SE333 Final Project Reflection

MCP Testing Agent for Apache Commons Lang

11.16.25

Krisnarong Sakmunwong
SE333 (Software Engineering)
DePaul University

***Abstract*—This project delivers a local Model Context Protocol (MCP) testing agent that drives the Apache Commons Lang (SE333/Defects4J) Maven codebase from VS Code. The agent exposes tools to run the full test suite, generate and parse JaCoCo coverage, identify low-coverage classes, scaffold JUnit tests, and perform simple git operations. Everything runs on my machine via a lightweight HTTP/SSE server (`server.py`). On my final run the suite completed with 2303 tests, 0 failures, and 0 errors; JaCoCo reported ∼95% instruction and ∼90% branch coverage across 158 classes. I summarize setup, what the agent did, what was hard (tool wiring, inner classes, Java 21/JaCoCo, locales/ports), and how this workflow can be extended.**

***Index Terms*—software testing, MCP, JaCoCo, coverage, unit testing, automation**

I. Introduction

The goal was to build a small, practical testing agent that *augments* normal JUnit/Maven work: run tests, surface coverage gaps, suggest where to focus, and help create test files quickly. My deliverables include a working MCP server, a VS Code agent prompt, clear setup instructions, coverage artifacts, and a reflection on the value and limits of AI-assisted testing.

II. Methodology

A. Setup

The repository contains: (1) `codebase/` (*Commons Lang* Maven project); (2) `server.py` (FastMCP server exposing tools); (3) `.github/prompts/tester.prompt.md` (agent config); (4) `.vscode/mcp.json` (HTTP/SSE client pointing to `http://127.0.0.1:8000/sse`); (5) `README.md` with step-by-step instructions.

Environment:

- Java 21 and Maven 3.x
- Python 3.11+ with `mcp[cli]` and `uvicorn`
- macOS terminal for running `mvn` and the MCP server

B. Tools Exposed by the Agent

The server provides:

- **`run_mvn_tests`**: executes `mvn clean test jacoco:report`.
- **`read_coverage`**: parses `target/site/jacoco/jacoco.xml`, reports overall coverage and lowest classes.
- **`generate_test_skeleton`**: creates/updates a JUnit file for a fully qualified class (top-level).
- **`review_class`**: quick static hints (file size, TODOs, println s).
- **`suggest_boundary_tests`**: integer boundary-value suggestions.
- **Git helpers**: `git_status`, `git_add_all`, `git_commit`, `git_push`, `git_pull_request`.

C. Workflow Loop

1) Enable coverage and run tests: `mvn -Djacoco.skip=false clean test jacoco:report`.
2) Call `read_coverage` to see overall coverage and the worst classes.
3) Use `generate_test_skeleton` on a target class, then add real assertions.
4) Re-run tests, inspect JaCoCo HTML, and iterate. Commit only on green builds.

III. Results

A. Build and Coverage

A clean run produced:

- **Tests:** 2303 run / 0 failures / 0 errors / 0 skipped.
- **Coverage (JaCoCo):** ∼95% instruction, ∼90% branch across 158 classes.
- **Artifacts:** HTML at `codebase/target/site/jacoco/index`

B. Agent Operations

VS Code discovered 10 tools. I used the agent to:

- Run the suite and parse coverage data.
- Identify low-coverage classes (e.g., date/time formatters; builder styles).
- Create or refresh test skeletons such as:
  - `ToStringBuilderAgentTest.java`
  - `ToStringStyleAgentTest.java`
  - `JavaUnicodeEscaperAgentTest.java`
  - `FastDateParserAgentTest.java`
  - `FastDatePrinterAgentTest.java`

Skeletons don't move coverage by themselves, but they anchor focused assertions. For example, I added tests around *ToStringBuilder* null handling/empty output and *FastDateParser* 24h parsing with `UTC` timezone to hit specific branches.

C. Coverage Patterns

Two patterns stood out:

1) **Date/Time complexity:** Locale and timezone-sensitive branches (e.g., 24h vs 12h fields, literal tokens) require carefully crafted inputs; one test rarely covers all paths.
2) **Configurable styles:** *ToStringBuilder/ToStringStyle* behavior changes with flag combinations and container shapes; improving branch coverage needs small, systematic combinations over "happy-path" tests.

IV. Challenges ("Hard Stuff")

Tool wiring and setup. The MCP server exposes SSE at `/sse`. A 404 at `/` is normal, but it's easy to think the server is broken. Port 8000 conflicts also happened; I used `lsof+kill` to clear the listener. I hit `ModuleNotFoundError: mcp` once—fixed by activating the venv and installing `mcp[cli]`.

JaCoCo with Java 21. I explicitly forced coverage with `-Djacoco.skip=false`. Ensuring the plugin version worked with my JDK avoided "coverage skipped" confusion.

Inner classes. JaCoCo lists entries like `Foo$Bar` (or `Foo.Bar`). My skeleton tool targets top-level files; trying to "generate for inner class" fails because there is no `Bar.java`. The practical fix was: generate tests for the outer class (`Foo`) and exercise the inner behavior via the public API. (A future improvement is to auto-map inner names to their outer file.)

Earlier test triage. During iteration I saw locale-related date parsing issues and a hex-parsing expectation difference; after environment fixes and re-runs, the final suite is fully green. The lesson: many "failures" are setup/env mismatches, not code defects.

V. Lessons Learned

AI helps aim the effort. The agent is great at finding where coverage is thin and creating structure for tests. The human still provides the oracles (expected outputs), especially for tricky logic like date/time formatting.

DX matters. Small quality-of-life details (clear tool descriptions, returning concrete file paths, and repeatable Maven steps) made the loop fast and reliable.

Coverage is a compass, not a guarantee. It tells you *where* to look; correctness still requires good assertions and edge cases.

VI. Future Work

- **Assertion synthesis:** derive expected outputs from golden cases or metamorphic relations (e.g., round-trips).
- **Locale/timezone sweeps:** automate matrices to hit formatter branches (DST, unusual locales).
- **Static analysis integration:** surface SpotBugs/PMD warnings next to coverage to prioritize risky code.
- **Inner-class mapping:** let the tool accept `Foo$Bar` and route to `Foo.java` automatically.
- **PR automation:** generate brief PRs with coverage deltas and failing-test repro notes.

Artifacts

Code and instructions: `https://github.com/yellownation10`

References

[1] JaCoCo Code Coverage Library, https://www.jacoco.org
[2] Apache Commons Lang, https://commons.apache.org/proper/commons-lang/