# Finding and Documenting Design Patterns

Software Design and Modelling, Università della Svizzera Italiana

Luca Di Bello

Saturday 12th 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 12$^{\text{th}}$ 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 Java bytecode

The project can be easily built using the provided *Gradle Wrapper* script `gradlew`. This script will download the required version of *Gradle*, configure the dependencies and build the project. 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 this report. For these reasons, the two steps have been skipped.

The output bytecode we are going to analyze is located in these two directories:

- `caffeine/build/classes/java/main`: Compiled caching library.
- `caffeine/build/classes/java/main`: Compiled tests of the project.

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 an highly customizable (currently configured with MacOS with Java 21) Makefile with useful goals to build and analyze the project. The Makefile can be found in the root of the repository.

# 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 detected patterns are the following (divided by category).

## 2.1 Creational Design Patterns

### 2.1.1 Factory Method

The tool detected the usage of the *Factory Method* design pattern in the following classes/interfaces. Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

In the following paragraphs, we will describe briefly each of them.

**NodeFactory interface** this class represents an abstract factory implementation (defined in an interface with some default implementations) designed to create factory instances for different types of nodes. The interface defines a static factory method `newFactory` that returns a new instance of a particular factory, which implementation is defined by the parameter *className* of type `String`. The interface also defines another static factory method `newFactory` that returns a new instance of a particular factory, which implementation is defined as a parameter *builder* of type `Caffeine<K, V>` (builder pattern, refer to next subsection for more details).

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

Each factory implementation is responsible for creating a particular type of node. The following are the factory methods that each factory implementation must implement:

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

*Caffeine* defines a *Interned* abstract factory which extends the *NodeFactory* interface. The *Interned* factories are used for creating interned nodes, which are nodes that are unique in the cache (only one instance of a particular node can exist at a time in the cache).

The *NodeFactory* (which via polymorphism can be an *Interned* factory) is used by the *BoundedLocalCache* to create new nodes for the cache.

**LinkedDeque interface** this interface represents a simple linked list implementation of a deque. The interface defines also `iterator` and `descendingIterator` methods that return new instances of `PeekingIterator<E>` for the linked list. They are defined as follows:

```
@Override
PeekingIterator<E> iterator();

@Override
PeekingIterator<E> descendingIterator();
```

These two methods are effectively factory methods as they must be implemented by any specialized *LinkedList* implementations in order to encapsulate the creation of the related *PeekingIterator* instances.

**LocalAsyncCache interface** this interface comprehends two factory methods that return new instances of internal data structures used by the cache. The methods are defined as follows:

```
/** Returns the backing {@link LocalCache} data store. */
LocalCache<K, CompletableFuture<V>> cache();

/** Returns the policy supported by this implementation and its
   configuration. */
Policy<K, V> policy();
```

For the same reasons as the previous example, these methods are considered valid factory methods.

**AsyncLoadingCache interface** this interface defines a factory method `synchronous` that returns a new instance of a `LoadingCache<K, V>`, representing a thread-safe view of the cache. This method represents a factory method as it encapsulates the creation of a new instance of a `LoadingCache`.

**StripedBuffer abstract class** as reported in the JavaDoc, lazily-initialized table of atomically updated buffers. In order to create a new buffer instance, the class defines an abstract factory method defined as follows:

```
protected abstract Buffer<E> create(E e);
```

#### 2.1.2 Singleton

### 2.2 Creational Design Patterns

#### 2.2.1 Adapter

#### 2.2.2 Decorator

#### 2.2.3 Bridge

### 2.3 Creational Design Patterns

#### 2.3.1 State

#### 2.3.2 Template

## 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.