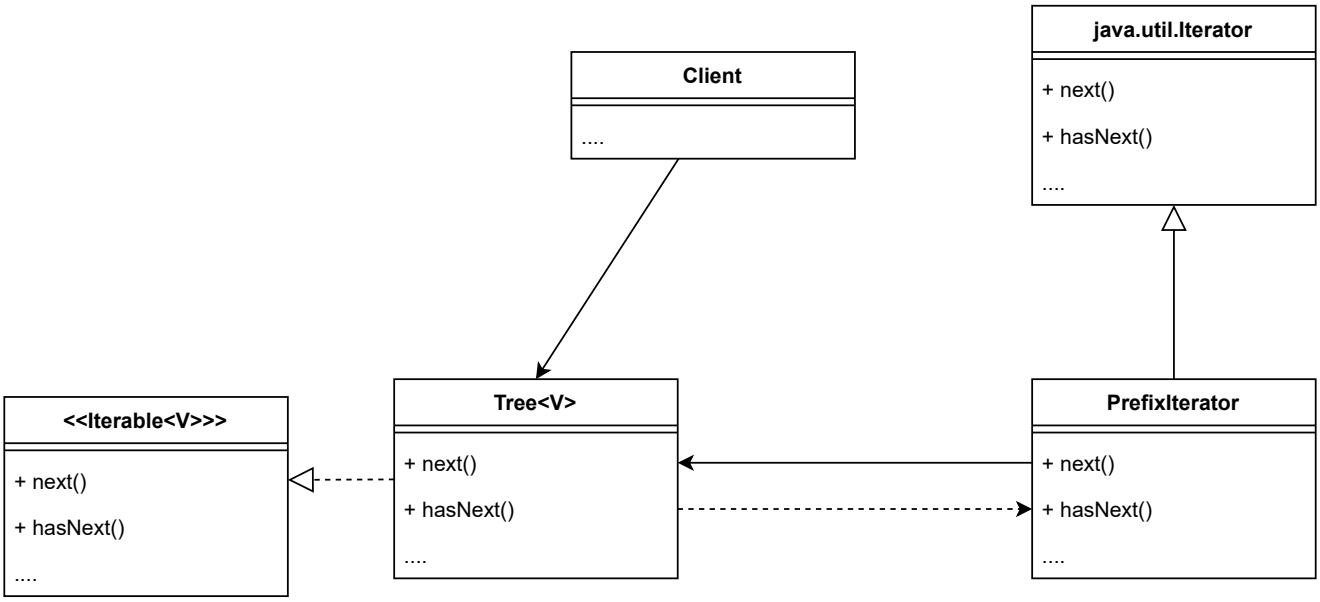
TP6 - Itérateur / Visiteur

#DesignPattern

Itérateur

L'itérateur permet de parcourir séquentiellement un agrégat (collection, conteneur) d'éléments sans en connaître sa structure interne. Pour cela, on délègue le parcours de l'aggrégat à un objet externe.



Ici on utilise l'exemple d'un Arbre qui se fait parcourir en parcours Préfixe, PostFixe et Infixe.

```
// On créé notre Iterator en parcours préfixe
// il va implémenter Iterator<V> et surcharger les méthodes
d'iteration, hasNext() et next()

package iterator;
public class PreFixIterator<V> implements Iterator<V> {

    private Tree<V> next;

    public PreFixIterator(Tree<V> root) {
        next = root;
    }

    @Override
```

```
    public boolean hasNext() {
        return (this.next != null);
    }

    @Override
    public V next() {
        System.out.println(this.next);
        V currentNode = this.next.getValue();
        if (next.getLeft() != null) {
            next = next.getLeft();
            return (currentNode);
        } else if (next.getRight() != null) {
            next = next.getRight();
            return (currentNode);

        } else if (next.getParent().getRight() == this.next) {
            next = null;
            return (currentNode);
        }
        return currentNode;
    }
}

/*****************************************************************************

// on crée notre arbre quelconque, Tree<V>, qui va implémenter
Iterable<V> et
// implémenter la méthode iterator()

import java.util.Iterator;
import iterator.*;

public class Tree<V> implements Iterable<V> {

    private Tree<V> left, right, parent;
    private V value;

    public Tree(V value, Tree<V> left, Tree<V> right) {
        this.value = value;
        this.left = left;
        this.right = right;
        this.parent = null;
```

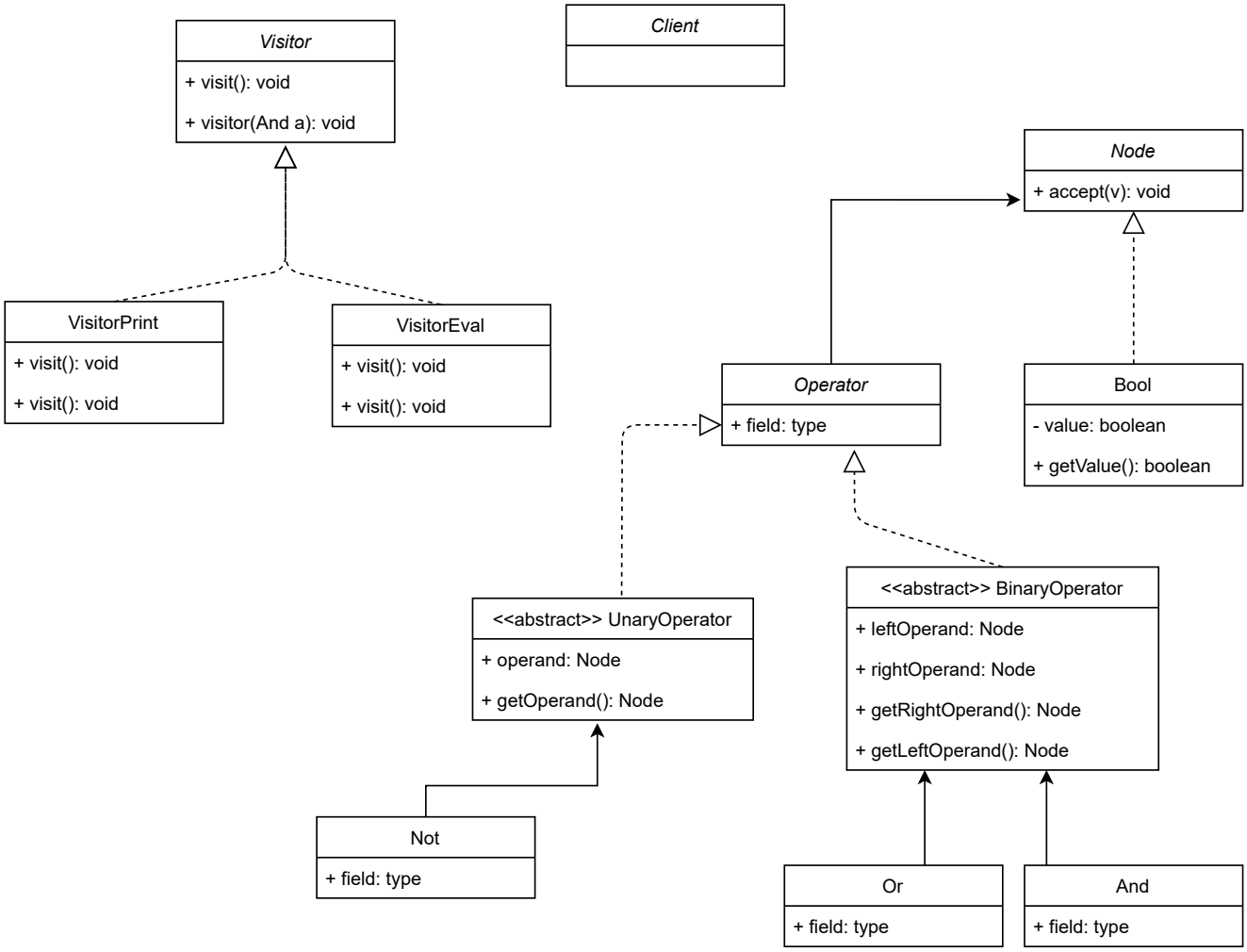
```
        if (this.left != null) {
            this.left.parent = this;
        }
        if (this.right != null) {
            this.right.parent = this;
        }
    }

    public Tree(V value) {
        this(value, null, null);
    }

    public Tree<V> getLeft() {
        return left;

    }

    public Tree<V> getRight() {
        return right;
    }

    public Tree<V> getParent() {
        return parent;
    }

    public V getValue() {
        return value;
    }

//  public Iterator<V> iterator() {
//      return new InOrderIterator<V>(this);
//  }

    public Iterator<V> iterator() {
        return new PreFixIterator<V>(this);
    }

//  public Iterator<V> iterator() {
//      return new PostFixIterator<V>(this);
//  }
}
```

```
/*************************************************************************

// Le client lui va créer l'arbre et instancier l'iterateur

public class Client{

    public static void main(String[] args) {
        Tree<String> tree = new Tree<String>("3", new Tree<String>("2",
null, null), new Tree<String>("1", null, null));
        Iterator<String> myIter = tree.iterator();
        if(myIter.hasNext()) {
            System.out.println(myIter.next());
        }
        if(myIter.hasNext()) {

            System.out.println(myIter.next());
        }
        if(myIter.hasNext()) {
            System.out.println(myIter.next());
        }

}
```

- Iterator définit une interface pour accéder aux élément et les parcourir.
- PrefixIterator implémente l'interface Iterator, assure le suivi de l'élément courant lors de la traversée de l'agrégat et détermine l'élément suivant (surcharge les méthodes d'Iterator)
- Iterable définit une interface pour la création d'un objet Iterator
- Tree implémente l'interface de la création de l'itérateur afin de retourner l'instance adéquate d'un itérateur correct.

Visiteur

Un visiteur permet de représenter une opération à effectuer sur les éléments d'une structure et de définir une nouvelle opération sans modifier les classes des éléments sur lesquels il opère. Pour arriver à cela, il faut externaliser les opérations d'une structure d'objet dans une hierarchie séparée et ajouter une méthode dans la structure d'objet pour accueillir des instance des opérations.



Ici on cherche à implémenter un système qui permet d'évaluer ou afficher des expressions booléennes

```
// crééons tout d'abord la structure de notre Noeud et de notre
opération
// le noeud va avoir la méthode accept qui va laisser passer un
visiteur afin
// qu'il puisse aller opérer sur le noeud.
public interface Node {

    public void accept(Visitor v);

}
/**************************************************************************

// On va avoir un élément Operator qui va universaliser nos éléments
concrets en dessous.
public interface Operator extends Node {

}
```

```
/*****************************************************************************

// on va construire nos éléments BinaryOperator et ses enfants Or et
And afin d'avoir notre
// structure d'opération.

public abstract class BinaryOperator implements Operator {

    protected Node leftOperand, rightOperand;

    public BinaryOperator(Node leftOperand, Node rightOperand) {
        this.leftOperand = leftOperand;
        this.rightOperand = rightOperand;
    }

    public Node getLeftOperand() {
        return leftOperand;
    }


    public Node getRightOperand() {
        return rightOperand;
    }
}
/*************************************    AND
************************************/
package boolexp;

import boolexp.visitor.Visitor;

public class And extends BinaryOperator {

    public And(Node leftOperand, Node rightOperand) {
        super(leftOperand, rightOperand);
    }

    public void accept(Visitor v) {
        v.visit(this);
    }
}
/*************************************    OR
************************************/
package boolexp;
```

```
import boolexp.visitor.Visitor;

public class Or extends BinaryOperator {

    public Or(Node leftOperand, Node rightOperand) {
        super(leftOperand, rightOperand);
    }

    public void accept(Visitor v) {
        v.visit(this);
    }

}

/**************************************************************************

// on va implémenter nos visiteurs et leurs fonctions visiteur
d'affichage et d'évaluation
public interface Visitor {

    public void visit(Bool n);

    public void visit(Or n);

    public void visit(And n);

    public void visit(Not n);
}
/*********************************    VisitorPrint
***************************/
public class VisitorPrint implements Visitor {

    @Override
    public void visit(Bool n) {
        System.out.print(n.getValue());
    }

    @Override
    public void visit(Not n) {
        System.out.print("(NOT ");
```

```
        n.getOperand().accept(this); // lorsque l'on accepte un
visiteur on va aller taper
        // dans la methode accept de notre opérande qui elle va appeler
la fonciton visit
        // de notre visiteur sur elle même.
        System.out.print(")");
    }

    @Override
    public void visit(And n) {
        System.out.print("(");
        n.getLeftOperand().accept(this);
        System.out.print(" AND ");
        n.getRightOperand().accept(this);
        System.out.print(")");
    }


    @Override
    public void visit(Or n) {
        System.out.print("(");
        n.getLeftOperand().accept(this);
        System.out.print(" OR ");
        n.getRightOperand().accept(this);
        System.out.print(")");
    }

}
/**********************************    VisitorEval
*****************************/
public class EvalVisitor implements Visitor {

    private boolean value;

    public boolean getValue() {
        return value;
    }

    @Override
    public void visit(Bool n) {
        value = n.getValue();
    }
```

```
    @Override
    public void visit(Not n) {
        n.getOperand().accept(this);
        value = !value;
    }

    @Override
    public void visit(And n) {
        n.getLeftOperand().accept(this);
        boolean left = value;
        n.getRightOperand().accept(this);
        value = left && value;
    }

    @Override
    public void visit(Or n) {

        n.getLeftOperand().accept(this);
        boolean left = value;
        n.getRightOperand().accept(this);
        value = left || value;
    }
}
/*************************************************************************

// le mClient va uniquement créer les expressions dans une Node et
demander à un visiteur
// de les visiter, l'expression va alors être traversée sans pour
autant modifier les
// classes qu'il visite

public class Client {

    public static void main(String[] args) {
        Node exp = new And(new Or(new Bool(true), new Bool(false)), new
Not(
                new Bool(true)));
        VisitorPrint v1 = new VisitorPrint();
        System.out.print("Expression booléenne : ");
        exp.accept(v1)
    }
```

```
}
```

- Visitor déclare une opération Visit pour chaque classe de Operator (sous éléments concrets).
- La signature de l'opération identifie la classe qui renvoie la requête de visite au visiteur
- Le visiteur détermine alors la classe concrète de l'élément visité et accède à cet élément directement à travers son interface.
- VisitorPrint (et VisitorEval) implémente les opérations (i.e. l'affichage et l'évaluation) déclarées par Visitor, fournit le contexte de l'algorithme et stock son état local (e.g. pour accumuler des résultats lors du parcours de la structure)
- Node définit une opération accept() qui prend un visiteur en paramètre
- And,Or,Not, etc... implémentent l'opération accept et fait appel à la méthode visit() du visiteur en se passant lui-même en paramètre