CSC-223
Lab 5
Groups of 2 or 3.

Builders for JSON Parsing and *Visitors* Over Geometry Figure Trees

In the last assignment you implemented code to construct an abstract syntax tree representing a geometry figure. As a means of verification, you used object-based unparsing of the AST to observe correctness of an input geometry figure object. Our goal with this assignment is to encapsulate operations like unparsing into more cohesive class implementations using *Builder* and *Visitor* design patterns.

What You Need to Do: `Builder` Implementation

In your `JSONParser` class implementation from the previous assignment, you created the Abstract Syntax Tree (AST) based on an input geometry figure encoded in a JSON format. In particular, the `JSONParser::public parse(String)` method creates and returns the AST corresponding to the geometry figure passed in as the argument. This method uses a *recursive-descent* approach to build the tree. In this approach, a tree is constructed by the `JSONParser` using a collection of methods that each build a subtree of the tree. That is, each such method creates one kind of node and calls other methods in the collection using recursion to build each subtree of that node. A recursive-descent parser relies on the fact that there is a structure or "grammar" that defines a legal geometry figure.

Problem. One problem with our approach is that our parser is actually doing two things instead of one: it is both determining (i.e., parsing) the structure of the figure and it is building the AST. Suppose that the user wants merely to determine whether the input geometry figure's JSON structure is legal. In that case, the tree returned by the `JSONParser` is not needed, and so it would be nice if we could avoid the construction process.

Solution. The solution is to separate the parsing of the program from the construction of the AST. Hence, we will implement a separate `Builder` class that performs the construction: whenever the `JSONParser` reaches a point where, in the old version, it constructed a new node, it should instead ask the `Builder` to construct and return the new node for the parser. Then, we can substitute different builders to be used with our parser depending on our needs. If we want the AST to be constructed, we use a builder that actually constructs the nodes in the same way the old parser did. But if we just want to test the legality of the program, then nothing needs to be built, and so we can use an "empty" builder that returns `null` when asked to build a node. Furthermore, if we just want to count the number of nodes that would be created if we were to build the AST, we could use a builder that is identical to the empty builder except it keeps a count of the number of nodes it was asked to build.

This idea of separating the actual construction of a complex object from the directing of that construction is called the *Builder* design pattern. A natural way to design *Builder* classes is to create a "default" *Builder* class whose methods all return `null`. This class can then be subclassed whenever the user wants to actually construct part or all of the AST. In this way, the user needs to override only those methods of interest.

The complete code for the `DefaultBuilder` class has been provided.

Your tasks for this portion of the assignment will be to (1) implement a `GeometryBuilder` class that inherits from `DefaultBuilder` class, but construct objects for the AST as well as (2) modify JSONParser to use a *Builder* object.

What You Need to Do: `Visitor` Implementation

In the last assignment you implemented your traversal technique of an AST by adding an `unparse` method to several classes in the AST hierarchy. In that case, adding a method to each class was not necessarily onerous; however, if we were to implement many other traversal algorithms over a geometry figure AST, our AST classes would become cluttered (to say the least). **Our goal is to have one file contain all code related to one traversal technique.** For example, if we wish to unparse a geometry figure AST, we would want to place all code related to that idea in a dedicated *unparser* class. Similarly, if we would want to implement a class that is able to generate a JSON object for a geometry figure AST, we would implement a `ToJSON` class.

To avoid clutter and (more importantly) to maintain separation of functionality for each type of traversal, we will use a visitor. The *visitor design pattern* is a well-established paradigm to accomplish the goal of separating functionality with a syntax tree while also providing a clear, consistent interface to traverse the tree. We will describe our implementation below, but it is beneficial to read a more general description of the double dispatching technique here.

The interface of our visitor for `ComponentNode` classes is shown in Figure 1. Each class we intend to 'visit' must define a corresponding visit method. We observe in Figure 1 that each method first takes a specific `ComponentNode` type and second accepts a general `Object o`. Depending on the traversal technique being implemented by a visitor, the object can be 'anything' required. For example, with an unparser we might pass a `StringBuilder` object as our object `o`.

```
public interface ComponentNodeVisitor
{
    Object visitFigureNode(FigureNode node, Object o);
    Object visitSegmentDatabaseNode(SegmentNodeDatabase node, Object o);
    Object visitSegmentNode(SegmentNode node, Object o);
    Object visitPointNode(PointNode node, Object o);
    Object visitPointNodeDatabase(PointNodeDatabase node, Object o);
}
```

Figure 1: The definition of our visitor interface over `ComponentNode` objects.

Our double dispatching for visitors is defined by our `ComponentNode` interface requiring each of our nodes implement a 'general' `accept` method as shown in Figure 2; a sample `accept` method implementation is shown in Figure 3. Observe that `accept` simply calls (dispatches) back to the input visitor class.

```
public interface ComponentNode
{
    Object accept(ComponentNodeVisitor visitor, Object o);
}
```

Figure 2: The `ComponentNode` interface definition of our visitor interface over `ComponentNode` objects.

```
public class FigureNode implements ComponentNode
{
  ...
    @Override
    public Object accept(ComponentNodeVisitor visitor, Object o)
    {
        return visitor.visitFigureNode(this, o);
    }
}
```

Figure 3: An example of the `accept` method for `FigureNode`.

```
node.getPointsDatabase().accept(this, <some object>);
```

Figure 4: A sample visitor call from `visitFigureNode` to `visitPointNodeDatabase`.

As an example of a visitor calling another method, consider unparsing a `FigureNode`'s point database object as shown in Figure 4. Observe that we can call `accept` on any other `ComponentNode` object with the current visitor object (`this`) and 'automatically' the correct visitor method for our object will be called (in this case our `PointNodeDatabase` object). To better understand this idea of double-dispatch, draw some pictures that show the relationship between a visitor class method and the corresponding `ComponentNode::accept` method and, in turn, what the `accept` method calls.

Unparser. One of your tasks will be to implement the `UnparseVisitor` class that inherits from `ComponentNodeVisitor` class that populates a `Stringbuilder` object similar to unparsing from the last assignment. Recall that unparsing requires two arguments: a `StringBuilder` and an indentation level. However, our general visitor definition allows only one object be passed to a visitor. In order to pass two objects, we will use a bit of a hack: the `SimpleEntry` object defined in `AbstractMap`; see Figure 5 and Figure 6 for how to respectively 'pack' and 'unpack' a `SimpleEntry` object during a visit.

```
StringBuilder sb = new StringBuilder();
UnparseVisitor unparser = new UnparseVisitor();
unparser.visitFigureNode((FigureNode)node,
                       new AbstractMap.SimpleEntry<StringBuilder, Integer>(sb, 0));
```

Figure 5: 'Packing' and calling the visitor unparser.

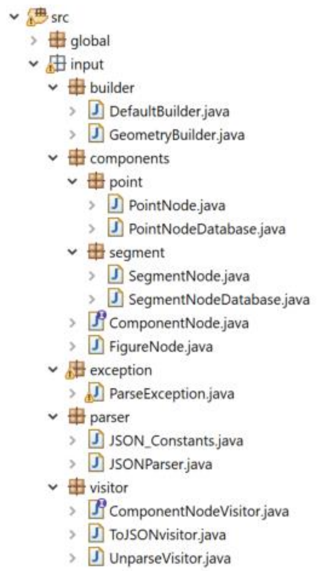
```
// Unpack the input object containing a Stringbuilder and an indentation level
@SuppressWarnings("unchecked")
AbstractMap.SimpleEntry<StringBuilder, Integer> pair =
                          (AbstractMap.SimpleEntry<StringBuilder, Integer>)(o);
StringBuilder sb = pair.getKey();
int level = pair.getValue();
```

Figure 6: 'Unpacking' inside a visit call in `UnparseVisitor`.

JSON Writer. Your last task is to implement `ToJSONvisitor`, a visitor that will convert the AST of a geometry figure back into a `JSONObject`. To verify construction, use `toString(int)` on the `JSONObject`. Observe that the tree structure of our unparsed output is a bit cleaner compared to `org.json` implementation.

Project Structure

A few code files have been provided. Some you will create. Some already exist from prior assignments. Please adhere to the consistent project structure shown in Figure 7.



```
src
  global
  input
    builder
      DefaultBuilder.java
      GeometryBuilder.java
    components
      point
        PointNode.java
        PointNodeDatabase.java
      segment
        SegmentNode.java
        SegmentNodeDatabase.java
      ComponentNode.java
      FigureNode.java
    exception
      ParseException.java
    parser
      JSON_Constants.java
      JSONParser.java
    visitor
      ComponentNodeVisitor.java
      ToJSONvisitor.java
      UnparseVisitor.java
```

Figure 7: The desired project structure for this assignment.

Commenting

Comment well. See old lab instructions for details.

Submitting: Source Code

For this lab, you will demonstrate your source code to the instructor, in person. Be ready to demonstrate (1) successful execution of all junit tests and (2) the github repository which includes commented source code (see above) and a clear commit history with meaningful commit messages *from all group members*.

Content Note

Elements of this assignment description were excerpted and / or adopted from Object-Oriented Design Using Java by Dale Skrien.