Finding and Documenting Design Patterns

Software Design and Modelling, Università della Svizzera Italiana

Luca Di Bello

Sunday 20th October, 2024

1 Introduction

This assignment asks to use the design 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[1] files and detect the usage of 13 different design patterns.

Furthermore, the chosen open-source project must satisfy 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 was used as it allows to filter repositories based on some predefined filters. 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
```

However, since the search filters available on GitHub cannot filter by LOC[2], the web application available at codetabs.com/count-loc was used in order to lines of code of a project without having to clone it locally.

1.1 Project selection

In order to learn more about the design patterns used in famous Java projects, the search was focused on active projects, with a large community and a good number of stars and forks. After some research, the following projects were initially selected:

- projectlombok/lombok: Library containing a set of useful Java annotations aimed at reducing 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).

  However, this project was later discarded as it use the `Ant` build system, which I am not familiar with. In order to avoid potential issues thus being able to focus on the main task of the assignment, I discarded this project and started looking for others using more familiar build systems such as `Maven` or `Gradle`.

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

  Initially, the assignment was started with this project in mind but, unfortunately, this project was discarded later on due to major issues encountered during the compilation phase. *RxJava* was using the `Gradle` build system, but the build process terminated with an unexpected error that was not possible to solve in a reasonable amount of time.

[1]Set of instructions produced during compilation that can be executed by the *Java Virtual Machine* (JVM)
[2]Acronym for *Lines of Code*

- benmanes/caffeine: A high-performance caching library for Java used in many famous projects such as *Apache Kafka*, *Cassandra* and *Neo4J*. It has 15.8k starts, 1.6k forks and has around 100k LOC. Unfortunately, representing a mature and well-maintained Java library only had 4 open issues on GitHub (as on the Saturday 12th October, 2024), rather than the required 10. After inquiring the teaching assistant, it was decided that this requirement could be overlooked as the low number of open issues could be attributed to the maturity of the project.

  As this project completely aligns with the project requirements and the personal objectives fixed for this assignment, it was decided to use this project for the assignment. Additionally, *Caffeine* uses the *Gradle* build system, which this time allowed to build the project without any complication. Refer to subsection 1.3 for more details about the building process.

1.2 High-level overview of the project structure

The *Caffeine* library, apart from offering an high-performance caching library, offers two adapters for the *Google Guava* and *JCache* libraries, allowing to easily integrate *Caffeine* with other existing projects.

The project is split into several packages, but the main packages of the library are the following:

- `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 (documentation available here).
- `com.github.benmanes.caffeine.cache.jcache`: Adapter for the *JSR-107 JCache* caching API (documentation available here)

This report describes the design patterns found in the core library of the project (`cache` package) leveraging `pattern4j` tool. The report will focus on the core library of the project, avoiding the analysis of the *Google Guava* and *JCache* since they represent only extensions of the core library, providing only compatibility with other libraries.

1.3 Building and analyzing resulting Java 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* library using the provided *Gradle Wrapper* script

The additional flags `-x test` and `-x javadoc` are used to skip the execution of the tests and the generation of the Java documentation. These two steps are quite time-consuming and would not add any value to the report. For these reasons, the two steps have been disabled during the build process. The compiled classes of the core library are located in the directory `caffeine/build/classes/java/main`. These files will then be analyzed by the `pattern4j` tool in order to detect the usage of design patterns.

To start the scanner in headless mode, the following console 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 bytecode produced during compilation 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, the repository contains a `Makefile` that allows to execute 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. The following list summarizes the design patterns found in the library:

- **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. However, the analyzer was not able to detect usage of this pattern in the `NodeFactory` interface (refer to subsubsection 2.1.1 for more details), hinting that is not reliable in certain complex cases.
- **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.

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 return different instances of the same type using polymorphism. The pattern is used in the *Caffeine* library to create particular configurations of data structures, such as synchronous or asynchronous caches, or to create particular types of cache node objects.

For example, the `NodeFactory` interface represents a particular implementation of the *Factory Method* pattern. It defines a two static constructor methods `newFactory` to create new `NodeFactory` instances based on a specified class name. 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 constructors leverage the *Singleton* design pattern in order to provide a single instance of the `NodeFactory` per class name. This behavior is achieved by using a static `ConcurrentMap` to store the instances of the `NodeFactory` class linked to the class name (passed explicitly via parameter, or embedded in the `builder` object). This way, the method can simply return the instance if it is already present in the map, or create a new one if it does not exist yet. 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 usage of the *Singleton* design pattern in the `NodeFactory` interface. This is a clear example of the limitations of the tool, as the pattern is clearly implemented in the codebase.

```
    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

Furthermore, the `NodeFactory` interface also defines 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<K,V>` 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.

The *Weigher* interface is used to calculate the weight of a given cache node. 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, which in this case is named `SingletonWeighter`. The enum definition is available in Listing 7.

```
enum SingletonWeigher implements Weigher<Object, Object> {
  INSTANCE;
  // dummy weigh method
  @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 `Weigher` interface offers a static `singletonWeigher()` utility method that simply reads the `INSTANCE` entry from the `SingletonWeigher` enum and 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

Additonally, this interface exhibit also some characteristics of the *Factory Method* pattern, offering custom methods that encapsulate the creation of different kinds of *Weigher* 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 utility methods to create different kinds of `Weigher<K,V>`

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 them to leverage polymorphism in order 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, this section will focus on the `WeakInterner` final class. This class represents a `BoundedLocalCache<E,V>` object adapted to the `Interner` interface by wrapping it in a `WeakInterner` object that implements the methods specified by the `Interner` interface. The interface defines only one method `intern(E sample)`, which is a method common to all interned classes, that allows to add given an object to the `BoundedLocalCache` instance only if it is not already present in the cache. The Listing 9 showcase the general structure of the class.

```java
final class WeakInterner<E> implements Interner<E> {
  final BoundedLocalCache<E, Boolean> cache;
  ...
  @Override public E intern(E sample) {
    // Append to cache only if not present
  }
}
```

Listing 9: `WeakInterner` adapter structure

In summary, The `WeakInterner` class is a concrete implementation of the `Interner` interface that adapts the `BoundedLocalCache` object to the `Interner` interface in order to provide a more specific behavior.

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, only the `Weigher-BoundedWeigher` pair will be presented as it represents the most idiomatic usage of this design pattern among the found instances.

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 specialized `Weigher` objects: `singletonWeigher` and `boundedWeigher`. The `BoundedWeigher` class is a decorates an existing `Weigher` object by adding a method `writeReplace()`, which is needed for serialization purposes (refer to the Oracle documentation). To do so, the class uses the internally the *delegate* object (`Weigher` instance passed by reference via constructor) to perform the actual work. The listing below showcases the main structure of the `BoundedWeigher` class:

```java
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) {
    ...
  }
}
```

Listing 10: `BoundedWeigher` final class decorating a `Weigher` instance

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

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 the *page replacement policy* for specific nodes. The class defines three kinds of task: `AddTask`, `UpdateTask` and `RemovalTask`. These task have a common structure: they all offer a single constructor method accepting a `Node<K, V>` object (subject on which the new policy will be applied) and they implement the `Runnable` interface. The `run` method (inherited from the `Runnable` interface) defines the behavior of the specific task. By extending the same interface, it is possible to store different kinds of task inside a common collection.

After further inspection, this structure represents the *Command* design pattern, where each type of task represents a particular *command* to alter the state of the cache *page replacement policy.* This is also confirmed by the fact that 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, and was detected as the *State* design pattern probably due to the similar taxonomy.

Also the *Bridge* design pattern was wrongly detected in several places across the codebase, probably due to the extensive use of interfaces and abstract classes that made the design pattern detection task more complex, reducing the accuracy of the results. Furthermore, a similar issue happened with the detection of the *Template Method* design pattern. For example, the method `ExpandOrRetry` of the `StripedBuffer` class was flagged as *Template Method*, even if was clearly a utility method as it was marked as *final.* Since subclasses do not allow to alter the behavior of the method, confirms that the *Template Method* design pattern is not used.

In conclusion, due to the size and complexity of this library, it was expected that *pattern4j* would produce false-positives, particularly for more flexible design patterns that can be easily manipulated to fit the architectural needs without affecting the benefits the pattern brings to the codebase (e.g., behavioral design patterns). Additionally, due to the extensive use of design patterns in order to provide a well-structured codebase, the design pattern detection task was expected to be more difficult. As a rough estimation, around 70% of the detected instances were correctly classified by the tool.

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 4th December, 2004). From the start *Caffeine* was expected to be a well-structured library with clear separation of concerns and a good usage of design patterns; assumptions which were all confirmed during the analysis made throughout the previous sections. Strangely, was expected that `pattern4j` would detect the usage of some additional design patterns, which are commonly used in utility libraries of this kind. The following list outlines the missing design patterns:

- *Observer* design pattern: in order to notify clients of changes in the cache (i.e. evictions, insertions, etc.), the library should implement the observer pattern.

- *Command* design pattern: the library should use the command pattern to encapsulate requests as objects, allowing to parameterize clients with queues, requests and operations.

After further analysis of the false-positives produced by the scanner, was possible to identify the use of the *Command* design pattern inside the `BoundedLocalCache` class (for more details, 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 implementation of the *Observer* design pattern, but rather only the use of certain related methods (i.e., *addListener*) offered by other project dependencies.