Compte Rendu : UML et design Patterns

TP N°3 : Le design pattern Composite

Réalisé par : -Bouabidi Selma

-Makhlouf Younes

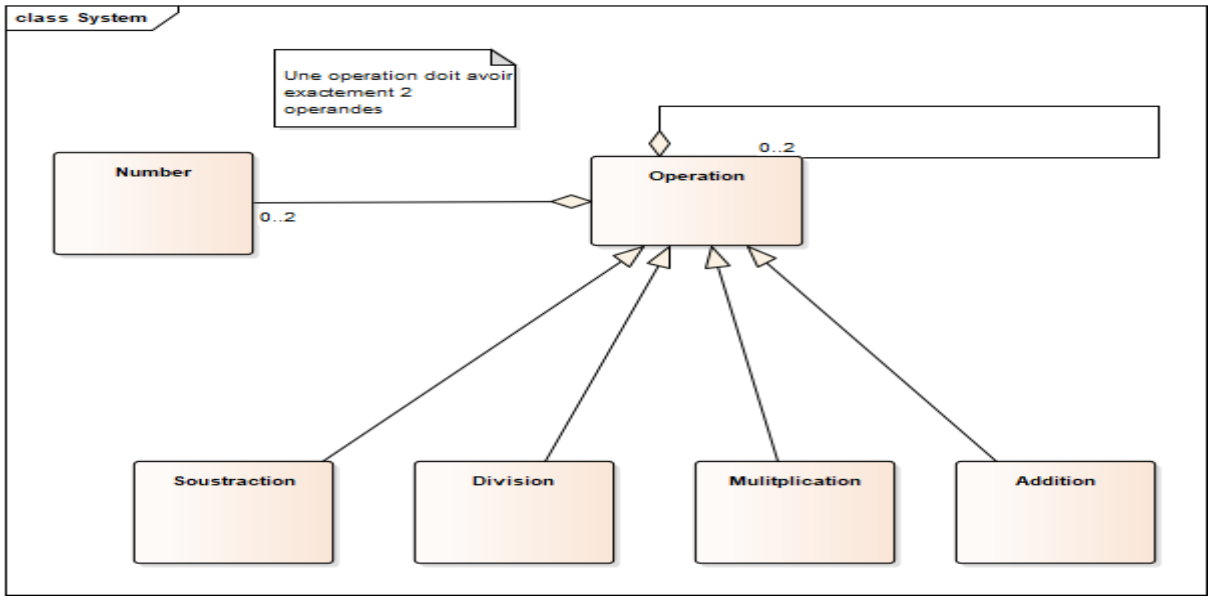
-Ellini Yessine

-Mansouri Samer

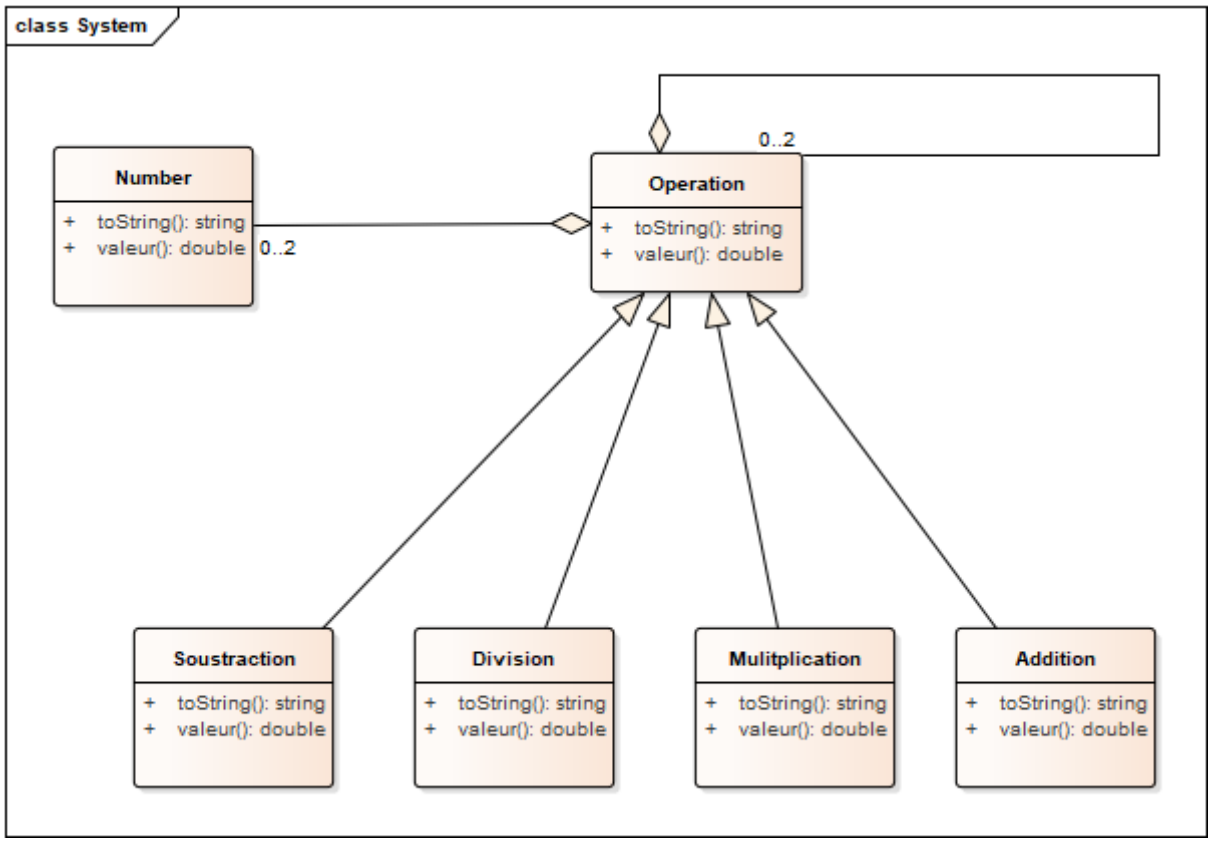
I. UML

1)

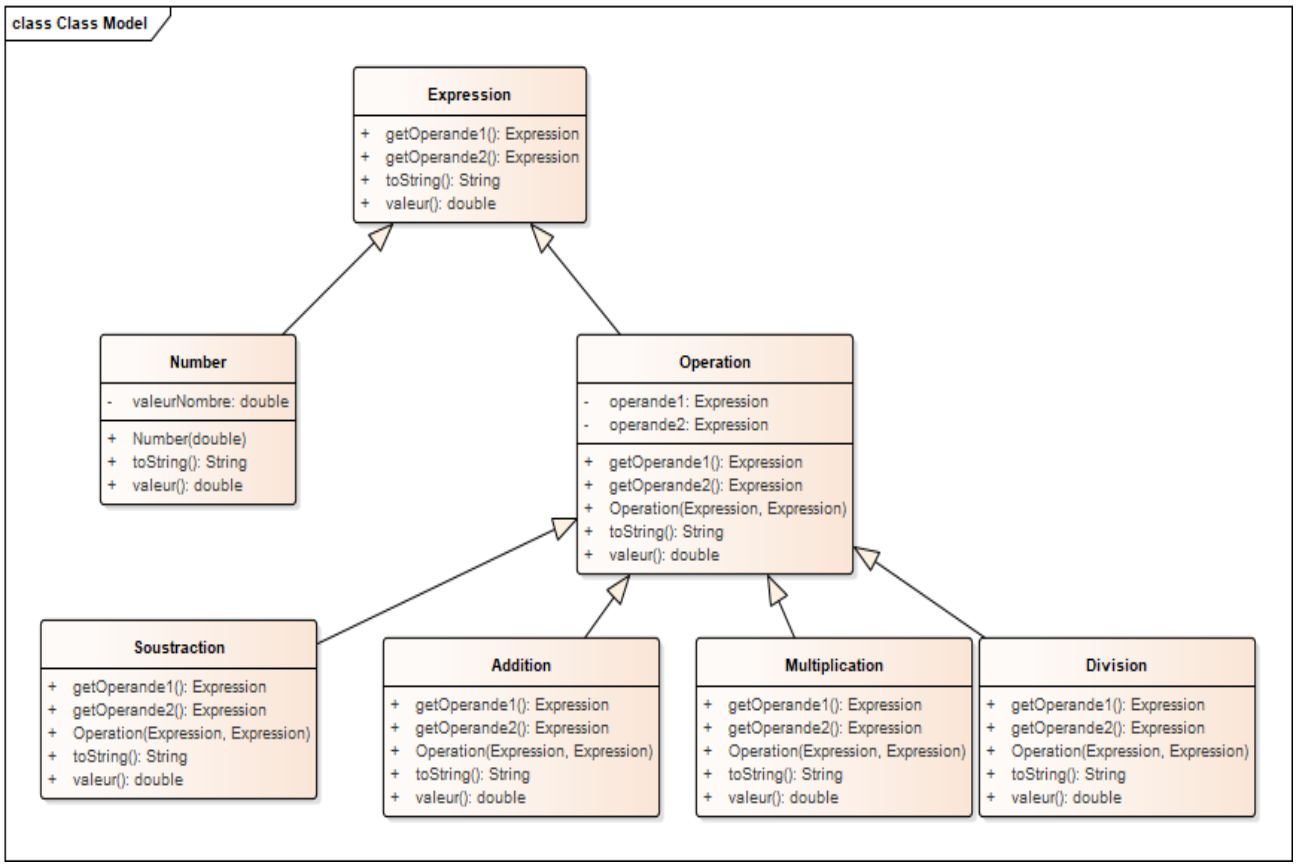
Sans le design pattern composite , une operation peut etre representée par ce diagramme :



1)b



3) En exploitant le modèle de conception composite, les nombres et les opérations sont implémentés selon la même interface d'expression. Les nombres sont assimilés à des feuilles, tandis que les nœuds représentent des opérations munies d'opérandes de type expression.



II. Java

Expression :

```
public class Expression {
    public Expression getOperande1(){
        return null;
    }
    public Expression getOperande2(){
        return null;
    }
    public double valeur(){
        return 0;
    }

    public String toString(){
        return "";
    }
}
```

Nombre:

```
public class Number extends Expression{
    private double valeurNombre;
    public Number(double number){
        valeurNombre=number;
    }
    public double valeur(){
        return valeurNombre;
    }
    public String toString(){
        return String.valueOf(valeurNombre);
    }
}
```

Operation :

```
public class Operation extends Expression{
    private Expression operande1;
    private Expression operande2;
    public Operation(Expression op1 , Expression op2){
        operande1=op1;
        operande2=op2;
    }
    public Expression getOperande1(){
        return operande1;
    }
    public Expression getOperande2(){
        return operande2;
    }

}
```

Addition:

```
public class Addition extends Operation{
    public Addition(Expression op1, Expression op2) {
        super(op1, op2);
    }

    public double valeur(){
        return getOperande1().valeur()+getOperande2().valeur();
    }
    public String toString(){
        return getOperande1().valeur()+"+" +getOperande2().valeur();
    }


}
```

Soustraction:

```
public class Soustraction extends Operation{
    public Soustraction(Expression op1, Expression op2) {
        super(op1, op2);
    }
    public double valeur(){
        if(getOperande2().valeur()!=0)
            return getOperande1().valeur()-getOperande2().valeur();
        else return 0;
    }
    public String toString(){
        return getOperande1().valeur()+"-" +getOperande2().valeur();
    }
}
```

Multiplication:

```
public class Multiplication extends Operation{
    public Multiplication(Expression op1, Expression op2) {
        super(op1, op2);
    }
    public double valeur(){
        return getOperande1().valeur()*getOperande2().valeur();
    }
    public String toString(){
        return getOperande1().valeur()+"*" +getOperande2().valeur();
    }
}
```

Division:

```
public class Division extends Operation{
    public Division(Expression op1, Expression op2) {
        super(op1, op2);
    }

    public double valeur(){
        //traiter le cas de division par 0
        if(getOperande2().valeur()!=0)
        return getOperande1().valeur()/getOperande2().valeur();

        else return 0;
    }
    public String toString(){
        return getOperande1().valeur()+"/" +getOperande2().valeur();
    }
}
```

Classe Calculatrice:

```
public class Calculatrice {
    public static void main(String[] args) {
        Expression n10 = new Number(10);
        Expression n6 = new Number(6);
        Expression n4 = new Number(4);
        Expression n5 = new Number(5);
        Expression n3 = new Number(3);
        Expression exp1 = new Soustraction(n10, n6);
        System.out.println(exp1 + " = " + exp1.valeur());
        Expression exp2 = new Multiplication(exp1, n5);
        System.out.println(exp2 + " = " + exp2.valeur());
        Expression exp3 = new Addition(n4, n3);
        System.out.println(exp3 + " = " + exp3.valeur());
        Expression exp4 = new Division (exp2,exp3);
        System.out.println(exp4 + " = " + exp4.valeur());
    }
}
```

Résultat de l'exécution:

```
10.0-6.0 = 4.0
4.0*5.0 = 20.0
4.0+3.0 = 7.0
20.0/7.0 = 2.857142857142857
```