

SWE 261 Project Part5

Testable Design

A testable design in software development emphasizes the ease with which individual components can be tested in isolation, promoting reliability, maintainability, and overall software quality. (Koskela, 2019) Here are some aspects and goals for creating a testable design:

1. Minimize Complex Private Methods

Aspect: Private methods are not directly testable from outside their class, which is generally acceptable for simple utility functions. However, embedding complex logic within private methods can obscure potential bugs and complicate unit testing.

Goal: Strive for simplicity in private methods to ensure that the bulk of the logic can be tested through public or protected interfaces. For more complex operations, consider using a more accessible method or a different class altogether to facilitate testing.

2. Limit Use of Static Methods

Aspect: Static methods belong to the class rather than any instance, making them suitable for stateless utility functions, such as mathematical operations. However, static methods that produce side effects or involve randomness are challenging to mock or stub in tests, reducing testability.

Goal: Use static methods judiciously, especially when dealing with operations that affect the application's state or require flexibility in testing. Employing instance methods instead can enhance testability by allowing these methods to be overridden or mocked.

3. Simplify Constructors

Aspect: Constructors inherently run when an instance is created, including in subclass instances, which can make bypassing their execution difficult in tests. Embedding substantial logic in constructors can hinder the ability to set up test conditions effectively.

Goal: Design constructors to be lightweight and free of complex logic. Initialize instances in a basic state and delegate more significant setup tasks to separate methods. This approach enhances the ability to control and modify behavior during testing.

4. Use the Singleton Pattern Cautiously

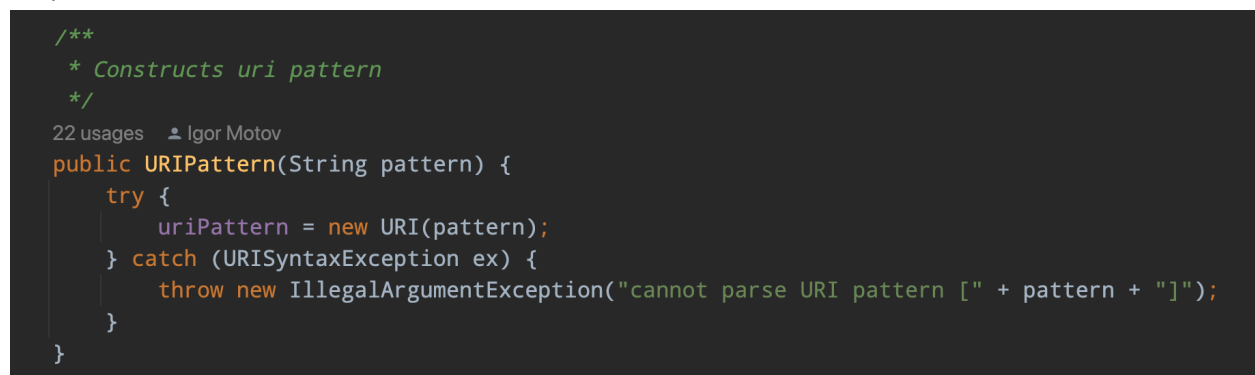
Aspect: The Singleton pattern guarantees a single instance of a class throughout the application lifecycle, which can be appropriate for certain use cases. However,

singletons can be problematic for testing, as they can carry state across tests and are difficult to substitute with mock implementations.

Goal: Evaluate the necessity of the Singleton pattern carefully, considering its implications on testability. If a singleton must be used, ensure it's for components that do not require substitution during testing, such as stateless utility classes. Alternatively, consider dependency injection to manage single instances without enforcing a strict singleton pattern, thereby improving testability.

In our Elasticsearch Project, we found class **URIPattern**, which is located at **org.elasticsearch.common.util.URIPattern.java**

The following screenshot shows the original code of the URIPattern constructor (image 5.1):

A screenshot of a code editor showing the constructor of the URIPattern class. The code is in Java and includes a Javadoc comment, a package declaration, and the constructor logic. The constructor takes a String pattern and attempts to create a URI object. If it fails, it throws an IllegalArgumentException with a message indicating the pattern could not be parsed. The code is color-coded: green for comments, orange for public and catch, and blue for class and exception names.

```
/**
 * Constructs uri pattern
 */
22 usages  Igor Motov
public URIPattern(String pattern) {
    try {
        uriPattern = new URI(pattern);
    } catch (URISyntaxException ex) {
        throw new IllegalArgumentException("cannot parse URI pattern [" + pattern + "]");
    }
}
```

(image 5.1)

The URIPattern class is designed to match URIs against specified patterns, useful for validating or filtering URIs based on components like schema, authority, path, query, and fragment. It stores the pattern as a URI object, established from a string input upon the class instantiation. The constructor attempts to parse this string into a URI object, throwing an IllegalArgumentException if the string fails to conform to URI syntax due to a URISyntaxException.

The class features a match method that normalizes the input URI for consistency before proceeding with the matching logic. The matchNormalized method, where the actual comparison takes place, handles both opaque and hierarchical URIs. In hierarchical URIs, it compares each component against the pattern, using simple pattern matching to allow for flexible matching criteria. For opaque URIs, the comparison is limited to the scheme, scheme-specific part, and fragment.

Additionally, a utility method enables matching a URI against an array of URIPattern objects, enhancing the class's versatility by allowing validation against multiple patterns. The pattern matching relies on the Regex.simpleMatch method from Elasticsearch, facilitating basic wildcard matching in the comparison process.

And we think some challenges would be prevented with that code, which are:

1. **Constructor's URI Dependency:** The URIPattern class constructor internally constructs a URI object, which closely binds URIPattern to the URI class's implementation. This tight coupling complicates testing the URIPattern in isolation, especially with malformed URIs intended for testing exception handling or specific edge cases.
2. **Exception Handling Within Constructor:** If the URI pattern provided to the constructor is invalid, it triggers an unchecked IllegalArgumentException. This approach is directly tied to the internal creation of URI objects, representing a significant level of coupling with the URIPattern class's functionality.

To enhance the testability of the URIPattern class, we decided to decouple the URI and URIPattern during the construction phase. This is achieved by introducing Dependency Injection for the URI in the URIPattern constructor, allowing us to inject URI instances during tests. This approach grants more control over the URIPattern behavior, making it easier to test various URI-related edge cases. The improved implementation is named **URIPatternV2**, located in **org/elasticsearch/common/util/URIPatternV2.java**. By doing so, we avoid handling exceptions in the constructor, thereby increasing the testability and flexibility of the class.

The following screenshot shows the more testable code of the URIPattern constructor (image 5.2):

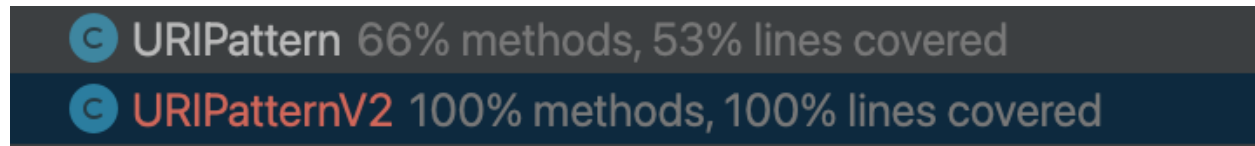
```
/**
 * Constructs a URIPattern with a URI object.
 * This constructor allows for dependency injection of URI, enhancing testability.
 *
 * @param uriPattern The URI object representing the pattern.
 */
5 usages
public URIPatternV2(URI uriPattern) {
    this.uriPattern = uriPattern;
}
```

(image 5.2)

After implementing the newer design of the URIPatternV2, we also designed some test cases to test the feature, located in **server/src/test/java/org/elasticsearch/common/util/URIPatternTestsV2.java**, which ensures the functionality and reliability of the URI pattern-matching logic. These test cases cover various scenarios, including matching URIs that are exactly the same, URIs with different schemes, paths, and even wildcard paths, as well as handling opaque

URIs. By systematically verifying each of these scenarios, we aim to confirm that the URIPatternV2 class correctly identifies matches according to its pattern, thereby validating its effectiveness in a range of standard and edge-case conditions. This comprehensive testing approach helps identify potential issues or limitations in the pattern-matching mechanism. It ensures that the URIPatternV2 class behaves as expected across diverse URI structures and pattern complexities.

After running coverage for both URIPattern and URIPatternV2, we can see the improvement in the coverage (image 5.3):



(image 5.3)

Mocking

Mocking is the practice of "faking" or simulating objects in software development to mimic the behavior of real objects in a controlled environment. (Roell, 2016) The utility of mocking includes:

1. **Isolation for Unit Testing:** Mocking enables the isolation of the unit of code being tested from its external dependencies, such as databases, web services, or third-party APIs. This ensures that the tests are focused solely on the functionality of the unit itself, without interference from external systems.
2. **Enabling Repeatable and Independent Tests:** Mock objects can be configured to return specific responses, allowing tests to be repeatable and produce the same results every time. Moreover, since mocks simulate dependencies, tests become independent of external systems and other tests, adhering to best practices in unit testing.
3. **Facilitating Testing of Edge Cases and Error Conditions:** By using mock objects, developers can easily simulate various scenarios, including edge cases and error conditions, that might be difficult or impractical to reproduce with real objects or systems.
4. **Speed and Efficiency:** Tests that utilize mocks typically run faster than those that interact with real dependencies, especially when those dependencies are slow or complex, such as databases or network services. This efficiency can lead to more frequent test execution and faster development cycles.
5. **Separation of Concerns (SoC):** Mocking supports the principle of SoC by allowing developers to separate the testing of business logic from its interactions with external systems. This separation ensures that changes in one area (e.g.,

the user interface) do not require changes in the business logic tests and vice versa.

6. Simplification of Test Environment Setup: Using mocks reduces the need for configuring complex test environments with all the external dependencies in place, which can be time-consuming and error-prone.
7. Reducing Brittleness: Tests that rely on real dependencies can become brittle, meaning they break easily whenever there are changes in those dependencies. Mocks help reduce this brittleness by simulating the dependencies so changes in external systems do not directly impact the tests.

The features that could be mocked and would be good to be tested with mocking are the `StreamInput` and `StreamOutput` (see image 5.4 for description) features from `server/src/main/java/org/elasticsearch/common/io/stream/StreamInput.java` and `server/src/main/java/org/elasticsearch/common/io/stream/StreamOutput.java`.

The `StreamInput` feature in Elasticsearch is a foundational component for deserialization, allowing for efficient and flexible data reading from a stream, typically during node-to-node communication. It extends the `InputStream` and includes methods for reading various data types, including primitive types, arrays, collections, and complex data structures. Key features include handling different versions of data formats through `TransportVersion`, optimizing methods for readability and maintainability, and ensuring compatibility and robust error handling across distributed systems. This class is crucial for internal communication within Elasticsearch, facilitating reliable and consistent data transfer.

Furthermore, the `StreamOutput` class in Elasticsearch serves as a core component for serialization, enabling efficient and flexible data writing to a stream commonly used in node-to-node communication. It extends `OutputStream` and provides methods to write various data types, including primitive types, arrays, collections, and complex data structures. It incorporates versioning through `TransportVersion` to manage different data formats, ensuring compatibility across different versions of Elasticsearch. `StreamOutput` is designed for readability and ease of use, with methods that closely mirror those in `StreamInput` for symmetry in serialization and deserialization processes. This class plays a vital role in the internal workings of Elasticsearch, ensuring data is accurately and efficiently serialized for network transmission.

We wrote test case via Mockito in order to test `writeTo(StreamOutput out)` feature in `server/src/main/java/org/elasticsearch/http/HttpStats.java`. And we added our test cases, `testWriteTo`, in `server/src/test/java/org/elasticsearch/http/HttpStatsTests.java`

This method is responsible for serializing `HttpStats` data, including server open connections, total open connections, client stats, and potentially HTTP route stats, depending on the transport version. Here is the screenshot of our test code (image 5.5)

```
@Rule
```

```

public MockitoRule mockitoRule = MockitoJUnit.rule();

@Mock
private StreamOutput streamOutputMock;

@Test
public void testWriteTo() throws IOException {
    StreamInput streamInputMock = mock(StreamInput.class);

    when(streamInputMock.readVLong()).thenReturn(1L);

    when(streamInputMock.readCollectionAsList(any())).thenReturn(emptyList());

    when(streamInputMock.getTransportVersion()).thenReturn(TransportVersion.current());

    HttpStats httpStats = new HttpStats(streamInputMock);

    verify(streamInputMock, times(2)).readVLong();
    verify(streamInputMock).readCollectionAsList(any());
    verify(streamInputMock).getTransportVersion();

    assertEquals(1L, httpStats.getTotalOpen());
    assertEquals(1L, httpStats.getServerOpen());
    assertEquals(emptyList(), httpStats.getClientStats());

    when(streamOutputMock.getTransportVersion()).thenReturn(TransportVersion.current());

    httpStats.writeTo(streamOutputMock);
    verify(streamOutputMock, times(2)).writeVLong(1L);
    verify(streamOutputMock).writeCollection(emptyList());
}

```

(image 5.5)

In our test setup for the `HttpStats` class, we initially faced the challenge of dealing with `StreamInput` and `StreamOutput`, which are integral for the deserialization and serialization processes, respectively. To instantiate `HttpStats`, we rely on a `StreamInput` mock. We meticulously define the behavior of `readVLong`, `readCollectionAsList`, and `getTransportVersion` methods on this mock to simulate the incoming data stream that `HttpStats` would typically handle.

Similarly, for testing the `writeTo` method of `HttpStats`, which is responsible for serializing its state to a `StreamOutput`, we create a `StreamOutput` mock. We mock the behavior of `getTransportVersion` to ensure consistency in the output data's versioning. This approach allows us to control the serialization process during our tests precisely.

We employ verification statements, such as `verify`, to confirm that `HttpStats` interacts with these mocks as expected. This includes checking that `HttpStats` reads values correctly from the `StreamInput` and writes the expected values to the `StreamOutput`. Through these verifications, we ensure that the serialization and deserialization logic within `HttpStats` functions correctly, independent of the actual data streams.

We noticed that within the `HTTP` folder, none of the methods utilizing `StreamInput` or `StreamOutput` were previously tested. By introducing mocks for these components, we have made these methods testable, isolating the `HttpStats` logic from the underlying I/O mechanisms. This isolation is crucial because it allows us to focus solely on the behavior of the `HttpStats` class in our tests.

Mocking the input and output streams has several advantages for our testing strategy. It enables us to control the input data meticulously and simulate various scenarios without relying on external systems or resources. This control leads to tests that are faster, more predictable, and more focused. Any failures in our tests can be directly attributed to the behavior of the `HttpStats` class rather than external factors, ensuring that our tests are both reliable and relevant.

Reference

1. Roell, J. (2018, March 8). *Mocks: What are they? when should you use them?*. LinkedIn.
<https://www.linkedin.com/pulse/mocks-what-when-should-you-use-them-jason-roell/>
2. Koskela, L. (2019, September 16). *Chapter 7. testable design · effective unit testing.* · Effective Unit Testing.
<https://livebook.manning.com/book/effective-unit-testing/chapter-7/>
3. <https://github.com/jiabaow/elasticsearch>