Assginment 1 – Software Design and Modelling

Volodymyr Karpenko   Claudio Maggioni

October 24, 2022

Contents

Listings

1 Project selection process

We have to choose a Java-based project on GitHub that follows the following requirements:

- 100 or more stars;
- 100 or more forks;
- 10 or more open issues;
- 50.000 or more lines of code.

Additionally, we added some less strict constraints that we thought would lead to a more significant and influential analysis:

- There must be evidence that the project follows business-oriented conventions. This excludes amateur or personal projects that might have fewer design pattern applications due to their nature.
- Repository data, documentation, and comments must be written in English. Many repositories that are at the top of the search results provided by the hard requirements are not in English, and this drastically hampers our ability to understand the code;
- The artifact the project produces must not rely on external components and have a streamlined build process, with all code stored in a single Maven/Gradle module. This improves our ability to tinker with the project more quickly and the pattern detection process, which requires all *.class* files related to the project to be stored in a single directory tree.

Additionally, instead of querying GitHub directly for projects, we decided to see if libraries we knew already in our Java development career would match the hard and soft requirements we set for ourselves.

Therefore, we considered the following GitHub repositories:

vavr-io/vavr a Java library for functional programming, was discarded as the project is less than 20.000 lines of code and does not meet the rigid requirements;

bitcoin4j/bitcoin4j a Java implementation of the bitcoin protocol, discarded as the project is distributed in several subprojects, and therefore the build process is nontrivial;

FasterXML/jackson-core is the core ”module“ of a Java JSON serialization and deserialization library. We chose this project because it meets the selection criteria. It does not rely on external components for its execution. Finally, the project structure uses a single Maven module for its sources and is thus easy to analyze.

1.1 The Jackson Core Project

As mentioned, Jackson is a library that offers serialization and deserialization capabilities in JSON format. It is highly extensible and customizable through a robust but flexible API. The library is divided into what the Jackson developers call “modules,” i.e., plug-ins that can augment the serialization and deserialization process. Some modules, like the *jackson-dataformat-xml* module, target different encoding languages like XML.

The chosen repository contains only the *core* module of Jackson. The *core* module implements the necessary library abstractions and interfaces to allow other modules to be plugged-in. Additionally, the *core* module implements the tokenizer and low-level abstractions to work with the JSON format. We will refer to this module as “Jackson” or “Jackson Core” interchangeably throughout this report.

We choose to analyze version 2.13.4 of the module (i.e. the code under the *git* tag *jackson-core-2.13.4*) because it is the latest stable version available at the time of writing.

After verifying that the project meets the hard requirements related to GitHub (more than 2000 stars, more than 600 forks, 35 open issues[1]), we ensured that the project had enough lines of code by using the cloc tool, which provided the following output shown in Figure 1. By looking at the results we can finally assert that the project contains 58.787 lines of Java code and this satisfies all the requirements.

Language	Files	Blank	Comment	Code
HTML	4846	18473	235544	1997020
Java	285	8532	20004	48783
CSS	3	18	69	990
Logos	2	260	212	605
Bourne Shell	3	35	62	223
XML	7	5	1	179
DOS Batch	1	35	0	153
Markdown	3	58	0	125
Maven	1	13	23	112
YAML	3	1	5	71
JavaScript	1	1	0	29
JSON	1	0	0	10
Properties	2	0	16	5
Total	5158	27431	255936	2048305

Figure 1: Output of the *cloc* tool for the Jackson Core project at revision *jackson-core-3.13.4*.

2 Analysis Implementation

The analysis is performed using the *Pattern4J*[2] developed at Concordia University. This program attempts to detect traditional design patterns by scanning the bytecode (i.e. the `.class` files) of a given project and by checking several heuristics. Due to the unceirtanty of this process we double-check each instance of a pattern found to use our own judgement and detect possible false positives.

Since the tool needs compiled *.class* files to perform the analysis, and since *jackson-core* is a standard Maven project, we compile the sources using the command `mvn clean compile`. The *pom.xml* of the library specifies Java 1.6 as a build target, which is not supported by JDK 17 or above. We used JDK 11 instead, as it is the most recent LTS version of the JDK to still support this target.

An XML dump of the *Pattern4j* analysis results is included in the submission as the file *analysis.xml*.

In the following sections each detection of the *Pattern4J* tool is reviewed to characterize if it is indeed not a false positive and if the design pattern is varied in any way in its application. For the sake of brevity, when referring to a class by its fully-qualified nomain name the prefix *com.fasterxml.jackson.core* is omitted as all classes in the Jackson core project reside in this package or in a sub-package of this package.

3 Structural Patterns

3.1 TO REWRITE Singleton Pattern

Ensure a class only has one instance and provide a global point of access to it. The tool found thirteen instances with the Singleton pattern. Doing a deeper analysis of the found instances, we

[1]as of 2022-10-19 (ISO 8601 date)

[2]`https://users.encs.concordia.ca/~nikolaos/pattern_detection.html`

discovered that some results are false positives. Example, *sym.Name1* has a private package constructor and a public static final instance of it, but reading the documentation, the class represents (short) JSON string literals and is initialized by client code.

```
public final class Name1 extends Name {
    private final static Name1 EMPTY = new Name1("", 0, 0);
    private final int q;

    Name1(String name, int hash, int quad) {
        super(name, hash);
        q = quad;
    }

    public static Name1 getEmptyName() {
        return EMPTY;
    }

    @Override
    public boolean equals(int quad) {
        return (quad == q);
    }

    @Override
    public boolean equals(int quad1, int quad2) {
        return (quad1 == q) && (quad2 == 0);
    }

    @Override
    public boolean equals(int q1, int q2, int q3) {
        return false;
    }

    @Override
    public boolean equals(int[] quads, int qlen) {
        return (qlen == 1 && quads[0] == q);
    }
}
```

Listing 1: Name1 class

sym.Name1, JsonLocation, DefaultIndenter, util.DefaultPrettyPrinter$FixedSpaceIndenter is not a singleton (detected cause of "convenient" default instance given as static final field), the constructor is not used, but the class is extensible

JsonPointer, filter.TokenFilter is like the above, but constructors are protected

JsonpCharacterEscapes, util.DefaultPrettyPrinter$NopIndenter, Version a singleton but with a public constructor that is never called in the module code, may be called in tests

io.JsonStringEncoder like above, but the class is final

util.InternCache, io.CharTypes$AltEscapes actual singleton, thread-unsafe initialization

io.ContentReference like above, but constructor is protected

3.2 Abstract Factory Pattern

Pattern4J detects only two instances of the abstract factory pattern:

TokenStreamFactory which indeed is a factory for **JsonParser** and **JsonGenerator** is a factory for JsonParser and JsonGenerator objects, although two overloaded factory methods exist on this class (one for each class) catering to a different combination of arguments. A concrete implementation of this factory is included in the form of the **JsonFactory** class, although other modules may add additional implementations to cater to different encodings (like the *jackson-dataformat-xml* module for XML);

TSFBuilder which is also a factory for concrete implementations of **TokenStreamFactory** allows slight changes in the serialization and deserialization rules (e.g., changing the quote character used in JSON keys from " to '). Like TokenStreamFactory, this class is only implemented by one class, JsonFactoryBuilder, within this module's scope. Moreover, as mentioned previously, this abstract factory will likely be extended by concrete implementations in other Jackson modules.

3.3 Builder Pattern

The builder pattern does not seem to be analyzed by *Pattern4J*, as the analysis output does not mention the pattern, even to report that no instances of it have been found (as is the case with other patterns, e.g., the observer pattern). A manual search in the source code produced the following results:

TSFBuilder is also a builder other than an abstract factory. As mentioned previously, this class allows slightly altering the serialization and deserialization rules used to build outputting JsonFactory objects. Each rule is represented by an object or enum instance implementing the util.JacksonFeature interface. TSFBuilder then provides several overloaded methods to enable and disable features represented by the interface. Enabled features are stored in several bitmask-protected int fields, which are then directly accessed by the constructor of the TokenStreamFactory concrete implementation to build;

JsonFactoryBuilder is a concrete factory implementation of **TSFBuilder** that builds **JsonFactory** instances;

util.ByteArrayBuilder provides facilities to build *byte[]* objects of varying length, akin to **StringBuilder** building **String** objects. This is not a strict implementation of the builder pattern per se (as Java arrays do not have a "real" constructor), but it is nevertheless included since the features it exposes (namely dynamic sizing while building) are decoupled by the underlying (fixed-size) array representation.

4 Creational Patterns

4.1 Adapter Pattern

TBD

4.2 Decorator Pattern

Decorator pattern lets you dynamically change the behavior of an object at run time by wrapping them in an object of a decorator class.

(com.fasterxml.jackson.core omitted for brevity)

JsonGenerator TBD

JsonParser TBD

Only in Pattern4j

4.3 Bridge Pattern

TBD

4.4 Composite Pattern

None found

4.5 Facade Pattern

TBD – *Pattern4J* does not detect this pattern

4.6 Proxy Pattern

None found

5 Behavioral Patterns

5.1 Command Pattern

None found

5.2 Observer Pattern

None found

5.3 State Pattern

Among the design patterns *Pattern4J* detects, the state pattern is detected in 5 classes. The state pattern is a variation of the strategy pattern where the concrete strategy used by the matching context is determined by the state of a finite state machine the context class implements. In other words, the state pattern chooses the concrete strategy to use through the state of the context.

By analyzing the *Pattern4J* results and the code, we can say that all the instances of the state pattern the tool finds are false positives. Namely:

util.DefaultPrettyPrinter *_inputDecorator* and *_outputDecorator* are fields flagged as states, thus flagging the class as a state pattern instance. However, no "state" akin to a finite-state machine is maintained by the class to determine which implementation of these fields to invoke. What is detected are more likely lightweight implementations of the strategy pattern, since these fields can be mutated through matching getters and setters. Additionally, the documentation of each of the matching *...Decorator* field types (namely interfaces) states that implementors are meant to be algorithms to pre-process input before the formatting process (a feature labeled as "decorator" w.r.t. the library, not to be confused with the decorator pattern);

util.DefaultPrettyPrinter _objectIndenter and _arrayIndenter are false positives too, and are likely strategy patterns too for the reasons described above.

util.DefaultPrettyPrinter _rootValueSeparator is flagged as a state field too, however the field is simply a boxed *String-like* immutable data structure (i.e. *SerializableString*) that is swapped during the pretty-printer parsing logic;

json.WriterBasedJsonGenerator _currentEscape is a false positive for the same reasons described above.

5.4 Strategy Pattern

Pattern4j detects no instance of the strategy pattern in Jackson, however the previous section regarding the state pattern referenced some false positives that were indeed applications of this pattern. Due to the flexibility of Jackson, there are many more instances of the strategy pattern to configure and customize the serialization and deserialization pipeline in several stages.

5.5 Template Method Pattern

Due to the extendibility of Jackson, it is of no surprise that the template method pattern is used extensively to create a class hierarchy that provides rich interfaces while maintaining behavioural flexibility. *Pattern4J* correctly detects several instances of the pattern, namely **JsonStreamContext**, **JsonGenerator**, **type.ResolvedType**, **JsonParser**, **base.ParserBase**, **base.GeneratorBase**, **base.ParserMinimalBase**. All these classes implement several concrete *public* methods throwgh the use of *protected abstract* methods. Although the concrete (i.e. the template) methods are usually not vary complex (as the pattern example shown in class), they still follow the principles of the template method pattern. We show as an example some template methods found in **base.ParserBase**:

```
@Override public void close() throws IOException {
    if (!_closed) {
        // 19-Jan-2018, tatu: as per [core#440] need to ensure no more data
        // assumed available
        _inputPtr = Math.max(_inputPtr, _inputEnd);
        _closed = true;
        try {
            _closeInput();
        } finally {
            // as per [JACKSON-324], do in finally block
            // Also, internal buffer(s) can now be released as well
            _releaseBuffers();
        }
    }
}

protected abstract void _closeInput() throws IOException;

protected void _releaseBuffers() throws IOException {
  /* implementation omitted */
}
```

Listing 2: Template method *void close()* and step methods *void _closeInput()* and *void _releaseBuffers()* in **base.ParserBase**.

Here the pattern is slightly modified by providing a default implementation of *void _releaseBuffers()*. In this case, child classes occasionally override the method with a body first calling *super()* and then adding additional buffer release code after.

5.6 Visitor Pattern

None found