Module 4 – Debugging and Refactoring with AI

CS5013 – Programming with AI
Lecture Notes

V. Krishna Nandivada
Department of Computer Science and Engineering, IIT Madras

1 The state you are in now

You have written code with AI (Modules 1–3), thought about how to prompt it well (Module 2), and let it autocomplete new code (Module 3). This module is about the other 90% of software engineering: making existing code work correctly, and making it easier to change. We call these activities *debugging* and *refactoring*, and they are the day-to-day of anyone who has been paid to write code for more than a year.

The good news: AI is spectacularly useful here. The bad news: it is also spectacularly *easy* to misuse. This chapter's job is to give you a mental model that keeps you on the useful side of that line.

2 Debugging with AI, one story at a time

Let me tell you the story of two students. Student A gets a null-pointer exception, copies the stack trace into chat, gets back an answer that says "add a null check on line 42", adds one, and moves on. Student B gets the same null-pointer exception, copies the trace *and* the method *and* the calling method into chat, asks "what most plausibly produced this null?", reads the reply, realises the null came from a config file that was never populated, and fixes the config code.

Both students shipped. Student A shipped a bug that will bite six months from now, because the null was symptomatic of a broken config path and the null check just silenced it. Student B shipped a fix.

The difference is not intelligence. It is workflow.

Explain, then fix

The first rule of AI-assisted debugging: *make the model explain before it patches.* When you ask "fix this", you are asking the model to skip the diagnosis step and jump to a code change. When you ask "what is the most likely cause?", you get an explanation you can check against your own reading of the code. If the explanation is wrong, you catch it before applying the patch.

Concretely, the two-prompt version is:

1. "Here is the code and the error. Explain what caused it and rank the top three most likely root causes."

2. "Given root cause #1 is correct, write the smallest patch that fixes it. Show only the diff."

This is slower than a single "fix this" prompt. It is also much less likely to produce Student A's outcome.

Context is your dial

The model can only see what you paste. For a simple bug, that means the failing method plus its immediate callers. For a subtle bug, that means the failing method plus the class definitions of every type it touches.

Do not paste your whole project. Extra code is not free context – it is noise that the model has to sort through, and it dilutes the signal from the code that actually matters. The right amount of context is "the smallest set of files that contains the bug". Finding that set is itself a debugging skill.

Habit worth cultivating

Before you ask the model anything, ask yourself: *where do I believe the bug is*? Then paste only that region. If your belief turns out to be wrong, widen the paste. Do not start wide.

Minimal repros and diff-driven diagnosis

Two techniques transfer directly from pre-AI debugging.

Minimal reproducing example. Halve the input until the bug disappears; the boundary is your minimal repro. Ask the model to help simplify: "given this failing test, remove lines that do not affect the failure". The model is good at this. And a minimal repro often *is* the fix: at 5 lines, the bug is usually visible.

Diff-driven diagnosis. When the bug is new, the change is small. `git diff HEAD~` usually contains the culprit. Paste *only* the diff – not the full file – and ask which line most plausibly caused the regression. A 20-line diff is dense with signal; a 2000-line file is not.

Ranking hypotheses

When a bug has three plausible causes, do not fix all three – ask the model to rank them. A useful prompt shape:

```
Given this failure and this code, list the three most likely
root causes in order of probability, and for each, name one
experiment that would confirm or eliminate it.
```

The output is a small decision tree you can walk yourself. You do one experiment, cross off one branch, do the next. This is much cheaper than blindly applying three "might fix it" patches – which is what happens when you ask the model to fix without ranking first.

Rubber-ducking with a model

The classic debugging technique is to explain your bug to a rubber duck, line by line. Half the time you find the fix while explaining – because you were carrying the wrong mental model, and articulating it forced you to see the gap.

AI works the same way, but the duck talks back. You type out your current understanding of the failure; the model asks clarifying questions; you answer them, and more often than not you catch the bug inside your own answer. The model does not need to be smart for this to work. It only needs to be a good listener that occasionally asks "are you sure that `X` is what you think it is?". Any modern chat assistant is that.

When AI debugging is the wrong tool

Three categories where you should not lean on AI:

- **Time-dependent bugs.** Race conditions, memory-ordering bugs, deadlocks. The model cannot see time. It can guess plausibly at what a race might look like, but it cannot confirm your race.
- **Environment-dependent bugs.** Bugs that only reproduce in production with real load, or with a specific customer's data. The model cannot see production.
- **Specification disagreements.** You think the code is wrong; a reviewer thinks the code is right; the argument is about what the code *should* do. No amount of AI can resolve a stakeholder disagreement.

For all three, AI is at best an accelerator on the parts you *can* see. Do not let it pretend to see more.

3 Refactoring, one move at a time

Refactoring is the art of *changing the shape of code without changing what it does.* That definition is short but load-bearing: the "without changing what it does" half is what makes refactoring safe. And it is only safe if you have tests. Without tests, "refactor" is a euphemism for "rewrite and hope".

The Fowler book [1] catalogues about 100 refactor moves. You do not need to memorise them. You need to know four:

- **Extract method**: turn a block of code into a named helper.
- **Inline method**: pull a helper back into its caller.
- **Rename**: change a variable, method, class, or file name to something that better describes what it does.
- **Change loop shape**: turn a for-each into a stream, or vice versa.

These are the four AI is good at. Every other refactor should first be decomposed into a sequence of these four.

Complexity in two flavours

You cannot refactor toward a target you cannot name. Two metrics give you names.

Cyclomatic complexity (McCabe, 1976): count the branches (`if`, `case`, `catch`, `&&`, `||`, `? :`) in a method and add 1. Under 5 is easy; 6–10 is acceptable; over 10 is a smell.

Cognitive complexity (Campbell, SonarSource, 2018): same idea, but nesting increments the count more than sibling branches. This matches how humans actually experience code: deeply nested `if` statements are harder to hold in your head than flat `if`–`return`–`if`–`return` sequences.

```
// Cyclomatic 5, cognitive high (nesting is expensive).
public String describe(int n) {
    if (n < 0) {
        if (n < -100) return "very negative";
        return "negative";
    } else if (n == 0) {
        return "zero";
    } else {
        if (n > 100) return "very positive";
        return "positive";
    }
}
```

```java
// Cyclomatic 5, cognitive low (guard clauses, no nesting).
public String describe(int n) {
    if (n < -100) return "very negative";
    if (n < 0) return "negative";
    if (n == 0) return "zero";
    if (n > 100) return "very positive";
    return "positive";
}
```

Both methods have the same behaviour; both have the same cyclomatic score; but the second is easier to read. This is what you should ask an AI to do for you.

Asking AI for a refactor: a prompt shape that works

The best refactor prompts have four parts:

1. The code (small, focused).
2. The refactor you want ("extract this branch into a helper", "flatten this if/else pyramid into guard clauses").
3. Constraints ("keep the public signature", "do not add new dependencies", "do not delete comments").
4. Output format ("show only the diff", "show the whole method rewritten").

The output-format bullet matters more than students expect. A diff is easier to review than a rewritten method. A rewritten method is easier to compile-check than a diff. Choose based on what you will do next.

Failure mode to watch: silent behaviour change

The single most common AI refactor bug is a refactor that *looks* equivalent but is not. Types change (`int` to `long`); early returns get lifted or dropped; null-handling gets "simplified" in a way that now throws where the original returned. Run the tests after every AI refactor. Every one.

Duplication removal: shape versus knowledge

DRY – Don't Repeat Yourself – is one of the most-quoted and least-understood principles in software. The original meaning (Hunt & Thomas, [2]) is: *every piece of knowledge should have a single, unambiguous representation in a system.* The version most students hear is: "don't write the same three lines twice".

Those two are not the same.

If two blocks of code look similar but encode different *knowledge*, deduplicating them creates a false abstraction that breaks the next time the two blocks need to diverge. The three tax functions:

```java
public double gstIndia(double p) { return p * 0.18; }
public double vatUk (double p) { return p * 0.20; }
public double gstSg (double p) { return p * 0.09; }
```

have the same *shape*. They do not encode the same knowledge: the rates come from different tax authorities and change independently. Extracting a common helper is only a good idea if a policy *across* the three arises – say, "round to the nearest paisa for India, nearest penny for the UK".

When you ask an LLM to remove duplication, prefix the request with: "*First tell me whether these blocks encode the same knowledge. Only then propose a refactor.*" The pause forces the judgment call.

Where AI hurts: three concrete failure modes

The one already flagged (silent behaviour change) is the loudest. Three more are common enough to name.

Import churn. Ask "tidy the imports" and you may get a wildcard sweep – `import java.util.*;` – where the project's style guide forbids wildcards. The diff becomes 80% import noise; checkstyle fails CI. The reverse also happens: a fine wildcard replaced by ten specific imports plus an unrequested reordering that touches every line. Fix: constrain the prompt ("do not touch import statements"), or reject the change.

Unwanted modernisation. An ordinary for-each loop gets "modernised" to a stream chain even when the loop was fine. Sometimes it changes behaviour. The canonical trap is that `Stream.toList()` returns an *unmodifiable* list, so caller code that later mutates the result throws `UnsupportedOperationException` at runtime. Fix: state "do not modernise" in the prompt, and always run tests.

Loss of comments. The comments the assistant tends to remove are the ones that mattered – because the refactored code is "clear enough". A comment that explained *why* a threshold is 800 (a compliance memo) vanishes; six months later someone bumps it to 850 "because the tests still pass". Fix: include "preserve all comments" as an explicit prompt constraint, and diff the comment lines before accepting the change.

4 Combining static analysis with AI

Static analysis is the boring, powerful cousin of AI. Tools like SpotBugs, PMD, and Error Prone read your Java code without running it and flag patterns that are known to be buggy: `equals` without `hashCode`, unchecked casts, unused imports, integer overflow in loop counters, and hundreds more.

They are boring because they run a rule engine, not a language model. They are powerful because the rules are *high precision*: when the tool says you have EQ_UNUSUAL, you almost certainly do.

LLMs and static analysers complement each other:

- Static analysers are high precision, low recall for *semantic* bugs. They find rule violations, not intent violations.
- LLMs are the reverse: they read intent, but they cannot run a fixed rule set at industrial scale.

The workflow that works:

1. Run the static analyser. Get the report.
2. Feed each finding to the LLM together with (a) the rule name, (b) the flagged code, (c) two or three lines of surrounding context.
3. Ask: "*Given the rule and the code, what is the intended fix? Show the smallest patch that satisfies the rule without changing behaviour.*"
4. Apply. Run tests.

The LLM's job is to translate "EQ_UNUSUAL at line 27" into "here's what the rule wants and here's the specific patch". The static analyser's job is to guarantee that the finding is worth your time in the first place.

Triage before you fix

On a legacy project, the first SpotBugs run will produce hundreds or thousands of findings. Do not fix them all. Ask the LLM to cluster the findings by rule, rank the clusters by frequency, and propose a per-cluster fix strategy. A weekend of triage is worth more than a month of unfocused fixing.

Handling false positives

Every static analyser produces false positives. The old workflow was to slap `@SuppressWarnings` on the line and forget. The better workflow with AI:

1. Ask: "Is this finding a true positive? Explain why."
2. If yes, fix.
3. If no, suppress *and* write a one-line comment explaining why the tool is wrong here.

The comment is the audit trail. The next reader (and the next static-analysis update) can decide whether the suppression is still justified. Without it, suppressions accumulate silently and no one remembers which are legitimate.

Automated fixers

Some static-analysis tools ship rule-based automatic fixers. SpotBugs has *Contrib* rules; PMD has *Auto-fix* for a subset; Error Prone offers *Refaster templates* that rewrite matching patterns.

The trade-off is mechanical. When a finding has a single canonical fix – add `@Override`, replace `==` with `equals()` on strings, insert a missing `break` in a switch fall-through – the automated fixer is fine. When the fix requires understanding *intent* ("did the author mean assignment or comparison here?"), route to the LLM instead. The two approaches are complementary, not competitive.

IDE integration

IntelliJ IDEA and VS Code (with the Java pack) both surface SpotBugs and PMD findings inline in the editor. Copilot Chat can be invoked on a finding with a "fix this warning" action; the chat receives the tool name, the rule ID, the message, and the current file, and produces a diff you can accept, reject, or edit.

This is the workflow you will use in the second hands-on session: open a Java file with a known set of SpotBugs findings; walk through each finding; ask the assistant to explain and propose a patch; apply the patch and re-run the analyser.

5 Edge cases and robustness

Ask a student to write a method `int average(int[] a)` and they will happily produce it. Ask them what should happen when the array is empty, contains `Integer.MAX_VALUE`, is null, or is a million entries long, and you will get five puzzled looks.

Enumerating edge cases is a task humans are bad at and LLMs are good at. It is one of the highest-leverage prompts in this whole course:

```
For the following method, list every edge-case input I should
test. Group by category: boundary values, null inputs, resource
limits, concurrency. For each, name the specific input value or
```

```
condition, and one line about the expected behaviour.

public int average(int[] a) { ... }
```

You will get a categorised list, most of which you would have missed. Some of the items will not apply to your setting. That is fine – filtering the list is much cheaper than generating it from scratch.

Your turn: enumerate the edge cases

Try this in class before reading the LLM's answer. The method is:

```
/** Days in month, Gregorian calendar. month is 1-indexed. */
int daysInMonth(int month, int year);
```

Take three minutes and list every edge-case input you would put in a test suite. A reasonably complete list:

- **Month boundaries**: 1, 12, 0, 13, negative, `Integer.MIN_VALUE`, `MAX_VALUE`.
- **Leap-year rules**: 2020 (leap: divisible by 4), 1900 (*not* leap: divisible by 100, not by 400), 2000 (leap: divisible by 400), 2100 (not leap).
- **Year boundaries**: year 1, year 0 (does not exist in the proleptic Gregorian calendar), negative year (BCE), `MAX_VALUE`.
- **Calendar reform**: 1582 – 4 October was followed by 15 October; pre-1582 dates were on the Julian calendar. Which does the specification intend?
- **Return-type sanity**: every valid month must return 28–31. Is there any path that could return 0 or 32?
- **Locale**: some Orthodox churches still use the Julian calendar for liturgical calculations. Does the doc say "Gregorian" or just "calendar"?

Most students find four or five categories on the first pass; almost none find the 1582 calendar reform or the year-0 quirk. That gap is the whole point: this is exactly what LLMs are good at enumerating and humans are not.

The natural next step is to convert the list into tests. Two prompts do it:

1. "*Write JUnit 5 tests for each edge case above, organised by category, with descriptive names.*"
2. "*Write a property-based test using jqwik that exercises the invariant 'average of a non-empty array is between min and max'.*"

Robustness beyond edge cases

Once you have unit tests for the enumerated edge cases, three layers of robustness testing sit on top of them. Each has an established Java tool, and LLMs can generate scaffolding and invariants for all three.

Fuzzing in detail (Jazzer)

The core idea of fuzzing is small enough to write in plain Java:

```
Random rng = new Random(0);
for (int i = 0; i < 100_000; i++) {
    byte[] bytes = new byte[rng.nextInt(200)];
    rng.nextBytes(bytes);
    String s = new String(bytes, StandardCharsets.UTF_8);
    try {
        JsonParser.parse(s);
    } catch (JsonException expected) {
        // declared by the API -- fine
    }
    // any OTHER Throwable escaping = a bug worth reporting
}
```

Jazzer (`github.com/CodeIntelligenceTesting/jazzer`) does the same thing, but two features make it dramatically more useful. First, it is *coverage-guided*: after every input, it observes which branches the program hit and mutates future inputs toward branches it has not reached yet. Second, it *shrinks* failures. When a random 12KB input causes a crash, Jazzer minimises it to the smallest crashing input – often a few bytes – so you can see exactly what triggered the bug. It also keeps a *corpus* of interesting inputs on disk across runs, so a restart does not lose progress.

Fuzzing is at its best on parsers, deserialisers, and protocol handlers – code whose input space is too large to enumerate but whose contract is easy to check: "if the input is valid, `parse` returns; if invalid, it throws `ParseException` – and nothing else".

Mutation testing in detail (PIT)

PIT (`pitest.org`) mutates your code slightly (a *mutant*) and re-runs your tests. A strong test suite *kills* every mutant – some test fails because of the change. A mutant that survives means your tests are not actually checking the line they appear to check.

Given the original:

```
if (score >= PASS) return "pass";
```

PIT injects mutants like:

```
if (score > PASS) return "pass"; // relational mutator
if (score >= PASS) return ""; // return-value mutator
if (true) return "pass"; // conditional-boundary mutator
```

PIT's report is a mutation score: "72%" means 28% of mutants survived. That 28% is your homework – write tests that would kill them. It is one of the few metrics that reliably distinguishes "lots of tests" from "tests that actually check the code".

Property-based testing in detail (jqwik)

Instead of asserting on *specific* inputs, assert an *invariant* that must hold for every valid input. jqwik generates hundreds; if one fails, it *shrinks* to a minimal counter-example.

```
@Property
void averageIsBetweenMinAndMax(
        @ForAll @Size(min = 1) int[] xs) {
    int avg = average(xs);
    IntSummaryStatistics s = Arrays.stream(xs).summaryStatistics();
    assertThat(avg).isBetween(s.getMin(), s.getMax());
}
```

Roughly 1000 random arrays per run (configurable). If one breaks the invariant, jqwik shrinks to the smallest failing array – typically something like `{Integer.MAX_VALUE, 1}` exposing an overflow in `average`. You write the property; jqwik supplies the inputs; the shrinker gives you a debuggable failure.

What LLMs can and cannot do here

LLMs are excellent at generating fuzz harnesses, mutation targets, and property invariants – all three "code that generates the tests". They cannot *run* these tools; that is your job. Use them as an accelerator, not a replacement.

Contract tests

Two components agree on a wire format. A contract test pins the format down as data, so either side can be checked in isolation – without spinning both up in an integration environment.

The agreed contract is a plain string the two sides share:

```
static final String ORDER_42_JSON =
    "{\"id\":42,\"total\":199}";
```

The provider's test asserts that its output matches the contract:

```
@Test void producesAgreedShape() {
    assertEquals(ORDER_42_JSON, orderService.getOrder(42));
}
```

The consumer's test asserts that it can parse the contract:

```
@Test void consumesAgreedShape() {
    Invoice inv = billingClient.buildInvoice(ORDER_42_JSON);
    assertEquals(42, inv.orderId());
    assertEquals(199, inv.amount());
}
```

Rename `"total"` to `"amount"` on the provider and the provider's own contract test fails – before billing ever sees the drift. The tools in production (Pact, Spring Cloud Contract) automate sharing the pact between two repositories; the essence is what you see above.

Adversarial input generation

Turn the LLM into a first-pass red team:

```
The method below is deployed behind an HTTP endpoint.
Suggest five adversarial inputs a malicious client could send
that would either (a) cause a crash, (b) produce incorrect output,
or (c) use excessive resources. For each, describe the
consequence and the smallest defensive check.

public Response search(String query) { ... }
```

The output will not be a full security review. It will be five plausible attacks with concrete defensive checks – which is often enough to catch the most obvious mistakes before a real security review happens.

6 Wrap-up

You now have four ideas to carry through the rest of the course:

1. AI debugging works when you make the model *explain before it patches*.
2. Refactoring is behaviour-preserving; tests decide; cyclomatic and cognitive complexity are the measuring sticks.
3. Static analysers and LLMs are complementary: rule-checked findings become LLM-explained patches.
4. Edge-case enumeration and adversarial-input generation are two of the most productive uses of AI for code quality.

Assignments 7 and 8 will make you do all four. Assignment 7 leans on the debugging and static-analysis workflows. Assignment 8 leans on refactoring for cyclomatic complexity.

References

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. 2nd edition, Addison-Wesley, 2018.

[2] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. 1st edition, Addison-Wesley, 1999.

[3] Thomas J. McCabe. "A Complexity Measure." IEEE Transactions on Software Engineering, SE-2(4):308–320, 1976.

[4] G. Ann Campbell. "Cognitive Complexity: A New Way of Measuring Understandability." SonarSource white paper, 2018.

[5] Robert C. Martin. *Clean Code*. Prentice Hall, 2008.