Finding and Documenting Design Patterns

Software Design and Modelling, Università della Svizzera Italiana

Luca Di Bello

Sunday 20th 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 stars, 100 forks, 10 open issues, and 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.

The following table summarizes the findings of the design pattern analysis tool:

- **Factory Method**: Detected in 5 different classes, with a total of 7 factory methods identified. In *Caffeine* is is used to create particular configuration of data structures, such as synchronous or asynchronous caches, or to create a certain type of cache node.
- **Singleton**: Found in 9 different classes, with a total of 9 singleton instances. Due to the versatility of the pattern, it has been used extensively in the library for various reasons. The analyzer unfortunately was not able to detect usage of this pattern in the `NodeFactory` interface (refer to subsubsection 2.1.1 for more details)
- **Adapter**: Detected in 6 different classes, with 6 adapter classes detected. This pattern allows the library to use internally the interface of a passed object to perform certain operations, while exposing a different interface to the client. For instance, `BoundedPolicy` adapts the functionality of a `BoundedLocalCache` object to perform certain operations on the cache, while exposing a `Policy` interface to the client.
- **Decorator**: Identified in 3 different classes. *Caffeine* library use this pattern to extend the functionality certain base classes. For example, the class `GuardedScheduler` decorates a `Scheduler` object to add additional functionality to the scheduler, while maintaining the original interface.
- **State**: This design pattern has been identified in 8 classes. This design pattern allows to encapsulate different behavioral states of an object in separate classes, allowing the object to change its behavior at runtime. Unfortunately, the scanner produces many false positives for this pattern, as it is difficult to distinguish between a state pattern and a simple delegation of functionality to another object. Please refer to subsection 2.3 for more details.
- **Bridge**: Found in 4 classes, this design pattern allows to decouple the object abstraction from its implementation, allowing both to change independently. Unfortunately, the scanner also in this case produced many false-positives, please refer to subsection 2.3 for more details.

- **Template Method**: Detected the use of this design pattern in 7 different classes. In most of the cases the library uses the template method in classes reguarding data structure definitions, in order to separate the logic of the data structure from the general logic, allowing subclasses to redefine certain behaviors. Unfortunately even in this case the scanner made some mistakes, refer to the subsection 2.3 for more details.

Due to the size of the library, and the amount of design patterns detected, the report will only analyze the most interesing usages of the design patterns found. For each design pattern, the report will provide details on how the pattern is implemented and what value it brings to the architecture of the library.

Note: due to the limited amount of pages, it is impossible to analyze in detail every detected design pattern and its usage inside the library. For this reason, certain design patterns, where the usage is not interesting or where the scanner produced multiple false-positives, will not be analyzed in the following sections.

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 this section will be presented only the `NodeFactory` instance as it represents the most complete and interesting example of the pattern usage.

The `NodeFactory` 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 class name: 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.

2.1.2 Singleton

The *Singleton* pattern is used to ensure that a particular class has only one instance. To address this requirement, the class usually defines a private constructor (to hide the constructor method from clients), and a static method to serve as the new entry point to the instance. This method will try to lazily create the instance, if it does not exist yet, and return it.

The tool detected the usage of the *Singleton* pattern in the *Caffeine* library in 9 different instances. Since in all detected instances the pattern is implemented in similar ways, I will present only one instance where the pattern is used in a more interesting way.

Weigher interface and SingletonWeighter enum The *Weigher* interface is used to calculate the weight of a cache entry. In order to prevent the creation of multiple `Weigher<K, V>` instances, the *Caffeine* library implements the *Singleton* pattern by leveraging a very common Java trick involving an *enum* type named `SingletonWeighter`. The *Singleton* enum is defined in Listing 7.

```
enum SingletonWeigher implements Weigher<Object, Object> {
  INSTANCE;
  @Override public int weigh(Object key, Object value) {
    return 1;
  }
}
```

Listing 6: `SingletonWeigher` enum definition

The *SingletonWeighter* enum implements the *Weigher* interface, and defines a single entry named `INSTANCE`, which is of type `SingletonWeigher`. Since the `SingletonWeighter` enum implements the `Weigher` interface, the `INSTANCE` entry is also of type `Weigher<Object, Object>`, effectively making it a singleton instance of the `Weigher` interface. This is only possible due to the intrinsic nature of *enum* types, where each entry is guaranteed to be unique and only instantiated once.

In order to retrieve the singleton instance in a more concise way, the method `singletonWeigher` performs an unchecked cast to the `Weigher<K, V>` type, effectively returning the singleton instance:

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

Listing 7: SingletonWeigher shorthand method to return the singleton instance using an unchecked cast

The report presents 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 that encapsulate the creation of different kinds of *Weighter* objects. The method signatures are defined in Listing 8.

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

Listing 8: Weigher interface factory methods

However, in order to be a correct *Factory Method* design pattern 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 not allowing subclasses to alter the return type of the methods via inheritance as required by the design pattern.

2.2 Structural Design Patterns

2.2.1 Adapter

The tool detected the usage of the *Adapter* design pattern in 6 different instances. To present a more interesting example, I will focus on the `BoundedLocalCache` abstract class. This abstract class represents a bounded cache implements the `LocalCache` interface to offer a consistent API to the client. The `BoundedLocalCache` class is defined as follows:

2.2.2 Decorator

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 (needed for serialization, refer to the Oracle documentation). The class uses internally the *delegate* object to perform the actual work. The listing below showcases the main structure of the `BoundedWeigher` class:

```
final class BoundedWeigher<K, V> implements Weigher<K, V>, Serializable {
  @SuppressWarnings("serial")
  final Weigher<? super K, ? super V> delegate;

  BoundedWeigher(Weigher<? super K, ? super V> delegate) {
    ...
  }
}
```

As `BounderWeigher` implements the `Weigher` interface, it must implement the same methods as the *delegate* object passed to the constructor. 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.

2.3 False Positives: State, Bridge and Template Method design patterns

The scanner produces many false positives for the following *State*, *Bridge* and *Template Method* design patterns.

For example, the scanner detected the usage of the *State* design pattern inside the `BoundedLocalCache`. This class implements a task-based approach to alter the state of specific nodes inside the cache using a *page replacement policy*. To update the page replacement policy, the class defines three types of task: `AddTask`, `UpdateTask` and `RemovalTask`. These task have a common structure, offering a single constructor accepting a `Node<K, V>` object, and implementing the `Runnable` interface. The `run` method of each task defines the behavior of that specific task.

The scanner detected the usage of the *State* design pattern since it detected that each task is modeling a specific state of the node object passed as reference. On the other hand, after further inspection, this structure represents perfectly the *Command* design pattern: the task represents a particular request to alter the state of the node, and the `run` method is the actual command that will be executed on the node. Furthermore, after the creation of a specific task, the program appends it to a global buffer named `writeBuffer`. This buffer will then be used to pull tasks and execute them at specific times (i.e. after a write operation). This represents the essence of the *Command* design pattern.

The *Bridge* design pattern on the other hand was detected in many places across the codebase, where it found

I was expecting the scanner to produce false positives, especially for behavioral design patterns, as it is difficult to distinguish between a simple delegation of functionality to another object and the actual usage of a design pattern. In this library, where the class hierarchy is very complex and well-structured, I was expecting the scanner to produce many false positives. Please refer to the **??** for more details about the false-positives I found

2.4 Expected Patterns

The *Caffeine* library is an important open-source project, that has been around for nearly 10 years (first commit on the Saturday 4$^{\text{th}}$ December, 2004). For this reasons, I was expecting to find a well-structured library with a clear separation of concerns and a good usage of design patterns. From the analysis made above, we can confirm all of the assupmtions. Unfortunately, I was also expecting the use of certain design patterns that were not found by the scanner, but that are commonly used for this type of utility library:

- Observer: in order to notify clients of changes in the cache (i.e. evictions, insertions, etc.), the library should implement the observer pattern.
- Command: the library should use the command pattern to encapsulate requests as objects, allowing to parameterize clients with queues, requests and operations.

Both of the patterns were not detected by the scanner, but after further analysis of false-positives produced by the scanner, was possible to identify the *Command* design pattern in the `BoundedLocalCache` class (refer to subsection 2.3).

On the other hand, by a manual inspection of the codebase (searching for common keywords used commonly in the observer pattern, i.e. *listener*, *observer*) it was not possible to find any usage of the observer pattern in the actual codebase, but rather the use of this design pattern when using some dependencies of the library.

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.