Le patron de conception Visiteur

Adrien CAUBEL

25 décembre 2021

Table des matières

Introduction

Nous vous proposons ici d'étudier le patron de conception Visiteur. Une première partie sera consacrée à l'étude de la de résolution dynamique des types en Java tout en proposant une solution intéressante introduisant le patron de conception Visiteur. Dans les partie suivantes, nous analyserons à travers différents exemple comment utiliser le patron Visiteur. Puis, nous finirons sa structure générale.

Première partie

Le problème de la résolution dynamique de types

Cette première partie présente brièvement la résolution dynamique de types en Java. Cette notion fait partie de branche du polymorphisme en orienté objet où nous souhaitons choisir la bonne méthode à appeler suivant le type dynamique de l'objet ainsi que les types statiques des paramètres.

Vous pouvez approfondir cette notion importante en vous références aux ressources en annexe du document [1].

1 Exemple

Pour mieux comprendre le problème de la répartition dynamique, regardons ensemble un exemple. Nous souhaitons créer une application où un vétérinaire peut soigner plusieurs animaux en même temps.

```
public Interface Animal {
   public void crier();
}

public Chien implements Animal {
   @Override
   public void crier() { System.out.println("Wouaf"); }
}

public Vache implements Animal {
   @Override
   public void crier() { System.out.println("Meuh"); }
}
```

```
public class Veterinaire {
   void parcourir(List<Animal> animals) {
      for(Animal a : animals) {
         a.crier(); // 1ère opération
      }
   }
}
```

```
main() {
   List<Animal> animals = new ArrayList<>();
   animals.add(new Chien());
   animals.add(new Vache());

   Veterinaire v = new Veterinaire();
   v.parcourir(animals);
}
```

```
OUTPUT :
Wouaf
Meuh
```

Le résultat de l'exécution de l'application est cohérent. En effet, à l'exécution la machine virtuelle exécute la méthode de l'objet qui a la bonne signature. Parmi toutes les redéfinitions situées au dessus du type dynamique de l'objet, ce sera celle la plus proche du type dynamique de l'objet. Ce mécanisme s'appelle la **liaison** dynamique.

Mais que se passe-t-il si nous décidons d'ajouter une méthode *opération* pour que notre vétérinaire puisse opérer un animal suivant les caractéristiques (le type) de l'animal. Nous rajoutons donc de surcharger la `operer()` dans `Veterinaire`.

```
public class Veterinaire {

   void parcourir(List<Animal> animals) {
      for(Animal a : animals) {
         a.crier(); // 1ère opération
         operer(a); // 2ème opération
      }
   }

   void operer(Chien c) {
      System.out.println("operation du chien");
   }

   void operer(Vache v) {
      System.out.println("operation de la vache");
   }

   // Obligatoire car la 2ème opération prend un type animal
   void operer(Animal a) {
      System.out.println("??");
   }
}
```

Le résultat obtenu en sortie de l'application est à gauche tandis que le résultat voulut à droite.

```
OUTPUT à l'éxécution :
Wouaf
??
Meuh
??
```

```
OUTPUT entendu :
Wouaf
operation du chien
Meuh
operation de la vache
```

Le résultat semble surprenant mais s'explique sans grade difficulté :

— Java peut déterminer dynamiquement le type de l'appelant. C'est-à-dire que lorsqu'on exécute le code `a.crier()` le type de `a` a été résolu dynamiquement à l'exécution. La première fois `a` est de type `Chien` et la seconde fois il est de type `Vache`.

— Mais, Java n'est pas capable de déduire dynamiquement le type des arguments des méthodes. Donc au moment de l'appel `operer(a)` la variable `a` est de type `Animal`. Cela explique donc l'affichage `??`.
C'est pour cela que nous disons que le langage Java n'est pas capable de faire du *double dispatch* [2]

2 Les solutions

2.1 Le mot-clé instanceof

Une première solution consiste à utiliser le mot-clé `instanceof` fourni par Java pour déterminer le type dynamique de l'objet `a` à l'exécution.

```
public class Veterinaire {
   for(Animal a : animals) {
      a.crier(); // 1ère opération

      if(a instanceof Chien) {
         operer((Chien) a); // 2ème opération
      } else if(a instanceof Vache) {
         operer((Vache) a); // 2ème opération
      }
   }
}
```

Nous pouvons noter que cette solution fonctionne parfaitement. Cependant elle est révélatrice d'un problème de conception.

2.2 Une seconde solution

Pour éviter un enchainement de structure conditionnelle, nous devons repenser la conception de notre application et repartir du problème initial : la liaison dynamique ne marque pas en argument. Comme évoqué au début de cette première partie, le polymorphisme permet de choisir la bonne méthode suivant le type dynamique et l'objet et suivant *types statiques des paramètres*. Or, dans l'exemple non fonctionnel lorsque nous réalisons l'appel `operer(a)` nous appelons `operer()` avec le type statique de `a`. Donc la méthode avec la signature `operer(Animal a)`.
Afin de pouvoir réaliser le bon appel, nous devons inverser l'objet appelant et nous assurer que ce soit l'objet `a` à l'origine de l'appel (étant donné que nous connaissons le type dynamique de l'objet à l'exécution). On souhaite ainsi dire que l'animal est opéré par le vétérinaire. Cela peut se traduit sous la forme de `a.opererPar(leVeterinaire)`.

Cette nouvelle opération implique plusieurs changements dans notre application :

- Nous devons premièrement ajouter une méthode `void opererPar(Veterinaire v)` dans l'interface `Animal`.
- Puis dans cette méthode `opererPar` nous devons réaliser un appel à la méthode `operer(T t)` du vétérinaire en précisant le type de l'animal.

2.2.1 Réécriture de l'interface et des implémentations

```
public interface Animal {
   void crier();
   void opererPar(Veterinaire v);
}

public class Chien implements Animal {
   @Override
   public void crier() {
      System.out.println("chien");
   }
```

```
    @Override
    public void opererPar(Veterinaire v) {
        // C'est le vétérinaire qui opère l'animal => v.operer(T t)
        // Ici on veut appeler operer(Chien c), or ici T == Chien => this
        // car nous savons qu'ici this == Chien
        v.operer(this);
    }
}

public class Vache implements Animal {
    ...
    @Override
    public void opererPar(Veterinaire v) {
        // C'est le vétérinaire qui opère l'animal => v.operer(T t)
        // Ici on veut appeler operer(Vache v), or ici T == Vache => this
        // car nous savons qu'ici this == Vache
        v.operer(this);
    }
}
```

Remarquez comment sont implémentées les méthodes `opererPar` :

1. Qui doit appeler la méthode `operer(T t)` ? Le vétérinaire `v`.
2. Comment connaitre le type `T` pour appeler la bonne méthode ? `this`.
   — Si nous sommes dans la classe `Chien` alors `this` de type statique `Chien` donc `T` aussi.
   — Si nous sommes dans la classe `Vache` alors `this` de type statique `Vache` donc `T` aussi.

Nous avons donc `v.operer(this)`.

2.2.2 Réécriture de la classe Veterinaire

```
public class Veterinaire {
    void parcourir(List<Animal> animals) {
        for(Animal a : animals) {
            a.crier(); // 1ère opération
            a.opererPar(this); // 2ème opération; leVeterinaire == this
        }
    }

    void operer(Chien c) { System.out.println("operation du chien"); }

    void operer(Vache v) { System.out.println("operation de la vache"); }
}
```

Remarquez comment nous appelons l'animal opéré :

1. Comment déterminer la bonne méthode `opererPar()` ? via la liaison dynamique sur un objet (ici `a`)
   — Si `a` de type dynamique `Chien` alors `chien#opererPar()`.
   — Si `a` de type dynamique `Vache` alors `vache#opererPar()`.
2. Que prend en paramètre la méthode `opererPar()` ? Le vétérinaire `this`.

Nous avons donc `a.opererPar(this)`.

2.2.3 Déroulement des opérations

Le fait de faire des allers-retours entre les classes via les appels à `this` peut être assez perturbante. Nous revenons donc ici sur l'enchainement des appels.

1. Le client lance l'application est appelle `v.parcourir(animals)`
2. La boucle `for` de `parcourirl` réalise les opérations
   (a) `a.crier()` qui est un appel classique (appel dynamique de la bonne méthode)
   (b) `a.opererPar(this)` qui est aussi un appel dynamique pour appeler la bonne méthode
   — car nous ne pouvons pas connaitre le type de `a` si celui-ci est passé en argument. Nous l'avons donc passé en appelant.
   — `this` ici est égal à l'objet courant de `Veterinaire`.
3. La méthode `opererPar()` est maintenant choisie dynamiquement. Ainsi, si :
   — `a` est de type `Chien` alors ça sera la méthode `opererPar()` de `Chien` qui sera appelée.
   — `a` est de type `Vache` alors ça sera la méthode `opererPar()` de `Vache` qui sera appelée.
4. Dans le corps de `opererPar(Veterinaire v)` nous réalisons l'appel `v.operer(this)` qui va appeler la méthode `operer()` de `Veterinaire` avec le bon type statique en paramètre. Ainsi si nous sommes dans la classe :
   — `Chien` alors `this` est de type `Chien` donc la méthode appelée est `operer(Chient c)`.
   — `vache` alors `this` est de type `vache` donc la méthode appelée est `operer(Vache c)`.

2.2.4 Conclusion

Cette solution est certes plus compliquée à comprendre que la première et nécessite de passer plus de temps dessus. Mais en contrepartie elle est très efficace et nous permettra de mieux comprendre le patron Visiteur. Car sans le dire, nous avons comblé l'absence de *double-dispatch [2]* (en Java) qui est centrale dans le patron Visiteur.

2.2.5 Java language spefication

Nous ouvrons ici une petite parenthèse pour citer les spécifications Java [3] concernant les liaisons dynamiques et statiques

*When a method is invoked (§15.12), the number of actual arguments (and any explicit type arguments) and **the compile-time types of the arguments are used**, at compile time, to determine the signature of the method that will be invoked (§15.12.2).*

`methode(T t)` prendre le type statique (= type à la compilation) de l'argument `t`

*If the method that is to be invoked is an **instance method**, the actual **method to be invoked will be determined at run time**, using dynamic method lookup (§15.12.4).*

`objet.methode()` prendre le type dynamique de l'objet `objet`

Deuxième partie
Le patron de conception Visiteur

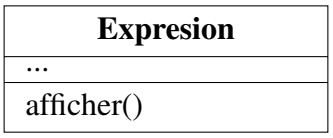
3 Le concept du patron Visiteur

3.1 Définition

Le patron Visiteur est rangé avec les patrons comportementaux. Ce dernier nous permet de séparer les algorithmes et les objets sur lesquels ils opèrent.

3.1.1 Interprétation de la définition

Séparer les algorithmes et les objets sur lesquels ils opèrent traditionnellement, pour une classe donnée, plusieurs algorithmes sont définis via des méthodes. Le patron Visiteur suggère de déplacer ces méthodes (donc les algorithmes) dans une classe différente. Par exemple, si nous souhaitons pouvoir afficher une expression arithmétique. Nous aurions classiquement.



Le patron Visiteur quant à lui suggère de séparer les méthodes de l'objet et ainsi avoir deux classes séparées.

Nous décrirons dans les sections suivantes comment cela s'harmonise.

4 Exemple d'une expression arithmétique

Pour mieux comprendre comment réaliser la séparation, nous allons nous appuyer les exemples d'une expression arithmétique [1] et d'une exportation XML.

4.1 Définition du problème

Nous souhaitons un outil capable d'analyser une expression arithmétique composée de constantes, d'inconnues (x, y, ...) et d'opérateurs afin de pouvoir l'afficher, la calculer ou faire tout autre traitement sur cette expression. À noter également qu'une expression arithmétique peut également être composée d'une autre expression arithmétique.

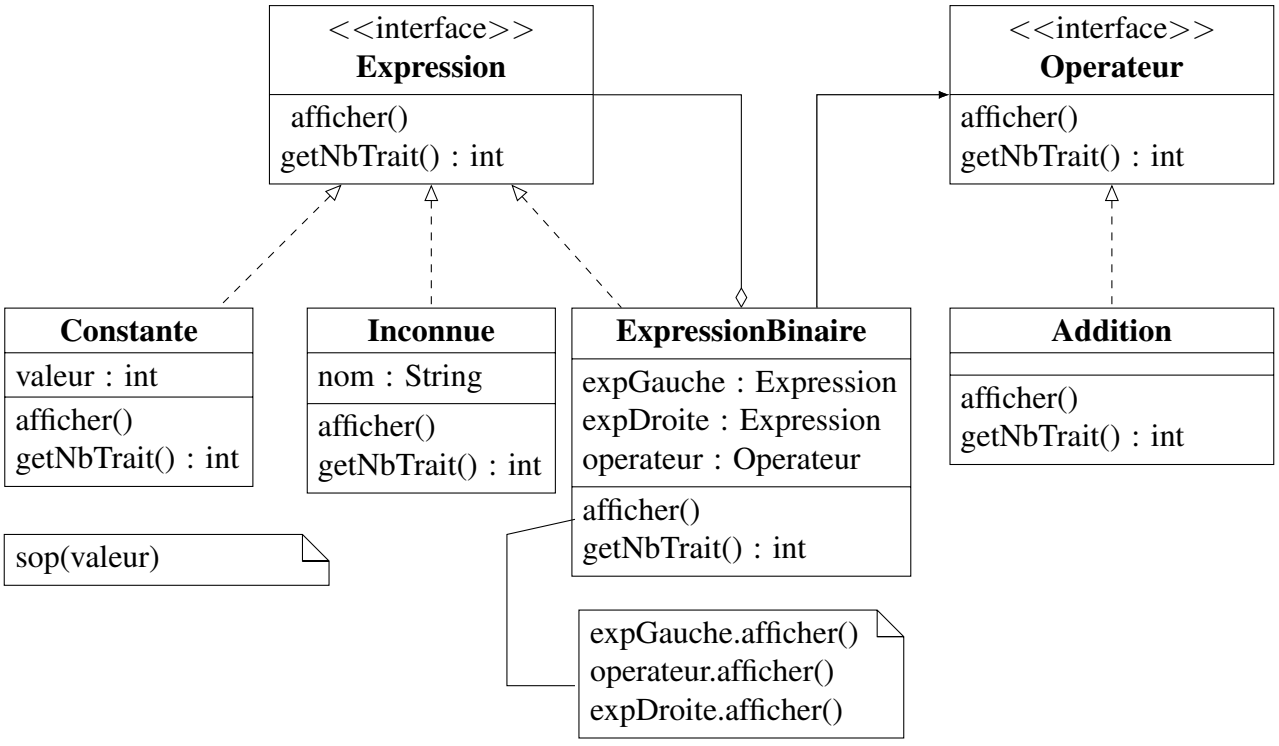
4.2 Une première solution

D'après l'énoncé, on peut définir plusieurs classes pour notre application. Ainsi, on aura `Constante`, `Inconnue` et `ExpressionBinaire` qui représente une expression de la forme $5+y$ par exemple. L'énoncé précise qu'une expression peut être composée elle-même d'une autre expression. Cela fait apparaitre la notion d'une interface commune pour les 3 classes définies précédemment. De plus, nous aurons besoin d'opérateurs. Pour le moment nous nous contenterons de `Addition` et `Multiplication`.

Ensuite, on souhaite pouvoir afficher l'expression et compter le nombre de traits dans une expression (cela est simplement à but éducatif).

- l'addition "+" = 2 traits
- la multiplication "∗" = 3 traits
- le reste = 0 trait

Pour se faire, nous devons définir les méthodes `afficher()` et `getNbTrait()` dans nos interfaces puis réaliser des implémentations pour chacune des classes.



1. Exemple issu d'un des cours du professeur Xavier Cregut [4]

Cette première architecture peut être satisfaisante, car nous spécifions une interface et que nous implémentons ensuite pour nos différentes formes d'expression (*Patron Interpreteur [5]*). Mais nous pouvons également apporter des critiques à cette conception.

4.2.1 Critiques de la solution

Cette solution bien que pouvant être satisfaisante se heurte à plusieurs problèmes

- Le traitement est éparpillé dans l'ensemble du diagramme de classe ce qui complique la compréhension et la maintenance de programme.
- Chaque classe est polluée par chaque traitement. En effet, au début nous avons une classe `Constante`. Si nous continuons à rajouter des trainements est-ce qu'on peut encore parler de constante (voir le *Principe de responsabilité unique*) [6] ?
- Pour ajouter un nouveau traitement, cela nécessite de modifier toutes les classes ce qui supprime de l'extensibilité.

4.3 Solution avec application du patron Visiteur

Le patron Visiteur nous forcer à *séparer les algorithmes et les objets sur lesquels ils opèrent.* Et comme présenté dans la partie 3.1.1 nous devons extraire les méthodes pour en créer des classes à part. Pour notre exemple, nous obtenons donc les deux classes `Afficheur` et `CompterTrait`.

Afficheur
afficher(e : Constante)
afficher(e : Inconnue)
afficher(e : ExpressionBinaire)
afficher(e : Addition)
afficher(e : Multiplication)

CompterTrait
getNbTrait(e : Constante) : int (0)
getNbTrait(e : Inconnue) : int (0)
getNbTrait(e : ExpressionBinaire) : int (0)
getNbTrait(e : Addition) : int (2)
getNbTrait(e : Multiplication) : int (3)

Et dans le diagramme de classe précédent, nous pouvons supprimer les méthodes `afficher()` et `getNbTrait()` dans toutes les interfaces et classes.

4.3.1 Difficulté pour afficher(e : ExpressionBinaire)

Si nous essayons d'implémenter la méthode `afficher(e : ExpressionBinaire)` comme précédemment nous allons nous heurter à un problème.

```
afficher(e : ExpressionBinaire) {
   afficher(e.getExpGauche);
   afficher(e.operateur);
   afficher(e.getDroit);
}
```

En effet, lors de la compilation, le type statique de `e.getExpGauche()` est `Expression` et ce n'est que lors de l'exécution que nous connaitrons de type réel de l'expression (comme évoqué dans la première partie du document).

Une solution pour résoudre ce problème serait d'utiliser le `instanceof`

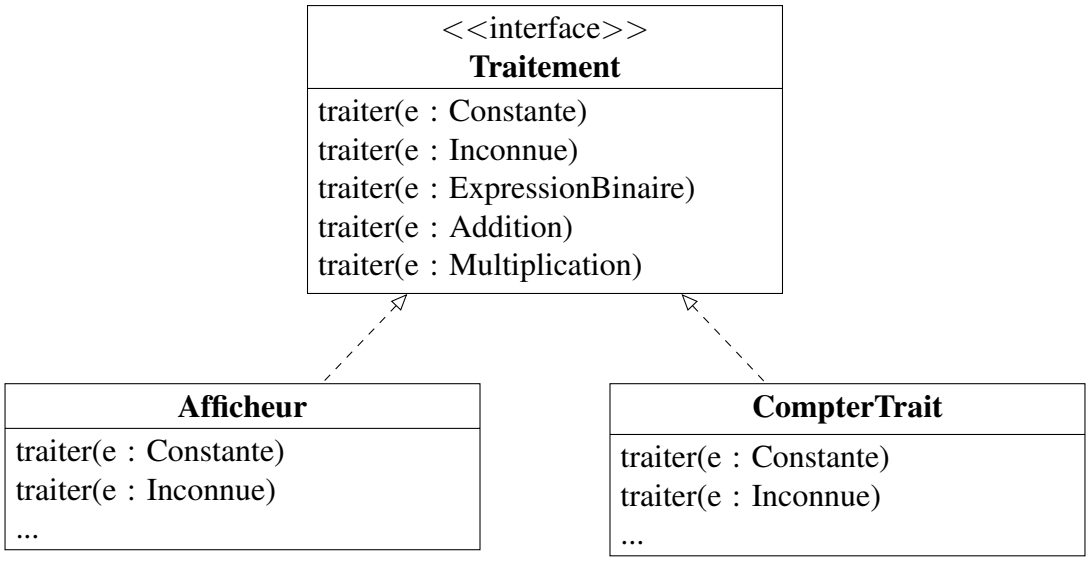
```
afficher(e : ExpressionBinaire) {
   if(e.getExpGauche instanceof Constante) {
      afficher( (Constante) e.getExpGauche); /* cast */
   } else if(e.getExpGauche instanceof Inconnue) {
```

```
      afficher( (Inconnue) e.getExpGauche); /* cast */
   } ...

   /* De même pour l'opérateur */

   /* De même pour l'expression droite */
}
```

Mais cette solution n'est pas du tout viable. Pour résoudre ce problème nous devons utiliser le type dynamique de `e.getGauche()` comme l'objet de la méthode `afficher()` et non comme argument (rappel 2.2.5).

4.3.2 Regrouper les classes sous une même interface

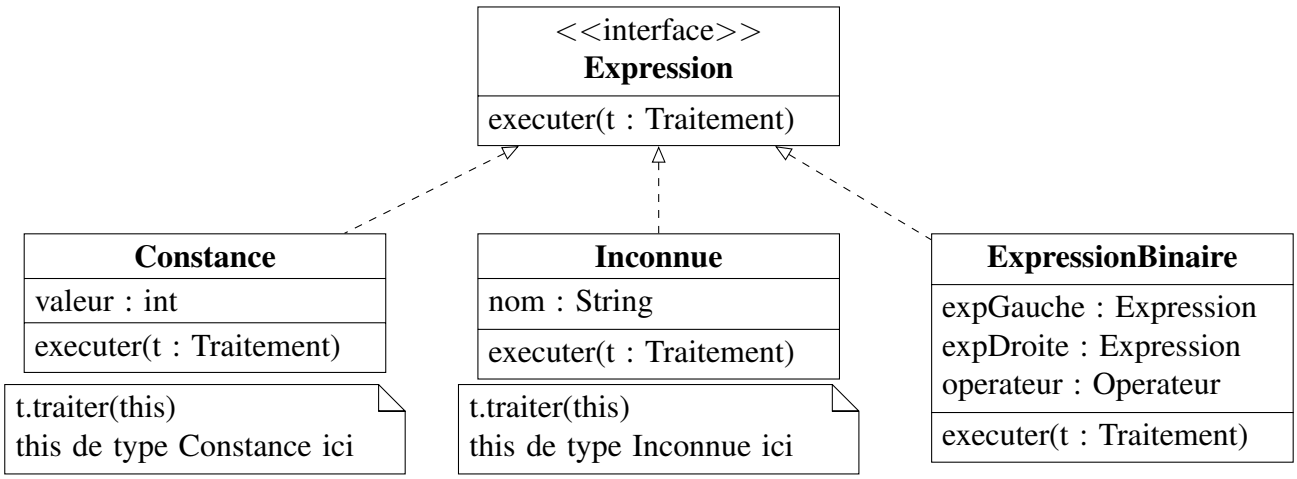
On peut également regrouper les classes `Afficheur` et `CompterTrait` sous une même interface pour simplifier la gestion. On aurait ainsi




```
class Afficheur {
   public void traiter(Constante e) {
      system.out.println(e.getValeur());
   }
   ...
   public void traiter(ExpressionBinaire e) {
      e.getGauche().xxxxxx(this);
      e.getOperateur().xxxxxx(this);
      e.getDroit().xxxxxx(this);
   }
   ...
}
```

Nous avions laisser volontairement le problème évoqué en 4.3.1 sans réponse. Nous y revenons ici pour apporter une solution.
Ci-dessus un extrait de la classe `Afficheur`. Comme abordé à la fin de la section 4.3.1 nous devons avoir `e.gauche`, `e.operateur` et `e.droit` comme instance d'une méthode et non comme argument.
Cette méthode devra être dans chacune des classes implémentant `Expression` et cette méthode pourra aussi être utilisée par `CompterTrait`, nous ne pouvons donc pas l'appeler *afficher()*.

Par convention nous l'appellerons *executer()* et elle prend en paramètre un objet de type `Traitement` afin de le traiter.



Le code précédent de la classe `Afficheur` devient donc

```
class Afficheur implements Traitement {
   public void traiter(Constante e) {
      system.out.println(e.getValeur());
   }
   ...
   public void traiter(ExpressionBinaire e) {
      e.getGauche().executer(this);
      e.getOperateur().executer(this);
      e.getDroit().executer(this);
   }
   ...
}
```

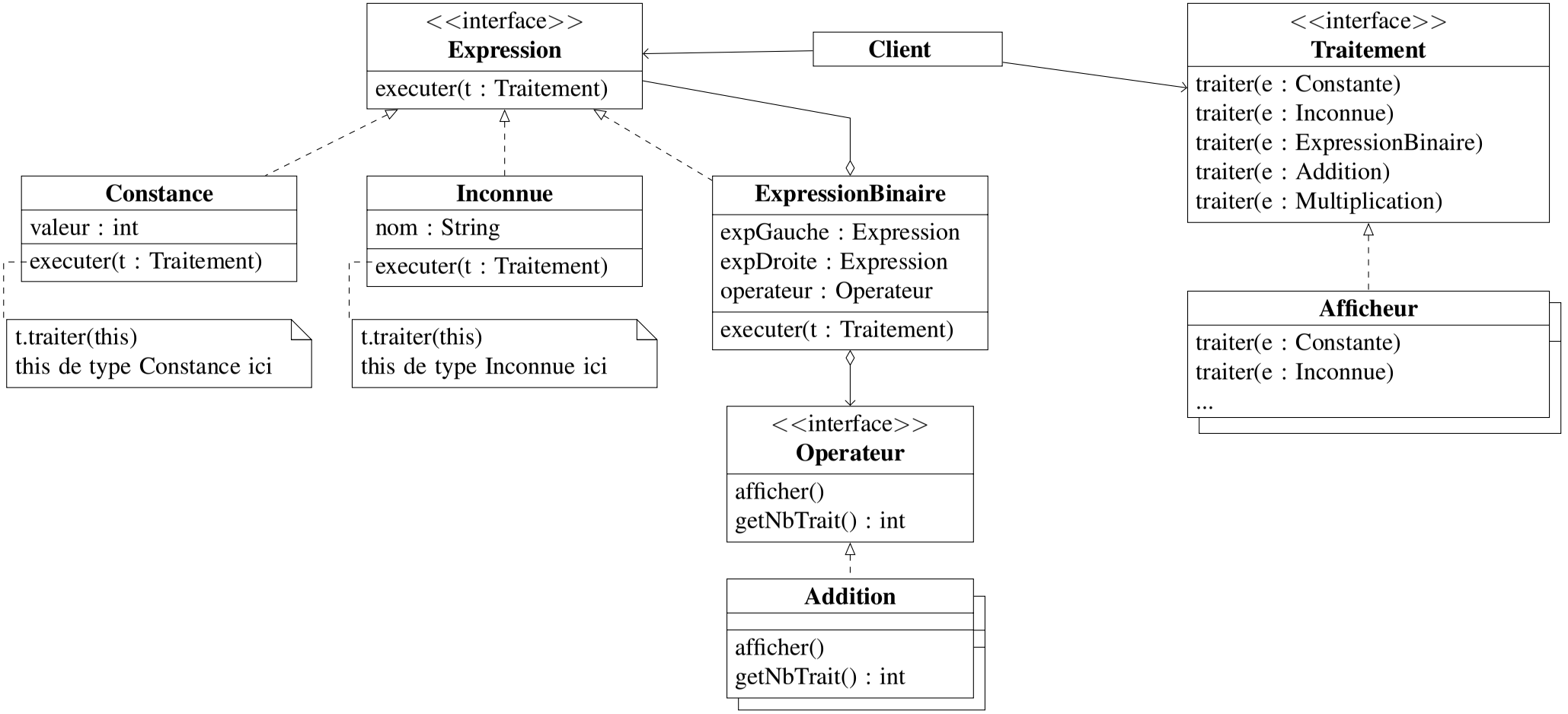
Et les implémentations de `Expression` sont

```
class Constante implements Expression {
   private int valeur;
   ...

   public void executer(Traitement t) {
      t.executer(this) // this ici vaut Constante
   }
}
```

Ainsi :

1. Si `e.getGauche` est de type `Constante`
2. Alors ça sera la méthode `executer()` de la classe `Constante` qui sera appelée
3. Ce qui a pour effet d'appeler la méthode `traiter(Constante c)` avec le paramètre de type `Constante`.
4. On a donc un retour à la classe `Afficheur` pour réaliser l'affichage

4.4 Diagramme de classe complet



4.4.1 Code implémentant la conception

La classe Afficheur

```
class Afficheur {
   public void traiter(Constante e) {
      system.out.println(e.getValeur());
   }
   ...
   public void traiter(ExpressionBinaire e) {
      e.getGauche().executer(this); /* this <=> e.gauche */
      e.getOperateur().executer(this); /* this <=> e.operateur */
      e.getDroit().executer(this); /* this <=> e.droit */
   }
   ...
   public void traiter(Addition e) {
      system.out.println( "+" );
   }
   ...
}
```

Suivant le type de `this` on appellera la méthode `executer()` avec la bonne signature.

Les implémentation de Expression

```
class Constante implements Expression {
   public void executer(Traitement t) {
      t.traiter(this) /* this typé Constante */
   }
}

class ExpressionBinaire implements Expression {
   public void executer(Traitement t) {
      t.traiter(this) /* this typé ExpressionBinaire */
   }
}
```

On appelle la méthode `traiter` définie dans la classe `Traitement` avec la bonne signature.

Les implémentations de Operateur

```
class Addition implements Operateur {
   public void executer(Traitement t) {
      t.traiter(this) /* this typé Addition */
   }
}

class Multiplication implements Operateur {
   public void executer(Traitement t) {
      t.traiter(this) /* this typé Multiplication */
   }
}
```

Exemple d'un code Client

```
/* e = 3 + x */
ExpressionBinaire e = new ExpressionBinaire(
  new Constante(3), new Inconnue(x), new Addition());

Afficheur afficheur = new Afficheur();
afficheur.traiter(e);
  --> e.gauche#executer(afficheur) // afficheur de type Afficheur
     -->  afficheur#traiter(e.gauche) // e.gauche de type Constante
        --> Sytem.out.println(constante)
  --> /* De même pour e.operateur */
  --> /* De même pour e.droit */
```

La principale difficulté dans la compréhension de ce patron est le fait de partir d'une méthode d'un visiteur concret puis de revenir dans une autre méthode de ce même visiteur.

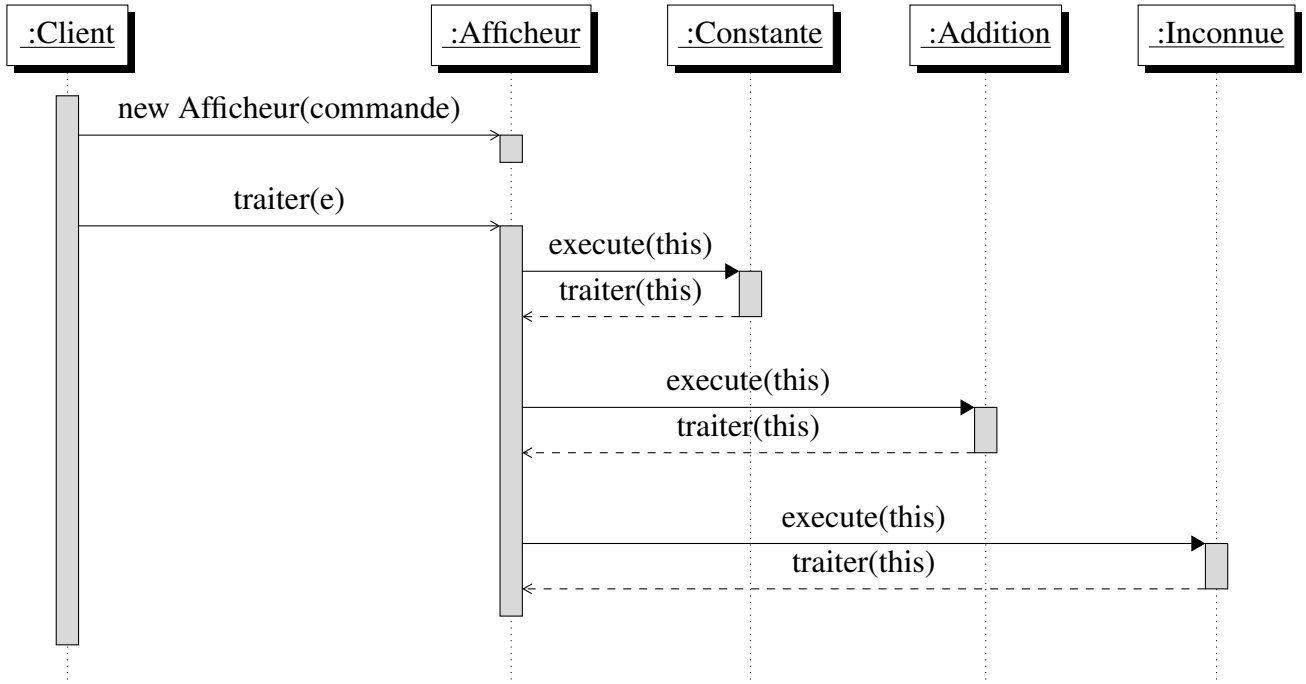
4.4.2 Déroulement des opérations

Nous revenons ici sur l'enchainement des opérations

1. Nous créons l'expression binaire $e = 3 + x$
2. Nous souhaitons l'afficher donc nous définissons un visiteur de type `Afficheur`.
3. Pour réaliser l'affichage nous appelons la méthode `traiter` sur notre afficheur en passant en argument l'expression binaire
   (a) Nous rentrons donc dans la méthode traiter
   (b) Qui va appeler en premier `e.getGauche().executer(this)` qui est un appel dynamique pour appeler la bonne méthode avec :
   — `e.getGauche()` est de type `Constante` (type dynamique pris en compte)
   — `this` est de type `Afficheur` (type statique pris en compte)
   (c) On appelle donc la méthode `traiter(Traitement t)` de la classe `Constante`
   (d) Cette méthode effectue un appel vers `t.traiter(this)` donc avec :
   — `t` de type `Afficheur`.
   — `this` de type `Constante`.
   (e) C'est donc la méthode avec la signature `traiter(Constante t)` qui est maintenant appelée dans `Afficheur`.
   (f) Nous avons maintenant l'affichage de la première partie de l'expression binaire
   (g) Nous recommençons au point 3.b) avec l'appel `e.getOperateur().executer(this)`.
   — avec `e.getOperateur()` de type `Addition`.
   (h) Puis une dernière fois au point 3.b) avec l'appel `e.getDroit().executer(this)`.
   — avec `e.getDroit()` de type `Inconnue`.

4.4.3 Diagramme de séquence de la solution

On peut également voir le déroulé des opérations à travers un diagramme de séquence.
On suppose que l'objet `ExpressionBinaire e` est déjà créé.



Les signatures respectives de `traiter(this)` sont :
- `traiter(Constante e)` qui affiche de la valeur 3.
- `traiter(Operateur e)` qui affiche d'un $+$.
- `traiter(Inconnue e)` qui affiche l'inconnue x.

4.5 Conclusion de l'exemple

Nous avons réussi à séparer les algorithmes qui sont maintenant définis dans les implémentations de `Traitement` et les objets sur lesquels ils opèrent. Puis grâce à cette nouvelle conception, il est très simple de rajouter une opération (un algorithme) : en créant une nouvelle classe implémente `Traitement`.

4.6 Amélioration

Si on regarde plus en détail l'interface `Traitement` on remarque qu'elle renvoie le type `void`. Pour l'affichage cela ne posait aucun problème, mais si on souhaite implémenter `CompterTrait` on ne souhaite pas forcément un affichage du nombre de traits. On voudrait, par exemple, récupérer un entier. Donc le type de retour des méthodes `traiter` ne serait plus uniquement, `void` mais également `int` pour renvoyer un entier. Nous avons donc besoin de la généricité.

4.6.1 Implémentation de la résolution du problème

```
public interface Traitement<T> {
   T traiter(Constante e);
   T traiter(Inconnue e);
   ...
}
```

```
public class Afficheur Traitement<Void> {
   void traiter(Constante e) { System.out.println(e.getValue(); }
   void traiter(Inconnue e) { System.out.println(e.getNom(); }
   ...
}

public class CompterTrait Traitement<Integer> {
   Integer traiter(Constante e { return 0}
   Integer traiter(Inconnue e) { return 0}

   Integer traiter(ExpressionBinaire e) {
      return e.getGauche().executer(this)
          + e.getOperateur().executer(this)
          + e.getDroit().executer(this);
   }
   ...
}
```

5 Exemple d'une exportation XML

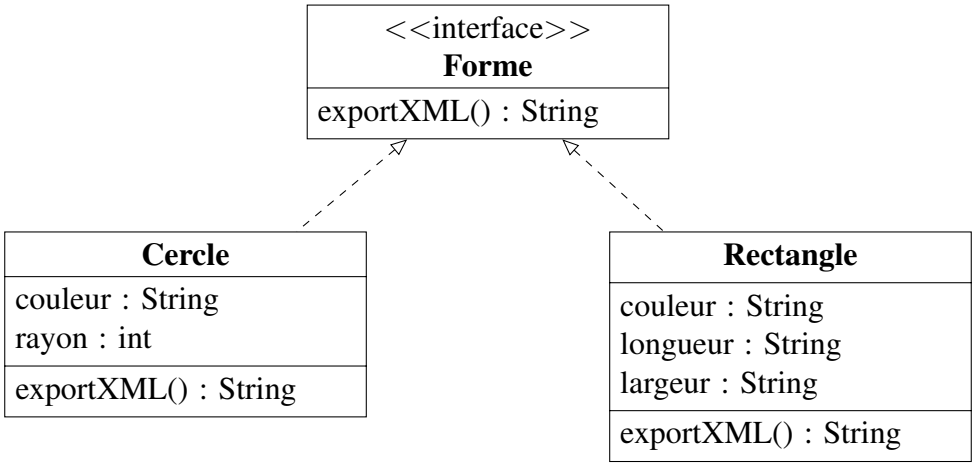
Pour mieux assimiler cette conception et la susceptibilité entre les appels de méthodes avec le mot-clé `this` ,nous vous proposons d'étudier un second exemple extrait du site *refactoring.guru [7]*

5.1 Définition du problème

Dans un premier temps, nous désirons pouvoir exporter des formes (rectangles, cercles, ...) au format XML.

5.2 Une première solution

Comme précédemment, nous développons une première version de notre application en implémentant les algorithmes d'exportation directement dans les classes représentantes nos formes.



On donne pour illustration le corps de la méthode `Cercle#exportXML()`.

```
return
    "<cercle>
        <couleur>" + this.couleur + "</couleur>
        <rayon>" + this.rayon + "</rayon>
    </cercle>"
```

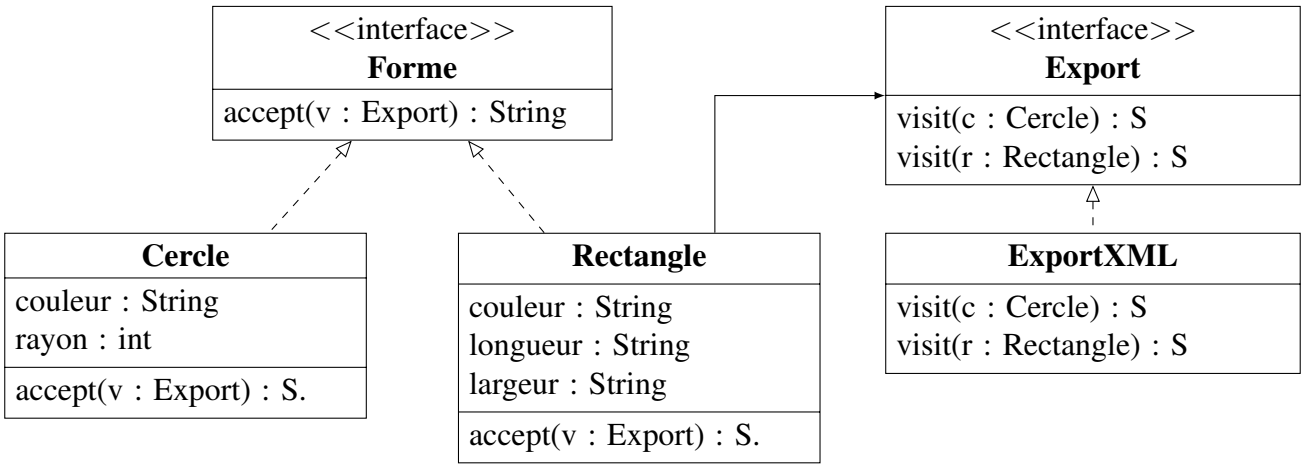
5.3 Solution avec le patron Visiteur

Le patron Visiteur nous demande de *séparer les algorithmes et les objets sur lesquels ils opèrent*. Nous devons donc créer :

1. Un visiteur pour l'exportation XML. Donc une classe `ExportXML`
2. Définir une méthode qui prend en paramètre ce visiteur dans l'interface `Forme`.

Comme nous l'évoquons depuis le début du document, nous devons nous efforcer à avoir une architecture qui permettra de facilement d'étendre notre application. Dans ce but, nous devons rajouter une interface `Export` et faire référer `ExportXML` à cette interface.

5.3.1 Diagramme de classe



5.3.2 Code implémentant la conception

La classe ExportXML

```
String visit(Cercle c) {
   return "<cercle>
             <couleur>" + c.getCouleur() + "</couleur>
             <rayon>" + c.gerRayon() + "</rayon>
           </cercle>"
}
```

Les implémentations de Forme

```
class Cercle {
   ...
   String accept(Export v) {
      return v.visit(this); /* this est de type Cercle */
   }
}

class Rectangle {
   ...
   String accept(Export v) {
      return v.visit(this); /* this est de type Rectangle */
   }
}
```

Exemple de code Client

```
ExportXML exportXML = new ExportXML();

Cercle c = new Cercle(5, "rouge");

result = c.accept(exportXML);
```

5.4 Ajouter un nouveau traitement

On souhaite maintenant ajouter un nouveau trainement qui permet d'exporter les formes au format JSON. Grâce à la solution précédent, cet ajout est très simple à implémenter. Il suffit de :

1. Créer un nouveau visiteur `ExportJSON`.
2. Implémenter les méthodes d'exportation pour les adapter au format JSON.

```
ExportXML exportXML = new ExportXML();
ExportJSON exportJSON = new ExportJSON();

Cercle c = new Cercle(5, "rouge");

result = c.accept(exportXML);
result2 = c.accept(exportJSON);
```

Il est donc très simple d'ajouter un nouveau traitement. En contrepartie, l'ajout d'une classe implémentant `Forme` est plus contraignant, car cette nouvelle classe peut arriver avec un lot d'opérations qui devront être ajoutées aux Visiteurs (`Export` et ses implémentations). Par conséquent, il est recommandé d'utiliser le patron Visiteur si vous êtes sûr qu'il n'y aura que peu de chance d'ajouter un nouvel élément concret à notre hiérarchie (ici `Forme`).

Donc, si la hiérarchie est stable, mais qu'on ajoute régulièrement des fonctionnalités ou qu'on change les algorithmes alors le patron Visiteur va nous aider à gérer ces changements.

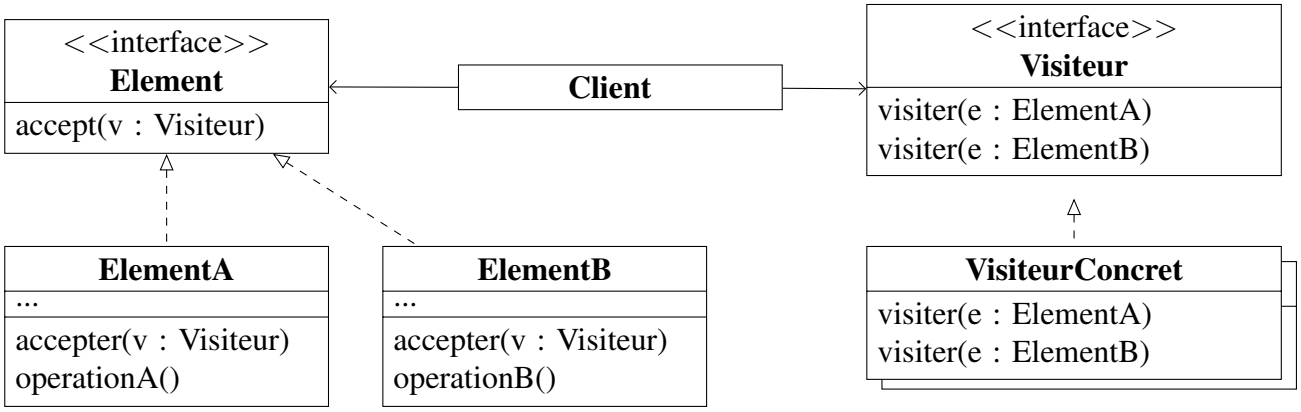
6 Structure du patron Visiteur

Comme vous l'aurez compris à travers ces deux exemples, le patron Visiteur nécessite :

— une interface `Visiteur` avec ses implémentations. Les méthodes de ces classes sont les algorithmes nécessaires au bon déroulement de l'application.

— une interface `Element` avec ses implémentations qui délègue le travail algorithmique aux visiteurs.

6.1 Conception UML

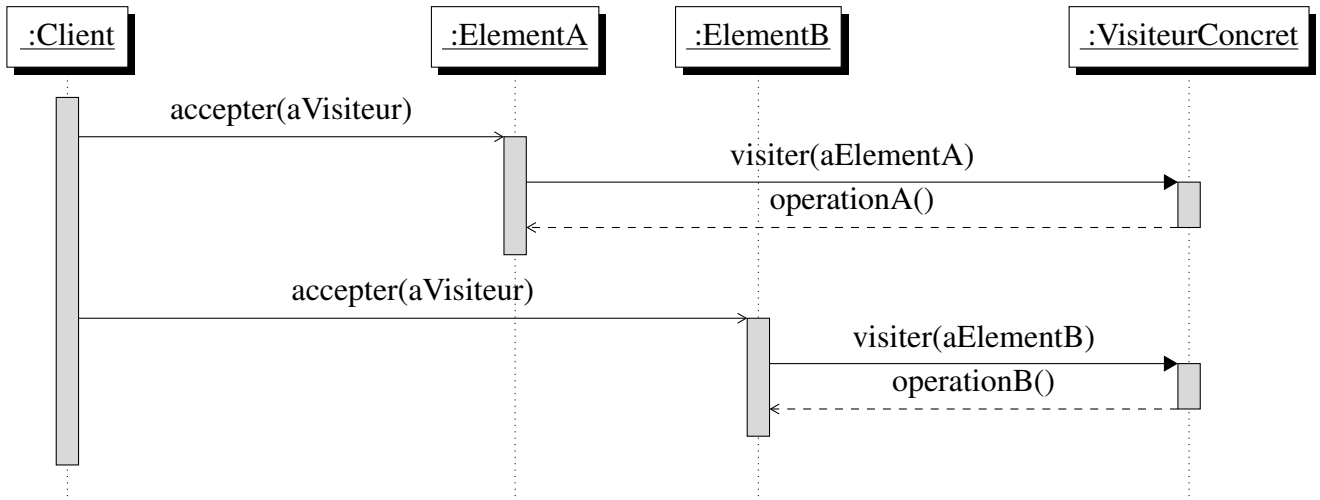


6.2 Participants

— **Visiteur** Déclarer une opération `visiter()` pour chaque classe concrète. Le nom de la fonction ainsi que sa signature doit désigner la classe qui envoie la requête au visiteur.

— **VisiteurConcret** Implémente chaque opération déclarée dans l'interface `Visiteur`. Chaque opération implémente une partie de l'algorithme correspondant à la classe de l'objet.

— **Element** Définit une opération `accepte(Visitor v)` qui prend en paramètre un visiteur.



Remarque : dans les exemples précédents nous n'avions pas d'autres méthodes en plus de `accept()` définie dans `Element`. En effet, le visiteur réalisé directement un traitement trivial et le renvoyé. Mais nous pouvons également faire en sorte que le visiteur appelle des méthodes des éléments concrets.

Conclusion

Avec le patron visiteur l'opération qui doit être exécutée dépend à la fois du type du `Visiteur` et du type de l'`Element`. Afin de pouvoir dépendre des deux types nous avons dû trouver une solution pour permettre ce *double-dispatch*. Au lieu d'avoir les opérations dans `Element` nous les exportons dans `Visiteur` et nous utilisons la méthode `accept()` pour faire la liaison au moment de l'exécution.

Ce patron permet donc :

- d'ajouter de nouvelles opérations facilement en créant simplement un nouveau visiteur.
- d'avoir une séparation entre les algorithmes et les classes définissant la structure de l'objet.

Dans ce document, nous sommes donc revenus sur la subtilité entre le type statique et le type dynamique. Puis nous avons étudié le patron Visiteur à travers deux exemples avant de présenter sa structure générale. Pour aller plus en détail sur ce patron de conception, nous vous recommandons la lecture du livre de référence sur les patrons de conception [8].

Références

[1] ENSIMAG. *Polymorphisme et liaison dynamique*. 2020. URL : https://programmation-orientee-objet.pages.ensimag.fr/poo/resources/fiches/05-Polymorphisme_liaisonDynamique/.

[2] WIKIPÉDIA. *Polymorphisme et liaison dynamique*. 2020. URL : https://fr.wikipedia.org/wiki/Dispatch_multiple.

[3] ORACLE. *Java Language and Virtual Machine Specifications*. 2020. URL : https://docs.oracle.com/javase/specs/#8.4..

[4] Xavier CREGUT. *TP - Patron de conception : le visiteur*. 2020. URL : http://cregut.perso.enseeiht.fr/ENS/2020-cnam-nfp121/cnam-nfp121-2020-deroulement-corrige012.html.

[5] WIKIPÉDIA. *Patron de conception Interpreteur*. 2020. URL : https://fr.wikipedia.org/wiki/Interpr%C3%A9teur_(patron_de_conception).

[6] Robert C. MARTIN. *Le Principe de responsabilité unique*. 2020. URL : https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html.

[7] Alexander SHVETS. *Design pattern Visitor*. 2020. URL : https://refactoring.guru/fr/design-patterns/visitor.

[8] Erich GAMMA, Richard HELM et Ralph JOHNSON. *Design patterns elements of reusable object oriented software*. Addison Wesley, 1998.