quelles différences a-t-on avec le patron Fabrique ? Donner quelques exemples d'extension du jeu d'aventure qui correspondraient à l'usage d'une Fabrique.
4. Donnez le diagramme de séquence du main. Par exemple avec le code de test suivant :



```
Chevalier perso = new Chevalier("JMI");  
  
SimpleFabriqueArme fabrique = new SimpleFabriqueArme();  
ComportementArme c = fabrique.creerComportementArme("Epee");  
  
perso.setArme(c);  
perso.frapper();
```



*Solution*



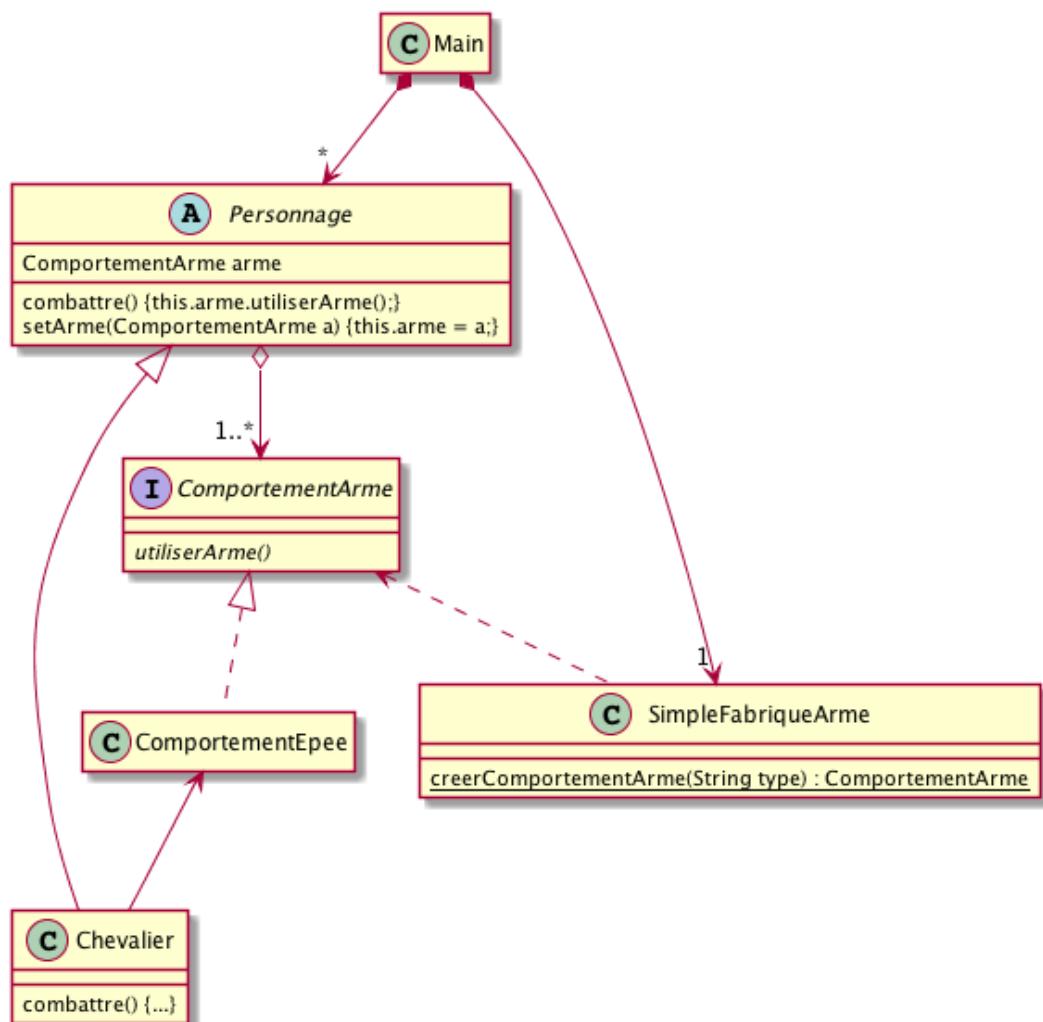
1. Implémentation

```

public class SimpleFabriqueArme {
    public ComportementArme creerComportementArme(String type) {
        ComportementArme compAdequat = null;
        if (type.equals("Epee")) {
            compAdequat = new ComportementEpee();
        }
        else if (type.equals("Arc")) {
            compAdequat = new ComportementArc();
        }
        else compAdequat = new ComportementArmeless();
        return compAdequat;
    }
}

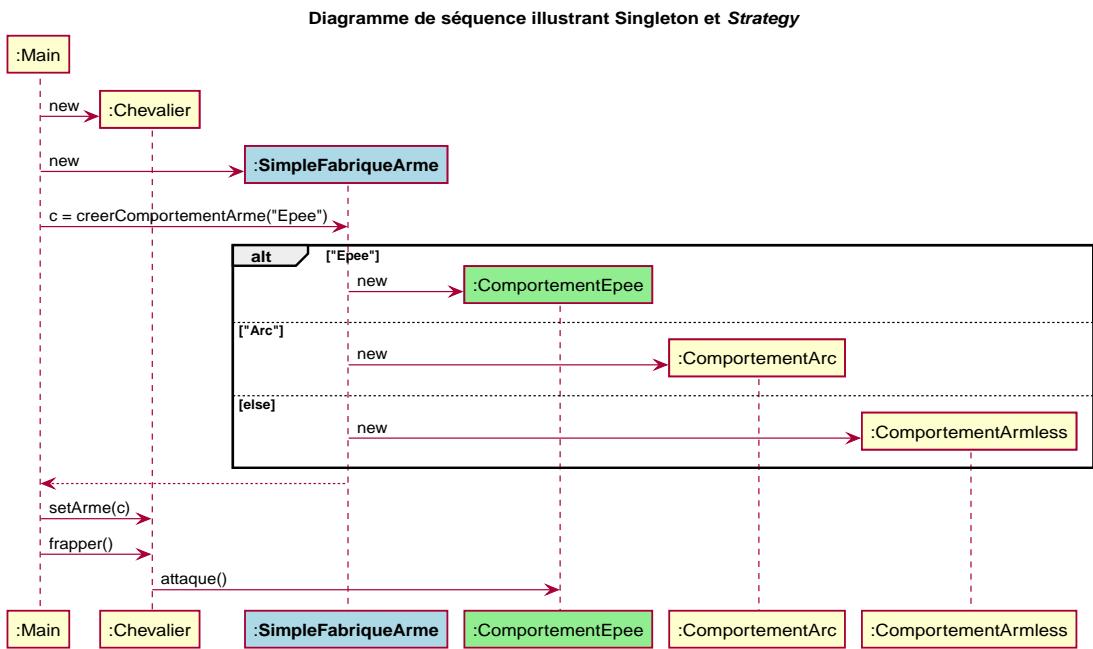
```

2. Diagramme de classe de la fabrique de comportements d'armes



3. Le créateur et le créateur concret sont dans une unique classe (**SimpleFabriqueArme**) car on a une seule fabrique unique pour le moment. On peut imaginer deux modes de jeu, avec des armes médiévales (épée en métal) ou modernes (épée sabre laser) qui seront traités dans deux fabriques concrètes différentes "FabriqueArmeMedievales" et "FabriqueArmeModernes".

#### 4. Diagramme de séquence du main.



## Appendix D: CPOA - Support TD 4



*Version corrigée*



Cette version comporte des indications pour les réponses aux exercices.

PreReq	<ol style="list-style-type: none"> <li>1. Je sais programmer en Java.</li> <li>2. J'ai conscience qu'il faut réfléchir avant de se lancer dans le codage.</li> <li>3. Je maîtrise patrons de conception.</li> <li>4. Je maîtrise les diagrammes UML de classe, de séquence et d'états</li> </ol>
ObjTD	Aborder quelques subtilités UML.
Durée	1 TD

### D.1. Rappel du cours



N'hésitez pas à (re)lire régulièrement le [Support de Cours](#).

### D.2. Différences entre dépendance, association, composition, agrégation

Soit le diagramme de classe partiel suivant :

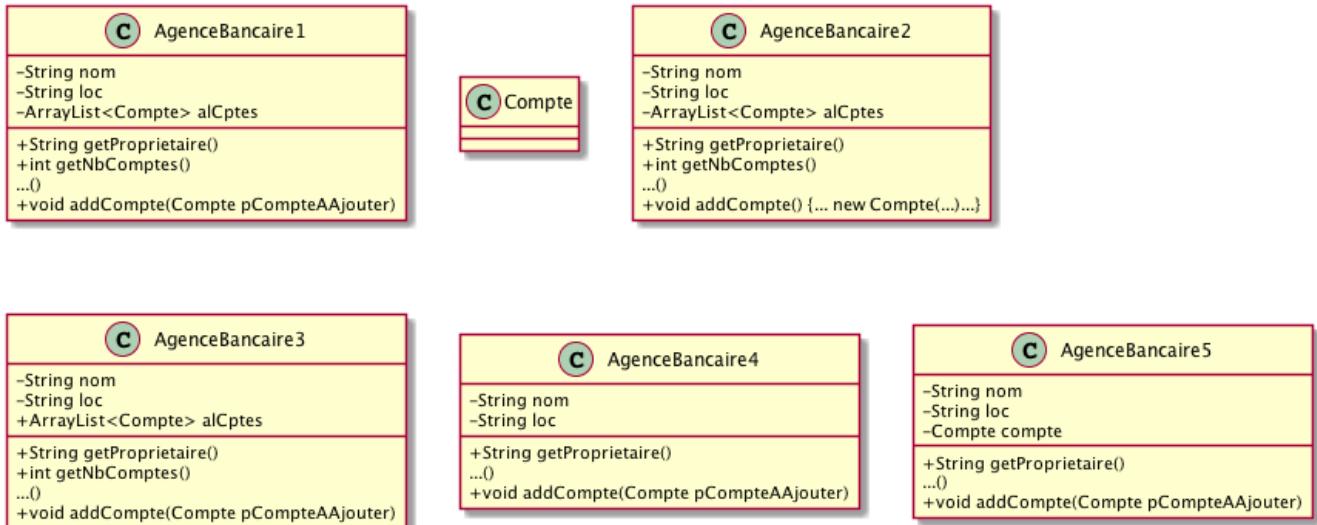


Figure 78. Diagramme de classe partiel

### QUESTION



Complétez en ajoutant les relations (dépendance, association, composition, agrégation) entre les classes.

### Solution

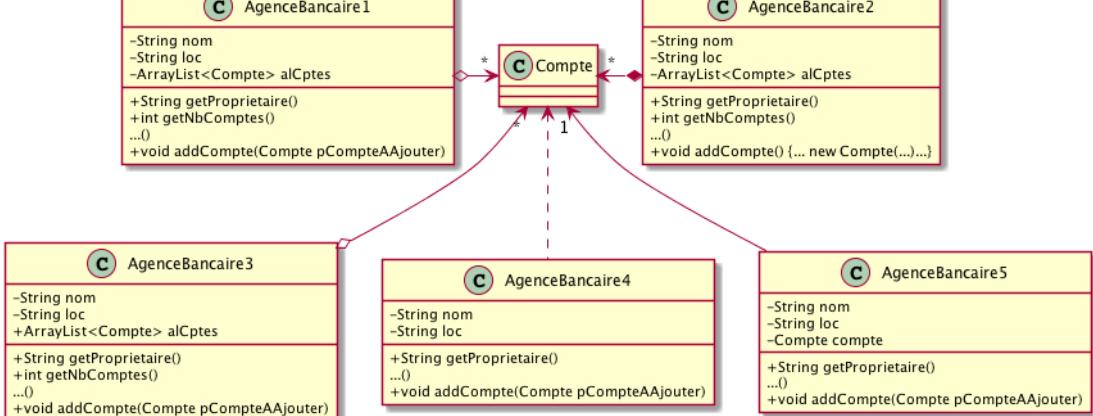


Figure 79. Diagramme de classe avec relations

## D.3. Patrons

## QUESTION

Pour chacun des diagrammes de classe partiels suivants (représentant des patrons que vous connaissez), complétez :

- le nom du patron dans la légende,
- en ajoutant les relations.

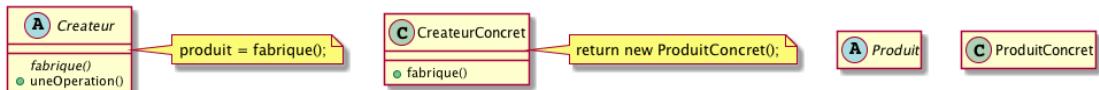


Figure 80. Patron ...

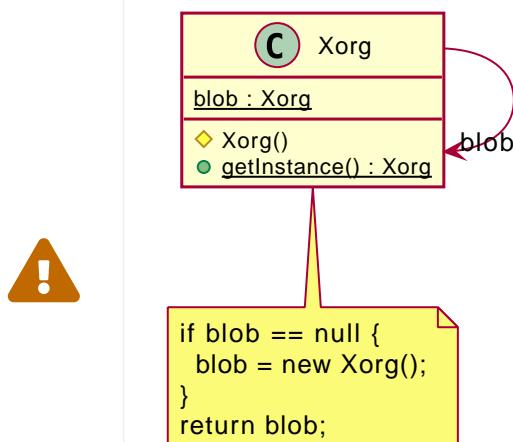


Figure 81. Patron ...

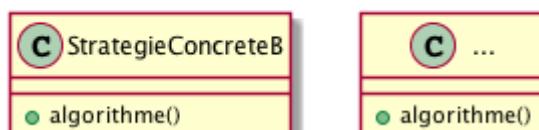
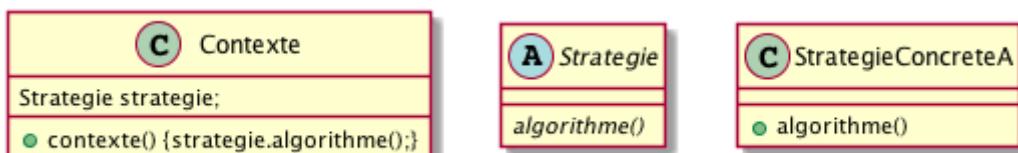


Figure 82. Patron ...



Solution



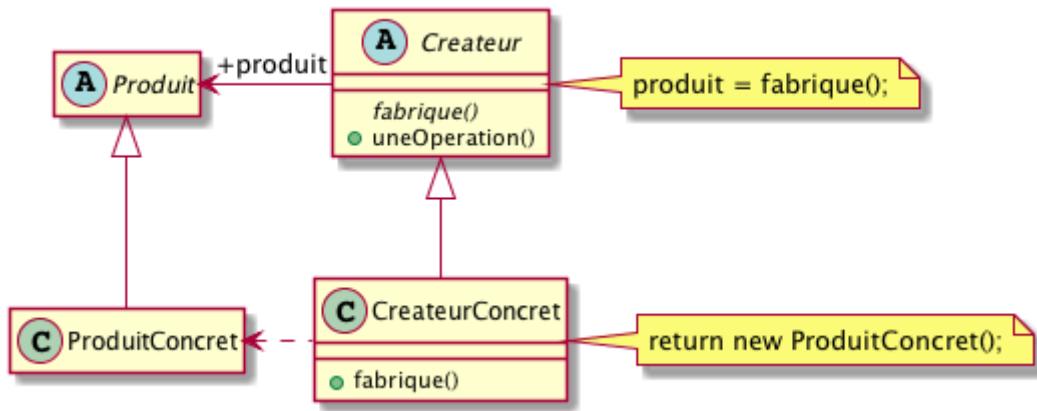


Figure 83. Patron Fabrique (simple)

Notez l'équivalence UML™ (ici noté en plantUML) entre :



- Produit "+produit" <- Createur
- Produit "1" <- Createur {+Produit produit;}

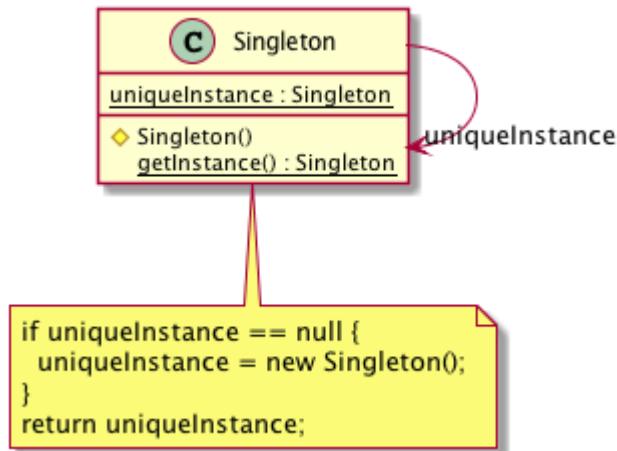


Figure 84. Patron Singleton

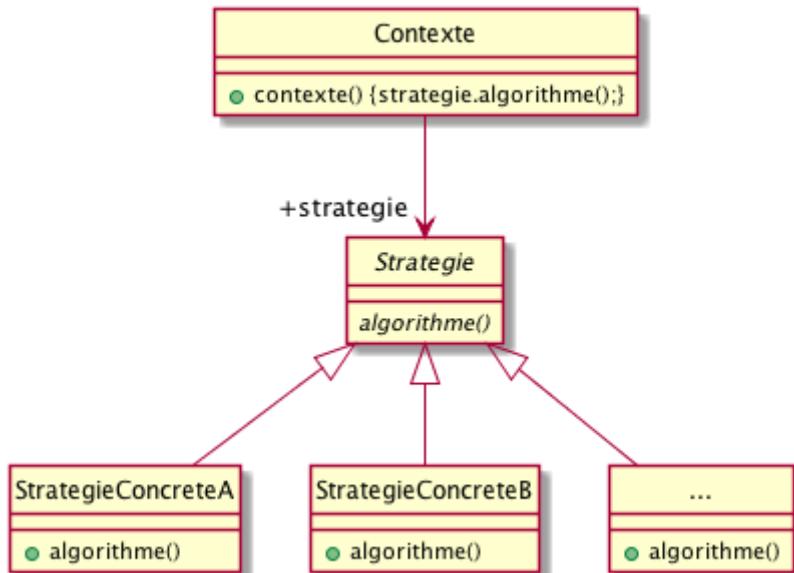


Figure 85. Patron Strategy

## D.4. Diagrammes de séquences

Vous devez documenter, à partir des extraits de codes Java suivants, l'application **ApplicationBanque**, développée en S2.



Vous refactorerez cette application en TP, l'objectif n'est donc pas pour l'instant de remédier aux problèmes de conception mais plutôt de les identifier.

Méthode statique **comptesDUnProprietaire** (**ApplicationAgenceBancaire.java**)

```
public static void comptesDUnProprietaire (AgenceBancaire ag, String nomProprietaire) {  
    Compte [] t;  
  
    t = ag.getComptesDe(nomProprietaire);  
    if (t.length == 0) {  
        System.out.println("pas de compte à ce nom ...");  
    } else {  
        System.out.println(" " + t.length + " comptes pour " + nomProprietaire);  
        for (int i=0; i<t.length; i++)  
            t[i].afficher();  
    }  
}
```



### QUESTION

Réalisez un diagramme de séquence illustrant le fonctionnement de cette méthode.

*Solution*

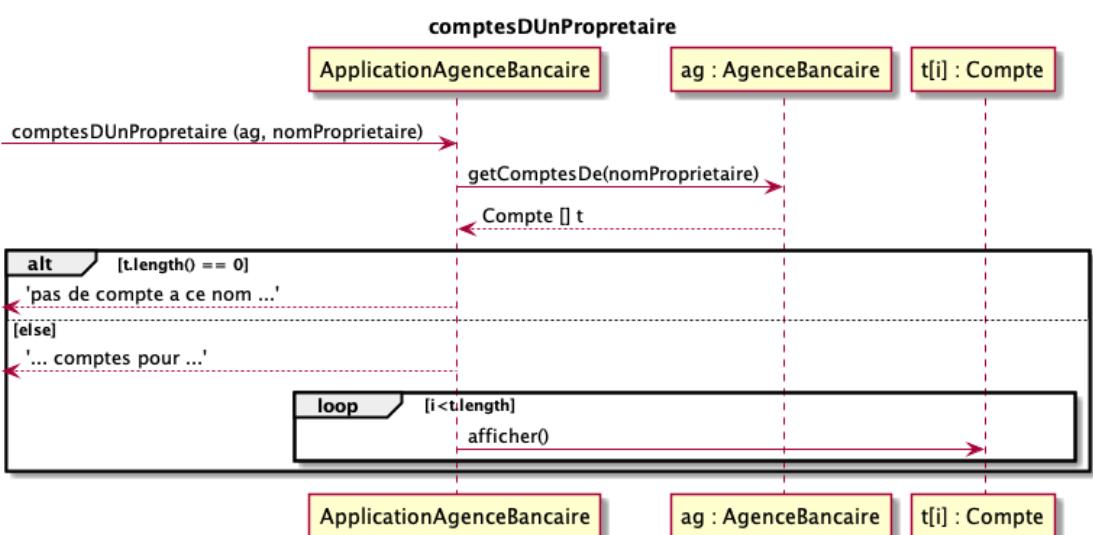


Figure 86. Diagramme de séquence de la méthode **comptesDUnProprietaire**

**ApplicationAgenceBancaire.java**

```

public class ApplicationAgenceBancaire {

    public static void main(String argv[]) {

        String choix;

        boolean continuer ;
        Scanner lect;
        AgenceBancaire monAg ;

        String nom, numero;
        Compte c;
        double montant;

        lect = new Scanner ( System.in );
        lect.useLocale(Locale.US);

        monAg = AccesAgenceBancaire.getAgenceBancaire();

        continuer = true;
        while (continuer) {

            ...
            choix = lect.next();
            choix = choix.toLowerCase();
            switch (choix) {
                case "q" :
                    System.out.println("ByeBye");
                    continuer = false;
                    break;
                case "l" :
                    monAg.afficher();
                    break;
                case "v" :
                    System.out.print("Num compte -> ");
                    numero = lect.next();
                    c = monAg.getCompte(numero);
                    if (c==null) {
                        System.out.println("Compte inexistant ...");
                    } else {
                        c.afficher();
                    }
                    break;
                case "p" :
                    System.out.print("Propriétaire -> ");
                    nom = lect.next();
                    ApplicationAgenceBancaire.comptesDUnPropretaire (monAg, nom);
                    break;
                case "d" :

                    ...
                    break;
                case "r" :

```

```

        ...
        break;
    default :
        ...
        break;
    }
}
}

public static void comptesDUnPropretaire (AgenceBancaire ag,
String nomProprietaire) {...}

public static void deposerSurUnCompte (AgenceBancaire ag,
String numeroCompte, double montant) {...}

public static void retirerSurUnCompte (AgenceBancaire ag,
String numeroCompte, double montant) {...}
}

```

*Extrait de AccesAgenceBancaire*

```

public class AccesAgenceBancaire {

    private AccesAgenceBancaire () {}
    public static AgenceBancaire getAgenceBancaire () {

        AgenceBancaire ag = new AgenceBancaire("CAISSE ECUREUIL", "PIBRAC");
        ...
    }
    ...
}

```

### QUESTION



1. Réalisez le diagramme de classe de l'application
2. Que vous rappelle la classe **AccesAgenceBancaire**?
3. Réalisez un diagramme de séquence illustrant le fonctionnement de cette application (**main**). On utilisera des blocs "ref" pour les appels aux méthodes statiques, et on ne s'occupera pas des scanners.



*Solution*



1. Diagramme de classe

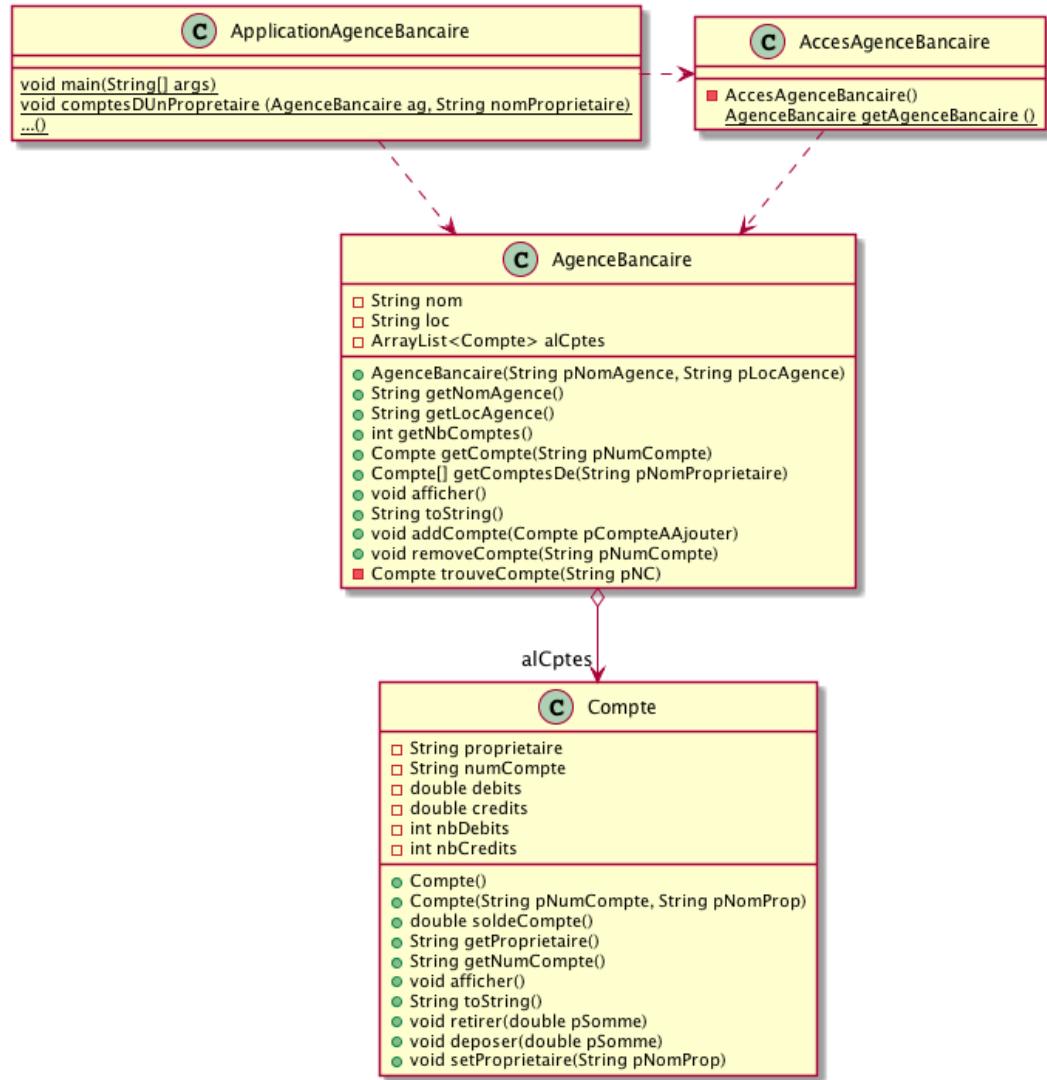


Figure 87. Diagramme de classe



Les étudiants ne peuvent avoir tous ces détails, mais ça leur servira pour le TP.

2. Cette s'approche de deux patrons que vous avez étudiés:
  - la structure du Singleton (`getInstance()` remplacé par `getAgenceBancaire ()`), sans la gestion de l'instance unique,
  - le comportement du créateur concrète de la Factory.
3. Diagramme de séquence

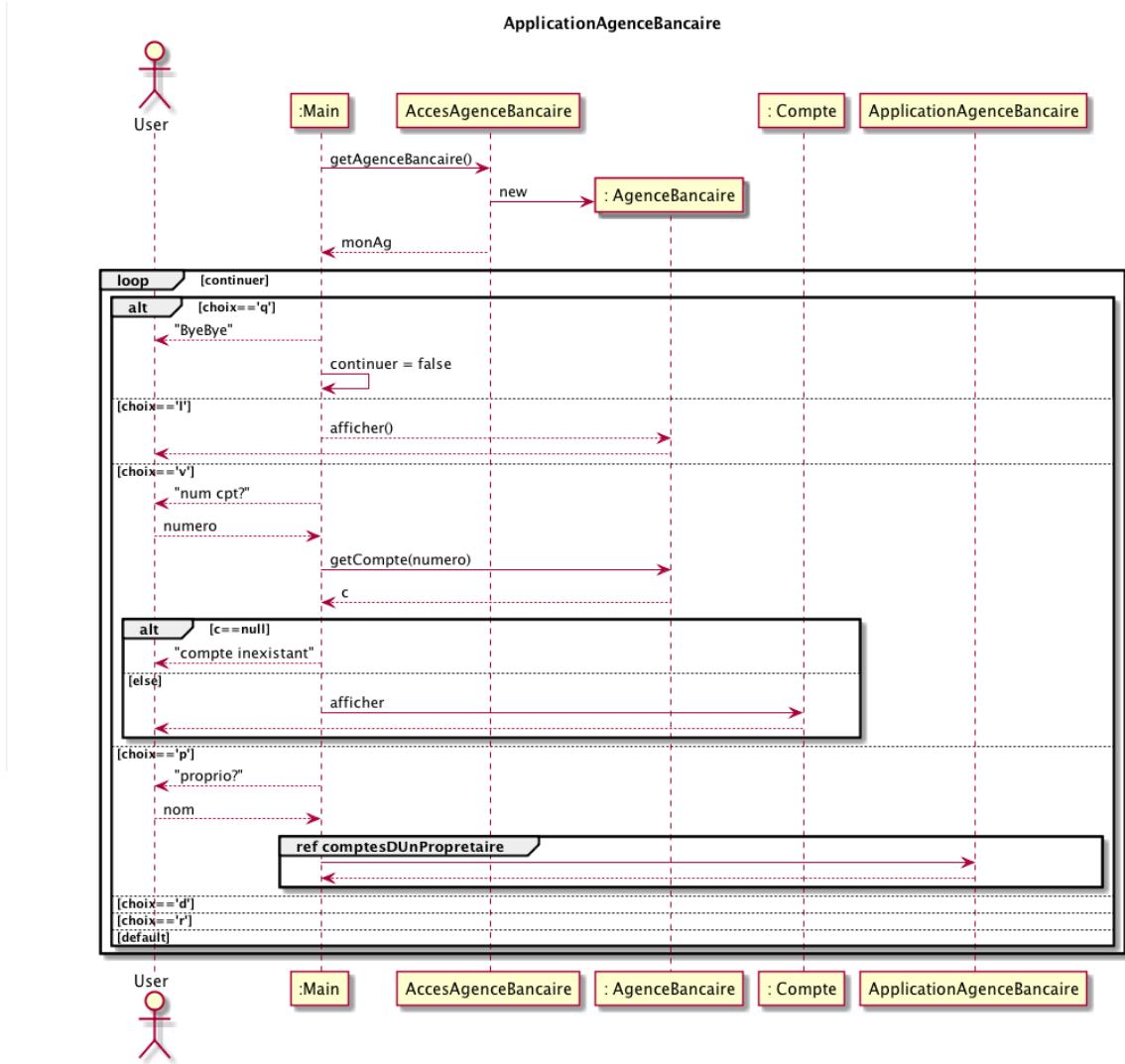


Figure 88. Diagramme de séquence de `ApplicationAgenceBancaire`

## D.5. Machines à état

### QUESTION



1. Dessinez le diagramme d'états correspondant aux feux tricolores (en France) classiques. Ajoutez la prise en compte de la panne dans un 2ème temps.
2. Dessinez le diagramme d'états correspondant au déroulement d'une partie d'échecs.

*Solution*



1. Feu tricolore



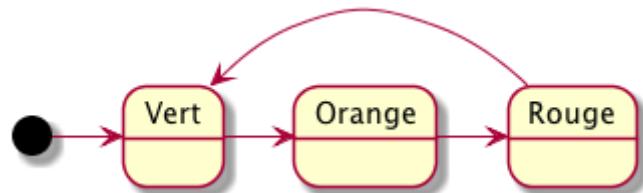


Figure 89. Diagramme d'état d'un feu tricolore classique

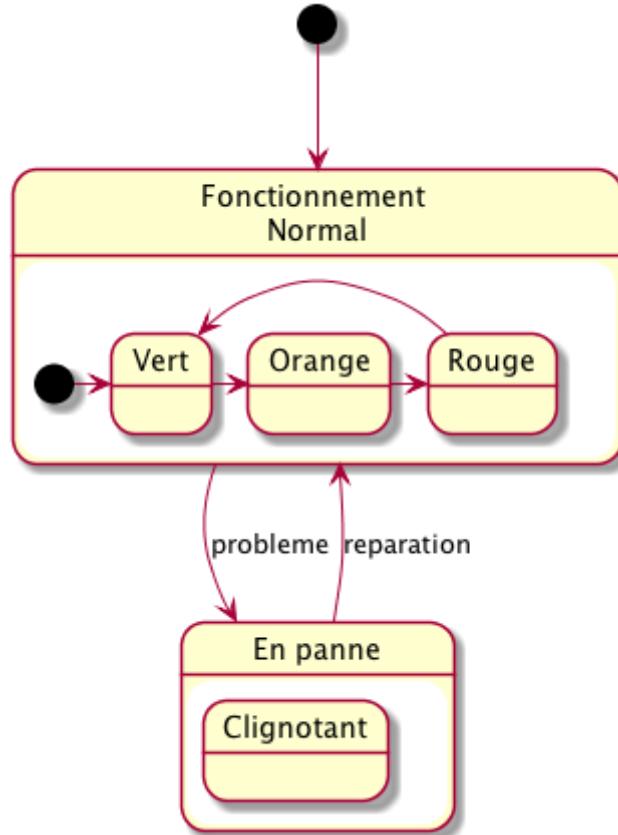


Figure 90. Diagramme d'état d'un feu tricolore avec panne

## 2. Echecs

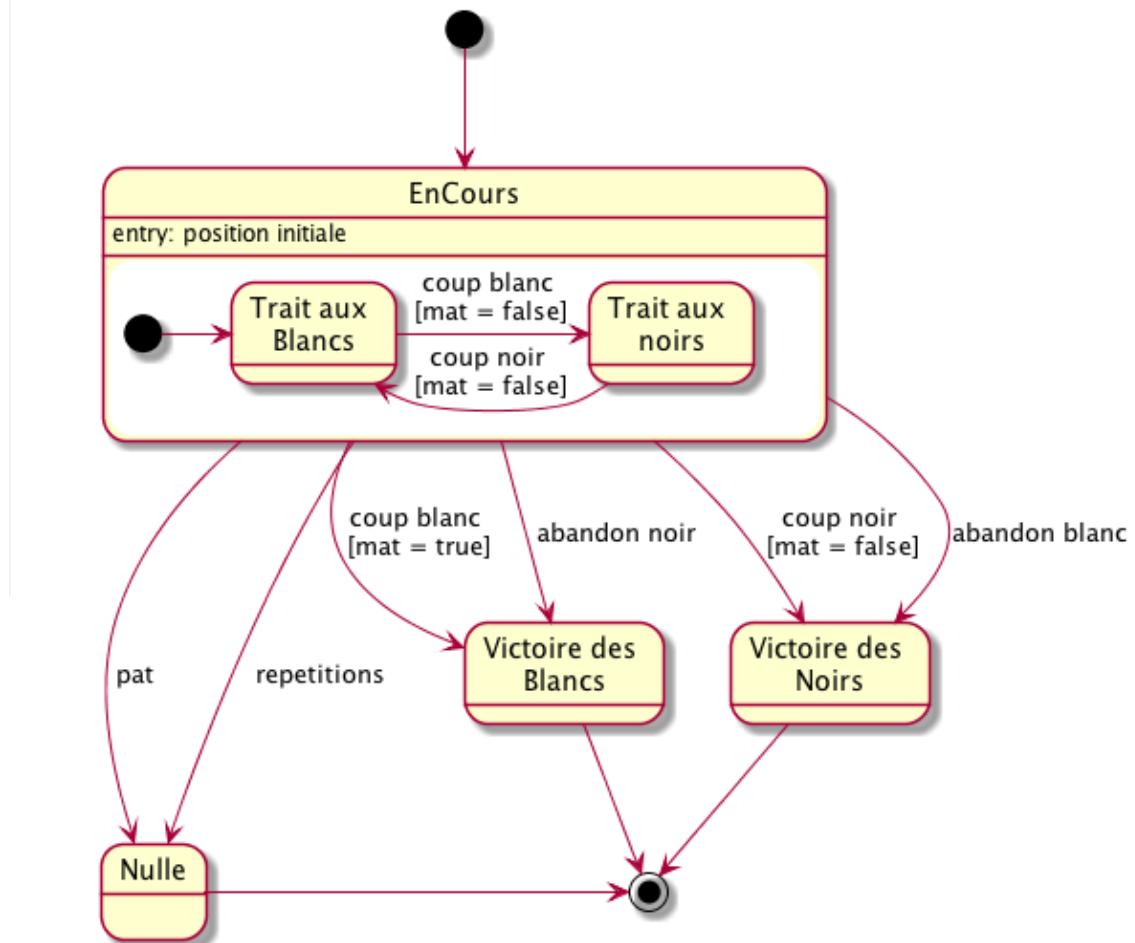


Figure 91. Diagramme d'état d'une partie d'échec

## Pour aller plus loin



### QUESTION

1. Peut-on, dans un code Java, faire la différence entre agrégation `1 <-> *` et association `1 -> *?`



### Solution

- En Java, un objet est toujours une référence, il n'y aura pas de changement de comportement de la JVM pour l'agrégation et la composition. Cependant, le modèle des classes et leur cycle de vie ne sera pas le même dans les deux cas:

- Cas de la composition

```
public class Voiture {  
    ArrayList<ComposantVoiture> mesParts;  
    ...  
    public Voiture (){  
        mesParts.add(new ComposantVoiture("essieu"));  
    }  
    public void afficher () {  
        mesParts.each...  
        ...  
        afficher();  
    }  
    public void detruire(){  
        mesParts.each...  
        ...  
        detruire();  
    }  
}
```



- Cas de l'agrégation

```
public class Voiture {  
    ArrayList<Passager> passagers;  
    public void ajouterPassager(Passager p) { passagers.add(p);  
    }  
    public void detruire(){  
        ...  
    }  
}
```

## Appendix E: CPOA - Support TD 5



### Version corrigée

Cette version comporte des indications pour les réponses aux exercices.

PreReq	1. Je sais programmer en <a href="#">Java</a> . 2. J'ai conscience qu'il faut réfléchir avant de se lancer dans le codage. 3. Je maîtrise quelques patrons de conception.
ObjTD	Aborder le patron <b>Observer</b> .
Durée	<b>1 TD et 2 TPs</b>

## E.1. Rappel du cours



N'hésitez pas à (re)lire régulièrement le [Support de Cours](#).

## E.2. Motivation

Dans cet exercice, nous allons explorer comment on peut maintenir la cohérence entre plusieurs objets qui sont liés sans pour autant maintenir d'association fortes entre ces objets.

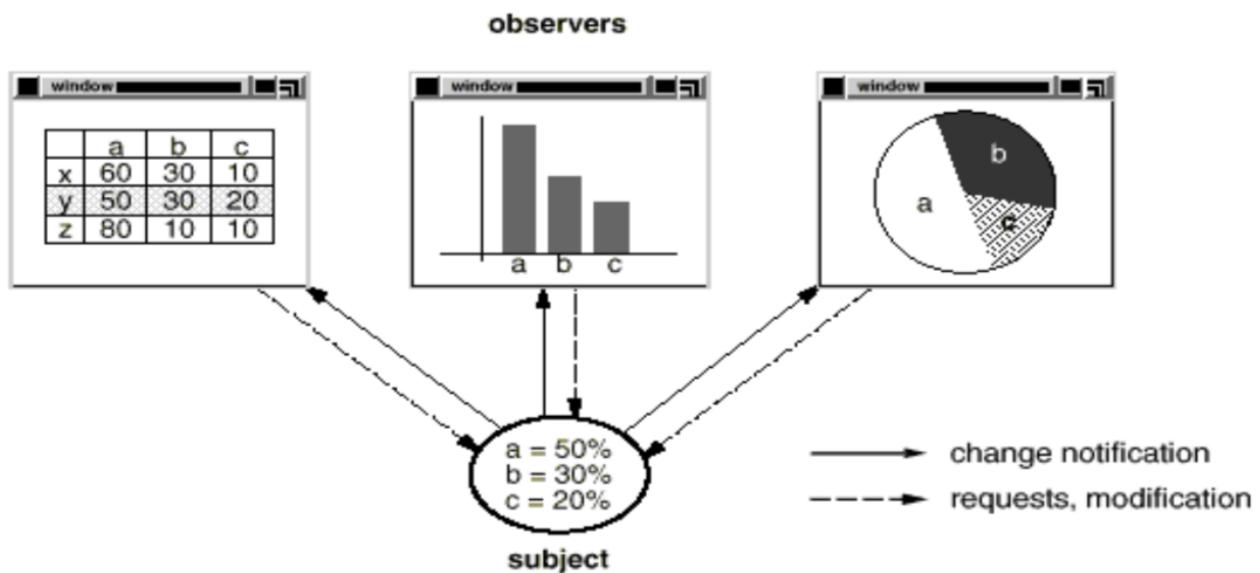


Figure 92. L'illustration classique du patron Observer

## E.3. Le patron Observer

### E.3.1. Définition

**Observateur** définit une relation entre objets de type un-à-plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

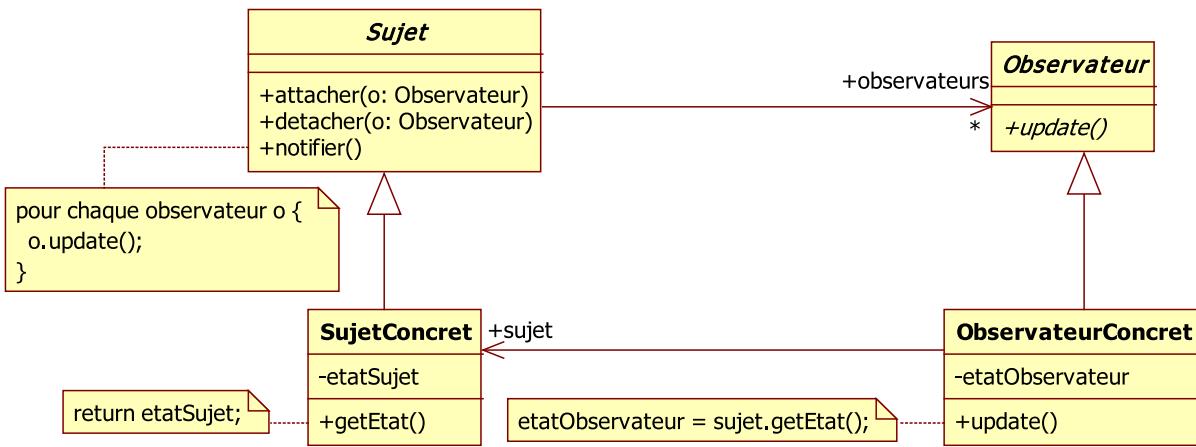


Figure 93. Modèle UML du patron Observer

Principe:

- les observateurs s'enregistrent auprès d'un sujet
- ce sujet est chargé de prévenir les observateurs (d'un changement).

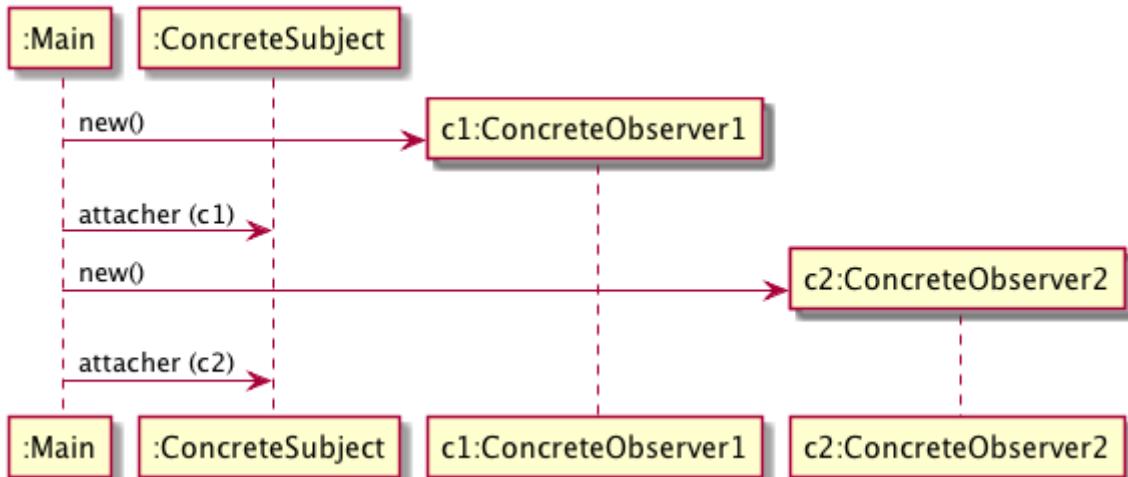


Figure 94. Exemple d'utilisation

On peut réaliser de plusieurs façon le `attacher`, mais il doit être fait pour que ça marche :

- par un objet extérieur (ici le main de l'application),
- par le `concreteObserver` lui-même. Cela reste rare, car le principe est que le sujet n'a pas connaissance de ses observateurs.

### E.3.2. Objectif

- Définir une dépendance un-plusieurs entre un objet observé et des objets observateurs de telle façon que quand l'observé évolue, ces observateurs sont informés et mis à jour automatiquement.

### E.3.3. Application

Le patron *Observer* est utilisable dans de nombreuses situations :

- Quand un concept a deux aspects, l'un dépendant de l'autre. Encapsuler ces aspects dans des objets séparés permet de les utiliser et les laisser évoluer de manière indépendante.
- Dès que le changement d'un objet entraîne le changement de plusieurs autres.
- Dès qu'un objet doit en notifier un certain nombre d'autres sans les connaître.

### E.3.4. Participants

- **Subject**
  - Garde trace de ses **observers**
  - Fournit une interface pour attacher et détacher les objets **Observer**
- **Observer**
  - Définit une interface pour les notifications **update**
- **ConcreteSubject**
  - Les objets observés.
  - Gère les objets **ConcreteObserver** intéressés.
  - Envoie une notification à ses observateurs quand il change d'état
- **ConcreteObserver**
  - Les objets observateurs.
  - Gère la cohérence avec l'état de l'observé.
  - Implémente l'interface **update()** d'`Observer`.

### E.3.5. Scenario



#### QUESTION

Pour vérifiez que vous avez compris son utilité, réalisez un diagramme de séquence illustrant l'utilisation du patron *Observer*. Commencez par l'appel d'une méthode `setState()` sur l'observé (le sujet).

### Solution

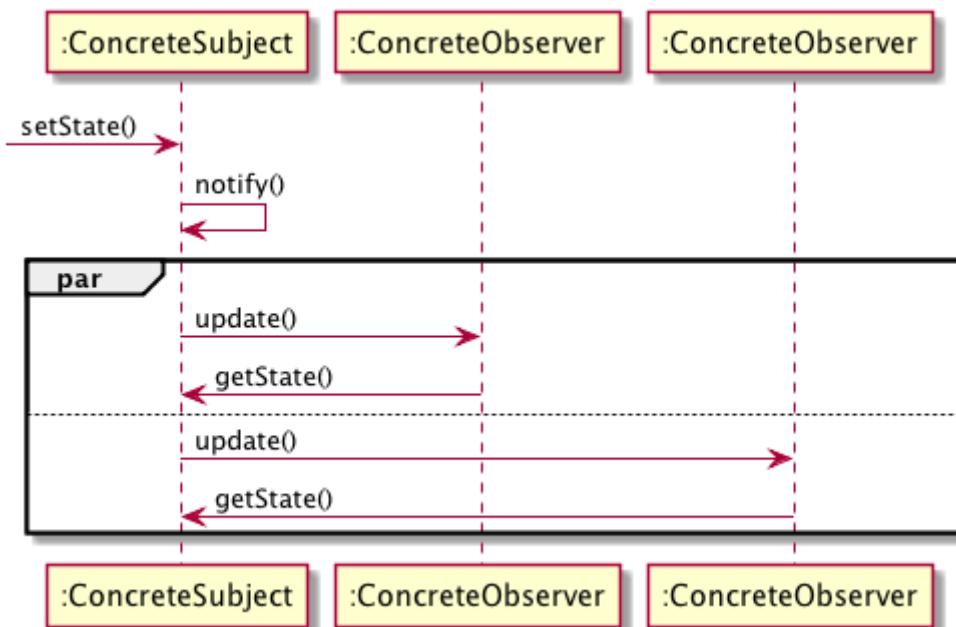


Figure 95. Exemple d'utilisation

### E.3.6. Bénéfices

- Couplage minimal entre **Subject** et **Observer**
  - On peut utiliser les sujets sans se soucier des observateurs et vice-versa
  - Les observateurs peuvent être ajoutés sans modifier nécessairement le sujet
  - Tout ce que le sujet connaît de ses observateurs, c'est leur liste
  - Le sujet n'a pas connaissance de la classe concrète des observateurs, juste qu'ils implémentent l'interface **update**
  - Sujet et observateur peuvent appartenir à des couches d'abstraction différentes
- Support à la diffusion d'événements
  - Le sujet envoie des notifications à tous ses observateurs
  - Les observateurs peuvent être ajoutés/retirés à la volée

### E.3.7. Limites

- Possibilité d'effets en cascade
  - Les observateurs ne sont pas nécessairement au courant les uns des autres. Il faut donc être prudent sur les effets des **update**
- Implication pour observateurs de déduire les changements à partir d'un simple **update**.

## E.4. Implémentation

## E.4.1. Problèmes

Nous allons maintenant nous intéresser à l'implémentation.



Si vous connaissez l'implémentation [Observer Java](#), essayez de l'oublier!

### QUESTION

Voici une liste des éléments, liés à l'implémentation, à résoudre. Pour chacune, essayez de fournir des éléments de réponse :



1. L'un est une interface, l'autre une classe. Serez-vous retrouver la nature de Observer et Observable ? Pourquoi ?
2. Comment le sujet garde la trace de ses observateurs ?
3. Et si un observateur veut observer plus d'un sujet ?
4. Qui déclenche les [update](#) ?
5. Comment s'assurer que les sujets mettent bien à jour leur état avant de notifier leur changement.
6. Quel niveau de détail au sujet de son changement doit transmettre le sujet à ses observateurs ?
7. Est-ce que les observateurs peuvent souscrire à des événements spécifiques ?
8. Est-ce qu'un observateur peut lui-même être un sujet ?
9. Peut-on faire en sorte qu'un observateur soit notifié qu'après qu'un certain nombre de sujets aient changé d'état ?

### *Solution*

#### 1. Association:

- `java.util.Observer` est une interface qui déclare uniquement la méthode `update(...)`
- `java.util.Observable` est une classe qui implémente:
  - `addObserver / delObserver`
  - `notifyObserver / setChanged`
  - ...

#### 2. Dans un attribut de type Array, liste chaînée (`ArrayList`), ...

#### 3. Le sujet peut s'identifier auprès des observateurs via l'interface `update`

#### 4. Par exemple :

- Le sujet quand il change d'état
- Les observateurs après qu'ils ait provoqué un ou plusieurs changements
- Des objets quelconques!

#### 5. En Java, seul l'objet lui-même peut "initier" une notification à ses observateurs grâce à une variable d'état à visibilité "protected". Il faudra s'assurer au sein de la classe concrète `Observable` que la séquence est respectée sous peine de ne pas avoir de notification.

#### 6. Par exemple :

- Push Model ⇒ Beaucoup
- Pull Model ⇒ Le moins possible

#### 7. On parle alors de *publish-subscribe*

#### 8. Oui ! Il héritera de `Observable` et implémentera `Observer`.

#### 9. Par exemple :

- Utiliser un objet intermédiaire qui agit comme médiateur

## E.4.2. *Observer en Java*

Java fournit des classes *Observable/Observer* pour le patron *Observer*. La classe `java.util.Observable` est la classe de base pour les sujets. Ainsi, toute classe qui veut être observée étant cette classe dont voici les caractéristiques :

- fournit des méthodes pour ajouter/enlever des observateurs
- fournit des méthodes pour notifier les observateurs
- une sous-classe concrète doit seulement s'occuper de notifier à chaque méthode modifiant l'état des objets (*mutators*)
- utilise un vecteur stockant les références des observateurs

L'interface `java.util.Observer` correspond aux observateurs qui doivent implémenter cette

interface.

## La classe `java.util.Observable`

### QUESTION

Voici la liste des méthodes de `java.util.Observable` :

```
public Observable()
public synchronized void addObserver(Observer o)
protected synchronized void setChanged()
public synchronized void deleteObserver(Observer o)
protected synchronized void clearChanged()
public synchronized boolean hasChanged()
public void notifyObservers(Object arg)
public void notifyObservers()
```

Retrouver les commentaires correspondants :



1. Adds an observer to the set of observers of this object
2. If this object has changed, as indicated by the `hasChanged()` method, then notify all of its observers and then call the `clearChanged()` method to indicate that this object has no longer changed. Each observer has its `update()` method called with two arguments: this observable object and the `arg` argument. The `arg` argument can be used to indicate which attribute of the observable object has changed.
3. Indicates that this object has changed
4. Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change. This method is called automatically by `notifyObservers()`.
5. Same as above, but the `arg` argument is set to null. That is, the observer is given no indication what attribute of the observable object has changed.
6. Tests if this object has changed. Returns true if `setChanged()` has been called more recently than `clearChanged()` on this object; false otherwise.
7. Construct an `Observable` with zero observers
8. Deletes an observer from the set of observers of this object

### Solution

```
public Observable()
// => Construct an 'Observable' with zero Observers
public synchronized void addObserver(Observer o)::
// => Adds an observer to the set of observers of this object
protected synchronized void setChanged()
// => Indicates that this object has changed
public synchronized void deleteObserver(Observer o)
// => Deletes an observer from the set of observers of this object
protected synchronized void clearChanged()
/* => Indicates that this object has no longer changed, or that it has
already
notified all of its observers of its most recent change. This method is
called
automatically by notifyObservers(). */
public synchronized boolean hasChanged()
/* => Tests if this object has changed. Returns true if setChanged()
has been
called more recently than clearChanged() on this object; false
otherwise. */
public void notifyObservers(Object arg)
/* => If this object has changed, as indicated by the hasChanged()
method, then
notify all of its observers and then call the clearChanged() method to
indicate that this object has no longer changed. Each observer has its
update() method called with two arguments: this observable object and
the
arg argument. The arg argument can be used to indicate which attribute
of
the observable object has changed. */
public void notifyObservers()
/* => Same as above, but the arg argument is set to null. That is, the
observer
is given no indication what attribute of the observable object has
changed. */
```

### QUESTION



Que doit-on rajouter pour adapter le diagramme de classe fait précédemment à l'implémentation Java de Observer/Observable ?

### Solution



- appeler la méthode *hasChanged()* dans *setState()*
- + renommer *notify* en *notifyObserver*

## L'interface java.util.Observer

*java.util.Observer*

```
/**  
 * This method is called whenever the observed object is changed. An  
 * application calls an observable object's notifyObservers method to have all  
 * the object's observers notified of the change.  
 *  
 * Parameters:  
 * o - the observable object  
 * arg - an argument passed to the notifyObservers method  
 */  
public abstract void update(Observable o, Object arg)
```

#### E.4.3. Observable/Observer par l'exemple

*ConcreteSubject.java*

```
/**  
 * A subject to observe!  
 */  
public class ConcreteSubject extends Observable {  
  
    private String name;  
    private float price;  
    public ConcreteSubject(String name, float price) {  
        this.name = name;  
        this.price = price;  
        System.out.println("ConcreteSubject created: " + name + " at " + price);  
    }  
    public String getName() {return name;}  
    public float getPrice() {return price;}  
    public void setName(String name) {  
        this.name = name;  
        setChanged();  
        notifyObservers(name);  
    }  
    public void setPrice(float price) {  
        this.price = price;  
        setChanged();  
        notifyObservers(new Float(price));  
    }  
}
```

### *NameObserver.java*

```
// An observer of name changes.  
public class NameObserver implements Observer {  
    private String name;  
    public NameObserver() {  
        name = null;  
        System.out.println("NameObserver created: Name is " + name);  
    }  
    public void update(Observable obj, Object arg) {  
        if (arg instanceof String) {  
            name = (String)arg;  
            System.out.println("NameObserver: Name changed to " + name);  
        } else {  
            System.out.println("NameObserver: Some other change to  
subject!");  
        }  
    }  
}
```

### *PriceObserver.java*

```
// An observer of price changes.  
public class PriceObserver implements Observer {  
    private float price;  
    public PriceObserver() {  
        price = 0;  
        System.out.println("PriceObserver created: Price is " + price);  
    }  
    public void update(Observable obj, Object arg) {  
        if (arg instanceof Float) {  
            price = ((Float)arg).floatValue();  
            System.out.println("PriceObserver: Price changed to " +  
price);  
        } else {  
            System.out.println("PriceObserver: Some other change to  
subject!");  
        }  
    }  
}
```

### TestObservers.java

```
// Test program for ConcreteSubject, NameObserver and PriceObserver
public class TestObservers {
    public static void main(String args[]) {
        // Create the Subject and Observers.
        ConcreteSubject s = new ConcreteSubject("Corn Pops", 1.29f);
        NameObserver nameObs = new NameObserver();
        PriceObserver priceObs = new PriceObserver();
        // Add those Observers!
        s.addObserver(nameObs);
        s.addObserver(priceObs);
        // Make changes to the Subject.
        s.setName("Frosted Flakes");
        s.setPrice(4.57f);
        s.setPrice(9.22f);
        s.setName("Sugar Crispies");
    }
}
```



#### QUESTION

Donnez la trace d'exécution du code ci-dessus.

*Solution*

*Test program output*



```
ConcreteSubject created: Corn Pops at 1.29
NameObserver created: Name is null
PriceObserver created: Price is 0.0
PriceObserver: Some other change to subject!
NameObserver: Name changed to Frosted Flakes
PriceObserver: Price changed to 4.57
NameObserver: Some other change to subject!
PriceObserver: Price changed to 9.22
NameObserver: Some other change to subject!
PriceObserver: Some other change to subject!
NameObserver: Name changed to Sugar Crispies
```

On oublie souvent que le moindre changement informe les deux observateurs.

Attention: l'exécution des méthodes observateur.update() n'est pas déterministe car chaque appel est effectué dans un thread spécifique.

## E.4.4. Limitations de Observable/Observer en Java

### Problème

Supposons que la classe que nous voulons rendre observable est déjà une sous-classe, par exemple :

```
class SpecialSubject extends ParentClass
```

## QUESTION

Puisque **Java** ne supporte pas l'héritage multiple, comment pouvons-nous avoir **ConcreteSubject** qui étende à la fois **Observable** et **ParentClass** ?

1. Proposez une solution
2. Définissez le diagramme de classe correspondant
3. Écrire l'implémentation **Java** (principalement **SpecialSubject**)



Deux méthodes de **java.util.Observable** sont **protected** : **setChanged()** et **clearChanged()**.

Vous pourrez pour vous aider utiliser le code de test suivant :



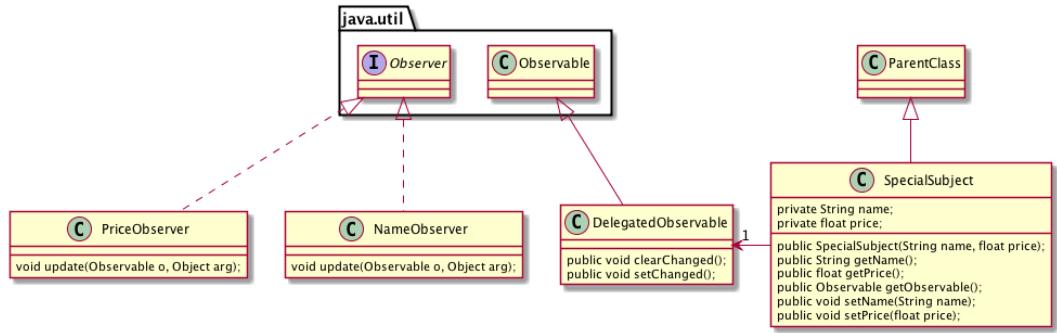
```
// Test program for SpecialSubject with a Delegated Observable.  
public class TestSpecial {  
    public static void main(String args[]) {  
        // Create the Subject and Observers.  
        SpecialSubject s = new SpecialSubject("Corn Pops", 1.29f);  
        NameObserver nameObs = new NameObserver();  
        PriceObserver priceObs = new PriceObserver();  
        // Add those Observers!  
        s.getObservable().addObserver(nameObs);  
        s.getObservable().addObserver(priceObs);  
        // Make changes to the Subject.  
        s.setName("Frosted Flakes");  
        s.setPrice(4.57f);  
        s.setPrice(9.22f);  
        s.setName("Sugar Crispies");  
    }  
}
```

Qui produit la même trace qu'à l'exercice précédent.

*Solution*



1. Une solution :
  - Utiliser une délégation
  - Nous aurons un **SpecialSubject** qui contient un **Observable**
  - Nous déléguons les aspects "observabilité" dont **SpecialSubject** a besoin à son objet contenu, de type **Observable**
2. Diagramme de classe :



### 3. Code Java :

```


/**
 * A subject to observe!
 * But this subject already extends another class.
 * So use a contained DelegatedObservable object.
 * Note that in this version of SpecialSubject we do
 * not duplicate any of the interface of Observable.
 * Clients must get a reference to our contained
 * Observable object using the getObservable() method.
 */
import java.util.Observable;

public class SpecialSubject extends ParentClass {
    private String name;
    private float price;
    private DelegatedObservable obs;

    public SpecialSubject(String name, float price) {
        this.name = name;
        this.price = price;
        obs = new DelegatedObservable();
        System.out.println("ConcreteSubject created: " + name + " at " +
price);
    }
    public String getName() {return name;}
    public float getPrice() {return price;}
    public Observable getObservable() {return obs;}
    public void setName(String string) {
        name = string;
        obs.setChanged();
        obs.notifyObservers(name);
    }
    public void setPrice(float f) {
        price = f;
        obs.setChanged();
        obs.notifyObservers(price);
    }
}


```

Explications (désolé pour l'anglais) :

What's this `DelegatedObservable` class? It implements the two methods of `java.util.Observable` that are protected methods: `setChanged()` and `clearChanged()`.



Apparently, the designers of Observable felt that only subclasses of Observable (that is, "true" observable subjects) should be able to modify the state-changed flag.

If `SpecialSubject` contains an `Observable` object, it could not invoke the `setChanged()` and `clearChanged()` methods on it. So we have `DelegatedObservable` extends `Observable` and override these two methods making them have public visibility.



**Java rule:** A subclass can change the visibility of an overridden method of its superclass, but only if it provides more access.

## QUESTION



Après avoir étudié avec votre prof préféré la solution à la question précédente :

1. Voyez-vous un problème dans cette implementation?
2. Pouvez-vous y apporter une solution?

### Solution

1. The problem: this version of `SpecialSubject` did not provide implementations of any of the methods of `Observable`. As a result, it had to allow its clients to get a reference to its contained `Observable` object using the `getObservable()` method. This may have dangerous consequences. A rogue client could, for example, call the `deleteObservers()` method on the `Observable` object, and delete all the observers! Let's have `SpecialSubject` not expose its contained `Observable` object, but instead provide "wrapper" implementations of the `addObserver()` and `deleteObserver()` methods which simply pass on the request to the contained `Observable` object.

2. The solution

```
import java.util.Observer;

public class SpecialSubject2 extends ParentClass {
    private String name;
    private float price;
    private DelegatedObservable obs;
    public SpecialSubject2(String name, float price) {
        this.name = name;
        this.price = price;
        obs = new DelegatedObservable();
    }
    public String getName() {return name;}
    public float getPrice() {return price;}
    public void addObserver(Observer o) {
        obs.addObserver(o);
    }
    public void deleteObserver(Observer o) {
        obs.deleteObserver(o);
    }
    public void setName(String name) {
        this.name = name;
        obs.setChanged();
        obs.notifyObservers(name);
    }
    public void setPrice(float price) {
        this.price = price;
        obs.setChanged();
        obs.notifyObservers(new Float(price));
    }
}
```



### QUESTION

Quelles modifications doit-on apporter au test?



### Solution



```
//s.getObservable().addObserver(nameObs);
s.addObserver(nameObs);
//s.getObservable().addObserver(priceObs);
s.addObserver(priceObs);
```

## Pour aller plus loin

### QUESTION



- Pour l'exemple du `price` et du `name`, on aurait pu faire que seuls les `PriceObserver` soient notifiés d'un changement de prix et que seuls les `NameObserver` soient notifiés d'un changement de nom. Comment faire?
- Faire le lien avec les listeners vus en Swing

### Solution



- Pas simple à faire car pas possible de "différencier" des types d'observateurs ...

Et le `notify` est déjà implémenté ...

Mais faisable:

1. avec la solution finale et 2 types de add/remove observeurs : add/removeNameObservers et add/removePriceobservers et gérer 2 Observable internes.
2. Modifier/surcharger notre implémentation de addObserver et notifyObservers, et rajouter un paramètre "typeEvenement" qui correspond au type d'évènement auquel on veut s'abonner. Mais cela nécessite de réimplémenter une grande partie de l'Observable en Java...

- Vous êtes-vous demandés dans quel ordre étaient notifiés les observateurs?
- Voir le fichier [Observer-suite.txt](#) sur le github pour compléter.

## Appendix F: CPOA - Support TP 4



### Version corrigée

Cette version comporte des indications pour les réponses aux exercices.

#### PreReq

1. Je sais programmer en Java.
2. J'ai conscience qu'il faut réfléchir avant de se lancer dans le codage.
3. Je m'aperçois que un problème de fond est la gestion des changements.

ObjTD	rendre "objet" du code a priori fonctionnel.
Durée	2 TP de 1,5h.

## F.1. Le but : "Objet-ivez" votre code

### F.1.1. Le problème

Reprendre l'application développée en S2 de gestion de comptes bancaires (fichier [TPMenus\\_applicationBanque.zip](#) à importer dans un workspace [eclipse](#)). Cette application utilise 3 classes principales :

- la classe Compte : les comptes bancaires proprement dits
- la classe AgenceBancaire : classe composée de Compte permettant de gérer une agence bancaire
- la classe ApplicationAgenceBancaire définissant le code fonctionnel de l'application : un jeu de menu permettant à un opérateur de banque de réaliser des opérations courantes.

Cette dernière classe (ApplicationAgenceBancaire) a été écrite dans une approche purement procédurale (ensemble de méthodes statiques).

### F.1.2. Le but

Le but de ce TP est de passer d'une approche fonctionnelle/procédurale du code applicatif à une approche objet. Nous allons réécrire et refactorer le code de l'application en plusieurs étapes pour obtenir un code :

- plus facile à maintenir car plus clair,
- plus facile à étendre,
- donc plus robuste aux changements.

## F.2. Etude et reprise de l'existant

**IMPORTANT**

*Attention PROFS*

Si c'est trop long : pour "Ajouter un compte" et "Supprimer un compte" ⇒ leur dire de faire des fonctions "bouchons".

Le but est surtout de faire le reste et non pas les fonctions.

1. Etudier le code proposé pour ApplicationAgenceBancaire.
2. Comprendre la structure des méthodes (menus, méthodes réalisant les opérations)
3. Ajouter les fonctions et sous-menu correspondant à l'exécution suivante :

--

Agence CAISSE ECUREUIL de PIBRAC

Menu Général

--

Choisir :

- 1 - Liste des comptes de l'agence
- 2 - Voir un compte (par son numéro)
- 3 - Menu opérations sur comptes
- 4 - Menu gestion des comptes

0 - Pour quitter ce menu

Votre choix ?

3

--

Agence CAISSE ECUREUIL de PIBRAC

Menu opérations sur comptes

--

Choisir :

- 1 - Déposer de l'argent sur un compte
- 2 - Retirer de l'argent sur un compte

0 - Pour quitter ce menu

Votre choix ?

0

Fin de Menu opérations sur comptes

--

Agence CAISSE ECUREUIL de PIBRAC

Menu Général

--

Choisir :

- 1 - Liste des comptes de l'agence
- 2 - Voir un compte (par son numéro)
- 3 - Menu opérations sur comptes
- 4 - Menu gestion des comptes

0 - Pour quitter ce menu

Votre choix ?

4

--

Agence CAISSE ECUREUIL de PIBRAC

Menu gestion des comptes

--

Choisir :

- 1 - Ajouter un compte
- 2 - Supprimer un compte

0 - Pour quitter ce menu

Votre choix ?

0

--

--

Choisir :

- 1 - Liste des comptes de l'agence
  - 2 - Voir un compte (par son numéro)
  - 3 - Menu opérations sur comptes
  - 4 - Menu gestion des comptes
- 0 - Pour quitter ce menu

Votre choix ?

1. Rencontrez-vous des difficultés pour ajouter des fonctions dans ce code qui devient "spaghetti" ?
2. Que pensez-vous de la maintenance de ce code dans 4 ans par un autre programmeur et qui devra ajouter des cascades de menus et de fonctions ?
3. N'avez-vous pas programmé plusieurs fois la même chose pour faire les menus à l'écran ?



#### *Attention PROFS*

#### **IMPORTANT**

1. Il y en a partout ... ajout d'une fonction ⇒ la fonction + modif du menu complet (affichage, contrôles des saisies, ...).
2. Maintenance compliquée
3. Sous-menus se ressemblent ...

## F.3. "Objet-iver" les fonctions

### F.3.1. Principe



Vous pouvez réfléchir 5 minutes avant de commencer cette partie : qu'est ce qui pourrait devenir objet et quelles classes seront à créer ?

Nous allons modifier le code en plusieurs classes en observant que :

1. Chaque fonction utilisateur pourrait être programmée séparément sous forme d'un objet que nous appellerons Action (option de menu) possédant :
  - a. le message affiché à l'écran pour "afficher" l'action dans un menu,
  - b. une méthode pour exécuter cette option de menu.
2. Un menu pourrait être programmé séparément sous forme d'un objet que nous appellerons ActionList (liste d'actions de menus) possédant :
  - a. le message affiché à l'écran pour "afficher" le menu comme un sous-menu de menu,
  - b. des méthodes pour ajouter/retirer des options de menus dans ce menu,
  - c. une méthode pour exécuter cette menu (affichage et déclenchement des actions).

### F.3.2. Les fonctions utilisateurs comme des objets

1. Faire une copie du code source précédent sous le nom applicationBanqueAction.
2. Créer les packages suivants dans Eclipse :

```
application
application.action
application.actionlist
```

3. Etudier la définition d'interface suivante et insérer sa définition dans votre projet dans le package action :

```
package action;
import banque.AgenceBancaire;
/**
 * An Action is an object that implements some action of a user's menu.<BR>
 * It is defined by a message, an optional code, an execute method to "do" the
action.
 */
public interface Action {
    /**
     * Message of the action (to show on screen).
     *
     * @return the message of the action
     */
    public String actionMessage();

    /**
     * Code of the action (may be used to identify the action among other ones).
     *
     * @return the code of the action
     */
    public String actionCode();

    /**
     * The method to call in order to "execute" the action on <code>ag</code>.
     *
     * @param ag the AgenceBancaire on which the action may act on
     * @throws Exception when an uncaught exception occurs during execution
     */
    public void execute(AgenceBancaire ag) throws Exception;
}
```

4. Déclarer une classe par action (option de menu) à utiliser. Commencer par "Liste des comptes de l'agence" :
  - a. Créer une classe (le nom ActionListeDesComptes paraît adapté) dans le package application.action,

- b. qui implémente Action,
  - c. dotée de deux attributs (message, code)
  - d. écrire le code dont un constructeur correctement paramétré,
  - e. la méthode execute(AgenceBancaire) réalisera l'affichage écran de la liste des comptes de l'agence bancaire en paramètre.
5. De la même manière, déclarer une classe pour l'action "Voir un compte (par son numéro)" (classe ActionVoirCompteNumero) dans le package application.action.

### F.3.3. Les menus utilisateurs comme des objets

1. Etudier la définition d'interface suivante et insérer sa définition dans votre projet dans le package action :

```

package action;

/**
 * An ActionList is an object that implements a end-user menu.<BR>
 * It is defined by a title (printed on top of the menu).<BR>
 * It is also defined by a list of different action objects that the menu
manages.<BR>
 * It is attended to :<BR>
 * - display the end-user menu numbered from 1 (list of messages of actions).<BR>
 * - display a quit option (0).<BR>
 * - wait for some user response.<BR>
 * - launch the requested action.<BR>
 */
public interface ActionList extends Action {
    /**
     * Title of the list of actions (menu).
     *
     * @return the title of the action list
     */
    public String listTitle();

    /**
     * The number of actions in the action list.
     *
     * @return number of actions in the action list.
     */
    public int size();

    /**
     * Add an action at the end of the list action if it does not yet exists.
     *
     * @param ac the action to add
     * @return true if action is added, else false
     */
    public boolean addAction(Action ac);
}

```

2. Déclarer une classe ActionListAgenceBancaire dans le package application.actionlist,
  - a. qui implémente ActionList,
  - b. dotée de quatre attributs (message, code, title, liste des actions). La liste des actions sera les différentes options que l'action list (menu) devra afficher.
  - c. écrire le code dont un constructeur correctement paramétré,
  - d. la méthode execute(AgenceBancaire) réalisera ce qui est défini dans la documentation. Le menu affiché sera identique à celui de la version antérieure de l'application. Chaque option de menu sera numérotée par execute() de 1 à n (nombre d'action) + 0 pour sortir du menu.

## F.3.4. Et maintenant : go ! Maintenance et extension facilitées

1. Créer une classe contenant un main et permettant :
  - a. de créer une instance de chaque classe Action créée,
  - b. de créer une instance de ActionListAgenceBancaire,
  - c. lancer execute() sur l'instance de ActionListAgenceBancaire.



Ca marche ?

2. Vous pouvez créer les autres actions et sous-menus.
3. Pourquoi ActionList hérite de Action à votre avis ?



On aurait pu utiliser un autre patron appelé Composite ... plus tard peut être

*Version corrigée*

**CAUTION**

Car cela permet d'ajouter un menu dans un menu (un ActionList dans un ActionList)

## F.4. Abstraire le problème

### F.4.1. Une nouvelle application ... et mince ...

Supposons que nous devions développer une application de gestion d'une liste d'élèves (classes Eleve et GroupeEleve). Elle est basée sur un menu permettant de :

- Voir la liste des élèves.
- Afficher un élève à partir du nom.
- Modifier les notes d'un élève.
- Ajouter un élève dans le groupe.
- Retirer un élève du groupe.
- ...

Ca vous rappelle des choses ?

Questions :

1. Considérant les nouvelles classes Eleve et GroupeEleve, peut-on réutiliser telles quelles (sans les modifier) les interfaces Action et ActionList dans la nouvelle application ?
2. Si oui : pourquoi ?
3. Si non : pourquoi ?

**CAUTION***Version corrigée*

Non car elles sont paramétrées par AgenceBancaire ...

### F.4.2. Abstrayons un peu le problème

Compte tenu des observations de la section précédente, il faudrait des classes Action et ActionList dont execute() prendrait en paramètre n'importe quel objet. Utiliser la classe Object ? Non, la généricité est là pour nous aider ...

1. Faire une copie du code source précédent sous le nom applicationBanqueActionGenerique.
2. Modifier les déclarations des interfaces Action et ActionList comme suit (attention, tout le code va devenir "erroné") :

```
package action;
/**
 * An Action is an object that implements some action of a user's menu.<BR>
 * It is defined by a message, an optional code, an execute method to "do" the
action.<BR>
 * It is parameterized by the type of object on which the action may act on
(execute on).
 *
 * @param <E> The type of object on which the action may act on.
 */
public interface Action <E> {
    /**
     * Message of the action (to show on screen).
     *
     * @return the message of the action
     */
    public String actionMessage();

    /**
     * Code of the action (may be used to identify the action among an action
list).
     *
     * @return the code of the action
     */
    public String actionCode();

    /**
     * The method to call in order to "execute" the action on <code>e</code>.
     *
     * @param e the object on which the action may act on
     * @throws Exception when an uncaught exception occurs during execution
     */
    public void execute(E e) throws Exception;
}

package action;
```

```

/**
 * An ActionList is an object that implements a end-user menu.<BR>
 * It is defined by a title (printed on top of the menu).<BR>
 * It is also defined by a list of different action objects that the menu
manages.<BR>
 * It is attended to :<BR>
 * - display the end-user menu numbered from 1 (list of messages of actions).<BR>
 * - display a quit option (0).<BR>
 * - wait for some user-response.<BR>
 * - launch the requested action.<BR>
 *
 * It is parameterized by the type of object on which the actions of the list
action may act on (execute on).<BR>
 *
 * @param <E> The type of object on which the list action may act on.
 */
public interface ActionList<E> extends Action<E>{
    /**
     * Title of the list of actions (menu).
     *
     * @return the title of the action list
     */
    public String listTitle();

    /**
     * The number of actions in the action list.
     *
     * @return number of actions in the action list.
     */
    public int size();

    /**
     * Add an action at the end of the list action if it does not yet exists.
     *
     * @param ac the action to add
     * @return true if action is added, else false
     */
    public boolean addAction(Action<E> ac);
}

```

3. Modifier chaque classe créée (les Action puis ActionList puis le main()) pour soit implémenter la bonne instantiation des interfaces, soit instancier correctement les objets.
4. Tout doit fonctionner.
5. Il n'y a plus qu'à faire la nouvelle application.

## F.5. Pour aller plus loin : complétons et encore plus d'abstraction

### F.5.1. Une interface ActionList plus complète

1. Faire une copie du projet précédent
2. Pour de vraies applications, ajouter à l'interface ActionList les opérations suivantes :

```
/**
 * Add an action in the list action at the specified index if it does not yet
exists.
 *
 * @param ac the action to add
 * @param index index to add the action
 * @return true if action is added, else false
 * @throws IndexOutOfBoundsException if (index < 0) || (index > size())
*/
public boolean addAction(Action<E> ac, int index);

/**
 * Remove an action from the list action at the specified index.
 *
 * @param index index to remove the action
 * @return true
 * @throws IndexOutOfBoundsException if (index < 0) || (index > size())
*/
public boolean removeAction(int index);

/**
 * Remove an action from the list action.
 *
 * @param ac the action to remove
 * @return true if action is removed (found), else false
*/
public boolean removeAction(Action<E> ac);

/**
 * List of the messages of actions contained in the action list
 *
 * @return an array of messages of the list action
*/
public String[] listOfActions() ;
}
```

### F.5.2. Quid d'ActionList ?

La classe ActionListAgenceBancaire qui met en oeuvre un menu (qui implémente ActionList) est ici

créeé spécifiquement pour AgenceBancaire. Mais cela est il nécessaire dans chaque application ? (en supposant ne rien afficher de l'AgenceBancaire). On devrait pouvoir faire une classe "générique" de gestion de menus composés d'actions et réutilisable dans chaque application.

Alors essayons :

1. Faire une copie du projet de la section précédente
2. Déplacer la classe ActionListAgenceBancaire dans le package action.
3. Renommer cette classe en un nom contenant "ActionList" et bien choisi. AbstractActionList serait TRES mal choisi.
4. Pour rendre cette classe générique (et non pas abstraite), modifier l'en-tête en

```
public class GenericActionList<E>
    implements ActionList<E>
```

5. Attention, tout le code va maintenant "klaxonner" en rouge ! normal ...
6. Modifier chaque fois que nécessaire pour utiliser le type générique E
7. Enlever tout accès à la classe AgenceBancaire (affichage nom de la banque, ...)
8. Vous devriez arriver au bout ...
9. Enfin dans le main il y aura quelques "klaxons warnings" sur la création d'objets de cette nouvelle classe car il faudra indiquer le type paramètre à la création.



ATTENTION : faire une classe générique n'est pas toujours aussi simple. Ici le cas a été simplifié à l'extrême.

### F.5.3. Troisième étape : abstraction encore plus

Le problème :

1. Supposons que l'on veuille utiliser notre application dans une système différent où saisies et affichages ne se font pas sur le terminal d'exécution de l'application ... Les instructions utilisant new Scanner (System.in) ou System.out.println ... deviennent obsolètes.
2. Tout comme l'agence utilisée dans les traitements, ces 2 éléments font maintenant partie du **contexte d'exécution** des actions.
3. D'autres éléments pourraient être utilisés : des transactions en cours (réservations aériennes), des bases de données, des connexions diverses, ...
4. Il faut donc créer un **contexte d'exécution** qui sera en paramètre des Action et ActionList.

Allons-y !

1. Faire une copie du projet de la section précédente (sans généréricité).
2. Dans le package application, créer une classe ApplicationContextAgenceBancaire implementant le pattern Singleton permettant d'accéder :

- a. A l'agence bancaire "en cours".
  - b. Au Scanner à utiliser. L'initialiser ici avec un Scanner sur System.in mais autre chose pourrait être utilisé (un fichier, un flux vers un terminal, ...).
  - c. A la sortie PrintStream à utiliser. Ici ce sera System.out mais autre chose pourrait être utilisé (un fichier, un flux vers un terminal, ...).
3. Refactorer tout le code :
- a. Les classes Action et ActionList utilisant maintenant le type ApplicationContextAgenceBancaire (à la place de AgenceBancaire)
  - b. Modifier les accès à l'agence bancaire en utilisant ApplicationContextAgenceBancaire.
  - c. Modifier les accès à l'entrée standard en utilisant ApplicationContextAgenceBancaire.
  - d. Modifier les accès à la sortie standard en utilisant ApplicationContextAgenceBancaire.
4. Ca marche ??

## About...

Document réalisé par [JMB](#) via [Asciidoc](#) (version [2.0.16](#)) de 'Dan Allen', lui-même basé sur [AsciDoc](#).