TD - Design Pattern *Visitor*

Cet énoncé s'étend sur 9 pages.

1. Design Patterns : Elements of Reusable Object-Oriented Software

Un Design Pattern (DP) ou patron de conception est une description d'une solution éprouvée à un problème récurrent dans un contexte de génie logiciel. C'est un modèle de solution (template) décrivant comment résoudre un problème (architectural) de développement logiciel (souvent orienté-objet). Il montre une structure d'objets et les relations entre eux. Chaque implémentation de patron ne peut pas être identique puisqu'appliquée dans des contextes différents.

Il existe différents types de pattern :

— **Créateurs** : Abstract Factory, Builder, Prototype, Singleton

— **Structurels** : Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy

— **Comportementaux** : Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

2. Design Pattern Visitor

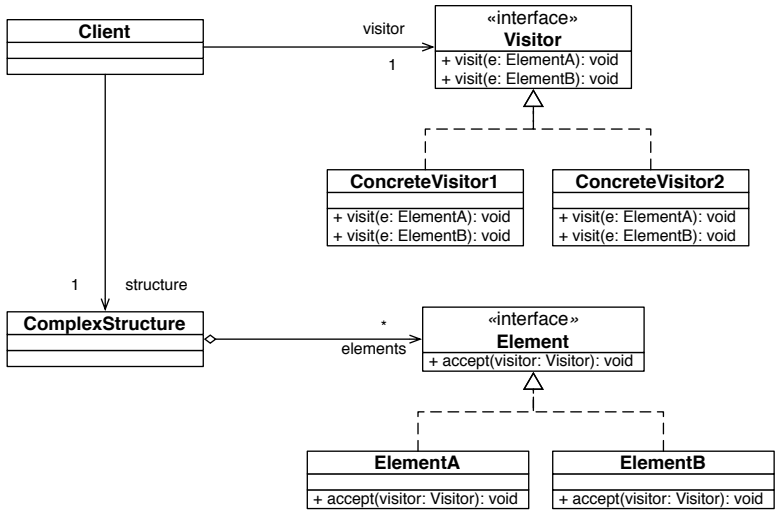

FIGURE 1 – Une description architecturale du DP Visitor

C'est un patron comportemental, dont l'architecture est décrite par le diagramme de la figure 1.

2.1. Cas d'utilisation

Ce patron est appliqué quand :

— des opérations similaires doivent être appliquées sur des objets de différents types regroupés dans une structure complexe (e.g., collection, composite) ;

— des opérations de différentes natures doivent être appliquées. Ce patron comportemental, permet de créer une classe concrète de visiteur pour chaque type d'opération et de séparer l'implémentation de la structure regroupant les objets à visiter ;

— la structure complexe regroupant les objets ne doit pas changer, mais doit probablement accueillir de nouvelles opérations. Il est aisé d'ajouter de nouveaux visiteurs (spécifiant les nouvelles opérations et leurs algorithmes) tout en gardant la structure inchangée, puisque ce patron sépare les visiteurs de la structure complexe.

2.2. Motivation

Les collections ou structures complexes (e.g., composite), largement utilisées en programmation orientée objet, peuvent contenir des objets de différents types (au sein d'une hiérarchie de classes). Dans ce cas, des opérations de différentes natures doivent être effectuées sur ces objets sans connaître leur type.

Une manière d'appliquer une opération spécifique sur chaque type d'objet dans la collection serait d'utiliser des blocs de `if` en conjonction avec l'opérateur `instanceof`. Cette approche est *overkill* : elle n'est pas flexible, alourdit la maintenance et n'est pas orientée objet.

2.3. Intention

— Spécifier une opération à appliquer sur des éléments de différents types dans une collection.

— Avec Visiteur, une nouvelle opération peut être définie sans changer les classes des éléments sur lesquels elle est appliquée.

2.4. Implémentation

Les classes participantes de ce patron de conception sont :

— **Visitor** : c'est une interface (de préférence) ou une classe abstraite utilisée pour déclarer les opérations à appliquer pour tous les types d'objets visitables. Habituellement, le nom de l'opération est le même et la distinction est faite au niveau de la signature des méthodes correspondantes. La résolution est effectuée à la compilation.

— **ConcreteVisitor** : déclare un type concret de visiteur, qui implémente toutes les méthodes de visite déclarées dans l'interface Visitor. Chaque visiteur est responsable de différentes opérations. Lorsqu'un nouveau visiteur est défini il doit être passé à la collection.

— **Visitable** : c'est une interface ou une classe abstraite qui déclare l'opération `accept`. C'est le point d'entrée qui permet à un objet d'être visité par l'objet visiteur. Chaque type d'objet dans la collection doit implémenter cette opération afin d'être visitable.

— **ConcreteVisitable** : le type concret des objets visitables. Le visiteur est passé à chaque objet via l'opération `accept`.

— **Collection** (ou structure complexe) : elle contient les objets qui peuvent être visités. Elle offre un mécanisme permettant d'itérer sur tous les éléments. Ce n'est pas pas nécessairement une collection. Cela peut être une structure complexe, comme un objet composite.

2.5. Limites

Il existe un fort couplage entre les objets à visiter et l'interface Visitor. De nouveaux visiteurs peuvent certes être facilement ajoutés à cette architecture, mais il est plus difficile d'ajouter de nouveaux objets à visiter, car il faut modifier l'interface. Cela a un impact sur tous les visiteurs concrets existant qui doivent tous implémenter la nouvelle opération...

Pour palier cet inconvénient, l'idée est de garder l'interface des visiteurs inchangée. Dans l'implémentation standard des visiteurs, la méthode à invoquer sur l'élément visité est déterminée à l'exécution. Maintenant un mécanisme appelé **réflectivité** permet de déterminer cette méthode à la compilation. Son implémentation n'est pas dans le scope de ce TD.

2.6. Exemple

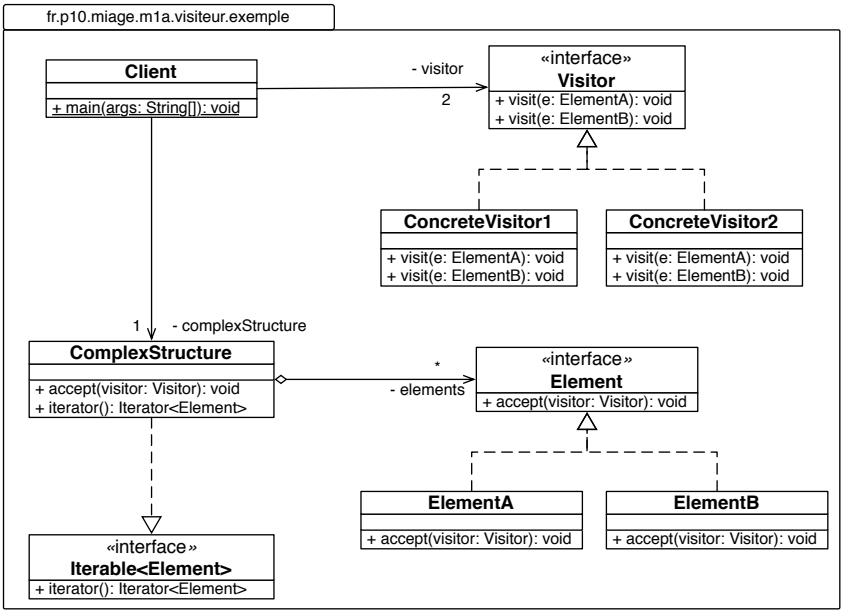

FIGURE 2 – Une instanciation du DP Visiteur : traitement des commandes de clients

La figure 2 montre un modèle pour le patron visiteur de la figure 1. Le code Java implémentant l'architecture de la figure 2 est fournie en annexe B.

2.7. Exercice

Le diagramme de la figure 3 montre une instanciation du patron visiteur. Ce diagramme modélise le concept de commande passée par des clients appartenant à des catégories de client. Chaque commande est composée de lignes de commande. Dans cette instanciation, il y a deux types concrets de visiteur, représentés par `PrintRapportCommandes` et `XMLRapportCommandes`.

Question 1. Implémenter, en Java, le modèle spécifiée dans la figure 3.

Le code, en Java, du programme principal, est fourni dans l'annexe A

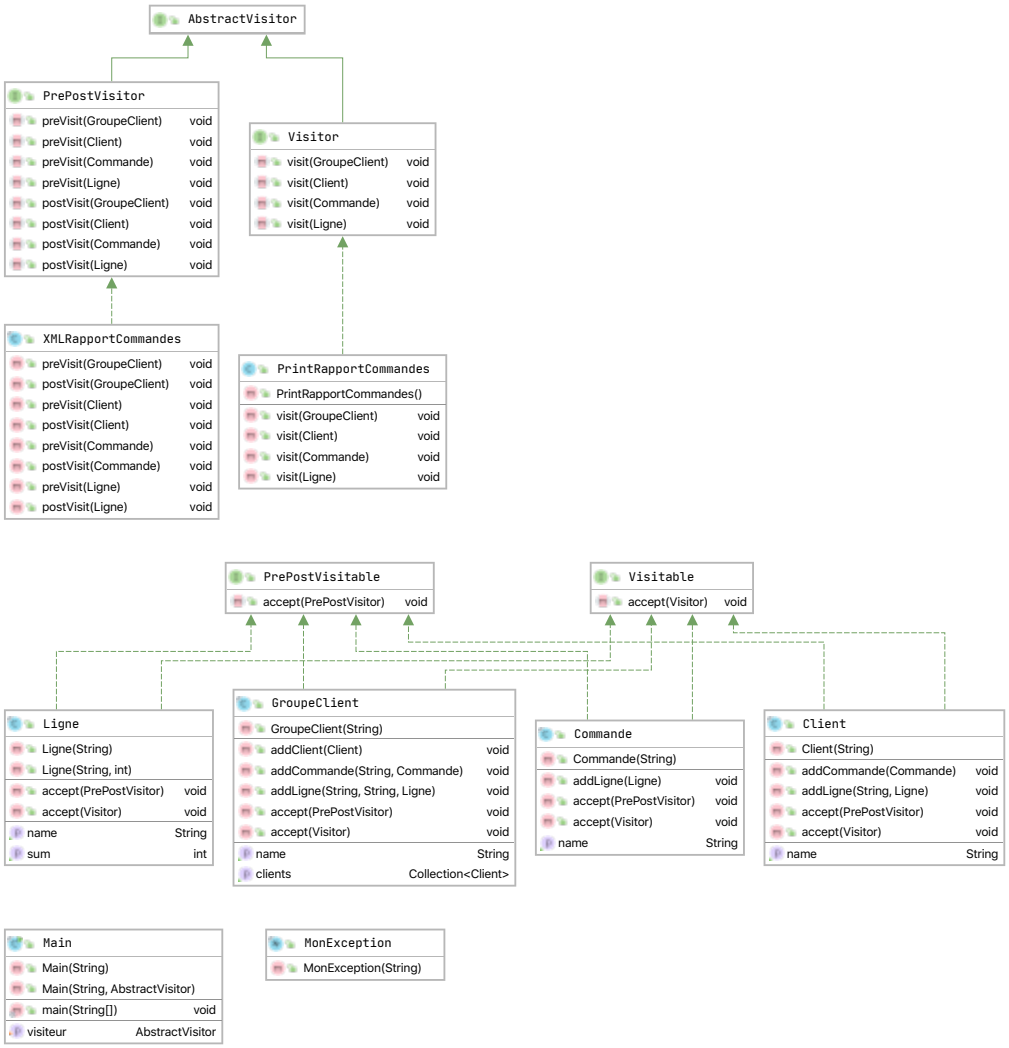

FIGURE 3 – Un instanciation du DP Visiteur : traitement de commandes par des clients

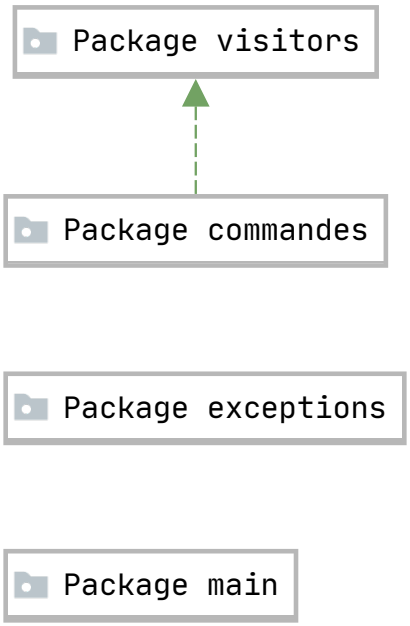

FIGURE 4 – Diagramme de package du DP visiteur de la figure 3

2.8. Exercice supplémentaire

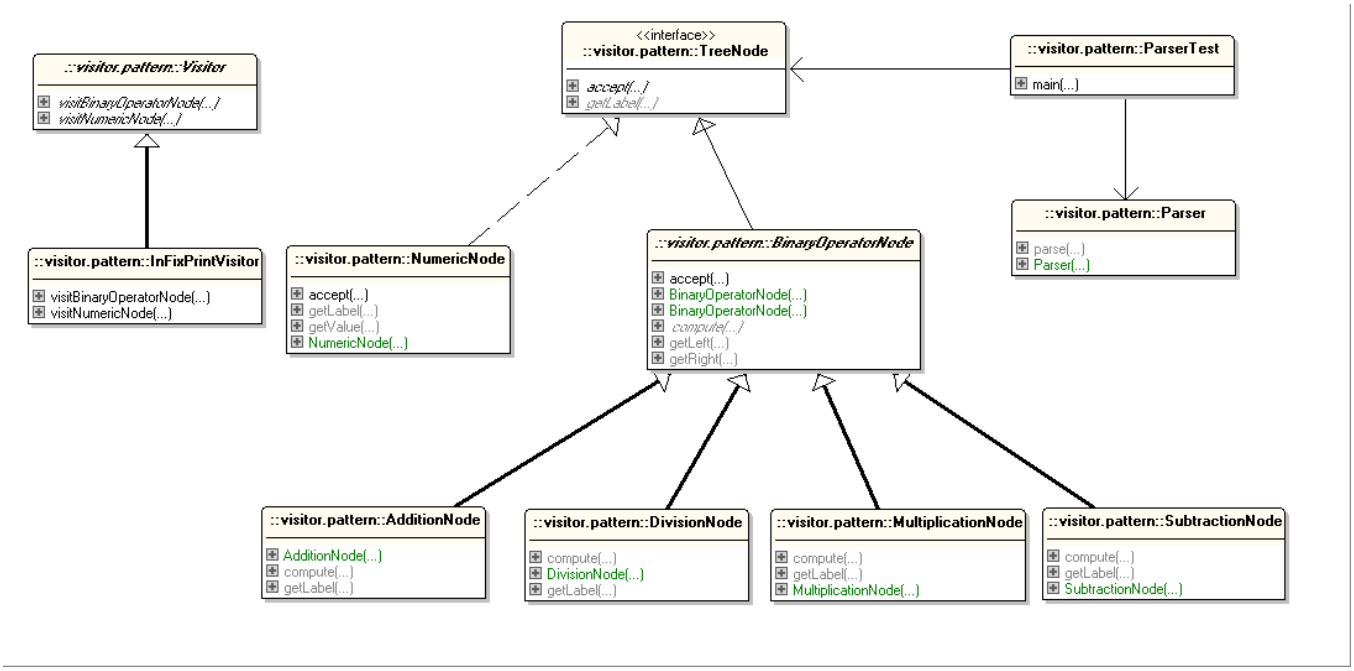

FIGURE 5 – Une instanciation du DP Visiteur : arbre d'une expression arithmétique

Le diagramme de la figure 5 montre l'arbre de syntaxe abstraite d'une expression arithmétique [1]. Le code de parcours de l'arbre relève du visiteur. Il est donc possible d'écrire différents visiteurs pour réaliser différentes tâches sur cet arbre, chaque visiteur définissant sa procédure de parcours, par exemple :

— l'affichage en notation prefix

— l'affichage en notation infix,

— l'affichage en notation postfix,

— l'évaluation de l'expression, etc.

Cependant, ce modèle ne supporte pas bien l'ajout d'un nouveau type de noeud dans l'arbre, par exemple un nouvel opérateur (impact immédiat sur tous les visiteurs, qui doivent prendre en compte le nouveau type de noeud, s'ils dépendent du type de noeud dans leurs parcours.)

Question 2. Quel design pattern serait le plus adapté à une extension de ce modèle avec un nouvel opérateur (e.g. l'opérateur modulo) ?

Si l'on déplace la logique de parcours dans l'arbre, i.e. la méthode `accept`, les visiteurs spécifiques risquent de ne plus fonctionner correctement (voire plus du tout), puisqu'une seule logique de parcours sera imposée. Dans ce cas, il ne sera possible d'écrire que des visiteurs réalisant une tâche compatible avec (ou

1. Source : http ://www.csi.ucd.ie/staff/meloc/DesignPatterns/Practicals/Visitor.htm

indépendante de) cette logique imposée (par exemple, compter le nombre de noeuds).

Question 3. Proposez, en Java, un type d'affichage (au choix) et l'évaluation d'une expression arithmétique respectant le modèle spécifiée dans la figure 5 selon le design pattern Visiteur.

2.9. Exercice supplémentaire

En utilisant le patron Visiteur, écrire un parcours d'arbre en *largeur et profondeur* pour chercher une valeur. On vous suggère la signature de méthode suivante :

— `visitTree (Tree t, List<Tree> trees, int i) : Tree`, où `i` est la valeur à chercher, `trees` est la liste des prochains éléments (sous-arbre) à parcourir et `t` est l'élément courant à examiner.

Écrire un parcours d'arbre pour récupérer les listes des éléments pairs et impairs. On vous suggère les signatures de méthodes suivantes :

— `visitPartition(Tree t, List<Tree> trees) : [List<Tree>, List<Tree>]`, renvoie le couple des éléments pairs et impairs, ou

— `visitPartition(Tree t, List<Tree> trees, List<Tree> pairs, List<Tree> impairs)`, où les deux derniers paramètres sont IN/OUT.

Annexe

A. Code Java du main de l'exercice 2.7

```
1  public final class Main {
     private final GroupeClient groupeClient;
3    private AbstractVisitor visiteur;

5    public Main(String nomGroupe) {
       this.groupeClient = new GroupeClient(nomGroupe);
7    }

9    public Main(String nomGroupe, AbstractVisitor v) {
       this(nomGroupe);
11     this.visiteur = v;
     }
13
     public void setVisiteur(AbstractVisitor v) {
15     this.visiteur = v;
     }
17
     public static void main(String[] args) throws MonException {
19     AbstractVisitor xmlVisitor = new XMLRapportCommandes();
       AbstractVisitor printVisitor = new PrintRapportCommandes();
21     //
       Main m = new Main("clients");
23     //
       Client c1 = new Client("bob");
25     Client c2 = new Client("joe");
       m.groupeClient.addClient(c1);
27     m.groupeClient.addClient(c2);
       //
29     Commande cde1 = new Commande("cde1");
       Commande cde2 = new Commande("cde2");
31     Commande cde3 = new Commande("cde3");
       m.groupeClient.addCommande("bob",cde1);
33     m.groupeClient.addCommande("bob", cde2);
       m.groupeClient.addCommande("joe", cde3);
35     //
       Ligne l1 = new Ligne("l1",100);
37     Ligne l2 = new Ligne("l2",200);
       Ligne l3 = new Ligne("l3",400);
39     Ligne l4 = new Ligne("l4",800);
       m.groupeClient.addLigne("bob", "cde1", l1);
41     m.groupeClient.addLigne("bob", "cde1", l2);
       m.groupeClient.addLigne("bob", "cde2", l3);
43     m.groupeClient.addLigne("joe", "cde3", l4);
       //
45     m.setVisiteur(printVisitor);
       m.groupeClient.accept((Visitor)m.visiteur);
47     //
       m.setVisiteur(xmlVisitor);
49     m.groupeClient.accept((PrePostVisitor)m.visiteur);
     }
51 }
```

La sortie de ce main est :

```
Le client joe doit 800 euros.
```

```
Le client bob doit 700 euros.
<groupe>
<client name="joe">
<commande name="cde3">
<ligne name="l4" sum=800 />
</commande>
</client>
<client name="bob">
<commande name="cde2">
<ligne name="l3" sum=400 />
</commande>
<commande name="cde1">
<ligne name="l1" sum=100 />
<ligne name="l2" sum=200 />
</commande>
</client>
</groupe>
```

B. Code Java implémentant le patron visitor de la figure 1

Listing 1 – Implémentation de Visiteur

```
1  // 1. Accepter les visiteurs.
   interface Element {
3     // dispatch
      void accept(Visitor visitor);
5  }
   class ElementA implements Element {
7     // accept(Visitor) implementation
      @Override
9     public void accept(Visitor visitor) {
        visitor.visit(this);
11    }
      @Override
13    public String toString() {
        return "Element of type ElementA";
15    }
   }
17 class ElementB implements Element {
      @Override
19    public void accept(Visitor visitor) {
        visitor.visit(this);
21    }
      @Override
23    public String toString() {
        return "Element of type ElementB";
25    }
   }
27
   // 2. Creer un type de base pour les visiteurs avec une methode de visite pour chaque type
     d'element a visiter.
29 interface Visitor {
      // deuxieme dispatch
31    void visit(ElementA a);
      void visit(ElementB b);
33 }

35 // 3. Creer un visiteur concret pour chaque operation a appliquer sur les elements.
   class ConcreteVisitor1 implements Visitor {
```

```
37    @Override
      public void visit(ElementA a) {
39       System.out.println("Visitor 1 visiting " + a.toString());
      }
41    @Override
      public void visit(ElementB b) {
43       System.out.println( "Visitor 1 visiting " + b.toString());
      }
45 }
   class ConcreteVisitor2 implements Visitor {
47    @Override
      public void visit(ElementA a) {
49       System.out.println("Visitor 2 visiting " + a.toString());
      }
51    @Override
      public void visit(ElementB b) {
53       System.out.println( "Visitor 2 visiting " + b.toString());
      }
55 }

57 // 4. Creer la structure complexe contenant les elements visitables.
   class ComplexStructure implements Iterable<Element> {
59    private List<Element> elements = new ArrayList<Element>();
      @Override
61    public Iterator<Element> iterator() {
         return this.elements.iterator();
63    }
      public void addElement(Element e){
65       this.elements.add(e);
      }
67 }

69 //5. Creer le client.
   class Client {
71    // 5. Creer les visiteurs et les passer a accept.
      public static void main( String[] args ) {
73       Element a = new ElementA();
         Element b = new ElementB();
75       ComplexStructure cs = new ComplexStructure();
         cs.addElement(a); cs.addElement(b);
77       Visitor v1 = new ConcreteVisitor1();
         Visitor v2 = new ConcreteVisitor2();
79       for (Iterator<Element> it = cs.iterator(); it.hasNext();){
            Element e = it.next();
81          e.accept(v1);
            e.accept(v2);
83
          }
85    }
   }
```