Object Orientation – Spring 2018 **Assignment 9**

The visitor pattern

Learning Goals

In this assignment, we ask you to design and implement a representation for logical formulas in Java. After having completed this assignment, you should be able to:

- Implement recursive types or self-referencing classes to represent (logical) formulas, where
  - a suitable interface/ abstract class is defined that serves as a base for all nodes in your formula tree;
  - each logical connective is implemented as an extension of this base class;
- introduce new operations using the visitor pattern;
- name the advantages and disadvantages of this particular design pattern;
- utilize UML (class/sequence) diagrams to design an object oriented program.

Logical Formules

This assignment is about representing and manipulating formulas from the field of proposition logic.

A *(logic) formula* consists of the constants (*true* and *false*), atomic formulas (represented as a String in our case) combined with the logic operators $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies) and $\neg$ (not). We can render the *syntax* of these formulas in the following way:

$$F ::= \textbf{true} \mid \textbf{false} \mid \textbf{String} \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Rightarrow F_2 \mid \neg F$$

We can display an actual formula as plain text or in tree form. The leaves in such a tree correspond to constants or atomic formulas and each internal node represents an application of one of the operators. This means that some internal nodes will have 2 children and others exactly 1 child. The tree shape has the advantage that it is often easier to introduce new (complex) operations.

In this assignment, we are therefore going to use a tree representation for our formulas that resembles the representation of numerical expressions we used in assignment 5. We will introduce a single interface as an abstraction of all different nodes, and introduce concrete classes implementing this interface for each type of node. We will make two changes compared to the expressions assignment:

1. In the assignment on expressions, operations were added to the base interface as abstract methods. In this assignment, we will leave the base interface unchanged and make use of the *visitor pattern* to implement new operations.
2. We will introduce a single concrete class to represent all binary operators and use the *strategy pattern* to realize their different behaviors.

In the lecture, it was explained that the visitor pattern is based on a designated interface (called `FormVisitor` in our case) in combination with an abstract method (called `accept`) in the base class. This abstract method can be placed into a 'Visitable' interface of its own, but we do not gain much with that, since the new interface can only be used inside the specific base class: it can not be used anywhere else. The base interface looks like this:

```
public interface Form {
    void accept( FormVisitor visitor ) ;
}
```

All of the tree's concrete nodes need to implement this interface. The `FormVisitor` type that we used here is, as explained before, an interface in itself:

```
public interface FormVisitor {
    void visit( Form form );
}
```

This interface needs to be adjusted later on.

All of the concrete nodes in the formula tree are classes that implement `Form`. Here is an example for a class of *not*-nodes:

```
public class NotForm implements Form  {
    private Form operand;

    public NotForm( Form oper ) {
        this.operand  = oper;
    }

    public Form getOperand() {
        return operand;
    }

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

This definition should be more or less self explanatory. Since the abstract method `accept` is included in `Form`, we are required to give a definition of `accept`. This definition should always (more or less) match the one given above.

Once we have added classes for all the other types of nodes to our project, we need to adapt `FormVisitor`. A quick remark: the binary nodes should be represented by a single class which gets its operator as an additional attribute. This is an example of the `strategy pattern`. You can use the the generic interface `BinaryOperator<T>` from the Java-api for this strategy attribute. You should implement the operations as an enumeration type as discussed in the lecture/tutorial.

Excercises

1. Complete the representation given above: Add new classes (with suitable attributes) for every type of node and adapt `FormVisitor`. Before you begin with your implementation, draw a UML *class diagram* of all the classes you need.

2. Add a number of static methods to your main class, each of which construct a tree of a logic formula. Think up a few interesting cases on your own. You will use these to test your operations.

3. Define a class `PrintFormVisitor` that implements the `FormVisitor` interface and prints the formula on screen as a text string. Try to print the formula with as little parentheses as possible by assigning priorities to the different operators. From high to low priority, the common order is: $\neg, \wedge, \vee, \Rightarrow$. So $\neg$ binds stronger than $\wedge$, etc.

4. Show the interactions between your different objects using a *sequence diagram*. This should portray how the visitor pattern works.

By assigning a boolean value to each proposition variable (the function to do that type of operation is commonly called a *valuation function*), we can compute the truth value of such a formula. You are now going to write an `EvalFormVisitor` that calculates the value of a formula. It is best to use a Java `Map<K,V>` for your valuation. Since `EvalFormVisitor` in fact performs a calculation, it is useful to adapt the `FormVisitor` interface and the abstract `accept` method so that `visit` as well as `accept` can return a value. We will use *generics* to implement this.

```
public interface Form {
    public <R> R accept( FormVisitor<R> visitor );
}

public interface FormVisitor<R> {
    R visit( NotForm form );
    ...
}
```

5. Adjust your classes and interfaces in a way that they can use these new interfaces. Make sure that your `PrintFormVisitor` works again. Hint: Use `Void`[1] as the return type of your `PrintFormVisitor`.

6. Implement an `EvalFormVisitor` and test it on different valuations and formulas.

Hand in

Before Sunday 22 april, 23:59 uur, on Blackboard. Concerning the UML diagrams: if you have drawn them by hand, you can best scan them in and send them as a `jpeg` file. Should you have used a tool, please save your file as a `pdf` and hand it in that way.

[1] `Void` is the class representation of the keyword `void` standing for the 'empty type'.