A Java Testing Framework Without Reflection

Lorenzo Bettini[a]
Dipartimento Statistica, Informatica, Applicazioni. Università degli studi di Firenze, 50134 Firenze, Italy

Keywords: Java, Reflection, Testing Framework.

Abstract: Java reflection allows a program to inspect the structure of objects at run-time and provides a powerful mechanism to achieve many interesting dynamic features in several Java frameworks. However, reflection breaks the static type safety properties of Java programs and introduces a run-time overhead; thus, it might be better to avoid reflection when possible. In this paper, we present a novel Java testing framework where reflection is never used: we implement the framework only with the Object-Oriented and functional programming mechanisms provided by Java. We will show that implementing and using such a framework is easy, and we avoid the run-time overhead of reflection. Our framework can be used with existing testing libraries and is meant to be extendable.

1 INTRODUCTION

Despite the soundness of its type system (Igarashi et al., 2001), Java allows for unsafe mechanisms such as reflection and dynamic class loading, which are known to break the static type system properties (Bodden et al., 2011; Smaragdakis et al., 2015; Landman et al., 2017; Braux and Noyé, 2000; Li et al., 2019; Livshits et al., 2005; Sobernig and Zdun, 2010; Park and Lee, 2001).

Java reflection allows a Java program to inspect the structure of its objects at run-time and change field values, and call methods through reflection, even if such members are private. When performing such reflective operations, we lose the static type checking since we manipulate objects and members by their names, relying on strings, which cannot be typed beforehand. Moreover, reflection also inevitably introduces run-time overhead.

In many situations, such dynamic mechanisms allow Java developers to recover most of the dynamic flexibility of dynamically typed languages. Moreover, there are several contexts where this enables the creation of dynamic frameworks or extensions to the Java language: ORM (Object–Relational Mapping) frameworks, visual editors, and dependency injection frameworks. Such reflection-based frameworks typically rely also on Java annotations with run-time scope.

Despite its usefulness in some contexts, and due to the above-mentioned drawbacks, we want to avoid reflection when the same goals can be achieved with standard and statically safe mechanisms of Java, mainly Object-Oriented features (inheritance, subtyping, and dynamic method lookup) and functional programming features (lambda expressions).

Reflection and runtime annotations are typically used in Java testing frameworks, like JUnit[1] and TestNG.[2] This paper introduces a new Java testing framework, JNRTEST (Java no reflection Test), which does not rely on reflection to provide the typical testing framework's features. JNRTEST is an open-source project,[3] based on Java 17, which can be seen as a prototypical proof of concept implementation to show that Java language mechanisms are powerful enough to implement a testing framework without reflection, annotations, and dynamic class loading. Moreover, our framework never uses `instanceof` and downcasts. To allow for easy extensions, we also avoid static methods.

In particular, with JNRTEST, we want to achieve the following desired properties:

1. Create a Java testing framework without reflection, relying only on statically typed safe linguistic mechanisms of Java.
2. Such a framework should be easy to implement and maintain for its developers, and

[a] https://orcid.org/0000-0002-4481-8096

[1] https://junit.org
[2] https://testng.org
[3] https://github.com/LorenzoBettini/jnrtest

Bettini, L.
A Java Testing Framework Without Reflection.
DOI: 10.5220/0012082800003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 369-376
ISBN: 978-989-758-665-1; ISSN: 2184-2833

3. easy to use for developers already familiar with other Java testing frameworks like JUnit.
4. Keep the implementation of the framework simple (the source code of its main parts can be described in a paper).
5. The framework should be usable with existing assertion libraries, starting from JUnit assertions up to other mainstream testing libraries like AssertJ. Moreover, other testing libraries should be usable with our testing framework. This includes testing libraries or other systems based on reflection: we do not use reflection in our implementation but allow reflection-based testing systems to be usable with JNRTEST.
6. Running tests with our framework should be faster than with JUnit.
7. The framework should be easily extendable, again without using reflection but only standard OO mechanisms.

We describe our framework's features by showing the main classes' implementation. We also present several examples of tests written with JNRTEST.

The paper is organized as follows. Section 2 presents the concepts of "test case" in JNRTEST, and Section 3 shows how to run tests in our framework. Section 4 shows how to write parameterized tests, and Section 5 how to extend our framework by possibly integrating it with existing testing libraries. Section 6 evaluates our framework concerning the above-listed properties. Section 7 presents a few related works, and Section 8 concludes the paper with hints on future work.

2 SPECIFYING TEST CASES

We assume the reader is familiar with JUnit (in particular, the latest version JUnit 5). We will describe the main concepts and features of JNRTEST by comparing them to the corresponding mechanisms of JUnit.

Moreover, JNRTEST does not introduce new assertion libraries: existing assertion libraries can be seamlessly used. For example, for simple tests, we can use JUnit assertions. For more complex tests, we can use Hamcrest and AssertJ (Leotta et al., 2020). In Section 5, we will show how to easily use other testing libraries with JNRTEST.

In JUnit, tests are specified with methods annotated with `@Test`. The method name implicitly represents the description of that test. In JUnit 5, the description can be specified with the additional annotation `@DisplayName("...")`. In JNRTEST, tests are specified as objects of this type:

```
public record JnrTestRunnableSpecification(
  String description, JnrTestRunnable testRunnable) {}
```

where `JnrTestRunnable` is a functional interface with the abstract method:

```
void runTest() throws Exception;
```

Since we must account for tests throwing exceptions, including checked ones, we could not use standard Java functional interfaces (e.g., `Runnable`) because they are meant for code throwing only unchecked exceptions.

Thus, the following JUnit test:

```
@Test @DisplayName("this is a test")
void aTest() {
  // ... test implementation
}
```

corresponds to a JNRTEST represented by this object:

```
new JnrTestRunnableSpecification(
  "this is a test",
  () -> {
    // ... test implementation
  }
);
```

In JNRTEST, `JnrTestRunnableSpecification` represents any possible executable code (typically in the shape of a lambda expression) that takes part in the test execution's lifecycle. Thus, it is also used for code to be executed before each/all and after each/all test methods (in JUnit 5, such lifecycle parts are represented by the annotations `@BeforeEach`, `@BeforeAll`, `@AfterEach` and `@AfterAll`, respectively).

Since test implementations in JNRTEST are Java lambdas, we do not need to follow any convention concerning static or instance methods like in JUnit (where, for example, `@BeforeEach` must be used for instance methods and `@BeforeAll` for static methods). Moreover, by looking at the previous JUnit 5 test, we are forced to use an additional annotation to specify a description containing spaces (or other characters that are not admissible in a method's name). In the presence of the annotation, the method's name is irrelevant, but we are forced to use a name because JUnit relies on a Java method to represent runnable test code.

JNRTEST specifications are stored in a `JnrTestStore`, which collects such specifications in five separate lists: one list for specifications representing tests and the other four lists for the above-mentioned four lifecycle parts. The class provides methods for inserting specifications in the proper collection and retrieving such collections.

The main entry point for specifying tests is the abstract class `JnrTestCase`, which contains a test store.

```
public class FactorialJnrTestCase extends JnrTestCase {
  private Factorial factorial;
  public FactorialJnrTestCase() {
    super("tests for factorial");
  }

  @Override
  protected void specify() {
    beforeAll("create factorial SUT",
      () -> factorial = new Factorial());
    test("case 0",
      () -> assertEquals(1, factorial.compute(0)));
    test("case 1",
      () -> assertEquals(1, factorial.compute(1)));
    test("case 2",
      () -> assertEquals(2, factorial.compute(2)));
    test("case 3",
      () -> assertEquals(6, factorial.compute(3)));
    test("case 4",
      () -> assertEquals(24, factorial.compute(4)));
  }
}
```

Figure 1: An example testing the factorial implementation.

The class provides `getStore`, which is not a simple getter; it ensures that the store is initialized lazily and only once:

```
public JnrTestStore getStore() {
  if (store == null) {
    store = new JnrTestStore();
    specify();
  }
  return store;
}
```

So, the actual specification of tests is carried out by the protected abstract method `specify`, which subclasses must implement. The class `JnrTestCase` provides a few methods to make runnable specifications easier to write and read. An example is shown in Figure 1, where we write the tests for a factorial implementation.

Thus, `beforeAll` is a utility method that corresponds to creating a `JnrTestRunnableSpecification` and storing it in the corresponding collection of the underlying store. The method `test` does the same for a test implementation. Other methods `beforeEach`, `afterAll`, and `afterEach` have the expected semantics.

3 RUNNING TESTS

Instances of `JnrTestCase` are executed through an instance of `JnrTestRunner`. It is a matter of using `JnrTestRunner` fluent API to specify the instances of `JnrTestCase` to run by using the method `testCase` several times and then to call `execute`:

```
public class JnrExamplesTestMain {
  public static void main(String[] args) {
    var recorder = new JnrTestRecorder()
          .withElapsedTime();
    var runner = new JnrTestRunner()
      .testCase(new FactorialJnrTestCase())
      .testCase(new AnotherTestCase())
      .testListener(recorder)
      .testListener(
        new JnrTestStandardReporter().withElapsedTime());
    runner.execute();
    System.out.println("\nResults:\n\n" +
      new JnrTestResultAggregator().aggregate(recorder));
    if (!recorder.isSuccess())
      throw new RuntimeException(
        "There are test failures");
  }
}
```

Figure 2: An example running test cases from a Java `main`.

```
new JnrTestRunner()
  .testCase(new MyTestCase())
  .testCase(new MyOtherTestCase()
  .execute();
```

Of course, a `JnrTestRunner` is meant to be run directly using a standard Java class with a `main` method.

As a design choice, we kept the semantics of `execute` as simple as possible: the method does not return any value or provide feedback about the run tests and their results.

On the other hand, `JnrTestRunner` accepts listeners that are notified about lifecycle events (e.g., when a test case and every single test are started and when they are finished) and about test results: success, (assertion) failure or error (due to an uncaught exception). Implementing such listeners is straightforward, and we already provide a few of them, e.g., for recording test results and reporting the test outcome on the console (possibly by recording test execution time). Figure 2 shows an example running a few test cases with our runner after configuring the runner with a few listeners.

The runner configuration shown in Figure 2 is typical when running tests with JNRTEST. In fact, we provide a dedicated class for such a configured runner, and the developer only has to specify the test cases to be run.

Running our tests from Maven or Gradle is also straightforward. No special plugin is required. Since we are not relying on runtime test discovery, it is enough to write a Java class with a main method like the one shown above and run that Java class with a standard Maven plugin like `exec-maven-plugin`.[4]

[4]In Gradle, the corresponding plugin can be used that allows running a generic Java application.

```
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>exec-maven-plugin</artifactId>
 <executions>
  <execution>
   <id>run-jnr-tests</id>
   <goals>
    <goal>java</goal>
   </goals>
   <phase>test</phase>
  </execution>
 </executions>
 <configuration>
  <classpathScope>test</classpathScope>
  <mainClass>
   examples.JnrExamplesTestMain
  </mainClass>
 </configuration>
</plugin>
```

Figure 3: An example running test cases from Maven.

We show an example in Figure 3: we only need to specify the fully qualified name of the Java main class and the "test" classpath scope. Since the main class shown above throws an exception if there are test failures, the Maven build will correctly fail in case of failed tests. Providing a custom Maven plugin for running JNRTEST tests would not add any benefit and require about the same number of XML code lines.

Having separate test suites in JNRTEST is also straightforward and does not require additional features in the framework itself: it is just a matter of adequately creating several Java main files.

By design, and due to the current implementation of storing executable tests in a list, `JnrTestRunner` executes the tests precisely in the order of their specification in the `JnrTestCase` instances. Similarly, it executes `JnrTestCase` instances according to their insertion in the `JnrTestRunner` instance. On the contrary, in JUnit, test cases and tests are run in an unpredictable, though deterministic, order.[5] JUnit still allows the developer to specify an order by the method name, display names, or by numbering the annotations, but not in the order of the definition. We believe that, although tests, especially unit tests, should not rely on the order in which they are executed, not having a predictable order only worsens things. In that respect, our predefined and simple natural ordering can be seen as an advantage, though testers should not abuse it. Moreover, by redefining `getStore`, the developer can also change the order of the stored tests, e.g., by sorting them alphabetically according to the descriptions or intentionally shuffling them in random order.

[5]https://junit.org/junit5/docs/current/\-user-guide/\#running-tests.

```
protected <T> void testWithParameters(
   String description,
   Supplier<Collection<T>> parameterProvider,
   JnrTestRunnableWithParameters<T> testRunnable) {
 testWithParameters(description, parameterProvider,
   Object::toString,
   testRunnable);
}
protected <T> void testWithParameters(
   String description,
   Supplier<Collection<T>> parameterProvider,
   Function<T, String> descriptionProvider,
   JnrTestRunnableWithParameters<T> testRunnable) {
 var parameters = parameterProvider.get();
 for (T parameter : parameters) {
   test(description + descriptionProvider.apply(parameter),
     () -> testRunnable.runTest(parameter));
 }
}
```

Figure 4: Parameterized tests API.

We refer to the source code on GitHub for the implementation of `JnrTestRunner`, which consists of a few lines of readable code.

4 PARAMETERIZED TESTS

JNRTEST also allows for parameterized tests. We did not have to extend the standard lifecycle of JNRTEST or add any specific mechanism to the store to support parameterized tests. `JnrTestCase` provides parameterized tests with the generic methods shown in Figure 4, which create several test implementations with the appropriate parameters provided by the lambda of type `Supplier<Collection<T>>`.[6]

For example, in Figure 5, we show the parameterized version of the factorial tests of Figure 1, where `pair` is one of our utility methods for creating pairs.

This is similar to a parameterized test in JUnit 5 that relies on annotations (in particular, `@ParameterizedTest`). For example, in JUnit 5, one could use `@CsvSource` (Comma Separated Values), which accepts an array of strings. As strings, the compiler cannot check their correct contents statically. JUnit has to interpret the strings at run-time, parsing them and splitting them according to the comma character (the separator). Then, it must ensure that the resulting split values have the same length as the method parameters. Finally, it must convert the strings to the method's parameter types. Our solution instead is entirely statically type safe; it is straightforward to implement and use, at the cost of using

[6]The functional interface `JnrTestRunnableWithParameters<T>` corresponds to a Java `Consumer<T>` but allows for checked exceptions.

```
...
  @Override
  protected void specify() {
...
    testWithParameters("input,output",
      () -> List.of(
        pair(0, 1),
        pair(1, 1),
        pair(2, 2),
        pair(3, 6),
        pair(4, 24)
      ),
      p -> assertEquals(p.second(),
        factorial.compute(p.first()))
    );
  }
...
```

Figure 5: An example in JNRTEST testing the factorial implementation with parameterized tests.

structures, like pairs, when we have more than one parameter.

The JNRTEST user can provide a custom representation for the current parameters, using the second version of `testWithParameters` (Figure 4) that takes a function responsible for returning the string description, and it is fully statically typed. In JUnit 5, the parameter `name` of the annotation `@ParameterizedTest` provides a custom description according to the parameter values. The argument is again a string, where placeholders for parameters can be used without any static check by the compiler.

5 EXTENSIONS

The developer can hook on a test case lifecycle by using the test case's test store and appropriately add a few runnables before/after each/all tests.

For example, the code in Figure 6 aims at recreating the same functionality of the JUnit 4 `TemporaryFolder` rule. This can be used as shown in Figure 7, where we have also used the assertions of AssertJ.

The mechanism we have just shown allows the developer to hook into the lifecycle of a test case and provide some data that the test case must explicitly retrieve. Another possibility is to implement a general extension for other testing frameworks like Mockito[7] or dependency injection frameworks like Google Guice.[8]

Note that while our framework does not use reflection, it does not prevent the developers from using reflection-based systems when implementing tests

[7]https://site.mockito.org/.

[8]https://github.com/google/guice.

```
public class JnrTestTemporaryFolder {
  private File temporaryFolder;

  public JnrTestTemporaryFolder(JnrTestCase testCase) {
    var before = testCase.getStore()
      .getBeforeEachRunnables();
    var after = testCase.getStore()
      .getAfterEachRunnables();
    before.add(0,
      new JnrTestRunnableSpecification(
        "create temporary folder",
        () ->
          temporaryFolder =
            Files.createTempDirectory
              ("jnrtest-temp-folder").toFile()
      ));
    after.add(
      new JnrTestRunnableSpecification(
        "delete temporary folder",
        this::delete
      ));
  }

  public File getTemporaryFolder() {
    return temporaryFolder;
  }
  private void delete() { ... }
}
```

Figure 6: An example simulating the JUnit `TemporaryFolder` rule.

```
public class ExampleTestCase extends JnrTestCase {
  private JnrTestTemporaryFolder testTemporaryFolder;

  public JnrTestTemporaryFolderExampleTestCase() {
    super("JnrTestTemporaryFolder example");
    this.testTemporaryFolder =
      new JnrTestTemporaryFolder(this);
  }

  @Override
  protected void specify() {
    test("temporary folder can be used",
      () -> {
        File temporaryFolder =
          testTemporaryFolder.getTemporaryFolder();
        var file = File.createTempFile(
          "a-test-file", null, temporaryFolder);
        assertThat(file).isFile().exists();
      }
    );
    ...
```

Figure 7: An example using `JnrTestTemporaryFolder`.

with JNRTEST. For example, even when implementing a `JnrTestCase`, the user can use Mockito. When doing that, the developer might want to annotate fields of the test case with Mockito annotations like `@Mock` and `@InjectMocks`. To avoid manually creating and injecting mocks in JUnit, the de-

```
public abstract class JnrTestCaseExtension {

 public <T extends JnrTestCase>
    T extendAll(T testCase) {
  var store = testCase.getStore();
  extend(testCase,
    store.getBeforeAllRunnables(),
    store.getAfterAllRunnables());
  return testCase;
 }

 public <T extends JnrTestCase>
    T extendEach(T testCase) {
  var store = testCase.getStore();
  extend(testCase,
    store.getBeforeEachRunnables(),
    store.getAfterEachRunnables());
  return testCase;
 }

 protected abstract <T extends JnrTestCase>
  void extend(T testCase,
    List<JnrTestRunnableSpecification> before,
    List<JnrTestRunnableSpecification> after);
}
```

Figure 8: The abstract extension class.

veloper can use the Mockito extension mechanisms, like the `MockitoJUnitRunner` in JUnit 4 or the `MockitoExtension` in JUnit 5.

For these reasons, we provide an abstract extension mechanism, whose code is shown in Figure 8.

That is not based on instance variables or static variables like in JUnit, and the implementation of the extension can be reused for fields meant to be initialized and cleared at the test case level or a single test level. The methods are generic, so the returned extended test case does not lose its static type.

Creating an extension for Mockito is trivial: extending this base class and implementing the single abstract method is enough, as shown in Figure 9. Our implementation is much more straightforward than the JUnit 4 runner or the JUnit 5 extension of Mockito (the reader might want to compare the few lines of Figure 9 with the sources of Mockito that can be found on its GitHub repository).

This extension can be used with any test case relying on Mockito annotations. For example, given the test case of Figure 10,[9] we can use our Mockito extension as follows:

[9]The example is intentionally simple, but it should be enough to demonstrate the use of Mockito and our extension to the reader familiar with Mockito. The Service/Repository cooperation is also a typical example of mocking, and the implementations of the simple service and repositories should be easily guessed from the shown tests.

```
...
import org.mockito.MockitoAnnotations;

public class JnrTestCaseMockitoExtension
 extends JnrTestCaseExtension {

 private AutoCloseable autoCloseable;

 @Override
 protected <T extends JnrTestCase>
  void extend(T testCase,
    List<JnrTestRunnableSpecification> before,
    List<JnrTestRunnableSpecification> after) {
  before.add(0, new JnrTestRunnableSpecification(
    "open mocks",
    () -> autoCloseable =
      MockitoAnnotations.openMocks(testCase)));
  after.add(new JnrTestRunnableSpecification(
    "release mocks",
    () -> autoCloseable.close()));
 }
}
```

Figure 9: The extension for Mockito.

```
public class StringServiceWithMockTestCase
 extends JnrTestCase {

 @Mock
 private StringRepository repository;

 @InjectMocks
 private StringService service;
...
 @Override
 protected void specify() {
  test("when repository is empty", () -> {
   assertThat(service.allToUpperCase())
    .isEmpty();
  });
  test("when repository is not empty", () -> {
   when(repository.findAll())
    .thenReturn(List.of("first", "second"));
   assertThat(service.allToUpperCase())
    .containsExactlyInAnyOrder("FIRST", "SECOND");
  });
 }
}
```

Figure 10: A test case using Mockito annotations.

```
StringServiceWithMockTestCase testCase =
 new JnrTestCaseMockitoExtension()
  .extendEach(
   new StringServiceWithMockTestCase());
```

To use the extension, we did not use any annotation: we just relied on the standard Object-Oriented mechanisms, which do not require additional reflection mechanisms besides the ones imposed by the external frameworks, like Mockito in this example. Moreover, by relying on the standard generics of Java,

we can also impose static-type safety without additional unsafe cast operations.

Creating an extension for Google Guice, allowing a test case to rely on dependency injection, is also trivial. We refer to the source code of our examples on GitHub.

This section's extension mechanisms do not require special treatment in our `JnrTestCase` and `JnrTestRunner` classes.

6 EVALUATION

This section evaluates our testing framework according to the desired properties described in Section 1.

First of all, concerning item 1, we never use reflection or dynamic class loading in our implementation. Note that this does not prevent our users from using existing testing libraries (item 5). As shown in Section 5, JNRTEST seamlessly integrates with testing libraries like Mockito. In particular, in Section 5, we showed that our framework could be easily extended (item 7). For example, we can create a reusable behavior like the temporary folder example (similar to JUnit 4 rules) with only a few lines of code.

Speaking of lines of code, the main source code of JNRTEST currently consists of 19 files and about 425 lines of code. This allowed us to show most of the interesting implementation code in this paper (item 4). JNRTEST has also been easy to implement (item 2): after a few experiments, carried out with Test-Driven Development methodologies (Beck, 2003), we quickly came up with the current design architecture, based on a small class hierarchy, where each class has a single and clear responsibility. In that respect, while Java annotations, as used in JUnit, might be a shortcut for avoiding class hierarchies and method overriding, they require much code to be processed. Annotation processing at run-time is also based on reflection heavily, and, as such, it is error-prone and harder to test. Our code, instead, can be easily tested without much effort.

Concerning testing our testing framework, we need a bootstrap phase: we have used JUnit 5 in that respect. This also confirmed the simplicity of our design and implementation: with about 10 unit tests, we fully covered our implementation. Then, starting from our JUnit tests, we created the JNRTEST versions of such tests: the code of test methods can be simply copied and pasted in a JNRTEST runnable specification (lambda expression). Thus, we also tested JNRTEST with JNRTEST.

This allowed us to have a first confirmation for item 6: running tests JNRTEST is faster than running tests in JUnit. For example, on a modern fast machine,[10] our test suite in JUnit takes about 1 second when run from Maven (using the `maven-surefire-plugin`). In contrast, our test suite in JNRTEST takes less than 0.3 seconds (using the `exec-maven-plugin`, see Section 3). On a slower machine,[11] the overhead of reflection is even more evident: about 3 seconds with JUnit, instead of about 0.6 seconds with JNRTEST. Additional benchmarks (with artificial examples) confirmed these first experimental results concerning performance.

Finally, as shown in Sections 2-4, JNRTEST is easy to use for developers (item 3). Developers familiar with JUnit should not have problems rapidly using JNRTEST. In particular, as already mentioned, a JUnit test case's body of test methods can be copied and pasted into a test specification's lambda expression of a `JnrTestCase`. Our simple test ordering (see Section 3) could be seen as an improvement concerning the JUnit unpredictable test execution order.

7 RELATED WORK

Reflection and annotations are heavily used in several Java frameworks, independently from testing. In some cases, some alternatives to such frameworks without reflection have been proposed. Our testing framework JNRTEST shares with such alternatives the main goals. To the best of our knowledge, JNRTEST is the first Java testing framework implemented without reflection.

For example, the dependency injection framework Google Guice, which we have mentioned in Section 5, is based on reflection. Dagger[12] is an alternative meant to be a "fully static, compile-time dependency injection framework" for Java and it is based on code generation. In JNRTEST, we do not need to generate code as an alternative to reflection: OOP mechanisms and functional programming are enough to implement the main tasks of a testing framework.

The Java web framework Spring[13] heavily relies on reflection. The Java web framework Takes[14] demonstrates that even a web framework can be implemented in Java with only OOP mechanisms. We share with Takes the same goal. Similarly to Takes, in JNRTEST, we do not even have public static methods nor `instanceof` and downcasts.

[10] Intel i7, 16 Gb RAM, NVMe SSD.
[11] PineBook PRO ARM, 4 Gb RAM, eMMC.
[12] https://dagger.dev/.
[13] https://spring.io/.
[14] https://github.com/yegor256/takes.

The shape of our test specifications, i.e., the signature of `test`, might remind us of test specifications in the Javascript testing framework Jest.[15] Indeed, we took some initial inspirations from Jest for specifying tests in JNRTEST. Of course, the rest of the context (programming language, static typing vs. dynamic typing) does not allow for additional comparisons between JNRTEST and Jest.

As a testing framework, JNRTEST shares with other testing frameworks in Java (JUnit and TestNG) and other statically typed and dynamically typed languages (e.g., CUnit for C, Pytest for Python, SUnit for Smalltalk, etc.), the same concepts of "test case", "test runner" and the ability to create a *test fixture*[16] for tests. Thus, JNRTEST should be easily usable for developers who are already familiar with existing testing frameworks, though its main context is the Java language.

8 CONCLUSIONS

In this paper, we presented the prototype implementation of a novel Java testing framework without using reflection. Besides showing proof of concept that we can avoid reflection with its drawbacks, our main goals were to implement something simple to maintain, easy to use, and efficient. Although JNRTEST is in its early development stage, it provides enough features to be usable in practice, as shown in the paper. Moreover, it can be extended and used with all the existing Java testing libraries.

Currently, JNRTEST does not support nested tests (a feature introduced in JUnit 5). We plan to investigate how to implement them in our framework. Another future extension is to handle skipping of tests, for example, based on predicates passed when calling the method `test`.

We plan to implement some form of IDE support. This will allow the developer to have a dedicated view in the IDE, similar to the JUnit view in Eclipse. Running JNRTEST tests from an IDE or Maven is already possible since, as shown in Section 3, it is enough to create a Java file with the `main` method.

We will investigate the implementation of a simple code transpiler that, given a JUnit test case, generates the corresponding `JnrTestCase`. This will allow us to perform further experiments and benchmarks with existing Java open-source projects. Although a more involved investigation of the execution speed of JNRTEST is required, our initial benchmarks are promising and match the known fact that reflection introduces run-time overhead.

[15] https://jestjs.io/.

[16] A predefined, reproducible and well-known state/environment for each test.

ACKNOWLEDGEMENTS

This work was partially supported by the PRIN project "T-LADIES" n. 2020TL3X8X.

REFERENCES

Beck, K. (2003). *Test Driven Development: By Example*. Addison-Wesley.

Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., and Mezini, M. (2011). Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, pages 241–250. ACM.

Braux, M. and Noyé, J. (2000). Towards Partially Evaluating Reflection in Java. In *PEPM*, pages 2–11. ACM.

Igarashi, A., Pierce, B., and Wadler, P. (2001). Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450.

Landman, D., Serebrenik, A., and Vinju, J. J. (2017). Challenges for static analysis of Java reflection: literature review and empirical study. In *ICSE*, pages 507–518. IEEE / ACM.

Leotta, M., Cerioli, M., Olianas, D., and Ricca, F. (2020). Two experiments for evaluating the impact of Hamcrest and AssertJ on assertion development. *Software Quality Journal*, 28(3):1113–1145.

Li, Y., Tan, T., and Xue, J. (2019). Understanding and Analyzing Java Reflection. *ACM TOSEM*, 28(2):7:1–7:50.

Livshits, V. B., Whaley, J., and Lam, M. S. (2005). Reflection Analysis for Java. In *APLAS*, volume 3780 of *LNCS*, pages 139–160. Springer.

Park, J. G. and Lee, A. H. (2001). Removing Reflection from Java Programs Using Partial Evaluation. In *Reflection*, volume 2192 of *LNCS*, pages 274–275. Springer.

Smaragdakis, Y., Balatsouras, G., Kastrinis, G., and Bravenboer, M. (2015). More Sound Static Handling of Java Reflection. In *APLAS*, volume 9458 of *LNCS*, pages 485–503. Springer.

Sobernig, S. and Zdun, U. (2010). Evaluating Java runtime reflection for implementing cross-language method invocations. In *PPPJ*, pages 139–147. ACM.