SE333 - Final Project

Sofia C. Barrios
SE333 Software Testing
Professor Dong Jae Kim

Abstract—**This project implements an AI-assisted agent to automatically generate, run, and iteratively improve JUnit tests for Java Maven projects. It leverages specification-based testing (boundary, equivalence class, decision table, contract) along with Checkstyle and JaCoCo coverage analysis to ensure code quality. The goal is to achieve at least 80% code coverage while maintaining passing tests and enforcing coding standards. This report summarizes methodology, coverage improvement patterns, lessons learned, and future enhancements.**

I. Introduction

This project explores an AI-assisted approach to test generation for Java Maven projects, utilizing the Model Context Protocol (MCP) framework.

The goal is to automatically generate, execute, and iteratively improve JUnit tests to achieve high code coverage while enforcing coding standards. By combining specification-based testing techniques with static and dynamic analysis tools, the system aims to reduce manual effort and improve software quality.

II. Methodology

The AI agent was implemented using the MCP framework in Python. Key components include:

- **Test Generation:** Basic JUnit tests plus specification-based tests (boundary, equivalence, decision table, contract).
- **Execution & Coverage Analysis:** Automatic Maven test execution and JaCoCo coverage parsing.
- **Static Analysis:** Checkstyle integration to enforce coding standards.
- **Iterative Improvement:** Feedback loop to refine tests until coverage goals are met.
- **Version Control Automation:** Git commits and pull requests created automatically based on test and coverage results.

III. Results & Discussion

The agent increased coverage by generating targeted tests for uncovered methods and edge cases.

A. Lessons Learned About AI-Assisted Development

- Automated test generation saved significant manual effort.
- Some edge cases still required manual guidance to ensure meaningful assertions.
- Debugging failures revealed that AI sometimes generated redundant or incomplete tests.

B. Test Case: "Perform Tester Prompt"

When instructed to "perform tester prompt," in the chat, the agent utilized MCP tools by considering the provided tester prompt md. This workflow served as a form of manual verification of the MCP tools, which was the primary approach during development. The coverage improvement process typically took 8–10 minutes to complete.

The sequence followed by the agent was:

1) **Read `main.py`** – the agent ingested the main project configuration to understand the current testing setup.
2) **Analyze coverage** – it executed `parse_results` to evaluate the current JaCoCo coverage report, identifying files and classes with the largest gaps.
3) **Generate specification-based tests** – for each targeted class, the agent used MCP tools to create additional tests, including:
   - Boundary value tests
   - Equivalence class tests
   - Decision table tests
   - Contract-based tests
4) **Run tests iteratively** – the agent executed the new tests, re-parsed results, and iteratively improved coverage until the targeted threshold was approached or achieved.

1) Coverage Improvement Summary: Despite starting with an already high coverage (>93%), the agent was able to improve metrics across multiple dimensions:

- **Line coverage:** 93.53% → 93.57% (+4 lines covered)
- **Branch coverage:** 90.27% (unchanged)
- **Instruction coverage:** 94.04% → 94.08% (+21 instructions covered)
- **Method coverage:** 93.78% → 93.86% (+2 methods covered)

Key improvements include:

- Addition of 400+ new test assertions targeting critical components.
- New comprehensive test classes:
  - `ConversionBoundaryTest` – 97 new assertions
  - `ConversionEquivalenceTest` – 85 new assertions
  - `ExtendedMessageFormatBoundaryTest` – 150+ assertions
  - `ToStringBuilderBoundaryTest` – 120+ assertions
- Focus on high-impact classes with lowest coverage, such as `Conversion.java`,

`ExtendedMessageFormat.java`, and `ToStringBuilder.java`.
- Enhanced test quality using boundary value analysis, equivalence class testing, contract-based checks, and decision table testing.

Considering the vastness of the codebase, even these slight improvements are significant. The generated test suite provides better error detection, improved regression testing, and more validation of critical functionality.

C. Future Enhancements

Future improvements to the AI-assisted testing system could include:

- **Static Analysis with SpotBugs:** Integrate SpotBugs to catch additional code issues.
- **Organized Reports:** Move JaCoCo and Checkstyle reports into a dedicated folder for better project organization.
- **Coverage Visualization:** Add charts or summaries to easily see which parts of the code are well-tested and which need more attention.