Assginment 1 – Software Design and Modelling

Volodymyr Karpenko Claudio Maggioni

October 26, 2022

Contents

1 Project selection process **1**
1.1 The Jackson Core Project 1

2 Analysis Implementation **2**

3 *Pattern4J* Accuracy and Quantitative Analysis **2**

4 Structural Patterns **3**
4.1 Singleton Pattern 3
4.2 Abstract Factory Pattern 4
4.3 Builder Pattern 5

5 Creational Patterns **5**
5.1 Adapter Pattern 5
5.2 Decorator Pattern 6
5.3 Bridge Pattern 7

6 Behavioral Patterns **7**
6.1 State Pattern 7
6.2 Strategy Pattern 8
6.3 Template Method Pattern 8

7 TBD Conclusions **9**

List of Figures

1 Output of the *cloc* tool for the Jackson Core project at revision *jackson-core-3.13.4*. 2
3 Quantitative summary of *Pattern4J* complete analysis output for the Jackson core project. 3
2 Results of the statistical analysis on the effectiveness of the *Pattern4J* tool as a class-by-class table (Figure 2a) and as a confusion matrix (Figure 2b). 10

Listings

1 The *sym.Name1* class. 3
2 Display the number of commits on a file and sort it by the number of commits. . . . 6
3 Command of Listing 2 executed on util.JsonParserDelegate class 6
4 Command of Listing 2 executed on util.JsonGeneratorDelegate class 6

5 Template method *void close()* and step methods *void _closeInput()* and *void _releaseBuffers()* in **base.ParserBase**. 8

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 non-blank lines of Java code and this satisfies all the requirements.

Language	Files	Blank	Comment	Code
HTML	4,846	18,473	235,544	1,997,020
Java	285	8,532	20,004	48,783
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	5,158	27,431	255,936	2,048,305

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 domain 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 *Pattern4J* Accuracy and Quantitative Analysis

As it would be very hard to check each class in the Jackson project for design patterns manually to get a true number of false positives and false negatives, we opt for a more statistical approach.

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

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

We select 20 classes at random from the project and we review the reported results for this subset. The classes were selected from the *target/classes* directory generated by Maven using the following command:

```
find . -name '*.class' | shuf -n 20 | sed 's#\.class##;s#/#.#'
```

The selected classes and the analysis results for both *Pattern4J* and our manual inspection are shown in Figure 2. Using those results, we can say that *Pattern4J* shows a false positive rate[3] of 53.8%, a false negative rate of 11.1%, and an accuracy of 75.0%.

Moreover, based on *Pattern4J* complete analysis output, we are able to report the overall findings, shown in Figure 3.

Design Pattern	Pattern Applications	Classes Covered
Factory Method	2	2
Singleton	13	13
Adapter	8	6
Decorator	2	2
State	5	3
Bridge	1	1
Template Method	7	7
Total	38	34

Figure 3: Quantitative summary of *Pattern4J* complete analysis output for the Jackson core project.

4 Structural Patterns

4.1 Singleton Pattern

Pattern4J found a lot of instances of the singleton pattern, namely 13. However, some discussion is required to understand the ways the Jackson core project applies this pattern, as the instances found are sometimes wildly different from the standard application or outright false positives.

```
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);
    }
```

[3] https://en.wikipedia.org/wiki/False_positive_rate

```
    @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: The *sym.Name1* class.

For example, *sym.Name1* (whose sources are in Listing 1) has a package-private constructor and a *public static final* instance of itself. This is enough for *Pattern4J* to flag the class as a singleton, as its constructor is never called in Jackson core other than for initializing the aforementioned field. However, by reading the documentation it is clear that the class is meant to be instantiated multiple times. Indeed, its purpose is to box and represent JSON string literals shorter than 4 bytes, implying the class is meant to be initialized by clients of the core Jackson module.

Several less-than-obvious results like this one are reported by the tool, namely:

sym.Name1, JsonLocation, DefaultIndenter, util.DefaultPrettyPrinter$FixedSpaceIndenter are not singletons and thus false positives. All these classes were detected because of "default" instances they include in themselves as *static final* fields and because their constructor, even if *public*, is never used in Jackson core itself. However, by checking the documentation all these classes are meant to be extended and instantiated in other Jackson modules;

JsonPointer, filter.TokenFilter are as described above, however having *protected* constructors;

JsonpCharacterEscapes, util.DefaultPrettyPrinter$NopIndenter, Version may be considered variations of the singleton pattern that however include a *public* constructor that is never called in the module code, but that may be called in tests. Given the *public* constructors, these classes are hardly solid singleton implementations. However, we gave the benefit of the doubt to Jackson developers as the use of the constructors in test code may hint to a purposefully open design to allow for testability;

io.JsonStringEncoder is as described above, however the class is declared as *final*;

util.InternCache, io.CharTypes$AltEscapes are both rather standard singleton pattern applications, however implemented with eager (non-lazy) initialization (i.e. storing the instance in a *public static final* field);

io.ContentReference is as described above, however having a *protected* constructor instead of a *private* one.

4.2 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.

4.3 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.

5 Creational Patterns

5.1 Adapter Pattern

Pattern4J found many instances of the adapter pattern, however all but one were shown to be false positives by checking the documentation and the code using each alleged adaptee. The matches found are reported and commented below. Matches are shown in the *[Adapter]* ← *[Adaptee]* format.

JsonFactory ← { **sym.ByteQuadsCanonicalizer, io.InputDecorator** } false positives, by reading the documentation it is clear the classes have different purposes and **JsonFactory** is merely using the other classes' functionality through composition;

base.ParserBase ← **json.JsonReadContext** false positive, **json.JsonReadContext** is instantiated several times in **base.ParserBase** by mutating the container field with the new instances;

base.ParserBase ← **json.JsonWriteContext** false positive, like above;

{ **util.DefaultPrettyPrinter, util.MinimalPrettyPrinter** } ← **util.Separators** false positives, another example of instances used through composition;

io.SerializedString ← **io.JsonStringEncoder** indeed an adapter pattern application, although the adaptee backing field is *private static final*. **io.SerializedString** is a class that wraps a String and allows it to be encoded using the **io.JsonStringEncoder** static instance, storing

the result and re-using it in case of multiple serialization request (a tecnique similar to memoization). Therefore, the main purpose of this adapter is not to adapt against any interface, but to wrap the functionality of the adaptee and store its results for re-use;

util.DefaultPrettyPrinter ← util.DefaultPrettyPrinter$Indenter (2 fields) false positives, both enclosed fields are simply used in a composition relationship.

5.2 Decorator Pattern

Decorator pattern lets you dynamically change the behaviour of an object at run time by wrapping them in an object of a decorator class. *Pattern4J* found two instances of the decorator patter, however we are probably in a case of ambiguity because in the documentation of the **util.JsonParserDelegate** is stated that the patter used is a delegation pattern.

```
/**
 * Helper class that implements
 * <a href="http://en.wikipedia.org/wiki/Delegation_pattern">delegation pattern</a>
 * for {@link JsonParser},
 * to allow for simple overridability of basic parsing functionality.
 * The idea is that any functionality to be modified can be simply
 * overridden; and anything else will be delegated by default.
 */
```

The decorator patter is similar to the delegation patter and the delegation patter is similar to the proxy patter, in-fact the delegation patter is also known as "proxy chains". We have a mismatch in this case because the decorato pattern uses a lot the delegation pattern to accomplish its task, so thats why we have a misclassification signalling the decorator pattern and not the proxy patter.

util.JsonParserDelegate is clearly a false positive because the developer stated explicitly the design patter used for the class. This class extends the **JsonParser** and fully implements the delegation patter.

util.JsonGeneratorDelegate there is no clear statement by the developer if the used designed pattern is different from the one detected by the tool, but we can make an assumption by looking on the authors contribution to the file.

```
git shortlog -n -s -- filename
```

Listing 2: Display the number of commits on a file and sort it by the number of commits.

Using the command presente in Listing 2 we can see the number of authors of a single file and the result is sorted by the number of commits. By running the command on **util.JsonParserDelegate** and **util.JsonGeneratorDelegate** we can clearly see that the top contributor to the file is the same for both the classes, the out-put of the command are presented for the classes in Listing 3 and Listing 4.

```
git shortlog -n -s -- JsonParserDelegate.java
 33 Tatu Saloranta
  3 Tatu
  1 Andrey Somov
  1 LokeshN
```

Listing 3: Command of Listing 2 executed on util.JsonParserDelegate class

```
git shortlog -n -s -- JsonGeneratorDelegate.java
38 Tatu Saloranta
```

```
2 Cowtowncoder
1 Andrey Somov
1 Logan Widick
1 Martin Steiger
1 Oleksandr Poslavskyi
1 Q.P.Liu
1 Volkan Yazıcı
```

Listing 4: Command of Listing 2 executed on util.JsonGeneratorDelegate class

Is more an empirical assumption, we can do a deeper analysis looking at the statistics of modified lines, but in general the structure of the **util.JsonGeneratorDelegate** is very similar to the one in **util.JsonParserDelegate**, so is more probable that is a false positive. This class extend the **JsonGenerator** class and it is implementing the delegate patter with a modification. In the constructor of the class we have a boolean parameter *delegateCopyMethods*, it is used in **writeObject, writeTree, copyCurrentEvent, copyCurrentStructure** methods to signal if delegate the function call to the delegator or to the super class.

5.3 Bridge Pattern

Pattern4J found one instance of the bridge pattern. The bridge pattern emphasises composition rather than inheritance. Implementation details are moved from one hierarchy to another with separate hierarchies of objects. Looking on the context of the library that is intended to be used as a core for a JSON parser and by looking at the class implemented it make even more sense. The class detected is **json.JsonGeneratorImpl** and the class used is **io.CharacterEscapes**. The **io.CharacterEscapes** class define the escape character used in the file and is particularly relevant to have this abstraction because there are different standards for string escaping on different systems. For example on unix systems the escape sequence is "\n", where on Microsoft operating systems the escape sequence is "\r\n" and on IBM mainframe systems the escape sequence is "\025".

6 Behavioral Patterns

6.1 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.

6.2 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.

6.3 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 5: 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.

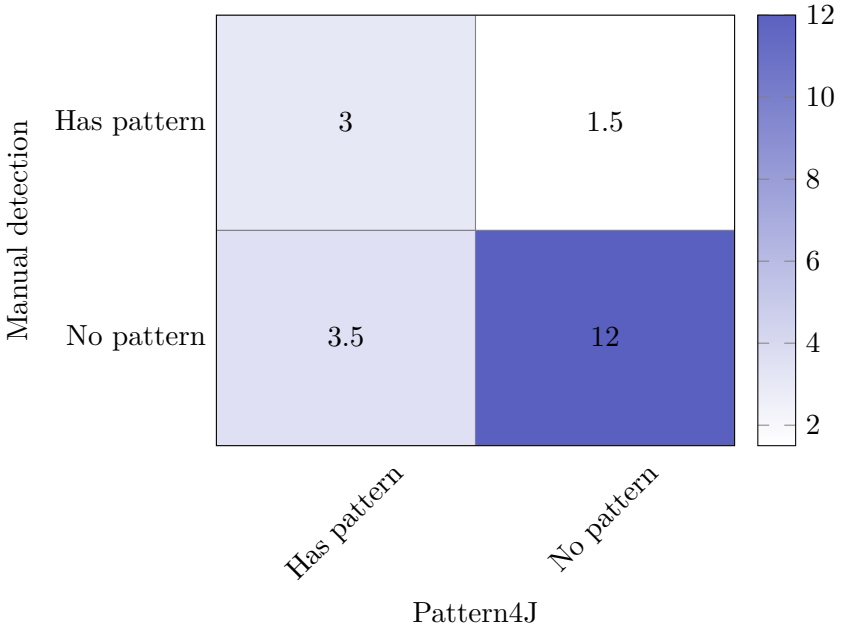
7 TBD Conclusions

TBD

a brief (possibly speculative) discussion about whether your findings are likely to be applicable to other projects or, conversely, they are probably unique to the project you selected – and why you think this to be the case.

Class	True positives	False positives	False negatives	True negatives	Notes
util.DefaultPrettyPrinter$Indenter		1			state and adapter, false positives
ObjectCodec				1	
type.WritableTypeId				1	
util.DefaultPrettyPrinter$NopIndenter	1				variation of singleton, true positive
json.PackageVersion			1		variation of singleton, false negative
io.UTF32Reader				1	
io.NumberInput				1	
json.JsonReadFeature				1	
io.SerializedString	1				variation of singleton, true positive
type.TypeReference				1	
JsonPointer		1			singleton, false positive
json.UTF8DataInputJsonParser				1	
format.InputAccessor$Std				1	
JsonStreamContext	1				template method (barely), true positive
filter.TokenFilter		0.5	0.5		singleton, false positive; strategy, false negative
util.VersionUtil				1	
json.UTF8JsonGenerator				1	
JsonParser$Feature				1	
exc.StreamReadException				1	
util.JsonGeneratorDelegate		1			decorator, false positive (a "delegation" instead, pattern not supported by *Pattern4J*)
Total	3	3.5	1.5	12	
Percentage	15.0%	17.5%	7.5%	60.0%	

(a) Table of classes analyzed manually against the *Pattern4J* tool results. Units are classes, the value 0.5 represents a pattern inside a class with two patterns.



(b) Confusion matrix comparing the *Pattern4J* detection results with our manual interpretation of the classes. Units are classes, the value 0.5 represents a pattern inside a class with two patterns.

Figure 2: Results of the statistical analysis on the effectiveness of the *Pattern4J* tool as a class-by-class table (Figure 2a) and as a confusion matrix (Figure 2b).