SE333 - AI-Assisted Testing Demonstration

Sofia C. Barrios
SE333 Software Testing
Professor Dong Jae Kim

***Abstract*—This report demonstrates the workflow of an AI-assisted agent for automated JUnit test generation in Java Maven projects using the Model Context Protocol (MCP) framework. It details the end-to-end process from user command to test generation, coverage improvement, and integration with GitHub Actions for continuous reporting.**

I. Introduction

This project demonstrates an AI-assisted approach that integrates MCP tools and an LLM to generate, execute, and iteratively refine tests while ensuring code quality. The agent also integrates with GitHub Actions to automatically update coverage and style reports.

II. Feature Walkthrough

A. User Input to Processing to Output

- **Input:** The user issues the command `improve coverage` to trigger the AI agent.
- **Processing:**
  1) Reads `main.py` to understand project configuration.
  2) Parses the JaCoCo coverage report to identify low-coverage classes and methods.
  3) Generates new tests using boundary, equivalence, decision table, and contract-based methods.
  4) Iteratively executes tests, refines assertions, and updates coverage.
- **Output:**
  - Updated JUnit test suite with additional assertions.
  - Updated coverage metrics (line, branch, instruction, method).
  - Automatically uploads changes to GitHub, triggering GitHub Actions to generate new JaCoCo and Checkstyle reports.

B. Prompts, Configuration, and MCP Commands

- Trigger command: `"improve coverage prompt"`.
- Configuration:
  - Maven project path (`main.py`).
  - MCP tool definitions for coverage parsing, test generation, and automated commits.
- Design reasoning: A single command automates test improvement end-to-end while integrating code quality analysis.

C. LLM-Based Reasoning and Automation Steps

- Determines which classes or methods require additional tests based on coverage gaps.
- Selects appropriate test types (boundary, equivalence, decision table, contract) based on method signatures.
- Generates dynamic assertions and iteratively refines tests.
- Pushes updates to the repository, triggering CI/CD via GitHub Actions.

D. Design Decisions, Trade-offs, and Constraints

- **Design:** Full automation using MCP + LLM to improve coverage while enforcing coding standards.
- **Trade-offs:**
  - Pros: Reduces manual effort, improves coverage, integrates CI/CD.
  - Cons: AI may generate redundant or incomplete tests; runtime increases with large projects.
- **Constraints:**
  - AI cannot access runtime state outside Maven execution.
  - Coverage and style analysis are limited to JaCoCo and Checkstyle outputs.

III. Conclusion

The demonstration shows that a single user command can trigger an end-to-end AI-assisted testing workflow, including coverage analysis, test generation, iterative refinement, and automated integration with GitHub Actions. This approach streamlines test improvement, enforces coding standards, and provides continuous feedback on project quality.