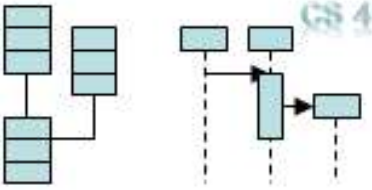

CS 414 Object Oriented Design

Spring 2018
Computer Science Department

Assignment A4

Colorado State University

Individual Assignment: Visitor Pattern

DUE: 11:59PM, Wednesday 28 February 2018

25 points

1. Objectives

The objectives are two-fold. The primary goal is to understand the use of the visitor design pattern to create applications by writing new concrete visitors without having to change the code corresponding to the basic object structure. A secondary goal is to be able to understand a medium-sized code-base that was either developed by someone else or generated by a code-generator. You will end up reading lots of code, but not have to write much of it yourself (less than 100 SLOC).

While you need to understand some terminology related to token analysis, parsing, grammars, and syntax trees, the coding for these will be kept to a minimum. In fact, most of this code will be generated and provided to you.

2. Visitor Background

First download the zip file that contains everything you need to begin the assignment. Unzip the file and note the contents. You will see an installation of jtb, javacc, and the JavaPrinter (contains code that you will work on). jtb and javacc were described in class, but you won't need to use them unless you want to regenerate the code that has been provided to you.

Extract the files from the zip file and you will find a root folder called `visitor`. Create an eclipse project from the `JavaPrinter` sub-folder as demonstrated in a lecture.

Refer to your lecture notes for a discussion of the visitor pattern and examples. In this assignment, you are given examples of several concrete visitors and you will need to implement one visitor.

We will use the Java Tree Builder (JTB) implementation of the visitor pattern. JTB is a front-end to the Java Compiler Compiler (JavaCC) parser generator. JTB takes a JavaCC grammar as input to generate the following (description taken from the JTB webpage http://compilers.cs.ucla.edu/jtb/):

- A set of syntax tree classes based on the productions in the grammar, utilizing the Visitor design pattern.
- Two interfaces: `Visitor` and `GJVisitor`. Two depth-first visitors: `DepthFirstVisitor` and `GJDepthFirst`, whose default methods simply visit the children of the current node.
- A JavaCC grammar `jtb.out.jj` with the proper annotations to build the syntax tree during parsing.
- Concrete visitors can be written by subclassing an existing visitor, such as `DepthFirstVisitor`, because you do not need to work with generics.

3. Problem Description

You will develop an application that takes an uncommented Java file and inserts comment skeletons for the following sections. The comments **should not** be indented. They should start from the left-most side. This is for simplicity in coding. That way, the visitor does not need to maintain any state information. It only needs to get some state information from the tree structure.

Do not modify the input file. Just output the commented program to standard output. Here is a sample input and output.

- Just before a new class is declared. You must handle nested classes that are named. Anonymous classes need to be ignored. For nested classes, output the following just above the class declaration (no line gaps between the end of comment and the class declaration):

```
/*************
New nested class NAME
*************/
```

  Otherwise, output the following just above the class declaration (no line gaps between the end of comment and the class declaration):

```
/*************
New class NAME
*************/
```

  In both cases, NAME refers to the class that is defined below the comment. Make sure you have the exact number of lines, whitespaces, *, etc, so that the grader can automatically compare the output. Copy and paste the above comment.

- Just before each method is declared. Output the following before the method declaration (no line gaps between the end of comment and the method declaration):

```
/*************
New method NAME
*************/
```

  Here, NAME refers to the method that is defined below the comment. Make sure you have the exact number of lines, whitespaces, *, etc, so that the grader can automatically compare the output.

- Just before each constructor is declared. Output the following before the constructor declaration (no line gaps between the end of comment and the constructor declaration):

```
/*************
New constructor NAME
*************/
```

  Here, NAME refers to the name of the constructor that is defined below the comment. Make sure you have the exact number of lines, whitespaces, *, etc, so that the grader can automatically compare the output.

- Just before each new variable (static or instance) has been declared. Output the following leaving no line gap between the end of comment and the start of the definition line.

```
// Class variable definition begins
```

  Make sure you cover all the cases, including the following:

```
private static final ClassName instanceVar;
public static final datatype(int, double,..) var;
int var; (implicit visibility)
public int var;
public Classname instance;
```

  Assume that only one variable is declared at a time. Variables declared inside methods (i.e., local variables) are not included.

- Just after each new variable (static or instance) has been declared. Output the following leaving no line gap between the end of definition and beginning of the comment.

```
// Class variable definition ends
```

  Make sure you cover all the cases, including the following:

```
private static final ClassName instanceVar;
public static final datatype(i.e., int, double,..) var;
int var; (implicit visibility)
public int var;
public Classname instance;
```

  Variables declared inside methods (i.e., local variables) are not included.

  As an example, the following output should occur:

```
// Class variable definition begins
        private static final double var1;
// Class variable definition ends
// Class variable definition begins
```

```
        private static final double var2;
// Class variable definition ends
```

Assume that the input Java program is syntactically correct with respect to the Java1.1 grammar. If it isn't, the JavaCC generated parser will just not work. It will give you a bunch of error messages when you run your visitor.

Your output program must compile correctly, i.e. inserting the comments should not change anything else in the input programs.

You must ensure that the comments are inserted as described above. No extra lines or spaces can be inserted because the grading script will flag them as errors even though you may be inserting the comments in the correct places. Copy-paste the comments shown in this document so that you can ensure correct formats.

---

4. Steps to follow

1. **Program the concrete visitor.**

   Create a file called `Commenter.java` inside the folder `visitor/JavaPrinter/visitor`. The class `Commenter` must be in the package `visitor` and must implement the concrete visitor to solve the problem described above. Find the concrete visitor called `TreeDumper.java` in the same directory and understand what it does, You must extend `TreeDumper` and override the necessary methods.

   To help keep the outputs consistent, You must use the `MainA4` file that is given to you, called `MainA4.java`. `MainA4` creates the `Commenter` visitor instance and uses it. Do not change `MainA4.java` file in any way. Your new code must be in exactly one file called `Commenter.java`.

2. **Compile the code and run it.**

   Either use eclipse or run the following commands to run the given example input:

   ```
   javac MainA4.java
   java MainA4 Test.java
   ```

---

5. Submission

Submit the `Commenter.java` file via Canvas.

---

6. Grading criteria

We will use the following criteria to award points. However, if your program does not compile, you will get **ZERO** points. This time (unlike in A1) we will not fix your program to enable compilation.

- Functionality : 24 points
  - 4 points for successfully commenting public classes
  - 4 points for successfully commenting nested classes
  - 4 points for successfully commenting methods
  - 4 points for successfully commenting constructors
  - 4 points for successfully commenting before the attributes
  - 4 points for successfully commenting after the attributes

  In each, part of the credit is assigned to correctly naming the entity being commented. Some part is allocated to what is printed, and the rest to where it is printed.

- One point is allocated to good programming style (e.g., appropriate commenting, naming conventions, etc).