

---

# Clean Code

*"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." — Martin Fowler*

# What we'll cover today

- 01 Why clean code: quality & technical debt
- 02 Readable code: names, comments, layout
- 03 Design principles
- 04 Code smells
- 05 A refactoring catalogue
- 06 Java tooling
- 07 A CI quality pipeline
- Practical: pipelines in GitLab

# 01

## Why clean code

Readability isn't an aesthetic preference — it's the maintenance bill, paid over and over.

WHAT IS QUALITY CODE?

# Four properties, one of them for humans



## Correct

Does what its specification claims.



## Robust

Handles edge cases and failures gracefully.



## Readable

Understood quickly, without prior context.



## Efficient

Uses resources appropriately — not wasteful, not over-built.

The first two are about the machine. **Readable** is about the next developer — and in maintenance, that is where the money goes.

## SOFTWARE ENTROPY & TECHNICAL DEBT

*"An evolving system increases its complexity unless work is done to reduce it."*

— Meir Lehman

Ward Cunningham's metaphor: shipping quick-and-dirty code is like taking on **debt**. A little speeds you up — *if* you pay it back with refactoring. Every minute spent on not-quite-right code is **interest** on that debt.

# Deliberate vs. accidental

## Deliberate

A conscious tradeoff, eyes open: "hardcode this for the demo; ticket filed to fix it after launch." A reasonable business decision — **as long as someone intends to repay it** and everyone knows it exists.

## Accidental ("reckless")

Shortcuts nobody chose: code that grew messy through neglect, a design never revisited, corners cut by a team that didn't know better. This is what the smells section helps you spot.

The failure mode isn't taking on debt — it's taking it on **without anyone noticing**, so it's never tracked, never prioritised, and quietly compounds.

# 02

## Readable code

The everyday craft: good names, few comments, and a layout that reads like a newspaper.

# A good name replaces a comment

**Auto-descriptive** & pronounceable — avoid abbreviations.

**Variables**: nouns — what they hold.

**Methods**: verbs — getUser(), computeTotal().

**Booleans**: questions — isValid(), hasPermission().

**Constants**: SCREAMING\_SNAKE\_CASE — never a magic number.

```
// Bad — what is 'd'? what is '4'?  
int d = 86400;  
if (u.s == 4) { ... }  
  
// Good — no comment needed  
final int SECONDS_PER_DAY = 86400;  
if (user.status == UserStatus.SUSPENDED)
```

# A comment often hides a naming failure

*"Comments are always a failure." — Robert C. Martin. They lie: code is refactored, comments are forgotten, and a stale comment misleads because readers trust it.*

**STALE** – comment says 3, loop says 5

```
// Retry up to 3 times
for (int i = 0; i < 5; i++) {
    if (tryConnect()) break;
}
```

**FIXED** – a name, no comment

```
final int MAX_ATTEMPTS = 5;
for (int i = 0; i < MAX_ATTEMPTS; i++) {
    if (tryConnect()) break;
}
```

# Comment the **why**, never the **how**



## Javadoc for public APIs

Document intent and contracts — what a caller can rely on, not the implementation.



## Algorithm citations

A non-obvious algorithm? Cite the paper or source, so a maintainer can find the *why*.



## Legal & license headers

Required by many organisations — a legitimate, stable use of comments.

The common thread: these explain **intent** a reader can't recover from the code itself. That is the one thing a comment does that a name cannot.

# A file should read like a newspaper

- ✓ **File size** ~200 lines; 500 is a warning sign.
- ✓ **Line length** 80–120 chars — beyond that, logic is too dense.
- ✓ **Indentation** consistent — enforced by tooling, not by convention.
- ✓ **Reading order** top-to-bottom — high-level first, details lower.

A reader should be able to stop at any point and still have grasped the most important parts.

READ ↓

**Class name & purpose**

Key public methods

Supporting logic

Private helper details

# 03

## Design principles

A shared vocabulary — KISS, DRY, YAGNI, SOLID, and the Law of Demeter.

# Keep it simple, don't repeat, don't over-build

## KISS

### Keep It Simple

The simplest approach that works.  
Complexity is a debt paid by every future reader.

## DRY

### Don't Repeat Yourself

One authoritative place per fact.  
Duplication means many places to fix one bug.

## YAGNI

### You Aren't Gonna Need It

Don't build for an imagined future. Unused code is dead weight — read, tested, maintained, unused.

Einstein, on KISS: *"If you can't explain it simply, you don't understand it well enough."*

# Five rules for classes that bend, not break

- S Single Responsibility**  
A class should have exactly one reason to change. *A Report that both computes figures and formats HTML answers to two teams — split it in two.*
- O Open / Closed**  
Open for extension, closed for modification. *Add a payment type by adding a class, not by editing a switch that already works — so you never risk shipped code.*
- L Liskov Substitution**  
A subclass must work wherever its parent does, without surprising the caller. *A Square that breaks Rectangle's width/height contract violates it.*
- I Interface Segregation**  
Many small, specific interfaces beat one large one, so a class never implements methods it doesn't need. *A read-only client shouldn't depend on `write()`.*
- D Dependency Inversion**  
Depend on abstractions, not concrete classes. *Take a `PaymentGateway` interface, not a `Stripe` class — the same constructor seam that made code testable in Lecture 2.*

# One class, one reason to change

BEFORE — math and formatting mixed

```
// TWO reasons to change this class
class Report {
    double total() { ... } // business
    String toHtml() {      // presentation
        return "<h1>" + total() + "</h1>";
    } }
}
```

AFTER — one job each

```
// the numbers
class Report { double total() { ... } }

// how it looks
class ReportHtmlView {
    String render(Report r) {
        return "<h1>" + r.total() + "</h1>"; } }
}
```

A change to the HTML now can't possibly break the totals — the two reasons to change live in two classes.

# Extend by adding, not by editing

BEFORE — a new shape means editing this

```
// closed for extension: risky to touch
double area(Shape s) {
    return switch (s.kind()) {
        case CIRCLE -> PI*s.r()*s.r();
        case SQUARE -> s.side()*s.side();
    }; };
```

AFTER — a new shape is a new class

```
interface Shape { double area(); }
class Circle implements Shape {
    public double area() { return PI*r*r; } }

// NEW — nothing above is edited
class Triangle implements Shape {
    public double area() { return 0.5*b*h; } }
```

Adding **Triangle** touches no existing, tested code — so it can't break what already ships.

# A subclass must keep its parent's promises

## VIOLATION – Square breaks Rectangle

```
class Rectangle {
  int w, h;
  void setWidth(int x) { w = x; }
  void setHeight(int x) { h = x; } }
class Square extends Rectangle {
  // width also changes height – surprise!
  void setWidth(int x) { w = x; h = x; } }
```

## FIX – don't force the "is-a"

```
// a caller of Rectangle expects:
r.setWidth(5); r.setHeight(4);
assert r.area() == 20; // Square gives 16

// model them as siblings, not parent/child
interface Shape { int area(); }
class Rectangle implements Shape { ... }
class Square implements Shape { ... }
```

If substituting the subclass surprises the caller, the inheritance is wrong — even when "a square is a rectangle" sounds true.

# Don't force a class to implement what it can't

## BEFORE – one fat interface

```
interface Document {  
    String read();  
    void write(String s); }  
  
class ReadOnlyDoc implements Document {  
    public String read() { ... }  
    public void write(String s) {  
        throw new UnsupportedOperationException(); } } // smell
```

## AFTER – split by capability

```
interface Readable { String read(); }  
interface Writable { void write(String s); }  
  
// implements only what it can honour  
class ReadOnlyDoc implements Readable {  
    public String read() { ... } }  
  
// a read-only client depends on Readable only
```

The tell-tale sign of a fat interface: a method stubbed out with **throw new UnsupportedOperationException()**.

# Depend on an interface, not a concrete class

BEFORE — welded to Stripe

```
class OrderService {  
    // builds its own concrete dependency  
    final StripeClient gw = new StripeClient();  
    boolean checkout(double amt) {  
        return gw.charge(amt); } }  
  
// can't swap providers, can't test offline
```

AFTER — inject an abstraction

```
interface PaymentGateway { boolean charge(double a); }  
class OrderService {  
    final PaymentGateway gw;  
    OrderService(PaymentGateway gw) {  
        this.gw = gw; } // injected  
    boolean checkout(double amt) {  
        return gw.charge(amt); } }
```

That constructor is the **seam** from Lecture 2 — a test passes a fake gateway; production passes the real one.

# Don't talk to strangers

```
// Train wreck – knows the whole graph  
var city = order.customer().  
    .getAddress().getCity();
```

```
// Ask, don't reach  
var city = order.getCustomerCity();
```

## A method should only call:

- methods on **itself** (this)
- objects it **created**
- objects passed in as **parameters**
- its own **direct fields**

A chain couples the caller to the entire object graph — refactoring anywhere breaks it.

# 04

## Code smells

Patterns that don't always mean a bug — but signal code that will get harder to maintain.

# Eight smells worth recognising

| Smell                      | What it signals                                      |
|----------------------------|------------------------------------------------------|
| Long Method                | Over 20–30 lines — usually several responsibilities  |
| Large Class                | Does too much; violates Single Responsibility        |
| Duplicate Code             | The same logic in more than one place                |
| Long Parameter List        | More than 2–3 params — a missing abstraction         |
| Feature Envy               | A method uses another class's data more than its own |
| Dead Code                  | Never called — dead weight that misleads readers     |
| Magic Numbers / Strings    | Literal values with no explanation in the logic      |
| Deeply Nested Conditionals | More than 2–3 levels — use early returns             |

# Put the logic where the data lives

**BEFORE** – Invoice reaches into Customer

```
class Invoice {
  double total(Customer c) {
    double d = c.tier()==GOLD ? 0.10
      : c.tier()==SILVER ? 0.05 : 0;
    return amount * (1 - d); }
}
```

**AFTER** – Customer owns its discount

```
class Customer {
  double discountRate() {
    return switch (tier) {
      case GOLD -> 0.10;
      case SILVER -> 0.05;
      default -> 0.0; }; } }
// Invoice: amount * (1 - c.discountRate())
```

# Flatten with early returns

BEFORE — drifting rightward

```
String classify(int age, boolean lic) {  
  if (age >= 18) {  
    if (lic) return "can drive";  
    else return "needs a license";  
  } else return "too young";  
}
```

AFTER — same logic, no nesting

```
String classify(int age, boolean lic) {  
  if (age < 18) return "too young";  
  if (!lic) return "needs a license";  
  return "can drive";  
}
```

Handle the exceptional cases first and return early — what remains at the bottom is the main path, un-indented.

THE WARNING

# Code smells compound

One smell is a minor concern. **Several in the same class** signal lost cohesion — and it will grow exponentially more expensive to modify.

Address smells incrementally during normal work — don't wait for a dedicated cleanup sprint that never comes.

# 05

## A refactoring catalogue

Named, repeatable moves — each a concrete answer to "how do I actually fix this smell?"

# Change the structure, not the behaviour



## Behaviour-preserving

Rearranging internal structure while the observable behaviour stays exactly the same.



## Only on tested code

You can only safely rearrange code you can already prove still works — hence TDD's Refactor step comes *after* Green.



## Small, reversible steps

One move at a time, run the tests, confirm nothing broke — then the next.

Refactor five things at once and then run the tests, and a failure tells you nothing about **which** change broke it.

# Separate the **what** from the **how**

**BEFORE** — one long method

```
void printInvoice(Invoice inv) {  
    double sub = inv.items()  
        .stream().mapToDouble(Item::price).sum();  
    double tax = sub * 0.20;  
    print("Subtotal: " + sub);  
    print("Tax: " + tax);  
    print("Total: " + (sub + tax)); }  

```

**AFTER** — the math has a name

```
void printInvoice(Invoice inv) {  
    InvoiceTotal t = calculateTotal(inv);  
    print("Subtotal: " + t.subtotal());  
    print("Tax: " + t.tax());  
    print("Total: " + t.total()); }  
  
// calculateTotal(inv) does the sum + tax  

```

The direct fix for **Long Method** — and for a nested branch, extract each branch into its own named method.

# Split a class along its seam

BEFORE — two responsibilities

```
// payroll AND address formatting
class Employee {
    double calculatePay() { ... }
    String formatMailingAddress() { ... }
}
```

AFTER — one job each

```
class Employee {
    final PayCalculator pay;
    final Address address;
    double calculatePay() { return pay.calculate(this); }
    String formatMailingAddress() { return address.format(); }
}
class PayCalculator { ... }  class Address { ... }
```

The fix for **Large Class** — split along the responsibility boundary instead of trimming in place.

# The smallest, cheapest refactor

```
// Before – what is 10000?  
if (order.total() > 10000)  
    flagForReview(order);
```



```
static final double  
    FRAUD_REVIEW_THRESHOLD = 10_000;  
if (order.total() > FRAUD_REVIEW_THRESHOLD)  
    flagForReview(order);
```

The name states the intent, and the value now lives in one place — the direct fix for the **Magic Numbers** smell.

# Let each type answer for itself

**BEFORE** — a switch every caller repeats

```
double area(Shape s) {  
    return switch (s.kind()) {  
        case CIRCLE -> PI*s.r()*s.r();  
        case SQUARE -> s.side()*s.side();  
    }; }  

```

**AFTER** — the switch disappears

```
abstract class Shape {  
    abstract double area(); }  
class Circle extends Shape {  
    double area() { return PI*r*r; } }  
class Square extends Shape {  
    double area() { return side*side; } }
```

Branching on a **type** is often better as an overridden method — and it kills the **Duplicate Code** the switch spawns at every call site.

# The everyday two, and the golden rule

## Rename

A name that was clear when written can stop being clear as the system evolves. Renaming costs nothing at compile time and pays back on every read.

## Inline

Extract Method's inverse: when a one-line method or a use-once variable adds indirection without clarity, remove it — fewer hops to follow.



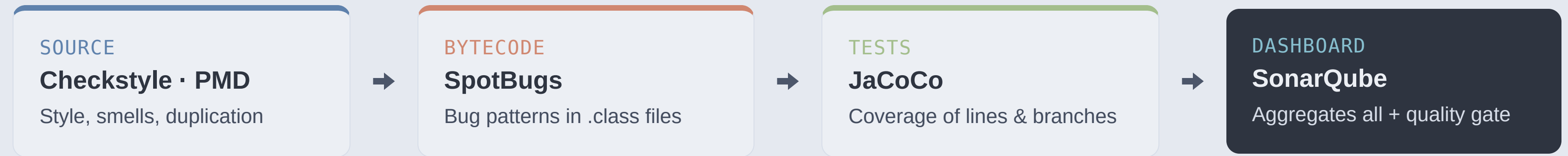
Every move small enough to run the tests **immediately** after. Refactor five things, then test, and a failure won't tell you which one broke it.

# 06

## Java tooling

Good practice needs tool support — enforced automatically, on every push.

# Five tools, four stages of a build



The key distinction: Checkstyle and PMD read your **source**; SpotBugs reads the compiled **bytecode** — so it catches a different class of bug.

# Checkstyle and PMD

## Checkstyle

Enforces **style**: naming, import order, Javadoc, line length, indentation. Configurable rule sets (Google, Sun, custom).  
Runs on source, before compilation.

```
mvn checkstyle:check
```

## PMD

Finds **potential bugs**: unused code, overly complex methods, empty catch blocks, naming violations. Its **CPD** sub-tool detects copy-pasted code.

```
pmd cpd --minimum-tokens 100
```

# SpotBugs and JaCoCo

## SpotBugs

Analyses compiled **bytecode**, not source — so it sees a different class of bug: null dereferences, resource leaks, broken equals/hashCode, infinite loops, security holes. The maintained successor to FindBugs.

## JaCoCo

Measures which lines, branches and methods your test suite actually executes. Generates HTML and XML reports, automatically during **mvn test**.

**Coverage is not quality.** A test that runs a line without asserting anything still counts — coverage is necessary, not sufficient.

# One dashboard, one gate

A unified platform that **aggregates** Checkstyle, PMD, SpotBugs and JaCoCo into a single view.

Tracks metrics **over time**: debt trend, code-smell growth, coverage evolution.

Runs as a cloud service (SonarCloud) or self-hosted.

## QUALITY GATE

- ✓ coverage > 80%
- ✓ no new critical bugs
- ✓ duplication < 3%

A configurable threshold that can **fail the CI pipeline**.

# Quality, enforced on every push



```
stages: [lint, build, test, quality]

checkstyle: { stage: lint,    script: mvn checkstyle:check }
build:      { stage: build,   script: mvn compile }
test:       { stage: test,    script: mvn test }
quality:
  stage: quality
  script: [ mvn spotbugs:check, mvn sonar:sonar ]
```

Cheapest checks first: style before compilation, SpotBugs and Sonar after tests so coverage data is ready. **Fail fast, fix fast.**

---

# Write code the next person can change

01

Readability is the maintenance bill — names over comments.

02

Smells signal debt; refactor in small, tested steps.

03

Tools make quality automatic — enforced on every push.

THE COURSE IN ONE LINE

Maintenance is most of the work — so make change **safe** (tests) and **cheap** (clean code).

Thank you — now to the practical

Questions?