Understanding anti-patterns

Anti-Patterns in depth God Object, singleton, Spaghetti Code, Lava Flow with real life example and analogies

Understanding **anti-patterns** is as important as knowing the right patterns—because these are common *bad solutions* to recurring problems, often found in legacy systems or rushed projects.

Let's explore the following **anti-patterns in depth**, each with:

- Definition
- **©** Real-life analogy
- Code example or symptom
- Why it's bad
- How to fix

1. God Object / God Class

Definition:

A **God Object** is a class that knows too much or does too much. It violates **Single Responsibility Principle (SRP)** and **Encapsulation**.

© Analogy:

Imagine a "Manager" in a company who:

- Does HR interviews,
- Approves accounts,
- Fixes network issues,
- Writes code.
- Prepares chai

This "superhuman" is a **bottleneck** and error-prone.

🦞 Example in Java:

```
java
CopyEdit
public class OrderManager {
    public void placeOrder() { /* logic */ }
    public void cancelOrder() { /* logic */ }
    public void calculateDiscount() { /* logic */ }
    public void sendConfirmationEmail() { /* logic */ }
    public void generateInvoicePDF() { /* logic */ }
}
```

🎽 Why it's bad:

Hard to test

- No modularity
- Break one thing, everything breaks
- Violates SRP, OCP

Solution:

Break into smaller classes:

```
java
CopyEdit
class OrderService { void placeOrder() {} }
class DiscountService { void calculateDiscount() {} }
class EmailService { void sendEmail() {} }
class InvoiceService { void generatePDF() {} }
```

2. Overused Singleton (Anti-pattern)

Definition:

While Singleton is a creational pattern, it becomes an **anti-pattern** when:

- It's misused as a global variable.
- It maintains state, becomes thread-unsafe or unmockable.

© Analogy:

Imagine a **printer room** with only one printer (singleton), and 1000 people need to print. Instead of scaling, we queue. Worse, people *change printer config* globally!

Example:

```
java
CopyEdit
public class ConfigManager {
    private static ConfigManager instance = new ConfigManager();
    private String env; // mutable!
    public static ConfigManager getInstance() { return instance; }
    public void setEnv(String env) { this.env = env; } // GLOBAL SIDE EFFECT!
}
```

Why it's bad:

- Global mutable state
- Difficult to test/mock
- Hidden dependencies
- Tight coupling

Solution:

- Use Spring Bean (singleton by default)
- Use **constructor injection** instead of static access
- Mark as @Component or @Service and inject via @Autowired / constructor

3. Spaghetti Code

Definition:

Code with **no structure**, tangled with **goto-like flow**, excessive if-else, no modularization.

© Analogy:

Imagine **cooked noodles** — tangled, can't trace one from start to end.

Example:

Why it's bad:

- Hard to test
- Difficult to debug
- No reuse

Solution:

- Apply Strategy Pattern
- Modularize into small, testable units
- Use polymorphism, not if-else



Definition:

Dead but dangerous code—legacy or experimental code that still exists, nobody knows why, but developers are scared to delete it.

© Analogy:

Like **lava** that has hardened and formed terrain, this old code still shapes current architecture.

Example:

```
java
CopyEdit
// Added by Rajeev in 2012. Don't remove. Breaks the world.
if (config.get("enableLegacyFlow") == true) {
    legacyProcess();
}
```

No one knows:

- Why it's there
- What it does
- What breaks if we remove it

🎽 Why it's bad:

- Technical debt
- Risk of bugs
- Blocks refactoring
- No ownership

Solution:

- Write tests to gain confidence
- Refactor in iterations
- Use **code coverage tools** to remove unused code
- Conduct code archaeology: git blame, commit history



Anti-Pattern	Root Problem	Fix Suggestion
God Object	Too many responsibilities	Apply SRP, split into smaller classes
C: 1 · Mr	01.1.1	le and the state of the state o

Singleton Misuse Global mutable state, tight coupling Use DI, avoid static access

Spaghetti Code Complex, nested, hard to follow Modular design, use OOP principles

Lava Flow Legacy, unused, risky code Refactor safely with tests

Where You Might See These in Spring Boot

Anti-Pattern Where in Spring Boot Projects

Singleton Misuse Static utility classes Spaghetti Code Controllers with logic

Lava Flow Legacy application.properties, unused beans, XMLs

Would you like a "smell checker" for these anti-patterns in a codebase or a sample project showing them and how to fix?

You said:

give me other popular anti pattern with java examples

ChatGPT said:

Absolutely! Beyond the common ones like God Object and Spaghetti Code, there are several other **popular anti-patterns** that often sneak into Java development—especially in large-scale or rushed projects.

Below is a curated list of **popular Java anti-patterns** with:

- Z Definition
- 💡 Code Example
- \(\begin{align*} \begin{align*} \begin
- How to Fix

1. Tight Coupling

Definition:

Classes are directly dependent on concrete implementations instead of abstractions.

§ Java Example:

```
java
CopyEdit
public class PaymentService {
    private CreditCardProcessor processor = new CreditCardProcessor(); // tight
coupling
}
```

🎽 Why it's bad:

- Hard to test
- Cannot switch implementation
- Breaks Open/Closed Principle

Fix:

Use interfaces + dependency injection (Spring-style):

```
java
CopyEdit
public class PaymentService {
    private final PaymentProcessor processor;

    public PaymentService(PaymentProcessor processor) {
        this.processor = processor;
    }
}
```

2. Premature Optimization

Definition:

Optimizing code before it's clear that performance is an issue.

Java Example:

```
java
CopyEdit
StringBuilder sb = new StringBuilder(1000); // guessed size?
// overly complex logic for no gain
```

Why it's bad:

- Wastes dev time
- Reduces readability
- Micro-optimization may hurt performance

Fix:

- Focus on readability and correctness first
- Profile before optimizing

3. Hardcoding / Magic Numbers

Definition:

Using literal values directly in code.

Java Example:

```
java
CopyEdit
if (status == 3) { // what is 3?
    // process complete
```

```
}
```

🎽 Why it's bad:

- Poor readability
- Hard to maintain

```
Fix:
```

```
java
CopyEdit
public static final int STATUS_COMPLETED = 3;
if (status == STATUS_COMPLETED) { ... }
```

Or use enum.

4. Null Checks Everywhere

Definition:

Scattered if (obj != null) checks instead of proper design.

Java Example:

```
java
CopyEdit
if (order != null && order.getCustomer() != null &&
order.getCustomer().getEmail() != null) {
    // send email
}
```

Why it's bad:

- Verbose
- Fragile
- Missed opportunities for encapsulation

Fix:

• Use Optional, fail-fast constructors, or Null Object pattern.

5. Poltergeist Objects

Definition:

Lightweight objects that serve no real purpose—just pass data around.

Java Example:

java

```
CopyEdit
public class OrderHelper {
    public void process(Order order) {
        orderService.validate(order);
        orderService.persist(order);
    }
}
```

🎽 Why it's bad:

- Extra abstraction
- No real encapsulation or behavior

Fix:

Let the actual service (like OrderService) do its job.

• 6. Cyclic Dependency

Definition:

Two or more classes/services depend on each other.

§ Spring Example:

```
java
CopyEdit
@Service
public class A {
     @Autowired B b;
}
@Service
public class B {
     @Autowired A a;
}
```

Why it's bad:

- Breaks design
- Causes BeanCurrentlyInCreationException in Spring

Fix:

- Use an interface
- Refactor shared logic into a third service

7. Golden Hammer

Definition:

Overusing a favorite pattern or tool for everything.

"To a person with a hammer, everything looks like a nail."

Java Example:

- Always using Singleton even for stateful services
- Using Map instead of proper modeling

Why it's bad:

- Misfit solutions
- Poor maintainability

Fix:

- Choose patterns based on context
- Favor domain modeling

8. Copy-Paste Programming

Definition:

Duplicating logic across multiple places.

? Java Example:

```
java
CopyEdit
// found in multiple classes
String sql = "SELECT * FROM users WHERE id=?";
```

Why it's bad:

- Hard to update
- Inconsistent behavior

Fix:

- Use shared utilities
- Abstract common logic

9. Overengineering

Definition:

Designing complex, unnecessary abstractions before they are needed.

Y Java Example:

```
java
CopyEdit
interface UserStrategyFactoryBuilderProviderSelector { ... }
```

Even though you only need 1 type of user.

Why it's bad:

Increases complexity

• Slows down delivery

Fix:

- YAGNI (You Ain't Gonna Need It)
- Keep it simple until needed

● 10. Too Many Layers (Layer Violation / Layer Obsession)

Definition:

Excessive or unnecessary layers between modules.

Java Example:

java CopyEdit Controller → Manager → Helper → DAO → Util → DB

Why it's bad:

- Hard to debug
- Poor performance
- Reduced cohesion

Fix:

- Use clean architecture: entities, use cases, interfaces
- Avoid over-abstraction

₹ Summary Table

Anti-Pattern	Symptom	Fix
Tight Coupling	Can't change impl	Use interfaces + DI
Premature Optimization	Complex without perf need	Profile before tuning
Magic Numbers	Constants in code	Use enums or named constants
Null Checks Everywhere	Nested null guards	Use Optional, Null Object
Poltergeist	Do-nothing helper classes	Collapse into real services
Cyclic Dependency	Beans depend on each other	Refactor with interfaces or third party
Golden Hammer	Misusing patterns everywhere	Choose tool per context
Copy-Paste Programming	Logic duplicated	DRY – reuse logic
Overengineering	Complex before necessary	Follow YAGNI
Too Many Layers	Layer explosion	Simpler architecture, fewer hops

Would you like me to generate a **Spring Boot project or code audit tool** to check for these antipatterns?