La Rochelle Université

La Rochelle Université
Licence Informatique
UE Génie logiciel 2

TD 2 - Design pattern : Visiteur

Version enseignant

1 Introduction

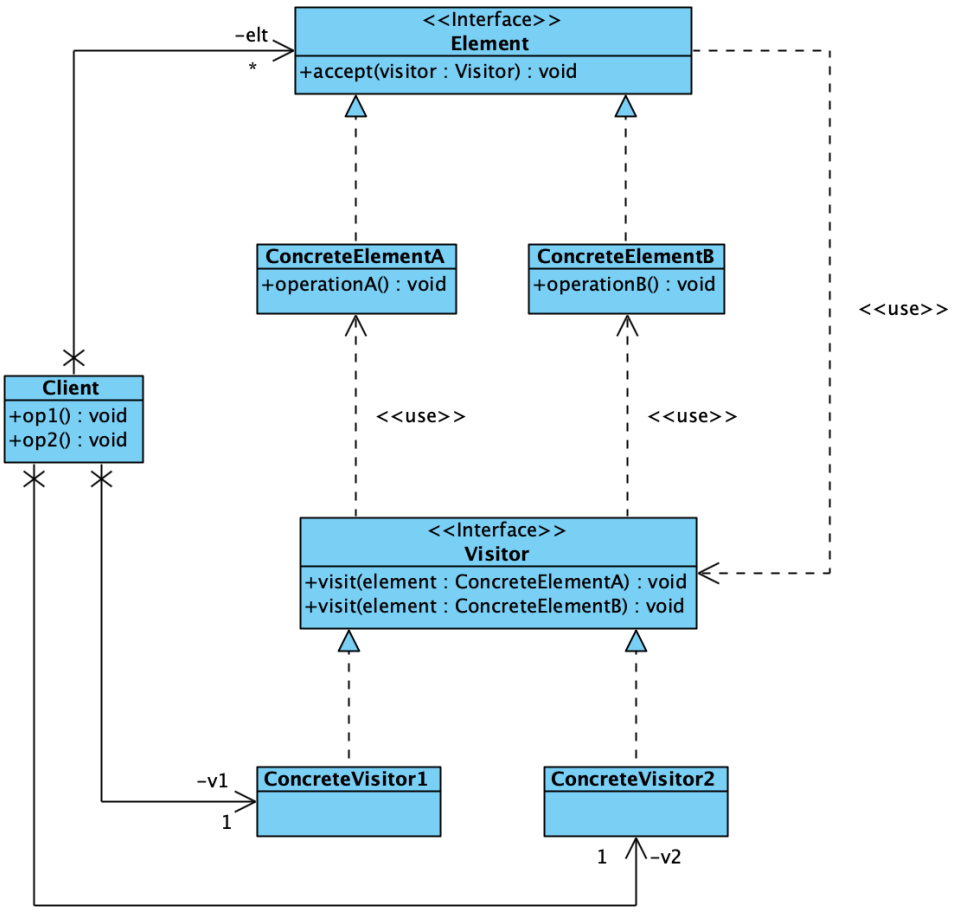

FIGURE 1 – Diagramme de classes type du design pattern Visitor

Le design pattern *Visiteur* (figure 1) permet de limiter le couplage entre les données et leurs traitements. Ces derniers sont implémentés dans des classes externes ce qui permet d'ajouter de nouveaux traitements aux structures sans avoir à les modifier. La classe "visitable" *Element* (qui représente la structure de données) possède une méthode *accept()* qui permet d'injecter le visiteur (le type de visiteur accepté est passé en paramètre de cette méthode). Les

visiteurs concrets implémentent une interface qui contient une méthode *visit()* par type d'objet "visitable". Dans l'objet "visitable" la méthode *visit()* du visiteur est appelé en lui passant l'objet lui- même.

Vous trouverez ci-dessous des extraits du code Java des classes *Client, ConcreteElementA* et *ConcreteVisitor1*.

```
1  public class Client
2  {
3     private List<Element> elt;
4     private ConcreteVisitor1 v1;
5     private ConcreteVisitor2 v2;
6
7     ...
8
9     public void op1()
10    {
11       for(Element element : this.elt)
12          element.accept(v1);
13    }
14 }
15
16 public class ConcreteElementA implements Element
17 {
18    @Override
19    public void accept(Visitor visitor)
20    {
21       visitor.visit(this);
22    }
23
24    public void operationA()
25    {
26       System.out.println("Traitement de ConcreteElementA dans operationA()");
27    }
28 }
29
30 public class ConcreteVisitor1 implements Visitor
31 {
32    @Override
33    public void visit(ConcreteElementA element)
34    {
35       element.operationA();
36    }
37
38    @Override
39    public void visit(ConcreteElementB element)
40    {
41       element.operationB();
42    }
43 }
```

2 Révision : le diagramme de classes

Soit le code présenté en annexe et implémentant le design pattern du Visitor.

— Donnez le diagramme de classes complet correspondant à ce code.

— Quelles classes correspondent aux classes *Client, ConcreteVisitor* et *ConcreteElement* par rapport au diagramme de classes de la section 1 ?

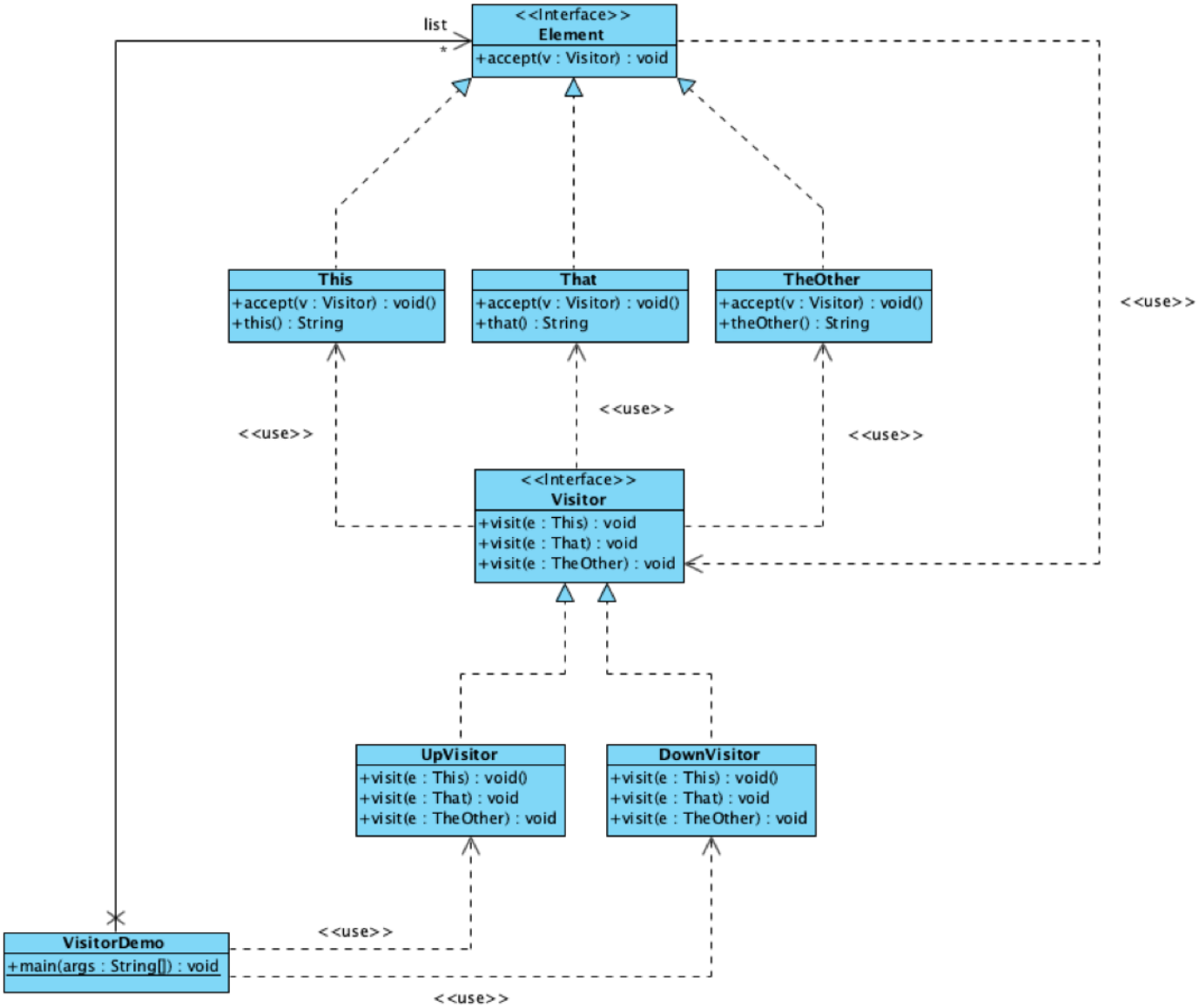

FIGURE 2 – Solution de l'exercice 2

Correspondance
Client <=> VisitorDemo
ConcreteVisitor <=> UpVisitor, DownVisitor
ConcreteElement <=> This, That, TheOther

3 Révision : le diagramme de séquences

Soit le code présenté en annexe et implémentant le design pattern du Visitor.

— Donnez le diagramme de séquences correspondant à l'exécution de ce code.

— Quelle est la sortie produite par la méthode *main*?

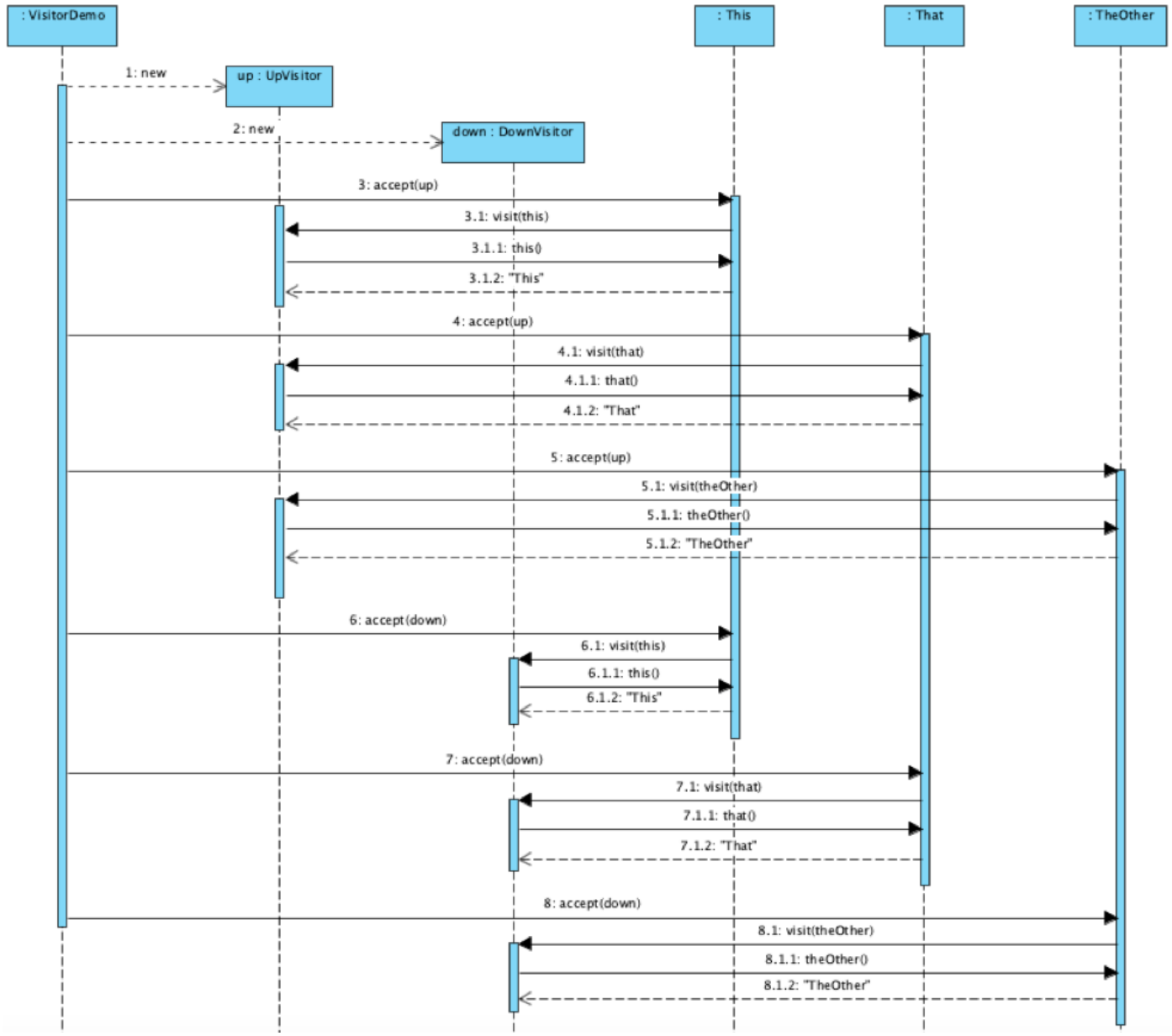

FIGURE 3 – Solution de l'exercice 3

Sortie console

```
do Up on This
do Up on That
do Up on TheOther
do Down on This
do Down on That
do Down on TheOther
```

Annexe

```
1  public interface Element {
2     public void accept(Visitor v);
3  }
4
5  public class This implements Element {
6     public void accept(Visitor v)
7     {
8        v.visit(this);
9     }
10
11    public String thiss()
12    {
13       return "This";
14    }
15 }
16
17 public class That implements Element {
18    public void accept(Visitor v)
19    {
20       v.visit(this);
21    }
22
23    public String that()
24    {
25       return "That";
26    }
27 }
28
29 public class TheOther implements Element {
30    public void accept(Visitor v)
31    {
32       v.visit(this);
33    }
34
35    public String theOther()
36    {
37       return "TheOther";
38    }
39 }
40
41 public interface Visitor {
42    public void visit(This e);
43    public void visit(That e);
44    public void visit(TheOther e);
45 }
46
47 public class UpVisitor implements Visitor {
48    public void visit(This e)
49    {
50       System.out.println("do Up on " + e.thiss());
51    }
52
53    public void visit(That e)
54    {
```

```
55          System.out.println("do Up on " + e.that());
56      }
57
58      public void visit(TheOther e)
59      {
60          System.out.println("do Up on " + e.theOther());
61      }
62  }
63
64  public class DownVisitor implements Visitor {
65      public void visit(This e)
66      {
67          System.out.println("do Down on " + e.thiss());
68      }
69
70      public void visit(That e)
71      {
72          System.out.println("do Down on " + e.that());
73      }
74
75      public void visit(TheOther e)
76      {
77          System.out.println("do Down on " + e.theOther());
78      }
79  }
80
81  public class VisitorDemo {
82      public static Element[] list = {new This(), new That(), new TheOther()};
83
84      public static void main(String[] args)
85      {
86          UpVisitor up = new UpVisitor();
87          DownVisitor down = new DownVisitor();
88          for(Element elt : list)
89              elt.accept(up);
90
91          for(Element elt : list)
92              elt.accept(down);
93       }
94  }
```

4 Refactoring

On cherche à remanier la structure présentée dans le diagramme de classes de la figure 4 pour utiliser le design pattern *Visiteur*.

— Modifier le diagramme de classes pour réaliser les traitements d'affichage et de calcul d'aire sur les figures (Composite, Circle, Rectangle, Square) en ajoutant le pattern *Visiteur*.

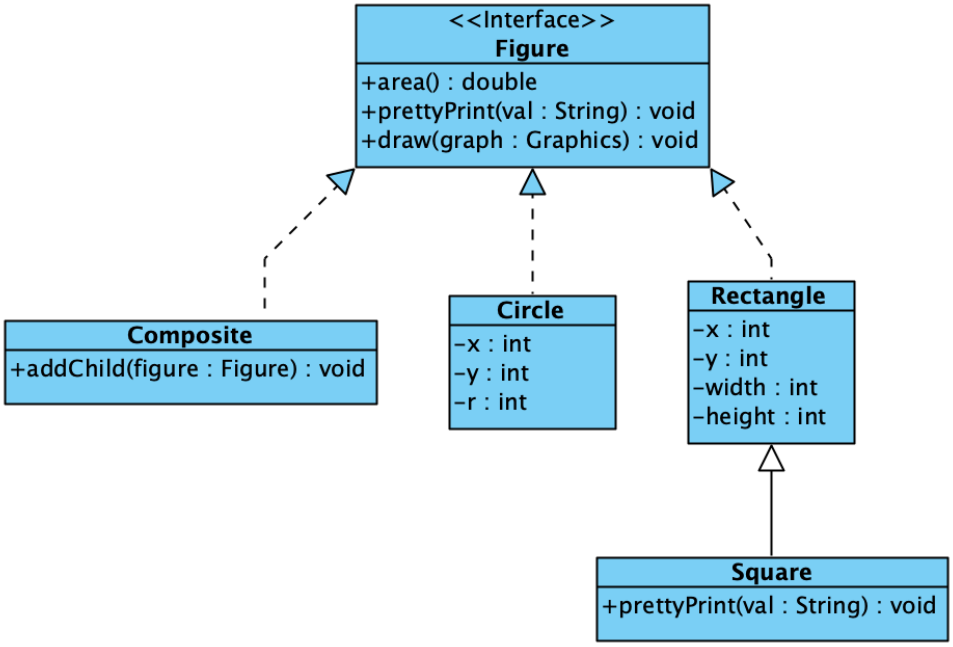

FIGURE 4 – Diagramme de classes à modifier

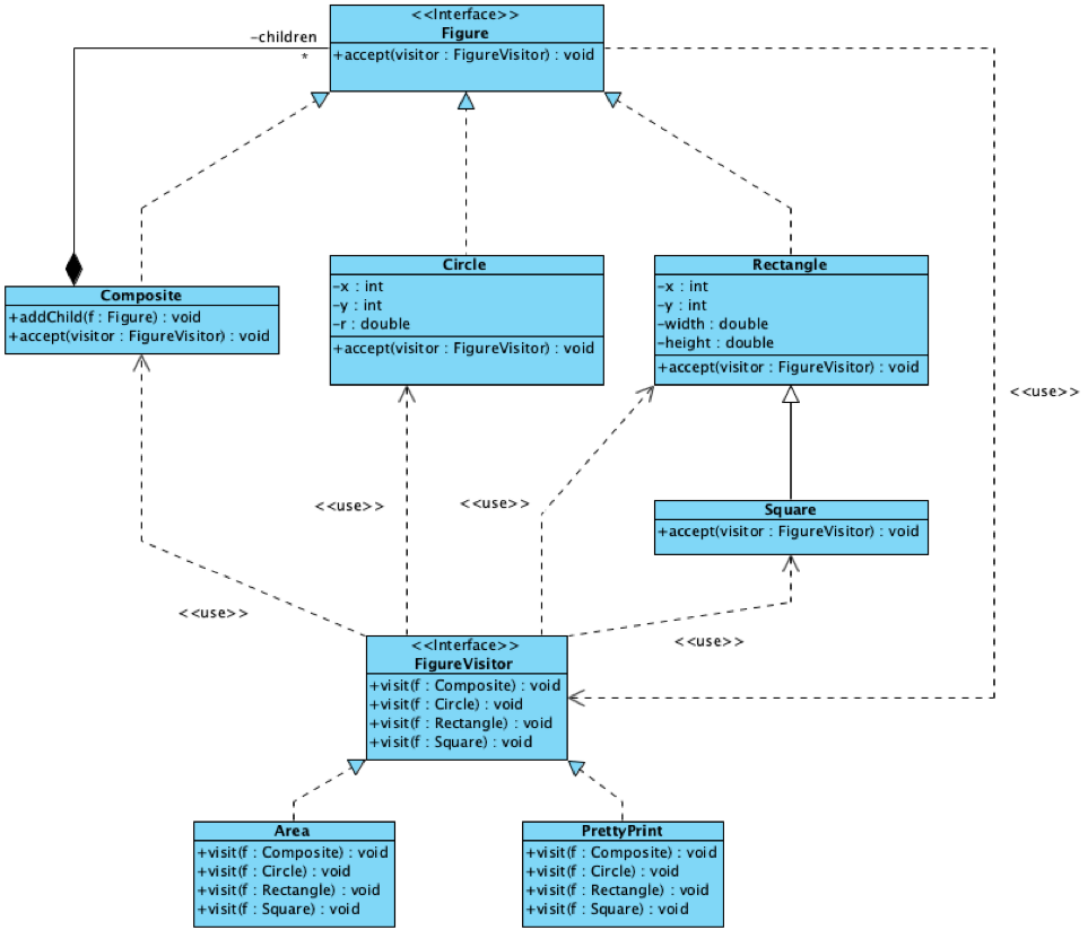

FIGURE 5 – Solution de l'exercice 4