Finding and Documenting Design Patterns

Software Design and Modelling, Università della Svizzera Italiana

Luca Di Bello

Saturday 19th October, 2024

1 Introduction

In this assingment we are going to use the pattern detection tool pattern4j to automatically detect the usage of design patterns in a chosen open-source Java project and document the result in a report. The tool is able to scan Java bytecode and detect the usage of 13 design patterns.

The assignment requires to choose a Java open-source project available on GitHub, which satisfies the following requirements:

- At least 100 starts
- At least 100 forks
- At least 10 open issues
- At least 50'000 lines of Java code (comments included)

In order to find a valid project, the GitHub search feature allows to filter repositories based on different criteria. For example, to find a Java project that satisfies the requirements listed above, the following query can be used:

```
stars:>100 forks:>100 language:java
```

On the other hand, in order to count the lines of code of a project without having to clone it locally, it is possible to use the following web application: codetabs.com/count-loc.

1.1 Project selection

In order to learn more about the design patterns used in famous Java projects possibly used in the industry, I looked for active projects, with a large community and a good number of stars and forks. In order to find projects

In order to learn more about design patterns, I decided to look for a project that is activelly used in the industry, with a large community and a good number of stars and forks. After some research I first selected a small set of projects that satisfied the requirements:

- projectlombok/lombok: Library containing a set of useful Java annotations to reduce boilerplate code in Java applications. It has 30.7k stars, 3.7k forks, and used actively in many famous projects. It has around 98k lines of Java code (comments excluded).

  This project unfortunately was discarded as it was using the `Ant` build system, which I am not familiar with. To avoid potential issues and to be able to focus on the main task of the assignment, I decided to look for other projects using `Maven` or `Gradle`.

- ReactiveX/RxJava: Library for composing asynchronous and event-based programs using observable sequences. It provides many bindings for different languages and platforms. It has 19.9k stars, 2.9k forks, and has about 395k lines of Java code (comments excluded).

  Initially, I started this assignment with this project in mind but, unfortunately, I encountered major problems during the compilation of the project which made me discard it. The project was using the `Gradle` build system but the build process terminated with an unexpected error that I was not able to debug.

- benmanes/caffeine: A high-performance caching library for Java used in many famous projects such as *Apache Kafka*, *Cassandra*, *Neo4J* and many others. It has 15.8k starts, 1.6k forks and has about 100k lines of Java code. This project unfortunately, representing a mature and well-maintained Java library (as on the Saturday 12th October, 2024) only had 4 open issues on GitHub.

  This project completely aligns with the project requirements and my personal objectives for this assignment. For these reasons, I decided to use this project for the assignment. Additionally, the project uses the `Gradle` build system, which I am more familiar with. I was able to build the project without any complication, refer to subsection 1.3 for more details.

1.2 High-level overview of the project structure

This project, apart from offering an high-performance caching library, offers two adapters for the Google Guava and JCache caching libraries. The project is divided into the following main packages:

- `com.github.benmanes.caffeine.cache`: Main package of the library, contains the core classes of the caching library.
- `com.github.benmanes.caffeine.cache.guava`: Contains classes that provide compatibility with the Google Guava caching library. [1]
- `com.github.benmanes.caffeine.cache.jcache`: Adapter for the *JSR-107 JCache* caching API. [2]

In this assignment, I will consider only `com.github.benmanes.caffeine.cache` package for the design pattern analysis along with the test code.

1.3 Building and analyzing resulting bytecode

The project can be easily built using the provided Gradle Wrapper script. This script will download the required version of *Gradle*, configure the project, start the build process. To do so, the following command has been used:

```
cd caffeine && ./gradlew build -x test -x javadoc
```

Listing 1: Building the Caffeine project with Gradle

The additional flags `-x test` and `-x javadoc` are used to skip the execution of the tests and the generation of the documentation, respectively. The two steps are quite time-consuming and would not add any value to the report. For these reasons, the two steps have been disabled.

In this report we are going to analyze the design patterns used in the core library of the project, avoiding the analysis of the Google Guava and JCache adapters. The compiled classes are located in the directory `caffeine/build/classes/java/main`.

In order to start the design pattern analysis tool `pattern4j` in headless mode, the following command can be used:

```
java -Xms32m -Xmx512m -jar ./tools/pattern4.jar \
  -target "./caffeine/build/classes/java/main" \
  -output "./results/out.xml"
```

Listing 2: Bash command to start the pattern4j design pattern analysis in headless mode

The tool will analyze the `class` files present in the target directory and return an XML file containing the detected design patterns and their locations in the code. It is also possible to start the GUI version of the tool by simply removing the `target` and `output` flags.

To streamline both building and analysis processes, I created a Makefile to build the project and start the design pattern analysis tool. The Makefile contains the following targets: `build`, `clean`, `analyze`, `analyze-gui`.

2 Analysis of results

After running the design pattern analysis tool on the entire `com.github.benmanes.caffeine.cache` package, the tool was able to find the usage of seven design patterns. In this report we will analyze the most idiomatic usages of the design patterns found in the *Caffeine* library. Each section will present a brief description of the design pattern and its usage in the library.

2.1 Creational Design Patterns

2.1.1 Factory Method

The tool detected the usage of the *Factory Method* design pattern in 5 different instances. This pattern is used to encapsulate the creation of objects, allowing subclasses to alter the type of objects that will be created. In the following paragraph, each instance will be briefly presented and analyzed. Due to the large amount of instances found, only 3 out of the 5 instances will be presented as they are the most representative of the pattern usage or showcase interesting implementations of the pattern.

NodeFactory interface This interface represents a particular implementation of the *Factory Method* pattern. The interface defines a two static constructor methods `newFactory` to create new `NodeFactory` instances. The method signatures are defined in Listing 3.

```
static NodeFactory<Object, Object> newFactory(String className)
static <K, V> NodeFactory<K, V> newFactory(Caffeine<K, V> builder, boolean
    isAsync)
```

Listing 3: NodeFactory interface static constructor methods

These two static constructor methods leverage the `Singleton` design pattern in order to provide a single instance of the `NodeFactory` per className: if the `newFactory` method is called with the same `className` parameter (either explictly passed as parameter or embedded in the `builder` object), the same instance of the `NodeFactory` will be returned. To keep track of different instances, both methods access internally a static `ConcurrentMap`, which pairs the provided `className` with the `NodeFactory` instance: if the `className` is already present in the map, the method will simply return the instance, otherwise it will create a new instance, add it to the map and return it. The code responsible for this behavior is presented in Listing 4.

It is important to highlight that the design pattern scanner, probably due to the complexity of the implementation, did not detect the this particular usage of the *Singleton* design pattern.

```
var factory = FACTORIES.get(className);
if (factory == null) {
  factory = FACTORIES.computeIfAbsent(
    className, NodeFactory::newFactory
  );
}
return (NodeFactory<K, V>) factory;
```

Listing 4: NodeFactory singleton design pattern implementation using a static `ConcurrentMap` instance

The `NodeFactory` interface two abstract factory methods to create new cache nodes (`Node` objects) of different types. The method signatures are defined in Listing 5.

```
Node<K, V> newNode(K key, ReferenceQueue<K> ref, V value,
  ReferenceQueue<V> valueReferenceQueue, int weight, long now);
Node<K, V> newNode(Object keyReference, V value,
  ReferenceQueue<V> valueReferenceQueue, int weight, long now);
```

Listing 5: NodeFactory abstract factory methods to create `Node` objects

Since both methods are `abstract`, each concrete implementation of the `NodeFactory` interface must define its own implementation of these methods, allowing to return different instances of the `Node` interface. This is a clear example of the *Factory Method* pattern usage.

As example, the `BoundedLocalCache` cache internally use a `NodeFactory` factory to create new cache nodes. This allows to easily switch between different implementations of the `Node` interface, depending on the cache type.

LinkedDeque interface This interface specify a linked list implementation of a deque. The interface apart from defining the methods to manage the deque, it also defines two abstract factory methods that return new instances of the *PeekingIterator* interface. The methods are defined as follows:

```
PeekingIterator<E> iterator();
PeekingIterator<E> descendingIterator();
```

Concrete implementations will need to implement these methods and return new custom iterators: since the *PeekingIterator* is an interface, the *Factory Method* pattern is used to encapsulate multiple possible implementations of the iterator.

Other instances The tool detected the usage of the *Factory Method* pattern in the following three other locations: *AsyncLocalCache::synchronous*, *AsyncLocalCache::policy* and *StripedBuffer::create(E)*. These are all valid implementations of the design pattern, but are not particularly interesting as they are similar to the ones presented above.

2.1.2 Singleton

The *Singleton* pattern is used to ensure that a particular class has only one instance. To do so, the class usually defines a private constructor in order to hide the constructor from other classes, and a static method to serve as an entry point to the instance: the method will create the instance if it does not exist, or return the existing instance if it does. The tool detected the usage of the *Singleton* pattern in the *Caffeine* library in 9 different instances. Since all instances are essentially the same, I will present the *Weighter* interface as an example of the pattern usage.

In this particular case, in order to prevent the creation of multiple `Weigher<K, V>` instances, the *Caffeine* library implements the *Singleton* pattern by leveraging a common trick involving *enum* types. The *SingletonWeighter* enum implements the *Weigher* interface, and defines a single instance of the

enum named `INSTANCE`. Thanks to the nature of *enum* types, the `INSTANCE` instance is guaranteed to be unique. The *Weigher* interface provides a static method to return the singleton instance of the `SingletonWeigher` enum. The method signature is defined as follows:

```
enum SingletonWeigher implements Weigher<Object, Object> {
  INSTANCE;
  ...
}

static <K, V> Weigher<K, V> singletonWeigher() {
  @SuppressWarnings("unchecked")
  var instance = (Weigher<K, V>) SingletonWeigher.INSTANCE;
  return instance;
}
```

This kind of *Singleton* pattern implementation is found in all 9 instances detected by the tool. The other instances are not presented in this report as they would not add any value to the analysis.

I decided to present the *Weighter* interface as an example of the *Singleton* pattern usage, as it also exhibit some characteristics of the *Factory Method* pattern, offering custom methods to encapsulate the creation of different kinds of *Weighter* objects. The method signatures are defined as follows:

```
static <K, V> Weigher<K, V> singletonWeigher() { ... }
static <K, V> Weigher<K, V> boundedWeigher(Weigher<K, V> w) { }
```

However, in order to be a correct factory method implementation, subclasses should be able to implement the *Weigher* interface and define their own custom implementations of the factory methods presented above, allowing to return different instances of the *Weigher* interface. This is not possible as both methods are *static*, thus they cannot be overridden by subclasses.

2.2 Structural Design Patterns

2.2.1 Adapter

This structural design pattern allows to *adapt* the interface of a particular class into the interface expected by the client. The tool detected the usage of the *Adapter* design pattern in

TO FINISH BROOOO

2.2.2 Decorator

This structural design pattern allows to add dynamically new functionalities to an object, without altering its definition. The tool detected the usage of the *Decorator* design pattern in 3 different instances: `GuardedScheduler` (decorator of `Scheduler`), `BoundedWeigher` (decorator of `Weigher`) and `GuardedStatsCounter` (decorator of `StatsCounter`). Since this design pattern is implemented in a similar way in all found instances, I will only present the `Weigher-BoundedWeigher` pair as an example as it is the most idiomatic across the codebase.

As we have seen in the previous section (refer to subsection 2.1.2), the `Weigher` interface offers two static methods for the creation of `Weigher` objects: `singletonWeigher` and `boundedWeigher`. The `BoundedWeigher` class is a concrete implementation of the `Weigher` interface that decorates another `Weigher` object by adding an additional method `writeReplace()` to the object. The `writeReplace()` method is used to replace the decorated object with the `BoundedWeigher` object when the object is serialized. The `BoundedWeigher` class is defined as follows:

```
final class BoundedWeigher<K, V> implements Weigher<K, V>, Serializable {
  private static final long serialVersionUID = 1;
```

```
  @SuppressWarnings("serial")
  final Weigher<? super K, ? super V> delegate;

  BoundedWeigher(Weigher<? super K, ? super V> delegate) {
    this.delegate = requireNonNull(delegate);
  }

  @Override
  public int weigh(K key, V value) {
    int weight = delegate.weigh(key, value);
    requireArgument(weight >= 0);
    return weight;
  }

  Object writeReplace() {
    return delegate;
  }
}
```

By implementing the `Weigher` interface, `BoundedWeigher` must implement the same methods as the *delegate* object passed to the constructor, and by encapsulating the *delegate* object function calls, the `BoundedWeigher` class is able to extend the default behavior of the object while maintaining the same interface. This is the essence of the *Decorator* pattern.

2.2.3 Bridge

2.3 Behavioral Design Patterns

2.3.1 Bridge

2.3.2 Template Method

3 Evaluation

References

[1] *Caffeine Wiki - Guava*. Accessed 11.10.2024. URL: https://github.com/ben-manes/caffeine/wiki/Guava.

[2] Ben Manes. *Caffeine Wiki - JCache*. Accessed 11.10.2024. URL: https://github.com/ben-manes/caffeine/wiki/JCache.