Module 4:
Debugging and Refactoring with AI

CS5013 – Programming with AI

V. Krishna Nandivada

Department of Computer Science and Engineering
Indian Institute of Technology Madras

Where we are

- Module 1: what AI coding assistants are and when to reach for them.

Where we are

- Module 1: what AI coding assistants are and when to reach for them.
- Module 2: prompt engineering – getting good outputs on purpose.

Where we are

- Module 1: what AI coding assistants are and when to reach for them.
- Module 2: prompt engineering – getting good outputs on purpose.
- Module 3: code generation and autocompletion – new code from a blank cursor.

Where we are

- Module 1: what AI coding assistants are and when to reach for them.
- Module 2: prompt engineering – getting good outputs on purpose.
- Module 3: code generation and autocompletion – new code from a blank cursor.
- **Module 4 (this module): debugging and refactoring** – working on *code that already exists*.

Where we are

- Module 1: what AI coding assistants are and when to reach for them.
- Module 2: prompt engineering – getting good outputs on purpose.
- Module 3: code generation and autocompletion – new code from a blank cursor.
- **Module 4 (this module): debugging and refactoring** – working on *code that already exists*.
- Module 5 onwards: testing, documentation, LLM APIs, ethics, evaluation.

Debugging with AI: the mental model

Two things “debugging with AI” can mean

- **Explain-this-error.** You paste a stack trace and get an English paragraph back.

Both are useful. They fail in different ways, and the workflows around them are different. We will treat them separately for the rest of this hour.

Two things “debugging with AI” can mean

- **Explain-this-error.** You paste a stack trace and get an English paragraph back.
- **Fix-this-code.** You paste code plus a symptom and get a patch back.

Both are useful. They fail in different ways, and the workflows around them are different. We will treat them separately for the rest of this hour.

Why AI is useful for debugging at all

- Error messages are *terse on purpose* – they were written for compiler authors, not for you.

Why AI is useful for debugging at all

- Error messages are *terse on purpose* – they were written for compiler authors, not for you.
- An AI assistant has seen many thousands of variations of the same error and can translate them into "what actually happened".

Why AI is useful for debugging at all

- Error messages are *terse on purpose* – they were written for compiler authors, not for you.
- An AI assistant has seen many thousands of variations of the same error and can translate them into "what actually happened".
- Debugging is often about *search*: which of a hundred possible causes is the actual one? A language model is a fast search engine over the space of plausible causes.

Example: a Java stack trace to a beginner

```
Exception in thread "main" java.lang.NullPointerException:
  Cannot invoke "String.length()" because "s" is null
    at com.example.Report.summarize(Report.java:42)
    at com.example.Main.main(Main.java:12)
```

To an experienced Java developer: "line 42 tried to call `length()` on a null `s`."

To a beginner: seven lines of unfamiliar words in three colours.

The “explain-this-error” workflow

1. Copy the entire stack trace (*all* the “caused by” chains, not just the top line).

The pause at step 4 is the whole point. A confident wrong diagnosis leads to a confident wrong patch.

The “explain-this-error” workflow

1. Copy the entire stack trace (*all* the “caused by” chains, not just the top line).
2. Paste it into chat with the surrounding code.

The pause at step 4 is the whole point. A confident wrong diagnosis leads to a confident wrong patch.

The “explain-this-error” workflow

1. Copy the entire stack trace (*all* the “caused by” chains, not just the top line).
2. Paste it into chat with the surrounding code.
3. Ask: “*Explain this error in plain English. What is the most likely root cause?*”

The pause at step 4 is the whole point. A confident wrong diagnosis leads to a confident wrong patch.

The “explain-this-error” workflow

1. Copy the entire stack trace (*all* the “caused by” chains, not just the top line).
2. Paste it into chat with the surrounding code.
3. Ask: “*Explain this error in plain English. What is the most likely root cause?*”
4. Read the explanation. Do *not* accept a proposed patch yet. First check the diagnosis matches your understanding.

The pause at step 4 is the whole point. A confident wrong diagnosis leads to a confident wrong patch.

The "explain-this-error" workflow

1. Copy the entire stack trace (*all* the "caused by" chains, not just the top line).
2. Paste it into chat with the surrounding code.
3. Ask: "*Explain this error in plain English. What is the most likely root cause?*"
4. Read the explanation. Do *not* accept a proposed patch yet. First check the diagnosis matches your understanding.
5. Then decide whether to fix by hand or ask for a patch.

The pause at step 4 is the whole point. A confident wrong diagnosis leads to a confident wrong patch.

Why include the surrounding code?

Without the code, the model must guess what "s" referred to. It will guess plausibly – and often wrongly.

With the code, the model can point at the exact line that produced the null and (often) at the line that *should* have populated it.

Rule of thumb: paste the failing method and any method it calls. Do not paste the whole repo – irrelevant code dilutes the signal.

The “fix-this-code” workflow

Shape of the prompt:

1. The code (short, focused – one method if possible).

The “fix-this-code” workflow

Shape of the prompt:

1. The code (short, focused – one method if possible).
2. The failing behaviour (“it throws NPE at line 42” or “it returns 0 for empty input, should return -1”).

The “fix-this-code” workflow

Shape of the prompt:

1. The code (short, focused – one method if possible).
2. The failing behaviour (“it throws NPE at line 42” or “it returns 0 for empty input, should return -1”).
3. The expected behaviour (“it should return the sum of positive elements, and 0 for an empty list”).

The “fix-this-code” workflow

Shape of the prompt:

1. The code (short, focused – one method if possible).
2. The failing behaviour (“it throws NPE at line 42” or “it returns 0 for empty input, should return -1”).
3. The expected behaviour (“it should return the sum of positive elements, and 0 for an empty list”).
4. An explicit constraint list (“keep the method signature”, “do not introduce new dependencies”).

Example prompt

```
The following method throws IndexOutOfBoundsException for
an input of length 1. Fix it. Keep the signature.

public int secondLargest(int[] a) {
    Arrays.sort(a);
    return a[a.length - 2];
}

Expected: return -1 for a length-0 or length-1 array.
```

What a good “fix” answer looks like

Three parts, in this order:

- **Diagnosis.** “`a.length - 2` is -1 when the array has one element, which is out of range.”

If the reply is only the patch, you should be more suspicious, not less. Ask “why does this fix it?” before applying.

What a good “fix” answer looks like

Three parts, in this order:

- **Diagnosis.** “`a.length - 2` is -1 when the array has one element, which is out of range.”
- **Patch.** The rewritten method with the guard clause.

If the reply is only the patch, you should be more suspicious, not less. Ask “why does this fix it?” before applying.

What a good "fix" answer looks like

Three parts, in this order:

- **Diagnosis.** "`a.length - 2` is -1 when the array has one element, which is out of range."
- **Patch.** The rewritten method with the guard clause.
- **Explanation of the patch.** "I added a check `if (a == null || a.length < 2) return -1;` at the top."

If the reply is only the patch, you should be more suspicious, not less. Ask "why does this fix it?" before applying.

Failure modes you will see this module

- **Wrong root cause.** The model diagnoses the symptom (“add a null check”) without identifying the upstream producer of the null.

Failure modes you will see this module

- **Wrong root cause.** The model diagnoses the symptom ("add a null check") without identifying the upstream producer of the null.
- **Symptomatic fix.** The patch silences the exception but leaves the bug (e.g., `try/catch` with an empty block).

Failure modes you will see this module

- **Wrong root cause.** The model diagnoses the symptom ("add a null check") without identifying the upstream producer of the null.
- **Symptomatic fix.** The patch silences the exception but leaves the bug (e.g., `try/catch` with an empty block).
- **Overfitting to the example.** A fix that handles the one input you showed but breaks other cases.

Failure modes you will see this module

- **Wrong root cause.** The model diagnoses the symptom ("add a null check") without identifying the upstream producer of the null.
- **Symptomatic fix.** The patch silences the exception but leaves the bug (e.g., `try/catch` with an empty block).
- **Overfitting to the example.** A fix that handles the one input you showed but breaks other cases.
- **Hallucinated API.** A "fix" that calls a method that does not exist on the type.

Debugging workflows in practice

From symptom to minimal repro

The single most valuable debugging skill is producing a *minimal reproducing example*: the smallest program that still exhibits the bug.

AI can help at every step:

- "Simplify this test case while keeping the failure."

A minimal repro often *is* the fix – once the code is small enough, the bug is visible.

From symptom to minimal repro

The single most valuable debugging skill is producing a *minimal reproducing example*: the smallest program that still exhibits the bug.

AI can help at every step:

- "Simplify this test case while keeping the failure."
- "What is the smallest input that triggers this branch?"

A minimal repro often *is* the fix – once the code is small enough, the bug is visible.

From symptom to minimal repro

The single most valuable debugging skill is producing a *minimal reproducing example*: the smallest program that still exhibits the bug.

AI can help at every step:

- "Simplify this test case while keeping the failure."
- "What is the smallest input that triggers this branch?"
- "Remove any code that does not affect the outcome."

A minimal repro often *is* the fix – once the code is small enough, the bug is visible.

Ranking hypotheses

When a bug has three plausible causes, do not fix all three. Ask the model to rank them.

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

The output is a small decision tree you can walk yourself.

Diff-driven diagnosis

When a bug appears *after* a change, ask about the diff, not the file.

Workflow:

1. `git diff HEAD~1` to isolate the recent change.

Diffs are cheap context. A 20-line diff is much more useful to the model than a 2000-line file.

Diff-driven diagnosis

When a bug appears *after* a change, ask about the diff, not the file.

Workflow:

1. `git diff HEAD~1` to isolate the recent change.
2. Paste *only* the diff into chat.

Diffs are cheap context. A 20-line diff is much more useful to the model than a 2000-line file.

Diff-driven diagnosis

When a bug appears *after* a change, ask about the diff, not the file.

Workflow:

1. `git diff HEAD~1` to isolate the recent change.
2. Paste *only* the diff into chat.
3. Ask: "*Which line in this diff most plausibly caused the following behaviour to change?*"

Diffs are cheap context. A 20-line diff is much more useful to the model than a 2000-line file.

Log-driven debugging

Ask the assistant to insert *diagnostic logging* instead of guessing at the fix.

```
Insert log statements in the method below so that we can
determine, at runtime, whether the branch on line 17 is
taken and what value ‘count‘ holds at line 22. Do not
change any logic.
```

This turns AI into an assistant that makes the bug *visible* without pretending to know where the bug is.

Rubber-ducking with a model

Classic technique: explain your bug to a rubber duck; often you find the fix while explaining.

AI works the same way, but the duck talks back.

You type out your understanding of the failure. The model asks clarifying questions. Half the time you catch the bug during your own reply.

When AI debugging fails

- Race conditions and other timing-dependent bugs – the model cannot see time.

When AI debugging fails

- Race conditions and other timing-dependent bugs – the model cannot see time.
- Bugs that require reading logs from production – the model cannot see logs it was not shown.

When AI debugging fails

- Race conditions and other timing-dependent bugs – the model cannot see time.
- Bugs that require reading logs from production – the model cannot see logs it was not shown.
- Bugs in code that depends on private / proprietary libraries the model has not seen.

When AI debugging fails

- Race conditions and other timing-dependent bugs – the model cannot see time.
- Bugs that require reading logs from production – the model cannot see logs it was not shown.
- Bugs in code that depends on private / proprietary libraries the model has not seen.
- Bugs where the "bug" is a specification disagreement – you and the reviewer disagree on what the code should do. No amount of AI can decide that for you.

AI + Debugging: summary

- Explain-this-error is the safest AI debugging use.

[1] Minimal reproducible examples

AI + Debugging: summary

- Explain-this-error is the safest AI debugging use.
- Fix-this-code needs diagnosis *before* patch.

[1] Minimal reproducible examples

AI + Debugging: summary

- Explain-this-error is the safest AI debugging use.
- Fix-this-code needs diagnosis *before* patch.
- Minimal repros[1], diffs, and log insertion are your best friends – they narrow context so the model can be precise.

[1] Minimal reproducible examples

AI + Debugging: summary

- Explain-this-error is the safest AI debugging use.
- Fix-this-code needs diagnosis *before* patch.
- Minimal repros[1], diffs, and log insertion are your best friends – they narrow context so the model can be precise.
- Distrust patches that appear without an explanation.

[1] Minimal reproducible examples

Refactoring: what and why

Refactoring: definition

Refactoring is changing the structure of code without changing its externally observable behaviour.

Two words carry the meaning:

- *Structure* – names, method boundaries, class layout, control flow shape.

If tests break, that is not refactoring – that is a bug fix or a change of contract.

Refactoring: definition

Refactoring is changing the structure of code without changing its externally observable behaviour.

Two words carry the meaning:

- *Structure* – names, method boundaries, class layout, control flow shape.
- *Observable behaviour* – the tests still pass; the callers see the same responses.

If tests break, that is not refactoring – that is a bug fix or a change of contract.

Why refactor: readability

Before – one pass? Three passes?

```
int f(int[] a) {
    int r = 0;
    for (int i = 0; i < a.length; i++)
        if (a[i] > 0) r += a[i];
    return r;
}
```

Why refactor: readability

Before – one pass? Three passes?

```
int f(int[] a) {
    int r = 0;
    for (int i = 0; i < a.length; i++)
        if (a[i] > 0) r += a[i];
    return r;
}
```

After – one pass.

```
int sumOfPositives(int[] values) {
    int sum = 0;
    for (int v : values)
        if (v > 0) sum += v;
    return sum;
}
```

Same behaviour, same complexity. A reader knows the intent from the signature alone.

Why refactor: modifiability

Before – the tax rate is copy-pasted in three places.

```
double gross(double p) { return p + p * 0.18; }
double taxOf(double p) { return p * 0.18; }
String label(double p) { return "GST 18% on " + p; }
```

Rate change to 12% $\Rightarrow$ edit three files; miss one and invoices go out wrong.

Why refactor: modifiability

Before – the tax rate is copy-pasted in three places.

```
double gross(double p) { return p + p * 0.18; }
double taxOf(double p) { return p * 0.18; }
String label(double p) { return "GST 18% on " + p; }
```

Rate change to 12% $\Rightarrow$ edit three files; miss one and invoices go out wrong. After – one place.

```
static final double GST_RATE = 0.18;
double gross(double p) { return p * (1 + GST_RATE); }
double taxOf(double p) { return p * GST_RATE; }
String label(double p) { return "GST " + (GST_RATE*100) + "% on " + p; }
```

Why refactor: testability

Before – computation and I/O tangled; you cannot unit-test the maths without a file.

```
double averageFromFile(String path) throws IOException {
    var lines = Files.readAllLines(Path.of(path));
    double sum = 0;
    for (String s : lines) sum += Double.parseDouble(s);
    return sum / lines.size();
}
```

Why refactor: testability

Before – computation and I/O tangled; you cannot unit-test the maths without a file.

```
double averageFromFile(String path) throws IOException {
    var lines = Files.readAllLines(Path.of(path));
    double sum = 0;
    for (String s : lines) sum += Double.parseDouble(s);
    return sum / lines.size();
}
```

After – pure function is unit-testable in isolation.

```
double average(List<Double> xs) {
    double sum = 0; for (double x : xs) sum += x;
    return sum / xs.size();
}
double averageFromFile(String path) throws IOException {
    return average(Files.readAllLines(Path.of(path))
        .stream().map(Double::parseDouble).toList());
}
```

Why refactor: fewer defects

Before – deep nesting hides an off-by-one.

```
String grade(int m) {
    if (m >= 0) {
        if (m < 100) { // off-by-one: excludes 100
            if (m >= 90) return "A";
            else if (m >= 75) return "B";
            else if (m >= 50) return "C";
            else return "F";
        }
    }
    return "invalid";
```

Why refactor: fewer defects

Before – deep nesting hides an off-by-one.

```
String grade(int m) {
    if (m >= 0) {
        if (m < 100) { // off-by-one: excludes 100
            if (m >= 90) return "A";
            else if (m >= 75) return "B";
            else if (m >= 50) return "C";
            else return "F";
        }
    }
    return "invalid"; // m == 100 wrongly labelled "invalid"
}
```

Why refactor: fewer defects

Before – deep nesting hides an off-by-one.

```
String grade(int m) {
    if (m >= 0) {
        if (m < 100) { // off-by-one: excludes 100
            if (m >= 90) return "A";
            else if (m >= 75) return "B";
            else if (m >= 50) return "C";
            else return "F";
        }
    }
    return "invalid"; // m == 100 wrongly labelled "invalid"
}
```

After – guard clauses make the boundary visible.

```
String grade(int m) {
    if (m < 0 || m > 100) return "invalid";
    if (m >= 90) return "A";
    if (m >= 75) return "B";
    if (m >= 50) return "C";
    return "F";
}
```

Why refactor at all?

- **Readability.** The next developer to touch this code should understand it in one pass, not three.

Why refactor at all?

- **Readability.** The next developer to touch this code should understand it in one pass, not three.
- **Modifiability.** A well-shaped method is easy to change without breaking neighbours.

Why refactor at all?

- **Readability.** The next developer to touch this code should understand it in one pass, not three.
- **Modifiability.** A well-shaped method is easy to change without breaking neighbours.
- **Testability.** Small, focused methods are easy to unit-test in isolation.

Why refactor at all?

- **Readability.** The next developer to touch this code should understand it in one pass, not three.
- **Modifiability.** A well-shaped method is easy to change without breaking neighbours.
- **Testability.** Small, focused methods are easy to unit-test in isolation.
- **Fewer defects.** Complex methods hide bugs; simple methods surface them.

Why refactor at all?

- **Readability.** The next developer to touch this code should understand it in one pass, not three.
- **Modifiability.** A well-shaped method is easy to change without breaking neighbours.
- **Testability.** Small, focused methods are easy to unit-test in isolation.
- **Fewer defects.** Complex methods hide bugs; simple methods surface them.
- ...

The two shapes of a refactor

- **Local refactor.** Rename a variable, extract a method, inline a constant. One file, minutes.

AI is superb at local refactors. It is dangerous at cross-cutting refactors unless you drive the strategy and it drives the mechanics.

The two shapes of a refactor

- **Local refactor.** Rename a variable, extract a method, inline a constant. One file, minutes.
- **Cross-cutting refactor.** Change an interface, introduce a new abstraction, remove a dependency. Many files, hours to days.

AI is superb at local refactors. It is dangerous at cross-cutting refactors unless you drive the strategy and it drives the mechanics.

Local refactor: rename for clarity

Before:

```
double c(double p, double q) {
    double x = p * q;
    return x - x * 0.10; // what does 0.10 mean?
}
```

Local refactor: rename for clarity

Before:

```
double c(double p, double q) {
    double x = p * q;
    return x - x * 0.10; // what does 0.10 mean?
}
```

After – one file, two minutes, no behaviour change:

```
double discountedPrice(double unitPrice, double quantity) {
    double lineTotal = unitPrice * quantity;
    return lineTotal - lineTotal * DISCOUNT_RATE;
}
```

Sweet-spot AI task: "rename c, p, q, x based on how they are used; do not change behaviour."

Cross-cutting refactor: add a parameter to an interface

```
interface PaymentGateway {
    Receipt charge(Customer c, Money amount); // old
    Receipt charge(Customer c, Money amount, String currency); // new
}
```

Every implementation must change:

- `StripeGateway`, `RazorpayGateway`, `FakeGateway` (tests), `MockGateway` (perf harness) – all four grow the parameter.

AI can *execute* each step, but only *you* decide the strategy (add-then-remove vs. big-bang; backfill vs. null-tolerate).

Cross-cutting refactor: add a parameter to an interface

```
interface PaymentGateway {
    Receipt charge(Customer c, Money amount); // old
    Receipt charge(Customer c, Money amount, String currency); // new
}
```

Every implementation must change:

- `StripeGateway`, `RazorpayGateway`, `FakeGateway` (tests), `MockGateway` (perf harness) – all four grow the parameter.
- Every caller of `charge(...)` must pass a currency – dozens of sites.

AI can *execute* each step, but only *you* decide the strategy (add-then-remove vs. big-bang; backfill vs. null-tolerate).

Cross-cutting refactor: add a parameter to an interface

```
interface PaymentGateway {
    Receipt charge(Customer c, Money amount); // old
    Receipt charge(Customer c, Money amount, String currency); // new
}
```

Every implementation must change:

- `StripeGateway`, `RazorpayGateway`, `FakeGateway` (tests), `MockGateway` (perf harness) – all four grow the parameter.
- Every caller of `charge(...)` must pass a currency – dozens of sites.
- Persistence: `Receipt` may need a currency column; migration script needed.

AI can *execute* each step, but only *you* decide the strategy (add-then-remove vs. big-bang; backfill vs. null-tolerate).

Cyclomatic complexity in one slide

The *cyclomatic complexity* of a method is (roughly) the number of linearly independent paths through it. In practice: count the branches (`if`, `else if`, `case`, `catch`, `&&`, `||`, `? :`) and add 1.

Rough targets:

- ≤ 5: easy to read and easy to test.

Cyclomatic complexity in one slide

The *cyclomatic complexity* of a method is (roughly) the number of linearly independent paths through it. In practice: count the branches (`if`, `else if`, `case`, `catch`, `&&`, `||`, `? :`) and add 1.

Rough targets:

- ≤ 5: easy to read and easy to test.
- 6–10: acceptable, but you should be able to name why the branches are all necessary.

Cyclomatic complexity in one slide

The *cyclomatic complexity* of a method is (roughly) the number of linearly independent paths through it. In practice: count the branches (`if`, `else if`, `case`, `catch`, `&&`, `||`, `? :`) and add 1.

Rough targets:

- ≤ 5: easy to read and easy to test.
- 6–10: acceptable, but you should be able to name why the branches are all necessary.
- > 10: refactor.

An example

```
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";
    }
}
```

Cyclomatic complexity here is 5 (four branches + one). Not terrible, but you can flatten it.

Flattened version

```
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";
}
```

Same complexity by the strict count, but easier to read: each `if` states a case and returns. No nested `else`. This is called “guard-clause style”.

Cognitive complexity

Cyclomatic complexity counts branches. *Cognitive* complexity counts *how hard the branches are to think about*: nested `ifs` cost more than sibling `ifs`.

SonarQube popularised this metric. Rule of thumb: nested control flow is your enemy.

The flattened method above has the same cyclomatic score as the nested version, but its cognitive complexity is much lower. This is the kind of gain a refactor should target.

AI helps: suggesting refactor targets

Prompt:

```
Rank the methods in Report.java by refactor priority.
For each, show cyclomatic complexity and one reason.
```

AI helps: suggesting refactor targets

Prompt:

```
Rank the methods in Report.java by refactor priority.
For each, show cyclomatic complexity and one reason.
```

Reply:

```
render(...) CC=14 deep nesting + duplicated header logic
summarise(...) CC=9 two responsibilities in one method
loadData(...) CC=3 fine, leave alone
```

A triage list you can act on – not a wall of findings.

AI helps: executing a specific refactor

Before – tax calculation inline:

```
for (LineItem li : items) {
    double tax = li.price() * 0.18;
    sum += li.price() + tax;
}
```

Prompt: “extract the tax calculation into a helper method”.

AI helps: executing a specific refactor

Before – tax calculation inline:

```
for (LineItem li : items) {
    double tax = li.price() * 0.18;
    sum += li.price() + tax;
}
```

Prompt: “extract the tax calculation into a helper method”. After:

```
for (LineItem li : items) sum += grossPrice(li);
...
private double grossPrice(LineItem li) {
    return li.price() * (1 + GST_RATE);
}
```

AI helps: renaming for clarity

Before – fields named after their type, not their meaning:

```
class Order {
    String s; // status
    List<Item> l; // line items
    long t; // placedAt (epoch ms)
}
```

Prompt: "rename fields of `Order` based on their usage across the project".

AI helps: renaming for clarity

Before – fields named after their type, not their meaning:

```
class Order {
    String s; // status
    List<Item> l; // line items
    long t; // placedAt (epoch ms)
}
```

Prompt: "rename fields of `Order` based on their usage across the project". After:

```
class Order {
    String status;
    List<Item> lineItems;
    long placedAt; // epoch ms
}
```

Every read/write site updated in the same diff.

AI helps: explaining why a refactor is safe

Before:

```
while (true) {
    if (queue.isEmpty()) break;
    process(queue.poll());
}
```

AI helps: explaining why a refactor is safe

Before:

```
while (true) {
    if (queue.isEmpty()) break;
    process(queue.poll());
}
```

After – and, crucially, *why*:

```
while (!queue.isEmpty()) process(queue.poll());
```

AI: "Both loops exit on the same condition; nothing mutates `queue` between the check and the `break`, so iteration count is identical. `process` is unchanged. Safe." The explanation is what makes it worth accepting.

AI helps: producing a diff, not a rewrite

Prompt: “Refactor as a unified diff. Extract the price computation into a helper. Do not rewrite unrelated lines.”

AI helps: producing a diff, not a rewrite

Prompt: "Refactor as a unified diff. Extract the price computation into a helper. Do not rewrite unrelated lines." Reply:

```
@@ -14,4 +14,3 @@ public double invoiceTotal(...)
     for (LineItem li : items) {
- double tax = li.price() * 0.18;
- sum += li.price() + tax;
+ sum += grossPrice(li);
     }
@@ +25,4 @@
+private double grossPrice(LineItem li) {
+ return li.price() * (1 + GST_RATE);
+}
```

Diff review $\ll$ before/after diff-in-your-head.

Where AI helps in refactoring

- **Suggesting** refactor targets: “rank the methods in this file by refactor priority”.

Where AI helps in refactoring

- **Suggesting** refactor targets: “rank the methods in this file by refactor priority”.
- **Executing** a specific refactor: “extract these three lines into a helper”.

Where AI helps in refactoring

- **Suggesting** refactor targets: “rank the methods in this file by refactor priority”.
- **Executing** a specific refactor: “extract these three lines into a helper”.
- **Renaming** for clarity: “suggest better names for these variables based on their usage”.

Where AI helps in refactoring

- **Suggesting** refactor targets: “rank the methods in this file by refactor priority”.
- **Executing** a specific refactor: “extract these three lines into a helper”.
- **Renaming** for clarity: “suggest better names for these variables based on their usage”.
- **Explaining** why a refactor is safe (or not).

Where AI helps in refactoring

- **Suggesting** refactor targets: “rank the methods in this file by refactor priority”.
- **Executing** a specific refactor: “extract these three lines into a helper”.
- **Renaming** for clarity: “suggest better names for these variables based on their usage”.
- **Explaining** why a refactor is safe (or not).
- **Producing** a diff you can review, rather than editing in place.

AI hurts: silent behaviour change

Before – classic short-circuit null guard:

```
if (a != null && a.length > 0) process(a);
```

Prompt: “clean up the guard”.

AI hurts: silent behaviour change

Before – classic short-circuit null guard:

```
if (a != null && a.length > 0) process(a);
```

Prompt: “clean up the guard”. After – “cleaner”, now crashes:

```
if (a.length > 0 && a != null) process(a);
```

AI hurts: silent behaviour change

Before – classic short-circuit null guard:

```
if (a != null && a.length > 0) process(a);
```

Prompt: “clean up the guard”. After – “cleaner”, now crashes:

```
if (a.length > 0 && a != null) process(a);
// NullPointerException when a == null
```

Same shape, reversed order, different semantics. Tests catch it – if you had one for the null case.

AI hurts: import churn

Before – three explicit imports:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
```

Prompt: “tidy the imports”.

AI hurts: import churn

Before – three explicit imports:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
```

Prompt: “tidy the imports”. After – wildcard plus reordering:

```
import java.util.*; // and every other util type comes along
```

Diff is huge, checkstyle rule “no wildcard imports” now fails, and your PR is 80% import noise.

AI hurts: unwanted modernisation

Before – an ordinary loop the caller *mutates* later:

```
List<String> names = new ArrayList<>();
for (User u : users)
    if (u.active()) names.add(u.name());
names.add("SENTINEL"); // caller appends
```

Prompt: “simplify”.

AI hurts: unwanted modernisation

Before – an ordinary loop the caller *mutates* later:

```
List<String> names = new ArrayList<>();
for (User u : users)
    if (u.active()) names.add(u.name());
names.add("SENTINEL"); // caller appends
```

Prompt: “simplify”. After – “modernised” to streams:

```
var names = users.stream()
                 .filter(User::active)
                 .map(User::name)
                 .toList();
names.add("SENTINEL");
```

AI hurts: unwanted modernisation

Before – an ordinary loop the caller *mutates* later:

```
List<String> names = new ArrayList<>();
for (User u : users)
    if (u.active()) names.add(u.name());
names.add("SENTINEL"); // caller appends
```

Prompt: “simplify”. After – “modernised” to streams:

```
var names = users.stream()
                 .filter(User::active)
                 .map(User::name)
                 .toList();
names.add("SENTINEL");  // UnsupportedOperationException
```

`Stream.toList()` returns an *unmodifiable* list. Runtime crash where compilation was fine.

AI hurts: loss of comments

Before – the comment is the whole point of the check:

```
// MSA-2019: reactor temperature capped per compliance memo #42.
// Do NOT remove without approval from safety board.
if (temperature > 800) return CAP;
```

Prompt: “simplify this file”.

AI hurts: loss of comments

Before – the comment is the whole point of the check:

```
// MSA-2019: reactor temperature capped per compliance memo #42.
// Do NOT remove without approval from safety board.
if (temperature > 800) return CAP;
```

Prompt: “simplify this file”. After – code identical, comment vanished:

```
if (temperature > 800) return CAP;
```

Next reader has no idea why 800. Next refactor bumps it to 850 “because the tests still pass”.

Where AI hurts in refactoring

- Silent behaviour changes – a refactor prompt returns code that is “simpler” but subtly different.

Every refactor needs the tests to run afterwards. Every one.

Where AI hurts in refactoring

- Silent behaviour changes – a refactor prompt returns code that is “simpler” but subtly different.
- Import churn – unnecessary reordering, removal, or addition of imports.

Every refactor needs the tests to run afterwards. Every one.

Where AI hurts in refactoring

- Silent behaviour changes – a refactor prompt returns code that is "simpler" but subtly different.
- Import churn – unnecessary reordering, removal, or addition of imports.
- "Modernisation" you did not ask for – `for` loops rewritten as streams even when the loop was fine.

Every refactor needs the tests to run afterwards. Every one.

Where AI hurts in refactoring

- Silent behaviour changes – a refactor prompt returns code that is "simpler" but subtly different.
- Import churn – unnecessary reordering, removal, or addition of imports.
- "Modernisation" you did not ask for – `for` loops rewritten as streams even when the loop was fine.
- Loss of comments – helpful comments deleted because the refactored code "doesn't need them".

Every refactor needs the tests to run afterwards. Every one.

Refactoring loops and duplication

The four common refactor moves

- **Extract method.** A block of code becomes a named helper.

AI is good at all four. The rest of this hour we focus on the last two, plus *duplication removal*.

The four common refactor moves

- **Extract method.** A block of code becomes a named helper.
- **Inline method.** A tiny helper is pulled back into its caller.

AI is good at all four. The rest of this hour we focus on the last two, plus *duplication removal*.

The four common refactor moves

- **Extract method.** A block of code becomes a named helper.
- **Inline method.** A tiny helper is pulled back into its caller.
- **Rename.** A variable, method, or class gets a better name.

AI is good at all four. The rest of this hour we focus on the last two, plus *duplication removal*.

The four common refactor moves

- **Extract method.** A block of code becomes a named helper.
- **Inline method.** A tiny helper is pulled back into its caller.
- **Rename.** A variable, method, or class gets a better name.
- **Replace loop with pipeline.** A `for` that collects results becomes a stream.

AI is good at all four. The rest of this hour we focus on the last two, plus *duplication removal*.

Extract-method refactor

Before:

```
public double invoiceTotal(List<LineItem> items) {
    double sum = 0;
    for (LineItem li : items) {
        double tax = li.price() * 0.18;
        double net = li.price() + tax;
        sum += net;
    }
    return sum;
}
```

Extract-method refactor: after

```
public double invoiceTotal(List<LineItem> items) {
    double sum = 0;
    for (LineItem li : items) {
        sum += grossPrice(li);
    }
    return sum;
}
private double grossPrice(LineItem li) {
    double tax = li.price() * 0.18;
    return li.price() + tax;
}
```

A one-line prompt (“extract the tax calculation into a helper”) gets this cleanly.

Loop-to-stream refactor

Before:

```
List<String> names = new ArrayList<>();
for (User u : users) {
    if (u.active()) {
        names.add(u.name().toUpperCase());
    }
}
```

After (streams):

```
List<String> names = users.stream()
    .filter(User::active)
    .map(u -> u.name().toUpperCase())
    .toList();
```

Loop-to-stream: when *not* to do it

- When the loop mutates two things at once (streams punish multi-purpose bodies).

AI will happily convert either direction on demand. *You* decide which direction is right.

Loop-to-stream: when *not* to do it

- When the loop mutates two things at once (streams punish multi-purpose bodies).
- When exception handling inside the body must propagate as-is.

AI will happily convert either direction on demand. *You* decide which direction is right.

Loop-to-stream: when *not* to do it

- When the loop mutates two things at once (streams punish multi-purpose bodies).
- When exception handling inside the body must propagate as-is.
- When the reader of your code is not fluent in streams – the team should agree on a house style.

AI will happily convert either direction on demand. *You* decide which direction is right.

The “rewrite this loop” prompt

Good prompt shape:

- Show the loop.

Ask for a diff because it is much easier to review a diff than to diff two versions in your head.

The “rewrite this loop” prompt

Good prompt shape:

- Show the loop.
- Say what you want: “rewrite as a stream / a while loop / a recursive method”.

Ask for a diff because it is much easier to review a diff than to diff two versions in your head.

The “rewrite this loop” prompt

Good prompt shape:

- Show the loop.
- Say what you want: “rewrite as a stream / a while loop / a recursive method”.
- Say what you do not want: “do not change behaviour on empty input”.

Ask for a diff because it is much easier to review a diff than to diff two versions in your head.

The “rewrite this loop” prompt

Good prompt shape:

- Show the loop.
- Say what you want: “rewrite as a stream / a while loop / a recursive method”.
- Say what you do not want: “do not change behaviour on empty input”.
- Ask for a diff, not a full rewrite.

Ask for a diff because it is much easier to review a diff than to diff two versions in your head.

Duplication removal: the DRY principle

DRY – "don't repeat yourself" – has a specific meaning: *every piece of knowledge should have a single, unambiguous representation in a system*.

It is not "never write the same three lines twice". Sometimes duplication is cheaper than the abstraction that removes it.

Ask: *is the same knowledge encoded in both places, or does the same shape happen to appear in two unrelated places?* If the former, remove the duplication. If the latter, leave it alone.

Duplication example

```
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; }
```

Same *shape*, different *knowledge*.

Refactoring into a table of `(country, rate)` pairs is only worth it if a policy across countries is coming – e.g., "compute tax with country-specific rounding rules".

The AI-driven duplication-removal loop

1. Ask: "*Show me the two most similar blocks in this file.*"

Step 2 is the one people skip. Skipping it produces the wrong abstraction.

The AI-driven duplication-removal loop

1. Ask: "*Show me the two most similar blocks in this file.*"
2. Ask: "*Do these blocks encode the same knowledge?*"

Step 2 is the one people skip. Skipping it produces the wrong abstraction.

The AI-driven duplication-removal loop

1. Ask: "*Show me the two most similar blocks in this file.*"
2. Ask: "*Do these blocks encode the same knowledge?*"
3. If yes: "*Extract the common code into a helper; preserve behaviour.*"

Step 2 is the one people skip. Skipping it produces the wrong abstraction.

The AI-driven duplication-removal loop

1. Ask: "*Show me the two most similar blocks in this file.*"
2. Ask: "*Do these blocks encode the same knowledge?*"
3. If yes: "*Extract the common code into a helper; preserve behaviour.*"
4. Run tests. Every time.

Step 2 is the one people skip. Skipping it produces the wrong abstraction.

The AI-driven duplication-removal loop

1. Ask: "*Show me the two most similar blocks in this file.*"
2. Ask: "*Do these blocks encode the same knowledge?*"
3. If yes: "*Extract the common code into a helper; preserve behaviour.*"
4. Run tests. Every time.
5. Read the diff before committing.

Step 2 is the one people skip. Skipping it produces the wrong abstraction.

AI + Refactoring: summary

- Refactoring is behaviour-preserving change. Tests decide.

AI + Refactoring: summary

- Refactoring is behaviour-preserving change. Tests decide.
- Cyclomatic and cognitive complexity give you a language for “too much”.

AI + Refactoring: summary

- Refactoring is behaviour-preserving change. Tests decide.
- Cyclomatic and cognitive complexity give you a language for “too much”.
- Extract-method, rename, and loop rewrites are AI’s sweet spot.

AI + Refactoring: summary

- Refactoring is behaviour-preserving change. Tests decide.
- Cyclomatic and cognitive complexity give you a language for “too much”.
- Extract-method, rename, and loop rewrites are AI’s sweet spot.
- Duplication removal needs judgment: shape versus knowledge.

AI + Refactoring: summary

- Refactoring is behaviour-preserving change. Tests decide.
- Cyclomatic and cognitive complexity give you a language for “too much”.
- Extract-method, rename, and loop rewrites are AI’s sweet spot.
- Duplication removal needs judgment: shape versus knowledge.
- Every refactor ends with a test run and a diff review.

Static analysis + LLM

Static analysis: the classical layer

Static analysis tools read your code without running it and report likely defects. In Java the big three are:

- **SpotBugs** – bytecode-level bug detection, descended from FindBugs.

In Python the analogous tool is Pylint; in JavaScript, ESLint; in Go, `go vet`. The idea is the same: a rule engine that ships known-bug patterns.

Static analysis: the classical layer

Static analysis tools read your code without running it and report likely defects. In Java the big three are:

- **SpotBugs** – bytecode-level bug detection, descended from FindBugs.
- **PMD** – source-level pattern detection.

In Python the analogous tool is Pylint; in JavaScript, ESLint; in Go, `go vet`. The idea is the same: a rule engine that ships known-bug patterns.

Static analysis: the classical layer

Static analysis tools read your code without running it and report likely defects. In Java the big three are:

- **SpotBugs** – bytecode-level bug detection, descended from FindBugs.
- **PMD** – source-level pattern detection.
- **Error Prone** – Google's javac plugin; flags patterns that compile but are almost certainly wrong.

In Python the analogous tool is Pylint; in JavaScript, ESLint; in Go, `go vet`. The idea is the same: a rule engine that ships known-bug patterns.

Example: a SpotBugs finding

```
public class Config {
    private String path;
    public String getPath() { return path; }
    public void setPath(String p) { path = p; }
    public boolean equals(Object o) {
        return path.equals(((Config)o).path);
    }
}
```

SpotBugs flags: `EQ_UNUSUAL` – overriding `equals` without `hashCode`. It gives you the rule and the line; it does not tell you what to write instead.

Where LLMs complement static analysis

Static analysis is high precision, low recall for *semantic* bugs. It finds rule violations, not intent violations.

LLMs are the reverse: they read intent, but they cannot reliably run a fixed rule set at scale.

The productive workflow: **run the static analyser first, then feed each finding to an LLM for context and fix suggestions**. This is the topic of the rest of this section.

The static-analysis-plus-LLM loop

1. Run SpotBugs (or PMD). Save the report.

The LLM turns a rule name (`EQ_UNUSUAL`) into an actionable patch. The static analyser guarantees the finding is real.

The static-analysis-plus-LLM loop

1. Run SpotBugs (or PMD). Save the report.
2. For each finding, feed the LLM: (a) the finding, (b) the surrounding code, (c) the *rule* the tool cites.

The LLM turns a rule name (`EQ_UNUSUAL`) into an actionable patch. The static analyser guarantees the finding is real.

The static-analysis-plus-LLM loop

1. Run SpotBugs (or PMD). Save the report.
2. For each finding, feed the LLM: (a) the finding, (b) the surrounding code, (c) the *rule* the tool cites.
3. Ask: "*Given the rule and the code, what is the intended fix? Show the smallest patch that satisfies the rule without changing behaviour.*"

The LLM turns a rule name (`EQ_UNUSUAL`) into an actionable patch. The static analyser guarantees the finding is real.

The static-analysis-plus-LLM loop

1. Run SpotBugs (or PMD). Save the report.
2. For each finding, feed the LLM: (a) the finding, (b) the surrounding code, (c) the *rule* the tool cites.
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. Read. Apply. Test.

The LLM turns a rule name (`EQ_UNUSUAL`) into an actionable patch. The static analyser guarantees the finding is real.

False positives and how to handle them

Static analysers produce false positives. Every project has them.

Old workflow: add a `@SuppressWarnings` annotation and forget.

Better workflow with AI:

- Ask: "*Is this finding a true positive? Explain why.*"

The comment is the audit trail.

False positives and how to handle them

Static analysers produce false positives. Every project has them.

Old workflow: add a `@SuppressWarnings` annotation and forget.

Better workflow with AI:

- Ask: "*Is this finding a true positive? Explain why.*"
- If yes, fix.

The comment is the audit trail.

False positives and how to handle them

Static analysers produce false positives. Every project has them.

Old workflow: add a `@SuppressWarnings` annotation and forget.

Better workflow with AI:

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

The comment is the audit trail.

Triaging a report of 500 findings

Real projects generate walls of findings on their first run. Do not fix them all. Triage first.

An LLM can help with triage:

- "Cluster these findings by rule and give the top-5 rules to address first."

A good report becomes a project plan, not a homework assignment.

Triaging a report of 500 findings

Real projects generate walls of findings on their first run. Do not fix them all. Triage first.

An LLM can help with triage:

- "Cluster these findings by rule and give the top-5 rules to address first."
- "For each cluster, is this a hotspot in the codebase or a cold path?"

A good report becomes a project plan, not a homework assignment.

Triaging a report of 500 findings

Real projects generate walls of findings on their first run. Do not fix them all. Triage first.

An LLM can help with triage:

- "Cluster these findings by rule and give the top-5 rules to address first."
- "For each cluster, is this a hotspot in the codebase or a cold path?"
- "Suggest a per-cluster fix strategy that we can apply mechanically."

A good report becomes a project plan, not a homework assignment.

Automated fixers: SpotBugs' plugin ecosystem

Some tools have automated fixers. SpotBugs has *Contrib* rules. PMD has *Auto-fix* for a subset. Error Prone has *refaster templates*.

These are rule-based. When a fix is safe and mechanical, use the automated fixer. When the fix requires understanding intent ("did the author mean assignment or comparison here?"), route to the LLM.

The two are complementary, not competitive.

IDE integration

IntelliJ IDEA and VS Code (with the Java pack) both surface SpotBugs and PMD findings inline.

Copilot Chat can be invoked on a finding with a "fix this warning" action. The chat receives:

- The tool name (SpotBugs).

Copilot then produces a diff. *You* decide whether to apply. This is the workflow you will use in the second hands-on session.

IDE integration

IntelliJ IDEA and VS Code (with the Java pack) both surface SpotBugs and PMD findings inline.

Copilot Chat can be invoked on a finding with a "fix this warning" action. The chat receives:

- The tool name (SpotBugs).
- The rule ID and message.

Copilot then produces a diff. *You* decide whether to apply. This is the workflow you will use in the second hands-on session.

IDE integration

IntelliJ IDEA and VS Code (with the Java pack) both surface SpotBugs and PMD findings inline.

Copilot Chat can be invoked on a finding with a "fix this warning" action. The chat receives:

- The tool name (SpotBugs).
- The rule ID and message.
- The current file.

Copilot then produces a diff. *You* decide whether to apply. This is the workflow you will use in the second hands-on session.

AI + Static Analysers: summary

- Static analysis gives you a rule-checked baseline of bug patterns.

AI + Static Analysers: summary

- Static analysis gives you a rule-checked baseline of bug patterns.
- LLMs cannot replace static analysis, and static analysis cannot replace LLMs.

AI + Static Analysers: summary

- Static analysis gives you a rule-checked baseline of bug patterns.
- LLMs cannot replace static analysis, and static analysis cannot replace LLMs.
- Combine them: analyser finds the finding, LLM explains and proposes the fix, you apply and test.

AI + Static Analysers: summary

- Static analysis gives you a rule-checked baseline of bug patterns.
- LLMs cannot replace static analysis, and static analysis cannot replace LLMs.
- Combine them: analyser finds the finding, LLM explains and proposes the fix, you apply and test.
- Triage before fixing – do not chase every finding in a legacy report.

Edge cases and robustness

What is an “edge case”?

An input that is technically legal but sits at the boundary of the type or specification.

Examples for a method `int average(int[] a)`:

- Empty array.

Edge cases are where most bugs live. Discovering them is a searchable problem, and LLMs are good searchers.

What is an “edge case”?

An input that is technically legal but sits at the boundary of the type or specification.

Examples for a method `int average(int[] a)`:

- Empty array.
- Length 1.

Edge cases are where most bugs live. Discovering them is a searchable problem, and LLMs are good searchers.

What is an “edge case”?

An input that is technically legal but sits at the boundary of the type or specification.

Examples for a method `int average(int[] a)`:

- Empty array.
- Length 1.
- Array containing `Integer.MAX_VALUE`.

Edge cases are where most bugs live. Discovering them is a searchable problem, and LLMs are good searchers.

What is an “edge case”?

An input that is technically legal but sits at the boundary of the type or specification.

Examples for a method `int average(int[] a)`:

- Empty array.
- Length 1.
- Array containing `Integer.MAX_VALUE`.
- Array containing negatives.

Edge cases are where most bugs live. Discovering them is a searchable problem, and LLMs are good searchers.

What is an “edge case”?

An input that is technically legal but sits at the boundary of the type or specification.

Examples for a method `int average(int[] a)`:

- Empty array.
- Length 1.
- Array containing `Integer.MAX_VALUE`.
- Array containing negatives.
- Very long array (memory).

Edge cases are where most bugs live. Discovering them is a searchable problem, and LLMs are good searchers.

What is an “edge case”?

An input that is technically legal but sits at the boundary of the type or specification.

Examples for a method `int average(int[] a)`:

- Empty array.
- Length 1.
- Array containing `Integer.MAX_VALUE`.
- Array containing negatives.
- Very long array (memory).
- Null array reference.

Edge cases are where most bugs live. Discovering them is a searchable problem, and LLMs are good searchers.

Your turn: name every edge case

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

Three minutes. Call out every edge-case input you would add to a test suite for this method.

Your turn: name every edge case

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

Three minutes. Call out every edge-case input you would add to a test suite for this method.

A reasonably complete list:

- *Month boundaries:* 1, 12; 0, 13; negative; `Integer.MIN_VALUE`, `MAX_VALUE`.
- *Leap-year rules:* 2020 (leap: div 4), 1900 (*not* leap: div 100, not 400), 2000 (leap: div 400), 2100 (not leap).
- *Year boundaries:* year 1, year 0 (does not exist in the proleptic Gregorian), negative year (BCE), `MAX_VALUE`.
- *Calendar reform:* 1582 – 4 Oct was followed by 15 Oct; pre-1582 is Julian. Which does the spec mean?
- *Return-type sanity:* every valid month must return 28–31. Is there any path that could return 0 or 32?
- *Locale / calendar choice:* Orthodox churches still use Julian. "Gregorian" in the doc or just "calendar"?

The “list edge cases” prompt

```
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.

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

This is a top-shelf use of LLMs. A junior developer will forget half the categories; the model won’t. Whether the resulting tests matter depends on your domain – but the *list* is free.

From edge cases to tests

Two follow-up prompts:

1. "*Write JUnit 5 tests for each edge case above.*"

Property-based tests are the natural companion to LLM-generated edge-case lists. We will do more of this in Module 5.

From edge cases to tests

Two follow-up prompts:

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

Property-based tests are the natural companion to LLM-generated edge-case lists. We will do more of this in Module 5.

Property-based testing with jqwik

Instead of asserting on *specific* inputs, assert an *invariant* that must hold for every valid input. The framework generates hundreds; if any fails, it *shrinks* the failure to the smallest 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()); // AssertJ
}
```

Property-based testing with jqwik

Instead of asserting on *specific* inputs, assert an *invariant* that must hold for every valid input. The framework generates hundreds; if any fails, it *shrinks* the failure to the smallest 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()); // AssertJ
}
```

- Roughly 1000 random arrays per run (configurable).

Property-based testing with jqwik

Instead of asserting on *specific* inputs, assert an *invariant* that must hold for every valid input. The framework generates hundreds; if any fails, it *shrinks* the failure to the smallest 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()); // AssertJ
}
```

- Roughly 1000 random arrays per run (configurable).
- If any fails, jqwik shrinks to a minimal case – typically something like `{Integer.MAX_VALUE, 1}` exposing an overflow in `average`.

Property-based testing with jqwik

Instead of asserting on *specific* inputs, assert an *invariant* that must hold for every valid input. The framework generates hundreds; if any fails, it *shrinks* the failure to the smallest 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()); // AssertJ
}
```

- Roughly 1000 random arrays per run (configurable).
- If any fails, jqwik shrinks to a minimal case – typically something like `{Integer.MAX_VALUE, 1}` exposing an overflow in `average`.
- You write the *property*; jqwik supplies the inputs.

Robustness testing beyond edge cases

- **Fuzzing** – generate random inputs and look for crashes. `jazzer` is the modern Java fuzzer.

LLMs can generate mutation targets and contract stubs; they cannot run the tools. Use them as an accelerator, not a replacement.

Robustness testing beyond edge cases

- **Fuzzing** – generate random inputs and look for crashes. `jazzer` is the modern Java fuzzer.
- **Mutation testing** – change your code slightly and see if any test fails. If not, your tests are weak. `PIT` is the Java tool.

LLMs can generate mutation targets and contract stubs; they cannot run the tools. Use them as an accelerator, not a replacement.

Robustness testing beyond edge cases

- **Fuzzing** – generate random inputs and look for crashes. `jazzer` is the modern Java fuzzer.
- **Mutation testing** – change your code slightly and see if any test fails. If not, your tests are weak. `PIT` is the Java tool.
- **Contract tests** – verify that a component still conforms to its published interface.

LLMs can generate mutation targets and contract stubs; they cannot run the tools. Use them as an accelerator, not a replacement.

Coverage-guided fuzzing with Jazzer

Fuzzing throws random bytes at code and reports every crash. The core idea, 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 = bug worth reporting
}
```

Coverage-guided fuzzing with Jazzer

Fuzzing throws random bytes at code and reports every crash. The core idea, 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 = bug worth reporting
}
```

- Jazzer does the same thing but *coverage-guided*: it watches which branches each input reaches and mutates toward unreached ones.

Coverage-guided fuzzing with Jazzer

Fuzzing throws random bytes at code and reports every crash. The core idea, 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 = bug worth reporting
}
```

- Jazzer does the same thing but *coverage-guided*: it watches which branches each input reaches and mutates toward unreached ones.
- It also *shrinks* a failure to a minimal crashing input, and saves a *corpus* of interesting inputs across runs.

Coverage-guided fuzzing with Jazzer

Fuzzing throws random bytes at code and reports every crash. The core idea, 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 = bug worth reporting
}
```

- Jazzer does the same thing but *coverage-guided*: it watches which branches each input reaches and mutates toward unreached ones.
- It also *shrinks* a failure to a minimal crashing input, and saves a *corpus* of interesting inputs across runs.
- Good for parsers, deserialisers, protocol handlers – code whose input space is too large to

Mutation testing with PIT

PIT *mutates* your code slightly (a *mutant*) and re-runs your tests. A strong test suite *kills* every mutant – some test fails. A surviving mutant means your tests do not actually check what you think they check.

Original:

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

Mutation testing with PIT

PIT *mutates* your code slightly (a *mutant*) and re-runs your tests. A strong test suite *kills* every mutant – some test fails. A surviving mutant means your tests do not actually check what you think they check.

Original:

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

PIT injects mutants:

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

Report: "Mutation score 72%" – 28% of mutants survived; write tests that would kill them.

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.

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

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

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.

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

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

Provider test – `OrderService` must produce it:

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

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.

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

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

Provider test – `OrderService` must produce it:

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

Consumer test – `BillingClient` must accept it:

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

Rename `"total"` → `"amount"` on the provider and its own contract test fails – *before* billing

Prompt: adversarial-input generation

```
The method below is deployed as 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) { ... }
```

This turns the LLM into a poor-person's red team. Real security review is more than this. But five plausible attacks in 30 seconds is a great starting point.

Where robustness prompts go wrong

- The model invents attacks that do not apply to your stack.

Filter the list. Use domain knowledge you already have. Do not paste all fifteen suggestions into your codebase as guards.

Where robustness prompts go wrong

- The model invents attacks that do not apply to your stack.
- The model over-reports: everything is a “potential vulnerability”.

Filter the list. Use domain knowledge you already have. Do not paste all fifteen suggestions into your codebase as guards.

Where robustness prompts go wrong

- The model invents attacks that do not apply to your stack.
- The model over-reports: everything is a “potential vulnerability”.
- The model repeats generic advice (“sanitize your inputs”) instead of concrete cases.

Filter the list. Use domain knowledge you already have. Do not paste all fifteen suggestions into your codebase as guards.

Where robustness prompts go wrong

- The model invents attacks that do not apply to your stack.
- The model over-reports: everything is a "potential vulnerability".
- The model repeats generic advice ("sanitize your inputs") instead of concrete cases.
- The model misses domain-specific attacks it has not seen.

Filter the list. Use domain knowledge you already have. Do not paste all fifteen suggestions into your codebase as guards.

AI + Robustness: summary

- Edge-case enumeration is one of the LLM's strongest contributions to code quality.

AI + Robustness: summary

- Edge-case enumeration is one of the LLM's strongest contributions to code quality.
- Robustness testing goes beyond enumeration – fuzzing, mutation testing, property-based testing.

AI + Robustness: summary

- Edge-case enumeration is one of the LLM's strongest contributions to code quality.
- Robustness testing goes beyond enumeration – fuzzing, mutation testing, property-based testing.
- Adversarial prompts turn the model into a first-pass red team.

AI + Robustness: summary

- Edge-case enumeration is one of the LLM's strongest contributions to code quality.
- Robustness testing goes beyond enumeration – fuzzing, mutation testing, property-based testing.
- Adversarial prompts turn the model into a first-pass red team.
- You still decide which findings matter.

Legacy code: reading before writing

You inherit code more often than you write it

- The typical software job: 80% reading, 20% writing.

You inherit code more often than you write it

- The typical software job: 80% reading, 20% writing.
- Most of the code you touch was written by someone else – often, someone who has since left.

You inherit code more often than you write it

- The typical software job: 80% reading, 20% writing.
- Most of the code you touch was written by someone else – often, someone who has since left.
- Two beginner reactions when faced with unfamiliar code:

You inherit code more often than you write it

- The typical software job: 80% reading, 20% writing.
- Most of the code you touch was written by someone else – often, someone who has since left.
- Two beginner reactions when faced with unfamiliar code:
  - *Rewrite it*: fastest way to introduce new bugs.

You inherit code more often than you write it

- The typical software job: 80% reading, 20% writing.
- Most of the code you touch was written by someone else – often, someone who has since left.
- Two beginner reactions when faced with unfamiliar code:
  - *Rewrite it*: fastest way to introduce new bugs.
  - *Patch it without reading*: fastest way to keep the old bugs.

You inherit code more often than you write it

- The typical software job: 80% reading, 20% writing.
- Most of the code you touch was written by someone else – often, someone who has since left.
- Two beginner reactions when faced with unfamiliar code:
  - *Rewrite it*: fastest way to introduce new bugs.
  - *Patch it without reading*: fastest way to keep the old bugs.
- There is a third option: *orient yourself first*. AI makes this dramatically faster.

The five-step orientation workflow

Before writing a single line of code in an unfamiliar project:

1. **Zoom out.** What is the repository shape?

Steps 1–4 are read-only. Step 5 is where you finally open the editor.

The five-step orientation workflow

Before writing a single line of code in an unfamiliar project:

1. **Zoom out.** What is the repository shape?
2. **Follow the data.** Where does input enter, and where does state live?

Steps 1–4 are read-only. Step 5 is where you finally open the editor.

The five-step orientation workflow

Before writing a single line of code in an unfamiliar project:

1. **Zoom out.** What is the repository shape?
2. **Follow the data.** Where does input enter, and where does state live?
3. **Explain-this-class.** Read the classes you will touch.

Steps 1–4 are read-only. Step 5 is where you finally open the editor.

The five-step orientation workflow

Before writing a single line of code in an unfamiliar project:

1. **Zoom out.** What is the repository shape?
2. **Follow the data.** Where does input enter, and where does state live?
3. **Explain-this-class.** Read the classes you will touch.
4. **Reverse the call graph.** Who calls the code you will change?

Steps 1–4 are read-only. Step 5 is where you finally open the editor.

The five-step orientation workflow

Before writing a single line of code in an unfamiliar project:

1. **Zoom out.** What is the repository shape?
2. **Follow the data.** Where does input enter, and where does state live?
3. **Explain-this-class.** Read the classes you will touch.
4. **Reverse the call graph.** Who calls the code you will change?
5. **Locate the ZIP code.** Which handful of files owns your task?

Steps 1–4 are read-only. Step 5 is where you finally open the editor.

Step 1: zoom out

Get the repository's shape in one glance:

```
tree -L 2 -I 'target|.git|node_modules'
cat pom.xml | head -40 # for maven or build.gradle, package.json
```

Then ask:

```
Given this directory layout and pom.xml, describe the
project's architecture in five bullet points. What are the
entry points? What are the module boundaries?
```

The reply gives you a map. It will be partly wrong – but even a partly-wrong map beats no map.