Assginment 1 – Software Design and Modelling

Volodymyr Karpenko Claudio Maggioni

October 18, 2022

1 Project selection process

We need to find a project that is a single unit in terms of compilation modules[1] self contained and with as little external dependencies as possible to ease the analysis project. Additionally, it would be nice if we choose a project that we already know as library clients.

1.1 Projects Considered

We considered the following GitHub repositories:

vavr-io/vavr a Java library for functional programming, discarded as the project is less than 20K LOC and doesn't meet the selection criteria;

bitcoin4j/bitcoin4j a Java implementation of the bitcoin protocol, discarded as the project is distributed in several subprojects;

FasterXML/jackson-core a Java JSON serialization and deserialization library. We chose this library because it meets the selection criteria, it doesn't rely on external components for its execution, and its project structure uses a single Maven module for its sources and thus easy to analyze.

1.2 The Jackson Core Library

As already mentioned, `Jackson` is a library that offers serialization and deseralization capabilities in JSON format. The library is highly extensible and customizable through a robust but flexible API and module suite that allows to change the serialization and deserialization rules, or in the case of the `jackson-dataformat-xml` module, to allow to target XML instead of JSON.

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 chose to analyze version 2.13.4 of the module (corresponding to the code under the git tag `jackson-core-2.13.4`) because it is the latest stable version available at the time of writing.

[1] A problem for Pattern4J as compiled `.class` files are distributed across several directories and would have to be merged manually for analyzing them

2 Analysis Implementation

We use *Pattern4* as a pattern detection tool. This tool needs compiled `.class` files in order to perform analysis. Therefore, as `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 compilation target, which is not supported by JDK 17 or above. We used JDK 11 instead, as it is the previous LTS version.

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

3 Structural Patterns

3.1 Singleton Pattern

Lots of false positives for the Singleton pattern. Example, com.fasterxml.jackson.core.sym.Name1 has a package private constructor and a public static final instance of it, but reading the documentation the class represents (short) JSON string literals and therefore is clearly initialized by client code.

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

sym.Name1, JsonLocation, DefaultIndenter, util.DefaultPrettyPrinter$FixedSpaceIndenter 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 like 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

Pattern4 detects only two instances of the abstract factory pattern:

TokenStreamFactory which indeed is a factory for **JsonParser** and **JsonGenerator** objects, although two overloaded factory methods exist on this class (one for each class) catering for 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 for different encodings (like the `jackson-dataformat-xml` module for XML);

TSFBuilder which is also a factory for concrete implementations of **TokenStreamFactory** to allow 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, namely **JsonFactoryBuilder**, whitin the scope of this module. And as mentioned previously, this abstract factory is also likely to be extended by concrete implementations in other *Jackson* modules.

3.3 Builder Pattern

The builder pattern does not seem to be analyzed by *Pattern4*, as the analysis output does not mention the pattern, even just to report that no instances of it have been found (as it 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 to alter slightly the serialization and deserialization rules used to build outputttting **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 an 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 Bridge Pattern

TBD

4.3 Composite Pattern

TBD

4.4 Facade Pattern

TBD

4.5 Proxy Pattern

TBD

5 Behavioral Patterns

5.1 Command Pattern

TBD

5.2 Observer Pattern

TBD

5.3 Strategy Pattern

TBD

5.4 Template Method Pattern

TBD

5.5 Visitor Pattern

TBD