



Université
de Limoges



Institut
Universitaire
de Technologie
du Limousin

M3105 - Conception et programmation
objet avancée

Patron de conception : Visiteur

Quentin DELAGE
Mylan DEVEAU

Mathieu ZANI
Clément GLADIN

Sommaire

- 1) Un patron de conception ?
 - a) Introduction
- 2) Visiteur ?
 - a) sdjkfh
 - b) sdfkh
- 3) Plus en détail
 - a) Exemple d'utilisation

Les patrons de conception

ou

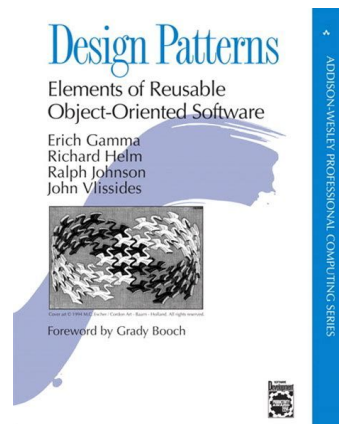
Design Patterns (en)

Introduction :

En développement -> arrangement caractéristique, c'est à dire un standard pour un problème donnée

3 principaux types (GoF) :

1. Création : construction des objets
2. Structure : structure commune
3. Comportement : déléguer les traitements



« Les patrons offrent la possibilité de capitaliser un savoir précieux né du savoir-faire d'experts »

F. BUSCHMANN



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

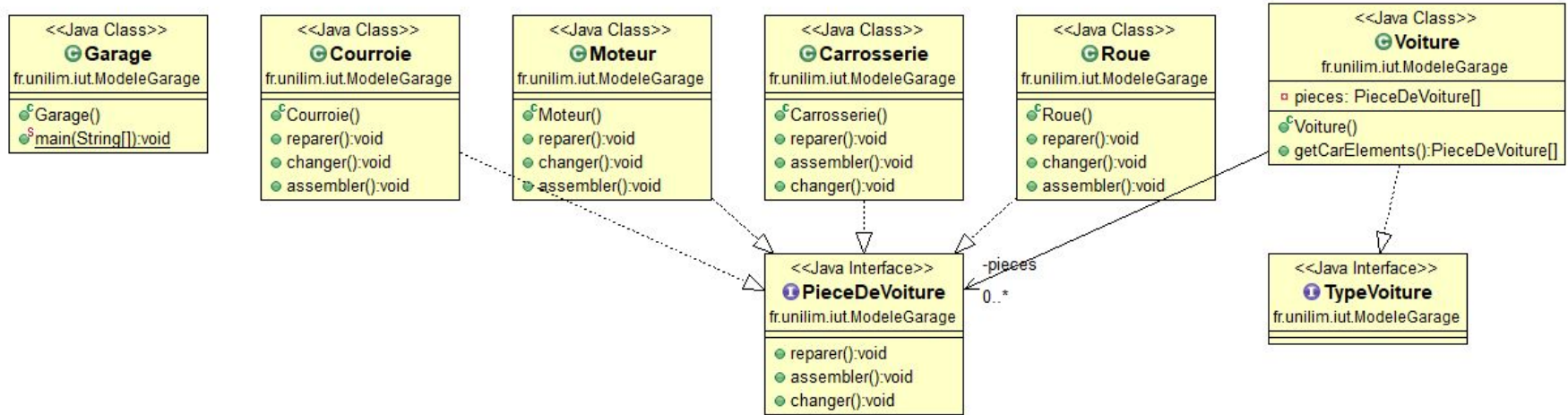
On Patterns and Pattern Languages

Volume 5



Frank Buschmann
Kevlin Henney
Douglas C. Schmidt

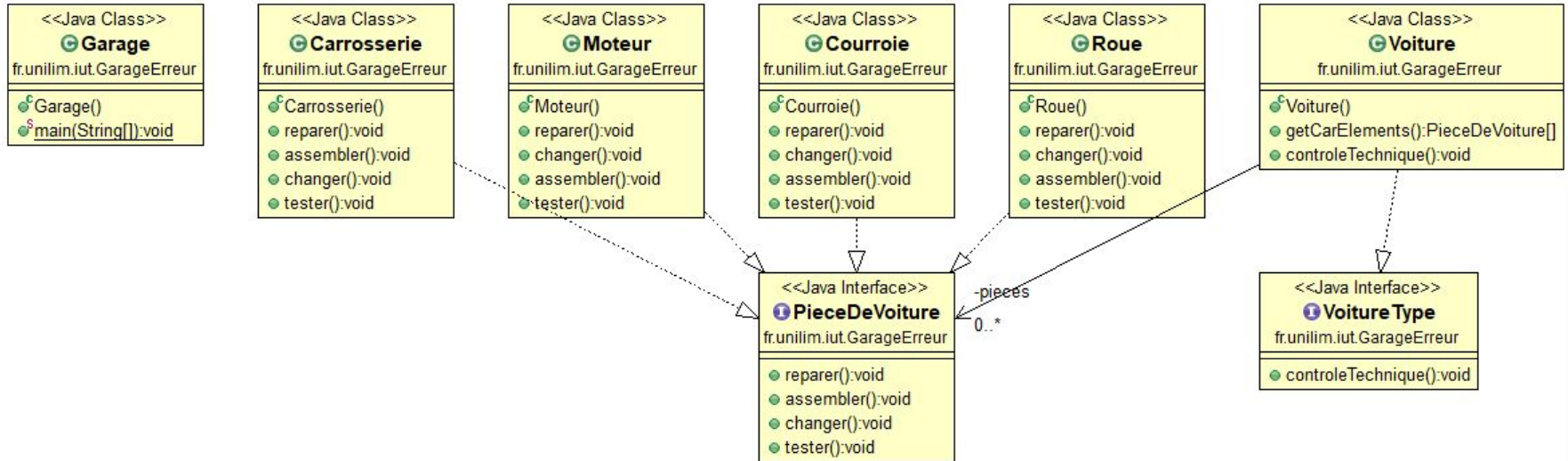
Exemple avec une modélisation de garage



On a déjà cet existant, un code testé et qui fonctionne.
On veut rajouter la possibilité d'effectuer le contrôle technique d'une voiture.

Exemple avec une modélisation de garage

On peut répondre à cette demande en changeant l'ancien modèle pour celui-ci.



Le modèle précédent répond bien à la demande cependant il y a plusieurs inconvénients :

- On ne respecte pas le Open-Closed Principe.
- On risque de modifier certains comportements
- Le code qui existait précédemment risque donc de devenir obsolète

Garage.java

```
public class Garage {  
  
    public static void main(String[] args) {  
        Voiture car = new Voiture();  
        System.out.println("J'ai construit la voiture");  
        car.controleTechnique();  
    }  
}
```

Affichage

```
J'assemble le moteur  
J'assemble la courroie  
J'assemble la roue  
J'assemble la roue  
J'assemble la carrosserie  
J'ai construit la voiture  
Je teste le Moteur  
Je teste la courroie  
Je teste la roue  
Je teste la roue  
Je teste la carrosserie  
J'ai fini le controle technique
```

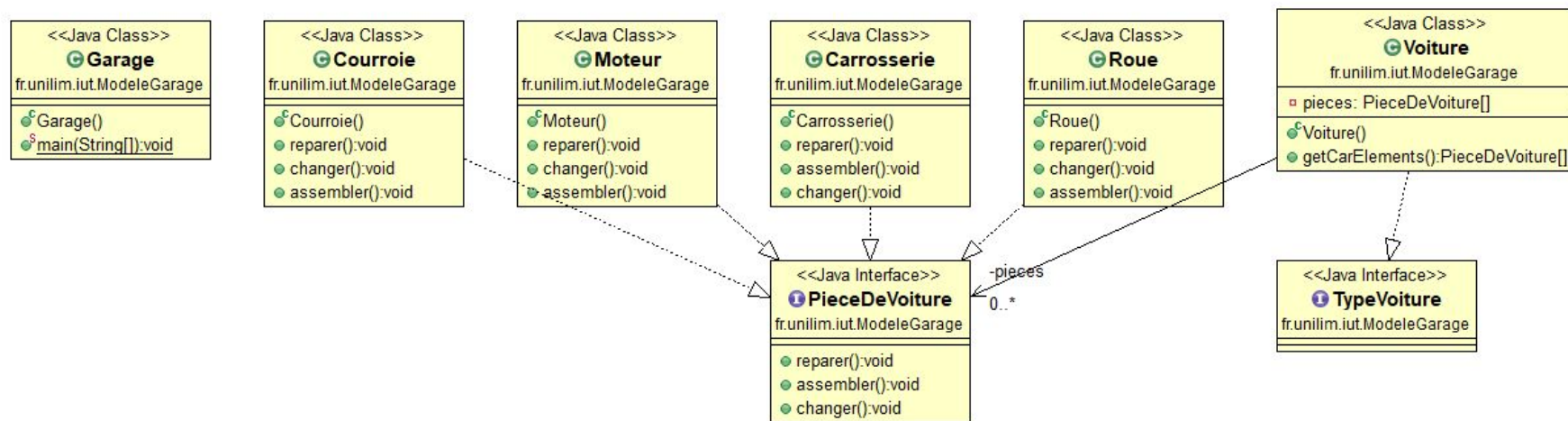
Ce que l'on peut tirer de ce programme

- On a bien répondu à la demande
- Cependant on modifie les comportements existants, ce qui pourrait avoir des effets inattendus
- Il faut modifier toutes les classes concernées par la modification, cela rend donc le programme moins lisible

Comment pourrait-on régler ces problèmes ?

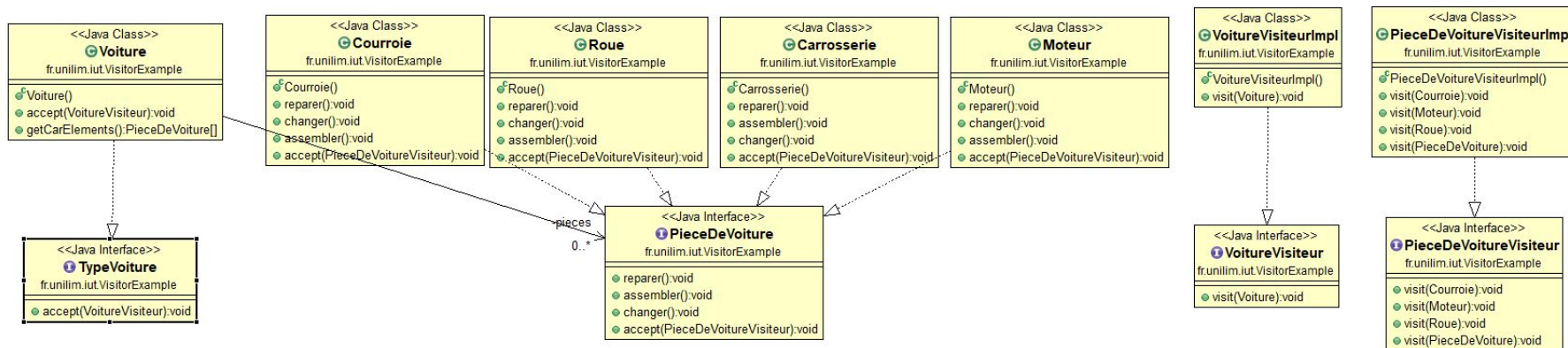
Faisons appel au pattern Visitor.

Rappel de l'existant



On veut rajouter un nouveau comportement qui sera illustré par `visit()`.
Cette fois on veut éviter de trop modifier le code des classes existantes qui ont déjà été testées.

Solution en utilisant Visitor



On regroupe toutes les actions sur les différentes classes dans une interface

PieceDeVoitureVisiteur.java

```
public interface PieceDeVoitureVisiteur {  
  
    void visit(Courroie courroie);  
  
    void visit(Moteur moteur);  
  
    void visit(Roue roue);  
  
    void visit(PieceDeVoiture carElement);  
  
}
```

PieceDeVoitureVisiteurImpl.java

```
package fr.unilim.iut.VisitorExample;  
  
public class PieceDeVoitureVisiteurImpl implements PieceDeVoitureVisiteur {  
  
    public void visit(Courroie courroie) {  
        System.out.println("Je visite la courroie");  
    }  
  
    public void visit(Moteur moteur) {  
        System.out.println("Je visite le moteur");  
    }  
  
    public void visit(Roue roue) {  
        System.out.println("Je visite le moteur");  
    }  
  
    public void visit(PieceDeVoiture carElement) {  
        String classPiece = new String(carElement.getClass().getSimpleName());  
        System.out.println("Je visite le/la " + classPiece);  
    }  
  
}
```

Une méthode accept est apparue dans les classes implémentant PieceDeVoiture

PieceDeVoiture.java

```
public interface PieceDeVoiture {  
  
    public void reparer();  
  
    public void assembler();  
  
    public void changer();  
  
    public void accept(PieceDeVoitureVisiteur visitor);  
}
```

Roue.java

```
public void assembler() {  
    System.out.println("J'assemble la roue");  
}  
  
public void accept(PieceDeVoitureVisiteur visitor) {  
    visitor.visit(this);  
}  
}
```

Le modèle précédent répond bien à la demande et les problèmes cités précédemment ont disparus.

Garage.java

```
public class Garage {  
  
    public static void main(String[] args) {  
        Voiture car = new Voiture();  
        System.out.println("J'ai construit la voiture");  
        VoitureVisiteur carVisitor = new VoitureVisiteurImpl();  
        carVisitor.visit(car);  
    }  
}
```

Affichage

```
J'assemble le moteur  
J'assemble la courroie  
J'assemble la roue  
J'assemble la roue  
J'assemble la carrosserie  
J'ai construit la voiture  
Je visite le/la Moteur  
Je visite le/la Courroie  
Je visite le/la Roue  
Je visite le/la Roue  
Je visite le/la Carrosserie
```

Le patron Visiteur

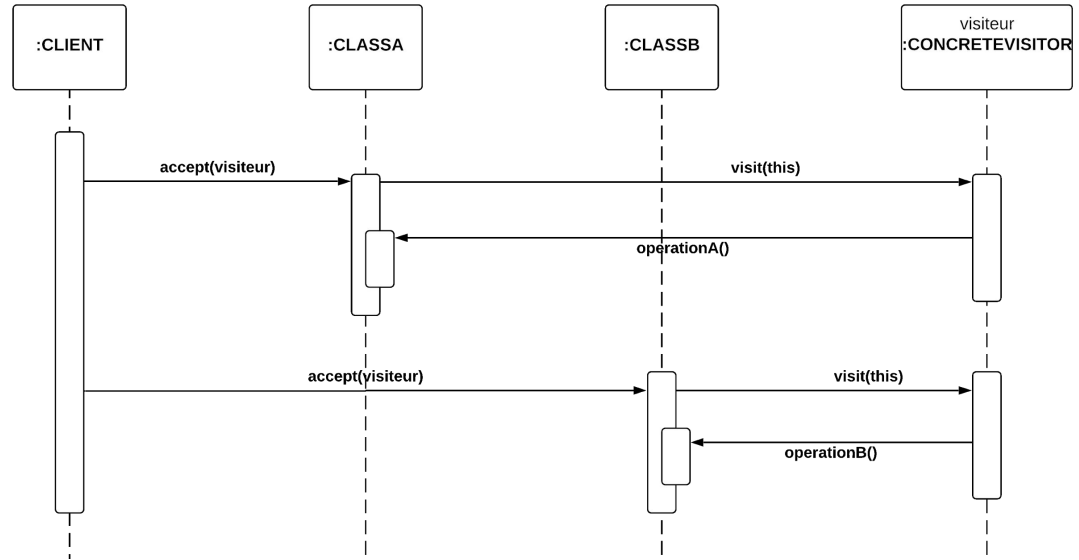
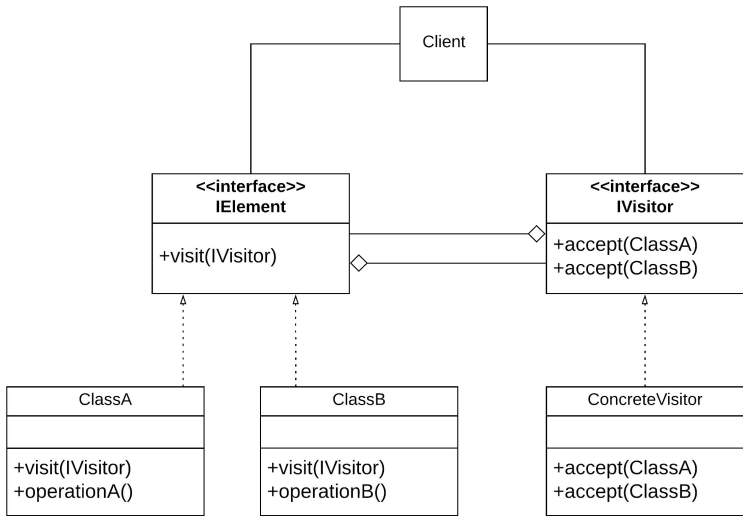
Type de GoF :

- Comportement

Intention :

- ajouter de nouvelles opérations sans modifier les classes des éléments sur lesquels elles opèrent.
- Représenter **UNE** opération à effectuer sur les élément d'une structure

Solution du pattern visiteur



Principes SOLID

Single Responsibility Principle
(principe de responsabilité unique)

Séparer les responsabilités d'une seule classe en plusieurs classes avec 1 seule responsabilité

Open/Closed Principle
(principe d'ouverture/fermeture)

Permet d'ajouter des fonctionnalités sans modifier une classe déjà terminée

Les limites du pattern visiteur

- Il est difficile et coûteux d'envisager un autre pattern au cas où celui-ci ne convient plus.
- Il est facile d'ajouter de nouvelles opérations, mais dur d'ajouter de nouveaux types.
- Il rend les classes dépendantes les une des autres

Lien entre les patterns

Le pattern Décorateur:

- divise les responsabilités
- facilite l'ajout d'opération
- permet l'extension sans la modification de classe existante
- flexibilité

Les classes et interfaces de la javadoc qui mettent en oeuvre le pattern:

paquetage :

javax.lang.model.util

assiste dans le développement des types et Program elements (les interfaces)

javax.lang.model.element

AnnotationValueVisitor

ElementVisitor

javax.lang.model.type

TypeVisitor

java.nio.file

SimpleFileVisitor

javax.faces.component.visit

visitCallback

Application dans un nouveau contexte

Gestion d'un épicerie : scan des prix TTC et pèse des poids

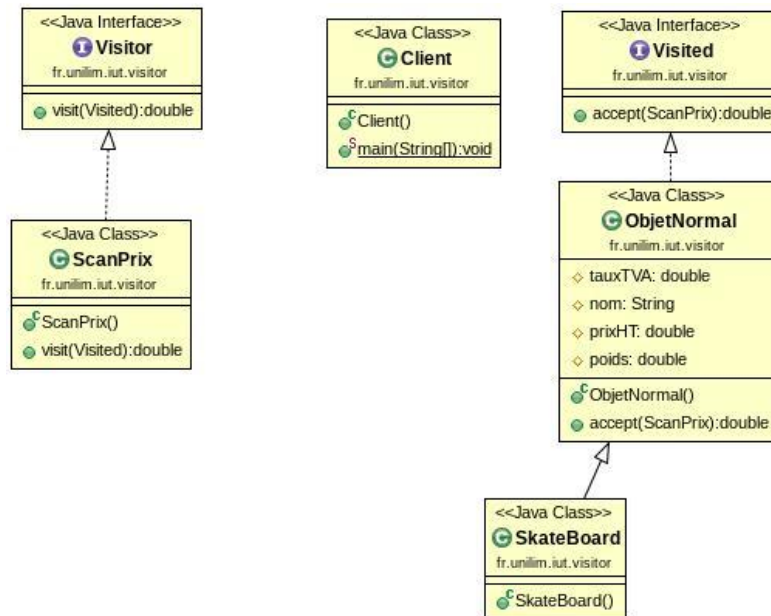
Visualisation sous Eclipse

Diagrammes de classes aux différentes étapes

Étape 0 - Interfaces vides :



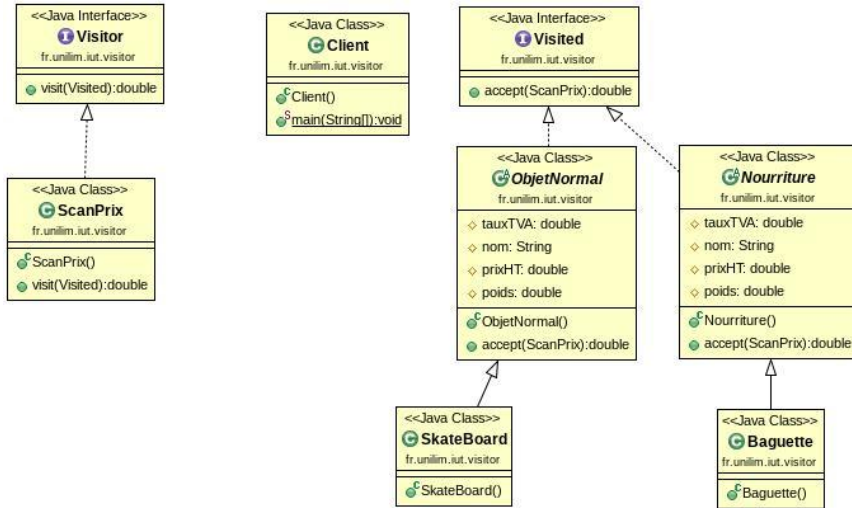
Étape 1 - Implémentation de base :



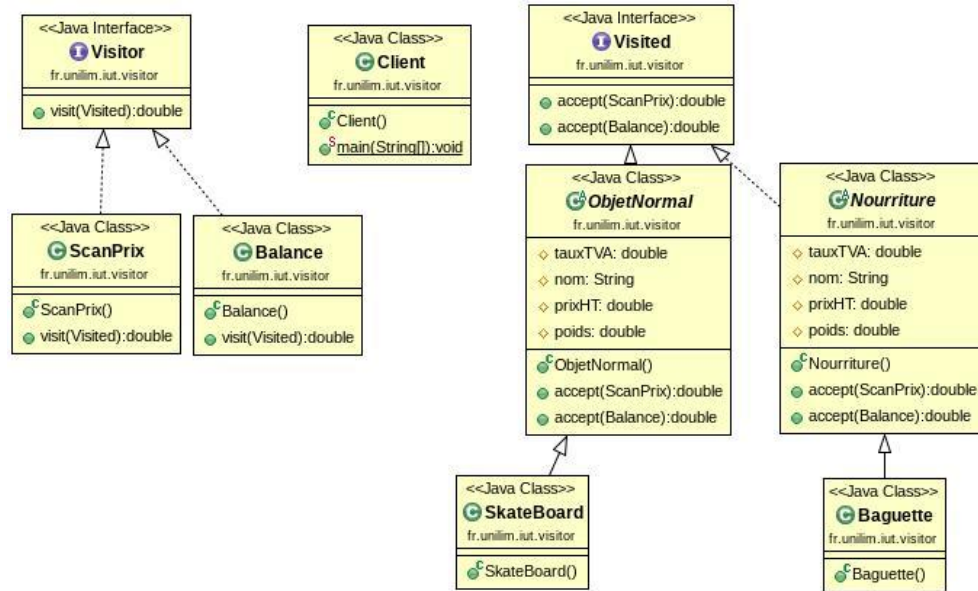
À suivre avec les commits sur ce dépôt :

<https://github.com/Dylage/M3105---DP-Visiteur/tree/master>

Étape 2 - Ajout d'un nouvel objet :



Étape 3 - Ajout d'un nouveau visiteur :



Liste des références utilisées

Sites internet :

Auteurs multiples, WikiBooks : https://fr.wikibooks.org/wiki/Patrons_de_conception/Visiteur

O. BOISSIER, G. PICARD, Cours de Mines Saint Étienne : <https://www.emse.fr/~picard/cours/2A/DesignPatterns.pdf>

Petru VALICOV, IUT de Montpellier-Sete : http://pageperso.lif.univ-mrs.fr/~petru.valicov/Cours/M3105/DP_x4.pdf

Alexandre BRILLANT, cours en ligne : <https://abrillant.developpez.com/tutoriel/java/design/pattern/introduction/>

Javadoc : <https://docs.oracle.com/javase/8/docs/api/javax/lang/model/element/package-summary.html>

Journaldev : <https://www.journaldev.com/31902/gangs-of-four-gof-design-patterns>

Dépôts Gits :

BingoKnight, dépôt Git : <https://github.com/BingoKnight/Visitor-Pattern>

cpatel10 : dépôt Git : <https://github.com/cpatel10/patterns-VisitorAndDecorator>

QCM

Avez-vous bien suivi ?

Lien raccourci : <https://urlz.fr/b1Kn>

Lien originel :

[https://tech.io/playgrounds/51758/qcm---
patron-de-conception-viteur](https://tech.io/playgrounds/51758/qcm---patron-de-conception-viteur)