CSCI5308
Adv Topics in Software Development

ASSIGNMENT - 1

Banner ID: B00980674

GitLab Assignment Link: https://github.com/OvaizAli/metrics-ASDC-A1

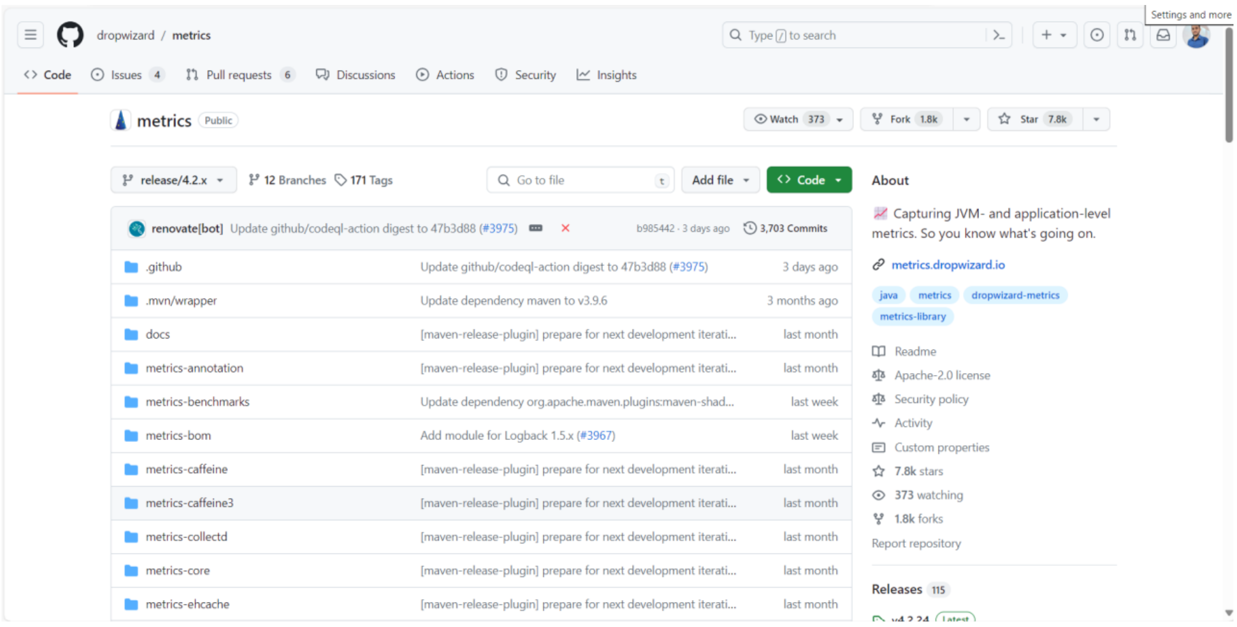
Table of Contents

Task 1

- Java-based open-source repository satisfying the given conditions.
  - Repository Link [1] : https://github.com/dropwizard/metrics

Table 1: Table indicating that all the Task 1 conditions are fulfilled.

S No	Given Condition	Condition Fulfilled	Justification
1	Must be a Maven or Gradle-based project	Yes	Repository is Maven based.
2	Must have at least 10,000 lines of code	Yes	It has around 50,000 lines of code.
3	Must have at least 50 stars	Yes	Repository has 7.8K stars.
4	Must have tests written using the JUnit framework	Yes	Test cases are written using Junit.
5	Must not be a tutorial or example repository	Yes	This repository is an industry standard opensource repository.
6	Must be active (at least one commit in the past one year).	Yes	The repository had commit in the last week.



dropwizard / metrics

Type / to search

Settings and more

Code · Issues 4 · Pull requests 6 · Discussions · Actions · Security · Insights

metrics Public

Watch 373 · Fork 1.8k · Star 7.8k

release/4.2.x · 12 Branches · 171 Tags · Go to file · Add file · Code

renovate[bot] Update github/codeql-action digest to 47b3d88 (#3975)	b985442 · 3 days ago	3,703 Commits
.github	Update github/codeql-action digest to 47b3d88 (#3975)	3 days ago
.mvn/wrapper	Update dependency maven to v3.9.6	3 months ago
docs	[maven-release-plugin] prepare for next development iterati...	last month
metrics-annotation	[maven-release-plugin] prepare for next development iterati...	last month
metrics-benchmarks	Update dependency org.apache.maven.plugins:maven-shad...	last week
metrics-bom	Add module for Logback 1.5.x (#3967)	last week
metrics-caffeine	[maven-release-plugin] prepare for next development iterati...	last month
metrics-caffeine3	[maven-release-plugin] prepare for next development iterati...	last month
metrics-collectd	[maven-release-plugin] prepare for next development iterati...	last month
metrics-core	[maven-release-plugin] prepare for next development iterati...	last month
metrics-ehcache	[maven-release-plugin] prepare for next development iterati...	last month

About

Capturing JVM- and application-level metrics. So you know what's going on.

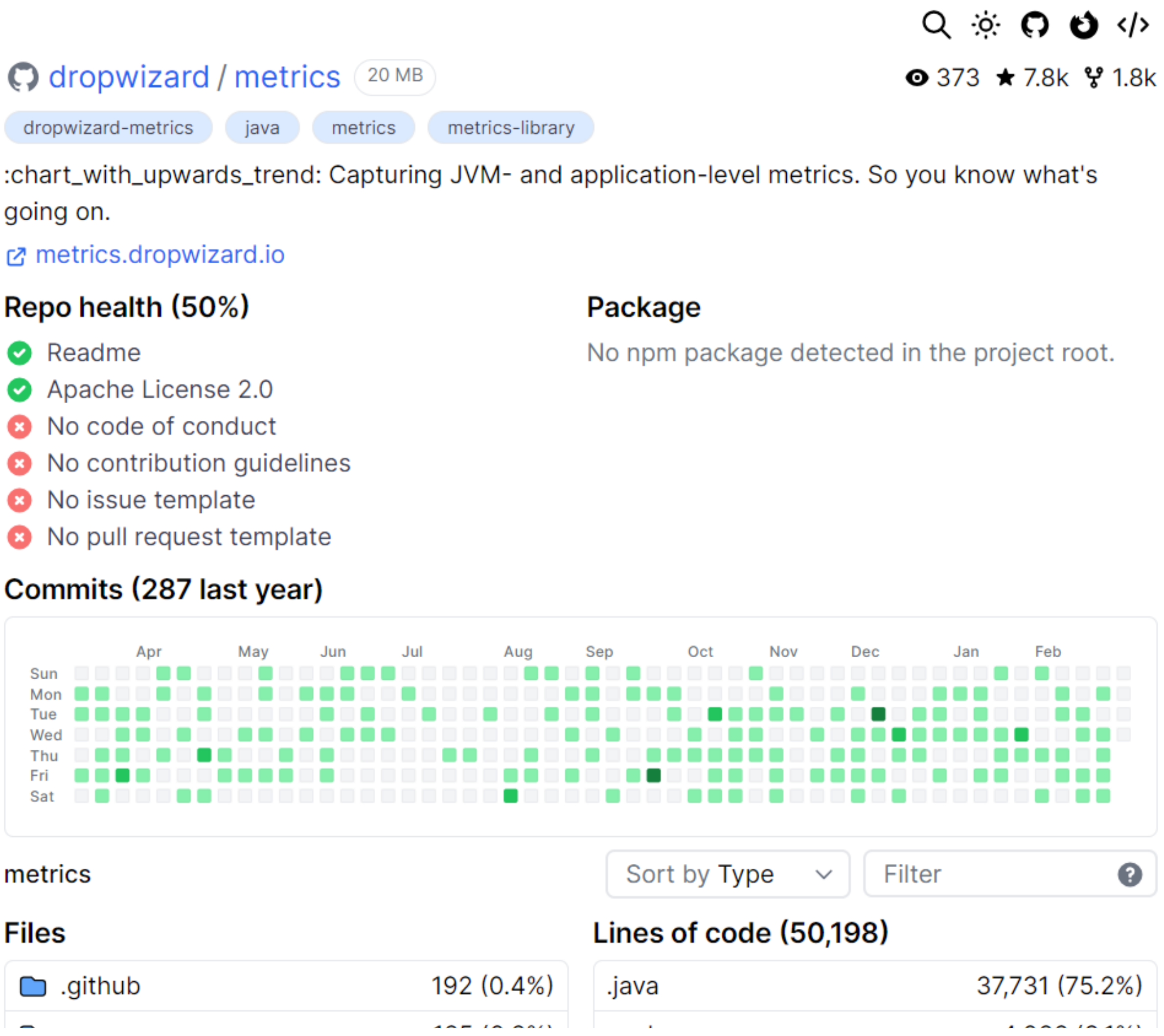
metrics.dropwizard.io

java · metrics · dropwizard-metrics · metrics-library

Readme · Apache-2.0 license · Security policy · Activity · Custom properties · 7.8k stars · 373 watching · 1.8k forks · Report repository

Releases 115

Figure 1: Screenshot of the actual repository.



dropwizard / metrics 20 MB

373 ★ 7.8k 1.8k

dropwizard-metrics · java · metrics · metrics-library

:chart_with_upwards_trend: Capturing JVM- and application-level metrics. So you know what's going on.

metrics.dropwizard.io

Repo health (50%)

- ✓ Readme
- ✓ Apache License 2.0
- ✗ No code of conduct
- ✗ No contribution guidelines
- ✗ No issue template
- ✗ No pull request template

Package

No npm package detected in the project root.

Commits (287 last year)

metrics

Sort by Type · Filter

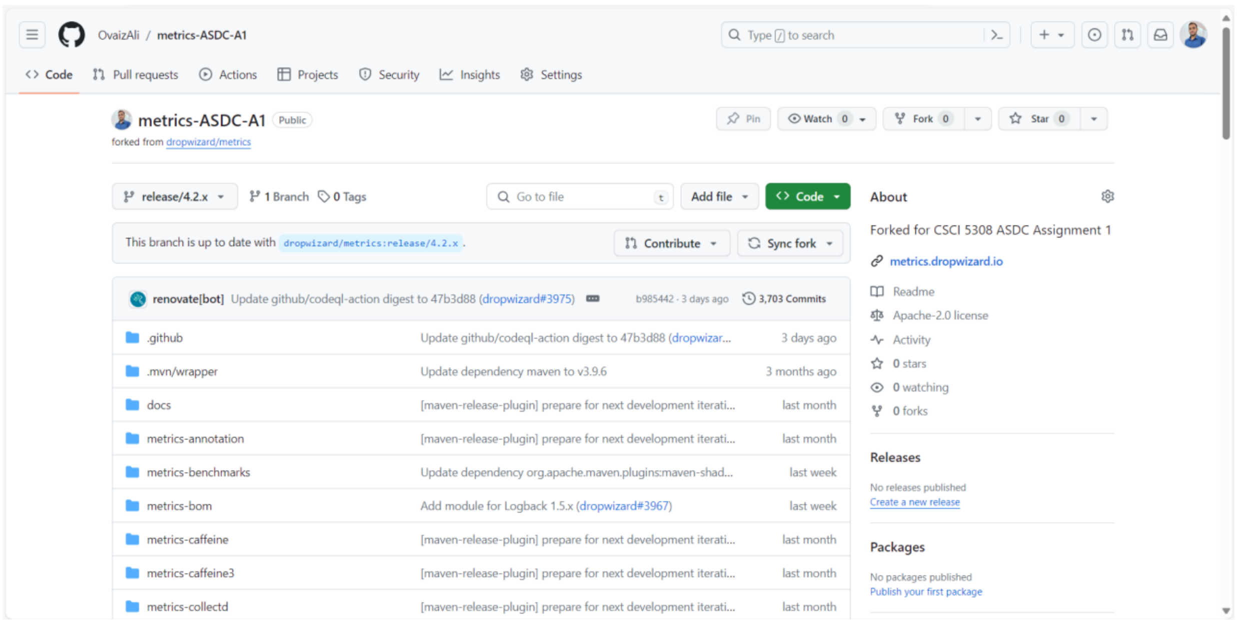
Files

.github	192 (0.4%)

Lines of code (50,198)

.java	37,731 (75.2%)

Figure 2: Screenshot for the evidence of the stated facts about the repository. [2]



OvaizAli / metrics-ASDC-A1

metrics-ASDC-A1 Public

forked from dropwizard/metrics

release/4.2.x · 1 Branch · 0 Tags

This branch is up to date with `dropwizard/metrics:release/4.2.x`.

renovate[bot] Update github/codeql-action digest to 47b3d88 (dropwizard#3975) · b985442 · 3 days ago · 3,703 Commits

.github	Update github/codeql-action digest to 47b3d88 (dropwizar...	3 days ago
.mvn/wrapper	Update dependency maven to v3.9.6	3 months ago
docs	[maven-release-plugin] prepare for next development iterati...	last month
metrics-annotation	[maven-release-plugin] prepare for next development iterati...	last month
metrics-benchmarks	Update dependency org.apache.maven.plugins:maven-shad...	last week
metrics-bom	Add module for Logback 1.5.x (dropwizard#3967)	last week
metrics-caffeine	[maven-release-plugin] prepare for next development iterati...	last month
metrics-caffeine3	[maven-release-plugin] prepare for next development iterati...	last month
metrics-collectd	[maven-release-plugin] prepare for next development iterati...	last month

About

Forked for CSCI 5308 ASDC Assignment 1

metrics.dropwizard.io

Readme
Apache-2.0 license
Activity
0 stars
0 watching
0 forks

Releases

No releases published
Create a new release

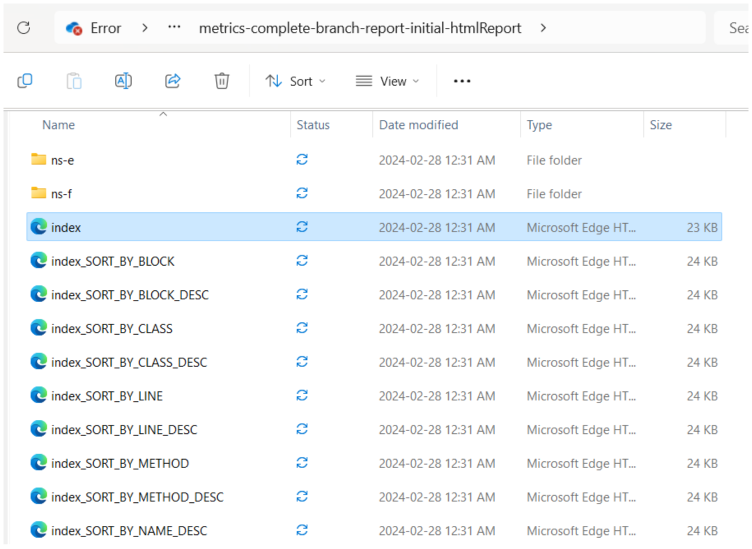
Packages

No packages published
Publish your first package

Figure 3: Screenshot indicating that the repository was forked for further tasks.

Task 2

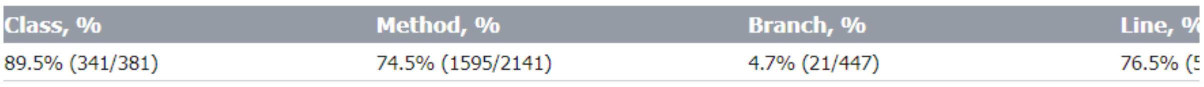
Provide quantitative measures of test implementation. Specifically, provide the total number of automated tests and code coverage (branch). You may use IDE features, IDE plugins, or external tools to obtain these numbers. Please clearly mention the used tool/mechanism.



Error > ··· metrics-complete-branch-report-initial-htmlReport >

Name	Status	Date modified	Type	Size
ns-e		2024-02-28 12:31 AM	File folder	
ns-f		2024-02-28 12:31 AM	File folder	
index		2024-02-28 12:31 AM	Microsoft Edge HT...	23 KB
index_SORT_BY_BLOCK		2024-02-28 12:31 AM	Microsoft Edge HT...	24 KB
index_SORT_BY_BLOCK_DESC		2024-02-28 12:31 AM	Microsoft Edge HT...	24 KB
index_SORT_BY_CLASS		2024-02-28 12:31 AM	Microsoft Edge HT...	24 KB
index_SORT_BY_CLASS_DESC		2024-02-28 12:31 AM	Microsoft Edge HT...	24 KB
index_SORT_BY_LINE		2024-02-28 12:31 AM	Microsoft Edge HT...	24 KB
index_SORT_BY_LINE_DESC		2024-02-28 12:31 AM	Microsoft Edge HT...	24 KB
index_SORT_BY_METHOD		2024-02-28 12:31 AM	Microsoft Edge HT...	24 KB
index_SORT_BY_METHOD_DESC		2024-02-28 12:31 AM	Microsoft Edge HT...	24 KB
index_SORT_BY_NAME_DESC		2024-02-28 12:31 AM	Microsoft Edge HT...	24 KB

Figure 4: Test cases reports gathered in one folder.



Class, %	Method, %	Branch, %	Line, %
89.5% (341/381)	74.5% (1595/2141)	4.7% (21/447)	76.5% (5

Figure 5: Statistics regarding the test coverage of the complete repository.

- ✓ Quantitative Measures of Test Implementation:

    - All **422 automated unit tests** were written using JUnit. Automated tests, including those written with JUnit, are executed automatically by a testing framework or tool without manual intervention, making them suitable for continuous integration and ensuring consistent and repeatable testing processes.

- ✓ We can estimate the following based on the coverage report:

    - **Class coverage:** 341 out of 381 classes were covered by all test cases.
    - **Method coverage:** 1595 out of 2141 methods were covered by all test cases.
    - **Branch coverage:** 21 out of 447 branches are covered by tests.
    - **Line coverage:** Initially 76.5% of lines were covered by the tests.
    - **Tool/Mechanism Used:** IntelliJ IDEA coverage runner is used for extracting the quantitative measures and gathering it in a folder. Since the repository I selected had multiple submodules in it, running tests to find the coverage for all the submodules collectively was difficult. However, through lots of trial and error, making configurations with regular expressions, and finally using a manual approach to test each module separately and then merging it into the test suite worked for me. Hence, this was a bit challenging yet learning part for me.

- ✓ Analysis of the coverage report:

    - Overall method coverage (74.5%) is relatively high, indicating a significant portion of the code's functions are tested.
    - However, there are variations in coverage across packages. Some packages like com.codahale.metrics.caffeine and com.codahale.metrics.health have near-perfect coverage, while others like io.dropwizard.metrics.httpasyncclient and com.codahale.metrics.benchmarks have significantly lower coverage. This suggests that testing efforts might be concentrated in specific areas, and some functionalities lack sufficient test coverage.

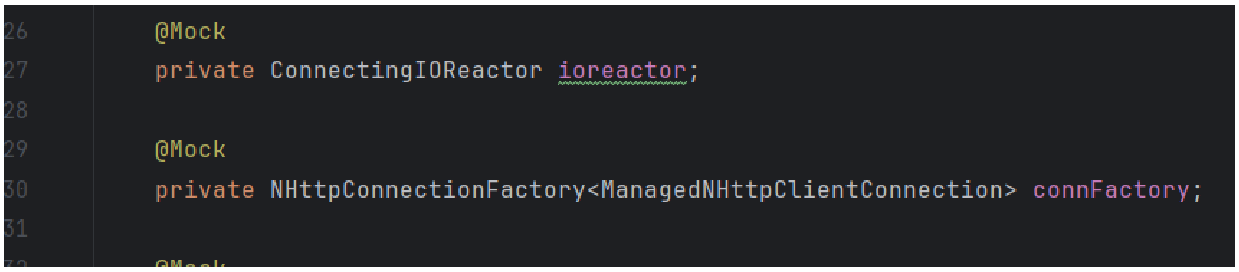
Task 3

Critique the test implementation. Provide at least three strong and weak aspects of the test implementation. If you do not find any strong (or weak) aspect, you may have six weak (or strong) aspects in your answer.

Each aspect must be elaborated using a paragraph (4-6 sentences) and at least an example.

✓ Strong Aspects:

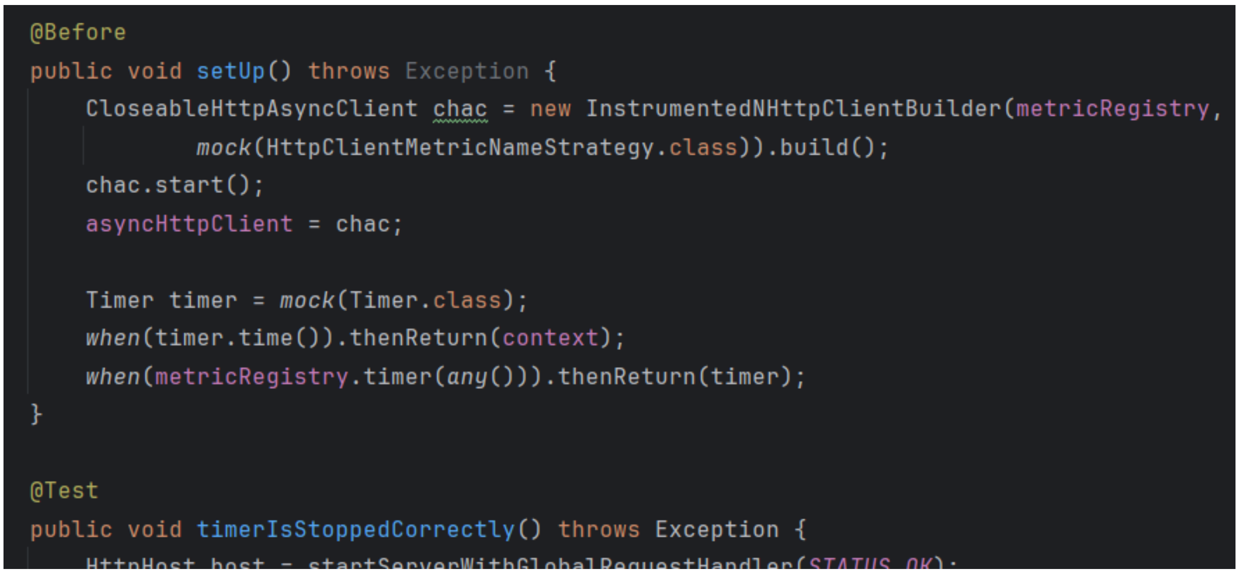
- The test cases implementation **effectively mocks dependencies** like in the InstrumentedNClientConnManager class ConnectingIOReactor and NHttpConnectionFactory (lines 27 and 30, respectively) using Mockito. By isolating the behavior of the InstrumentedNClientConnManager class from its dependencies, the tests focus solely on the logic within the class itself. This approach ensures that any changes in the implementations of dependencies do not affect the outcome of the tests, thus promoting maintainability and stability.



```
26        @Mock
27        private ConnectingIOReactor ioreactor;
28
29        @Mock
30        private NHttpConnectionFactory<ManagedNHttpClientConnection> connFactory;
31
32        @Mock
```

Figure 6: Screenshot depicting the 1st strong aspect.

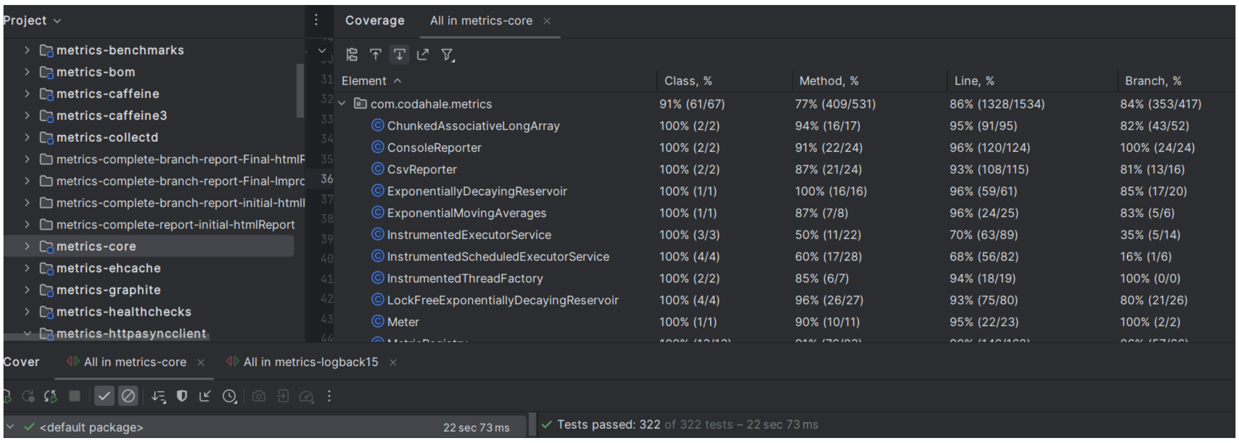
- The **test setup in all test classes is clear, readable, and consistent**. Test methods and variables **follow naming conventions**, enhancing readability and maintainability. For example, each dependency is properly initialized and passed to the constructor of the InstrumentedNHttpClientBuilder within the setup annotated with @Before, contributing to the clarity and organization of the tests. For instance, the method name timerIsStoppedCorrectly succinctly conveys the purpose of the test, facilitating immediate understanding for developers.



```
@Before
public void setUp() throws Exception {
    CloseableHttpAsyncClient chac = new InstrumentedNHttpClientBuilder(metricRegistry,
            mock(HttpClientMetricNameStrategy.class)).build();
    chac.start();
    asyncHttpClient = chac;

    Timer timer = mock(Timer.class);
    when(timer.time()).thenReturn(context);
    when(metricRegistry.timer(any())).thenReturn(timer);
}

@Test
public void timerIsStoppedCorrectly() throws Exception {
    HttpHost host = startServerWithGlobalRequestHandler(STATUS_OK);
```

Figure 7: Screenshot depicting the 2nd strong aspect.

- All test cases exhibit **excellent test coverage being tested in isolation**. They thoroughly explore various scenarios related to metrics, including cache hits, misses, successful and failed loads, and eviction with different causes making sure every branch, method, class, and maximum lines are tested. This comprehensive coverage extends to MetricsStatsCounter, Gauge, meter, and metrics core class functionality, as demonstrated by the well-defined test cases within the metrics-core submodule. (refer below image)

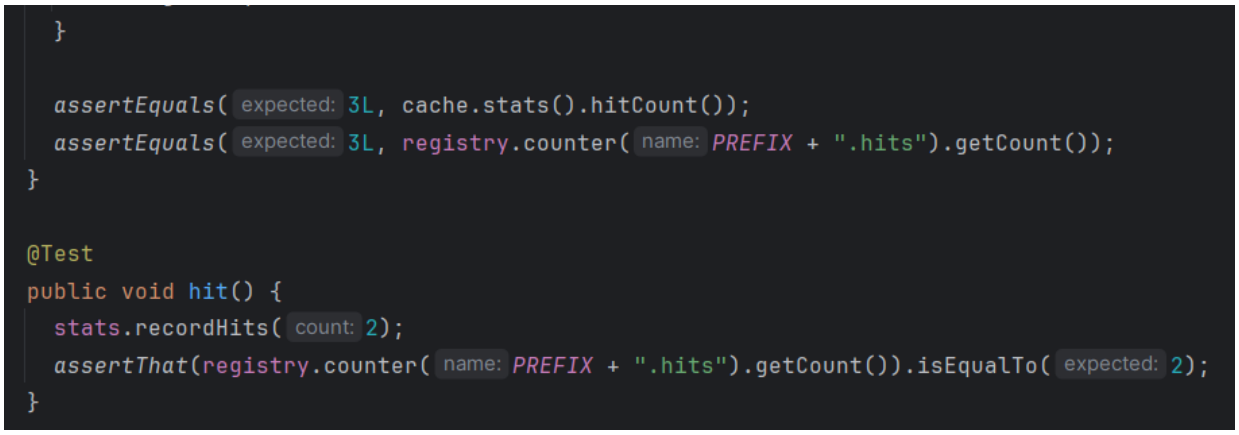


Element	Class, %	Method, %	Line, %	Branch, %
com.codahale.metrics	91% (61/67)	77% (409/531)	86% (1328/1534)	84% (353/417)
ChunkedAssociativeLongArray	100% (2/2)	94% (16/17)	95% (91/95)	82% (43/52)
ConsoleReporter	100% (2/2)	91% (22/24)	96% (120/124)	100% (24/24)
CsvReporter	100% (2/2)	87% (21/24)	93% (108/115)	81% (13/16)
ExponentiallyDecayingReservoir	100% (1/1)	100% (16/16)	96% (59/61)	85% (17/20)
ExponentialMovingAverages	100% (1/1)	87% (7/8)	96% (24/25)	83% (5/6)
InstrumentedExecutorService	100% (3/3)	50% (11/22)	70% (63/89)	35% (5/14)
InstrumentedScheduledExecutorService	100% (4/4)	60% (17/28)	68% (56/82)	16% (1/6)
InstrumentedThreadFactory	100% (2/2)	85% (6/7)	94% (18/19)	100% (0/0)
LockFreeExponentiallyDecayingReservoir	100% (4/4)	96% (26/27)	93% (75/80)	80% (21/26)
Meter	100% (1/1)	90% (10/11)	95% (22/23)	100% (2/2)

```
<default package>    22 sec 73 ms
Tests passed: 322 of 322 tests – 22 sec 73 ms
```

Figure 8: Screenshot depicting the 3rd strong aspect.

- The test cases demonstrate comprehensive **assertion clarity**. Assertions such as assertEquals, assertThat, and assertThrow in the MetricsStatsCounterTest class, as well as throughout the module, enhance readability and expressiveness. These assertions effectively communicate the expected behavior and outcomes of the tests, ensuring rigorous validation of the system's functionality. Hence, this **prevents Assertion Roulette** test smell.

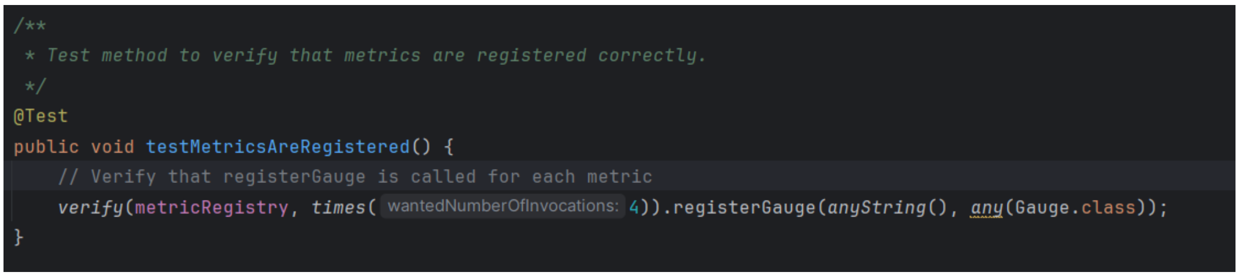


```
    }

    assertEquals( expected: 3L, cache.stats().hitCount());
    assertEquals( expected: 3L, registry.counter( name: PREFIX + ".hits").getCount());
}

@Test
public void hit() {
    stats.recordHits( count: 2);
    assertThat(registry.counter( name: PREFIX + ".hits").getCount()).isEqualTo( expected: 2);
}
```

Figure 9: Screenshot depicting the 4th strong aspect.

✓ Weak Aspects:

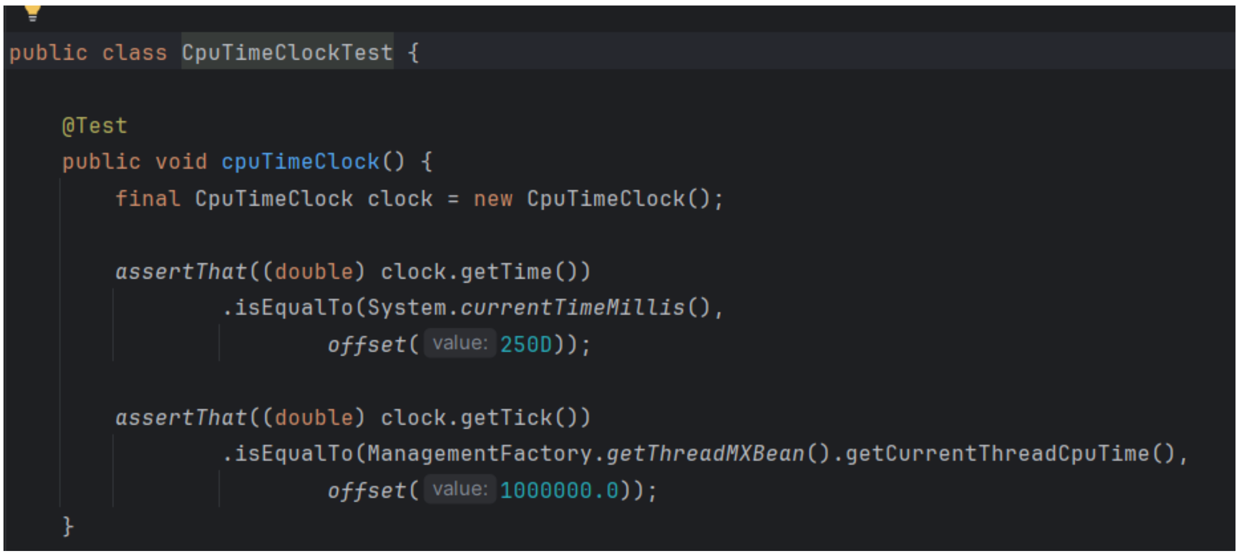
- The tests **implicitly rely on the behavior** of the superclass, as seen in the MetricRegistry implementation where they depend on the correct implementation of the registerGauge method. While Mockito enables mocking dependencies, it does not verify the correctness of the behavior of the mocked objects. Thus, if issues arise with the MetricRegistry implementation or if the registerGauge method behaves unexpectedly, the tests may pass erroneously. To mitigate this risk, integration tests or additional verification mechanisms should be employed to ensure the correctness of the metric registration process.



```java
/**
 * Test method to verify that metrics are registered correctly.
 */
@Test
public void testMetricsAreRegistered() {
    // Verify that registerGauge is called for each metric
    verify(metricRegistry, times( wantedNumberOfInvocations: 4)).registerGauge(anyString(), any(Gauge.class));
}
```

Figure 10: Screenshot depicting the 1st weak aspect.

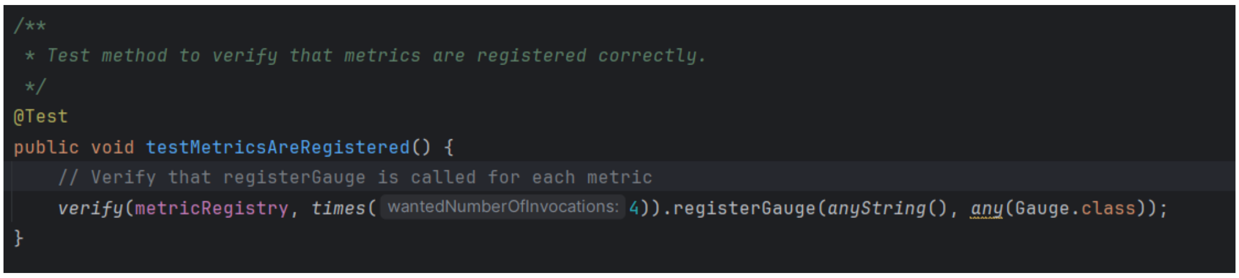
- Additionally, the test cases **make implicit assumptions**, as evident in the ClockTest class where they assume the accuracy and consistency of System.currentTimeMillis() and System.nanoTime(). However, this assumption may not always hold true in certain environments or under specific conditions. Explicitly addressing these assumptions or employing more robust testing strategies would enhance the reliability of the tests.



```java
public class CpuTimeClockTest {

    @Test
    public void cpuTimeClock() {
        final CpuTimeClock clock = new CpuTimeClock();

        assertThat((double) clock.getTime())
                .isEqualTo(System.currentTimeMillis(),
                        offset( value: 250D));

        assertThat((double) clock.getTick())
                .isEqualTo(ManagementFactory.getThreadMXBean().getCurrentThreadCpuTime(),
                        offset( value: 1000000.0));
    }
```

Figure 11: Screenshot depicting the 2nd weak aspect.

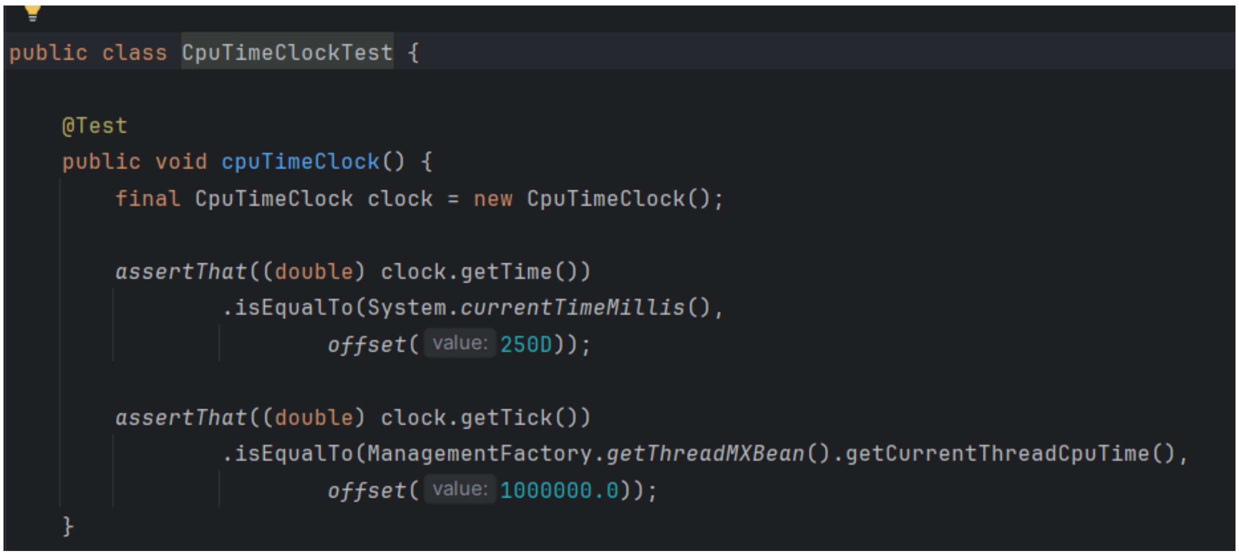
- The test method **lacks coverage for negative scenarios**. For example, the testMetricsAreRegistered method focuses solely on validating the positive scenario where metrics are successfully registered. However, it fails to encompass negative testing to verify how the InstrumentedNClientConnManager behaves in exceptional cases, such as when the MetricRegistry is null or when metric registration fails. Integrating negative test cases would enhance the test suite's effectiveness by validating the class's behavior under adverse conditions, thereby improving its overall reliability and resilience.



```java
/**
 * Test method to verify that metrics are registered correctly.
 */
@Test
public void testMetricsAreRegistered() {
    // Verify that registerGauge is called for each metric
    verify(metricRegistry, times( wantedNumberOfInvocations: 4)).registerGauge(anyString(), any(Gauge.class));
}
```

Figure 12: Screenshot depicting the 3rd weak aspect.

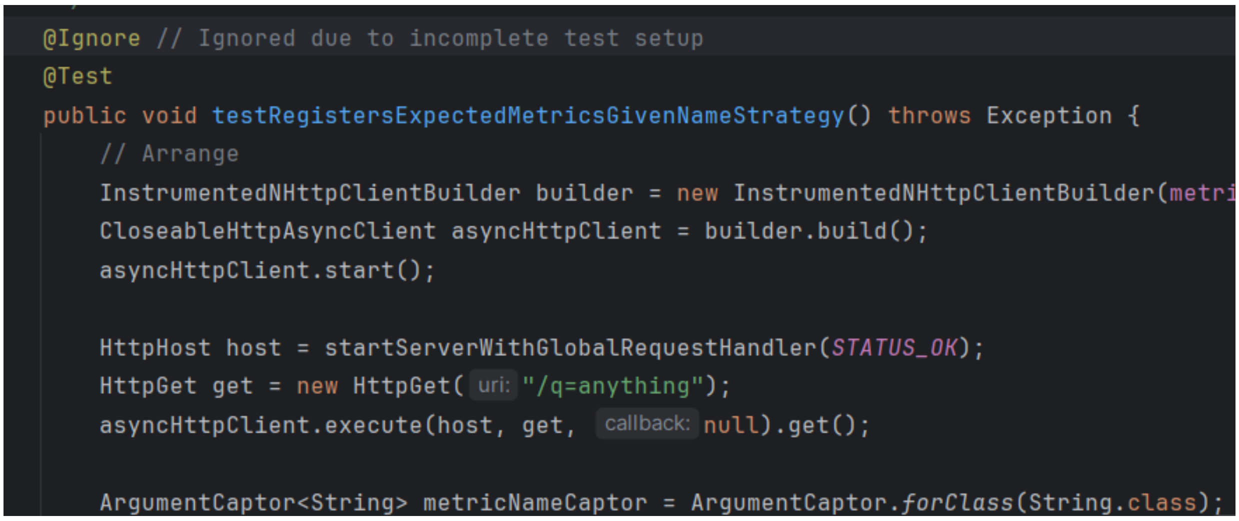
- **Magic numbers** are used in the ClockTest class, such as 100.0 and 1000000.0, as offsets in the isEqualTo assertions. Without proper documentation or context explaining their significance, these numbers could be confusing. Consider replacing them with named constants or providing explanatory comments to improve code clarity and maintainability.



```java
public class CpuTimeClockTest {

    @Test
    public void cpuTimeClock() {
        final CpuTimeClock clock = new CpuTimeClock();

        assertThat((double) clock.getTime())
                .isEqualTo(System.currentTimeMillis(),
                        offset( value: 250D));

        assertThat((double) clock.getTick())
                .isEqualTo(ManagementFactory.getThreadMXBean().getCurrentThreadCpuTime(),
                        offset( value: 1000000.0));
    }
```

Figure 13: Screenshot depicting the 4th weak aspect.

- Lastly, the **presence of @Ignore annotations** in the InstrumentedHttpClientsTimerTest class indicates a potential issue with flaky tests. Ignoring tests without addressing the underlying causes can lead to overlooked defects and reduced confidence in the test suite. It's crucial to investigate and address the flakiness to maintain the reliability and effectiveness of the tests.



```
@Ignore // Ignored due to incomplete test setup
@Test
public void testRegistersExpectedMetricsGivenNameStrategy() throws Exception {
    // Arrange
    InstrumentedNHttpClientBuilder builder = new InstrumentedNHttpClientBuilder(metri
    CloseableHttpAsyncClient asyncHttpClient = builder.build();
    asyncHttpClient.start();

    HttpHost host = startServerWithGlobalRequestHandler(STATUS_OK);
    HttpGet get = new HttpGet( uri: "/q=anything");
    asyncHttpClient.execute(host, get,  callback: null).get();

    ArgumentCaptor<String> metricNameCaptor = ArgumentCaptor.forClass(String.class);
```

Figure 14: Screenshot depicting the 5th weak aspect.

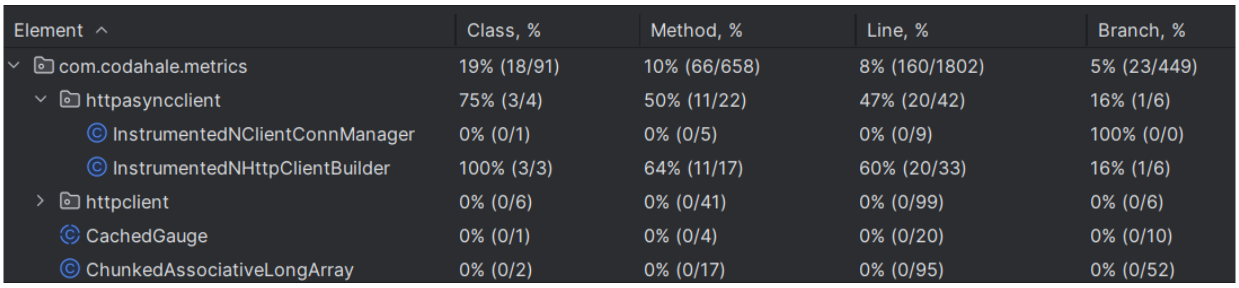
Task 4

Implement at least three new tests for the repository. It could be for new source code elements (new class or method) or for existing code. The newly added tests must not fail due to compilation issues; however, it is fine if they identify a new bug in the project.

The newly added tests must improve the code coverage and not be trivial (for example, creating a new test from an existing test by changing the passed parameters; or, creating a test case for getter/setter method).

✓ Working on **“metrics-httpasyncclient”** Module

- Initially the module had following test coverage.



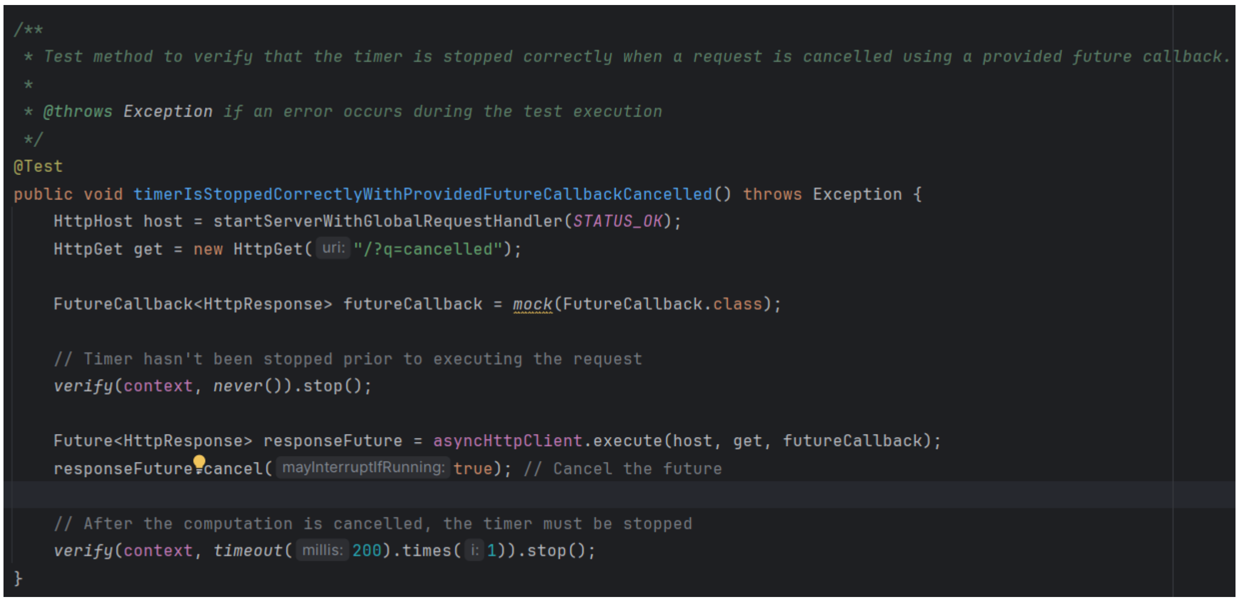
Element	Class, %	Method, %	Line, %	Branch, %
com.codahale.metrics	19% (18/91)	10% (66/658)	8% (160/1802)	5% (23/449)
httpasyncclient	75% (3/4)	50% (11/22)	47% (20/42)	16% (1/6)
InstrumentedNClientConnManager	0% (0/1)	0% (0/5)	0% (0/9)	100% (0/0)
InstrumentedNHttpClientBuilder	100% (3/3)	64% (11/17)	60% (20/33)	16% (1/6)
httpclient	0% (0/6)	0% (0/41)	0% (0/99)	0% (0/6)
CachedGauge	0% (0/1)	0% (0/4)	0% (0/20)	0% (0/10)
ChunkedAssociativeLongArray	0% (0/2)	0% (0/17)	0% (0/95)	0% (0/52)

Figure 15: Initial test coverage for “metrics-httpasyncclient” module.

- Following test cases were wrote to enhance the test coverage of **“metrics-httpasyncclient”** module.

- **Note:** While I did consider parameterizing the test cases, I opted against it because I wanted to specifically test the concurrent execution, which involves precise timing and coordination between multiple requests. Parameterizing such aspects of the tests might not be straightforward and could lead to complexities that obscure the intended focus of the tests. Therefore, I concluded that keeping the test methods separate would allow for clearer validation of each scenario independently.

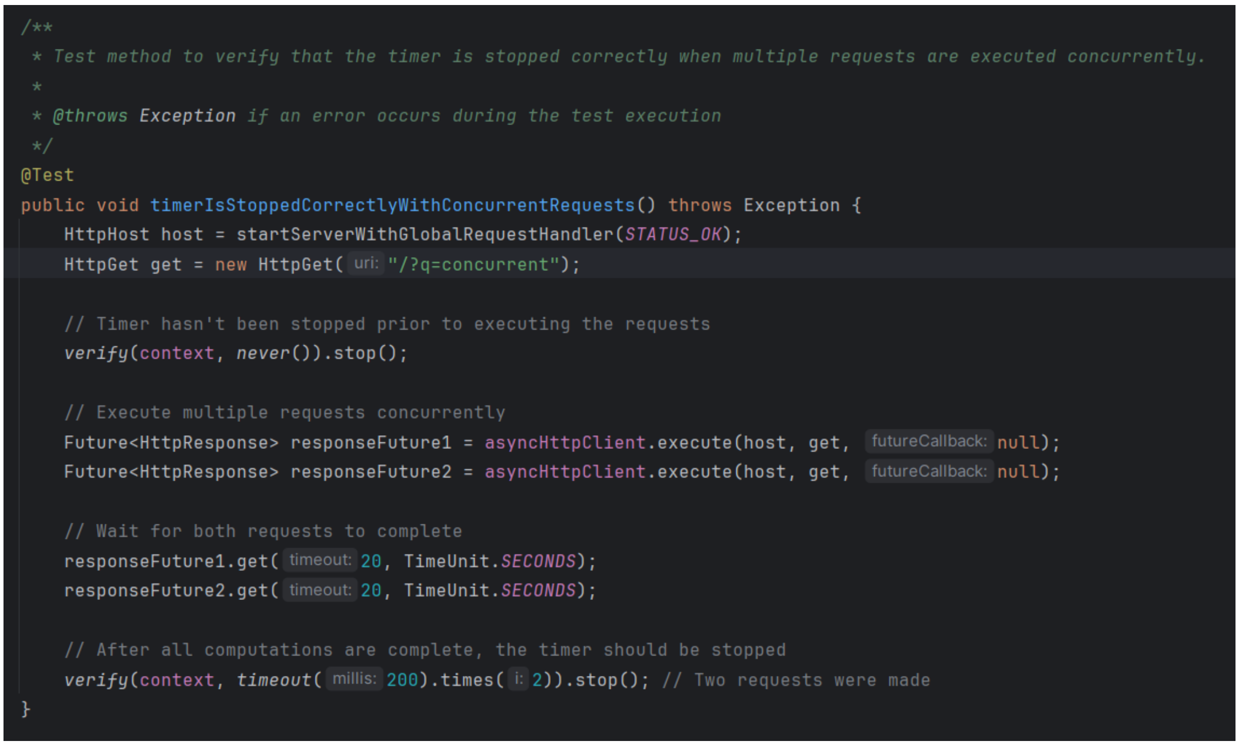
- **Tests Case 1:** The following test method ensures the proper termination of a timer when a request is canceled using a provided future callback. It starts by configuring a test server to respond with a success status code and prepares an HTTP GET request for the designated endpoint "/?q=cancelled". A mock future callback is then created for handling asynchronous responses. Before executing the request, it verifies that the timer associated with the test context has not been stopped. Upon executing the request, the method immediately cancels the future response. Finally, it verifies that the timer halts correctly after the cancellation, validating the expected behavior of the system under such circumstances.



```
/**
 * Test method to verify that the timer is stopped correctly when a request is cancelled using a provided future callback.
 *
 * @throws Exception if an error occurs during the test execution
 */
@Test
public void timerIsStoppedCorrectlyWithProvidedFutureCallbackCancelled() throws Exception {
    HttpHost host = startServerWithGlobalRequestHandler(STATUS_OK);
    HttpGet get = new HttpGet( uri: "/?q=cancelled");

    FutureCallback<HttpResponse> futureCallback = mock(FutureCallback.class);

    // Timer hasn't been stopped prior to executing the request
    verify(context, never()).stop();

    Future<HttpResponse> responseFuture = asyncHttpClient.execute(host, get, futureCallback);
    responseFuture.cancel( mayInterruptIfRunning: true); // Cancel the future

    // After the computation is cancelled, the timer must be stopped
    verify(context, timeout( millis: 200).times( i: 1)).stop();
}
```

Figure 16: Testcase 1 for "metrics-httpasyncclient" module.

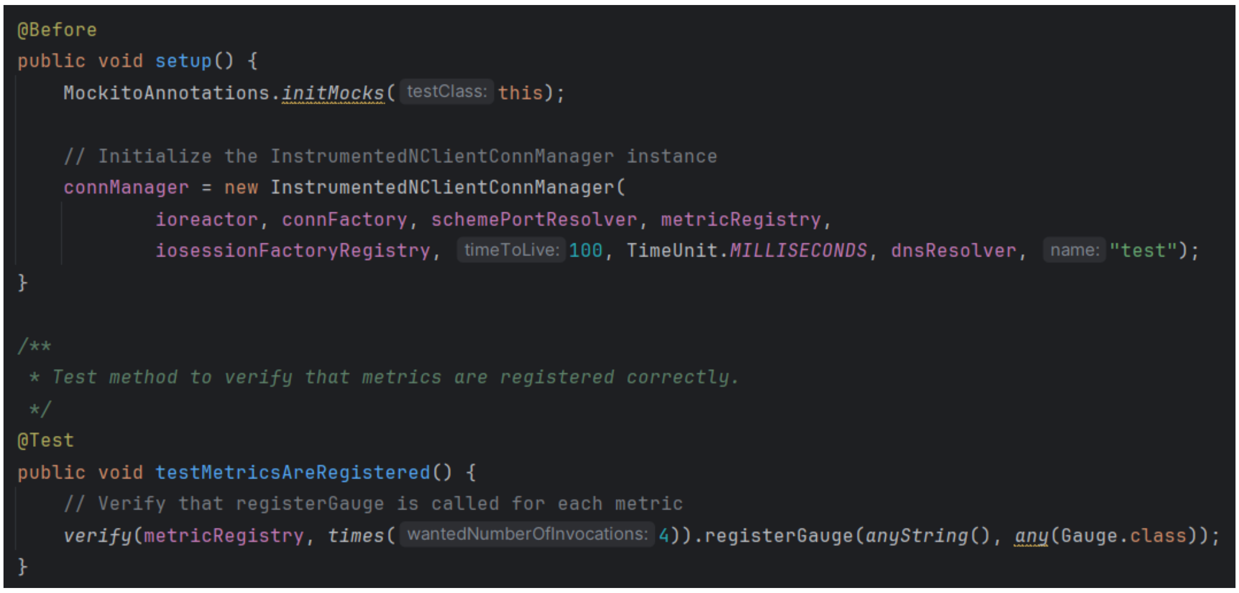
- **Tests Case 2:** The following test method validates the accurate termination of a timer when handling concurrent requests. It begins by setting up a test server to respond with a success status code. Two HTTP GET requests are then dispatched concurrently to the server's designated endpoint. After awaiting the completion of both requests within a specified timeout period, the method confirms that the timer halts correctly, ensuring the system's robustness under concurrent operations.



```java
/**
 * Test method to verify that the timer is stopped correctly when multiple requests are executed concurrently.
 *
 * @throws Exception if an error occurs during the test execution
 */
@Test
public void timerIsStoppedCorrectlyWithConcurrentRequests() throws Exception {
    HttpHost host = startServerWithGlobalRequestHandler(STATUS_OK);
    HttpGet get = new HttpGet( uri: "/?q=concurrent");

    // Timer hasn't been stopped prior to executing the requests
    verify(context, never()).stop();

    // Execute multiple requests concurrently
    Future<HttpResponse> responseFuture1 = asyncHttpClient.execute(host, get,  futureCallback: null);
    Future<HttpResponse> responseFuture2 = asyncHttpClient.execute(host, get,  futureCallback: null);

    // Wait for both requests to complete
    responseFuture1.get( timeout: 20, TimeUnit.SECONDS);
    responseFuture2.get( timeout: 20, TimeUnit.SECONDS);

    // After all computations are complete, the timer should be stopped
    verify(context, timeout( millis: 200).times( i: 2)).stop(); // Two requests were made
}
```

Figure 17: Testcase 2 for "metrics-httpasyncclient" module.

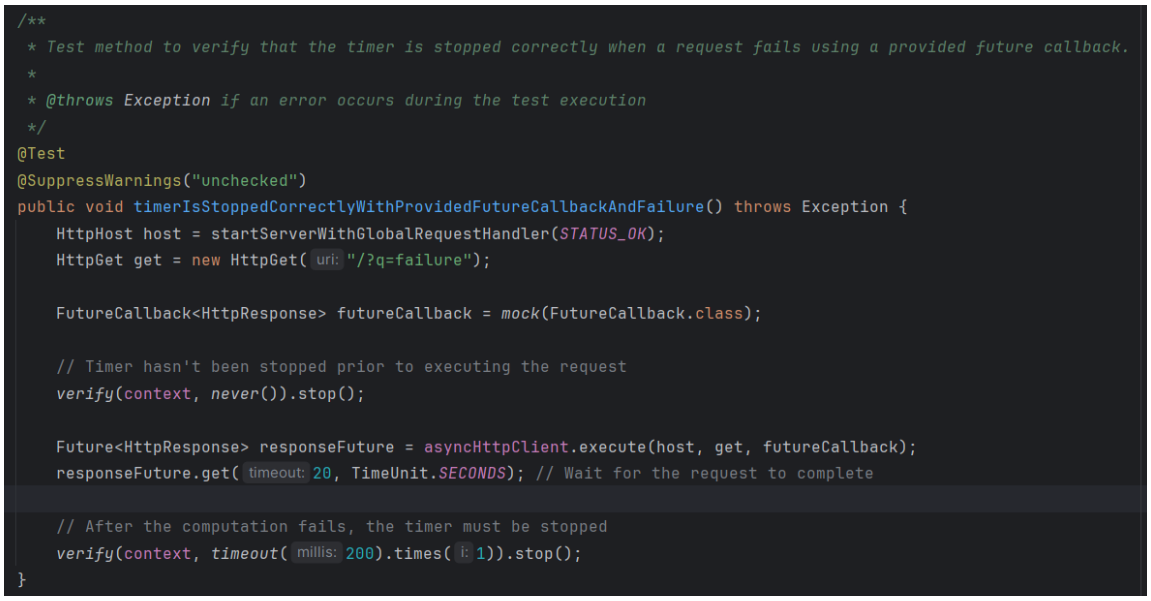
- **Tests Case 3:** The following test method aims to ensure the accurate registration of metrics within the system. It verifies that the `registerGauge` method is called for each metric precisely four times. By employing Mockito's verification functionality, the test confirms that the metrics are appropriately registered with the metric registry, thus validating the correct functioning of the metrics registration process. Moreover, it is for the "InstrumentedNClientConnManager" class which earlier had no test cases written.



```java
@Before
public void setup() {
    MockitoAnnotations.initMocks( testClass: this);

    // Initialize the InstrumentedNClientConnManager instance
    connManager = new InstrumentedNClientConnManager(
            ioreactor, connFactory, schemePortResolver, metricRegistry,
            iosessionFactoryRegistry,  timeToLive: 100, TimeUnit.MILLISECONDS, dnsResolver,  name: "test");
}

/**
 * Test method to verify that metrics are registered correctly.
 */
@Test
public void testMetricsAreRegistered() {
    // Verify that registerGauge is called for each metric
    verify(metricRegistry, times( wantedNumberOfInvocations: 4)).registerGauge(anyString(), any(Gauge.class));
}
```

Figure 18: Testcase 3 for "metrics-httpasyncclient" module.

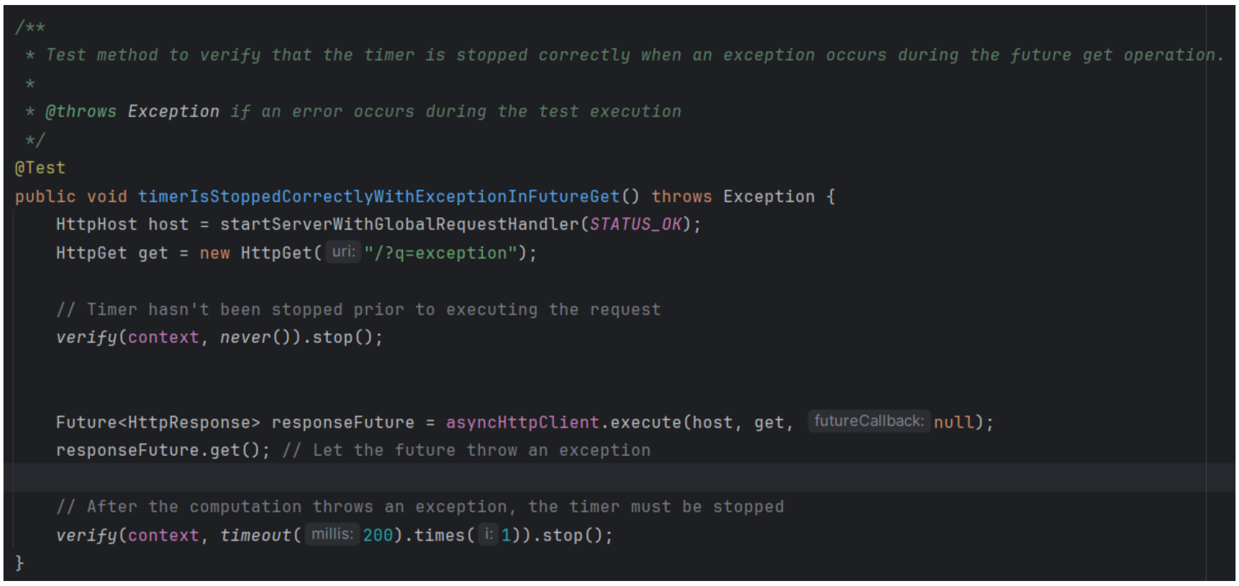
- **Tests Case 4:** The following test method verifies the accurate termination of a timer when a request fails, utilizing a provided future callback. It begins by configuring a test server to respond with a success status code and prepares an HTTP GET request for the designated endpoint "/?q=failure". A mock future callback is created to handle asynchronous responses. Prior to executing the request, it confirms that the timer associated with the test context remains active. Upon executing the request and waiting for completion within a specified timeout period, the method ensures that the timer halts correctly after the request fails, thus validating the expected behavior of the system in response to such failures.



```java
/**
 * Test method to verify that the timer is stopped correctly when a request fails using a provided future callback.
 *
 * @throws Exception if an error occurs during the test execution
 */
@Test
@SuppressWarnings("unchecked")
public void timerIsStoppedCorrectlyWithProvidedFutureCallbackAndFailure() throws Exception {
    HttpHost host = startServerWithGlobalRequestHandler(STATUS_OK);
    HttpGet get = new HttpGet( uri: "/?q=failure");

    FutureCallback<HttpResponse> futureCallback = mock(FutureCallback.class);

    // Timer hasn't been stopped prior to executing the request
    verify(context, never()).stop();

    Future<HttpResponse> responseFuture = asyncHttpClient.execute(host, get, futureCallback);
    responseFuture.get( timeout: 20, TimeUnit.SECONDS); // Wait for the request to complete

    // After the computation fails, the timer must be stopped
    verify(context, timeout( millis: 200).times( i: 1)).stop();
}
```

Figure 19: Testcase 4 for "metrics-httpasyncclient" module.

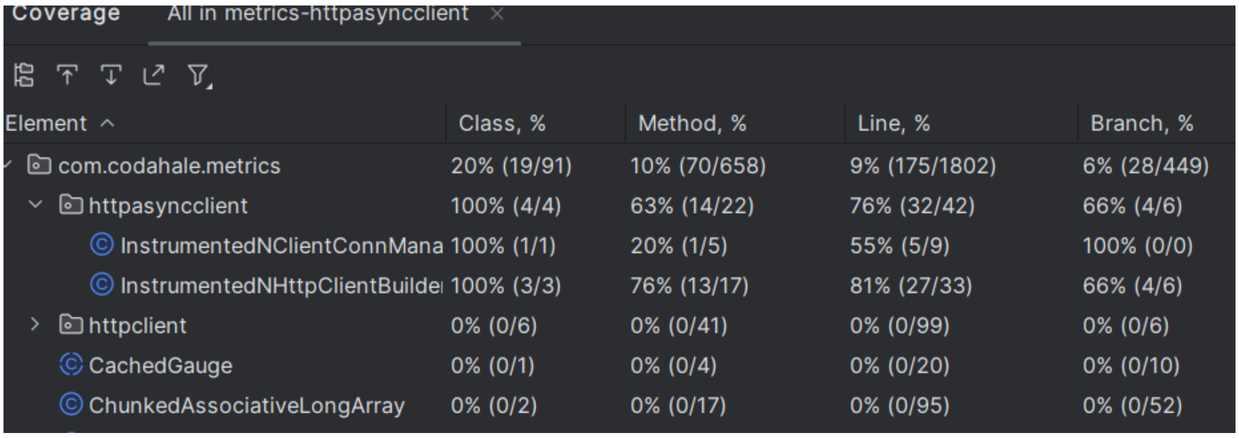
- **Tests Case 5:** The following test method ensures the correct termination of a timer when an exception occurs during the future operation. It starts by setting up a test server to respond with a success status code and prepares an HTTP GET request for the designated endpoint "/?q=exception". Prior to executing the request, it confirms that the timer associated with the test context remains active. Upon executing the request and allowing the future to throw an exception during the get operation, the method verifies that the timer halts correctly after the exception occurs, thereby validating the expected behavior of the system in handling such exceptions.



```java
/**
 * Test method to verify that the timer is stopped correctly when an exception occurs during the future get operation.
 *
 * @throws Exception if an error occurs during the test execution
 */
@Test
public void timerIsStoppedCorrectlyWithExceptionInFutureGet() throws Exception {
    HttpHost host = startServerWithGlobalRequestHandler(STATUS_OK);
    HttpGet get = new HttpGet( uri: "/?q=exception");

    // Timer hasn't been stopped prior to executing the request
    verify(context, never()).stop();


    Future<HttpResponse> responseFuture = asyncHttpClient.execute(host, get,  futureCallback: null);
    responseFuture.get(); // Let the future throw an exception

    // After the computation throws an exception, the timer must be stopped
    verify(context, timeout( millis: 200).times( i: 1)).stop();
}
```

Figure 20: Testcase 5 for "metrics-httpasyncclient" module.

- Finally, the test cases significantly increased the test coverage, as demonstrated in the following screenshot.

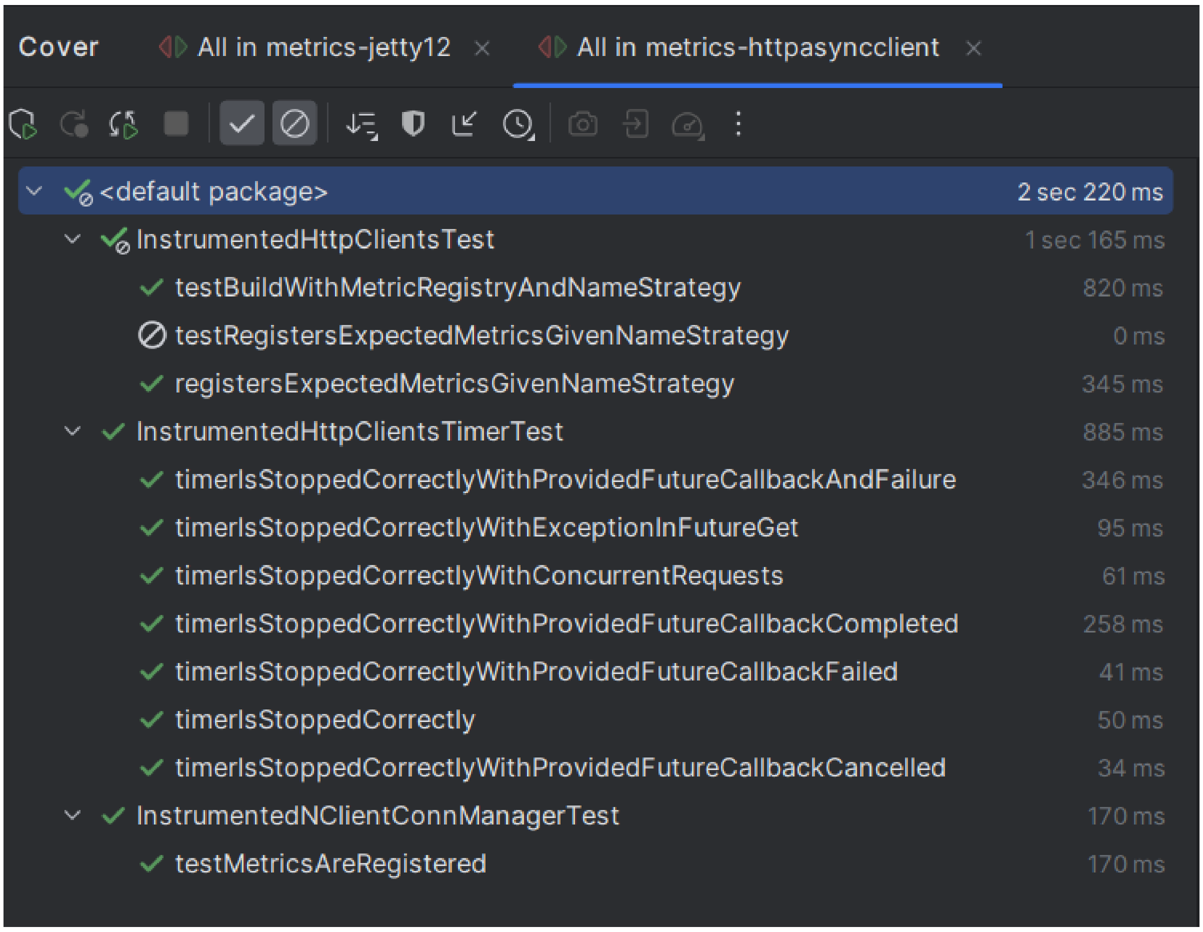


Coverage   All in metrics-httpasyncclient

Element	Class, %	Method, %	Line, %	Branch, %
com.codahale.metrics	20% (19/91)	10% (70/658)	9% (175/1802)	6% (28/449)
httpasyncclient	100% (4/4)	63% (14/22)	76% (32/42)	66% (4/6)
InstrumentedNClientConnMana	100% (1/1)	20% (1/5)	55% (5/9)	100% (0/0)
InstrumentedNHttpClientBuilde	100% (3/3)	76% (13/17)	81% (27/33)	66% (4/6)
httpclient	0% (0/6)	0% (0/41)	0% (0/99)	0% (0/6)
CachedGauge	0% (0/1)	0% (0/4)	0% (0/20)	0% (0/10)
ChunkedAssociativeLongArray	0% (0/2)	0% (0/17)	0% (0/95)	0% (0/52)

Figure 21: Final Improved test coverage of "metrics-httpasyncclient" module.

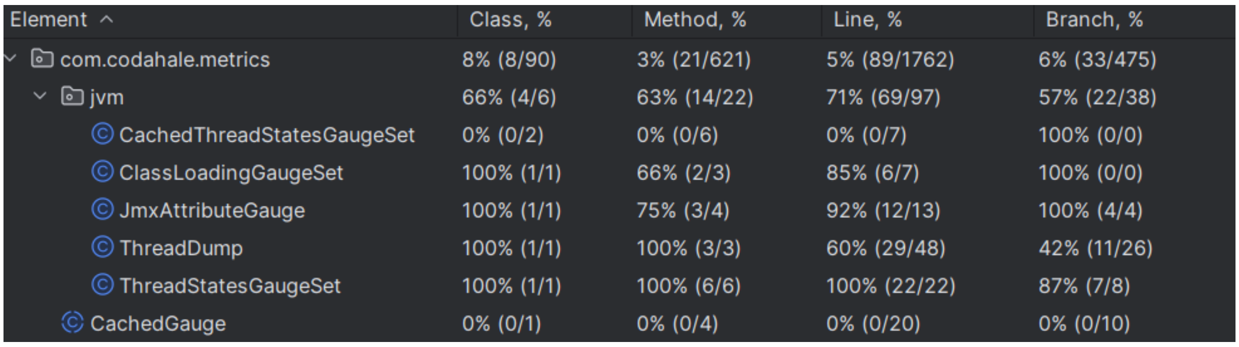
- The following screenshot shows that the added test cases did not break out the code and compiled successfully.



```
Cover    All in metrics-jetty12 ×    All in metrics-httpasyncclient ×

<default package>                                                    2 sec 220 ms
  InstrumentedHttpClientsTest                                        1 sec 165 ms
    testBuildWithMetricRegistryAndNameStrategy                          820 ms
    testRegistersExpectedMetricsGivenNameStrategy                         0 ms
    registersExpectedMetricsGivenNameStrategy                           345 ms
  InstrumentedHttpClientsTimerTest                                      885 ms
    timerIsStoppedCorrectlyWithProvidedFutureCallbackAndFailure         346 ms
    timerIsStoppedCorrectlyWithExceptionInFutureGet                      95 ms
    timerIsStoppedCorrectlyWithConcurrentRequests                        61 ms
    timerIsStoppedCorrectlyWithProvidedFutureCallbackCompleted          258 ms
    timerIsStoppedCorrectlyWithProvidedFutureCallbackFailed              41 ms
    timerIsStoppedCorrectly                                              50 ms
    timerIsStoppedCorrectlyWithProvidedFutureCallbackCancelled           34 ms
  InstrumentedNClientConnManagerTest                                    170 ms
    testMetricsAreRegistered                                            170 ms
```

Figure 22: Coverage runner output showing the code was able to compile.

- ✓ Working on **"metrics-jvm"** Module.
  - Initially the module had following test coverage.

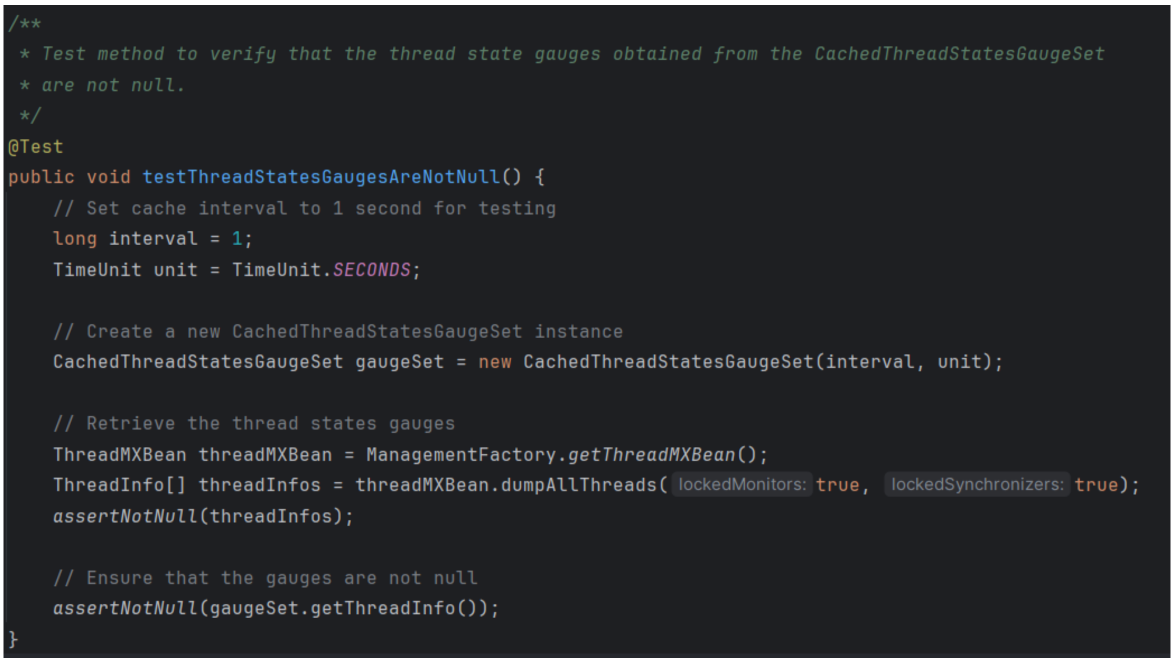


Element	Class, %	Method, %	Line, %	Branch, %
com.codahale.metrics	8% (8/90)	3% (21/621)	5% (89/1762)	6% (33/475)
jvm	66% (4/6)	63% (14/22)	71% (69/97)	57% (22/38)
CachedThreadStatesGaugeSet	0% (0/2)	0% (0/6)	0% (0/7)	100% (0/0)
ClassLoadingGaugeSet	100% (1/1)	66% (2/3)	85% (6/7)	100% (0/0)
JmxAttributeGauge	100% (1/1)	75% (3/4)	92% (12/13)	100% (4/4)
ThreadDump	100% (1/1)	100% (3/3)	60% (29/48)	42% (11/26)
ThreadStatesGaugeSet	100% (1/1)	100% (6/6)	100% (22/22)	87% (7/8)
CachedGauge	0% (0/1)	0% (0/4)	0% (0/20)	0% (0/10)

Figure 23: Initial test coverage for "metrics-jvm" module.

  - Following test cases were wrote to enhance the test coverage of **"metrics-jvm"** module.
    - **Tests Case 6:** The following test method verifies that the thread state gauges obtained from the `CachedThreadStatesGaugeSet` are not null. It sets the cache interval to 1 second for testing purposes and initializes a new `CachedThreadStatesGaugeSet`

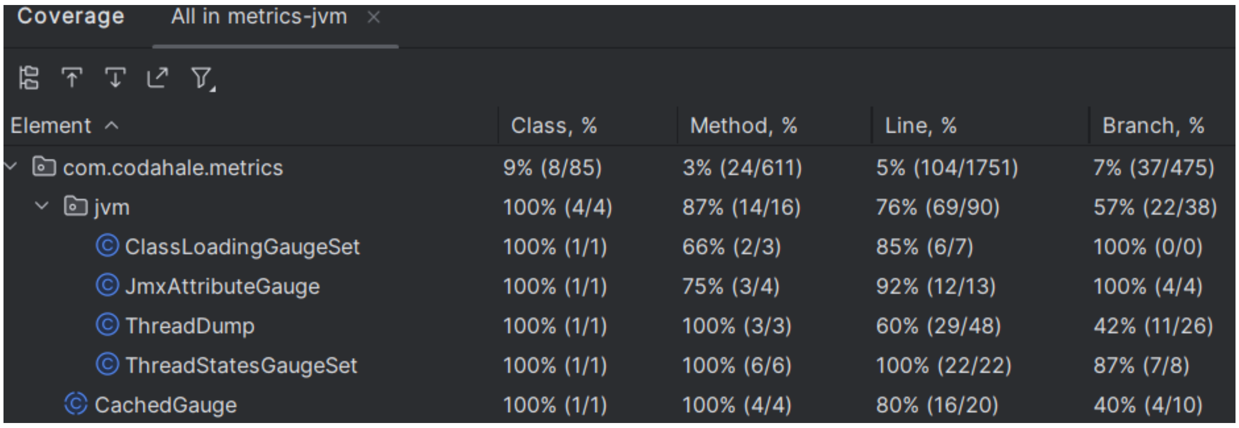
instance. Then, it retrieves the thread states using `ThreadMXBean` and ensures that the obtained thread info is not null. Finally, the method validates that the gauges obtained from the `CachedThreadStatesGaugeSet` are also not null, confirming the proper functionality of the gauge set in providing thread state information. Moreover, it is for the "CachedThreadStatesGaugeSet" class which earlier had no test cases written.



```java
/**
 * Test method to verify that the thread state gauges obtained from the CachedThreadStatesGaugeSet
 * are not null.
 */
@Test
public void testThreadStatesGaugesAreNotNull() {
    // Set cache interval to 1 second for testing
    long interval = 1;
    TimeUnit unit = TimeUnit.SECONDS;

    // Create a new CachedThreadStatesGaugeSet instance
    CachedThreadStatesGaugeSet gaugeSet = new CachedThreadStatesGaugeSet(interval, unit);

    // Retrieve the thread states gauges
    ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
    ThreadInfo[] threadInfos = threadMXBean.dumpAllThreads( lockedMonitors: true,  lockedSynchronizers: true);
    assertNotNull(threadInfos);

    // Ensure that the gauges are not null
    assertNotNull(gaugeSet.getThreadInfo());
}
```

Figure 24: Testcase 6 for "metrics-jvm" module.

- Finally, the test cases significantly increased the test coverage, as demonstrated in the following screenshot.

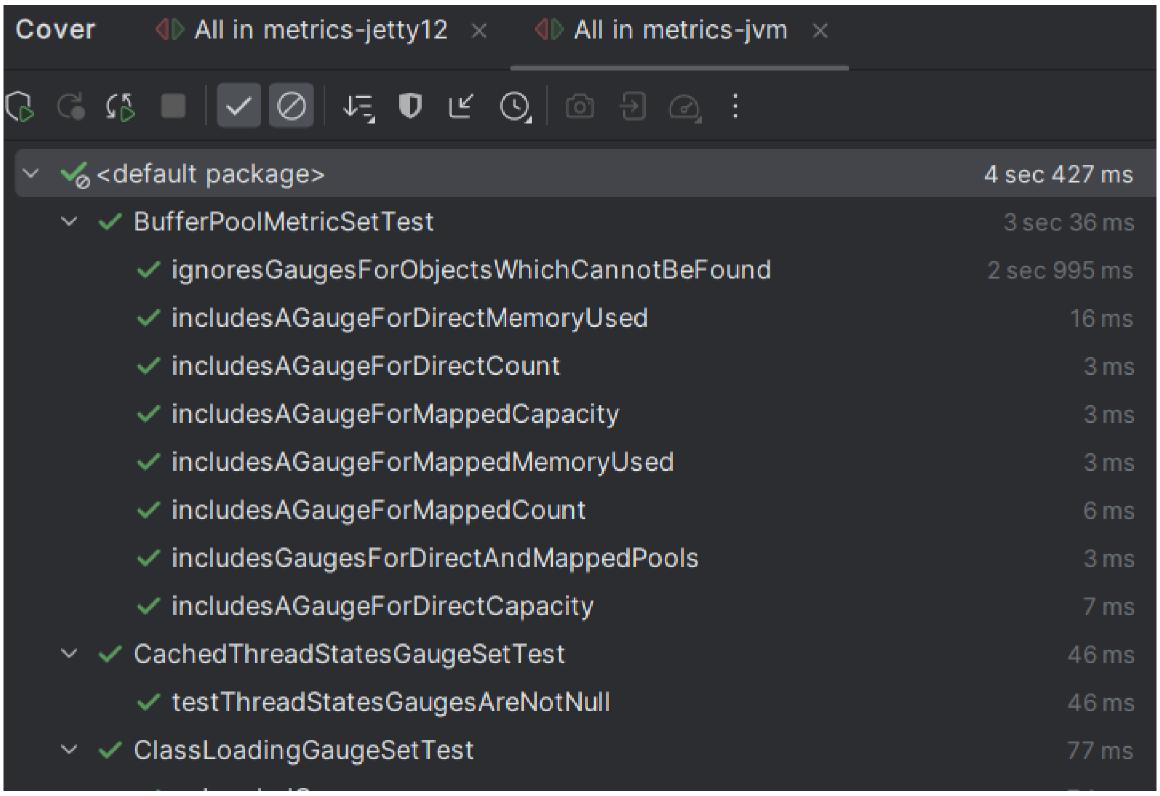


Coverage   All in metrics-jvm

Element	Class, %	Method, %	Line, %	Branch, %
com.codahale.metrics	9% (8/85)	3% (24/611)	5% (104/1751)	7% (37/475)
jvm	100% (4/4)	87% (14/16)	76% (69/90)	57% (22/38)
ClassLoadingGaugeSet	100% (1/1)	66% (2/3)	85% (6/7)	100% (0/0)
JmxAttributeGauge	100% (1/1)	75% (3/4)	92% (12/13)	100% (4/4)
ThreadDump	100% (1/1)	100% (3/3)	60% (29/48)	42% (11/26)
ThreadStatesGaugeSet	100% (1/1)	100% (6/6)	100% (22/22)	87% (7/8)
CachedGauge	100% (1/1)	100% (4/4)	80% (16/20)	40% (4/10)

Figure 25: Final test coverage for "metrics-jvm" module.

- The following screenshot shows that the added test cases did not break out the code and compiled successfully.

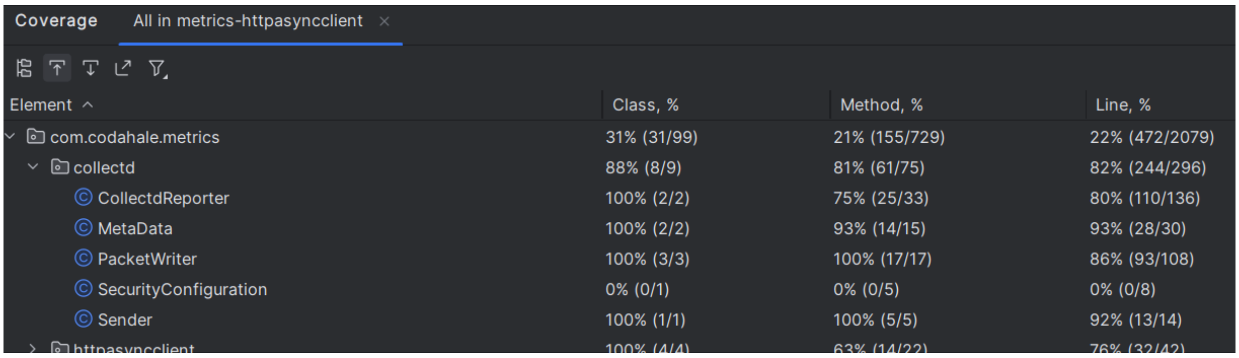


```
Cover    All in metrics-jetty12 ×    All in metrics-jvm ×

<default package>                                   4 sec 427 ms
  BufferPoolMetricSetTest                            3 sec 36 ms
    ignoresGaugesForObjectsWhichCannotBeFound       2 sec 995 ms
    includesAGaugeForDirectMemoryUsed                      16 ms
    includesAGaugeForDirectCount                            3 ms
    includesAGaugeForMappedCapacity                         3 ms
    includesAGaugeForMappedMemoryUsed                       3 ms
    includesAGaugeForMappedCount                            6 ms
    includesGaugesForDirectAndMappedPools                   3 ms
    includesAGaugeForDirectCapacity                         7 ms
  CachedThreadStatesGaugeSetTest                           46 ms
    testThreadStatesGaugesAreNotNull                       46 ms
  ClassLoadingGaugeSetTest                                 77 ms
```

Figure 26: Coverage runner output showing the code was able to compile.

✓ Working on **"collectd"** Module.

- Initially the module had the following test coverage.

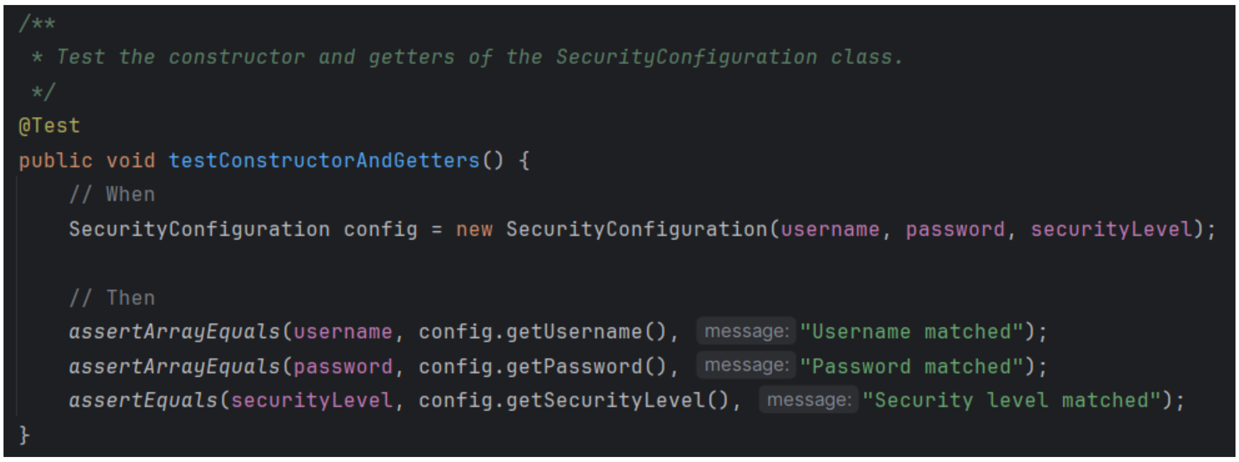


Coverage   All in metrics-httpasyncclient ×

Element	Class, %	Method, %	Line, %
com.codahale.metrics	31% (31/99)	21% (155/729)	22% (472/2079)
collectd	88% (8/9)	81% (61/75)	82% (244/296)
CollectdReporter	100% (2/2)	75% (25/33)	80% (110/136)
MetaData	100% (2/2)	93% (14/15)	93% (28/30)
PacketWriter	100% (3/3)	100% (17/17)	86% (93/108)
SecurityConfiguration	0% (0/1)	0% (0/5)	0% (0/8)
Sender	100% (1/1)	100% (5/5)	92% (13/14)
httpasyncclient	100% (4/4)	63% (14/22)	76% (32/42)

Figure 27: Initial test coverage for "collectd" module.

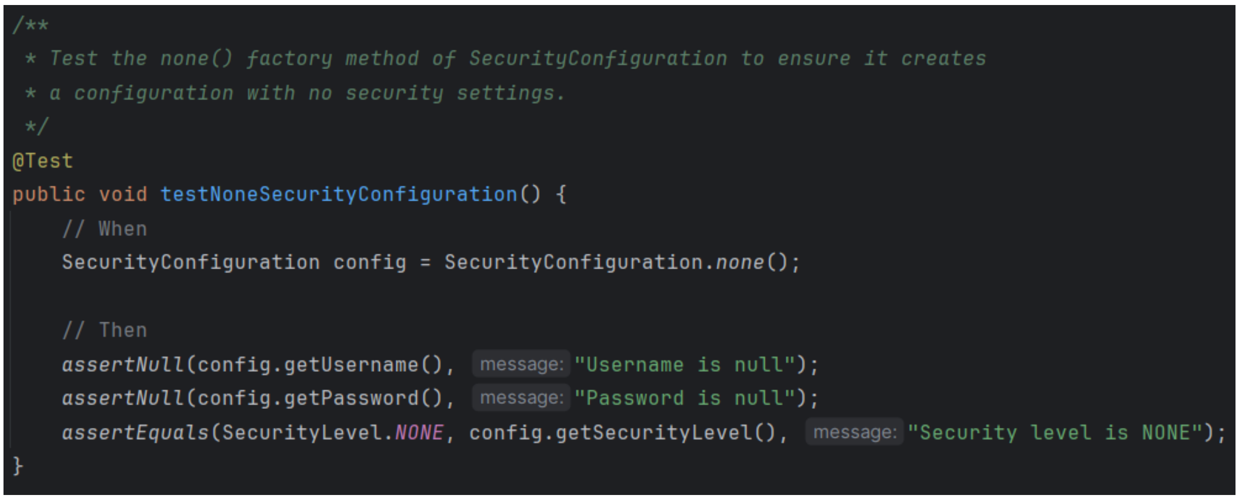
- **Tests Case 7:** The following test case evaluates the initialization and accessors of the `SecurityConfiguration` class. It sets up an instance with predefined values for username, password, and security level, then verifies that the getter methods retrieve the correct values. Specifically, it checks if the retrieved username and password match the expected ones, and if the security level matches the expected security level. These assertions ensure that the constructor properly initializes the object and that the getters retrieve the expected data. Note: Defined this for a class which has no written test cases.



```
/**
 * Test the constructor and getters of the SecurityConfiguration class.
 */
@Test
public void testConstructorAndGetters() {
    // When
    SecurityConfiguration config = new SecurityConfiguration(username, password, securityLevel);

    // Then
    assertArrayEquals(username, config.getUsername(),  message: "Username matched");
    assertArrayEquals(password, config.getPassword(),  message: "Password matched");
    assertEquals(securityLevel, config.getSecurityLevel(),  message: "Security level matched");
}
```

Figure 28: Testcase 7 for "collectd" module.

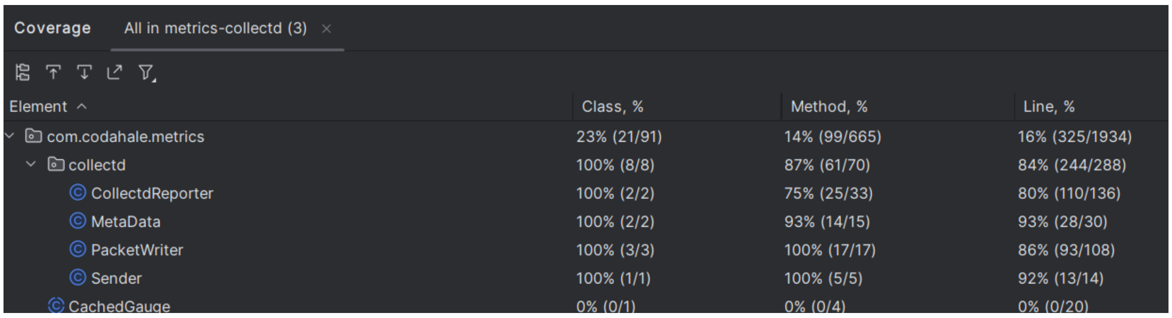
- **Tests Case 8:** The following test verifies the behavior of the `none()` factory method in the `SecurityConfiguration` class, ensuring that it correctly creates a configuration object with no security settings. It calls the `none()` method to obtain a `SecurityConfiguration` instance and then checks that the retrieved username and password are both `null`, indicating no security settings. Additionally, it verifies that the security level of the configuration is set to `SecurityLevel.NONE`. These assertions confirm that the `none()` factory method initializes the configuration object as expected, ensuring that no security settings are applied.



```
/**
 * Test the none() factory method of SecurityConfiguration to ensure it creates
 * a configuration with no security settings.
 */
@Test
public void testNoneSecurityConfiguration() {
    // When
    SecurityConfiguration config = SecurityConfiguration.none();

    // Then
    assertNull(config.getUsername(),  message: "Username is null");
    assertNull(config.getPassword(),  message: "Password is null");
    assertEquals(SecurityLevel.NONE, config.getSecurityLevel(),  message: "Security level is NONE");
}
```

Figure 29: Testcase 8 for "collectd" module.

- Finally, the test cases significantly increased the test coverage, as demonstrated in the following screenshot.

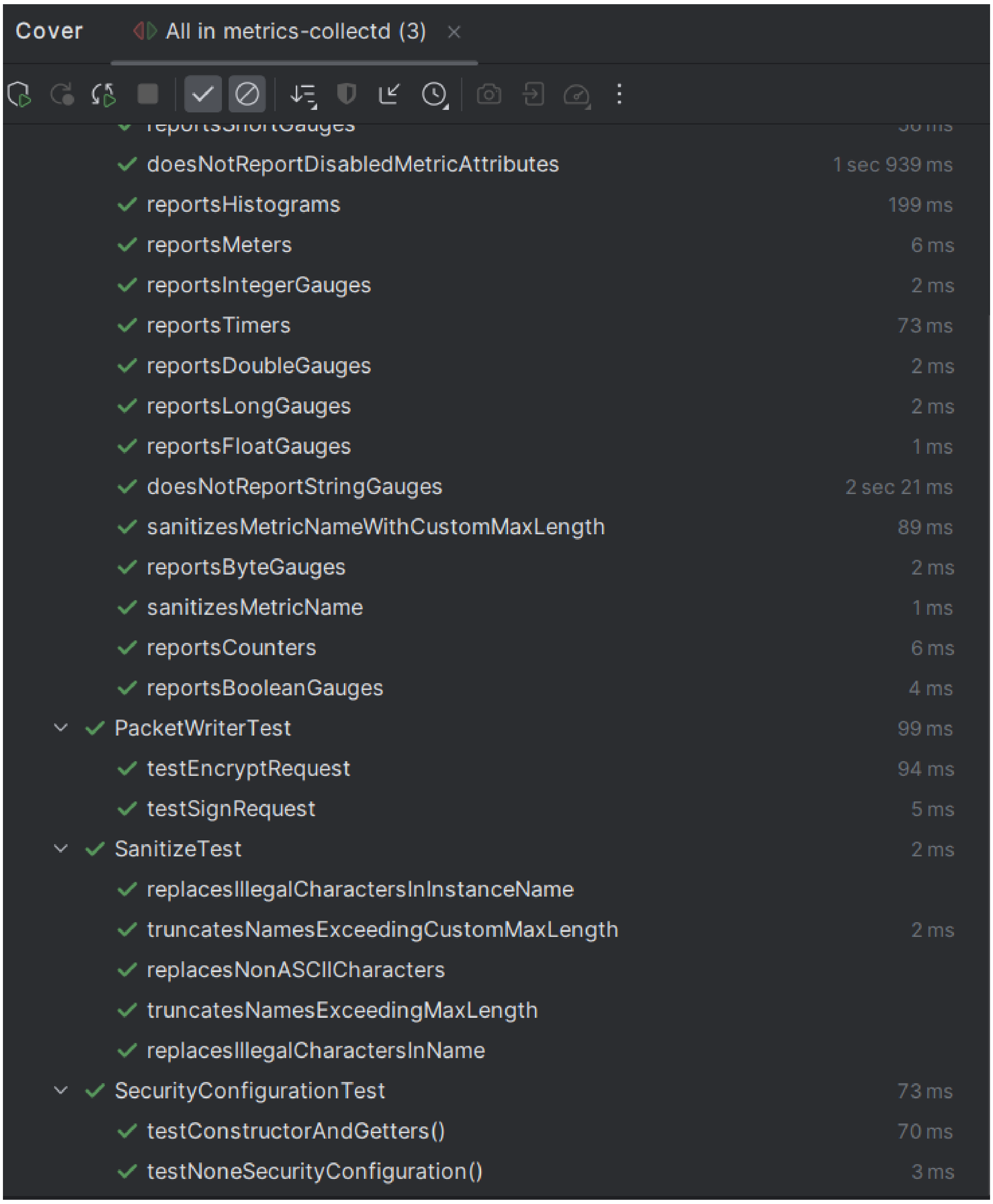


Coverage   All in metrics-collectd (3)

Element	Class, %	Method, %	Line, %
com.codahale.metrics	23% (21/91)	14% (99/665)	16% (325/1934)
collectd	100% (8/8)	87% (61/70)	84% (244/288)
CollectdReporter	100% (2/2)	75% (25/33)	80% (110/136)
MetaData	100% (2/2)	93% (14/15)	93% (28/30)
PacketWriter	100% (3/3)	100% (17/17)	86% (93/108)
Sender	100% (1/1)	100% (5/5)	92% (13/14)
CachedGauge	0% (0/1)	0% (0/4)	0% (0/20)

Figure 30: Final test coverage for "collectd" module.

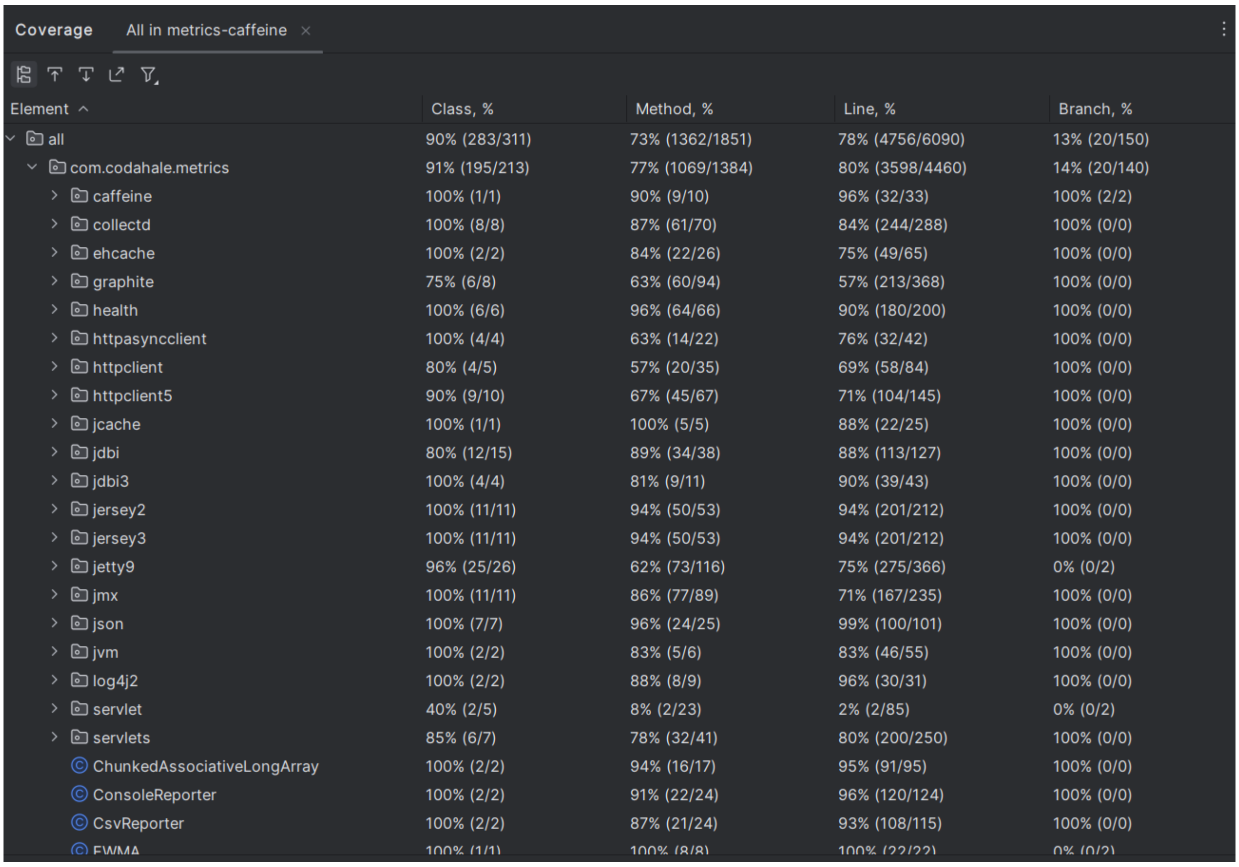
- The following screenshot shows that the added test cases did not break out the code and compiled successfully.



```
Cover    All in metrics-collectd (3) ×

  ✓ doesNotReportDisabledMetricAttributes          1 sec 939 ms
  ✓ reportsHistograms                                    199 ms
  ✓ reportsMeters                                          6 ms
  ✓ reportsIntegerGauges                                   2 ms
  ✓ reportsTimers                                         73 ms
  ✓ reportsDoubleGauges                                    2 ms
  ✓ reportsLongGauges                                      2 ms
  ✓ reportsFloatGauges                                     1 ms
  ✓ doesNotReportStringGauges                      2 sec 21 ms
  ✓ sanitizesMetricNameWithCustomMaxLength                89 ms
  ✓ reportsByteGauges                                      2 ms
  ✓ sanitizesMetricName                                    1 ms
  ✓ reportsCounters                                        6 ms
  ✓ reportsBooleanGauges                                   4 ms
✓ PacketWriterTest                                        99 ms
  ✓ testEncryptRequest                                    94 ms
  ✓ testSignRequest                                        5 ms
✓ SanitizeTest                                             2 ms
  ✓ replacesIllegalCharactersInInstanceName
  ✓ truncatesNamesExceedingCustomMaxLength                 2 ms
  ✓ replacesNonASCIICharacters
  ✓ truncatesNamesExceedingMaxLength
  ✓ replacesIllegalCharactersInName
✓ SecurityConfigurationTest                               73 ms
  ✓ testConstructorAndGetters()                           70 ms
  ✓ testNoneSecurityConfiguration()                        3 ms
```

Figure 31: Coverage runner output showing the code was able to compile.

Finally, the following screenshot demonstrates the comprehensive test coverage achieved by the addition of these 8 test cases. As a result, the branch coverage has increased by **more than 3 times**, and the **line coverage has improved by 2%**. Although class coverage remains the lowest, the overall test coverage has significantly improved for the selected repository.


Coverage   All in metrics-caffeine

Element	Class, %	Method, %	Line, %	Branch, %
all	90% (283/311)	73% (1362/1851)	78% (4756/6090)	13% (20/150)
com.codahale.metrics	91% (195/213)	77% (1069/1384)	80% (3598/4460)	14% (20/140)
caffeine	100% (1/1)	90% (9/10)	96% (32/33)	100% (2/2)
collectd	100% (8/8)	87% (61/70)	84% (244/288)	100% (0/0)
ehcache	100% (2/2)	84% (22/26)	75% (49/65)	100% (0/0)
graphite	75% (6/8)	63% (60/94)	57% (213/368)	100% (0/0)
health	100% (6/6)	96% (64/66)	90% (180/200)	100% (0/0)
httpasyncclient	100% (4/4)	63% (14/22)	76% (32/42)	100% (0/0)
httpclient	80% (4/5)	57% (20/35)	69% (58/84)	100% (0/0)
httpclient5	90% (9/10)	67% (45/67)	71% (104/145)	100% (0/0)
jcache	100% (1/1)	100% (5/5)	88% (22/25)	100% (0/0)
jdbi	80% (12/15)	89% (34/38)	88% (113/127)	100% (0/0)
jdbi3	100% (4/4)	81% (9/11)	90% (39/43)	100% (0/0)
jersey2	100% (11/11)	94% (50/53)	94% (201/212)	100% (0/0)
jersey3	100% (11/11)	94% (50/53)	94% (201/212)	100% (0/0)
jetty9	96% (25/26)	62% (73/116)	75% (275/366)	0% (0/2)
jmx	100% (11/11)	86% (77/89)	71% (167/235)	100% (0/0)
json	100% (7/7)	96% (24/25)	99% (100/101)	100% (0/0)
jvm	100% (2/2)	83% (5/6)	83% (46/55)	100% (0/0)
log4j2	100% (2/2)	88% (8/9)	96% (30/31)	100% (0/0)
servlet	40% (2/5)	8% (2/23)	2% (2/85)	0% (0/2)
servlets	85% (6/7)	78% (32/41)	80% (200/250)	100% (0/0)
ChunkedAssociativeLongArray	100% (2/2)	94% (16/17)	95% (91/95)	100% (0/0)
ConsoleReporter	100% (2/2)	91% (22/24)	96% (120/124)	100% (0/0)
CsvReporter	100% (2/2)	87% (21/24)	93% (108/115)	100% (0/0)
EWMA	100% (1/1)	100% (8/8)	100% (22/22)	0% (0/2)

Figure 32: Improved final test coverage of complete repository.

References

1. Dropwizard Metrics. (2024). Dropwizard Metrics. [Online]. Available: https://github.com/dropwizard/metrics. Accessed on: Mar 01, 2024.
2. ghloc. (n.d.). GitHub Lines of Code Counter. [Online]. Available: https://ghloc.vercel.app/. Accessed on: Mar 01, 2024.