Università della Svizzera Italiana

Documenting Design Patterns

Pablo Landrove Pérez-Gorgoroso

October 24, 2025

1. Selection of the project

Several projects were initially considered for analysis, including:

- Arduino: Hardware and software platform designed for building electronic projects.
- Hadoop: Framework for distributed storage and processing of large datasets.
- Kafka: Distributed event streaming platform used for real-time data pipelines.
- Termux: Terminal emulator and Linux environment app for Android.

However, each of these projects presented practical challenges that led to their exclusion. The Arduino project could not be built, preventing further analysis. Both Hadoop and Kafka use Scala in addition to Java. Furthermore, Pattern4J failed to detect any design patterns in Kafka, making it unsuitable for evaluation. Termux, on the other hand, required Android Studio and numerous additional dependencies to build, which significantly complicated the setup process.

For these reasons, all of these projects were ultimately discarded in favor of selecting one that could be built easily and analyzed entirely.

2. Selected Project

The project we selected to analyze is Google's Guava. This is an open-source Java library developed by Google that provides a set of core utilities to make Java programming easier and more efficient. It includes tools such as collections, new data structures, caching framewors, etc... The reason we chose this project is because it was easy to build without additional dependencies and with Pattern4J we are able to find many different Design Patterns.

Some stats about this project:

- 351863 lines of Java code.
- 11.1k forks.
- 51.2k stars.

- 621 open issues.

3. Project Analysis

We first ran Pattern4J on the project repository and encountered issues related to its size (`OutOfMemoryError`). To address this, we increased the JVM memory allocation to 8 GB. Table 3.1 presents the number of detected instances for each design pattern. It is important to note that these figures may not perfectly reflect the actual number of patterns present in the project, as false positives and false negatives can occur. Given the large number of detections (710), it is difficult to precisely estimate the proportion of false positives. However, the subset of patterns that we manually examined, along with several additional verified examples, appeared to be correctly identified by Pattern4J, suggesting that its classifications are generally reliable.

Pattern	Count
Factory Method	36
Singleton	149
Adapter	68
Decorator	47
State	172
Strategy	4
Bridge	17
Template	205
Proxy	5
Proxy2	2
Chain of Responsibility	5

Table 3.1: Design Patterns reported by Pattern4J.

3.1. Design Pattern Coverage

It is also interesting to analyze the presence of design patterns across the classes in the project. First, we determine the total number of classes by counting all `.java` files containing class definitions. This can be done with the following command run in the repository:

```
find .  -name "*.java" | xargs grep -c "class " | wc -l
```

This yields a total of 27,834 classes.

Next, we can export the output of Pattern4J as an `XML` file and use it to analyze all detected design patterns. From this analysis, we find that 424 classes participate in at least one design pattern. This means that approximately 1.5% of the classes play a role in a design pattern.

3.2. SetBuilder Example

Let us examine one of the reported instances of the Factory Method pattern. In this case, a single class and three methods are involved:

- `SetBuilderImpl`: Creator

- `SetBuilderImpl::add(E)`: Factory method
- `SetBuilderImpl::copy(E)`: Factory method
- `SetBuilderImpl::build(E)`: Factory method

Inspecting the code, we can see that `SetBuilderImpl` is a private abstract class defined in `ImmutableSet.java`. This class functions as a builder for creating instances of `ImmutableSet` and includes several concrete methods, such as `addDedupedElement` and `ensureCapacity`, which provide a consistent way of managing internal state across all subclasses.

In addition, `SetBuilderImpl` declares three abstract methods `add`, `copy`, and `build` which are implemented by its subclasses. There are three such subclasses: `EmptySetBuilderImpl`, `RegularSetBuilderImpl`, and `JdkBackedSetBuilderImpl`, each providing a specific implementation of the abstract methods defined in `SetBuilderImpl`. We can see the implementation in Code 1.

Overall, this example demonstrates multiple design patterns at work. The `SetBuilderImpl` and its subclasses clearly implement the **Factory Method** pattern, with the abstract class defining the interface for creating objects and the subclasses providing concrete implementations. At the same time, the use of a builder class to assemble instances of `ImmutableSet` reflects aspects of the **Builder** pattern, ensuring that objects are constructed in a controlled and consistent manner across different variants. This combination of two design patterns in a single example is particularly interesting, as it shows a complementary use of established design principles.

Other design patterns that the tool detects in this class are Template and State. We can also see that one of the subclasses, `EmptySetBuilderImpl` is a Singleton. The main reason being that this empty set with no objects will always be equal, so it wouldn't make sense to have multiple different instances. We can see the code in Code 2 the implementation of this class.

3.3. AbstractService example

The `AbstractService` class serves as an base for services implementing the `doStart` and `doStop` methods. It belongs to the concurrency library, which provides support for asynchronously operating components such as RPC servers, web servers, and timers. The methods `doStart` and `doStop` encapsulate the logic required to manage the initialization and termination of these services in a consistent manner. The tool identifies this class as an adapter in multiple cases involving different adaptees, which is reasonable given that it defines a unified interface for controlling a variety of asynchronous services. It is likely that Pattern4J does not identify all the adaptees involved in this Adapter pattern. The tool reports only three, but given the size and purpose of the class, it is reasonable to assume that additional subclasses exist that were not detected.

3.4. Ordering

The `Ordering` class encapsulates a `Comparator` while extending its functionality through a series of utility methods related to ordered collections, such as `min`, `max`, `leastOf`, and `reverse`. This design provides a more expressive and flexible interface for working with ordered data structures, enabling developers to chain and combine comparison logic in a fluent and reusable manner.

Two subclasses `NullsFirstOrdering` and `NullsLastOrdering` extend `Ordering` to introduce specific handling for `null` values. Both classes take another `Ordering` object as a parameter and override the comparison behavior to define how `null` elements should be treated: `NullsFirstOrdering` ensures that `null` values are considered smaller than any non-null element, whereas `NullsLastOrdering` places them at the end of the ordering sequence. This design allows developers to adapt existing comparators seamlessly without rewriting their logic.

According to Pattern4J, these subclasses are identified as **Decorators**, while the main `Ordering` class acts as the **Decoratee**. This classification is consistent with the intent of the Decorator design pattern, which enables the dynamic extension or modification of an object's behavior without altering its structure. In this context, `NullsFirstOrdering` and `NullsLastOrdering` wrap existing `Ordering` instances to provide additional behavior handling `null` values while maintaining the same interface and ensuring composability. We can observe the implementation of `NullsFirstOrdering` in Code 3, which illustrates how the class delegates the comparison logic to the underlying `Ordering` instance while injecting the new behavior that handles `null` elements explicitly.

4. Generalizability of the Findings

Some of the design patterns identified in this study illustrate principles that are broadly applicable to other projects, while others are more closely tied to the specific design philosophy of the analyzed library.

The `Ordering` and its subclasses provide a clear and canonical example of the **Decorator** pattern. Their implementation aligns closely with the pattern's intent: extending behavior dynamically without modifying the underlying class. This approach is common and widely applicable, particularly in libraries or frameworks that aim to provide composable and reusable components. The way `NullsFirstOrdering` and `NullsLastOrdering` wrap existing `Ordering` instances to alter comparison logic could easily serve as a model for other projects that require flexible extensions of similar abstractions.

In contrast, the combination of the **Factory Method** and **Builder** patterns observed in the `SetBuilderImpl` hierarchy represents a more specialized but highly effective design strategy. These two patterns complement each other naturally the Factory Method which methods the Concrete Factories must implement for the creation of the instances, delegating the logic to those Classes, while the Builder manages the consistency constraints regardless of the Concrete Factory. This synergy could be extended to many other contexts where complex object creation processes need to be standardized yet remain adaptable. Consequently, although this implementation is specific to Guava's immutable collection framework, the underlying architectural principle has broader applicability and could inspire similar designs in other software systems that emphasize immutability and controlled construction.

A. Code Listings

Code 1: Implementation of `EmptySetBuilderImpl`

```
1 private abstract static class SetBuilderImpl<E> {
2     // The first `distinct` elements are non-null.
```

```
    // Since we can never access null elements, we don't mark this
        nullable.
    E[] dedupedElements;
    int distinct;
    @SuppressWarnings("unchecked")
    SetBuilderImpl(int expectedCapacity) {
      this.dedupedElements = (E[]) new Object[expectedCapacity];
      this.distinct = 0;
    }
    /** Initializes this SetBuilderImpl with a copy of the deduped
        elements array from toCopy. */
    SetBuilderImpl(SetBuilderImpl<E> toCopy) {
      this.dedupedElements = Arrays.copyOf(toCopy.dedupedElements,
          toCopy.dedupedElements.length);
      this.distinct = toCopy.distinct;
    }
    /**
     * Resizes internal data structures if necessary to store the
         specified number of distinct
     * elements.
     */
    private void ensureCapacity(int minCapacity) {
      if (minCapacity > dedupedElements.length) {
        int newCapacity =
            ImmutableCollection.Builder.expandedCapacity(
                dedupedElements.length, minCapacity);
        dedupedElements = Arrays.copyOf(dedupedElements,
            newCapacity);
      }
    }

    /** Adds e to the insertion-order array of deduplicated
        elements. Calls ensureCapacity. */
    final void addDedupedElement(E e) {
      ensureCapacity(distinct + 1);
      dedupedElements[distinct++] = e;
    }

    abstract SetBuilderImpl<E> add(E e);

    /** Adds all the elements from the specified SetBuilderImpl to
        this SetBuilderImpl. */
    final SetBuilderImpl<E> combine(SetBuilderImpl<E> other) {
      SetBuilderImpl<E> result = this;
      for (int i = 0; i < other.distinct; i++) {
        /*
         * requireNonNull is safe because we ensure that the first
             'distinct' elements have been
         * populated.
         */
        result = result.add(requireNonNull(other.dedupedElements[i
            ]));
      }
      return result;
    }

    abstract SetBuilderImpl<E> copy();
```

```
    /**
     * Call this before build(). Does a final check on the internal
         data structures, e.g. shrinking
     * unnecessarily large structures or detecting previously
         unnoticed hash flooding.
     */
    SetBuilderImpl<E> review() {
      return this;
    }
    abstract ImmutableSet<E> build();
  }
```

Code 2: Implementation of `EmptySetBuilderImpl`

```
private static final class EmptySetBuilderImpl<E> extends
    SetBuilderImpl<E> {
    private static final EmptySetBuilderImpl<Object> INSTANCE = new
        EmptySetBuilderImpl<>();
    static <E> SetBuilderImpl<E> instance() {
        return (SetBuilderImpl<E>) INSTANCE;
    }
    private EmptySetBuilderImpl() {
        super(0);
    }
    @Override
    SetBuilderImpl<E> add(E e) {
        return new RegularSetBuilderImpl<E>(Builder.
            DEFAULT_INITIAL_CAPACITY).add(e);
    }
    @Override
    SetBuilderImpl<E> copy() {
        return this;
    }
    @Override
    ImmutableSet<E> build() {
        return ImmutableSet.of();
    }
}
```

Code 3: Implementation of `NullsFirstOrdering`

```
    final class NullsFirstOrdering<T extends @Nullable Object>
        extends Ordering<@Nullable T>
    implements Serializable {
  final Ordering<? super T> ordering;

  NullsFirstOrdering(Ordering<? super T> ordering) {
    this.ordering = ordering;
  }
  @Override
  public int compare(@Nullable T left, @Nullable T right) {
    if (left == right) {
      return 0;
    }
    if (left == null) {
      return RIGHT_IS_GREATER;
    }
```

```
16      if (right == null) {
17        return LEFT_IS_GREATER;
18      }
19      return ordering.compare(left, right);
20    }
21    @Override
22    public <S extends @Nullable T> Ordering<S> reverse() {
23      // ordering.reverse() might be optimized, so let it do its
            thing
24      return ordering.<T>reverse().<@NonNull S>nullsLast();
25    }
26    @Override
27    public <S extends @Nullable T> Ordering<@Nullable S> nullsFirst()
          {
28      return (Ordering<@Nullable S>) this;
29    }
30    @Override
31    public <S extends @Nullable T> Ordering<@Nullable S> nullsLast()
          {
32      return ordering.<@NonNull S>nullsLast();
33    }
34    @Override
35    public boolean equals(@Nullable Object object) {
36      if (object == this) {
37        return true;
38      }
39      if (object instanceof NullsFirstOrdering) {
40        NullsFirstOrdering<?> that = (NullsFirstOrdering<?>) object;
41        return this.ordering.equals(that.ordering);
42      }
43      return false;
44    }
45
46    @Override
47    public String toString() {
48      return ordering + ".nullsFirst()";
49    }
50  }
```