Don’t Let Your Code Skip Leg Day

🏋 *Train your coding skills like your gym routine*

Apipana.io® CODING SOCIETY

#pulpocon - 2025
#gastrotech

Who I am?

Jesús María Villar Vázquez

Just a Developer working for @apipana.io #wearehiring

20 years of experience

I love Testing, eXtreme Programming and Coding

geeksusma

Object Calisthenics

9 rules for a better object-oriented code

What does “Calisthenics” mean?

🧘 What is Calisthenics?

"Calisthenics" = bodyweight training. No machines. No shortcuts. Just movement and control.

💪 It builds strength through *discipline and repetition*.

🏗️ In programming, Object Calisthenics means:

→ Training your design muscles
→ Practicing good habits, the hard way
→ No procedural “shortcuts”

🧠 Like going to the gym... for your code

What is Object Calisthenics?

A set of 9 rules to help you write better object-oriented code.

🧠 Designed to challenge your habits.
💡 Like fitness drills — strict, but for your own good.
📦 Helps produce modular, clean, testable code.

Created by: Jeff Bay (in 2005)
First introduced in: "Object Calisthenics" chapter of the book **"The ThoughtWorks Anthology" (2008)**

Why Should You Care?

Because "high quality code" is not just an unicorn. It is a goal easy to achieve with discipline

✅ Improves readability and maintainability
✅ Encourages better object modeling
✅ Discourages primitive obsession and procedural style
✅ Trains your mind to **think in objects**, not just classes

👴 Old habits die hard. These rules break them.

Are these rules mandatory?

❌ NO — These are **exercises**, not dogma.
✅ YES — They push your design muscles to the limit.

You can break the rules later...
But first, learn what they teach.

🙃 *“Learn the rules like a pro, so you can break them like an artist.”*
— Pablo Picasso

🧱 Rule #1: Only One Level of Indentation per Method

💡 **Why?**
Deep nesting = mental overhead. Flat methods = clarity.

🧠 **Takeaway**: One level of indentation forces method extraction, which leads to focused, reusable logic.

🚫 **Bad Example:**

```
void process(List<Order> orders) {
    for (Order order : orders) {
        if (order.isValid()) {
            if (!order.isExpired()) {
                System.out.println("Processing " +
order);
            }
        }
    }
}
```

✅ **Good Example:**

```
void process(List<Order> orders) {
    for (Order order : orders) {
        processIfEligible(order);
    }
}

void processIfEligible(Order order) {
    if (!isEligible(order)) return;
    System.out.println("Processing " + order);
}
```

🧱 Rule #2: Don’t use the “else” keyword

💡 **Why?**
else is often a crutch. It hides dual responsibilities and makes logic harder to test. Leads to “spaghetti-code”

🧠 **Takeaway**: Eliminate branching where possible. Use early returns or polymorphism.

🚫 **Bad Example:**

```
String getDiscount(Customer customer) {
    if (customer.isPremium()) {
        return "20%";
    } else {
        return "0%";
    }
}
```

✅ **Good Example:**

```
String getDiscount(Customer customer) {
    if (customer.isPremium()) return "20%";
    return "0%";
}
```

pulpocon - 2025

🧱 Rule #3: Wrap all primitives and Strings

💡 **Why?**
Primitives have no behavior and no identity. Wrapping them gives meaning and validation. Also it makes the act of reading closer to the human

🧠 **Takeaway**: Value objects improve safety, expressiveness, and encapsulation.

🚫 **Bad Example:**

```
public void register(String name, int age) {
    if (name.length() < 3) throw new
IllegalArgumentException();
}
```

✅ **Good Example:**

```
public class Name {
    private final String value;

    public Name(String value) {
        if (value.length() < 3) throw new
IllegalArgumentException();
        this.value = value;
    }
}
```

🧱 Rule #4: First-Class Collections

💡 **Why?**
A collection is not a bucket. Add behavior and rules to it. Go for immutability to prevent side effects.

🧠 **Takeaway**: Collections should be encapsulated in their own class — no public `.get()` allowed. Each mutation returns a **new immutable object**.

🚫 **Bad Example:**

```
class Team {
    List<Player> players;

    public List<Player> getPlayers() {
        return players;
    }

    public void add(Player player) {
        players.add(player);
    }
}
```

```
public class Team {
    private final Players players;

    public Team(Players players) {
        this.players = players;
    }

    public Team addPlayer(Player player) {
        return new Team(players.add(player));
    }

    public Optional<Player> findByName(String
name) {
        return players.findByName(name);
    }
}
```

✅ **Good Example:**

```
public class Players {
    private final List<Player> players;

    public Players(List<Player> players) {
        this.players = List.copyOf(players);
// immutable copy
    }

…

    public Optional<Player> findByName(String
name) {
        return players.stream()
            .filter(p ->
p.name().equalsIgnoreCase(name))
            .findFirst();
    }
}
```

pulpocon - 2025

🧱 Rule #5: One dot per line

💡 **Why?**
Chained dots (`a.b().c().d()`) imply strong coupling and a violation of the Law of Demeter.

🧠 **Takeaway**: Talk to your friends, not your friends' friends. Hide navigation behind meaningful methods.

🚫 **Bad Example:**

```
order.getCustomer().getAddress().getCity();
```

✅ **Good Example:**

```
order.getShippingCity();
```

🧱 Rule #6: Don’t abbreviate

💡 **Why?**
Abbreviations save keystrokes but cost clarity — especially in teams. You are **coding** not sending a tweet.

🧠 **Takeaway**: Be clear. Let the IDE help you type — that’s what it’s for. Respect the **DSL** and the **Ubiquitous Language.**

🚫 **Bad Example:**

```
public class Cstmr {
    Addr addr;
}
```

✅ **Good Example:**

```
public class Customer {
    Address address;
}
```

🧱 Rule #7: Keep All Entities Small

💡 **Why?**
Big classes do too much. They're hard to understand, test, and change.

🧠 **Takeaway**: Small classes are composable. The brain likes bite-sized pieces.

🚫 **Bad Example:**

```
public class Invoice {
    private List<Item> items;
    private Customer customer;
    private LocalDate dueDate;

    public void addItem(Item item) {
        ..
    }
    public double calculateTotal() {
        …
    }
    public void sendEmailNotification() {
        // email sending logic
    }
    public void print() {
        // printing logic
    }
    public void archive() {
        // archive logic
    }
}
```

🧱 A class should:

- Do **one** thing
- Be under **50 lines** (as an exercise)
- Have few responsibilities

🧱 Rule #8: No Classes with More Than Two Instance Variables

💡 **Why?**
More variables → more state → more complexity. Classes are free!

🧠 **Takeaway**: More than 2 fields? Time to refactor. Break into meaningful concepts.

🚫 **Bad Example:**

```java
public class Order {
    Product product;
    Customer customer;
    Address shippingAddress;
    LocalDate deliveryDate;
}
```

✅ **Good Example:**

Better Design:

- Extract `ShipmentDetails`
- Extract `CustomerInfo`

🧱 Rule #9: No Getter/Setter Allowed

💡 **Why?**
Exposing state means you're just writing structs, not objects. Objects should do things, not just carry data.

🧠 **Takeaway**: Tell don't ask. (Tell objects what to do — don't ask them about their internals and act for them)

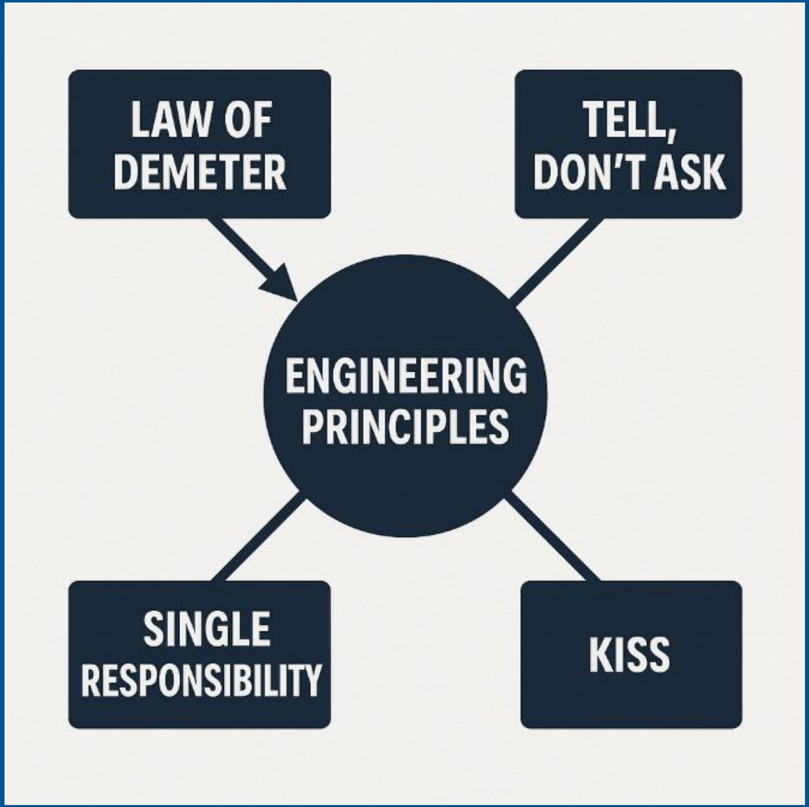
🚫 **Bad Example:**

```
public String getName() {
    return name;
}
```

✅ **Good Example:**

```
public void greet() {
    System.out.println("Hello, my
name is " + name);
}
```

🧠 Engineering Principles



🧱 Principle #1: Law of Demeter

💬 "Only talk to your immediate friends."

Also known as the **"Principle of Least Knowledge"**
Avoid **chain calls** and indirect access to foreign objects.

🧠 Takeaway:

Avoid "train wrecks":
`a.getB().getC().getD()`
Let objects **do the work**, not expose their guts.

🚫 Things to avoid:

Deep coupling: you need to know the full object graph.

Fragile to changes in intermediate classes.

✅ Better if:

You **delegate** behavior to the owner of the data.

Promotes **encapsulation** and **stable APIs**

🧱 Principle #2: Tell Don’t Ask!

💬 **"Don’t ask an object for data, then make decisions.**
Tell it what to do."

🧠 **Takeaway:**

Smart objects, dumb services.

Design **active objects** that **do**, not **get and wait**.

🚫 **Things to avoid:**

The **caller takes responsibility** for decisions.

The **logic is spread** across multiple places.

✅ **Better if:**

Encapsulates logic inside the domain

Makes the object **more responsible**, **less dumb**

🧱 Principle #3: **KISS.** Keep It Simple and Stupid

💡 Simpler code is easier to understand, test, and change.

Simplicity often beats elegance.
Avoid “clever” solutions unless truly necessary.

🚫 Smells:

Hard to read

Difficult to debug

Over-optimized

🧠 Takeaway:

"Clever" is the opposite of **clear**.

Avoid → Over Engineering and Premature Optimizations

Explore → YAGNI

✅ Better if:

Easier to **read, test, and modify**

Separation of concerns

🧱 Principle #4: **Single Responsibility Principle (SRP)**

💡 **Every class or method should have one reason to change**

When responsibilities **multiply**, maintenance gets messy.

🧠 **Takeaway:**

"When everything changes for one reason, you're in trouble."

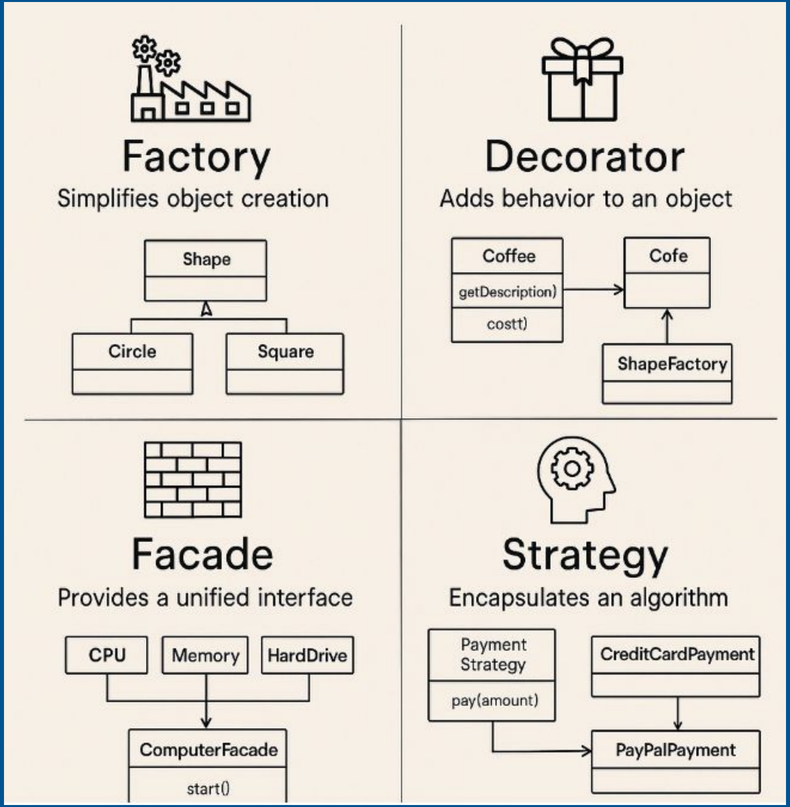
🚫 **Bad:**

```
class ReportService {

    public String generateReport(Data d) { ... }

    public void saveToFile(String report) { ... }

    public void sendEmail(String report) { ... }

}
```

✅ **Good:**

```
class ReportGenerator {public String generate(Data
d) { ... }}

class ReportRepository {public void save(String
report) { ... }}

class ReportNotificator {public void send(String
report) { ... }}
```

🧩 Design Patterns



🏭 Factory Pattern

💡 What it solves:

When object creation logic is complex, **centralize and abstract** it.

Useful to hide construction logic and keywords.

So the way of creating an object is more human friendly.

🧠 Takeaway:

Simplify object creation and reduce coupling between code and classes.

```
import java.util.UUID;

public class Id {

  private final String value;

  private Id(String value) {
    this.value = value;
  }

  public static Id random() {
    return new
Id(UUID.randomUUID().toString());
  }

  public static Id fixed(String value) {
    // TODO: Add validation logic for fixed
IDs
    return new Id(value);
  }

  @Override
  public String toString() {
    return value;
  }
}
```

🎁 Decorator Pattern

💡 What it solves:

When you want to add behavior **dynamically**, without subclassing.

Is a good example of "**composition over inheritance**"

You want to **wrap functionality** around an object

🧠 Takeaway:

Add responsibilities without rewriting code — like **wrapping a gift**.

```
interface Coffee {
    String getDescription();
    double cost();
}

class SimpleCoffee implements Coffee {
    public String getDescription() { return "Coffee";
}
    public double cost() { return 2.0; }
}

class MilkDecorator implements Coffee {
    private final Coffee base;
    public MilkDecorator(Coffee base) { this.base =
base; }

    public String getDescription() { return
base.getDescription() + ", milk"; }
    public double cost() { return base.cost() + 0.5; }
}

Coffee coffee = new MilkDecorator(new SimpleCoffee());
System.out.println(coffee.getDescription()); //
Coffee, milk
```

🧱 Facade Pattern

💡 What it solves:

When a system is **too complex** — give it a clean, unified interface.

Chain calls to subsystems as if you were cooking a recipe.

You want to reduce **dependency** on low-level details

🧠 Takeaway:

Facade cleans up the mess — **like a friendly receptionist** to your system.

```
class CPU { void freeze() {} void jump(long position)
{} void execute() {} }
class Memory { void load(long position, byte[] data)
{} }
class HardDrive { byte[] read(long lba, int size) {
... } }

class ComputerFacade {
    private final CPU cpu = new CPU();
    private final Memory memory = new Memory();
    private final HardDrive hd = new HardDrive();

    public void start() {
        cpu.freeze();
        memory.load(0, hd.read(0, 1024));
        cpu.jump(0);
        cpu.execute();
    }
}

ComputerFacade computer = new ComputerFacade();
computer.start(); // simple API for a complex operation
```

🧠 Strategy Pattern

💡 What it solves:

When you need to **choose behavior at runtime**,

keep algorithms **interchangeable**, because you have **multiple algorithms** for a task.

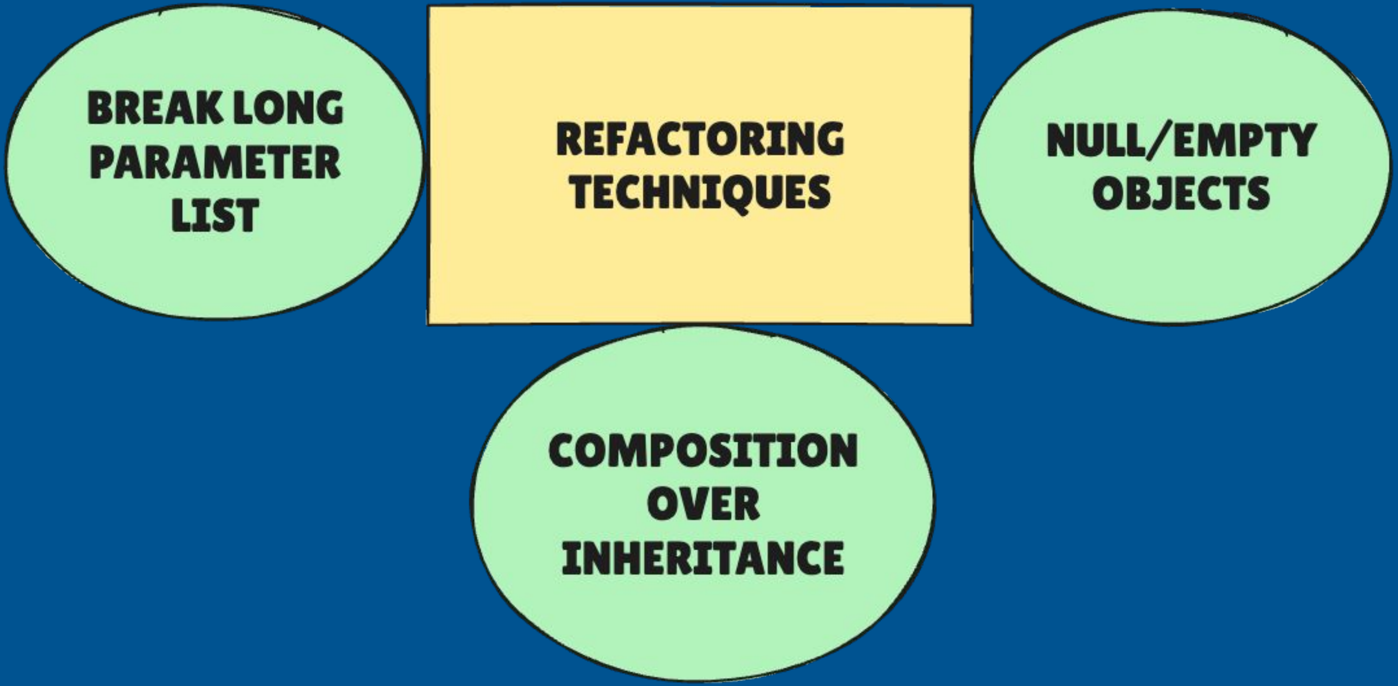
You want to **encapsulate logic** and avoid conditionals

🧠 Takeaway:

Cleanly switch logic without `if-else` soup.

```java
interface PaymentStrategy {
    void pay(double amount);
}

class CreditCardPayment implements PaymentStrategy {
    public void pay(double amount) { /* logic */ }
}

class PayPalPayment implements PaymentStrategy {
    public void pay(double amount) { /* logic */ }
}

class Checkout {
    private final PaymentStrategy payment;

    public Checkout(PaymentStrategy payment) {
        this.payment = payment;
    }

    public void process(double amount) {
        payment.pay(amount);
    }
}

Checkout checkout = new Checkout(new PayPalPayment());
checkout.process(100.0);
```

🧹 Refactoring Techniques



📦 Break Long Parameter Lists

🧠 The Problem:

Too many parameters lead to:

- Confusion 🤯
- Wrong order bugs
- Difficult testing and usage

✅ Refactoring Options:

1. **Group related values** into objects
2. Use a **Parameter Object**
3. Leverage **Builder Pattern** when values are optional

💬 Motivation:

"Code should read like a story,
not like a tax form."

⚙️ Composition over Inheritance

🧠 The Problem:

- Deep inheritance chains create **rigid** and **fragile** hierarchies.

```
class Bird {
    void fly() {}
}

class Ostrich extends Bird {
    // oops… can't fly
}
```

✅ Refactoring Options:

Favor **composition**: build objects by combining behavior.

💬 Motivation:

"Don't ask 'is-a' — ask 'can-do'."

```
interface Movement {
    void move();
}

class Flying implements Movement {
    public void move() { fly(); }
}

class Walking implements Movement {
    public void move() { walk(); }
}

class Ostrich {
    private final Movement movement = new Walking();
}
```

🕳️ Introduce Null/Empty Object

🧠 The Problem:

- `null` is a liar — it pretends to be a valid value until runtime chaos erupts. Only brings cognitive load

```
User user = repo.findById(id);
if (user != null) {
  //logic goes here
}
..
User user = repo.findById(id);
if (!user.isEmpty()) {
  //logic goes here
}
```

✅ Refactoring Options:

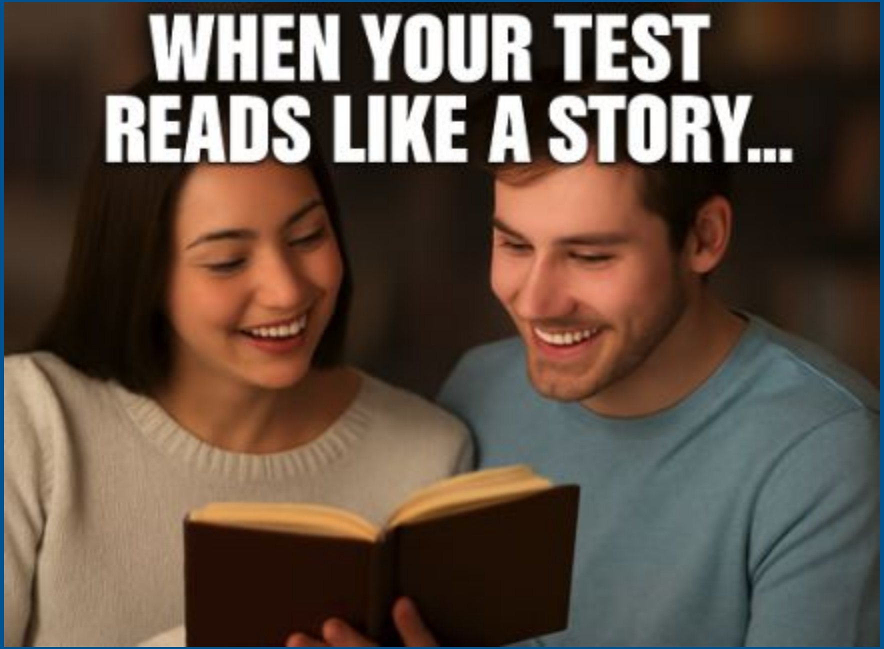
Instead of returning `null`, return an **Empty Object** or **Null Object** with safe defaults.

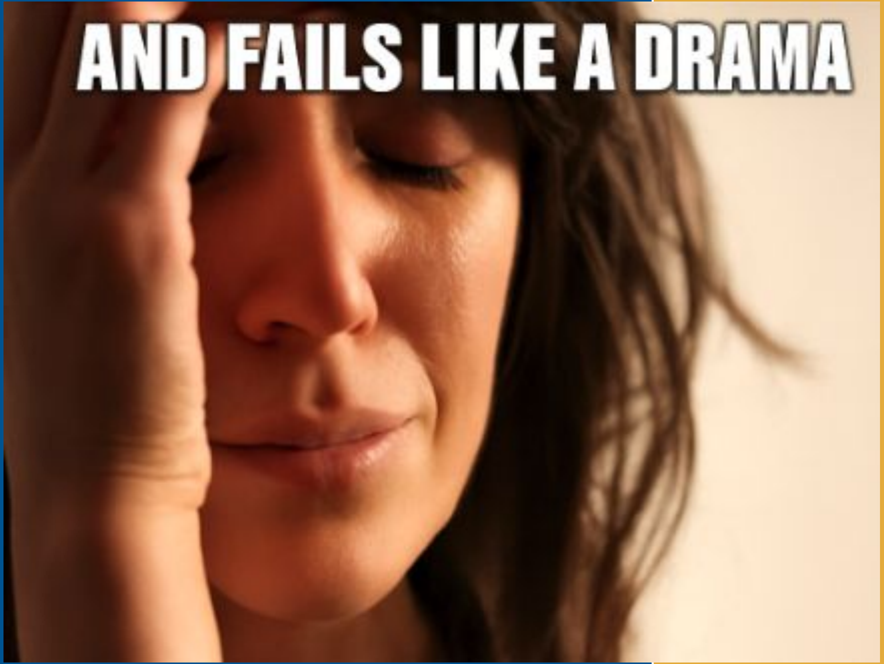
💬 Motivation:

“Make illegal states unrepresentable.”

```
final class User {

  private static final User EMPTY = new User("");
  private final String name;
  private User(String name) {
    this.name = name;
  }
  static User empty() {
    return EMPTY;
  }
  boolean isEmpty() {
    return this == EMPTY;
  }
}
```

✅ Simple Design = Testable Design / Testing Techniques





💬 Introduce DSL (Domain-Specific Language)

🧠 The Problem:

Tests often look like **low-level scripts**, hard to read and even harder to maintain.

✅ Solution:

Create a **test DSL** that speaks the domain's language.

💬 Motivation:

> "Tests should describe behavior, not mechanics."

🚫 Bad Example:

```
User user = new User("John", "Doe",
"john@example.com");

assertTrue(user.getEmail().contains("@"));
```

✅ Good Example:

```
User user = aRegisteredUser(); //helper

assertThat(user).hasValidEmail();
```

👩‍👧‍👦 Mother Objects

🧠 The Problem:

Test data becomes repetitive, verbose, or unclear.

✅ Solution:

Create **Mother Objects** — factories that produce typical test instances.

💬 Motivation:

“Test clarity is as important as production clarity.”

🚫 **Bad Example:**

```
PersonalData data = PersonalData.with("Joey", "Ramone", "joey-passport");
```

✅ **Good Example:**

```
PersonalData data = RamonesMotherObject.joey();

public class RamonesMotherObject {

    public static final String JOEY = "Joey";

    public static final String JOEY_PASSPORT = "joey-passport";

    public static final String RAMONE = "Ramone";

    public static PersonalData joey() {

        return PersonalData.with(JOEY, RAMONE, JOEY_PASSPORT);

    }

}
```

🔍 Matchers

🧠 Why Matchers?

They allow **declarative assertions** that improve **readability and intent**.

✅ Solution:

Use custom matchers to keep the **DSL**

💬 Motivation:

"Good matchers remove boilerplate from your tests

— and expose what really matters."

✅ AssertJ:

```
assertThat(user.getEmail()).isNotBlank()

.contains("@");
```

✅ Custom Matcher:

```
assertThat(user).hasValidEmail();

public static Condition<User> hasValidEmail() {

    return new Condition<>(u ->
u.getEmail().contains("@"), "valid email");

}
```

🧰 Helpers

🧠 Purpose:

Is to **keep you infra setup out from the test** to put the focus in what is important.

💬 Motivation:

"Good helpers make tests short and simples and agnostics from protocols or external systems"

✅ Example for Http Controller:

```
public void checkGetForSingleObject(String endpoint, String
expectedBody) throws Exception {
        String contentAsString =
                doGet(endpoint)
                        .andExpect(status().isOk())
                        .andReturn()
                        .getResponse()
                       .getContentAsString(StandardCharsets.UTF_8);
        assertThat(contentAsString).isEqualTo(expectedBody);
    }
    private ResultActions doGet(String endpoint) throws Exception {
        return
this.mockMvc.perform(get(endpoint).contentType(MediaType.APPLICATIO
N_JSON));
    }
```

✅ Make Helper a Matchers goes together:

```
public static void
assertResponseContainsAProblem(MockHttpServletResponse response,
HttpStatus expectedStatus, ErrorDTO expectedError) throws
JsonProcessingException, UnsupportedEncodingException {

        ObjectMapper objectMapper = new ObjectMapper();

        assertThat(response)
                .isNotNull()
                .hasFieldOrPropertyWithValue("status",
expectedStatus.value());

assertThat(response.getHeaders("Content-Type").get(0)).isEqualTo(Me
diaType.APPLICATION_PROBLEM_JSON.toString());

assertThat(response.getContentAsString()).isEqualTo(objectMapper.wr
iteValueAsString(expectedError));
    }
```

🔁 TDD (Test-Driven Development)

🧠 Why:

Behaviours! Behaviours! not implementation details. And baby steps

✅ Cycle:

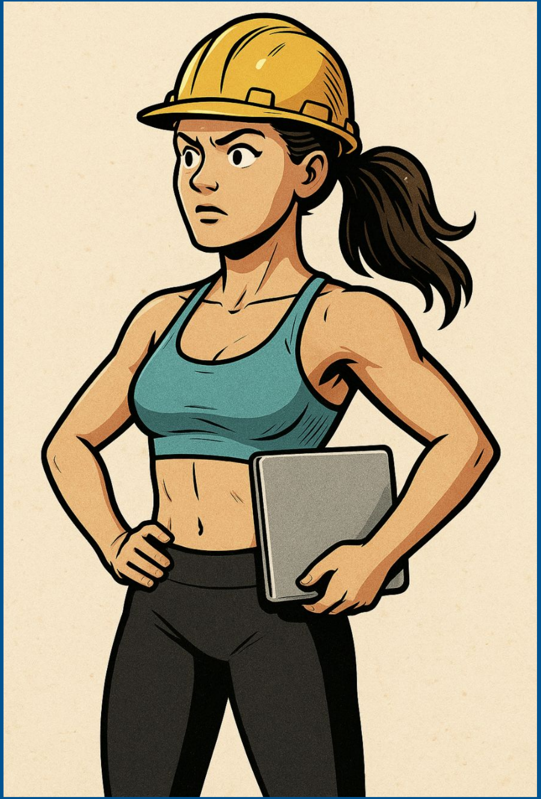
1. **Red** — Write a test, watch it failing
2. **Green** — Make it pass (KISS)
3. **Refactor** — Improve

💬 Motivation:

“Things go better in short steps. Keep the feedback in the loop!”

Questions?

Exercise - Let’s training our brains



Simple but a real world use case

We want to save an employee

The Personal Data (Name and Last Name) is mandatory and can't be empty

The Id will be fixed by the System (not the persistence layer)

Ensure the Employee is not duplicated, use the passport to ensure that validation

Assume the Controller Layer/Presentation/Upper Layer is already implemented