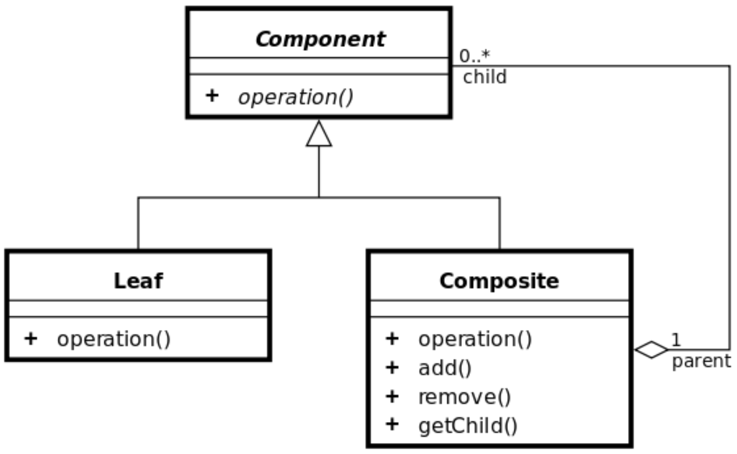
Individual work: know how to reuse a Composite Pattern

1. Recall structure and participants of a Composite



Component

- Abstraction of all components, including composites.
- Common interface to all objects in the composition (recursive methods represented by the method named operation on the schema).

Leaf

- Objects leaves in the representation of diagram objects for the hierarchy of composition.
- Implementation of the recursive methods stop case represented by the method named operation on the schema.

Composite

- Internal node objects in the representation of diagram objects for the hierarchy of composition.
- Implementation of the body of recursive methods with recursive call or delegation to sub trees nodes or children, represented by the method named operation on the schema.
- Implementation of tree handling methods (add, remove, getChild).

2. A Composite for Boolean Logic Expressions

Three-variable logic function

In everyday language, we want to model the fact that we pick up our phone when we decide to call, and the phone does not ring or when the phone rings and we decide to answer. The Boolean expression, that we named pick up the phone, with three variables can be modeled in Boolean logic by: pick up the phone = (NOT phone ring AND decision to call) OR (phone ring AND decision to answer). Given the following three Boolean variables:

- a, having as value interpretation "phone ring"
- b, having as value interpretation "decision to answer"
- c, having as value interpretation "decision to call"

The Boolean expression d = "pick up" is a logical function of the 3 preceding variables and can be written `d = ((!a.c)+(a.b))`

The truth table of this function is then as follows (on the right).

Truth table of pick up

a	b	c	pick up
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

The goal of this individual work is to apply the Composite for resolving elegantly and effectively this problem of modeling and programming with objects.

A few precisions:

1. In our representation as a character strings, the logical NOT operator will be represented by the character “!”, the logical AND operator will be represented by the character “.”, and the logical OR operator will be represented by the character “+”.
2. The logical AND operator has the same truth table as the && operator in Java.
3. The logical OR operator has the same truth operator as the || operator in Java.

Let the code of the next class modeling this logical expression:

```
public class LogicalExpressionWith3Variables {

  private TableOfSymbols symbols;

  public Satisfiable constructLogicalEpressionWith3Variables() {
    this.symbols = new TableOfSymbols();
    BooleanVariable a = new BooleanVariable("a", "Phone ring");
    this.symbols.addVariable(a);
    BooleanVariable b = new BooleanVariable("b", "Decision to answer");
    this.symbols.addVariable(b);
    BooleanVariable c = new BooleanVariable("c", "Decision to call");
    this.symbols.addVariable(c);
    BinaryOperand leftConjunction = new And();
    UnaryOperand notA = new Not();
    notA.setOperande(a);
    leftConjunction.setLeftOperand(notA);
    leftConjunction.setRightOperand(c);
    BinaryOperand rightConjunction = new And();
    rightConjunction.setLeftOperand(a);
    rightConjunction.setRightOperand(b);
    BinaryOperand pickUp = new Or();
    pickUp.setLetfOperand(leftConjunction);
    pickUp.setRightOperand(rightConjunction);
    return pickUp;
  }

  public void whenPickUpYourPhone(Satisfiable pickUp) {
      for (int i = 0; i <= Math.pow(2,3) -1; i++) {
            this.symbols.fixTruthValue(i,3);
            if (pickup.isSatisfiable()) {
                  System.out.print("I pick up my phone when: ");
                  for (int j = 0; j < 3; j++) {
                        System.out.print(this.symbols.interpretation(j));
                        if (j < 2)
                              System.out.print(" and ");
                  }
                  System.out.println();
            }
      }
  }

  public void whenNotPickUpYourPhone (Satisfiable pickUp) {
      for (int i = 0; i <= Math.pow(2,3) -1; i++) {
            this.symbols.fixTruthValue(i,3);
            if (!pickUp.isSatisfiable()) {
                  System.out.print("I don't pick up my phone when: ");
                  for (int j = 0; j < 3; j++) {
                        System.out.print(this.symbols.interpretation(j));
                        if (j < 2)
                              System.out.print(" and ");
                  }
                  System.out.println();
            }
      }
  }

  public static void main(String[] args) {
      expressionWith3Variables customer = new LogicalExpressionWith3Variables();
      Satisfiable pickUo = customer.constructLogicalEpressionWith3Variables();
      System.out.print("Evaluation of Boolean expression: ");
      System.out.println(pickUp);
      for (int i = 0; i <= Math.pow(2,3) -1; i++) {
            customer.symbols.fixTruthValue(i,3);
            System.out.println(pickUp.isSatisfiable());
      }
      customer.whenPickUpYourPhone(pickUp);
      customer.whenNotPickUpYourPhone(pickUp);
  }
}
```

The code of the TableOfSymbols class is given in annex.

Question 1 (2 points) :

Give the object diagram produced by the method `constructLogicalEpressionWith3Variables`.

Question 2 (3 points) :

Match the domain classes to the pattern participants and match the methods to code with the operations present in the structure of the Composite pattern. Make a copy of this table completed by you.

Composite Pattern	**Domain**
Component	
Leaf	
Composite	
Common Operations	
Specific operations of composite	
Specific operations of leaves	

Question 3 (2 points) :

Give the most complete class diagram

Question 5 (7 points) :

Program the classes of your UML diagram so that the execution of the main program gives:

```
Evaluation of Boolean expression: ((!a.c)+(a.b))
false
true
false
true
false
false
true
true
I pick up my phone when: not Phone ring and not Decision to answer and Decision to
call
I pick up my phone when: not Phone ring and Decision to answer and Decision to
call
I pick up my phone when: Phone ring and Decision to answer and not Decision to
call
I pick up my phone when: Phone ring and Decision to answer and Decision to call
I don't pick up my phone when: not Phone ring and not Decision to answer and not
Decision to call
I don't pick up my phone when: not Phone ring et Décision de répondre et non
Décision d'appeler
I don't pick up my phone when: Phone ring and not Decision to answer and not
Decision to call
I don't pick up my phone when: Phone ring and not Decision to answer and Decision
to call
```

Question 6 (3 points) :

Write the code for the Implication, Inhibition and Equivalence classes:

- The *implication* (written =>) is written as follows (a=>b) and is equivalent to the expression (!a+b)
- The *inhibition* (written \) is written as follows (a\b) and is equivalent to the expression (a. !b)

- The *equivalence* (written <=>) is written as follows (a <=>b) and is equivalent to the expression (a.b)+(!a. !b)

Question 7 (3 points) :

Four-variable logic function

A good student wonders if it is wise to go out one evening. He must decide based on four variables:

- a = he has enough money
- b = he finished his homework
- c = public transport is on strike
- d = family automobile is available

This student can go out if:

- he has enough money, a = true
- he finished his homework, b = true
- public transport is not on strike, c = false
- or if family automobile is available, d = true

The logical expression goOut can be written in function of the state of the variables a, b, c et d in the following maner: goOut = ((a.b).(!c+d))

Draw the object diagram and program the method `constructLogicalEpressionWith4Variables()` corresponding to the boolean expression.

Annexe :

```
import java.util.AbstractList;
import java.util.ArrayList;

public class TableOfSymbols {
	private AbstractList<VariableBooléenne> variables;

	public TableOfSymbols() {
		this.variables = new ArrayList<VariableBooléenne>();
	}

	public void addVariable (VariableBooléenne variable) {
		this.variables.add(variable);
	}

	public void fixTruthValue(boolean... values) {
		for (int i = 0; i < Math.min(values.length,variables.size()); i++)
			this.variables.get(i).fixTruthValue(values[i]);
	}

	public void fixTruthValue(int value, int nbVariables) {
		boolean[] listOfParametres = new boolean[nbVariables];
		int shift = 0;
		do {
			int remainder = value % 2;
			listOfParametres[nbVariables-1-shift++] = remainder == 1;
			value /= 2;
		} while (value != 0);
		this.fixTruthValue(listOfParametres);
	}

	public String interpretation(int numVariable) {
		return this.variables.get(numVariable).interpretation();
	}
}
```