CS 520
Homework 4
Automatic Test Generation: Coverage, Quality, and AI

Due: **Tuesday, December 8, 2026, 9:00 AM EST**. Students may work together on this assignment, but each student must submit their own assignment. If you work with others, you must clearly state the names of the students you worked with. Your responses to questions must be your own, but there is no penalty for working together. You are allowed to use AI tools to assist with this assignment (in fact, it is expected).

Late assignments will not be accepted without **prior** permission.

This assignment takes time. Do not wait until the last minute to start the assignment. Setting up the tools and running all the coverage configurations can take several hours.

Before you start, write down how much you trust (from 1, very low, to 5, very high) tests generated by Randoop and tests generated by an AI agent. You will need these ratings in Task 13. As you work, keep a record of both the elapsed time and your *active time* (the time you spent actively working) on (a) setting up Randoop, (b) running Randoop, (c) setting up your AI agent, and (d) each of the Plan, Generate, and Critique phases with the agent. You will need these times in Tasks 3, 11, and 12.

Overview and goal

In this assignment you will use an automated test generator and an AI coding agent to create unit tests for an existing Java program. You will compare the generated tests with the developer-written tests and with one another.

A test suite can execute a large amount of code without checking whether that code behaves correctly. An automatically generated assertion may describe what the current program does rather than what it *should* do. A passing test suite is useful evidence, but it is not a proof of correctness. Recall the problem of overfitting in automated program repair: passing the available tests does not establish that a patch is correct. Here you will examine the other side of the same problem—whether generated tests provide useful checks of behavior.

Your goal is to understand what Randoop and an AI agent do well, where they fall short, and what a coverage report can and cannot tell you. You do not need to show that either tool is "better." You need to support your conclusions with your own results.

Acknowledgement

This assignment builds on an assignment originally designed by Wes Weimer and colleagues for EECS 481 (Software Engineering) at the University of Michigan. They also provided the edited JFreeChart build used here.

Getting started

It is your responsibility to download, compile, and run the subject program and associated tools. You may use an AI agent, documentation, and the internet to help with setup. Class staff cannot provide technical support for setting up your environment, but AI is great at providing such help. You may run the work locally or in a compatible remote Linux environment. If you are using Windows, you will likely need to rely on

WSL: Windows Subsystem for Linux (https://learn.microsoft.com/en-us/windows/wsl/install). Again, use AI to help you understand if it is needed and how to set it up. We recommend Java 17. Record your operating system, Java and Maven versions, Randoop version, and AI tool. Before you change anything in the project, put it under version control (see "Restore injected faults" in the Technical Hints).

Subject program: JFreeChart

Use the *course-provided, edited* JFreeChart 1.5.6 archive: https://people.cs.umass.edu/~brun/class/2026Fall/CS520/hw4/jfreechart-1.5.6-edit.zip. Do not substitute a different JFreeChart release. The edited build is intended to support JaCoCo coverage reporting. Unzip the archive and run:

```
java -version
mvn -version
mvn clean initialize test
```

Run the Maven commands from the JFreeChart project directory. A successful build should produce a JaCoCo HTML report under `target/site/jacoco/`. Check that it actually exists. If the commands do not work in your environment, investigate and document what you changed.

Randoop

Use Randoop 4.3.4 to generate Java unit tests. Read its user manual. A typical invocation has the following shape (you must adapt the classpath to include JFreeChart and its dependencies):

```
java -cp "target/classes:/path/to/randoop-all-4.3.4.jar:..." \
  randoop.main.Main gentests \
  --classlist=classes.txt --time-limit=120
```

The example uses Unix classpath separators. On Windows, use the separator appropriate for your environment. Randoop's generated JUnit tests may need separate compilation and test configuration; do not assume that copying them into the project will automatically make Maven run them (see the Technical Hints). Preserve the generated source files and record any adaptations needed to execute them.

AI agent

Use an AI coding agent that can edit files and run commands (for example, GitHub Copilot's agent mode in VS Code) to write JUnit tests for the *same classes* you selected for Randoop. You may choose the agent, but report its name and model if known. Do not claim that its generated expected values are correct merely because the tests pass.

Use three phases:

1. **Plan (no code).** Provide the relevant source, the developer-plus-Randoop coverage report (from Task 4), the JUnit version, and constraints. Ask for a testing plan without code. Challenge the plan and ask what it overlooks.

2. **Generate.** Ask the agent to implement the plan in no more than 20 test files, as JUnit 5 (Jupiter) tests like the developer tests, with file names ending in `AITest.java` (e.g., `XYPlotAITest.java`) so that you can run them separately (see the Technical Hints). Aim for tests that compile and pass on the unmodified program. Report any that do not, explain why, and exclude them from your coverage runs. A test that fails because it reveals a real defect in JFreeChart is a finding worth reporting.

3. **Critique.** Review the tests yourself, including their assertions. Ask the agent to revise them if necessary. Keep the original and revised versions of at least one test if you make a substantive change.

Terminology and measurement rules

- **Developer-provided tests:** tests included in the JFreeChart archive.
- **Randoop tests:** the regression tests generated by Randoop, excluding the developer tests. Randoop's error-revealing tests are reported in Task 3 but excluded from coverage measurements.
- **AI tests:** tests written or revised by the AI agent, excluding the other suites.
- **Combined tests:** the union of specified suites. Combined coverage is not the sum of their percentages.
- **Test oracle:** the check that determines whether a test's observed behavior is correct. Executing a method is not necessarily checking its result.
- **Incremental coverage:** The additional coverage obtained by adding one test suite to another. For example, if developer-written tests cover 60% of the lines and the combined developer-written and Randoop tests cover 75%, Randoop's incremental line coverage is 15 percentage points.

For all comparisons, use the same JFreeChart version, instrumented classes, and coverage metric. Measure the developer suite, Randoop suite, AI suite, and relevant combinations *separately*; otherwise you cannot attribute coverage to the correct suite. Report line and branch coverage (from `jacoco.csv`; see the Technical Hints), not instruction coverage. An unavailable branch metric (for example, a class with no branches) is not the same as 0% coverage. Do not edit production code for coverage experiments, except for the temporary faults requested below.

Your Tasks

Your report must answer two short questions and then the 13 numbered tasks below. Include figures, tables, commands, and short code excerpts where requested. A result that contradicts your expectations is still a result.
Begin your report by answering these two questions, labeled Q1 and Q2:

Q1. In 100 words or fewer, how does Randoop's feedback-directed random test generation work, and where do the assertions in its regression tests come from? (See the Randoop user manual and Pacheco et al., "Feedback-directed Random Test Generation," ICSE 2007.)

Q2. In 100 words or fewer, what is a test oracle? Explain why a test suite with 100% line coverage can still miss a bug.

Then complete the following tasks, numbering your responses 1–13:

1. **Choose your classes.** Create your own `classes.txt` containing 8 fully qualified JFreeChart classes from at least three packages. Include at least two classes you expect to be difficult for Randoop to test. In 150 words or fewer, say what classes you chose and explain why you chose these classes and what you expect to be easy or hard to test. Do not choose an interface or abstract class as one of your difficult classes merely because it cannot be instantiated directly.
2. **Establish the developer baseline.** Run the developer-provided JFreeChart tests and generate a JaCoCo report. Report the total number of tests run, any failures, overall line and branch coverage, and the line and branch coverage of each of your selected classes. Include a readable excerpt or screenshot of the coverage report. This run is also the developer configuration for Tasks 4 and 9; save its `jacoco.csv` and Maven output (see Deliverables) so that you do not need to rerun it.

3. **Generate Randoop tests.** Run Randoop for at least 120 seconds, unless it fails before that limit. Give the exact command, configured limit, elapsed time, and number of generated tests. Compile and execute the generated tests. Report regression and error-revealing tests separately (see the Technical Hints): how many of each were generated, and how many regression tests pass, fail, or cannot run, and how you handled them. Do not silently delete failing tests. Inspect at least two error-revealing tests (if Randoop generated any) and explain whether each indicates a setup issue, a generated oracle problem, or a possible defect.

4. **Compare coverage.** For each of the classes you chose, measure line and branch coverage for the developer-only, Randoop-only, and developer-plus-Randoop suites. Give a table of results for your selected classes and their aggregate. Compute Randoop's *incremental* contribution beyond the developer suite in percentage points. Identify one class where the combined suite improves coverage over the developer suite and one where it does not. If every class improves, or none does, say so. Explain how you excluded the developer tests when measuring Randoop-only coverage.

5. **A class Randoop tests well.** Select a class from your list where Randoop achieves higher line coverage than the developer tests. If no class on your list qualifies, choose the class with the largest incremental Randoop line coverage, say so, and answer the same questions. Inspect two generated test methods that exercise this class. For each, explain what behavior of the class the tests actually checked. Compare these tests with a developer-written test for the class. If no developer test targets the class directly, use the developer test that covers it most, and say so. What are Randoop's tool-generated tests testing that the developer-written tests did not? Why is Randoop more likely to generate such a test? What do you think of the quality of the tool-generated and developer-provided tests? The readability? Suppose a test failed. Would the test's failure help you to find the bug?

6. **A class Randoop tests poorly.** Select a class from your list, different from the one in Task 5, with relatively low Randoop line coverage. An interface or abstract class does not count as a poorly tested class merely because it cannot be instantiated directly. Explain why the tool may struggle with it, referring to constructors, object state, required call sequences, or inputs as appropriate. Identify a specific line or branch missed by the Randoop tests. Also answer Task 5's questions about test quality, readability, and whether a failure would help you find the bug, and offer a hypothesis for why Randoop was unable to produce a better test: bring in your knowledge of how Randoop works. If Randoop generated no tests that exercise this class, say so and explain why.

7. **Plan and generate AI tests.** Use your AI agent and the Plan–Generate–Critique process described above. Target the same classes and, in particular, code left uncovered by the developer-plus-Randoop suite. Include your initial no-code prompt, a brief account of the agent's plan and your criticism, the number of generated test files and methods, and the outcome of compiling and running them. Show one example where the agent changed its approach after your feedback. If it did not change its approach, explain what happened.

8. **Inspect the agent's work.** Inspect the helper code, scripts, or commands the agent created or ran. Did you know your agent created them? Describe what they do and how they decide that a test passes. Identify one assumption or weakness that could allow an incorrect behavior to pass. If the agent created no separate scripts, inspect its test code and commands instead. State what you changed to make the tests run, if anything, and why.

9. **Measure AI coverage.** Generate a JaCoCo report for the AI tests alone and for the developer-plus-Randoop-plus-AI combination. Compare all five configurations in one table: developer, Randoop, developer-plus-Randoop, AI, and all combined. Report line and branch coverage using the same scope.

Identify one line or branch reached by the AI suite that the developer-plus-Randoop suite did not cover, if one exists. If there is no improvement, explain that result. Do not equate a coverage increase with better assertions.

10. **Audit AI-generated tests.** Do all three:

    (a) **Manual fault injection.** Choose three methods, in your selected classes, exercised by the AI tests. For each, make one small, plausible, temporary fault yourself (for example, an off-by-one error or a reversed comparison). Each fault must compile and must actually change the method's behavior. Run *only* the AI tests. Give the file and location, a one-line description or diff, whether the change caused a test to fail, and the failing test's name. Restore the original code after each trial. You must select the faults yourself; the agent may help run the tests but may not select the faults. Did the tests help you find the faults?

    (b) **Assertion audit.** Examine ten AI test methods, or all if fewer than ten. Spread them across at least three of your classes (or across all the classes the AI tests cover, if fewer than three), and do not pick only the strongest or weakest tests. Classify each as having a meaningful behavioral assertion, a weak or missing assertion, or an expected value that mirrors the implementation (for example, a value copied from what the code currently returns, or computed with the same formula the code uses, rather than derived independently). Create a table of the number of methods that fall into each category and give one example of each category. How did you decide what counted as a meaningful assertion? A weak assertion? If categories overlap, explain how you resolved the overlap.

    (c) **Verdict.** In 3–5 sentences, explain what the fault injections and assertion audit reveal that the coverage report does not.

    If a fault is not detected, that is not a grading failure. Your explanation matters.

11. **Process ledger.** For each phase (Plan, Generate, Critique), report the elapsed time and your active time, and explain briefly whether it was useful. If you skipped a phase, report zero rather than inventing an activity. Quote your most useful and least useful prompts, and explain what made them useful or unhelpful. Identify one point at which you checked the agent's work instead of accepting its answer.

12. **Compare Randoop and AI.** In a small table, compare the tools on setup effort, elapsed generation time, your active time, coverage, readability, and usefulness of assertions. Give at least one strength and one limitation of each tool, supported by your experiments. Discuss when you would choose one over the other. When comparing coverage, account for differences between the suites, such as the number of tests each tool generated and the effort each required.

13. **Trust and final judgment.** In no more than 300 words total, discuss your trust in the generated suites. For each tool, identify one decision where you checked or relied on its output, and rate your perceived trust before and after the assignment on a scale from 1 (very low) to 5 (very high). Use evidence from your assertion audit, fault injections, and coverage comparisons to say whether that trust was justified. Describe one feature of a tool or its presentation that might make a user over-trust or under-trust it without changing the quality of its tests. Finally, state what evidence you would require before retaining these tests in a long-lived project.

Deliverables

You will submit, on Canvas, a zip file containing:

1. Submit a `report.pdf` containing:

   (a) Your name, whom you worked with, and what AI tools you used.

   (b) Your answers to questions Q1 and Q2 (see Your Tasks).

   (c) Your responses to Tasks 1–13, numbered 1–13 to match the task list above, including the requested tables, coverage figures, code excerpts, and explanations. Figures must have readable labels; explain relevant patterns in text rather than relying only on color.

   (d) A short reproducibility appendix: operating system, Java, Maven, Randoop, JaCoCo, and AI tool versions (where known); your `classes.txt` contents; the exact commands you ran; any changes to the supplied build; and enough information to reproduce the reported coverage configurations.

   (e) An AI-use appendix naming the tools used for code, tests, analysis, or writing; the tasks they assisted with; the significant prompts or interactions; and what you accepted, rejected, or changed. You need not paste an entire long chat transcript, but include enough evidence to document the use clearly.

2. Submit your `classes.txt` and, if you changed it, your final `pom.xml`.

3. Submit the tests, as `.java` source files: all Randoop-generated tests (regression and error-revealing), Randoop's console output, the final AI tests, and the original version of at least one AI test you substantively revised in the Critique phase (if you revised any).

4. Submit all scripts you wrote, and all helper code or scripts the AI agent created, to build, run, or measure the tests.

5. Submit the raw coverage output for each of the five configurations (developer, Randoop, developer-plus-Randoop, AI, and all combined): the `jacoco.csv` file JaCoCo produced and the Maven console output of that run.

6. For each of the three injected faults (Task 10), submit the diff of the fault and the console output of running the AI tests on the faulty code.

Organize your zip file as follows:

```
lastname_firstname.zip
  report.pdf
  classes.txt
  pom.xml                  (only if you changed it)
  scripts/                 (your scripts and the agent's helper code)
  tests/
    randoop/               (Randoop .java files and console output)
    ai/                    (final AI .java files)
    ai-original/           (original versions of revised AI tests)
  coverage/
    developer/             (jacoco.csv and Maven output)
    randoop/
    developer-randoop/
    ai/
    all/
  faults/
```

```
fault1/                 (diff and AI test output)
fault2/
fault3/
```

Do not include the JFreeChart project directory, compiled `.class` files, HTML coverage reports, or build directories.

If a tool does not run, report the failure, the relevant error, your attempts to fix it, and any partial results. An accurate account of an incomplete experiment is preferable to fabricated data.

Grading

The assignment is worth 100 points. **If your zip file is missing the report, the generated tests, or the raw coverage output, you will get a grade of 0 for the corresponding part(s), unless your report explains that a tool failed to produce them.**

We will grade the correctness of the measurements, the evidence you provide, and the quality of your interpretation. There is no preferred coverage percentage or preferred winner between Randoop and AI.

A correct figure with an incomplete description, or a well-supported analysis where a tool failed, may receive partial credit. Claims about test quality must be supported by the actual assertions, faults, or behavior inspected. Good-faith work that falls short of a complete experiment can still earn credit; unsupported or invented measurements cannot.

We expect to grade the assignment approximately with this rubric:

5pt: The report contains your name, the names of the students you worked with (if any), and the AI tools you used, and its AI-use appendix documents which tools assisted with which tasks, the significant prompts, and what you accepted, rejected, or changed.

5pt: Q1: The description of how Randoop generates tests and where its regression assertions come from is correct, is in your own words, and is 100 words or fewer.

5pt: Q2: The explanation of what a test oracle is and why 100% line coverage can still miss a bug is correct, is in your own words, and is 100 words or fewer.

5pt: The reproducibility appendix describes the environment, commands, and changes to the supplied build in enough detail that we could reproduce the results, and the submitted scripts and `pom.xml` are the ones used.

5pt: The zip file contains the Randoop and AI tests, the raw coverage output for all five configurations, and the fault diffs and outputs, following the required directory layout. The coverage numbers and test counts in the report are consistent with this raw output.

4pt: Task 1 (choose your classes): `classes.txt` lists 8 fully qualified JFreeChart classes from at least three packages, including at least two expected to be difficult for Randoop. The explanation of the choices and expectations is 150 words or fewer.

5pt: Task 2 (developer baseline): the report gives the number of tests run, any failures, overall line and branch coverage, and the line and branch coverage of each selected class, with a readable excerpt or screenshot of the JaCoCo report.

5pt: Task 3 (Randoop tests): the report gives the exact command, the configured time limit (at least 120 seconds), the elapsed time, and the number of generated tests, reporting regression and error-revealing tests separately, with the number of regression tests that pass, fail, or cannot run. Failing tests are not

silently deleted, and at least two error-revealing tests (if any) are explained as a setup issue, an oracle problem, or a possible defect.

5pt: Task 4 (compare coverage): a table gives line and branch coverage for the developer-only, Randoop-only, and developer-plus-Randoop suites for each selected class and in aggregate (computed from covered and total counts). Randoop's incremental contribution is given in percentage points, and the report identifies a class where the combined suite improves coverage over the developer suite and one where it does not (or says that every class, or none, improves), and explains how the developer tests were excluded for the Randoop-only measurement.

7pt: Task 5 (a class Randoop tests well): the class is one where Randoop achieves higher line coverage than the developer tests (or, if none exists, the class with the largest incremental Randoop line coverage, and the report says so). The analysis explains what two generated test methods actually check, compares them with a developer-written test, and discusses what Randoop tests that the developers did not and why, test quality, readability, and whether a failure would help find the bug.

8pt: Task 6 (a class Randoop tests poorly): the class differs from Task 5's and has relatively low Randoop line coverage, and the report identifies a specific missed line or branch. The hypothesis for why Randoop struggled is plausible and grounded in how Randoop works (e.g., constructors, object state, call sequences, or inputs), and the report reflects on the tests as in Task 5.

8pt: Task 7 (plan and generate AI tests): the report includes the initial no-code prompt, an account of the agent's plan and the student's criticism, the number of generated test files (at most 20) and methods, the outcome of compiling and running them, and an example of the agent changing its approach after feedback (or an explanation of why it did not).

5pt: Task 8 (inspect the agent's work): the report describes the helper code, scripts, or commands the agent created or ran (or its test code and commands), how they decide that a test passes, one assumption or weakness that could let incorrect behavior pass, and any changes made to make the tests run.

6pt: Task 9 (measure AI coverage): line and branch coverage are reported in one table, using the same scope, for all five configurations (developer, Randoop, developer-plus-Randoop, AI, and all combined). The report identifies a line or branch reached by the AI suite that the developer-plus-Randoop suite did not cover (or explains why there is none) and does not equate a coverage increase with better assertions.

9pt: Task 10 (audit AI-generated tests):

- 4pt: Manual fault injection: three faults, selected by the student, in methods of the selected classes exercised by the AI tests; each fault compiles and changes behavior. For each, the report gives the file and location, a description or diff, whether a test failed, and the failing test's name. Each fault was reverted.
- 3pt: Assertion audit: ten AI test methods (or all, if fewer) are classified, with a table of counts, an example of each category, and an explanation of the classification criteria and of how overlaps were resolved.
- 2pt: Verdict: 3–5 sentences explaining what the fault injections and assertion audit reveal that the coverage report does not, supported by the results.

4pt: Task 11 (process ledger): for each phase, the report gives the elapsed time and active time and whether the phase was useful (zero for skipped phases). It quotes the most and least useful prompts, explains why, and identifies one point at which the student checked the agent's work.

4pt: Task 12 (compare Randoop and AI): a table compares the tools on setup effort, elapsed generation time, active time, coverage, readability, and usefulness of assertions. The report gives at least one strength and one limitation of each tool, supported by the experiments, discusses when to choose each, and accounts for differences between the suites, such as the number of tests and the effort required.

5pt: Task 13 (trust and final judgment): the discussion is 300 words or fewer. For each tool, it identifies a decision where the student checked or relied on its output and rates trust before and after (1–5). It uses evidence from the assertion audit, fault injections, and coverage comparisons, describes a tool or presentation feature that might cause over- or under-trust, and states what evidence would be required to retain the tests in a long-lived project.

Appendix: Technical Hints

Java version: We recommend Java 17. The supplied build compiles JFreeChart for Java 8 and uses JaCoCo 0.8.12. If you see JaCoCo errors such as `Unsupported class file major version` on a newer Java, switch to Java 17 or 21.

What the edited build does: The supplied `pom.xml` adds the JaCoCo plugin, which instruments the code when the tests run and writes a report during the `test` phase, and a JUnit 4 dependency, so that Randoop's tests compile. The developer tests use JUnit 5 (Jupiter). Maven's test runner (Surefire) is configured to run only classes whose names end in `Test`.

Running Randoop: Compile JFreeChart first (`mvn compile`), then get its dependency classpath with

```
mvn dependency:build-classpath -Dmdep.outputFile=cp.txt
```

and run Randoop with `target/classes`, the contents of `cp.txt`, and `randoop-all-4.3.4.jar` on the classpath. The options `--junit-output-dir` and `--junit-package-name` control where Randoop writes its tests and which package they are in.

Randoop's output: Randoop writes two kinds of tests. *Regression tests* (`RegressionTest0.java`, `RegressionTest1.java`, ...) record the current behavior and should pass on the unmodified code. *Error-revealing tests* (`ErrorTest0.java`, ...) are expected to fail: each one shows behavior that Randoop considers a likely bug, such as a violated contract. Randoop also writes suite classes, `RegressionTest.java` and `ErrorTest.java`, that run all the numbered classes of each kind. Count, and report on, the two kinds separately. Inspect the error-revealing tests for Task 3, but keep them out of your coverage measurements (e.g., move them out of `src/test/java` after inspecting them); submit them anyway.

Running JUnit 4 tests: Randoop generates JUnit 4 tests. The supplied build compiles them but will not run them until you add the JUnit Vintage engine as a test dependency: group `org.junit.vintage`, artifact `junit-vintage-engine`, version 5.12.2 (matching the Jupiter version in `pom.xml`). The Vintage engine requires JUnit 4.12 or later, so also change the `junit:junit` version from 4.11 to 4.13.2. Report these changes.

Running only some tests: Surefire's `-Dtest` option selects which test classes run, and JaCoCo then measures only those tests. For example,

```
mvn clean test -Dtest='RegressionTest' -Dsurefire.failIfNoSpecifiedTests=false \
  -Dmaven.test.failure.ignore=true
```

runs only the Randoop regression suite. Because `RegressionTest` runs all the numbered classes, selecting it together with `RegressionTest*` runs every Randoop test twice. Because the AI tests' names end in `AITest`, you can select them with `-Dtest='*AITest'`. To measure the developer tests alone, run them before adding any generated tests (Task 2), or temporarily move the generated tests out of `src/test/java`. Always check the `Tests run:` line in the Maven output to confirm that the intended tests, and only those, ran.

Keep coverage runs separate: JaCoCo execution data can accumulate between test runs, and `mvn clean` deletes the previous report. Start each configuration with `mvn clean test ...`, save the console output (e.g., by appending `| tee maven-output.txt`), and copy `target/site/jacoco/jacoco.csv` into your submission directory before the next run.

Failing tests can prevent the coverage report: If any test fails, Maven reports `BUILD FAILURE` and may stop before JaCoCo writes its report, leaving `jacoco.csv` missing (or, without `clean`, left over from an earlier run). For coverage runs, add `-Dmaven.test.failure.ignore=true`, which records failing tests without stopping the build. After every run, check that `jacoco.csv` exists and was just written, and record how many tests failed.

Reading coverage: JaCoCo's HTML report shows a percentage for instructions and branches but only missed and total counts for lines. Use `jacoco.csv` instead: it has one row per class, with columns `LINE_MISSED`, `LINE_COVERED`, `BRANCH_MISSED`, and `BRANCH_COVERED`. Line coverage is $\mathit{covered}/(\mathit{covered}+\mathit{missed})$, and similarly for branches. Nested classes appear as separate rows (e.g., `Outer.Inner`); decide whether to include them in a class's numbers, say which you chose, and be consistent.

Use the same scope: For per-class comparisons, use the same class list and coverage metric throughout. When aggregating, use covered and total lines or branches, not the unweighted mean of class percentages.

No branches is not 0%: If a class has no branches (both branch columns are 0), report its branch coverage as N/A and do not treat it as zero.

A passing generated test is not necessarily a correct test: Randoop records current behavior in regression assertions. AI may do the same. Check expected results against documented behavior or an independent argument when possible.

Restore injected faults: Before you start, put the unzipped project under version control (`git init`, then commit everything). Make each fault separately, save it with `git diff > fault.diff`, run the AI tests, and restore the code with `git checkout -- src/main` before the next experiment. Use `git status` to confirm that your final coverage runs use unmodified production code.

Long-running or failed experiments: Record the actual commands, runtime, errors, and partial output. Do not report a run as successful solely because a command exited without a visible error.