

Code Smells Catalog

A code smell is not a bug — the code compiles, runs, and produces correct output. It's a signal that the underlying design is likely to cause trouble later: harder to change safely, easier to break accidentally, harder for someone else (including future you) to read.

The smells listed are grouped according to the categories used by codesmells.org and are tagged with one of three tiers: the ★ **core** smells were covered in lecture and are the most common smells to watch for, the † **extended** smells include additional naming, commenting, and readability smells not covered in class, and the unmarked **auxiliary** smells get into deeper structural problems. For even more smells, check out codesmells.org.

1 Lexical Abusers

Lexical abusers are smells at the surface level of the code — code that hides or misleads about what it does.

1.1 Mysterious Name ★

Description: A name (of a class, method, or variable) that doesn't reveal what it holds or does — a reader has to look at the implementation to figure out what it is for.

Example:

```
public boolean chk ( Card c, TableauColumn col ) { ... }
```

1.2 Magic Number ★

Description: An unexplained literal number embedded directly in code, with nothing telling the reader what it represents.

Example:

```
if ( card.getRank().getValue() == 13 ) { ... }
```

1.3 Fallacious Comment †

Description: A comment that no longer matches the code around it.

Example:

```
// applies a flat 10% loyalty discount
public double applyDiscount ( Order order ) {
    return order.getTotal() * discountTier(order);
}
```

1.4 Fallacious Method Name †

Description: A name that actively misleads about what a method does.

Example:

```
// 'getTotal' sounds like a getter, but it also updates state
public double getTotal ( Order order ) {
    order.setTotal(recalculate(order)); // hidden side effect
    return order.getTotal();
}
```

2 Bloaters

Bloaters are code smells that often result in methods, classes, or code blocks growing so large that they become unwieldy. This decreases readability and makes code harder to maintain or modify.

2.1 Long Method ★

Description: A method you have to scroll through — it does too many distinct things to describe in one sentence.

Example: A single `processOrder()` method that validates the order, calculates the total, applies a discount, charges payment, and sends a confirmation email — five responsibilities in one method.

2.2 Large Class ★

Description: A class with a list of methods you have to scroll through — it has too many responsibilities to describe in one sentence.

Example: An `Order` class that validates itself, saves itself to a database, sends notification emails, and generates a PDF invoice is really four responsibilities living in one class.

2.3 Long Parameter List ★

Description: A method or constructor requiring many parameters. (“Many” can be as few as “more than two”.)

Example:

```
public Card ( int rankValue, String suitName, boolean faceUp, int x, int y )
```

2.4 Data Clumps ★

Description: The same small group of variables shows up together, repeatedly.

Example:

```
public void printLabel ( String street, String city, String zip ) { ... }  
public void validateAddress ( String street, String city, String zip ) { ... }
```

2.5 Primitive Obsession

Description: Using a plain primitive such as `int` or `String` where a small, purpose-built type would be clearer and safer.

Example:

```
public void placeCard ( int rankValue, int suitValue ) { ... }
```

3 Dispensables

Dispensables are code elements whose removal would make the codebase cleaner and easier to read without any meaningful impact on overall functionality.

3.1 Duplicate Code ★

Description: The same logic written more than once.

Example: Both `Order.calculateDiscount()` and a method in `Invoice` independently compute the same discount calculation.

3.2 Dead Code ★

Description: Code no longer called from anywhere, left behind after changes.

Example: A private method `legacyCalculateTax()` left in `Order` after tax calculation was rewritten to use a separate `TaxTable` class — nothing calls it anymore, but a future maintainer still has to read past it and might mistake it for something still in use.

3.3 Comments as Deodorant ★

Description: A comment explaining what confusing code does where a better name or structure would make the comment unnecessary.

Example:

```
// check if the customer qualifies for free shipping
if (o.getTotal() > 50 && o.getCustomer().getYears() > 1
    && !o.getItems().isEmpty()) { ... }
```

3.4 Data Class

Description: A class with only instance variables and getters/setters but no real behavior, which can mean that logic that should be in this class is elsewhere instead.

Example: An `Address` class exposing only `getStreet()/setStreet()`, `getCity()/setCity()`, and `getZip()/setZip()`, with no `formatForLabel()` of its own. Every class that needs a formatted shipping label has to assemble it by pulling the fields out itself, instead of asking `Address` to do it.

3.5 Lazy Class

Description: A class or method that does so little it barely justifies existing on its own.

Example: A `PriceFormatter` class whose entire content is one method: `format(double price)` { return "\$" + price; }.

3.6 Speculative Generality

Description: Flexibility built into the code for a need that doesn't actually exist yet ("we might need this someday") — an extracted abstraction, an unused parameter, a configuration option with only one real setting, or the like.

Example: A `PaymentGateway` interface with `CreditCardGateway` as its only implementation, created on the theory that PayPal or crypto support might be added later — but the store has never accepted anything but credit cards and no second implementation has ever been written.

4 Obfuscators

Obfuscators are code smells that make code harder to read at a glance.

4.1 Complicated Boolean Expression ★

Description: A conditional with double negatives or long chains of `&&/||/!` that's hard to parse at a glance.

Example:

```
if (!(order.isCancelled() || !order.isPaid())
    && order.getTotal() > 0) { ... }
```

4.2 Vertical Separation †

Description: Related pieces of code placed far apart, forcing a reader to scroll back and forth to reconstruct why something is there.

Example: In `processOrder()`, a `shippingDiscount` variable is declared near the top of the method but isn't used until well past a block of unrelated payment-processing logic later on.

4.3 Obscured Intent †

Description: A catch-all for code that is difficult to figure out. It is often the result of other smells — any of the Lexical Abusers, Vertical Separation, Complicated Boolean Expression, Clever Code, Imperative Loops, and so on — and is what Comments as Deodorant is an attempt to hide.

Example:

```
total = total * 0.9 - 5 > 0 ? total * 0.9 - 5 : 0;
```

Nothing here names what's happening — a 10% discount, then a flat \$5 coupon, floored at zero. The intent has to be reverse-engineered from the arithmetic.

4.4 Clever Code †

Description: Code that works but uses an obscure, bespoke technique where a well-known, conventional one would do the same job more readably.

Example:

```
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

Using XOR to swap the values of `a` and `b` instead of the usual method using a temporary variable:

```
int temp = a;
a = b;
b = temp;
```

5 Couplers

Couplers are code smells that violate the core design principle of loose coupling by creating excessive dependencies between different classes.

5.1 Feature Envy ★

Description: A method that accesses another class's data more than its own.

Example: A method in `OrderProcessor` repeatedly calls `customer.getAddress().getStreet()` and `customer.getAddress().getCity()` to build a shipping label.

5.2 Insider Trading ★

Description: A class knows too much about the internal details of another class — through direct access to private members, a getter returning a mutable reference to internal state, or a subclass depending on its superclass's implementation details.

Example: An `OrderProcessor` method gets `Cart`'s list of items and directly manipulates it:

```
List<Item> items = cart.getItems();
for (Item item : items) {
    if (!inventory.isInStock(item)) {
        items.remove(item);
    }
}
```

5.3 Message Chains

Description: A long sequence of calls drilling through several objects.

Example: `order.getCustomer().getAddress().getCity()`

6 Change Preventers

Change preventers hinder the ability to modify, add to, or otherwise further develop software. A typical instance forces a series of edits in multiple places just to implement a single, simple modification.

6.1 Shotgun Surgery ★

Description: A single conceptual change requires editing many different classes because responsibility for that one concept is scattered across all of them.

Example: Adding a new payment method requires editing `Order`, `Invoice`, and `ReceiptPrinter` — responsibility for “how a payment is represented and processed” should live in one place instead of three.

6.2 Divergent Change

Description: The mirror image of Shotgun Surgery: one class has to change for many unrelated reasons.

Example: `Order` has to be edited when the discount rules change, and again when the receipt-printing format changes, and again when the database schema changes — three unrelated reasons to change.

6.3 Callback Hell

Description: Code nested many levels deep across chained callbacks, obscuring what actually triggers what. (Can be common in heavily event-driven code.)

Example:

```
orderService.submit(order, result ->
    paymentService.charge(result, chargeResult ->
        notifier.send(chargeResult, sendResult -> {
            ...
        })))
```

The actual sequence — submit, then charge, then notify — is buried inside three levels of nested callbacks instead of reading top to bottom.

7 Object-Oriented Abusers

Object-oriented abusers are code smells that fail to fully or correctly apply object-oriented design principles.

7.1 Conditional Complexity ★

Description: Branching on an object's type or a type code, especially when the same branching logic is duplicated in more than one place — a sign polymorphism should be doing this work instead.

Example:

```
switch (paymentType) {
    case CREDIT_CARD: return chargeCard(order);
    case PAYPAL: return chargePayPal(order);
}
```

7.2 Alternative Classes with Different Interfaces

Description: Two classes doing essentially the same job but with differently-named methods, preventing the classes from being used interchangeably.

Example: `CreditCardGateway.charge(double amount)` and `PayPalGateway.makePayment(double amount)` both charge the customer, but `OrderProcessor` can't treat them the same way without an if statement because the names are different.

7.3 Refused Bequest

Description: A subclass that doesn't want or use much of what it inherits, often overriding inherited methods to do nothing or throw an exception.

Example:

```
public class GiftOrder extends Order {
    @Override
    public void applyDiscount(double percent) {
        throw new UnsupportedOperationException("gift orders can't be discounted");
    }
}
```

7.4 Temporary Field

Description: A field that's only meaningful in certain circumstances, sitting unused or irrelevant the rest of the time.

Example: An `Order` field `refundReason` that's only ever set (or meaningful) once an order has been refunded — for every other order it is just `null`.

8 Data Dealers

8.1 Middle Man

Description: A class that does almost nothing except delegate every call to another class.

Example: An `OrderFacade` whose every method immediately forwards to `OrderProcessor`:

```
public void charge(Order o) { processor.charge(o); }
public void refund(Order o) { processor.refund(o); }
// ...repeated for every OrderProcessor method, unchanged
```

8.2 Tramp Data

Description: Data passed through a chain of methods that never actually use it, just relay it further along.

Example: A `zip` parameter threaded through `processOrder(Order order, String zip) → calculateShipping(order, zip) → applyRegionalTax(order, zip)`, where only the last method actually uses it — the first two just relay it along.

8.3 Global Data

Description: Data modifiable from anywhere in the codebase.

Example: A `public static double taxRate` on a shared `TaxConfig` class.

9 Functional Abusers

9.1 Mutable Data

Description: State that changes unexpectedly, causing side effects that are hard to trace.

Example: A `Cart` object passed to both `OrderProcessor` and `DiscountEngine`. `DiscountEngine` removes an item from the cart as a side effect of computing a discount, so by the time `OrderProcessor` reads the cart, its contents have silently changed underneath it.

9.2 Side Effects

Description: A method that does more than its name promises.

Example: A method called `getTopCard()` that also secretly flips the card face-up — a caller who only wants to look at the top card gets an unexpected mutation along with it.

10 Refactorings

Smell	Refactoring
Mysterious Name ★	Rename
Magic Number ★	Replace Magic Number with Named Constant
Fallacious Comment †	Update or remove the comment; consider Extract Method / Rename instead
Fallacious Method Name †	Rename
Long Method ★	Extract Method
Large Class ★	Extract Class
Long Parameter List ★	Introduce Parameter Object
Data Clumps ★	Extract Class
Primitive Obsession	Replace Primitive with Object
Duplicate Code ★	Extract the shared logic into one place; have all call sites use it
Dead Code ★	Delete it
Comments as Deodorant ★	Extract Method / Rename, so the code no longer needs the comment
Data Class	Move behavior that uses this data onto the class itself
Lazy Class	Inline Class
Speculative Generality	Collapse Hierarchy / Inline Class / Remove Parameter
Complicated Boolean Expression ★	Decompose Conditional
Vertical Separation †	Move declarations closer to their use, or Extract Method
Obscured Intent †	Rename; Extract Method; Replace Magic Number with Named Constant
Clever Code †	Substitute Algorithm — replace with the well-known, conventional approach
Feature Envy ★	Move Method
Insider Trading ★	Tighten the interface — expose only what's needed through public methods
Message Chains	Hide Delegate
Shotgun Surgery ★	Move Method / Move Field, then consolidate related logic into one place
Divergent Change	Extract Class — split by reason-to-change
Callback Hell	Restructure control flow (e.g. named handler methods, or an event/state-based structure once covered)
Conditional Complexity ★	Replace Conditional with Polymorphism
Alternative Classes with Different Interfaces	Rename Method to unify naming, or extract a common interface
Refused Bequest	Replace Inheritance with Delegation
Temporary Field	Extract Class for the field and its associated logic
Middle Man	Remove Middle Man
Tramp Data	Restructure the call chain so data reaches its destination directly
Global Data	Encapsulate Variable — control access through methods

Smell	Refactoring
Mutable Data	Encapsulate Variable; prefer immutability where reasonable
Side Effects	Separate Query from Modifier — split into a pure query and a separate action