From “AI Spaghetti” to SOLID Architecture

You receive two classes that were fully generated by an AI assistant:

text

CopyEdit

src/main/java/aibadcode/InvoiceService.java (see below)

src/main/java/aibadcode/Invoice.java (see below)

They *compile*, but the design is a minefield of technical-debt traps.  
 Your task is to **surface those problems, explain why they matter, then refactor the codebase into a clean, SOLID-compliant solution.**

### **🛠 Part 1 — Code-Review & Problem Discovery**

1. **Clone the starter repo** (or drop the two files into a new project).
2. **Run a compile+static-analysis pass** with any tools you like (SonarLint, IntelliJ inspections, IDE warnings, etc.).
3. **Write a report (review.md)** that lists every significant issue you find.  
    Organise findings in a table with the columns below:

| **#** | **Category** | **Issue description** | **SOLID / CWE ref** | **Real-world impact** |
| --- | --- | --- | --- | --- |
|  | Design / Security / Scalability / Concurrency / Testability | *one sentence per issue* | e.g. SRP-01, CWE-798 | *why this hurts in prod* |

*Minimum*: identify **10** distinct issues (there are more than 15).  
 *Hints*: hard-coded secrets, God class, thread saf-ety, unchecked SQL, etc.

### **🛠 Part 2 — Test Harness (Expose the Pain)**

1. Create **unit or integration tests** that *fail* because of the discovered issues.  
    *Examples*
   * Concurrency test that shows HashMap race conditions.
   * Mutation test proving that hard-coded JWT secret can be read.
   * Mock email session returning null → NullPointerException.

Deliver **src/test/java/\*\*** with at least **3 failing tests** that highlight different categories of flaws.

### **🛠 Part 3 — Refactor to SOLID & Patterns**

1. **Slice responsibilities** into ports + adapters (hexagonal / onion style).
2. Apply at least **three** GOF design patterns that suit the problems (e.g. Repository, Strategy, Decorator, Factory).
3. Replace hard-coded values with config injection (no frameworks required—constructor args or env vars are fine).
4. Make the cache thread-safe (choose your own simple implementation; no external libs required).
5. Leave the public use-case API in InvoiceService *intact* (so tests can be re-run).

### **📦 Deliverables**

css

CopyEdit

/review.md ← problem table

/src/main/java/\*\* ← refactored classes & interfaces

/src/test/java/\*\* ← failing tests BEFORE, green tests AFTER

/README.md ← how to build & run

### **✅ Evaluation Rubric**

| **Weight** | **Criterion** | **Excellent** | **Needs Work** |
| --- | --- | --- | --- |
| 40 % | **Problem discovery** | ≥12 legitimate issues across ≥4 categories, clearly explained | <8 issues, vague impact |
| 30 % | **Refactor quality** | SOLID honoured; adapters swap cleanly; secrets externalised | God class remains; tight coupling |
| 20 % | **Tests** | Three+ focused tests fail pre-refactor, all green post-refactor | Tests missing or still fail |
| 10 % | **Clarity** | Readable code & docs, Javadoc on every public type | Sparse comments, unclear README |

### **⏱ Suggested Timeline**

| **Timebox** | **Activity** |
| --- | --- |
| **45 min** | Static review + fill in review.md |
| **30 min** | Write failing tests |
| **90 min** | Incremental refactor (commit after each green build) |
| **15 min** | Final green-red-green test run & README polish |

### **Starter Files (unchanged from prompt)**

<details> <summary><code>InvoiceService.java</code></summary>

java

CopyEdit

package aibadcode;

/\* … full code exactly as provided in the prompt … \*/

</details> <details> <summary><code>Invoice.java</code></summary>

java

CopyEdit

package aibadcode;

import java.math.BigDecimal;

public class Invoice {

private final long id;

private final BigDecimal amount;

public Invoice(long id, BigDecimal amount) {

this.id = id;

this.amount = amount;

}

public long id() { return id; }

public BigDecimal amount(){ return amount; }

}

</details>

### **📚 What you’ll reinforce**

* Reading AI-generated code with a critical eye.
* Mapping concrete smells back to SOLID principles.
* Driving refactor with failing tests (red-green-refactor cycle).
* Explaining design trade-offs like a senior engineer.

Good luck—show that AI code can become production-worthy in your hands!