AI-Assisted Test Generation and Coverage Improvement Using the Model Context Protocol (MCP)

Jericho Guiang
SE333
DePaul University
jguiang@depaul.edu

***Abstract*—This project explores the use of the Model Context Protocol (MCP) to build an intelligent software testing agent capable of autonomously generating, executing, and iteratively improving JUnit test cases. The system integrates a Python-based MCP server with a Java Maven project instrumented using JaCoCo for coverage analysis, as well as automated Git workflows. Results demonstrate effective end-to-end automation and highlight meaningful insights into AI-assisted software engineering.**

***Index Terms*—MCP, test automation, software agents, AI-assisted development, JaCoCo, JUnit, code coverage.**

I. Introduction

Modern software engineering increasingly incorporates AI-driven tooling to accelerate testing, debugging, and development workflows. This project implements an intelligent testing agent that combines MCP with Java, Maven, and JaCoCo to automatically generate and refine test cases.

The objective is to simulate a fully automated AI testing pipeline that identifies coverage gaps, produces new tests, and incrementally increases code coverage with minimal user intervention. Two creative extensions—a specification-based test generator and an AI-powered code review tool—further demonstrate the applicability of AI to both dynamic and static program analysis.

II. Methodology

The system architecture consists of three major components.

A. MCP Server

A FastMCP server provides MCP tool endpoints for:

- running Maven tests,
- extracting and parsing JaCoCo coverage results,
- generating JUnit test templates,
- performing Git automation (status, add, commit, push),
- executing creative extensions.

B. Iterative Test Improvement Pipeline

A Python script (`phase4_agent.py`) implements the iterative improvement cycle:

1) run `mvn test`,
2) parse the JaCoCo XML coverage report,
3) extract uncovered methods,
4) generate new JUnit tests,
5) re-run the test suite,
6) commit improvements through Git automation.

C. Creative Extensions

Two additional MCP tools were implemented:

- **Specification-Based Testing**: performs boundary value analysis and equivalence partitioning.
- **AI Code Review Agent**: performs static analysis, identifies code smells, and recommends refactoring improvements.

III. Results and Discussion

A. Coverage Improvement Patterns

The initial test coverage of the provided Java project was measured at **48.28%**. The agent successfully parsed JaCoCo output, identified uncovered methods (including the constructor and `main()`), and generated new test skeletons using the MCP tools.

Although the analyzed sample project contained limited logic—meaning the new tests did not dramatically increase coverage—they demonstrated a fully automated end-to-end workflow that would scale significantly better in real-world codebases.

B. Observations About AI-Generated Tests

The AI-generated tests showed several consistent patterns:

- Tests compile correctly and follow standard JUnit structure.
- Assertions require human judgment when behaviors are unspecified.
- Methods with limited logic (e.g., constructors or empty `main()`) produce limited measurable coverage improvements.

Overall, the toolchain successfully demonstrated automated coverage analysis, test generation, and Git-based iteration.

IV. AI-Assisted Development Insights

Several insights emerged:

- AI drastically reduces the time required to scaffold test suites.
- MCP tools allow seamless chaining of build, analysis, and version-control operations.

- Human verification remains essential when business rules or expected behavior are unclear.
- LLMs excel at structured template generation but struggle with deeper semantic correctness.

AI acted as a powerful accelerator rather than a substitute for human-driven testing.

V. Future Enhancement Recommendations

Potential improvements include:

- adding mutation testing (e.g., PIT) to measure behavioral correctness,
- adding assertion inference via symbolic execution or runtime tracing,
- integrating stronger security scanning tools such as CodeQL,
- supporting automated bug localization from failing tests,
- enabling reinforcement-style iterative improvement across multiple runs.

VI. Conclusion

This project demonstrates a practical implementation of an AI-driven testing agent that integrates MCP, Java, Maven, JaCoCo, and Git automation. Although coverage gains were modest due to the simplicity of the analyzed codebase, the system successfully showcases the potential of AI-assisted software testing and automated development workflows.