M3105 - Conception et programmation objet avancée

Patron de conception : Visiteur

Quentin DELAGE
Mylan DEVEAU
Mathieu ZANI
Clément GLADIN

Sommaire

Les patrons de conception

ou

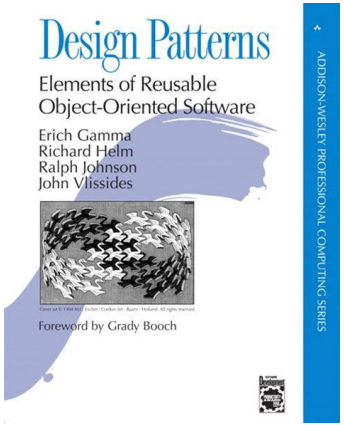
Design Patterns (en)

Introduction :

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

3 principaux types (GoF) :

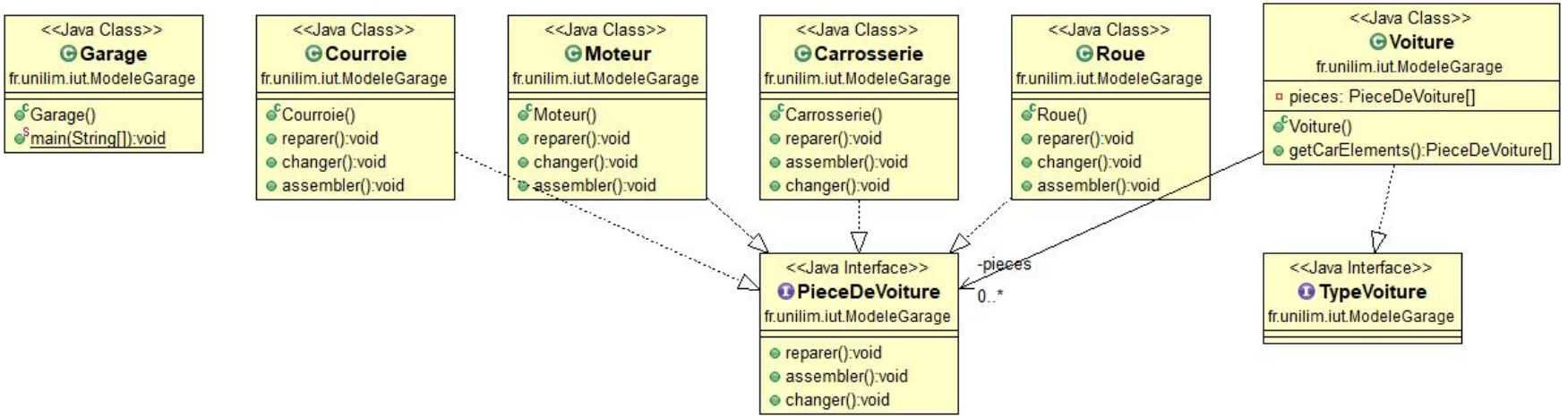
1. Création : construction des objets
2. Structure : structure commune
3. Comportement : déléguer les traitements



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

F. BUSCHMANN

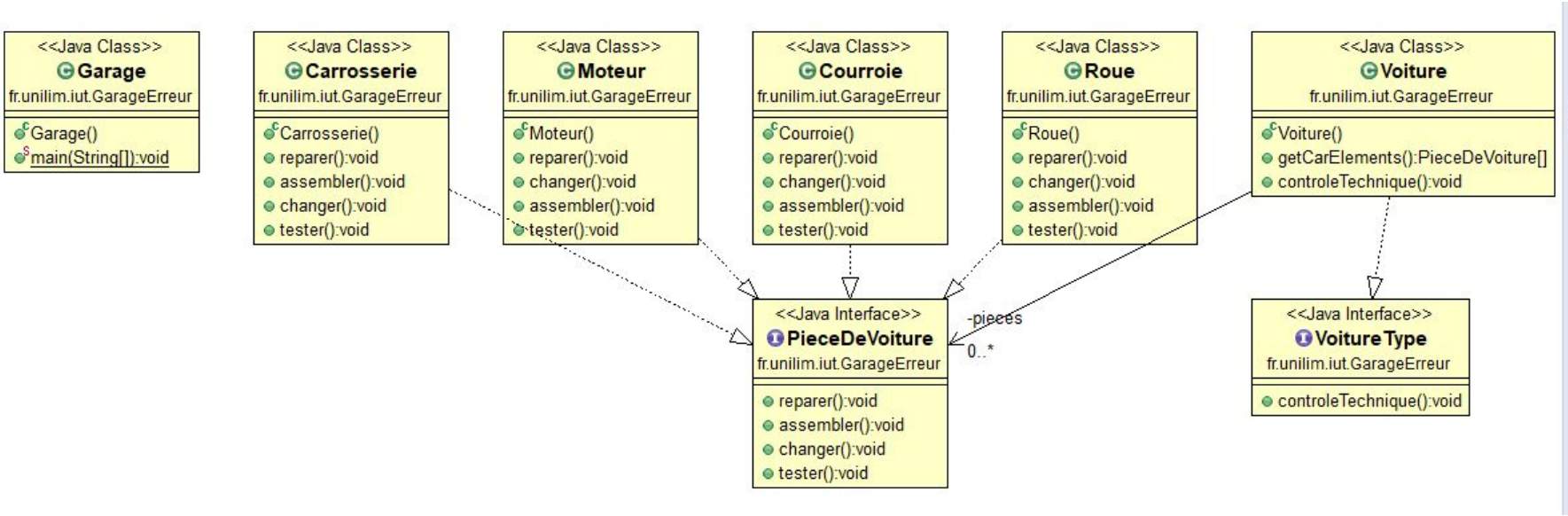
Exemple avec une modélisation de garage



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 Principle.
- On risque de modifier certains comportements
- Le code qui existait précédemment risque donc de devenir obsolète

Garage.java

```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
```

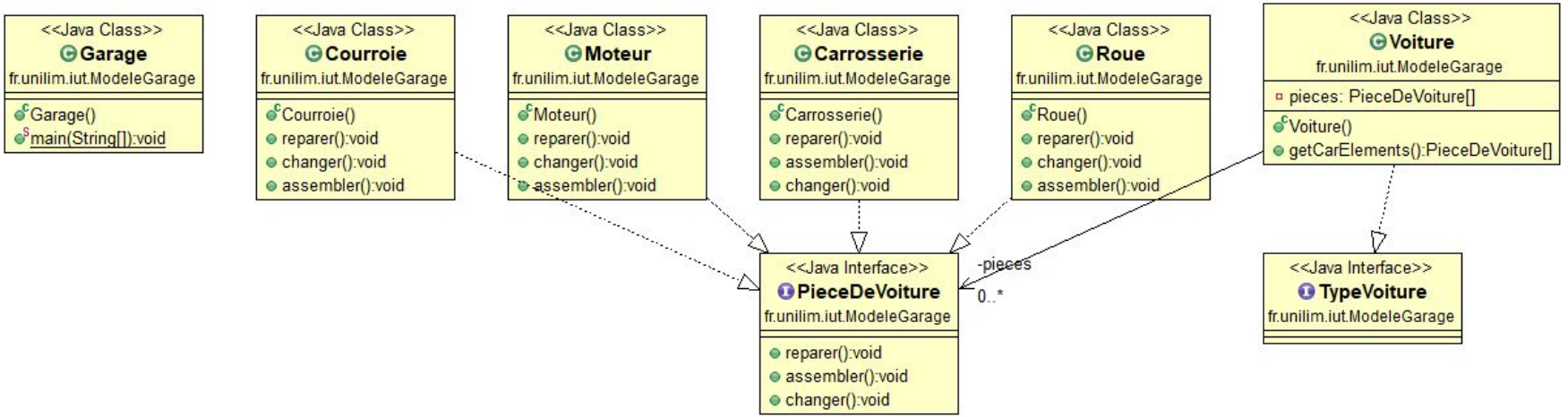
Ce que l’on peut tirer de ce programme

- 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 ?

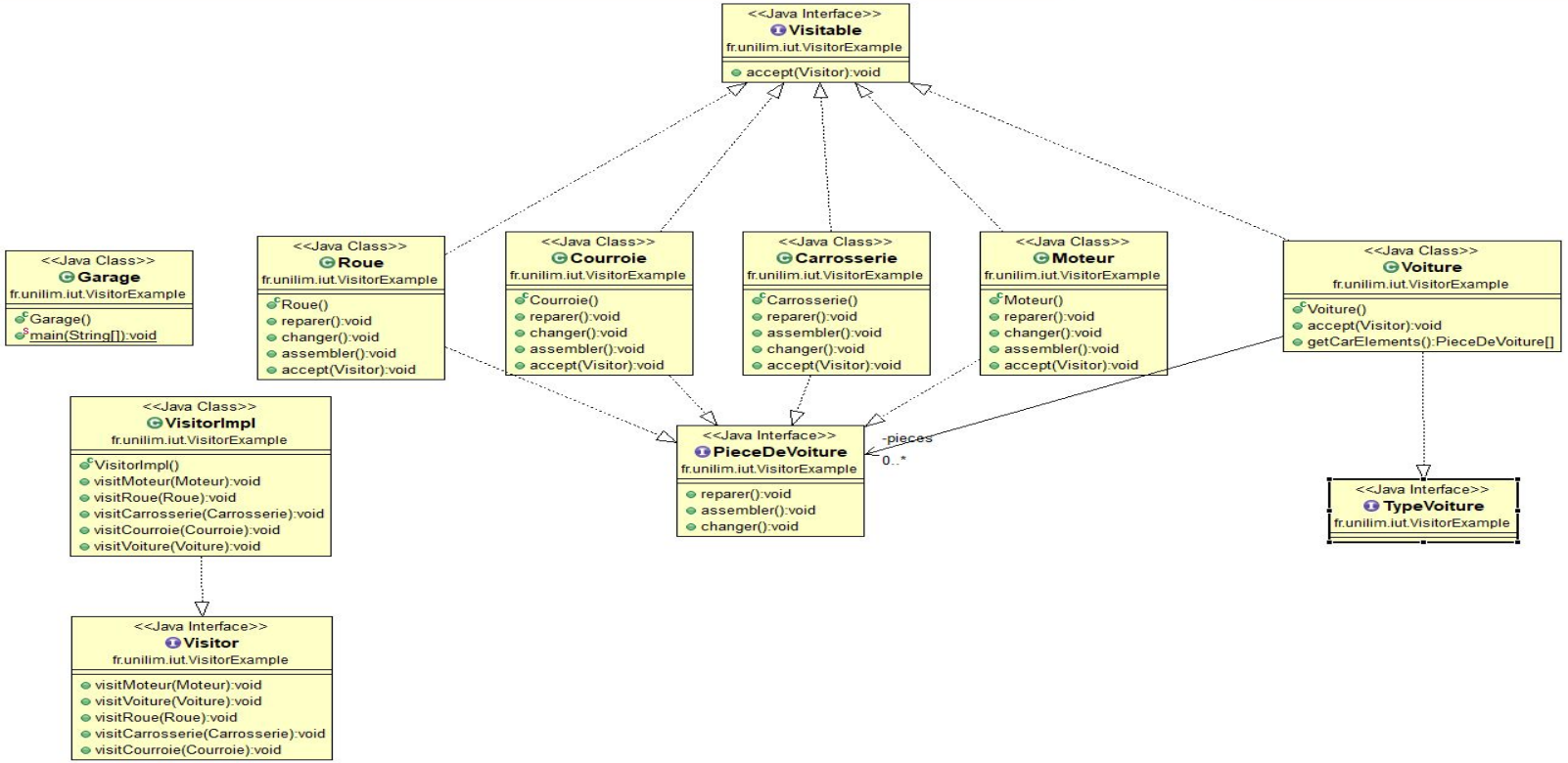
Tentons autre chose.

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

Visitor.java

```java
public interface Visitor {

    void visitMoteur(Moteur moteur);

    void visitVoiture(Voiture voiture);

    void visitRoue(Roue roue);

    void visitCarrosserie(Carrosserie carrosserie);

    void visitCourroie(Courroie courroie);

}
```

Les classes que l'on veut étendre vont désormais implémenter Visitable.java

Visitable.java

```java
public interface Visitable {
    public void accept(Visitor visitor);
}
```

Roue.java

```java
public class Roue implements PieceDeVoiture,Visitable{


    public void reparer() {
        System.out.println("Je répare la roue");

    }

    public void changer() {
        System.out.println("Je change la roue");
    }

    public void assembler() {
        System.out.println("J'assemble la roue");
    }

    public void accept(Visitor visitor) {
        visitor.visitRoue(this);
    }
}
```

On créé une classe qui va implémenter Visitor.

VisitorImpl.java

```java
public class VisitorImpl implements Visitor {

    public void visitMoteur(Moteur moteur) {
        System.out.println("Je visite le moteur");

    }

    public void visitRoue(Roue roue) {
        System.out.println("Je visite la roue");
    }

    public void visitCarrosserie(Carrosserie carrosserie) {
        System.out.println("Je visite la carrosserie");
    }

    public void visitCourroie(Courroie courroie) {
        System.out.println("Je visite la courroie");
    }

    public void visitVoiture(Voiture voiture) {
        PieceDeVoiture tab[] = voiture.getCarElements();
        for (PieceDeVoiture element : tab) {
            if (element instanceof Roue) {
                this.visitRoue((Roue) element);
            } else if (element instanceof Moteur) {
                this.visitMoteur((Moteur) element);
            } else if (element instanceof Courroie) {
                this.visitCourroie((Courroie) element);
            } else if (element instanceof Carrosserie) {
                this.visitCarrosserie((Carrosserie) element);
            }
        }
        System.out.println("J'ai visité tous les éléments de la voiture");
    }

}
```

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

Garage.java

```java
public class Garage {

    public static void main(String[] args) {
        Voiture car = new Voiture();
        System.out.println("J'ai construit la voiture");
        Visitor visitor = new VisitorImpl();
        visitor.visitVoiture(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 moteur
Je visite la courroie
Je visite la roue
Je visite la roue
Je visite la carrosserie
J'ai visité tous les éléments de la voiture
```

Le patron Visiteur

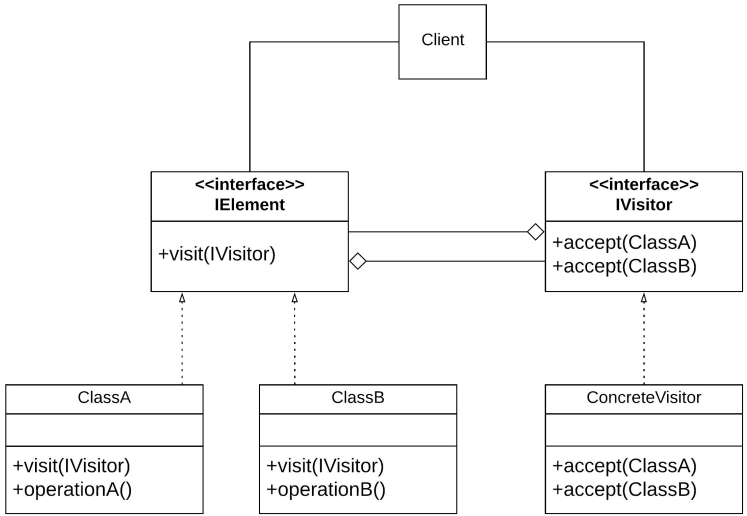
Type de GoF :

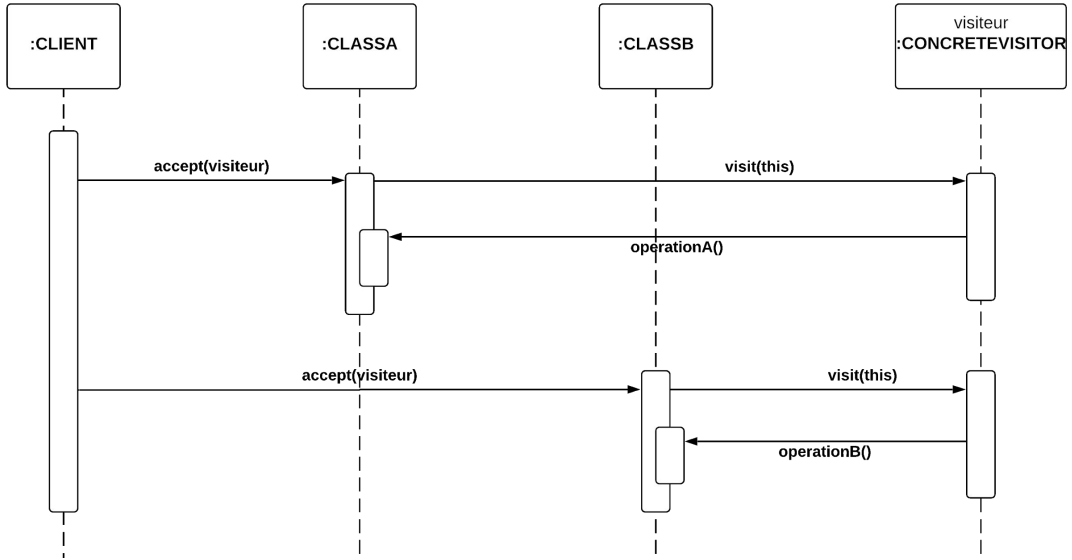
- Comportement

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 seul classe en plusieurs classes avec 1 seul responsabilité

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

Permet d'ajouter des fonctionnalitées 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 unes 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

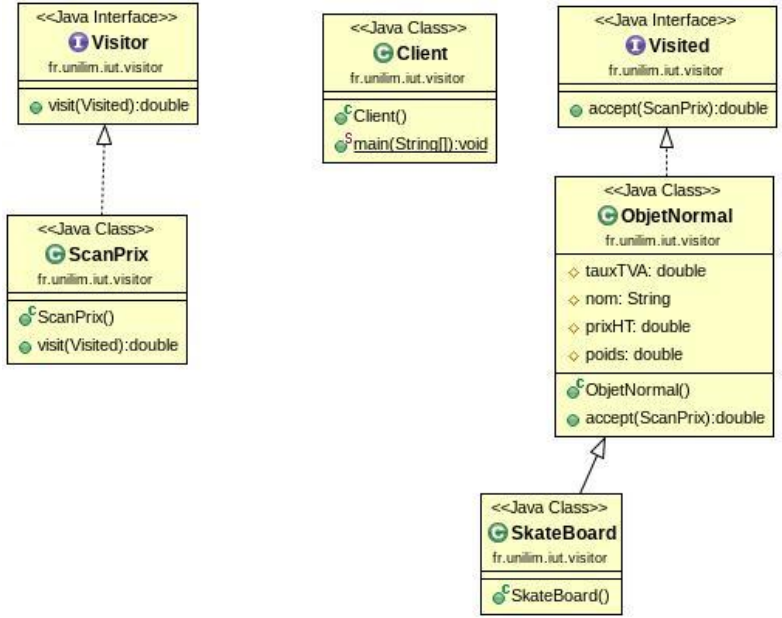
Visualisation sous Eclipse

Diagrammes de classes aux différentes étapes

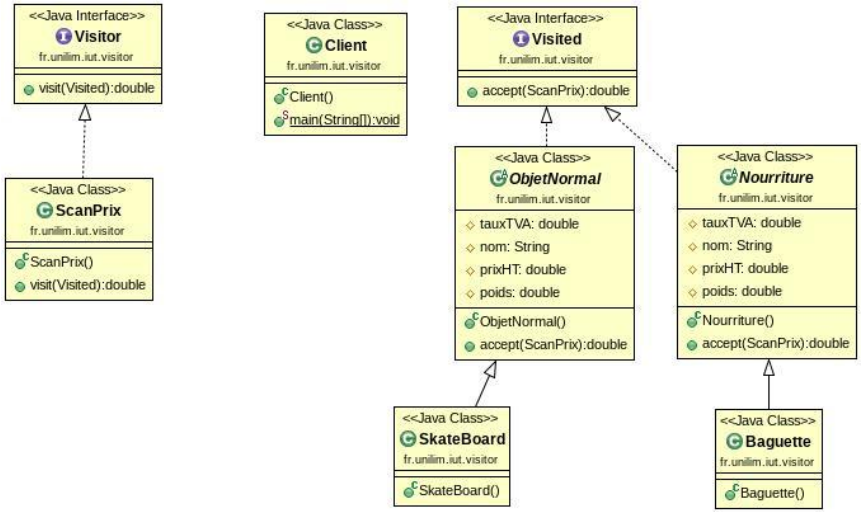
À suivre avec les commits sur ce dépot : https://github.com/Dylage/M3105---DP-Visiteur/tree/master

Étape 0 - Interfaces vides :

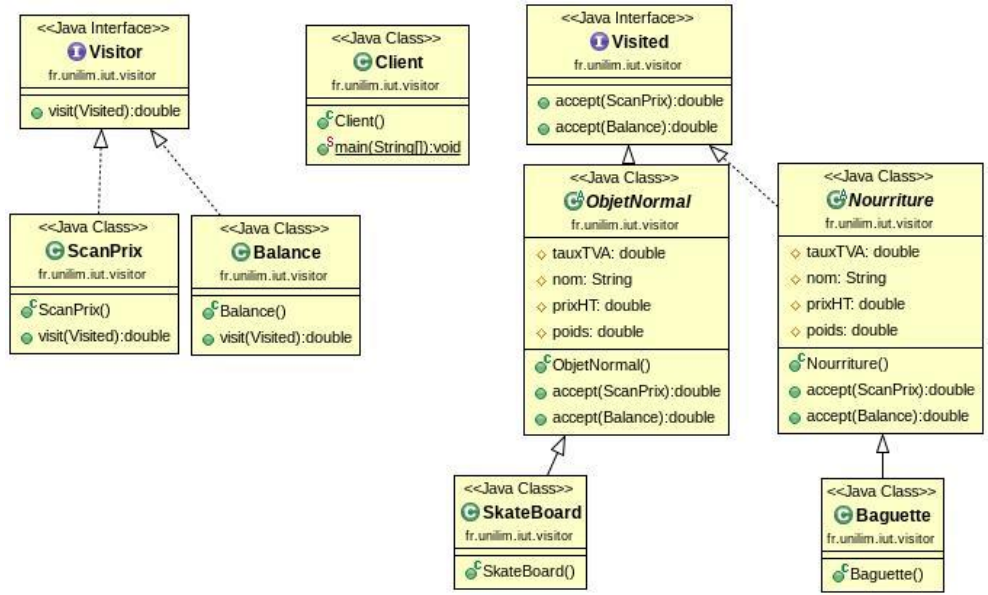
Étape 1 - Implémentation de base :



Étape 2 - Ajout d’un nouvel objet :



Étape 3 - Ajout d’un nouveau visiteur :



Liste des références utilisées

Sites internets :

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

Arkey: https://blog.arkey.fr/2010/05/06/les-visiteurs-une-question-de-nommage-et-le-double-dispatch/

Cdiese: https://cdiese.fr/design-pattern-visiteur/

Codingame: https://www.codingame.com/playgrounds/8339/design-pattern-visitor/design-pattern-visiteur

Dépôts Gits :

BingoKnight, dépot Git : https://github.com/BingoKnight/Visitor-Pattern

cpatel10 : dépot Git : https://github.com/cpatel10/patterns-VisitorAndDecorator

Liste des références utilisées

Exemples d’utilisation du pattern Visitor :

https://www.codingame.com/playgrounds/8339/design-pattern-visitor/exemple-dutilisation
https://www.baeldung.com/java-visitor-pattern
https://sourcemaking.com/design_patterns/visitor/java/1
https://www.journaldev.com/1769/visitor-design-pattern-java

QCM

Avez-vous bien suivi ?

Lien : https://tech.io/playgrounds/51758/qcm---patron-de-conception-viteur