✅ 1. Design Smells (with Java examples)

🔹 Imperative Abstraction

Abstraction that exposes unnecessary details.

java

CopyEdit

```
abstract class PaymentProcessor {
    void logTransaction() {
        System.out.println("Logging...");
    } // Imperative detail inside abstraction
}
```

🔹 Multifaceted Abstraction

A class does too many unrelated things.

java

CopyEdit

```
class UserAccountManager {
    void registerUser() {}
    void login() {}
    void sendPromotionalEmail() {} // Unrelated responsibility
}
```

🔹 Unnecessary Abstraction

A class created for no real need.

java

CopyEdit

```
abstract class BaseLogger {
    abstract void log(String message);
}
```

```
class ConsoleLogger extends BaseLogger {
    void log(String message) {
        System.out.println(message);
    }
}
```

Could be a simple utility class, no need for abstraction.

🔹 **Unutilized Abstraction**

Abstract class/interface never extended or used.

java

CopyEdit

```
interface UnusedService {
    void serve();
}
```

🔹 **Deficient Encapsulation**

Fields that should be private are public.

java

CopyEdit

```
class Product {
    public String name; // Should be private
}
```

🔹 **Unexploited Encapsulation**

No methods operate on internal data.

java

CopyEdit

```
class Rectangle {
    private int width;
    private int height;
    // Getters/Setters only, no behavior
}
```

🔹 Broken Modularization

Unrelated concerns in one module/class.

java

CopyEdit

```
class ReportManager {
    void generatePDF() {}
    void sendEmail() {}
    void logReport() {}
}
```

🔹 Cyclic-Dependent Modularization

Two modules depend on each other.

java

CopyEdit

```
class A {
    B b;
}

class B {
    A a;
}
```

🔹 Insufficient Modularization

All logic in one huge class.

java

CopyEdit

```
class GodClass {
    void manageOrders() {}
    void processPayments() {}
    void updateInventory() {}
}
```

🔹 Hub-like Modularization

One class is excessively depended on.

java

CopyEdit

```
class Utility {
    // Used everywhere
}
```

🔹 Broken Hierarchy

Inheritance used improperly.

java

CopyEdit

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

class Ostrich extends Bird {
    void fly() {
        throw new UnsupportedOperationException();
    }
}
```

🔹 Cyclic Hierarchy

Inheritance loops (usually theoretical or conceptual) java

CopyEdit

```
// Impossible in Java directly, but conceptual
// A -> B -> C -> A (violation)
```

🔹 Deep Hierarchy

Too many levels of inheritance.

java

CopyEdit

```
class A {}
class B extends A {}
class C extends B {}
class D extends C {}
class E extends D {} // Too deep
```

🔹 Missing Hierarchy

No use of inheritance where beneficial.

java

CopyEdit

```
class Dog {
    void bark() {}
}

class Cat {
    void meow() {}
}
// Could share a superclass Animal
```

🔹 Multipath Hierarchy

A class inherits multiple paths from the same root.

java

CopyEdit

```java
interface A { void f(); }
interface B extends A {}
interface C extends A {}

class D implements B, C {
    public void f() {}
}
```

🔹 Rebellious Hierarchy

Subclasses override behavior inconsistently.

java

CopyEdit

```java
class Animal {
    void sound() {
        System.out.println("Some sound");
    }
}

class Cat extends Animal {
    void sound() {
        throw new RuntimeException(); // Rebellious
    }
}
```

🔹 Wide Hierarchy

Too many subclasses for one class.

java

CopyEdit

```java
class Shape {} // 15+ subclasses like Circle, Square, Star, Triangle, etc.
```

✅ 2. Implementation Smells (with Java examples)

🔹 Abstract Function Call From Constructor

Calling abstract method in constructor.

java

CopyEdit

```
abstract class AbstractClass {
    AbstractClass() {
        doSomething(); // Abstract method
    }

    abstract void doSomething();
}
```

🔹 Complex Conditional

java

CopyEdit

```
if ((age > 18 && !isStudent) || (income > 50000 && hasCar)) {
    // Complex
}
```

🔹 Complex Method

java

CopyEdit

```
void process() {
    for (...) {
        if (...) {
            // many levels of nesting
        }
    }
}
```

🔹 **Empty catch clause**

java

CopyEdit

```
try {
    // code
} catch (Exception e) {
    // empty
}
```

🔹 **Long Identifier**

java

CopyEdit

```
String thisIsAVeryLongVariableNameThatShouldBeShorter = "data";
```

🔹 **Long Method**

java

CopyEdit

```
void calculate() {
    // 100+ lines of code
}
```

🔹 **Long Parameter List**

java

CopyEdit

```
void createUser(String name, String email, String phone, String address, String dob) {}
```

🔹 **Long Statement**

java

CopyEdit

```
System.out.println("This is a very long string that continues
and continues and should probably be broken down.");
```

🔹 **Magic Number**

java

CopyEdit

```
if (score > 70) {} // What is 70?
```

🔹 **Missing default**

java

CopyEdit

```
switch (day) {
    case "MON": break;
    case "TUE": break;
    // missing default
}
```

✅ 3. Object-Oriented Metrics

Let’s use this class:

java

CopyEdit

```
class Sample {
    private int a, b;     // NOF = 2
    public int x;         // NOPF = 1

    public void m1() {}   // NOM = 2, NOPM = 1
    private void m2() {}
}
```

- **LOC (Lines of Code):** Count total lines inside class/method
- **CC (Cyclomatic Complexity):** Count decision points (if, for, while, switch)
- **PC (Parameter Count):** Number of parameters in method
- **NOF (Number of Fields):** All fields
- **NOPF (Number of Public Fields):** `x` only
- **NOM (Number of Methods):** 2
- **NOPM (Number of Public Methods):** 1
- **WMC (Weighted Methods per Class):** Sum of all method complexities
- **NC (Number of Children):** Count of direct subclasses
- **DIT (Depth of Inheritance Tree):** How far down the inheritance
- **LCOM (Lack of Cohesion in Methods):** How unrelated methods are
- **FANIN:** How many methods/classes call this class
- **FANOUT:** How many external classes this class calls

Here's a **complete explanation** of the **17 design smells**, **10 implementation smells**, and **object-oriented metrics** with:

- **Definitions**
- **Java code examples**

✅ 1. DESIGN SMELLS

These relate to **bad design choices** in class structures, modularization, and hierarchy.

Design Smell	Definition	Java Code Example
1. Imperative Abstraction	Abstraction that reveals unnecessary implementation logic instead of high-level behavior.	`java abstract class PaymentService { void log() { System.out.println("Logging..."); } }`
2. Multifaceted Abstraction	A class or interface handles unrelated responsibilities.	`java class UserManager { void login() {} void sendEmail() {} } // mix of user and email logic`
3. Unnecessary Abstraction	Abstract class or interface used without a valid need.	`java abstract class Logger { abstract void log(String msg); } class ConsoleLogger extends Logger { void log(String msg) { System.out.println(msg); } }`
4. Unutilized Abstraction	Abstraction never extended/implemented by any class.	`java interface Payment { void pay(); } // Never used anywhere`
5. Deficient Encapsulation	Exposing internal fields publicly.	`java class Person { public String name; }`
6. Unexploited Encapsulation	Class with only data, but no behavior.	`java class Rectangle { private int width; private int height; // Only getters/setters }`
7. Broken Modularization	A class contains code from different, unrelated modules.	`java class ReportManager { void generate() {} void sendEmail() {} }`

8. Cyclic-Dependent Modularization	Modules/classes depend on each other circularly.	`java class A { B b; } class B { A a; }`
9. Insufficient Modularization	One class tries to do everything ("God Object").	`java class SystemManager { void controlUI() {} void saveToDB() {} void sendSMS() {} }`
10. Hub-like Modularization	A class is overly depended upon by many others.	`java class Util { static void log() {} static void convert() {} }`
11. Broken Hierarchy	Subclass breaks behavior expectations of superclass.	`java class Bird { void fly() {} } class Ostrich extends Bird { void fly() { throw new UnsupportedOperationException(); } }`
12. Cyclic Hierarchy	Inheritance loops (conceptually)	Not directly possible in Java due to compiler constraints
13. Deep Hierarchy	Inheritance tree is too deep.	`java class A {} class B extends A {} class C extends B {} class D extends C {} // etc.`
14. Missing Hierarchy	Similar classes with no common parent.	`java class Dog {} class Cat {} // No Animal superclass`
15. Multipath Hierarchy	Multiple paths to the same superclass/interface.	`java interface A {} interface B extends A {} interface C extends A {} class D implements B, C {}`
16. Rebellious Hierarchy	Subclass overrides expected behavior incorrectly.	`java class Engine { void start() {} } class ElectricEngine extends Engine {`

		`void start() { throw new RuntimeException(); } }`
17. Wide Hierarchy	A class has too many subclasses.	`java class Shape {} class Circle extends Shape {} class Square extends Shape {} class Triangle extends Shape {} // many more`

✅ 2. IMPLEMENTATION SMELLS

These relate to **bad coding practices**, reducing readability, maintainability, or robustness.

Implementation Smell	Definition	Java Example
1. Abstract Function Call From Constructor	Calling abstract method from constructor can cause null or unexpected behavior.	`java abstract class A { A() { doSomething(); } abstract void doSomething(); }`
2. Complex Conditional	A condition that is too complicated to understand.	```java if ((user.isActive() && !user.isBanned())
3. Complex Method	A method with too many branches or logic layers.	`java void process() { if (...) { for (...) { if (...) { ... }}}}`
4. Empty catch clause	Swallowing exceptions silently.	`java try { ... } catch (Exception e) {}`

5. Long Identifier	Variables or method names that are excessively long.	`java String thisIsAnExtremelyLongNameThatIsHardToRead;`
6. Long Method	A method that has too many lines (e.g., >30 LOC).	`java void calculate() { // 100+ lines }`
7. Long Parameter List	Too many parameters in a method (>3).	`java void createUser(String name, String email, String phone, String dob, String address)`
8. Long Statement	A statement that is very long and hard to read.	`java System.out.println("This is a very long string with a lot of data and it goes on and on...");`
9. Magic Number	Using numbers directly in code without explanation.	`java if (salary > 30000) // what is 30000?`
10. Missing default	`switch` statement lacks a `default` case.	`java switch (type) { case 1: break; case 2: break; } // missing default`

✅ 3. OBJECT-ORIENTED METRICS (with simple class example)

Sample Class:

java
CopyEdit

```
class Vehicle {
    private String name;         // Field
    private int speed;           // Field
    public int wheels;           // Public Field

    public void drive() {        // Public Method
        if (speed > 0) {         // Decision point
            System.out.println("Driving...");
        }
    }

    private void stop() {}       // Private Method
}
```

Metric	Definition	Value (Example)
LOC	Lines of code in a class/method	10 lines
CC (Cyclomatic Complexity)	Count of decision points (`if`, `for`, `switch`, etc.) + 1	2
PC (Parameter Count)	Number of parameters in a method	0 (both methods)
NOF	Number of fields	3
NOPF	Number of public fields	1 (`wheels`)
NOM	Number of methods in class	2
NOPM	Number of public methods	1 (`drive`)
WMC	Sum of CC of all methods	2 (for `drive`) + 1 (for `stop`) = 3
NC	Number of direct subclasses	Depends on other classes
DIT	Inheritance depth from root	1 if `Vehicle extends Object`
LCOM	Lack of Cohesion in Methods	If methods use different fields → high LCOM

FANIN	Number of classes calling this class	2 (if called in `Main` and `Garage`)
FANOUT	Number of other classes this class uses	0

✅ 1. Rules to Identify Design Smells

Design Smell	Identification Rule
1. Imperative Abstraction	Abstract class or interface contains concrete methods with low-level logic (e.g., logging, printing, internal loops).
2. Multifaceted Abstraction	A class has methods handling **unrelated responsibilities**, often violating **Single Responsibility Principle**.
3. Unnecessary Abstraction	Abstraction (interface/abstract class) has only one implementation, or exists without clear benefit.
4. Unutilized Abstraction	Interface/abstract class not implemented/extended by any concrete class.
5. Deficient Encapsulation	Public fields (`public int x;`) or getters/setters that expose internal mutable state.
6. Unexploited Encapsulation	Class has only data (fields + getters/setters), no behavioral methods.
7. Broken Modularization	A class contains logic from **different business domains or modules** (e.g., UI + DB).
8. Cyclic-Dependent Modularization	Two or more classes/modules depend on each other directly or indirectly (cyclic imports or fields).

9. Insufficient Modularization	Class is large (>500 LOC), many responsibilities, hard to test; often a "God Class".
10. Hub-like Modularization	Class has high **fan-in** (used by many others), especially utility/helper classes.
11. Broken Hierarchy	Subclass breaks behavior contract (Liskov Substitution Principle violation).
12. Cyclic Hierarchy	Conceptually cyclic or overly tangled inheritance; not common in Java due to compiler error.
13. Deep Hierarchy	Class has >5 levels of inheritance (`DIT > 5`).
14. Missing Hierarchy	Multiple similar classes with duplicated code and **no common superclass or interface**.
15. Multipath Hierarchy	Class implements multiple interfaces that extend the same base (diamond problem-like).
16. Rebellious Hierarchy	Subclass redefines methods in a way that breaks expected behavior.
17. Wide Hierarchy	Superclass has many (e.g., >10) subclasses — indicates over-generalization.

✅ 2. Rules to Identify Implementation Smells

Implementation Smell	Identification Rule
1. Abstract Function Call From Constructor	Constructor calls an abstract method or a method that could be overridden.
2. Complex Conditional	`if`, `while`, `for` with many logical operators (`&&`, `
3. Complex Method	Method with many branches, loops, exception handling – **Cyclomatic Complexity** > **10**.

4. Empty Catch Clause	`catch (Exception e) {}` or similar, with no logging or rethrow.
5. Long Identifier	Variable or method name >30 characters or unreadable naming.
6. Long Method	Method has >30 lines of code.
7. Long Parameter List	Method has >4 parameters (especially primitive types or strings).
8. Long Statement	Line of code >120 characters; hard to read/understand.
9. Magic Number	Direct use of numbers without named constants (e.g., `if (salary > 10000)`).
10. Missing default	`switch` statement with no `default` case.

✅ 3. Rules to Compute Object-Oriented Metrics

Metric	Rule
LOC (Lines of Code)	Count total number of lines in method or class (excluding comments/blank lines).
CC (Cyclomatic Complexity)	`1 + number of decisions (if, for, while, case, catch, &&,
PC (Parameter Count)	Number of parameters in method declaration.
NOF (Number of Fields)	Count of all class fields (private, protected, public).
NOPF (Public Fields)	Count of fields declared as `public`.
NOM (Number of Methods)	Count of all methods in the class.
NOPM (Public Methods)	Count of `public` methods in the class.
WMC (Weighted Methods per Class)	Sum of cyclomatic complexity of all methods.

NC (Number of Children)	Count of subclasses that directly inherit from the class.
DIT (Depth of Inheritance Tree)	Distance from current class to root (`Object`).
LCOM (Lack of Cohesion in Methods)	Measures if methods share fields. **High LCOM = low cohesion.**
FANIN	Count of other classes/methods that use this class.
FANOUT	Count of classes used by this class (via method call, field, etc.).

Java Code Smell Rules

```
// Example Java class to demonstrate detection of code smells and object-oriented metrics


public class SmellExample {


  // === Metric: NOF (Number of Fields), NOPF (Number of Public Fields) ===

  public int publicField1; // NOPF +1

  private int privateField2; // NOF +1


  // === Smell: Long Parameter List ===

  public void methodWithTooManyParams(int a, int b, int c, int d, int e) { // PC = 5 (Smell)

    // === Smell: Magic Number ===

    if (a > 1000) { // Magic number = 1000

      // === Smell: Complex Conditional ===
        if (b > 5 && c < 10 || d == 20) {
            System.out.println("Complex condition"); // Increases CC
        }
    }
}

// === Smell: Long Method ===
public void longMethod() { // LOC > 30
    for (int i = 0; i < 10; i++) {
        System.out.println(i);
    }
    for (int i = 0; i < 10; i++) {
        System.out.println(i);
    }
    // Repeat similar blocks to increase LOC and CC
}

// === Smell: Abstract Function Call from Constructor ===
public SmellExample() {
    init(); // Should avoid calling overridable method from constructor
}
```

```
protected void init() {
    System.out.println("Init");
}

// === Smell: Empty Catch Block ===
public void catchBlockExample() {
    try {
        int a = 1 / 0;
    } catch (Exception e) {
        // empty catch
    }
}

// === Smell: Missing Default in Switch ===
public void switchExample(int x) {
    switch (x) {
        case 1: System.out.println("One"); break;
        case 2: System.out.println("Two"); break;
        // missing default
    }
}
```

```
    // === Metric: NOM (Number of Methods), NOPM (Number of Public Methods), WMC (sum of CC), CC (Cyclomatic Complexity) ===
    public void simpleMethod() {
        int x = 1; // CC = 1
    }

    private void privateMethod() {
        if (true) { // +1 CC
            System.out.println("True");
        }
    }
}

// === Metric: DIT (Depth of Inheritance Tree), NC (Number of Children) ===
class Base {}
class Child1 extends Base {} // DIT = 2
class Child2 extends Base {} // NC = 2 for Base

// === Design Smell: Unutilized Abstraction ===
abstract class AbstractUnused {} // No subclass

// === Design Smell: Multifaceted Abstraction ===
class UserManager {
    public void createUser() {}
    public void connectToDB() {} // Mixed concerns (user logic + DB)
}

// === Design Smell: Wide Hierarchy ===
class Animal {}
class Dog extends Animal {}
class Cat extends Animal {}
class Cow extends Animal {}
class Goat extends Animal {}
class Sheep extends Animal {}
class Lion extends Animal {}
class Tiger extends Animal {}
class Monkey extends Animal {}
class Elephant extends Animal {}
class Horse extends Animal {}
class Giraffe extends Animal {} // More than 10 subclasses => Wide Hierarchy

// === Design Smell: Unnecessary Abstraction ===
interface SingletonInterface {
    void doSomething();
}
class SingletonImpl implements SingletonInterface { // Only 1 implementation
    public void doSomething() {}
}


// === Design Smell: Broken Hierarchy ===
class Bird {
    public void fly() {}
}
class Ostrich extends Bird { // Ostrich can't fly
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Ostrich can't fly");
    }
}
```

I've implemented a comprehensive Java example that demonstrates how to detect each code smell and object-oriented metric using clear, labeled code blocks. This includes:

- **Design Smells** like Unnecessary Abstraction, Wide Hierarchy, Broken Hierarchy, etc.
- **Implementation Smells** such as Long Methods, Magic Numbers, and Empty Catch Clauses.

- **OO Metrics** like LOC, CC, DIT, NOM, WMC, etc.