Hands-On 2 - Design Patterns

February 2023

arnaud.nauwynck@gmail.com

This document: https://github.com/Arnaud-Nauwynck/presentations
/tree/main/java/TP-design-patterns/handson-2-design-patterns.pptx

Outline

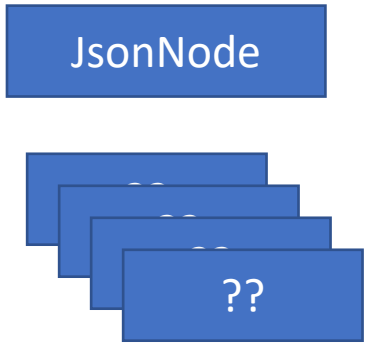
The goal of this Hands-On is to recognize and model as UML Classes many Design Patterns, for FileFormat / Grammar / Compiler / Math

Composite,
Proxy,
Adapter,
Interpreter,
Visitor,
…

Exercise 1 : model JSON schema in UML classes

test.json

```json
{
    "field0": null,
    "field1": false,
    "field2": 123,
    "field3": 3.1425926,
    "field4": [  false, 123, 3.14, [], {} ],
    "field5": {
        "subField1": 0
    }
}
```



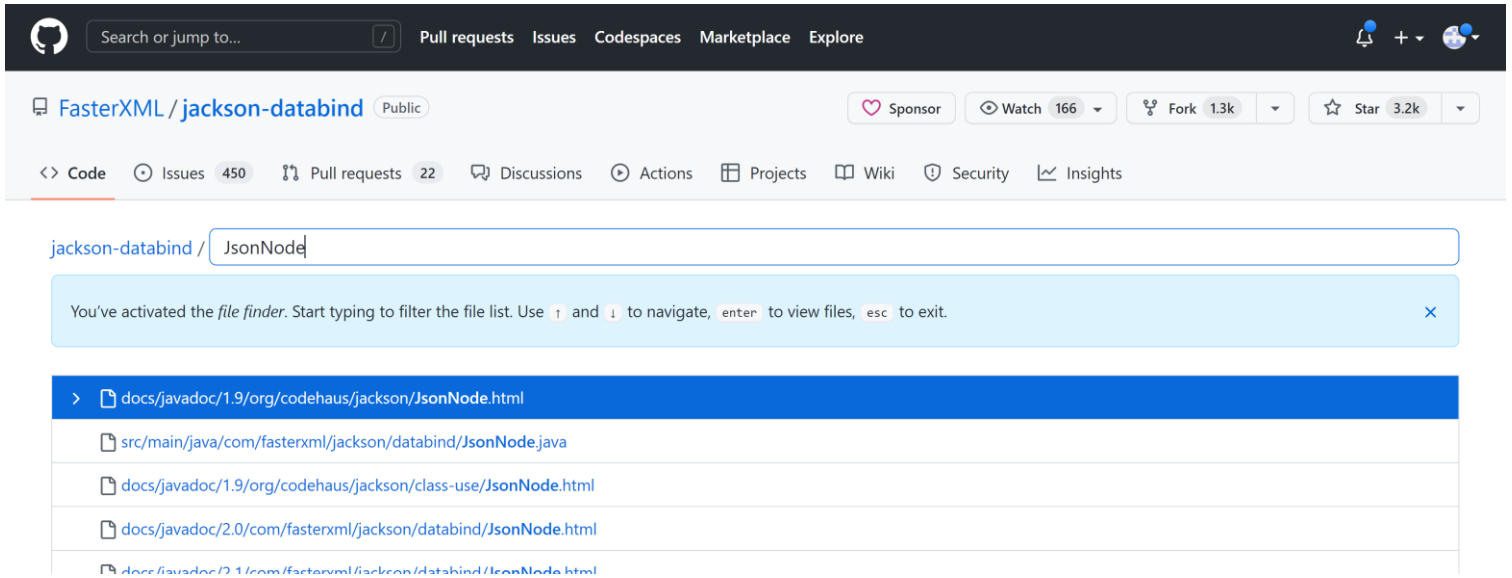
JSON grammar:

- « json element » are called « JsonNode »
- null is a valid element
- There are terminal elements with values « Number », « Text », « Boolean »
- Elements can be combined in array: [element1, element2, … elementN]
- Elements can be combined in Object: { name1: element1, name2: element2, … }

Exercise 2: Source code of Java Json library: « Jackson »

a/ open url https://github.com/FasterXML/jackson-databind

b/ type « T » (github search for file) .. Then type « JsonNode »

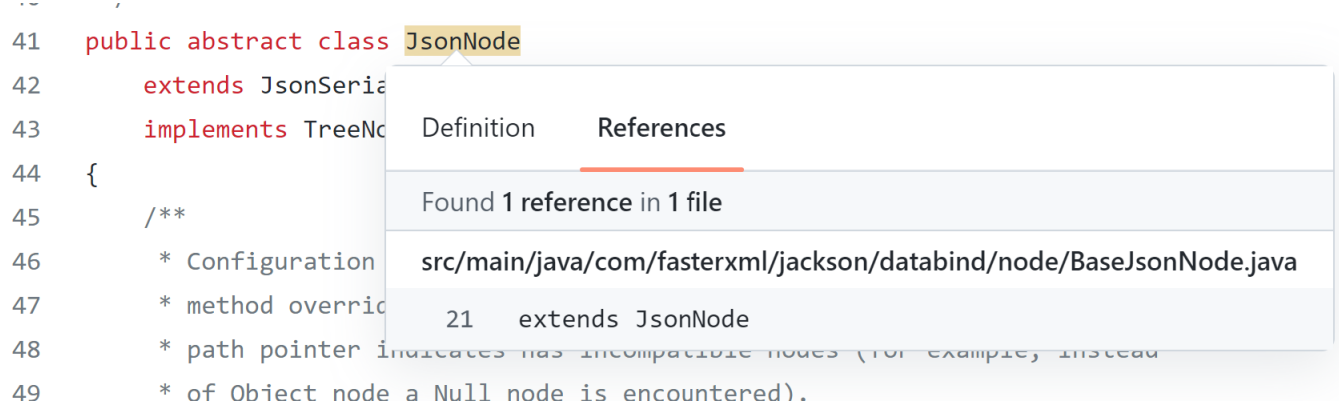


c/ read first javadoc lines

```
15  /**
16   * Base class for all JSON nodes, which form the basis of JSON
17   * Tree Model that Jackson implements.
18   * One way to think of these nodes is to consider them
19   * similar to DOM nodes in XML DOM trees.
```

Exercise 3 : search sub-classes hierarchy of JsonNode in jackson

In github directly



Or download source-code

```
$ git clone --depth 1 https://github.com/FasterXML/jackson-databind
Cloning into 'jackson-databind'...
```

cd src/main/java
grep -R "extends BaseJsonNode"
grep -R "extends ContainerNode"
grep -R "extends ValueNode"

Exercise 4: create a Java Project, + Re-implement a minimal JsonNode class and sub-classes

```
public abstract class JsonNode {

  // nested static classes: all known sub-classes  of JsonNode

  public static class NullJsonNode extends JsonNode {}
  public static class TextJsonNode extends JsonNode {  public String value; }
  public static class NumericJsonNode extends JsonNode {  public double value; }
  public static class BooleanJsonNode extends JsonNode {  public boolean value; }

  public static class ArrayJsonNode extends JsonNode {  public List<JsonNode> child; }
  public static class ObjectJsonNode extends JsonNode {  public Map<String,JsonNode> fields; }

}
```

Exercise 5: abstract methods…

Declare abstract method to « clone » any JsonNode, and recursively

=> Override all sub-classes, to implement correctly

Declare another abstract method to « dump as text » any JsonNode, and recursively

=> Override all sub-classes, to implement correctly

Exercise 6 : check code caracteristics

Characteristic 1: the model AST classes are polluted with any applicative code

Characteristic 2: each model AST class contains many methods,
the code become intrinsically complex (many features)

Characteristic 3: each features is split between several classes

Exercise 5 : add the Visitor design-pattern (to ex 4)

```
public abstract class JsonNode {

  public abstract void visit(JsonNodeVisitor visitor);

  // nested static classes: all known sub-classes  of JsonNode
  // for all XXXX sub-class
  public static class XXXXJsonNode extends JsonNode {
      @Override
      public void visit(JsonNodeVisitor visitor) {
            visitor.caseXXXX(this);
      }
   }

}
```

```
public abstract class JsonNodeVisitor {

   public abstract caseNull(NullJsonNode p);
   public abstract caseText(TextJsonNode p);
   public abstract caseNumber(NumberJsonNode p);
   ….

}
```

Exercise 6: implement a concrete JsonNodeVisitor to clone (recursively) a Node

```
public class JsonNodeCloner extends JsonNodeVisitor {

   public JsonNode result;

   @Override
   public caseNull(NullJsonNode p) {  .. }
  @Override
   public abstract caseText(TextJsonNode p) {  .. }
  @Override
   public abstract caseNumber(NumberJsonNode p) {  .. }
   ….

}
```

Exercise 7 : call the visitor, write Junit test(s)

```
public class JsonNodeClonerTest {
   // sut = System Under Test
   JsonCloner sut = new JsonCloner();

  @Test
  public void testClone() {
    // given
    JsonNode node = …
    // when
    sut.visit(node);
    // then
    JsonNode clonedRes = sut.result;
    Assert.assertEquals(…, clonedRes…);
  }

}
```

Exercise 8 : write another Visitor, to dump as indented Json text

```
public class JsonNodeDumper extends JsonNodeVisitor {

    private final PrintStream out;
    private int indentLevel = 0;

    public JsonNodeDumper(PrintStream out) { this.out = out; }

    protected void incrIndent() { this.indentLevel++; }
    protected void decrIndent() { this.indentLevel--; }
    protected void printIndent() { for(int i = 0; i < indentLevell; i++) out.print(«  »); }
    protected void printIndentedLine(String text) { printIndent(); out.print(text); out.print(«\n»); }

   @Override
   public caseNull(NullJsonNode p) { printIndentedLine(«null») ;  }
   @Override
   public abstract caseText(TextJsonNode p) {  .. }
   …

}
```

Exercise 9: … check code caracteristics

Characteristic 1: the model AST classes are not polluted with any applicative code

Characteristic 2: the model AST class contains only getter/setter/constructor + Visitor

Characteristic 3: different features are written in separated Visitor classes,
1 feature = 1 Visitor

Characteristic 4: each Visitor has his own private utility method, encapsulated
(example: indentation logic)