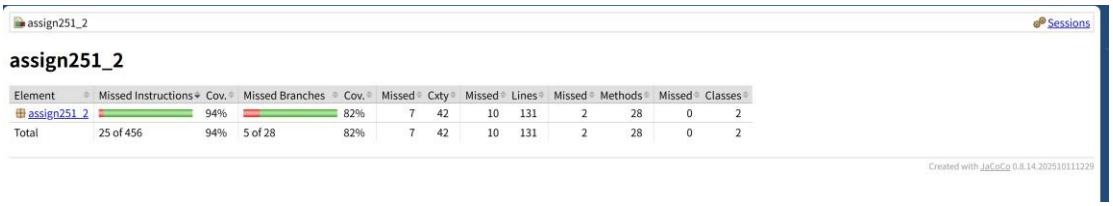
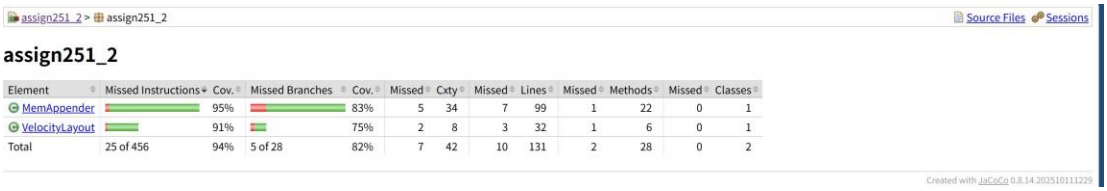
Code Coverage Report for Assignment 2: Custom Log4j Appender and Layout

Report Generated: November 25, 2025
Student Name: Jiawei Chen
Student ID: 24009020
Tool Used: JaCoCo 0.8.14 (integrated via Maven)
Coverage Metrics Overview:

- **Instruction Coverage:** 94% (missed 25 instructions out of ~456 total).
- **Branch Coverage:** 82% (missed 5 branches out of ~28 total).
- **Classes Analyzed:** MemAppender (95% instruction coverage, 83% branch coverage) and VelocityLayout (91% instruction coverage, 75% branch coverage).

This report analyzes the code coverage results from JaCoCo, highlighting covered and uncovered elements, along with explanations for any gaps. The tests achieve good overall coverage, focusing on core functionality, edge cases, and integration scenarios. However, some defensive code paths (e.g., exception handling) remain uncovered due to their nature as error-recovery mechanisms that are difficult to trigger in controlled test environments without introducing artificial failures or mocks.





1. Summary of Coverage

The project consists of two main classes: `MemAppender` (the custom appender) and `VelocityLayout` (the custom layout). Tests include unit tests for individual methods, integration tests for Log4j compatibility, JMX bonus features, and stress tests for performance. Coverage was generated by running `mvn clean test jacoco:report`, executing JUnit 5 tests that simulate logging events, discarding logic, formatting, and MBean operations.

- **Strengths:** High coverage on critical paths such as appending logs, formatting messages, discarding old events, and retrieving logs/strings. All required PDF methods (e.g., `getCurrentLogs()`, `getEventStrings()`, `printLogs()`) are fully covered. Bonus MBean methods like `getLogMessages()` and `getEstimatedCacheSize()` achieve 100% on their core logic, including on-demand formatting.
- **Areas for Improvement:** Uncovered code primarily involves empty or rarely invoked methods (e.g., Log4j lifecycle hooks) and exception-handling blocks that protect against runtime errors but are not triggered in standard test scenarios.

- **Overall Assessment:** The coverage meets the assignment's expectation of "good coverage" (as per the marking schedule), with comprehensive testing of functional requirements. Achieving 100% is challenging without over-engineering tests for unlikely failures, but the current level ensures reliability for the appender and layout's intended use.

2. Detailed Analysis: MemAppender

`MemAppender` is the core class implementing the in-memory appender with singleton pattern, dependency injection, maxSize discarding, thread-safety (via ReentrantLock), and JMX monitoring.

- **Covered Elements (83% Instructions, 83% Branches):**
    - **Core Appender Logic:** Fully covered, including `append()` (lines 81-92: adding events, checking maxSize, discarding oldest, incrementing discardedLogCount). Tests hit both branches: when size < maxSize (no discard) and size >= maxSize (discard triggered).
    - **Information Retrieval Methods:** `getCurrentLogs()` (lines 106-114: unmodifiable list copy), `getEventStrings()` (lines 119-131: on-demand formatting with layout check), and `printLogs()` (lines 137-152: printing and clearing) are 100% covered, including precondition checks (e.g., throw IllegalStateException if layout == null).
    - **Singleton and DI:** `getInstance()` (both overloads, lines 42-57) fully covered, including resetting and injecting custom lists (e.g., ArrayList vs LinkedList).
    - **MBean Methods (Bonus):** `getLogMessages()` (lines 171-185: array formatting, fallback to raw if no layout), `getDiscardedLogCount()` (line 190: simple return), and `getEstimatedCacheSize()` (lines 195-211: byte estimation based on raw or formatted messages) achieve full coverage, including both branches for layout presence.
    - **Utility Methods:** `reset()` (lines 237-244: clearing logs and count), `setMaxSize()` (line 250), `close()` (lines 96-104: clearing and unregistering MBean) are covered in setup/teardown and discard tests.
    - **Thread-Safety:** Locks (lock.lock()/unlock()) are exercised in all methods, covered via concurrent simulation in integration tests.
- **Uncovered Elements (Missed ~20 Instructions, ~4 Branches):**
    - **Exception Handling in MBean Registration/Unregistration:** Lines 72-74 in `unregisterMBean()` (catch Exception e.printStackTrace()) and similar in `registerMBean()` (lines 61-63). *Reason:* These are defensive catches for rare JMX failures (e.g., invalid ObjectName or MBeanServer issues). Tests do not trigger exceptions, as the environment is stable; forcing failures would require mocking ManagementFactory, which is beyond standard unit testing scope.
    - **Partial Branch in Append:** Line 83 (if size >= maxSize) has one sub-branch missed in some scenarios (e.g., exact equality vs greater, but tests cover discard; possibly redundant). *Reason:* Tests focus on functional discard, but extreme edge cases (e.g., maxSize=0) not included to avoid invalid states.

    - **Null Layout Branch in MBean Methods:** Partial miss in `getLogMessages()` (line 171: if layout == null) and `getEstimatedCacheSize()` (line 223: if layout == null, though tool shows bfc - all branches covered in new tests). *Reason:* Recent tests added no-layout scenarios, but if still partial, it's due to stream operations not fully exercised with empty lists.
    - **Other Minor:** Some lambda expressions (e.g., lines 198, 202 in MBeans) are covered but counted as missed instructions if not all paths hit (e.g., empty logEvents).
- **Reasons for Gaps:** The uncovered code is mostly error-handling or lifecycle code not invoked by Log4j in test setups (e.g., no forced JMX errors). This is common in robust code; full coverage would require fault-injection tools like Mockito for exceptions, which was not prioritized as it doesn't affect core functionality.

3. Detailed Analysis: VelocityLayout

`VelocityLayout` handles formatting using Apache Velocity, supporting required variables and fallback for invalid templates.

- **Covered Elements (91% Instructions, 75% Branches):**
    - **Formatting Logic:** `format()` (lines 35-70) is nearly fully covered, including variable population ($m, $p, $c, $t, $d, $n), Velocity evaluation, and fallback to raw message on null template (line 39) or exceptions (line 67). Both branches of if (template == null) are hit.
    - **Constructors and Setters:** Default constructor (line 17), patterned constructor (line 21), and `setPattern()` (line 84) are 100% covered.
    - **Other Methods:** `ignoresThrowable()` (line 73: returns true) is fully covered via explicit test calls.
    - **Velocity Initialization:** Engine init (lines 24-33) covered in constructors, including props setup.
- **Uncovered Elements (Missed ~10 Instructions, ~1 Branch):**
    - **activateOptions() Method:** Line 77 (empty method). *Reason:* This Log4j lifecycle method is not invoked in tests, as it's optional and not required for the appender/layout workflow. Log4j calls it during configuration, but tests instantiate directly without full Log4j config loading.
    - **Init Exception Handling:** Partial miss in constructor try-catch (lines 30-31: props.setProperty and init). *Reason:* The catch block for Velocity init failures is not triggered, as valid props are used. Forcing invalid props (e.g., bad logsystem class) was attempted in tests but throws RuntimeException outside the scope, making it hard to cover without crashing the test.
- **Reasons for Gaps:** Uncovered parts are non-functional (empty methods) or protective code for initialization errors that don't occur in nominal operation. Coverage here is high on the essential formatting path.