JOT JOURNAL OF OBJECT TECHNOLOGY
Journal of Object Technology | RESEARCH ARTICLE

Implementing a Java testing framework from scratch without reflection

Lorenzo Bettini*
*Dipartimento di Statistica, Informatica, Applicazioni 'Giuseppe Parenti', Università degli Studi di Firenze, Italy

ABSTRACT Java reflection allows a program to inspect and manipulate program structure at run time and is a powerful mechanism used in many Java frameworks, including testing frameworks such as JUnit. However, reflection shifts key checks from compile time to run time and can introduce runner-level overhead; therefore, it is often desirable to avoid it when comparable functionality can be achieved with ordinary, statically checked mechanisms. In this paper, we present JNRTEST (Java no reflection Test), a Java testing framework built from scratch that avoids reflection, runtime annotations, and dynamic class loading. Instead, JNRTEST relies on object-oriented and functional programming mechanisms provided by modern Java: tests and lifecycle hooks are registered through explicit, typed APIs and executed directly by the runner, while integrations are implemented via an event-driven listener model. Overall, JNRTEST serves as a proof of concept that a practical Java testing framework can be built around a reflection-free core using ordinary, statically checked language mechanisms.
This work extends our previous presentation with a comprehensive description of design principles, implementation details and new capabilities, and an expanded evaluation (implementation-effort indicators, code-quality/adequacy metrics, and performance benchmarks against JUnit 5). JNRTEST supports parameterized tests, parallel execution, test filtering and selection, and explicit extension mechanisms. We also demonstrate integration with widely used libraries (e.g., Mockito, AssertJ, Testcontainers) and Java tools such as Maven.

KEYWORDS Java, Reflection, Testing Framework.

1. Introduction

Software testing is an essential methodology of software development that provides many guarantees about the correctness, reliability, and robustness of applications. Manual testing is cumbersome, error-prone, and requires too much additional effort (Dijkstra 1972; Myers 1979). Instead, automated testing frameworks simplify the testing process by providing mechanisms for defining and executing tests. Java has several testing frameworks, including JUnit and TestNG, the former being the most widely used testing framework in the Java ecosystem (Louridas 2005; Ma'ayan 2018; Rakshith & Manjunath 2020). Such testing frameworks rely heavily on Java reflection and runtime mechanisms to provide their features.

Java reflection allows a Java program to inspect the structure of its objects at runtime. The developer can obtain information on the members of an object's type without knowing such a type in advance. The developer can change field values and call methods through such information, even if such members are private. Java reflection is a long-established and widely used feature of Java, and it is often the right tool when a system genuinely requires runtime adaptation. In many situations, dynamic mechanisms, such as reflection and dynamic class loading, allow Java developers to recover most of the dynamic flexibility of dynamically typed languages. In several contexts such mechanisms enable the creation of dynamic frameworks or extensions to the Java language (see, e.g., (Chiba 2000; Feigenbaum 2004; Forax et al. 2005; Hoffer et al. 2013; Karaorman et al. 1999; Kirby et al. 1998; Morris 2002; Parson 1999; Welch & Stroud 1999, 2001; Winter et al. 2016)). Examples of the use of reflection are ORM (Object–Relational Mapping) frame-

JOT reference format:
Lorenzo Bettini. *Implementing a Java testing framework from scratch without reflection*. Journal of Object Technology. Vol. 25, No. 02, 2026.
Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2026.25.02.a2

works for automatically converting the structure of objects into database schemas and records, visual editors for visually manipulating objects representing components of GUIs or diagrams, and dependency injection frameworks for creating complex object graph structures, allowing the developers to map interfaces to implementations easily. Such reflection-based frameworks typically rely on Java annotations with runtime scope.

In the late nineties and early 2000s, we personally employed Java reflection and dynamic class loading in several projects to implement one of the first Java mobile agents systems (Bettini 1998; Bettini et al. 1998) and code mobility mechanisms (Bettini, De Nicola, & Pugliese 2002; Bettini 2003), including a prototypical implementation for software updates in Java based on mobile agents (Bettini, De Nicola, & Loreti 2002). The availability of reflection and dynamic class loading in Java since its early versions was crucial to implementing such systems (we started working on Java version 1.1), and it probably contributed to the widespread adoption of Java as a programming language at that time. Reflection and dynamic class loading were a distinguishing feature of Java with respect to other mainstream programming languages available at that time.

However, reflective operations (e.g., member lookup and invocation) shift key checks from compile time to runtime, since the compiler can no longer validate method existence, signature compatibility, or access constraints along reflective call paths. Indeed, the Java type system has been proved sound in (Igarashi et al. 2001). However, even a statically typed language like Java allows the programmer to circumvent and break the static type safety in a program, with runtime mechanisms such as unsafe down-casts, dynamic class loading, and reflection, as reported in several papers, such as, for example, (Bodden et al. 2011; Smaragdakis et al. 2015; Landman et al. 2017; Braux & Noyé 2000; Y. Li et al. 2019; Livshits et al. 2005; Sobernig & Zdun 2010). There are also some proposals to avoid reflection or remove it from existing Java programs (Drechsler & Mocenigo 2007; Park & Lee 2001). Moreover, these runtime mechanisms inevitably introduce runtime overhead.

We do not intend to criticize dynamic mechanisms such as reflection and dynamic class loading. Our main goal is to investigate whether it is possible to implement a testing framework in Java without relying on reflection and other runtime unsafe mechanisms and use only statically checked and type-safe mechanisms of the Java language, mainly Object-Oriented features (like inheritance, subtyping, and dynamic method lookup) and functional programming features (like lambda expressions).

In (Bettini 2023), we presented a preliminary version of JnrTest (Java no reflection Test)[1], a new open-source Java testing framework, built from scratch, which does not rely on reflection, annotations, or dynamic class loading, while still providing the typical features of a testing framework. Moreover, our framework never uses `instanceof` and down-casts. To allow for easy extensions, we also avoid static methods.

This paper extends the previous presentation of JnrTest by providing a more detailed description of the framework's design and implementation, including the main classes and their relationships. Moreover, JnrTest has also been extended with new features, such as test selection and filtering, and the ability to run tests in parallel. We also present several examples of tests written with JnrTest, showing how it can be used to write and run tests in a type-safe manner without relying on reflection or annotations, and how it integrates with the Java ecosystem's tools like build tools and code coverage. We present more examples of extending and integrating the framework with existing testing libraries and frameworks. Moreover, we present some additional tools that can be used with JnrTest. Finally, we have a new section on the background of testing frameworks, and a new section on the main design choices we made in implementing JnrTest. The evaluation section has also been extended accordingly.

The design of JnrTest is guided by the following goals:

- **Reflection-free core:** implement discovery, execution, and extension without reflection, runtime annotations, dynamic class loading, or similarly unsafe runtime mechanisms, so that the framework remains statically checkable by the Java compiler.
- **Minimalism:** keep the core API and implementation small and explicit, avoiding complex configuration and reducing accidental complexity for framework maintainers.
- **Interoperability:** remain compatible with the broader Java ecosystem (e.g., assertion, mocking, and other testing libraries), even if those libraries internally rely on reflection.
- **Extensibility:** support reusable behaviors and framework extensions via ordinary object-oriented composition and explicit method calls.
- **Performance:** reduce runner-level overhead by avoiding annotation scanning and reflective invocation.

This paper explores the following research questions:

RQ1 **Feasibility.** Can a practical Java testing framework be built using only ordinary, statically checked Java mechanisms (e.g., inheritance, generics, and lambda expressions), i.e., without relying on reflection or runtime annotations for discovery and execution?

RQ2 **Trade-offs.** What design trade-offs arise when replacing annotation/reflection-based discovery and invocation with explicit APIs and functional specifications?

RQ3 **Engineering implications.** What engineering implications does avoiding reflection have for the framework implementation (e.g., analyzability/maintainability indicators such as concentrated complexity, code coverage, mutation-test adequacy, code quality, and a compact core)?

RQ4 **Performance.** How does a reflection-free execution model affect runner-level overhead and end-to-end execution time compared to JUnit 5?

We address these questions by presenting JnrTest's design and architecture (RQ1–RQ2, Sections 3–7), and by evaluating code quality/adequacy and performance with complementary measurements (RQ3–RQ4, Section 9).

We describe our framework's features by showing the implementation of the main classes. In general, the description of

[1] https://github.com/LorenzoBettini/jnrtest

JNRTEST features is driven by several examples of tests written with JNRTEST. To better highlight our testing framework's features, we also often compare it with JUnit.

It is essential to clarify that our aim is not to criticize JUnit or position JNRTEST as a competitor. JUnit is a mature and widely adopted framework that has shaped Java testing practices for decades. Instead, we present JNRTEST as a proof of concept showing that modern Java features are sufficient to build a full-featured testing framework without relying on reflection or runtime annotations. More specifically, JNRTEST illustrates how a testing framework can keep its mechanics within the statically checked subset of Java while still supporting extensibility and competitive performance. We hope JNRTEST serves as a useful reference design point for future Java testing frameworks.

The paper is organized as follows. Section 2 provides background on testing frameworks, in particular, JUnit, and discusses the role of reflection and annotations. Section 3 presents the main design choices behind JNRTEST. Section 4 introduces the concept of test classes and test cases in JNRTEST. Section 5 describes how tests are executed in our framework. Section 6 explains how to write parameterized tests. Section 7 discusses extensibility and integration with existing testing libraries. Section 8 presents a few additional tools that can be used with JNRTEST. Section 9 evaluates the framework including code quality and performance. Section 10 reviews related work. Section 11 concludes the paper and outlines directions for future work.

2. Background

A testing framework typically consists of several key components that define how tests are structured and executed:

Test Case: A test case aims at representing a single scenario that verifies a specific behavior of the system under test (SUT). Each test case includes input values, execution conditions, and expected outcomes.

Test Fixture: A test fixture is a well-known set of objects that are common to several test cases. A fixture consists also of the preconditions and environment state required for a test case to execute. The creation of the test fixture often involves setting up necessary objects and initializing resources; similarly, after the execution of a test case, the fixture should be cleaned up, e.g., by releasing resources.

Test Runner: A test runner is responsible for executing test cases and reporting the results. It also handles the lifecycle of the test execution process, including the creation and cleanup of the fixture, running tests in isolation, handling dependencies, and generating reports. Moreover, a test runner should run all tests, even if some of them fail or throw exceptions.

Assertions: assertions validate the expected behavior of the system. They compare actual results with expected values and indicate whether a test passes or fails.

Java, as a mainstream programming language, has several testing frameworks, including JUnit and TestNG, each offering various features to facilitate software verification and validation. In this paper, we focus on JUnit, which is the most widely used testing framework in the Java ecosystem (Louridas 2005; Ma'ayan 2018; Rakshith & Manjunath 2020). JUnit was first developed by Kent Beck and Erich Gamma in the late nineties (Beck & Gamma 1998; Gamma & Beck 1999), as part of the growing interest in lightweight, repeatable unit testing for object-oriented programming, especially in Java. The framework was strongly influenced by Design Patterns (Gamma et al. 1995) and the principles of Extreme Programming (XP) (Beck 2000). JUnit quickly became the de facto standard for unit testing in Java, and its design has influenced many other testing frameworks.

In its earliest versions (JUnit 1.x to 3.x), JUnit required test classes to inherit from the base class `TestCase`. Test methods were identified by their method names, specifically those public methods beginning with the prefix `test`, and they had to be `void` and take no arguments:

```
public class MoneyTest extends TestCase {
  public void testMoneyEquals() {
    // ... test implementation
  }
  public void testSimpleAdd() {
    // ... test implementation
  }
}
```

Thus, the concept of "test case" is implemented by each single "test" method in such a derived class. Each method represents a specific scenario to be verified, and the framework uses the naming convention to discover and execute test methods.

Concerning the creation and cleaning of the state common to all the test methods of the same class, the developer overrides the methods `setUp` and `tearDown`, respectively. This approach relied on several design patterns, particularly the *Template Method*. The `TestCase` base class provides a template method, `run`, which ensures that the `setUp` and `tearDown` methods are called at the appropriate times, before and after the execution of each test method. This design ensures that each test is executed in a controlled environment, with proper initialization and cleanup.

To execute a single test case, the developer could create an anonymous class, overriding the method `runTest` to call a specific test case method, e.g.,

```
TestCase test = new MoneyTest("simple add") {
  public void runTest() {
    testSimpleAdd();
  }
};
test.run();
```

The `TestCase` base class' template method `run` makes sure to call the proper `setUp` and `tearDown` methods at the proper time, before and after calling `runTest`. This pattern allows for precise control over the execution of individual test cases, but it can be verbose and repetitive, especially for larger test suites.

To execute several test cases of the same class, this approach had to be repeated by creating a "suite", for example:

```java
TestSuite suite = new TestSuite();
suite.addTest(
  new MoneyTest("money equals") {
    protected void runTest() {
      testMoneyEquals();
    }
  }
);
suite.addTest(
  new MoneyTest("simple add") {
    protected void runTest() {
      testSimpleAdd();
    }
  }
);
```

This way of running tests was called "static" (Beck & Gamma 1998) since it relied entirely on Java statically checked features. It provided compile-time safety, as the compiler checked method names and signatures. However, it was verbose due to the lack of language features such as lambda expressions or annotations, which were introduced in later versions of Java.

For this reason, JUnit also provided the "dynamic" way, which relied on reflection: one had to create an instance of the class providing the name of the test case method to run as a string:

```java
TestSuite suite = new TestSuite();
suite.addTest(new MoneyTest("testMoneyEquals"));
suite.addTest(new MoneyTest("testSimpleAdd"));
```

While being more compact, this solution is not type safe since a mistake in the method's name goes unnoticed until you get an exception at runtime, while executing the tests. This trade-off between conciseness and type safety is a recurring theme in the evolution of Java testing frameworks, due to the linguistic features of Java at that time.

After the introduction of annotations in Java, JUnit 4 also evolved into a new way of specifying test cases: annotations. In particular, the inheritance from a base class was not needed anymore. Test methods are annotated with `@Test` and creation and cleaning of the common state are delegated to methods with the proper annotations: `@Before` (in JUnit 5, `@BeforeEach`) annotates a method executed before each test, `@BeforeClass` (in JUnit 5, `@BeforeAll`) annotates a method executed once before all tests; The annotations starting with `@After` have the corresponding semantics concerning methods to be executed after test executions. Moreover, the use of patterns like template method was not needed anymore, since the proper lifecycle execution is delegated to a separate test runner, which, by relying completely on reflection, retrieves annotated methods and executes them accordingly.

3. Design choices

This section describes our design choices in implementing the JNRTEST testing framework. It highlights the emphasis on compile-time safety by avoiding Java reflection and other unsafe runtime mechanisms, discusses the trade-offs involved in this decision, and provides an overview of the framework's architecture.

3.1. Emphasis on Compile-Time Safety

A central design choice in JNRTEST is avoiding Java reflection and other unsafe runtime mechanisms, e.g., the Java Service Provider Interface (SPI) and dynamic class loading. While such mechanisms can provide flexibility, they also introduce drawbacks. These mechanisms can obscure the structure of the code, hinder static analysis, and delay error detection until runtime.

This way, JNRTEST code base can leverage the Java compiler's type checking to catch errors early in development, thereby avoiding delayed error detection at runtime. Many methods in JNRTEST are "generic", allowing for reusable statically checked code relying on Java's type inference and generics. By avoiding unsafe runtime mechanisms, JNRTEST ensures that the structure and behavior of tests are explicit and visible at compile time: the relationships between test classes, test cases, and the framework are established using Java constructs such as functional programming, inheritance, and method overriding. This approach leverages the Java compiler's type checking to catch errors early in development of tests. For example, if a test class does not implement the required methods correctly, or if a method signature is incorrect, the compiler will report an error immediately. This stands in contrast to frameworks that rely on reflection and annotations, where such errors may only be detected when the tests are executed. In JUnit, forgetting to annotate a method with `@Test` means that the method will not be executed as a test, and this omission may go unnoticed. JNRTEST avoids this pitfall by making the concept of a test explicit in the framework's API.

Importantly, JNRTEST does not propose a new testing methodology. It provides an alternative *mechanism* to specify tests and lifecycle hooks (and to execute them), while the test logic itself—assertions, fixtures, and the overall structure of test cases—follows the same principles as in mainstream frameworks such as JUnit and TestNG. In other words, JNRTEST changes how tests are *declared and discovered*, not what developers should test or how they design test scenarios.

JNRTEST is designed to be simple, minimal and maintainable. The framework avoids unnecessary complexity in both its API and its configuration. This minimalism extends to the framework's dependencies: using JNRTEST does not require adding multiple JAR files or managing a complex dependency graph. In contrast, JUnit 5, while powerful and modular, often requires the user to include several artifacts and manage their compatibility. A missing or incompatible dependency in JUnit 5 can result in confusing runtime errors, whereas JNRTEST's compile-time checks prevent such issues from arising.

3.2. Trade-offs and Rationale

Historically, reflection was attractive in testing frameworks because it enabled low-ceremony test discovery and invocation: a runner can scan test classes, locate test methods by convention or metadata, and execute them without requiring developers to manually register tests. JUnit's original design (Section 2) explicitly emphasizes minimizing developer effort ("no more work than absolutely necessary") and uses reflection to implement a *pluggable selector* that invokes a test method from its name

(i.e., from a string) (Gamma & Beck 1999). This avoids the maintenance burden (and potential omissions) of hand-written suites, which JUnit notes can otherwise be needed as an explicit entry point (Beck & Gamma 1998). In modern JUnit, reflective inspection of annotations supports richer features such as tagging and filtering tests during discovery/execution and dedicated platform-level discovery designed to help IDEs and build tools identify tests reliably.

A further pragmatic reason for the popularity of reflection in Java testing frameworks is that it enables a largely *declarative* programming model with a comparatively small explicit API surface. Already in JUnit 3, the "dynamic" mode used reflection to locate and invoke the test method based on its name, reducing boilerplate at the cost of shifting certain errors to runtime. JUnit 4 pushed this direction further by removing the required superclass and letting the framework discover `@Test` and lifecycle methods via runtime metadata (Beck & Savoia 2006; Tudose 2020). Tests can be "just methods" without boilerplate code and no need to extend specific base classes or implement interfaces.

In this design space, the framework developer can avoid committing to a large inheritance hierarchy or a fully anticipated set of hook interfaces upfront; instead, new behaviors can often be introduced by extending the annotation vocabulary and the reflective interpreter that processes it.[2]

In the small, reflection can indeed simplify certain aspects of framework design—most notably discovery and dispatch, since a runner can locate and execute tests without requiring explicit registration. While reflection can reduce the amount of up-front API surface that a framework must expose to users (e.g., no base classes or explicit registration), it typically shifts complexity into the framework's discovery/execution engine. Compared to ordinary instantiation and method calls, reflective instantiation/invocation removes many compiler checks (e.g., signature compatibility) and introduces additional failure modes that must be handled explicitly at run time (e.g., access checks, argument mismatches, and reflective wrappers around user exceptions) (Oracle 2024). In addition, reflection-heavy designs often require dedicated discovery and diagnostic logic (e.g., classpath scanning, consistent selection semantics, and actionable error reporting when tests are not discovered), further concentrating complexity in framework internals. This relocation has concrete maintenance consequences: reflective calls are a known obstacle to static reasoning and evolution tooling, and even common IDE refactorings are not guaranteed to be behavior preserving in the presence of reflection, motivating specialized techniques for safer refactorings of reflective Java programs (Thies & Bodden 2012). Accordingly, the engineering effort is not eliminated; it is largely moved from the user-facing test API into framework internals (discovery, invocation, error reporting, and the precise semantics of the annotations metadata).

While annotation-based frameworks like JUnit may allow developers to write slightly less code in some cases, this convenience comes at the cost of reduced static guarantees and increased reliance on runtime mechanisms. Our design goal is to use modern Java features, such as functional interfaces and lambda expressions, to allow for concise and expressive code without resorting to reflection or annotations. Users define test classes and lifecycle hooks explicitly using OOP principles, such as inheritance and method overriding. However, Java lambda expressions and functional interfaces provide a way to express test specifications and lifecycle hooks concisely, reducing boilerplate compared to traditional OOP approaches available in earlier Java versions (at the time of JUnit 3), while maintaining static compiler checks.

In the end, a testing framework (Section 2) must provide users a way to define tests and lifecycle hooks and a way to execute them. Whether this is done via reflection and annotations or via explicit APIs and functional programming constructs is a design choice with trade-offs. With the introduction of annotations in Java, in JUnit 4, tests and lifecycle hooks became "just methods" marked with metadata, and the framework could discover and invoke them reflectively. In JnrTest, tests and lifecycle hooks are "just lambda expressions" registered via an explicit API, and the framework can invoke them directly.

In that respect, as also shown in Section 10, the programming language plays a crucial role in shaping the design of testing frameworks. In languages without reflection or annotations, such as C++ or Rust, testing frameworks rely on macros, templates, or explicit registration to define and execute tests. In languages with functional programming features (e.g., JavaScript, Python and Lua), testing frameworks often use first-class functions and closures to define tests and lifecycle hooks concisely. Maybe, if lambda expressions were introduced in Java before annotations, we would have seen a different design for JUnit and other frameworks.

By prioritizing explicitness and compile-time safety, JnrTest aims at providing a solid foundation for testing that is less prone to subtle errors and easier to reason about. The absence of reflection and annotations also increases the chance to keep the implementation of the framework simple (Section 9.3), more amenable to static analysis tools for code quality and reliability (Section 9.4), and reduces runtime overhead (Section 9.5).

3.3. Architecture Overview

Before turning to concrete APIs and examples, Figure 1 provides a conceptual view of the framework. At a high level, user test suites *declare* executable specifications and lifecycle hooks, which are collected into a per-suite store. A runner then *selects* (optionally) and *executes* these specifications in a deterministic order, while emitting structured execution events. Reporting and integrations are implemented as listeners that subscribe to these events, keeping observation concerns separate from execution. Thus, tests are expressed as explicit executable specifications, organized per suite, executed by a runner with optional selection, and observed through an event-driven listener interface for reporting and integration.

4. Specifying tests

In this section, we introduce the main concepts and features of JnrTest, highlighting the differences and similarities with JUnit to help readers familiar with mainstream Java testing

[2] https://docs.junit.org/current/writing-tests/annotations.html

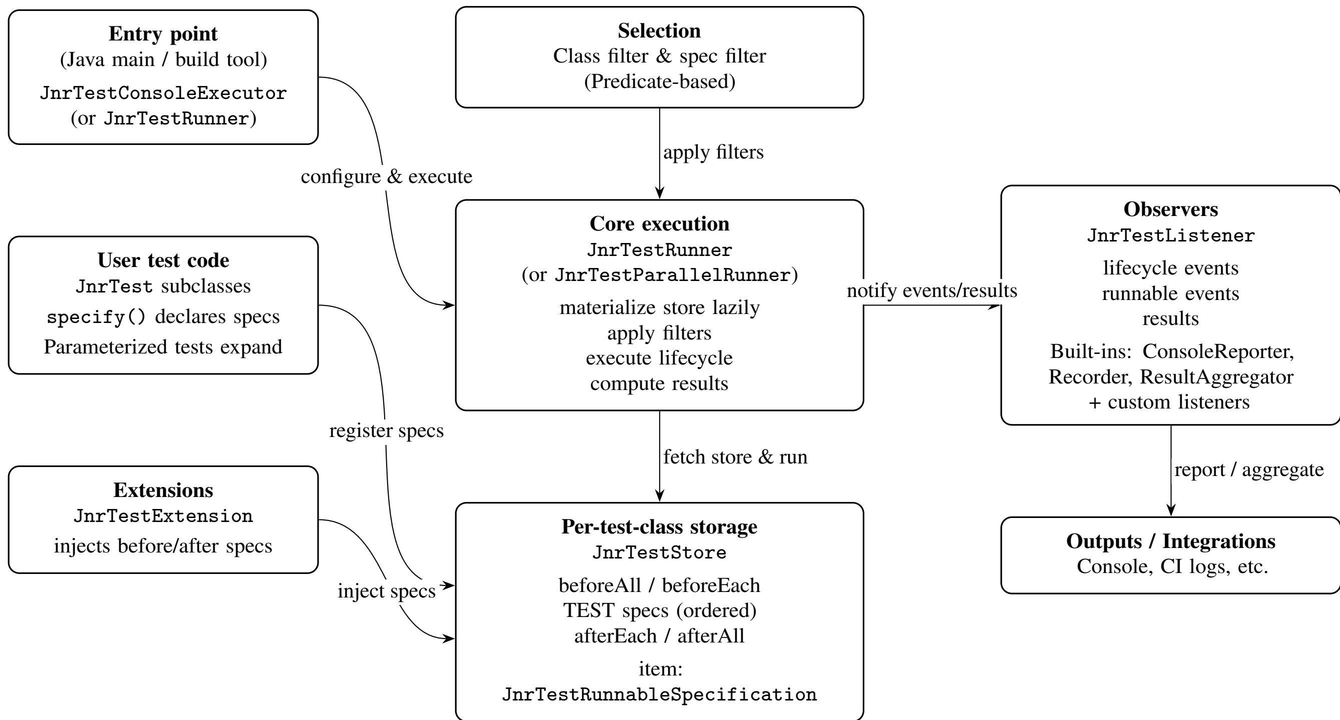

Figure 1 High-level architecture of JnrTest.

frameworks understand JNRTEST's approach. Thus, we assume the reader is familiar with JUnit (in particular, the latest version JUnit 5).

In the following, unless required by the context, we will often refer to "test" as a synonym for "test case".

JNRTEST does not introduce its own assertion library. Instead, it is designed to work seamlessly with existing assertion libraries. For simple tests, one can use JUnit assertions, while for more advanced needs, libraries such as Hamcrest and AssertJ (Leotta et al. 2020) are fully supported. Section 7 will demonstrate how to integrate other testing libraries with JNRTEST, further enhancing its flexibility.

In JUnit 5, tests are defined as methods annotated with `@Test`. The method name serves as the test's identifier, and in JUnit 5, you can provide a more descriptive name using the `@DisplayName("...")` annotation. In contrast, JNRTEST adopts a more object-oriented and functional approach: each test is represented as an object of the following type, which encapsulates both a description and the executable code.

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

where `JnrTestRunnable` is the following functional interface:

```
@FunctionalInterface
public interface JnrTestRunnable {
  void run() throws Exception;
}
```

Since we must account for tests throwing exceptions, including checked ones, we cannot use the functional interfaces provided by Java (e.g., `Runnable`), which only support unchecked exceptions. This design ensures that test code can throw all exception types.

To illustrate the difference with JUnit, consider the following JUnit test method:

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

In JNRTEST, the same test is represented by an object that pairs a human-readable description with a lambda expression containing the test logic:

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

This approach decouples test identification from annotations and method signatures and leverages Java's functional programming features for expressiveness.

In JNRTEST, `JnrTestRunnableSpecification` can represent any executable code involved in the test lifecycle, not just the test itself. For example, it is also used for code that should run before or after each or all tests (analogous to JUnit 5 annotations `@BeforeEach`, `@BeforeAll`, `@AfterEach`, and `@AfterAll`), as shown in the following.

Unlike JUnit, which distinguishes between static and instance methods for lifecycle hooks (e.g., `@BeforeEach` must be used for instance methods, while `@BeforeAll` must be used for static methods), JNRTEST imposes no such restrictions. All test and lifecycle code is specified as lambda expressions, so

there is no need to follow static or instance methods conventions. Furthermore, in JUnit, the method name is often irrelevant when a display name is provided, but still required to define a method. In JNRTEST, the description is always explicit and decoupled from Java method names.

All test and lifecycle specifications are stored in a `JnrTestStore`, which organizes them into five separate lists: one for test specifications and four for the different lifecycle hooks. The store provides methods for adding and retrieving specifications, but it is typically managed internally and not accessed directly by users.

The main entry point for specifying tests is the abstract class `JnrTest`, which contains a test store. The `getStore` method is not a simple getter: it ensures that the store is initialized lazily and only once, by invoking the abstract `specify` method implemented by subclasses.

```
1 public JnrTestStore getStore() {
2   if (store == null) {
3     store = new JnrTestStore();
4     specify();
5   }
6   return store;
7 }
```

This design guarantees that test specifications are set up exactly once, avoiding accidental re-initialization.[3]

To make test definitions more concise and readable, `JnrTest` provides utility methods that hide the underlying details of the store: `test` for specifying a test case, `beforeAll`, `beforeEach`, `afterAll`, and `afterEach` for lifecycle hooks, with the same semantics of the annotations `@Before` and `@After` of JUnit.

An example is shown in Figure 2, 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 case.

The following section will show how to execute and integrate these test specifications into your development workflow.

5. Running tests

Instances of (subclasses of) `JnrTest` are executed by creating a `JnrTestRunner`, which acts as the orchestrator for test execution. In particular, the `JnrTestRunner` is responsible for managing the lifecycle of tests, executing them (including "before" and "after" methods), and notifying listeners about the test execution events (Section 5.1). The runner uses a fluent API: add each `JnrTest` instance to the runner using the `add` method, and then invoke `execute` to run all the specified tests:

```
1 new JnrTestRunner()
2   .add(new MyTest())
3   .add(new MyOtherTest()
4   .execute();
```

Thus, a JNRTEST suite is assembled explicitly by instantiating test classes and registering them with the runner via `add`.

[3] Being a protected method, unless the subclass wrongly calls `specify` several times, we can be sure that the store is initialized only once.

```
1  public class FactorialJnrTest extends JnrTest {
2
3    private Factorial fact;
4
5    public FactorialJnrTest() {
6      super("tests for factorial");
7    }
8
9    @Override
10   protected void specify() {
11     beforeAll("create factorial SUT",
12       () -> fact = new Factorial());
13     test("case 0",
14       () -> assertEquals(1, fact.compute(0)));
15     test("case 1",
16       () -> assertEquals(1, fact.compute(1)));
17     test("case 2",
18       () -> assertEquals(2, fact.compute(2)));
19     test("case 3",
20       () -> assertEquals(6, fact.compute(3)));
21     test("case 4",
22       () -> assertEquals(24, fact.compute(4)));
23   }
24 }
```

Figure 2 An example testing the factorial implementation.

When executing the suite, the runner processes each provided instance: it accesses its store through `getStore()` (which lazily evaluates `specify()` exactly once, Section 4) and then executes all registered specifications on that same instance.

A `JnrTestRunner` is meant to be invoked from a Java class with a `main` method. This makes it easy to integrate with any Java application or build tool (see also Section 5.3). In fact, mainstream IDEs, such as Eclipse, Visual Studio Code and IntelliJ, provide mechanisms (such as context menus, keyboard shortcuts, or run configurations) to run Java classes with a `main` method directly. The IDEs detect the presence of the `main` method and offer options to execute it, making it convenient for developers to run their JNRTEST tests without additional setup.

A key design decision in JNRTEST is that the `execute` method is deliberately kept simple. It does not return a value or provide direct feedback about the test results. Instead, `JnrTestRunner` supports the registration of listeners. Listeners are notified of lifecycle events (such as when a test case or individual test starts and finishes) and test outcomes (success, assertion failure, or error due to an uncaught exception).

In Section 5.1, we describe the listener mechanism in detail. JNRTEST provides several built-in listeners, such as recording results and reporting outcomes to the console, including optional test execution timing. Figure 3 demonstrates a typical configuration, where multiple test cases are run and listeners are attached to collect and display results. Some test cases in the example also use extension mechanisms (see Section 7) to demonstrate JNRTEST's extensibility.

The runner configuration shown in Figure 3 is typical for JNRTEST projects. To simplify common use cases further, JNRTEST provides a convenience class, `JnrTestConsoleExecutor`, which preconfigures the runner with default listeners. With this class, developers only need to

```java
public class JnrExamplesTestMain {
  public static void main(String[] args) {
    var recorder = new JnrTestRecorder()
      .withElapsedTime();
    var runner = new JnrTestRunner()
      .add(new FactorialJnrTest())
      .add(new JnrTestMockitoExtension()
        .extendEach(
          new StringServiceWithMockTest()))
      .add(new JnrTestGuiceExtension(
        new RepositoryInMemoryGuiceModule())
          .extendAll(
            new StringServiceWithGuiceTest()))
      .testListener(recorder)
      .testListener(
        new JnrTestConsoleReporter()
          .withElapsedTime());
    runner.execute();
    System.out.println("\nResults:\n\n" +
      new JnrTestResultAggregator()
        .aggregate(recorder));
    if (!recorder.isSuccess())
      throw new RuntimeException(
        "There are test failures");
  }
}
```

Figure 3 An example running test cases from a Java `main`.

```java
public class JnrExamplesTestMain {
  public static void main(String[] args) {
    JnrTestConsoleExecutor()
      .add(new FactorialJnrTest())
      .add(new JnrTestMockitoExtension()
        .extendEach(
          new StringServiceWithMockTest()))
      .add(new JnrTestGuiceExtension(
        new RepositoryInMemoryGuiceModule())
          .extendAll(
            new StringServiceWithGuiceTest()))
      .execute();
  }
}
```

Figure 4 An example running test cases from a Java `main` with `JnrTestConsoleExecutor`.

specify the test cases to be run, as shown in Figure 4 (which has the same semantics as Figure 3).

In Section 8.2, we present a tool that automatically generates such Java `main` classes by discovering all `JnrTest` subclasses in a source directory.

Having separate test suites in JNRTEST does not require additional features in the framework. It is simply a matter of creating multiple Java main files, each specifying a different set of test cases to be run. This approach gives developers complete control over test organization and execution.

A notable difference from JUnit is the order in which tests are executed. In JNRTEST, test cases are run in the exact order they are specified in the `JnrTest` instances, which, in turn, are executed in the order they are added to the runner. In contrast, JUnit executes test cases and tests in an unpredictable, though

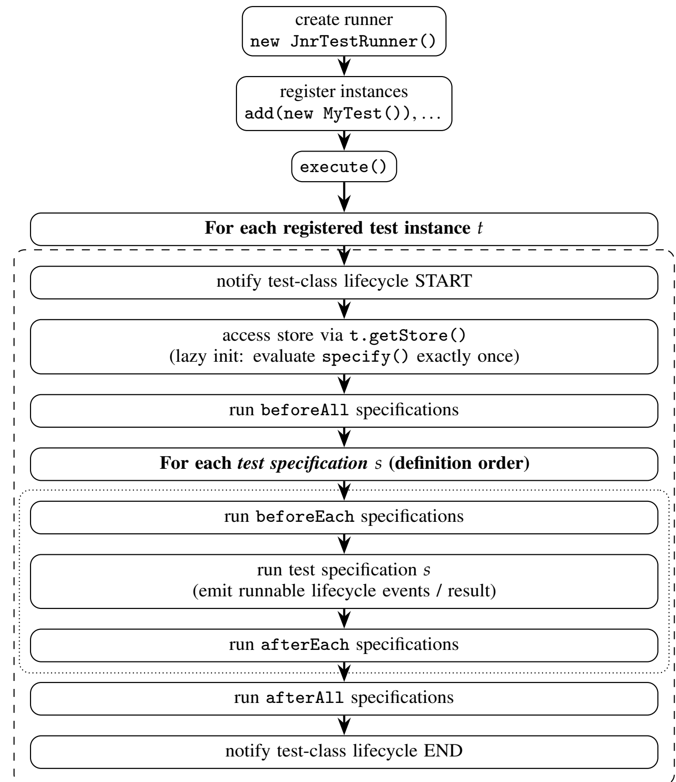

Figure 5 Execution lifecycle implemented by `JnrTestRunner` for JNRTEST test instances registered via `add`.

deterministic, order.[4] While JUnit provides mechanisms to influence test order, it cannot guarantee the order of definition due to its reliance on reflection. JNRTEST's natural and predictable ordering can be advantageous for reproducibility and debugging, though tests should not rely on execution order for correctness. Developers can also override the default order by customizing the `getStore` method, for example, to sort or shuffle tests as needed.

Figure 5 summarizes the high-level execution lifecycle implemented by `JnrTestRunner`. We refer to the Appendix for further and deeper details on the implementation of the main parts of `JnrTestRunner`.

Note that for quick testing, especially when following the TDD approach, you can start with a Java class containing a `main` method using the `JnrTestConsoleExecutor` (Figure 4), and define the tests inside an anonymous subclass of `JnrTest`, as shown in Figure 6. This way, you can quickly write and run tests without creating separate classes.

Later, when all tests work, you can refactor the anonymous class into a proper subclass of `JnrTest` using the IDE refactoring tools. For example, in Eclipse, you can select the anonymous class and use the "Refactor" ⇒ "Convert Anonymous Class to Nested" menu option to create a new class with a proper name inside the same file. Then, apply the second refactoring to move the new class to a separate file, automatically updating the `main` method to use the new class ("Refactor" ⇒ "Move Type to New

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

```java
public class JnrExamplesTestMain {

  public static void main(String[] args) {
    new JnrTestConsoleExecutor()
    .add(new JnrTest("tests for factorial") {
      private Factorial factorial;

      @Override
      protected void specify() {
        // as before
      }
    })
    .execute();
  }
}
```

Figure 6 An example running tests specified in an anonymous class (the tests for factorial are as shown in Figure 2

```java
public interface JnrTestListener {
  /** Notifies the listener of a test class
    lifecycle event */
  void notify(JnrTestLifecycleEvent ev);
  /**
   * Notifies the listener of a test runnable
    lifecycle event */
  void notify(JnrTestRunnableLifecycleEvent ev);
  /**
   * Notifies the listener of a test result */
  void notify(JnrTestResult result);
}
```

Figure 7 The interface for test listeners.

File").

5.1. Test Listeners

As mentioned above, `JnrTestRunner` supports the registration of listeners. Listeners are notified of lifecycle events (such as when a test case or individual test starts and finishes) and test outcomes (success, assertion failure, or error due to an uncaught exception). Listeners decouple the core test execution logic from reporting and result collection, allowing extending or customizing the framework's behavior without modifying its internals. This design also enables users to implement their listeners.

Figure 7 shows the Java interface for listeners. Each method in the interface corresponds to a different type of event that can occur during test execution. The `JnrTestLifecycleEvent` notifies about the lifecycle of a test class, such as when it starts or ends. The `JnrTestRunnableLifecycleEvent` provides notifications for the lifecycle of individual test runnables. The `JnrTestResult` notification delivers the outcome of a test, indicating whether it succeeded, failed an assertion, or encountered an error.

Depending on the listener's needs, some events can be ignored. To facilitate this, JNRTEST also provides an adapter implementation, `JnrTestListenerAdapter`, a superclass with all methods implemented as empty, so users can override only the methods relevant to their use case, reducing boilerplate and

```java
public class JnrTestConsoleReporter extends
    JnrTestListenerAdapter implements
    JnrTestReporterInterface {
  private JnrTestStatistics testStatistics = new
    JnrTestStatistics();

  @Override
  public void notify(JnrTestLifecycleEvent ev) {
    if (ev.status() == JnrTestStatus.START) {
      testStatistics.reset();
      show(ev.toString());
    }
    if (ev.status() == JnrTestStatus.END) {
      show(String.format("Tests run: %d,
    Succeeded: %d, Failures: %d, Errors: %d",
        testStatistics.getTotalTests(),
        testStatistics.getSucceeded(),
        testStatistics.getFailed(),
        testStatistics.getErrors()));
    }
  }

  @Override
  public void notify(JnrTestResult result) {
    switch (result.status()) {
      case FAILED -> {
        testStatistics.incrementFailed();
        result.throwable().printStackTrace();
      }
      case ERROR -> {
        testStatistics.incrementErrors();
        result.throwable().printStackTrace();
      }
      default -> testStatistics
        .incrementSucceeded();
    }
    show(result.toString());
  }

  // Print the message to the console
  private void show(String message) { ... }
}
```

Figure 8 An example of a test listener that shows the results of the executed tests on the console.

improving code clarity.

As shown before, JNRTEST provides several built-in listeners, such as recording results and reporting outcomes to the console, including optional test execution timing. For example, the `JnrTestConsoleReporter` class shown in Figure 8 is a listener that shows the results of the executed tests on the console. It provides immediate feedback to the user, including statistics and error details, and can optionally display test execution timing. We used this listener in the example shown in Figure 3.

5.2. Filtering/Selecting tests

JNRTEST provides a flexible and expressive API for filtering and selecting which tests to execute, possibly combining multiple criteria. Filtering can be applied at two levels:

– *Class filter:* Selects which test classes (i.e., instances of `JnrTest`) are included in the run.

– *Specification filter:* Selects which individual test specifications (i.e., test cases within a class) are executed.

Filters are expressed as Java `Predicate` objects. They can be combined using logical operations such as AND, OR, and negate, by using the methods provided by the Java `Predicate` interface `and`, `or`, and `negate`. Filter methods `specificationFilter` and `classFilter` accepting a `Predicate<JnrTestRunnableSpecification>` or `Predicate<JnrTest>`, respectively, are available in the classes `JnrTestRunner` and `JnrTestConsoleExecutor`. For convenience, JNRTEST also provides methods for filtering by description using a string representing a regular expression. We will show a few examples of how to specify filters.

To run only test classes whose description matches a given pattern, we provide the `filterByClassDescription` method:

```
JnrTestRunner runner = new JnrTestRunner()
    .add(new JnrTest("CalculatorTest") { ... })
    .add(new JnrTest("StringUtilsTest") { ... });
// Only run test classes whose description starts
     with "Calculator"
runner.filterByClassDescription("Calculator.*");
runner.execute();
```

To run only those test specifications whose description matches a pattern, we provide `filterBySpecificationDescription`:

```
runner.filterBySpecificationDescription(".*
    important.*");
runner.execute();
```

Filters can be combined at both levels. For example, to run only specifications with description containing `important` in classes whose description starts with `First`:

```
runner.filterByClassDescription("First.*");
runner.filterBySpecificationDescription(".*
    important.*");
runner.execute();
```

The developer could also rely on some system properties representing the filters to be applied, and use them to configure the runner. Following the Java convention, the system properties can be set on the command line when running the Java main class, using the syntax `-D<propertyname>=<propertyvalue>`, for example:

```
java -Djnrtest.classFilter="First.*" -Djnrtest.
    specificationFilter=".*important.*" ...
```

When using Maven or Gradle (see Section 5.3), system properties can be set in the configuration of the `exec-maven-plugin` or the Gradle equivalent.

5.3. Integration with Java tools

JNRTEST integrates seamlessly with most Java development tools and build systems.

Running JNRTEST tests with Maven or Gradle is straightforward. No special plugin is required. Because JNRTEST does not rely on runtime test discovery, you write a Java class with a `main` method (as shown in Figures 3 and 4) and execute it using the Maven plugin `exec-maven-plugin`.[5]

```
<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 9 An example running test cases from Maven.

Figure 9 shows a typical Maven configuration: You only need to specify the fully qualified name of the Java main class and set the classpath scope to “test”. If the main class throws an exception when there are test failures, the Maven build will fail as expected.

JNRTEST also integrates well with code-coverage tools such as JaCoCo.[6] For example, in Eclipse, you can use the JaCoCo Eclipse plugin to run a main class with the command “Coverage As” ⇒ “Java Application”. From Maven, you configure the `jacoco-maven-plugin` as usual (details are outside the scope of this paper) and use `exec-maven-plugin` with the goal `exec` (instead of `java`), which provides more control to execute the Java main class through the JaCoCo Java agent.

Integration with other tools, such as the PIT mutation testing framework,[7] is still under investigation.

5.4. Running tests in parallel

JNRTEST supports running tests in parallel. Currently, this feature is implemented in the `JnrTestParallelRunner` class, which extends `JnrTestRunner`. The implementation is straightforward: it uses Java’s *ForkJoinPool* to execute test class instances concurrently by redefining the method returning the stream of `JnrTest` instances by returning a parallel stream:

```
public class JnrTestParallelRunner extends
    JnrTestRunner {
@Override
protected Stream<JnrTest> getTestClassesStream(){
  return super.getTestClassesStream().parallel();
}
}
```

Thus, currently, JNRTEST can run test class instances in parallel, but not individual test specifications within the same test class instance. This aligns with our design decision that all

[5] In Gradle, a similar plugin can be used to run a generic Java application.
[6] https://www.eclemma.org/jacoco/.
[7] https://pitest.org/

test specifications within a test class instance should be executed in the order they are defined, to avoid unexpected side effects. Moreover, all test specifications within a test class instance share the same state, which is initialized in the `before` specification.

We provide a thread-safe version of the our listeners, which ensure that results are collected correctly even when tests are run in parallel. Similarly, we provide a thread-safe version of the executor, `JnrTestParallelConsoleExecutor`, which correctly initializes and configures the parallel runner and the thread-safe listeners.

In the future, we could extend the parallel runner to allow for a custom strategy for running test specifications in parallel, or by allowing the user to specify how many test specifications to run in parallel.

This design choice keeps the framework consistent and straightforward while providing expressive and flexible parameterized testing capabilities.

6. Parameterized tests

Parameterized tests are a powerful feature in modern testing frameworks, allowing a single test definition to be executed with a range of input values (Tillmann & Schulte 2005; Tillmann et al. 2010; Lam et al. 2018). This approach helps avoid code duplication and ensures that the tested logic is exercised over various scenarios.

In JNRTEST, support for parameterized tests is built directly into the framework, without requiring changes to the default test lifecycle or the underlying store. Parameterized tests in JNRTEST are provided by `JnrTest` with the generic methods shown in Figure 10. These methods rely on the passed parameters to create several test implementations, with the appropriate parameters provided by the lambda of type `Supplier<Collection<T>>`.[8]

The first version of `testWithParameters` takes a description, a parameter provider, and a test body. It uses the default string representation of each parameter for reporting. The second version adds a `descriptionProvider`, a function that generates a custom string description for each parameter value. This allows for more informative and readable test output, especially when parameters are complex objects or structures.

For example, in Figure 11, we show the parameterized version of the factorial tests (see also the basic version in Figure 2), where `pair` is one of our utility methods for creating pairs (we have other utility methods for triples).

In JUnit 4, writing a parameterized test is known to be cumbersome and complicated. JUnit 5 simplified parameterized tests considerably and also provides several built-in “sources” (e.g., `@CsvSource`, `@CsvFileSource`, `@MethodSource`). For simple cases, these annotations can be very convenient. For example, in JUnit 5, the factorial test could be written as in Figure 12.

By comparing the two examples, we can note that JUnit 5 still relies on annotations and on an implicit mapping between an external “source” and the parameters of a test method. In

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

```
1  protected <T> void testWithParameters(String
       description,
2      Supplier<Collection<T>> parameterProvider,
3      JnrTestRunnableWithParameters<T> testRunnable
       ) {
4    testWithParameters(description,
       parameterProvider,
5      Object::toString,
6      testRunnable);
7  }
8  protected <T> void testWithParameters(String
       description,
9      Supplier<Collection<T>> parameterProvider,
10     Function<T, String> descriptionProvider,
11     JnrTestRunnableWithParameters<T> testRunnable
       ) {
12   var parameters = parameterProvider.get();
13   for (T parameter : parameters) {
14     test(description + descriptionProvider.apply(
       parameter),
15       () -> testRunnable.runTest(parameter));
16   }
17 }
```

Figure 10 Parameterized tests API.

```
1  ...
2    @Override
3    protected void specify() {
4  ...
5      testWithParameters("input,output",
6        () -> List.of(
7          pair(0, 1),
8          pair(1, 1),
9          pair(2, 2),
10         pair(3, 6),
11         pair(4, 24)
12       ),
13       p -> assertEquals(p.second(),
14         fact.compute(p.first()))
15     );
16   }
17 ...
```

Figure 11 An example in JNRTEST testing the factorial implementation with parameterized tests.

particular, for this example, we used `@CsvSource` (Comma Separated Values), which accepts an array of strings. As strings, the compiler cannot check their contents statically. JUnit has to interpret the strings at runtime, parse them, and convert them to the method's parameter types. When the parameters are objects, this typically requires additional configuration (e.g., custom converters).

In contrast, the JNRTEST approach to parameterized tests is entirely based on ordinary Java values and on the Java type system. The test receives parameters as concrete objects, and the compiler enforces type safety at every stage. This means that mismatches in the number or types of parameters are expressed as ordinary Java compilation errors, rather than as runtime failures during argument conversion. The user supplies a collection of parameters—which can be of any type, including complex objects or structures—and a lambda that describes how to run

```
...
  @ParameterizedTest
  @CsvSource({
    "0, 1",
    "1, 1",
    "2, 2",
    "3, 6",
    "4, 24"
  })
  void testFactorial(int input, int expected) {
    assertEquals(expected, factorial.compute(
    input));
  }
...
```

Figure 12 An example in JUnit 5 testing the factorial implementation with parameterized tests.

the test for each parameter. There is no need for string parsing, reflection, or framework-specific converters.

This also affects "easiness" in practice. JUnit 5 can be more concise for very small examples involving only primitives encoded as strings. However, as soon as parameters come from non-trivial sources (files, generated combinations, domain objects) the developer typically writes Java code anyway (e.g., `@MethodSource`) and must still bridge from that source to the test method signature. In JNRTEST, the parameter provider is already just Java code (a `Supplier<Collection<T>>`), so obtaining parameters from any source is uniform and requires no additional framework integration. Our solution is thus completely statically type safe, because the structure of the parameters is explicit and checked by the compiler, at the cost of using a structure when we have more than one parameter (e.g., pairs or records).

The output of running the parameterized test of Figure 11 is as follows (assuming the test runner is configured with our test reporter as shown in Figure 3):

```
[SUCCESS] input,output(0,1)
[SUCCESS] input,output(1,1)
[SUCCESS] input,output(2,2)
[SUCCESS] input,output(3,6)
[SUCCESS] input,output(4,24)
```

The JNRTEST user can provide a custom representation for the current parameters, using the second version of `testWithParameters` (Figure 10) that takes a function responsible for returning the string description:

```
testWithParameters("",
... as before
  p -> String.format("factorial(%d) -> %d",
     p.first(), p.second()),
... as before
);
```

And the output will become:

```
[SUCCESS] factorial(0) -> 1
[SUCCESS] factorial(1) -> 1
[SUCCESS] factorial(2) -> 2
[SUCCESS] factorial(3) -> 6
[SUCCESS] factorial(4) -> 48
```

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 can be used without any static check by the compiler.

Concerning JUnit 5 several built-in "sources" (e.g., `@CsvSource`, `@CsvFileSource`, `@MethodSource`), our design intentionally keeps such facilities out of the core framework. Because JNRTEST only requires a parameter provider of type `Supplier<Collection<T>>`, libraries can offer reusable providers without any special hooks. For example, a small utility library can expose helpers that read CSV/JSON/YAML files, generate cartesian products of input domains, or create randomized inputs, and then return a `Collection<T>` of strongly typed values (e.g., Java records), which can be passed directly to `testWithParameters`. In other words, JNRTEST does not need framework-level support for each source, since any source can be expressed as a typed Java provider.

7. Extensions

Developers often need to extend the functionality of a testing framework to suit their specific needs, such as managing resources, integrating with other libraries, etc. In JNRTEST, we provide a flexible and explicit extension mechanism that allows developers to hook into the test class lifecycle and add custom behavior without relying on reflection or annotations. In particular, we provide two ways to extend the functionality of a test class: by using the test class' store to add runnables that execute before or after each test, or by creating a dedicated extension class that encapsulates the desired behavior.

7.1. Using the test store for extensions

The developer can hook into a test class lifecycle by using the test class' test store and appropriately add a few runnables before/after each/all tests. For example, in (Bettini 2023), we showed how to use this mechanism to recreate the functionality of JUnit 4's `TemporaryFolder` rule in JNRTEST. Here we show two other examples of this mechanism.

The code in Figure 13 shows how to create an extension for managing a Testcontainers[9] container in JNRTEST. Testcontainers is a popular Java library that provides lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that can run in a Docker container, making it easy to write integration tests that depend on external resources. By adding runnables to the store, the extension ensures to start the container before each test and stop it after each test.

This extension can be used in a test class as shown in Figure 14. The extension is instantiated and registered in the test class constructor, and its functionality is accessed via a getter method. This implementation ensures that the container is started before each test and stopped after, providing a clean and isolated environment for every test execution. To start and stop the container only once per test class, the extension must be modified to use `getBeforeAllRunnables()` and `getAfterAllRunnables()` instead.

[9] https://testcontainers.com/

```java
import org.testcontainers.containers.
    GenericContainer;

public class JnrTestContainer<T extends
    GenericContainer<T>> {
  private final T container;

  public JnrTestContainer(
      JnrTest testClass, T container) {
    this.container = container;
    var before = testClass.getStore().
      getBeforeEachRunnables();
    var after = testClass.getStore().
      getAfterEachRunnables();
    before.add(0,
      new JnrTestRunnableSpecification(
      "start testcontainer", container::start));
    after.add(new JnrTestRunnableSpecification(
      "stop testcontainer", container::stop));
  }

  public T getContainer() {
    return container;
  }
}
```

Figure 13 An extension for managing a Testcontainers container in JNRTEST.

```java
public class ExampleTest extends JnrTest {
  private JnrTestContainer<?> c;

  public ExampleTest() {
    super("Testcontainers example");
    this.c = new JnrTestContainer<>(
      this,
      new GenericContainer<>("postgres:15")
        .withExposedPorts(5432)
    );
  }

  @Override
  protected void specify() {
    test("container is running", () -> {
      assertTrue(c.getContainer().isRunning());
      // We can now use the container in tests
    });
  }
}
```

Figure 14 A test class using the Testcontainers extension in JNRTEST.

```java
import org.assertj.swing.fixture.FrameFixture;

public class JnrTestAssertJSwing {
  private FrameFixture fixture;

  public JnrTestAssertJSwing(JnrTest testClass,
    JFrame frame) {
    var before = testClass.getStore().
      getBeforeEachRunnables();
    var after = testClass.getStore().
      getAfterEachRunnables();
    before.add(0, new
    JnrTestRunnableSpecification(
      "show frame",
      () -> {
        fixture = new FrameFixture(frame);
        fixture.show();
      }
    ));
    after.add(new JnrTestRunnableSpecification(
      "clean up AssertJ Swing fixture",
      () -> fixture.cleanUp()
    ));
  }

  public FrameFixture getFixture() {
    return fixture;
  }
}
```

Figure 15 An extension for managing an AssertJ Swing window in JNRTEST.

The code in Figure 15 shows how to create an extension for managing an AssertJ Swing[10] `FrameFixture` in JNRTEST. AssertJ Swing is a library for functional Swing UI testing, providing a fluent API for simulating user interactions and verifying the state of Swing components in Java desktop applications. Its class `FrameFixture` provides methods to interact with Swing `JFrame`, such as clicking buttons, entering text, and verifying the visibility of components.

This extension ensures that the Swing window is shown before each test and cleaned up after each test and that the corresponding `FrameFixture` is properly configured to interact with the window under test, following the same extension pattern as the previous examples. This extension can be used in a test class as shown in Figure 16.

7.2. Using a dedicated extension class

We also provide an abstract base class for writing reusable extensions, whose code is shown in Figure 17. An extension is a plain Java object that receives a concrete test instance and registers additional lifecycle actions in its store. The method `extendAll` registers actions that run once for the whole test class (using the `beforeAll`/`afterAll` lists). The method `extendEach` registers actions that run around every single test (using the `beforeEach`/`afterEach` lists). Both methods delegate to the single abstract method `extend`, which is responsible for adding `JnrTestRunnableSpecification` instances to the provided lists. Finally, both methods return the same test instance, and

[10] https://joel-costigliola.github.io/assertj/assertj-swing.html

```
public class ExampleSwingTest extends JnrTest {
  private JnrTestAssertJSwing sw;

  public ExampleSwingTest() {
    super("AssertJ Swing example");
    GuiActionRunner.execute(() -> {
      JFrame frame = new MyJFrame();
      // configure frame as needed
      return frame;
    });
    sw = new JnrTestAssertJSwing(this, frame);
  }

  @Override
  protected void specify() {
    test("window is visible", () -> {
      sw.getFixture().requireVisible();
      // further AssertJ Swing assertions...
    });
  }
}
```

Figure 16 A test class using the AssertJ Swing extension in JNRTEST.

thanks to generics this preserves the static type of the test class and avoids unsafe casts when chaining extensions.

This extension mechanism is mainly intended to integrate with other libraries or frameworks that need to perform setup/teardown around tests, such as initializing annotated fields or injecting dependencies. We demonstrate it by implementing extensions for popular libraries like Mockito[11] (for mocking (Marri et al. 2009; Mostafa & Wang 2014)) and Google Guice,[12] (a dependency injection framework (Prasanna 2009)), which are commonly used in Java projects. Unlike JUnit, where such integration is typically enabled through framework-discovered annotations and reflection, in JNRTEST the extension is applied explicitly using ordinary Java code. JNRTEST itself remains reflection-free, but it does not prevent developers from using reflection-based systems when implementing tests.

For example, a `JnrTest` can still use Mockito. In such a case, the developer may want to annotate fields of the test class with Mockito annotations like `@Mock` and `@InjectMocks`. In JUnit, mock initialization is typically delegated to framework-provided integration points, such as `MockitoJUnitRunner` in JUnit 4 or `MockitoExtension` in JUnit 5. In JNRTEST, creating such an integration is straightforward: it is enough to extend the base class of Figure 17 and implement the single abstract method, as shown in Figure 18. The extension registers a setup action that opens the mocks before each test, and a teardown action that closes the returned `AutoCloseable` after the test. Our implementation is much more compact than the corresponding JUnit 4 runner or JUnit 5 extension in Mockito's codebase.

This extension can be used with any test class relying on Mockito annotations.

Let us consider a typical Service/Repository cooperation example, where a service class uses a repository to retrieve data.

[11] https://site.mockito.org/.
[12] https://github.com/google/guice.

```
public abstract class JnrTestExtension {

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

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

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

Figure 17 The abstract extension class.

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

public class JnrTestMockitoExtension
  extends JnrTestExtension {

  private AutoCloseable closeable;

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

Figure 18 The extension for Mockito.

```java
public class StringServiceWithMockTest
  extends JnrTest {

  @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())
        .containsExactly("FIRST", "SECOND");
    });
  }
}
```

Figure 19 A test class using Mockito annotations.

```java
public class StringService {
  private StringRepository repository;
  // constructor...
  public Collection<String> allToUpperCase() {
    return repository.findAll().stream()
      .map(String::toUpperCase).toList();
  }
}
```

The example is intentionally simple, but it should be enough to demonstrate how Mockito and our extension cooperate for readers familiar with Mockito. The Service/Repository pattern is also a typical scenario where mocking is useful.

Given the test class of Figure 19, we can use our Mockito extension as follows:

```java
StringServiceWithMockTest testClass =
  new JnrTestMockitoExtension()
    .extendEach(
      new StringServiceWithMockTest());
```

To use the extension, we only rely on standard Java object-oriented mechanisms. No additional reflection is required beyond what is already imposed by the external framework (Mockito in this example).

Creating an extension for Google Guice, allowing a test class to rely on dependency injection, is also straightforward, as shown in Figure 20. Given a Guice `Module`, which configures how implementations should be used to build the object graph, the extension can register an action that injects members into the test instance.

This extension can be used with a test class relying on Guice annotations.

An example is shown in Figure 21. The test class uses Guice's `@Inject` annotation to inject dependencies, such as a repository and a service. It is initialized with our extension, which takes a Guice `Module` that defines how the dependencies should be wired. In this example, we use

```java
...
import com.google.inject.Guice;
import com.google.inject.Module;

public class JnrTestGuiceExtension
  extends JnrTestExtension {

  private Module module;

  public JnrTestGuiceExtension(Module module) {
    this.module = module;
  }

  @Override
  protected <T extends JnrTest>
    void extend(T testClass,
      List<JnrTestRunnableSpecification> before,
      List<JnrTestRunnableSpecification> after) {
    before.add(0, new
    JnrTestRunnableSpecification(
      "inject members", () ->
        Guice.createInjector(module)
          .injectMembers(testClass)));
  }
}
```

Figure 20 The extension for Google Guice.

an in-memory repository for testing purposes defined in the `StringRepositoryInMemoryGuiceModule` class.

```java
StringServiceWithGuiceTest testClass =
  new JnrTestGuiceExtension(
    new StringRepositoryInMemoryGuiceModule())
  .extendAll(
    new StringServiceWithGuiceTest()))
```

Note that this section's extension mechanisms do not require special treatment in our `JnrTest` and `JnrTestRunner` classes.

Recall from Section 4 that when calling `getStore` for the first time, the method `specify` is invoked. This invocation triggers the *declaration* of tests and lifecycle hooks (i.e., registering the corresponding specifications in the store), but it does not execute any test code yet. Therefore, the code shown in this section can safely call `getStore` and then add further specifications to the same store.

That is also why, when adding to the "before" list, the above extensions always insert the specification at the beginning (index "0"). At execution time, this guarantees that the extension-provided setup runs before any `beforeEach`/`beforeAll` hook registered by the test class itself. For example, in Figure 21 the test class registers a `beforeEach` hook that calls `repo.deleteAll()`, and our Guice extension ensures that the injected field `repo` is available before that hook is executed. More generally, this ordering ensures that test hooks and test bodies can rely on state prepared by the extension (e.g., injected fields or Mockito-initialized mocks).

8. Additional Tools

In this section, we briefly present some additional tools that we have developed to complement JNRTEST. These are not part of the core framework but can be useful for developers using

```java
public class StringServiceWithGuiceTest extends
    JnrTest {

  @Inject
  private StringRepository repo;

  @Inject
  private StringService service;
...
  @Override
  protected void specify() {
    beforeEach("clear repository", () ->
      repo.deleteAll());
    test("when repository is empty", () -> {
      assertThat(service.allToUpperCase())
        .isEmpty();
    });
    test("when repository is not empty", () -> {
      repo.saveAll(List.of("first", "second"));
      assertThat(service.allToUpperCase())
        .containsExactly("FIRST", "SECOND");
    });
  }
}
```

Figure 21 A test class using Google Guice annotations.

JNRTEST in their projects. They are deployed as a separate Maven artifact. Moreover, they are also meant to implement IDE integrations, which we plan to release in the future.

Such tools act at the source code level. The classes described in this section are meant to be called from standalone Java applications or IDE plugins.

8.1. Test Discovery

The `JnrTestDiscovery` class provides automated discovery of JNRTEST test classes in a source directory with the method `List<String> discover(String srcDir)`. This tool parses Java source files using the Eclipse JDT compiler[13] and identifies all classes that extend `JnrTest` and can be instantiated (i.e., are not abstract) with a public no-argument constructor (including a default one or inherited from a superclass). The tool returns a list of such classes' fully qualified names.

The discovery process uses type hierarchy analysis based on resolved bindings to correctly identify both direct and indirect subclasses of `JnrTest`.

Such a list of discovered test classes can then be used by other tools, such as the test runner generator described in the following or IDE integrations, to facilitate test execution and management.

8.2. Test Runner Generation

The `JnrTestMainGenerator` class generates a Java main class to execute all JNRTEST subclasses of a source directory. It provides the method `generateMain(String srcDir, String outputDir, String outputClass)` that takes as input the source directory to scan for test classes, the output directory where to write the generated Java source file, and the name of the output class (including package name). The class uses the `JnrTestDiscovery` described above to find all test classes in the specified source directory. The generated class includes a `fillTestRunner` method that instantiates all discovered test classes and adds them to a `JnrTestRunner`, and a `main` method that creates a `JnrTestConsoleExecutor` to run the tests, similar to the one shown in Section 5.

This tool simplifies the process of creating a test runner for a project using JNRTEST, allowing developers to quickly generate a main class that can execute all their tests without manually listing them. When new JNRTEST test classes are created, it is enough to re-run this generator to update the test runner accordingly.

If developers need more control over test execution (e.g., filtering tests, configuring the executor), they can create a minimal Java main class with the configured `JnrTestRunner` or executor and call the `fillTestRunner` method of the generated Java main class to populate it with all discovered tests. Such a custom class needs to be written only once, and it can be reused even when new test classes are added to the project since it relies on the re-generated method to fill the runner.

8.3. JUnit 5 to JnrTest Migration

The `JUnit5ToJnrTestGenerator` class provides automated generation of JNRTEST test classes from existing JUnit 5 test classes. This generator reads JUnit test source files and creates new JNRTEST subclass source files in a separate output directory, leaving the original JUnit files unchanged. This generator, by leveraging the Eclipse JDT AST (Abstract Syntax Tree) rewrite capabilities, performs a direct source-level conversion that transforms JUnit test definition syntax into JNRTEST syntax while preserving the original code structure, comments, and method bodies.

The generation process includes:

- Creating a new test class with a `JnrTest` suffix added to the original name
- Setting the superclass to extend `JnrTest`
- Removing JUnit annotation imports (`@Test`, `@BeforeEach`, etc.)
- Converting annotated methods into calls within a `specify()` method
- Using `@DisplayName` values as test descriptions
- Preserving all test logic and comments from the original source

The generator supports the standard JUnit 5 lifecycle annotations: `@BeforeAll`, `@BeforeEach`, `@AfterAll`, `@AfterEach`, and `@Test`. Each annotated method is converted into an appropriate call to `beforeAll`, `beforeEach`, `afterAll`, `afterEach`, or `test`, with the method body wrapped in a lambda expression.

Since the original JUnit test files remain intact, this tool allows for a gradual migration strategy where both test frame-

[13] The Eclipse JDT compiler (often referred to as ECJ, for Eclipse Compiler for Java) is the Java compiler implementation that ships with the Eclipse Java Development Tools (JDT). It can be run as a standalone compiler outside of the Eclipse IDE. As a full Java compiler, targeting the same language specification as the standard Java compiler (javac), it can parse Java source, perform type resolution and error checking, and generate JVM bytecode. In our context, we use only its parsing and type resolution capabilities.

works can coexist during the transition period. This can significantly accelerate the adoption of JnrTest in existing projects by automating the generation of JnrTest versions of existing JUnit test suites. Alternatively, it can be used to create JnrTest prototypes of existing JUnit tests for evaluation purposes before fully committing to the migration.

Note that this automatic conversion does not cover specific JUnit features like extensions, parameterized tests, and other advanced mechanisms. In such cases, manual migrations may be necessary to fully adapt the tests to JnrTest. Such limitations are typical of other advanced automated refactoring and migration tools like OpenRewrite.[14] In the future, we plan to evaluate OpenRewrite providing a custom recipe for migrating JUnit tests to JnrTest. Currently, though, the JDT AST rewrite capabilities used in our generator are sufficient for the basic migration tasks we aim to support.

We employed this generator to run the second set of benchmarks presented in Section 9, converting our JUnit test suites to JnrTest versions automatically.

9. Evaluation

In this section, we address the empirical parts of our research questions (RQ3–RQ4) and evaluate JnrTest along the design goals stated in the Introduction and in Section 3. We first provide a *general assessment* of the framework's programming model and achieved design constraints (e.g., reflection-free core and ecosystem interoperability). We then state the current *limitations* that delimit the scope of the present prototype. Next, we report *implementation-effort* indicators extracted from the public git history and *code-quality/maintainability* indicators from continuous analysis (including coverage and mutation testing). Finally, we evaluate *performance* by comparing runner-level overhead and end-to-end execution time against JUnit 5 across multiple platforms.

9.1. General Assessment

This subsection summarizes which design constraints are met and what user-facing capabilities the framework currently provides.

The core principle of avoiding reflection and other unsafe runtime features has been fully achieved: JnrTest never uses reflection or dynamic class loading in its implementation. JnrTest code base is fully statically typed, relying exclusively on Java's compile-time type system to ensure type safety.

Sections 4-6 provide illustrative examples of how tests and lifecycle hooks are *declared and executed* in JnrTest. These examples are intended to document the framework's programming model and semantics (e.g., explicit registration, deterministic execution order, and predicate-based selection), rather than to serve as empirical evidence about developer usability. A dedicated empirical study on learnability and developer experience is left as future work (Section 11). In particular, a JUnit test method body can often be moved verbatim into the body of a JnrTest test specification, since JnrTest does not prescribe a different testing methodology. In fact, in Section 8.3, we describe a prototypical tool that automatically generates JnrTest test classes from existing JUnit test classes, preserving the original test logic and comments.

The JnrTest approach to specifying tests offers several advantages over traditional annotation-based frameworks like JUnit:

- *Explicit Descriptions:* Test descriptions are always explicit and decoupled from method names.
- *Functional and Object-Oriented:* By representing tests as objects containing metadata and executable code, JnrTest leverages Java's functional programming features (lambdas) for concise and readable test definitions.
- *No Static/Instance Restrictions:* There is no need to distinguish between static and instance methods for lifecycle hooks.
- *Seamless Integration:* Existing assertion libraries and external tools can be used without restriction, allowing developers to adopt JnrTest incrementally or alongside other frameworks.
- *Single Point of Specification:* The abstract `specify` method ensures all tests and lifecycle events are defined in one place.

Even parameterized tests (Section 6) benefit from this design, as parameters can be defined and manipulated as first-class objects, enabling complex parameter generation and transformation logic, statically checked by the compiler, without relying on reflection or annotations, with a minimal boilerplate for the user.

Our test runner has the single responsibility of executing tests defined in JnrTest subclasses, without any additional features or responsibilities. Test listeners are fully decoupled from the runner, allowing users to customize reporting and logging behavior without modifying the core execution logic. In that respect, we already provide several built-in reporters, as shown in Section 5, but users can also implement custom ones by implementing the corresponding interfaces.

Our test runners and executors are meant to be executed from a Java main class. This allows the integration with build systems like Maven or Gradle, as well as IDEs, by simply invoking a Java main class. Moreover, JnrTest tests can also be integrated with Java tools like JaCoCo for code coverage measurement.

Additionally, we believe our deterministic test ordering (detailed in Section 5) represents an improvement over JUnit's unpredictable test execution sequence, providing more consistent and reproducible test behavior.

As shown in Section 7, JnrTest, through its extensibility mechanisms, also integrates with reflection-based testing libraries like Mockito and dependency injection frameworks like Google Guice. While we avoid reflection in our implementation, users can still employ reflection-based systems alongside JnrTest. We also showed the creation of reusable behaviors, and the integration with established testing frameworks like Testcontainers, and AssertJ Swing with only a few lines of code.

[14] https://docs.openrewrite.org/

Metric (git-derived)	Value
Repository created (first commit date)	2023-01-12
"First usable prototype" window	2023-01-12 → 2023-02-23
Elapsed calendar time to first prototype	42 days
Total commits (repository HEAD)	402
Distinct commit dates (any author)	71
Human commits (L. Bettini)	345 (85.8%)
Bot commits (Dependabot)	57 (14.2%)

Table 1 Git-based development indicators for JNRTEST (commit timestamps and author attribution).

9.2. Limitations

This subsection lists current gaps and missing integrations to avoid overgeneralizing the results.

Despite the usability of JNRTEST with code coverage tools like JaCoCo, our testing framework is not currently integrated with mutation testing tools such as PIT. Moreover, while JNRTEST tests can be executed within IDEs by invoking a Java main class, we do not yet provide dedicated IDE plugins or integrations to enhance the developer experience. For example, results are currently reported on the console (with our default provided reporters), and we do not yet offer graphical test runners or IDE-specific views. The prototype additional tools described in Section 8 are not yet integrated into popular IDEs. However, they provide some foundational capabilities (e.g., test discovery and test runner generation) that can facilitate future IDE integration efforts.

9.3. Implementation Effort

This subsection reports objective, reproducible development indicators derived from the public repository history.

The JNRTEST implementation is a single-developer project: all framework design and implementation work was carried out by the paper's author. The repository also contains automated dependency-update commits produced by *Dependabot*, a GitHub bot that periodically opens pull requests to update declared dependencies; these bot commits do not reflect framework design/implementation effort.

To provide objective, reproducible evidence, we extracted summary indicators directly from the git history of the public repository. In particular: (i) repository start date from the first commit timestamp, (ii) total number of commits, (iii) number of distinct calendar days with at least one commit, and (iv) per-author commit counts. All values below can be reproduced with standard git commands (e.g., `git log`, `git rev-list`, `git shortlog`).

Time to first version (2023). Commit timestamps indicate that the implementation started on 2023-01-12. A coherent, usable first prototype emerged during an initial development burst ending on 2023-02-23, after the merge of a sequence of feature branches/pull requests that introduced the core framework capabilities, including (as reflected in merge messages and commit subjects) test extensions, test cases, listeners/recording, parameter handling, examples, elapsed-time reporting, temporary-folder support, extension mechanisms, corresponding to the submission of (Bettini 2023). Thus, based on public commit timestamps, the calendar time from project inception to a feature-complete first prototype is approximately six weeks (42 days). We emphasize that git timestamps provide an objective *timeline* and a lower bound on development activity (days on which work was committed), but they do not directly translate into person-hours.

From the first version to the current version. After the initial 2023 prototype, the project evolved incrementally. A major extension phase is visible in later bursts of commits in 2025 and early 2026, which add substantial new capabilities (as highlighted by merge commits), such as refactorings, parallel execution/executor support, benchmarking infrastructure, filtering/selection improvements, and coverage/mutation-testing configuration. These later changes account for the difference between the early prototype described in 2023 and the current codebase discussed in the paper. More details on metrics and code quality are provided in Section 9.4. This compactness helps explain why a single experienced Java developer could implement and maintain the framework and why the early prototype could be reached quickly.

Comparison to feature-similar, mature frameworks. A direct cost comparison with JUnit-like frameworks is inherently difficult because such frameworks are mature, widely adopted, and have evolved for decades with broad community contributions, ecosystem integration, and backward-compatibility constraints. In contrast, JNRTEST is a research-driven implementation whose design explicitly targets minimalism and compile-time explicitness, which is reflected both in the compact size of the codebase and in the rapid convergence to a usable prototype evidenced by the early commit timeline.

9.4. Code Quality and Maintainability

This subsection reports internal quality and test-adequacy indicators (static-analysis metrics, coverage, and mutation testing) as evidence about maintainability and correctness support of the core of JNRTEST, addressing RQ3.

We rely on GitHub Actions as our continuous integration system. An automatic build process, based on Maven, is started on GitHub Actions for each commit on our GitHub repository. The process runs all our tests (implemented with JUnit 5) and all our examples (implemented with JNRTEST) to ensure that everything works correctly after each change. We use all the virtual environments provided by GitHub Actions, e.g., Linux, Windows, and macOS. This infrastructure gives us reasonable confidence that our framework works correctly across different operating systems. Moreover, our build on GitHub Actions also keeps track of code quality metrics, described next.

To assess internal code quality, and in general, code metrics including complexity, we rely on SonarCloud (the cloud offering of SonarQube[15]) as a static-analysis platform that operationalizes widely used code-quality rules and metrics in continuous

[15] https://www.sonarsource.com/products/sonarcloud/

Metric	Value
Lines of Code (LOC)	874
Statements	299
Functions	152
Classes	30
Files	28
Comment Lines	394
Comments (%)	31.1%
Derived: LOC per class (874/30)	29.1
Derived: Functions per class (152/30)	5.1

Table 2 Size-related metrics (SonarCloud).

inspection settings.[16] SonarQube is not only widely adopted in practice, but it is also explicitly analyzed and compared in the empirical software engineering literature: (Avgeriou et al. 2020) report SonarQube as the most popular technical-debt (Kruchten et al. 2012; Tom et al. 2013; Z. Li et al. 2015) measurement tool in their study and summarize the available empirical evidence on the validity of tool-reported measures. Prior work has also investigated realistic SonarQube usage and remediation behavior on large datasets of reported issues (Marcilio et al. 2019). Moreover, SonarQube is routinely treated as a mainstream static analysis tool in comparative studies (Lenarduzzi et al. 2023), and its reported issues have been examined for relationships with faults and changes in large-scale empirical analyses (Lenarduzzi et al. 2020). Finally, SonarQube's technical-debt assessment is historically connected to the SQALE method (Letouzey & Ilkiewicz 2012), and practitioner-oriented literature documents its integration into quality monitoring workflows (Campbell & Papapetrou 2013).

Table 2 summarizes size-related metrics: the core codebase comprises 874 lines of code distributed across 28 files and 30 classes, for a total of 152 functions and 299 statements. This corresponds to roughly 29.1 LOC per class (874/30) and about 5.1 functions per class (152/30). Documentation density is relatively high, with 394 comment lines (31.1%).

We then inspected complexity metrics (Table 3). Cyclomatic complexity (McCabe 1976) estimates control-flow complexity by counting linearly independent paths through the control-flow graph, whereas cognitive complexity (Campbell 2018) targets understandability by penalizing breaks in the linear reading flow and nested constructs. Overall control-flow complexity is moderate (184 total cyclomatic complexity; ≈1.21 per function, 184/152) but concentrated in a small set of key orchestration classes: `JnrTestRunner` (30), `JnrTest` (24), and `JnrTestConsoleReporter` (19) together account for 73/184 ≈39.7% of total cyclomatic complexity, while the top five classes exceed 102/184 ≈55.4%. Cognitive complexity is low in absolute terms (44 total) and is dominated by reporting/execution components (e.g., `JnrTestConsoleReporter` 12 and `JnrTestRunner` 8), whereas most remaining components exhibit negligible cognitive complexity (including 46 components with a score of 0).

Finally, Table 4 reports dynamic evidence. The accompanying test suite comprises 104 unit tests, all passing (0 errors, 0 failures, 0 skipped; 100% success), and the reported results indicate complete coverage of the analyzed code: 433 executable lines and 66 conditions are fully covered, yielding 100% line and 100% condition coverage (0 uncovered lines and 0 uncovered conditions).

[16] The link to our SonarCloud project can be found in the GitHub repository of JnrTest.

Cyclomatic Complexity (total = 184)

Component	Score
`JnrTestRunner.java`	30
`JnrTest.java`	24
`JnrTestConsoleReporter.java`	19
`JnrTestConsoleExecutor.java`	15
`JnrTestStatistics.java`	14
`JnrTestRecorder.java`	13
`JnrTestThreadSafeRecorder.java`	13
`JnrTestFilters.java`	11
`JnrTestResultAggregator.java`	11
`JnrTestStore.java`	11
`JnrTestThreadSafeConsoleReporter.java`	6
`JnrTestConsoleParallelExecutor.java`	4
`JnrTestExtension.java`	3
`JnrTestListenerAdapter.java`	3
`JnrTestReporterInterface.java`	2
`JnrTestLifecycleEvent.java`	1
`JnrTestParallelRunner.java`	1
`JnrTestRecorderInterface.java`	1
`JnrTestResult.java`	1
`JnrTestRunnableLifecycleEvent.java`	1
Components with score 0 (not listed)	8
Top 3 subtotal	73
Top 3 share of total	39.7%
Top 5 subtotal	102
Top 5 share of total	55.4%
Average per function (184/152)	1.21

Cognitive Complexity (total = 44)

Component	Score
`JnrTestConsoleReporter.java`	12
`JnrTestRunner.java`	8
`JnrTestRecorder.java`	6
`JnrTestThreadSafeRecorder.java`	6
`JnrTestFilters.java`	4
`JnrTestResultAggregator.java`	3
`JnrTest.java`	2
`JnrTestThreadSafeConsoleReporter.java`	2
`JnrTestConsoleExecutor.java`	1
Hidden components with score 0	46

Table 3 Complexity metrics and per-component contributors (SonarCloud).

Together, these results suggest that the project achieves high structural test adequacy while keeping most modules simple, with remaining complexity localized to a few central execution/reporting classes. In this context, we use *simple* to mean predominantly low per-function control-flow and understandability complexity (average cyclomatic complexity ≈ 1.21 per function and cognitive complexity concentrated in few components; Table 3).

The current overall code quality and maintainability of the JnrTest on SonarCloud can be seen in Figure 22, showing that all quality-gate conditions are satisfied with no issues in any of the analyzed dimensions (bugs, vulnerabilities, code smells, coverage, duplications, and technical debt).

Metric	Value
Unit Tests	104
Errors	0
Failures	0
Skipped	0
Success	100%
Coverage (overall)	100%
Lines to Cover	433
Uncovered Lines	0
Line Coverage	100%
Conditions to Cover	66
Uncovered Conditions	0
Condition Coverage	100%

Table 4 Test execution and coverage metrics (SonarCloud).

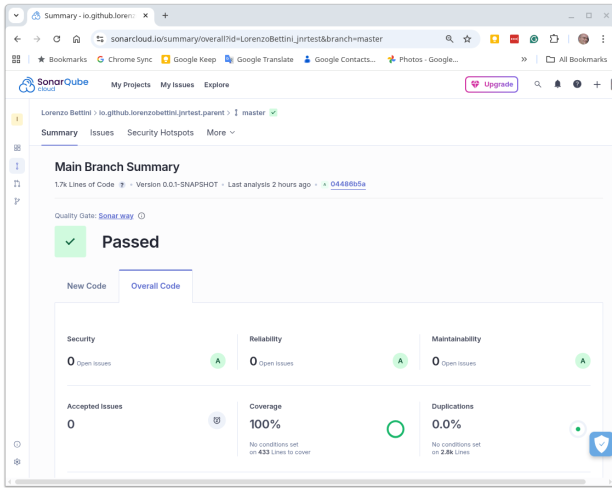

Figure 22 SonarCloud quality gate status for JNRTEST.

Code coverage can be a faulty metric (Marick et al. 1999; Hemmati 2015), and reaching 100% can often be considered too much work with no guarantees. However, code coverage can still be valuable (Chen et al. 2002; Ivanković et al. 2019) if used wisely. In particular, we use code coverage as a guide to identify untested parts of the codebase, rather than as an absolute measure of quality. To get even more confidence about the effectiveness of our test suite, we also employ mutation testing, as described next. Mutation testing (Woodward 1993; Papadakis et al. 2019) is another valuable technique for assessing test suite effectiveness. By introducing small changes (mutations) to the code and checking if the existing tests catch these changes, we can identify weaknesses in our tests. We use the PIT mutation testing tool, which is widely adopted in the Java ecosystem, to run mutation tests as part of our CI pipeline.

In our build, we always ensure we have 100% code coverage and 100% killed mutants, which gives us high confidence in the effectiveness of our test suite. The build is configured to fail if we go below these thresholds, ensuring that we maintain high standards as the codebase evolves.

9.5. Performance

This subsection addresses RQ4 by quantifying runner-level overhead and end-to-end execution time relative to JUnit 5. The benchmark code is part of the JNRTEST Maven build (enabled via a dedicated Maven profile) and can be reproduced from the public JNRTEST source repository. All benchmark results reported in this subsection (including raw outputs) are archived in a separate repository, distinct from the source repository: https://github.com/LorenzoBettini/jnrtest-performance.

Because the end-to-end benefit of reducing framework overhead depends on the fraction of time spent in the test bodies, Amdahl's law (Amdahl 1967) suggests that speedups should decrease as tests become more computation- or I/O-heavy. We therefore report two complementary measurements: (i) a controlled benchmark designed to expose per-test overhead, and (ii) an end-to-end benchmark on a realistic test suite where the test bodies are non-trivial.

Experimental setup. We executed the benchmarks on GitHub Actions runners (Ubuntu, Windows, macOS) and on two Linux machines with different performance profiles. The first machine is a Dell Pro Max Tower T2 ($24 \times$ Intel Core Ultra 9 285K, 32 GB RAM), representing modern high-end desktop hardware. The second machine is an older Dell XPS 13 laptop (Intel Core i7-8550U, 16 GB RAM), representing lower-end mobile hardware. For JUnit 5 we rely on the `maven-surefire-plugin` execution, and for JNRTEST we invoke the runner via the `exec-maven-plugin`. We report the times printed by the corresponding runners (JUnit per-class "Time elapsed" and JNRTEST per-class "Time elapsed"), and totals are computed as the sum of per-class times. For both JUnit 5 and JNRTEST, the reported values are runner-reported wall-clock times for executing a given test class (and its associated framework work), rather than isolated timings of individual assertion statements. This choice reflects the practical cost paid by developers when running tests, and it keeps the comparison focused on end-to-end test execution at the class granularity.

Benchmark 1: controlled overhead microbenchmark. We generate pairs of test classes (JUnit 5 and JNRTEST) that execute the *same* test body a varying number of times (10, 100, 1000, 5000). The test body is intentionally small but not empty, since it performs two AssertJ-based checks on in-memory data (string containment predicates on a collection of strings). This benchmark is meant to highlight runner overhead (test discovery, invocation, and reporting), which tends to dominate when test bodies are short.

To better attribute the source of the overhead gap, we also run the generated JUnit 5 tests with `@TestInstance(TestInstance.Lifecycle.PER_CLASS)`. By default, JUnit creates a fresh instance of the test class for each `@Test` method, while PER_CLASS reuses a single instance per test class and thus removes the per-test instance-construction component. This isolates whether the observed differences are primarily explained by object creation versus runner-level work such as discovery, dispatch, and reporting. Moreover, this aligns JUnit's test-instance lifecycle with JNRTEST, where all specifications of a test class execute on the same test-class instance (shared fixture state, Section 5).

Table 5 reports the execution times across all platforms, and Figures 23 and 24 visualize the scaling trend. Across GitHub

Platform / Runner	10	100	1000	5000
Ubuntu (CI)				
JUnit 5	0.033	0.154	0.726	1.491
JUnit 5 PER_CLASS	0.033	0.149	0.591	1.226
JNRTEST	0.001	0.005	0.020	0.031
macOS (CI)				
JUnit 5	0.015	0.070	0.332	0.588
JUnit 5 PER_CLASS	0.016	0.075	0.231	0.563
JNRTEST	0.001	0.003	0.007	0.019
Windows (CI)				
JUnit 5	0.046	0.178	0.952	1.851
JUnit 5 PER_CLASS	0.056	0.179	0.704	1.682
JNRTEST	0.001	0.006	0.022	0.036
Dell Tower (Linux)				
JUnit 5	0.010	0.030	0.112	0.238
JUnit 5 PER_CLASS	0.009	0.028	0.099	0.225
JNRTEST	0.001	0.002	0.003	0.010
Dell XPS 13 (Linux)				
JUnit 5	0.034	0.144	0.783	1.269
JUnit 5 PER_CLASS	0.037	0.155	0.534	0.989
JNRTEST	0.001	0.004	0.015	0.027

Table 5 Benchmark 1 (generated tests): wall-clock time in seconds for executing the same test body a varying number of times.

Actions runners, at 5000 test executions the runtime drops from 1.491s to 0.031s on Ubuntu, from 1.851s to 0.036s on Windows, and from 0.588s to 0.019s on macOS. On the two Linux machines we observe the same pattern, with a drop from 0.238s to 0.010s on the Dell Tower and from 1.269s to 0.027s on the Dell XPS 13 laptop. Enabling PER_CLASS reduces JUnit 5 times in this benchmark, for example on Ubuntu at 5000 tests the time decreases from 1.491s to 1.226s, and on the Dell XPS 13 laptop the time decreases from 1.269s to 0.989s. However, PER_CLASS does not close the gap to JNRTEST, which remains tens of times faster in this overhead-dominated regime. Overall, the separation between JUnit 5 and JNRTEST widens as the number of tests grows, indicating a substantially smaller per-test framework cost for JNRTEST, compatible with an execution model that avoids annotation scanning and reflective invocation within the framework.

Benchmark 2: end-to-end execution of a realistic suite. To complement the overhead-focused microbenchmark, we also measure a realistic scenario by executing the JNRTEST project test suite. We run the suite as JUnit 5 tests, then translate the same test classes to JNRTEST specifications (using the `JUnit5ToJnrTestGenerator` tool described in Section 8.3) and run them with the JNRTEST runner. In both cases, the executed test code is the same, and only the execution engine differs. The suite consists of 12 test classes and 104 test cases overall (Section 9.4). Enabling `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` for this suite did not materially change the totals, so we omit those results.

Table 57reports total times (sum of per-class times) and speedups across CI runners and the two Linux machines. On GitHub Actions, JNRTEST is faster by about 7% on Ubuntu (1.631s vs. 1.519s), 11% on Windows (2.337s vs. 2.104s), and 43% on macOS (2.738s vs. 1.910s). On the Linux machines, the speedups are consistent with the CI Linux results, at about 10% on the Dell Tower (0.744s vs. 0.677s) and 11% on the Dell XPS 13 laptop (1.519s vs. 1.370s). This behavior is expected in an end-to-end suite, where a larger fraction of the runtime is spent in test bodies and test-support logic, thereby bounding the impact of reducing runner overhead.

Platform	JUnit 5 (s)	JNRTEST (s)	Speedup
Ubuntu (CI)	1.631	1.519	1.07×
macOS (CI)	2.738	1.910	1.43×
Windows (CI)	2.337	2.104	1.11×
Dell Tower (Linux)	0.744	0.677	1.10×
Dell XPS 13 (Linux)	1.519	1.370	1.11×

Table 6 Benchmark 2 (real test suite, 104 test cases): total execution time in seconds (sum of per-class times) and speedup (JUnit divided by JNRTEST).

Discussion and relation to reflection overhead (Amdahl-law interpretation). The two benchmarks are consistent with Amdahl's law (Amdahl 1967), since the more overall runtime is dominated by test code (which is identical in both systems), the less any runner-level optimization can impact the end-to-end time. Benchmark 1 indicates that, when the test body is small, runner overhead is a large fraction of runtime and can be reduced dramatically by JNRTEST's execution model (no annotation scanning, no reflective invocation, and a more direct runner architecture). The PER_CLASS configuration reduces JUnit 5 times in Benchmark 1, confirming that per-test instance creation can contribute when tests are extremely fine-grained, but the remaining gap to JNRTEST indicates that additional runner-level overheads (beyond instance construction) dominate in this regime. In Benchmark 2, where test bodies and test-support code contribute a larger fraction of runtime, the resulting speedups are smaller but remain positive across all platforms and hardware.

Importantly, these measurements do *not* claim that the entire improvement comes solely from "removing reflection", since the difference also includes other sources of overhead in the JUnit execution stack during the test phase (e.g., test discovery and execution through the JUnit Platform). These measurements exclude fixed costs outside the runner (e.g., Maven startup and plugin initialization). Nevertheless, Benchmark 1 demonstrates that runner-level overhead can be substantial in highly granular test suites, and Benchmark 2 shows that reducing this overhead can still yield measurable end-to-end benefits in a realistic setting, even when Amdahl's law limits the maximum achievable gain.

Threats to validity. The benchmarks are executed in CI environments and on a limited set of physical machines, so absolute timings may vary due to shared infrastructure, OS configuration, and hardware differences. However, we observe consistent

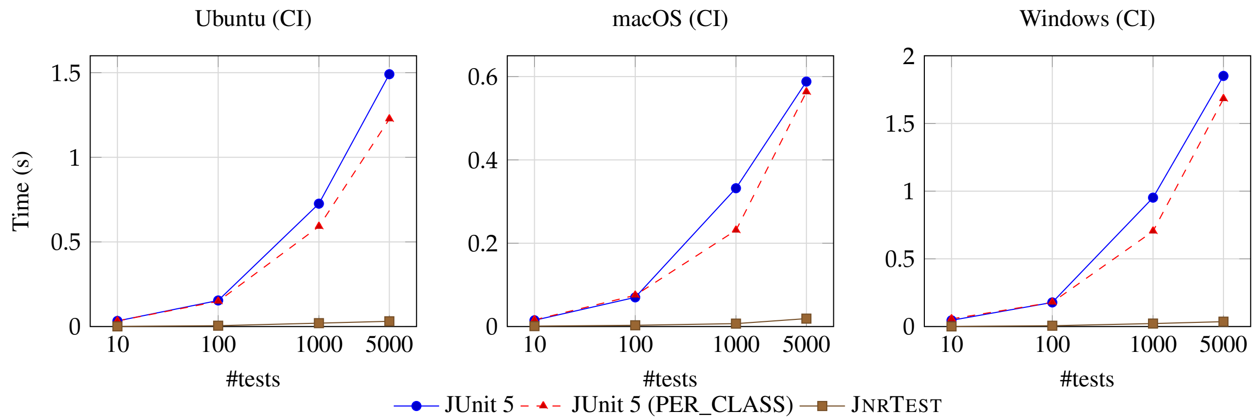

Figure 23 Benchmark 1 on GitHub Actions: execution time vs. number of tests (log-scale x-axis).

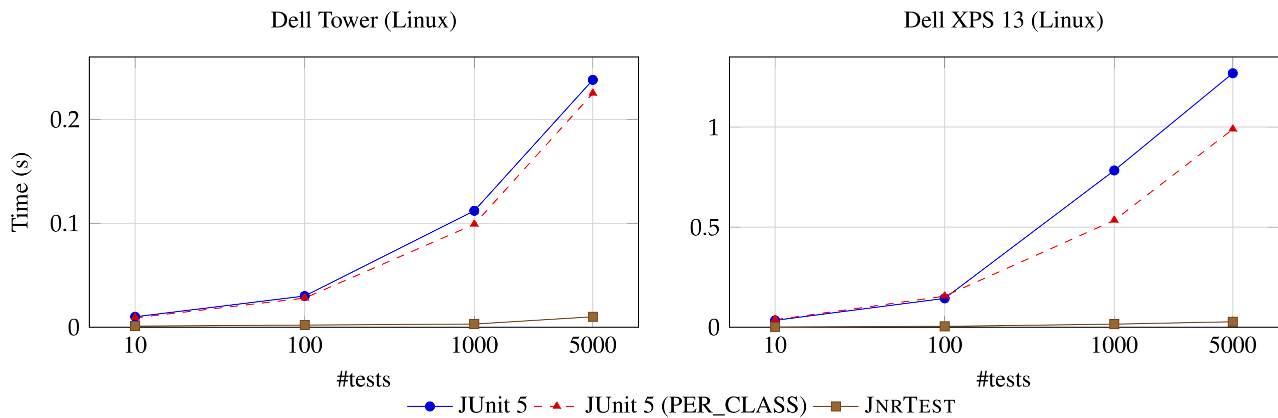

Figure 24 Benchmark 1 on two Linux machines: execution time vs. number of tests (log-scale x-axis).

trends across three CI operating systems and two very different Linux machines, which reduces the likelihood that the observed speedups are an artifact of a single environment. Future work includes repeating each measurement multiple times and reporting confidence intervals, and adding JMH-based microbenchmarks to isolate specific costs (for example reflective invocation versus direct calls).

Note that we report runner-reported times rather than external wall-clock measurements of the full Maven invocation, so the results reflect the execution phase as observed by the test runners and may not capture fixed overheads outside the runner (e.g., Maven startup and plugin initialization). Because JUnit 5 is executed via `maven-surefire-plugin` and JNRTEST via `exec-maven-plugin`, end-to-end wall-clock comparisons of full Maven builds can include plugin-specific fixed costs that are not represented by the runner-reported times. In informal end-to-end runs of the full Maven invocation (not systematically benchmarked), the fixed overhead of `maven-surefire-plugin` appears more noticeable than that of `exec-maven-plugin` when the project contains thousands of generated tests, whereas it is less apparent on the realistic suite. A systematic comparison of full Maven invocation times is left to future work.

10. Related Work

Because runtime reflection is the dominant mechanism for discovery and extension in mainstream Java testing frameworks, there are relatively few directly comparable Java testing frameworks built entirely on non-reflective mechanisms. We therefore first discuss testing frameworks in languages where reflection is absent or not used for discovery, and then position JNRTEST within the broader Java and JVM ecosystem. Finally, we summarize complementary work on reflection avoidance in other Java frameworks, empirical studies of testing practice, and prior work on reflection performance. The remainder of this section provides extended context on implementation approaches and empirical evidence relevant to our design choices.

Table 7 summarizes the main testing approaches discussed in this section along the key design axes that matter for JNRTEST. The table highlights that JUnit historically already offered a reflection-free style (JUnit 3 "static" suites), but that modern mainstream Java frameworks primarily rely on runtime reflection for discovery and extension; JNRTEST can be seen as revisiting the explicit style with modern Java lambdas to reduce verbosity.

Approach	Test specification surface	Discovery / invocation mechanism	Reflection	Static checks on test signatures
JUnit 3 (static suites) (Beck & Gamma 1998)	Subclass `TestCase`; explicit `TestSuite` building	Explicit registration (suite construction); direct calls (no lookup)	No	Yes
JUnit 3 (dynamic naming) (Beck & Gamma 1998)	Subclass `TestCase`; methods named `test*`	Name-based selection; reflective lookup + `invoke`	Yes	Partial (checked at runtime)
JUnit 4 (Beck & Savoia 2006; Tudose 2020)	Methods annotated `@Test`, `@Before`/`@After`, etc.	Reflective scan of annotations; reflective invocation	Yes	No (runtime only)
JUnit 5	Annotations + extension API (callbacks, parameter resolution)	Reflective discovery/invocation + extension dispatch	Yes	No (runtime only)
JNRTEST (this work)	Subclass `JnrTest`; register tests/hooks as lambdas	Explicit registration in `specify()`; direct invocation	No	Yes
C++ (e.g., Catch2 / GTest)	Macros/templates (often with lambdas for bodies)	Compile-time expansion + static registration	No	Yes
Rust (+ `rstest`) (Klabnik et al. 2024)	Functions + attribute macros (e.g., `#[test]`)	Compile-time macro expansion; no runtime metadata needed	No	Yes

Table 7 Positioning JNRTEST relative to lightweight and/or reflection-free testing approaches (including historical JUnit 3).

Reflection-free testing frameworks outside Java (closest comparisons). Languages like C++, which lack built-in reflection, have long adopted reflection-free testing by necessity. Frameworks such as Google Test[17] and Catch2[18] rely on macros, templates, and static registration to define and organize tests. For instance, test cases are written using macros, which expand into statically linked code that registers tests without relying on runtime introspection. Similarly, Catch2 supports parameterized tests, fixtures, and test filtering via macros and compile-time constructs. These frameworks achieve type safety and predictable behavior, and often use lambdas or inline expressions to express assertions concisely. The explicitness and static guarantees in these C++ frameworks mirror the approach taken in JNRTEST, which favors lambda-based test specifications and compile-time configuration over reflection or runtime annotations.

Rust (Klabnik et al. 2024), another statically typed language, also avoids runtime reflection entirely, instead relying on compile-time mechanisms for test specification and discovery. Tests in Rust are ordinary functions annotated with `#[test]`, which are expanded by procedural macros during compilation. While these annotations may resemble Java's annotations, they do not rely on runtime metadata, and any signature mismatches (e.g., wrong function type) are caught by the compiler. Frameworks like `rstest`[19] offer parameterized testing using attribute macros and static expansion, eliminating the need for runtime conversion or introspection. Rust's type-driven approach aligns closely with JNRTEST 's design, where test logic and metadata are encoded in the type system and functional constructs, ensuring both type safety and predictable test execution without reflection. Rust also provides an explicit "escape hatch" from some compile-time guarantees via `unsafe`. Empirical studies show that `unsafe` is sometimes necessary but can often be avoided in practice (Zhang et al. 2023), and ecosystem-level analyses observe that vulnerable code regions disproportionately involve `unsafe` constructs (Zheng et al. 2024; Cui et al. 2024). This perspective is analogous to our stance on Java reflection: powerful and standard, but best kept deliberate, localized, and encapsulated when avoidable.[20]

Positioning within Java testing frameworks (concepts and extension mechanisms). 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" for tests. Thus, JNRTEST should be easily usable for developers already familiar with existing testing frameworks, though its main context is the Java language.

While JNRTEST shares fundamental concepts with existing frameworks, its extension mechanism differs significantly from mainstream approaches. JUnit 5 introduced a sophisticated extension model based on annotations like `@ExtendWith` and interfaces such as `BeforeEachCallback`, which rely heavily on reflection for discovering and invoking extension methods. TestNG follows a similar pattern with listeners and method interceptors. In contrast, JNRTEST's extension mechanism (detailed in Section 7) leverages object-oriented composition and explicit method calls, providing compile-time type safety and eliminating the runtime overhead associated with reflective extension

[17] https://github.com/google/googletest
[18] https://github.com/catchorg/Catch2
[19] https://github.com/la10736/rstest

[20] See also empirical evidence on how `unsafe` is used in real code (Astrauskas et al. 2020) and foundational work explaining why the safety story relies on carefully structured unsafe abstractions (Jung et al. 2018).

discovery.

Functional / DSL-style test specification (within Java and other dynamic ecosystems). Within the Java ecosystem, Lambda Behave[21] shares with JNRTEST the goal of leveraging lambda expressions for test specifications, providing a more functional approach to testing compared to annotation-based frameworks. However, Lambda Behave focuses primarily on behavior-driven development (BDD) with a specific DSL for describing specifications, whereas JNRTEST aims to be a general-purpose testing framework that maintains simplicity while avoiding reflection entirely. Similarly, Expectations[22] provides a DSL for expressing test expectations using lambda expressions and method references, emphasizing readability and expressiveness. While Expectations shares with JNRTEST the use of functional programming constructs, it is designed as an assertion library that works within existing test frameworks like JUnit, rather than as a complete testing framework replacement that avoids reflection at the framework level. The Spock Framework[23] for Groovy also provides an expressive approach to testing with its specification-based syntax. Still, it operates in a dynamically typed environment and relies on Groovy's metaprogramming capabilities, which contrasts with JNRTEST's emphasis on static type safety and compile-time guarantees in pure Java.

The shape of our test specifications, i.e., the signature of `test`, might remind us of test specifications in the JavaScript testing framework Jest.[24] Indeed, we took some initial inspirations from Jest for specifying tests in JNRTEST. Similarly, in the Lua ecosystem, the Busted testing framework provides a BDD-style DSL where test suites and test cases are expressed as anonymous functions (closures) passed to combinators such as `describe` and `it`, leveraging the language's first-class functions rather than reflection.[25] Of course, the rest of the context (programming language, static typing vs. dynamic typing) does not allow for additional comparisons between JNRTEST and Jest and Busted.

Reflection avoidance in non-testing Java frameworks (context for implementation choices). Reflection and annotations are heavily used in several Java frameworks, independently of testing, and in some cases alternatives have been proposed that aim to reduce or eliminate reflective execution. For example, the dependency injection framework Google Guice, which we have used in Section 7, is based on reflection. To "address many of the development and performance issues that have plagued reflection-based solutions," an alternative dependency injection framework has also been proposed by Google: Dagger.[26] Indeed, Dagger is meant to be a "fully static, compile-time dependency injection framework" for Java. It is based on code generation: it automatically generates code that "mimics the code that a user might have hand-written". Thus, it is still based on annotations `@Inject`, `@Provides`, etc., but it processes them at compile-time to generate code. 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[27] heavily relies on reflection. The Java web framework Takes[28] is a demonstration that even a web framework can be implemented in Java with only OOP mechanisms. We share the same goal as Takes. Similarly to Takes, in JNRTEST, we do not even have public static methods nor `instanceof` and downcasts.

Functional programming trends in Java framework design (context). Modern Java functional programming features in framework design have gained attention in recent years. JNRTEST's approach of representing tests as lambda expressions aligns with broader trends toward functional programming in Java, as seen in frameworks like Spring's WebFlux (Deinum et al. 2023) and various reactive programming libraries. This functional approach reduces boilerplate code and enables more expressive and composable test definitions, similar to the benefits observed in functional programming languages like Scala's ScalaTest framework (Hunt 2014).

Empirical studies of testing practice and maintainability (context). Gonzalez et al. (Gonzalez et al. 2017) systematically investigated the prevalence of maintainability-oriented testing patterns in open-source software. Their results show that while some patterns are commonly adopted, others are less frequently used, suggesting that there is still room for improvement and further tool support in test code maintainability. In contrast, our work on JNRTEST focuses on using statically checked Java language features, avoiding unsafe use of reflection, which also undermines performance to some extent, providing complementary insights into the design and evolution of Java testing frameworks.

Islam et al. (Islam et al. 2023) present a comprehensive analysis of the evolution of software testing practices in Java open-source projects. Their study documents how the adoption of testing frameworks and techniques has changed over the years, reflecting broader trends in the Java ecosystem. While their work focuses on the macro-level evolution and adoption patterns, our work on JNRTEST addresses specific technical challenges—such as type safety, performance, and extensibility—in designing modern Java testing frameworks. Thus, our contribution complements their findings by providing a concrete framework that responds to the evolving requirements identified in their empirical analysis.

Reflection overhead and performance evidence (bridge to evaluation). Our performance evaluation (Section 9) contributes empirical evidence to understanding reflection's impact on test execution speed in the context of testing frameworks. The performance improvements we observe align with the well-documented overhead of reflection in Java applications (Livshits et al. 2005; Tudose et al. 2013; Landman et al. 2017; Evans 2023), and demonstrate the practical benefits of our design ap-

[21] https://richardwarburton.github.io/lambda-behave/
[22] https://expectations.dsl.builders/
[23] https://spockframework.org/
[24] https://jestjs.io/.
[25] https://lunarmodules.github.io/busted/
[26] https://dagger.dev/.
[27] https://spring.io/.
[28] https://github.com/yegor256/takes.

proach. Future research could extend our benchmarking methodology to larger codebases and more diverse testing scenarios.

11. Conclusions and Future Work

In this paper, we presented the prototype implementation of a Java testing framework without using reflection. We do not circumvent the Java static type system, and we do not incur the run-time overhead of reflection.

Currently, JNRTEST does not support nested tests (a feature introduced in JUnit 5). We plan to investigate how to implement them in our framework. However, we do not see the lack of nested tests as a huge limitation. Since test execution has a predefined and deterministic order (Section 5), the developer can easily organize tests in JNRTEST and create test suites. Another future extension is to handle skipping of tests, for example, based on predicates passed when calling `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. The developer can quickly jump to the corresponding code in a `JnrTest` through this view. The additional tools described in Section 8 are a first step in this direction.

Using the JUnit 5 migrator tool described in Section 8.3, we plan to create a benchmark suite of JNRTEST test classes converted from existing JUnit 5 test suites of open-source projects. This benchmark will allow us to perform a more extensive evaluation of JNRTEST compared to JUnit 5 in terms of performance, similar to the preliminary evaluation presented in Section 9.

Concerning a broad behavioral equivalence with JUnit across arbitrary existing test suites, a direct mutation-testing comparison that mutates test bodies is of limited value for our contribution, since both JUnit tests and JNRTEST tests ultimately execute the same user-provided code and would be affected similarly. More informative future work is to evaluate *framework-mechanics equivalence* on a curated benchmark of specifications implemented in both styles, focusing on discovery/selection behavior, lifecycle ordering, parameterized-test expansion, and reporting/diagnostics under systematic perturbations of metadata or registration (e.g., missing annotations vs. missing registrations, conflicting lifecycle hooks, or filtering corner cases).

As future work, we also plan to evaluate developer-facing usability with an empirical study (e.g., controlled tasks and/or observational studies on realistic codebases). Such a study should focus on the *mechanics* of declaring tests and lifecycle hooks—because JNRTEST does not change testing methodology or test logic, but only how tests are declared and discovered. In particular, we do not argue that existing projects should migrate wholesale from JUnit to JNRTEST. Instead, we position JNRTEST as a proof-of-concept and a reference design point in the space of Java testing frameworks, showing what becomes possible when test declaration and execution rely only on ordinary, statically checked Java mechanisms. Accordingly, any adoption would most naturally happen (i) in new codebases, (ii) in modules where reflection/annotation-based discovery is undesirable, or (iii) in settings where teams explicitly value a minimal and statically analyzable core over "automatic" discovery.

References

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the spring joint computer conference* (p. 483–485). ACM.

Astrauskas, V., Matheja, C., Poli, F., Müller, P., & Summers, A. J. (2020). How do programmers use unsafe rust? *Proc. ACM Program. Lang.*, *4*(OOPSLA), 136:1–136:27.

Avgeriou, P. C., Taibi, D., Ampatzoglou, A., Fontana, F. A., Besker, T., Chatzigeorgiou, A., … others (2020). An overview and comparison of technical debt measurement tools. *IEEE software*, *38*(3), 61–71.

Beck, K. (2000). *Extreme programming explained: embrace change*. Addison-Wesley Professional.

Beck, K., & Gamma, E. (1998). Test infected: Programmers love writing tests. *Java Report*, *3*(7), 37–50.

Beck, K., & Savoia, A. (2006). *JUnit 4 and Java SE 5: Better Testing by Design.* JavaOne Conference 2006, Session TS-1580 (slides). Retrieved 2026-01-09, from https://docs.huihoo.com/javaone/2006/tools/ts-1580.pdf

Bettini, L. (1998). *Progetto e Realizzazione di un Linguaggio di Programmazione per Codice Mobile (Design and Implementation of a Programming Language for Mobile Code)* (Unpublished master's thesis). Dip. di Sistemi e Informatica, Univ. di Firenze. (In Italian)

Bettini, L. (2003). A Java package for class and mixin mobility in a distributed setting. In *Proc. of Int. Workshop on scientific engineering of distributed Java applications (FIDJI)* (Vol. 2952, p. 12-22). Springer.

Bettini, L. (2023). A Java Testing Framework Without Reflection. In *ICSOFT* (pp. 369–376). SCITEPRESS.

Bettini, L., De Nicola, R., Ferrari, G., & Pugliese, R. (1998). Interactive Mobile Agents in X-KLAIM. In *Proc. of the Int. IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)* (p. 110-115). IEEE Computer Society Press.

Bettini, L., De Nicola, R., & Loreti, M. (2002). Software Update via Mobile Agent Based Programming. In *Proc. of ACM SAC, Special Track on Agents, Interactions, Mobility, and Systems* (pp. 32–36). ACM Press.

Bettini, L., De Nicola, R., & Pugliese, R. (2002). KLAVA: a Java package for distributed and mobile applications. *Software – Practice and Experience*, *32*(14), 1365–1394.

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

Braux, M., & Noyé, J. (2000). Towards Partially Evaluating Reflection in Java. In *Proceedings of PEPM* (pp. 2–11). ACM.

Campbell, G. A. (2018). Cognitive complexity: an overview and evaluation. In *Proceedings of the international conference on technical debt, techdebt@icse* (p. 57-58). ACM.

Campbell, G. A., & Papapetrou, P. P. (2013). *Sonarqube in action*. Manning Publications Co.

Chen, M.-H., Lyu, M. R., & Wong, W. E. (2002). Effect of

code coverage on software reliability measurement. *IEEE Transactions on reliability*, *50*(2), 165–170.
Chiba, S. (2000). Load-Time Structural Reflection in Java. In *Proceedings of ECOOP* (Vol. 1850, pp. 313–336).
Cui, M., Sun, S., Xu, H., & Zhou, Y. (2024). Is unsafe an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming. In *Proceedings of ICSE* (pp. 106:1–106:13). ACM.
Deinum, M., Rubio, D., & Long, J. (2023). Spring WebFlux. In *Spring 6 Recipes: A Problem-Solution Approach to Spring Framework* (pp. 205–240). Springer.
Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, *15*(10), 859-866.
Drechsler, R. L., & Mocenigo, J. M. (2007, May). The Yoix(r) scripting language: A different way of writing Java(tm) applications. *Software Practice and Experience*, *37*(6), 643–667.
Evans, B. (2023). *The performance implications of Java reflection.* Retrieved from https://blogs.oracle.com/javamagazine/post/java-reflection-performance (Accessed: 2025-07-18)
Feigenbaum, B. (2004). Java Reflection & Smalltalk-like Method Dispatching. *Dr. Dobb's Journal of Software Tools*, *29*(7), 42–45.
Forax, R., Duris, É., & Roussel, G. (2005). Reflection-based implementation of Java extensions: the double-dispatch use-case. *Journal of Object Technology*, *4*(10), 49–69.
Gamma, E., & Beck, K. (1999). Junit: A cook's tour. *Java report*, *4*(5), 27–38.
Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.
Gonzalez, D., Santos, J. C., Popovich, A., Mirakhorli, M., & Nagappan, M. (2017). A Large-Scale Study on the Usage of Testing Patterns That Address Maintainability Attributes: Patterns for Ease of Modification, Diagnoses, and Comprehension. In *International Conference on Mining Software Repositories (MSR)* (p. 391-401). IEEE.
Hemmati, H. (2015). How effective are code coverage criteria? In *International conference on software quality, reliability and security* (pp. 151–156). USA: IEEE.
Hoffer, M., Poliwoda, C., & Wittum, G. (2013). Visual reflection library: a framework for declarative GUI programming on the Java platform. *Computing and Visualization in Science*, *16*(4), 181–192.
Hunt, J. (2014). Scala Testing. In *A Beginner's Guide to Scala, Object Orientation and Functional Programming* (pp. 365–382). Springer.
Igarashi, A., Pierce, B., & Wadler, P. (2001, May). Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, *23*(3), 396–450.
Islam, A., Hewage, N. T., Bangash, A. A., & Hindle, A. (2023). Evolution of the practice of software testing in Java projects. In *International Conference on Mining Software Repositories (MSR)* (pp. 367–371). IEEE.
Ivanković, M., Petrović, G., Just, R., & Fraser, G. (2019). Code coverage at Google. In *Esec/fse* (p. 955–963). USA: ACM.
Jung, R., Jourdan, J., Krebbers, R., & Dreyer, D. (2018). RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, *2*(POPL), 66:1–66:34.
Karaorman, M., Holzle, U., & Bruno, J. (1999, July). jContractor: A reflective Java library to support design by contract. In *Proceedings of Reflection* (Vol. 1616, pp. 175–196). Springer.
Kirby, G. N. C., Morrison, R., & Stemple, D. W. (1998). Linguistic Reflection in Java. *Software – Practice & Experience*, *28*(10), 1045–1077.
Klabnik, S., Nichols, C., & Krycho, C. (2024). *The Rust Programming Language* (3rd ed.). No Starch Press.
Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *IEEE Softw.*, *29*(6), 18–21.
Lam, W., Srisakaokul, S., Bassett, B., Mahdian, P., Xie, T., Lakshman, P., & De Halleux, J. (2018). A Characteristic Study of Parameterized Unit Tests in. NET Open Source Projects. In *Proceedings of ECOOP* (Vol. 109, pp. 5:1–5:27). Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
Landman, D., Serebrenik, A., & Vinju, J. J. (2017). Challenges for static analysis of Java reflection: literature review and empirical study. In *Proceedings of ICSE* (pp. 507–518). IEEE / ACM.
Lenarduzzi, V., Pecorelli, F., Saarimäki, N., Lujan, S., & Palomba, F. (2023). A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software*, *198*, 111575.
Lenarduzzi, V., Saarimäki, N., & Taibi, D. (2020). Some SonarQube issues have a significant but small effect on faults and changes: A large-scale empirical study. *Journal of Systems and Software*, *170*, 110750.
Leotta, M., Cerioli, M., Olianas, D., & Ricca, F. (2020). Two experiments for evaluating the impact of Hamcrest and AssertJ on assertion development. *Software Quality Journal*, *28*(3), 1113–1145.
Letouzey, J.-L., & Ilkiewicz, M. (2012). Managing Technical Debt with the SQALE Method. *IEEE Software*, *29*(6), 44–51.
Li, Y., Tan, T., & Xue, J. (2019, April). Understanding and Analyzing Java Reflection. *ACM Transactions on Software Engineering and Methodology*, *28*(2), 7:1–7:50.
Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, *101*, 193–220.
Livshits, V. B., Whaley, J., & Lam, M. S. (2005). Reflection Analysis for Java. In *Proceedings of APLAS* (Vol. 3780, pp. 139–160). Springer.
Louridas, P. (2005). JUnit: unit testing and coiling in tandem. *IEEE Software*, *22*(4), 12-15.
Ma'ayan, D. D. (2018). The Quality of Junit Tests: An Empirical Study Report. In *International Workshop on Software Qualities and their Dependencies (SQUADE)* (p. 33-36). IEEE.
Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W., & Pinto, G. (2019). Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube. In *International Conference on Program Comprehension (ICPC)* (pp. 209–219). IEEE.

Marick, B., Smith, J., & Jones, M. (1999). How to misuse code coverage. In *Interational conference on testing computer software* (pp. 16–18).
Marri, M. R., Xie, T., Tillmann, N., De Halleux, J., & Schulte, W. (2009). An empirical study of testing file-system-dependent software with mock objects. In *Workshop on Automation of Software Test* (pp. 149–153). IEEE.
McCabe, T. J. (1976). A complexity measure. *IEEE Trans. Software Eng.*, *2*(4), 308–320.
Morris, D. S. (2002). Automatically grading Java programming assignments via reflection, inheritance, and regular expressions. In *Frontiers in Education Conference.* IEEE.
Mostafa, S., & Wang, X. (2014). An empirical study on the usage of mocking frameworks in software testing. In *International conference on quality software* (pp. 127–132). IEEE.
Myers, G. J. (1979). *The art of software testing*. John Wiley & Sons.
Oracle. (2024). *The Reflection API – The Java Tutorials.* https://docs.oracle.com/javase/tutorial/reflect. (Accessed: 2026-01-09)
Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., & Harman, M. (2019). Mutation testing advances: an analysis and survey. In *Advances in computers* (Vol. 112, pp. 275–378). USA: Elsevier.
Park, J. G., & Lee, A. H. (2001). Removing Reflection from Java Programs Using Partial Evaluation. In *Proceedings of Reflection* (Vol. 2192, pp. 274–275). Springer.
Parson, D. (1999, October 3–5). Using Java Reflection to Automate Extension Language Parsing. In *Proceedings of DSL* (pp. 67–80). USENIX Association.
Prasanna, D. R. (2009). *Dependency Injection: Design Patterns Using Spring and Guice* (1st ed.). Manning.
Rakshith, D., & Manjunath, A. (2020). A Comprehensive Study on Automation Testing using JUnit. *International Research Journal of Engineering and Technology*, *7*(7), 2388–2392.
Smaragdakis, Y., Balatsouras, G., Kastrinis, G., & Bravenboer, M. (2015). More Sound Static Handling of Java Reflection. In *Proceedings of APLAS* (Vol. 9458, pp. 485–503). Springer.
Sobernig, S., & Zdun, U. (2010). Evaluating Java runtime reflection for implementing cross-language method invocations. In *Proceedings of PPPJ* (pp. 139–147). ACM.
Thies, A., & Bodden, E. (2012). RefaFlex: Safer Refactorings for Reflective Java Programs. In *Proceedings of ISSTA* (pp. 1–11). ACM.
Tillmann, N., de Halleux, J., & Xie, T. (2010). Parameterized unit testing: theory and practice. In *Proceedings of ICSE* (p. 483–484). ACM.
Tillmann, N., & Schulte, W. (2005). Parameterized unit tests. In *Proceedings of ESEC/FSE* (p. 253–262). ACM.
Tom, E., Aurum, A., & Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, *86*(6), 1498–1516.
Tudose, C. (2020). *JUnit in Action* (3rd ed.). Manning Publications.
Tudose, C., Odubăşteanu, C., & Radu, S. (2013). Java Reflection Performance Analysis Using Different Java Development. In *Advances in Intelligent Control Systems and Computer Science* (pp. 439–452). Springer.
Welch, I., & Stroud, R. (1999). From Dalang to Kava – the Evolution of a Reflective Java Extension. In *Proceedings of Reflection* (Vol. 1616, pp. 2–21). Springer.
Welch, I., & Stroud, R. J. (2001). Kava — Using Byte Code Rewriting to Add Behavioural Reflection to Java. In *Proceedings of the Conference on Object-Oriented Technologies and Systems* (pp. 119–130). USENIX Association.
Winter, V. L., Reinke, C., & Guerrero, J. (2016). Certifying a Java type resolution function using program transformation, annotation, and reflection. *Software Quality Journal*, *24*(1), 115–135.
Woodward, M. R. (1993). Mutation testing—its origin and evolution. *Information and Software Technology*, *35*(3), 163–169.
Zhang, Y., Kundu, A., Portokalidis, G., & Xu, J. (2023). On the Dual Nature of Necessity in Use of Rust Unsafe Code. In *Proceedings of ESEC/FSE* (pp. 2032–2037). ACM.
Zheng, X., Wan, Z., Zhang, Y., Chang, R., & Lo, D. (2024). A Closer Look at the Security Risks in the Rust Ecosystem. *ACM Trans. Softw. Eng. Methodol.*, *33*(2), 34:1–34:30.

APPENDIX

In this appendix, we show the relevant parts of the implementation of the JNRTEST runner, in particular, how it implements the lifecycle for running test classes and tests.

The first parts of the implementation of the runner are shown in Figure 25. For each test class, we access its store (which, in case, gets initialized), and we execute its lifecycle (method `execute`). We apply possible filters to the test classes (method `getTestClassesStream`). Note that this method is meant to be overridden to process test classes in a different way, for example, to run tests in parallel (as we showed in Section 5.4).

The other parts of the lifecycle implementation are shown in Figure 26. During the execution, we notify the listeners about events. In particular, `executeSafely` is also responsible for establishing the result of a runnable specification (“success”, “failure”, “error”). We rely on assertion methods (JUnit or AssertJ) to throw the Java exception `AssertionError` when an assertion fails. Listeners are notified of results as well. The result also contains the caught exception in case of failures and errors.

About the authors

Lorenzo Bettini is an Associate Professor in Computer Science at DISIA Dipartimento di Statistica, Informatica, Applicazioni ‘Giuseppe Parenti’, Università di Firenze, Italy, since February 2016. Previously, he was an Assistant Professor (Researcher) in Computer Science at Dipartimento di Informatica, Università di Torino, Italy. His research interests cover design, theory, and implementation of programming languages (in particular Object-Oriented languages and Network-aware languages) with IDE support. You can contact the author at lorenzo.bettini@unifi.it or visit https://www.lorenzobettini.it.

```java
public void execute() {
  getTestClassesStream().forEach(this::executeTestClass);
}

protected Stream<JnrTest> getTestClassesStream() {
  Predicate<JnrTest> classFilter = filters.getClassFilter();
  if (classFilter == null) {
    return testClasses.stream();
  }
  return testClasses.stream()
      .filter(classFilter);
}

private void executeTestClass(JnrTest testClass) {
  var description = testClass.getDescription();
  notifyTestLifecycleEvent(new JnrTestLifecycleEvent(description, JnrTestStatus.START));
  executeTestClass(testClass.getStore());
  notifyTestLifecycleEvent(new JnrTestLifecycleEvent(description, JnrTestStatus.END));
}

private void executeTestClass(JnrTestStore store) {
  executeBeforeAll(store);
  executeTestRunnables(store);
  executeAfterAll(store);
}

private void executeBeforeAll(JnrTestStore store) {
  executeLifecycleRunnables(store.getBeforeAllRunnables(), JnrTestRunnableKind.BEFORE_ALL);
}

// similar methods for afterAll, beforeEach, afterEach
```

Figure 25 The first parts of the implementation of the lifecycle.

```java
private void executeLifecycleRunnables(List<JnrTestRunnableSpecification> runnables,
    JnrTestRunnableKind kind) {
  for (var runnable : runnables) {
    executeSafely(runnable, kind, null);
  }
}

private void executeTestRunnables(JnrTestStore store) {
  List<JnrTestRunnableSpecification> runnablesToExecute;
  Predicate<JnrTestRunnableSpecification> specFilter = filters.getSpecificationFilter();
  if (specFilter == null) {
    runnablesToExecute = store.getRunnableSpecifications();
  } else {
    runnablesToExecute = store.getRunnableSpecifications().stream()
        .filter(specFilter)
        .toList();
  }

  for (var runnableSpecification : runnablesToExecute) {
    executeBeforeEach(store);
    executeSafely(runnableSpecification, JnrTestRunnableKind.TEST,
        d -> notifyTestResult(new JnrTestResult(d, JnrTestResultStatus.SUCCESS, null)));
    executeAfterEach(store);
  }
}

private void executeSafely(JnrTestRunnableSpecification testRunnableSpecification,
    JnrTestRunnableKind kind,
    Consumer<String> successConsumer) {
  var description = testRunnableSpecification.description();
  var testRunnable = testRunnableSpecification.testRunnable();
  try {
    executeSafely(testRunnable, kind, description);
    if (successConsumer != null) {
      successConsumer.accept(description);
    }
  } catch (Exception e) {
    notifyTestResult(new JnrTestResult(description, JnrTestResultStatus.ERROR, e));
  } catch (AssertionError assertionError) {
    notifyTestResult(new JnrTestResult(description, JnrTestResultStatus.FAILED, assertionError));
  }
}

private void executeSafely(JnrTestRunnable testRunnable, JnrTestRunnableKind kind, String description)
     throws Exception {
  try {
    notifyTestRunnableLifecycleEvent(
        new JnrTestRunnableLifecycleEvent(description, kind, JnrTestRunnableStatus.START));
    testRunnable.run();
  } finally {
    notifyTestRunnableLifecycleEvent(
        new JnrTestRunnableLifecycleEvent(description, kind, JnrTestRunnableStatus.END));
  }
}
```

Figure 26 The other parts of the implementation of the lifecycle.