Software Specification and Design

Engi 9874, 2020

Due Sept 28 @ 11:59PM.

For each question you will be marked on programming style as well as correctness. To see my opinion about what constitutes good programming style see http://www.engr.mun.ca/~theo/Courses/ds/pub/style.pdf. In short:

- All .java files must be professionally commented; in particular, each file should contain a comment header that gives your name, student number, and mun email address. Each subroutine and class should have a comment at the start of it. I encourage you to use the "javadoc" conventions for comments.
- Code and comments must be consistently indented; tab stops should be set every 4 characters.
- Names must be chosen carefully and spelled correctly. (Use names starting with lower case letters for variables and methods; use names starting with upper case letters for classes and interfaces.)
- Use subroutines to avoid redundant coding.
- Keep control structures and data structures simple.

All classes must be tested by you prior to being submitted. You are welcome to share test code with each other.

The assignment is to be done alone. Each file should contain the following declaration in comments near the top. "This file was prepared by [your name here]. It was completed by me alone.". If you obtained help in doing the assignment, do not include this declaration, but rather an explanation of the nature of any help that you received in doing the assignment.

Q0. Boolean Expressions

[**Learning Objectives**: This question requires you to not only implement an interface in multiple ways but also to write some polymorphic code. In particular you should find that you have fields whose type is an interface and to make calls to methods whose exact implementation you do not know. You will also use an abstract class. This question uses the Abstract Factory pattern and the composite pattern. **end of learning objective**]

For this part, we will together create a simple program to print a truth table. For example, given an input of

```
a /\ b \/ ~(a /\ c)
```

the program will print the following

```
a       b       c       ((a /\ b) \/ ~ (a /\ c))
false   false   false   true
false   false   true    true
false   true    false   true
false   true    true    true
true    false   false   true
true    false   true    false
true    true    false   true
true    true    true    true
```

Luckily for you, I've written the hardest parts.
Here is how it works: (See class TruthTableMain.)

- Step 0. Two objects work together to create a tree representation of the input expression. One object is a Parser; it analyzes the input and decides how the tree should be built. The other class is an ExpressionFactory; it looks after the details of building the tree.

  The above example would be converted to a tree containing 8 nodes. One node for each occurrence of a variable and one for each of the 4 operators. Each node will be an object that implements the interface ExpressionNodeI, which I will provide.

- Step 1. The first line of the output is printed. This involves printing the tree.

- Step 2. The tree is 'evaluated' once for each possible assignment of values to variables. After each evaluation, the result is printed.

Your job is to create (or complete) the following classes:

- ExpressionFactory: Responsible for creating expression nodes.

- Various nonpublic classes that implement the ExpressionNodeI interface.

- Implement at least one abstract class that has at least 2 subclasses. For example, you could have one abstract class that is a super class for all your concrete implementations of ExpressionNodeI, or you could have an abstract class that is a super class for all classes that represent nodes with 2 children.

Each concrete class that implements the ExpressionNodeI interface must implement an appropriate constructor and 2 methods:

- The first is

  public boolean evaluate(Environment env) ;

  This method computes the value of the expression in a given environment. The environment maps each variable a boolean value (false or true). Each expression node object will have to evaluate its children (if any) and then compute and return the appropriate boolean value.

- The second is

  public void printTo(PrintWriter p) ;

  This method sends a textual representation of the expression to the given PrintWriter object. 'And'- and 'or'-expressions should always be surrounded by parentheses. (PrintWriter is an interface that supports a print(String) method.)

Submit the source code for your expressionTree.tree package as a single .zip file called tree.zip that contains your .java files directly.

Q1. Sets

[**Learning Objectives**: The first question requires you to implement an interface and to write some polymorphic code. In particular you should find that you have to implement methods with parameters whose type is an interface and to make calls to methods whose exact implementation you do not know. This assignment also requires you to write at least one iterator class, which will be a class whose objects are mutable. You will use the Abstract Factory pattern and the Iterator pattern. **end of learning objective**]

For this question you will implement an abstract data type representing finite sets of integers between -2^{31} up to (and including) $2^{31} - 1$.

Implement a public class set.implementation.SetFactory that implements interface set.SetFactoryI. All other classes should be nonpublic. Classes that implement the set.SetI interface be immutable (i.e., the objects they describe should be immutable) and should not be public. There are various data structures you could use to achieve this goal; you could even use different strategies depending on the nature of the set, e.g. large and sparse sets could represented one way, while small, dense sets could be represented another way.

You will need to read about the java.util.Iterator and java.util.Iterable interface. These are generic interfaces. Conceptually we want java.util.Iterator<**int**>, however Java does not allow primitive types as type arguments and so I used java.util.Iterator<Integer>. The java.lang.Integer class is a class that described immutable objects that contain an int value. Luckily conversions between Integer and **int** and usually implicit, so you can usually ignore the difference between Integer and **int**. For example you can write

```
for( int i : s ) { System.out.println(i) ; }
```

where s is a reference to a SetI object. The iterator object will deliver references to Integer objects, but they will be implicitly converted to **int** values before being assigned to i.

You will need to override the equals method for classes that implement the SetI interface. Writing equals methods can be tricky. I suggest that you use the following implementation:

```
@Override
public boolean equals( Object other ) {
    if( other instanceof SetI ) {
        SetI otherAsSetI = (SetI) other ;
        return otherAsSetI.subset(this) && this.subset( otherAsSetI ) ;
    } else {
        return false ;
    }
}
```

(It is conventional that, when you override 'equals' in Java, you should also override the 'hashCode' method. The reason is that any two objects that are equal should have the same hash code value, otherwise your class can not be used as the key type in hash tables. But you may ignore this convention, if you wish, for this assignment.)

Submit your implementation folder as a .zip file called implemention.zip that contains your source code directly in it.