Code Smells

A **code smell** is a feature of the code that suggests something is wrong. They were introduced in the refactoring[1] book (chapter 3), and also extended and discussed in the clean code[2] book, chapter 17.

Typically we would remedy a code smell through a series of refactoring steps.

Comment smells

- Obsolete comments or comments that hold information better maintained by other means like a version control system. For example information about when changes were made, or the version number.
  Remedy: Delete them! Give Git a chance!
- Redundant comments that do little more than restate what the code says. Not useful, and might not change when the code changes, resulting to misleading comments.
  Remedy: Delete them!
- Commented-out code. By now probably totally out of sync with the system.
  Remedy: Terminate with extreme prejudice!

Naming smells

- Non-descriptive names.
- Names at wrong levels of abstractions.
- Ambiguous names.
- Type encoding in names.
- Names that hide side-effects.
- Function names that don't correctly say what the functions do.
  Remedy for all: Rename them, until you find a good name.

Function smells

- Too many arguments on a function.
  Remedy: If some arguments often go together, extract them into a parameter object.
- Flag arguments (booleans). These suggest a function is doing too many things.
  Remedy: Possibly create separate functions. Maybe convert the booleans to some kind of enum if that is more appropriate?
- Wrong Levels of Abstraction. The code in each method should be at the same level of abstraction.
  Remedy: Extract method, giving suitable names to the new methods.
- Feature Envy. A function that uses more methods from another class than its own possibly belongs to that other class.
  Remedy: Move the function to the other class.
- Inappropriate Static functions. When creating static functions, make sure they would not more naturally be housed in the class of one of their parameters.
  Remedy: Move the function to the other class.

[1] https://learning.oreilly.com/library/view/refactoring-improving-the/9780134757681/ch03.xhtml
[2] https://learning.oreilly.com/library/view/clean-code/9780136083238/chapter17.html

- Long function. Probably means it is doing way too much.
  Remedy: Extract till you drop!

Behavioral smells

- Obvious behavior not implemented: Functions should implement reasonably expected behavior.
  Remedy: Add tests for this behavior and change your function to match.
- Incorrect Behavior at the Boundaries.
  Remedy: Add tests for this behavior and change your function to match.
- Code Duplication should be avoided (Don't Repeat Yourself)
  Remedy: Extract Method.
- Parts of the code that have too much knowledge.
  Remedy: Find ways to reduce the coupling between classes!
- Dead code. It cannot be reached. IntelliJ will actually complain about that.
  Remedy: Figure out what's wrong!
- Artificial Coupling. Don't make parts of your application know about each other if they don't have to.
  Remedy: Remove the unnecessary dependencies. Might require moving some functions around.
- Misplaced Responsibility. Code should go where it is most expected to be found, not where it is most convenient for the programmer.
  Remedy: Move functions around, or extract methods to create new wrappers. And keep thinking of your user.
- Hidden temporal couplings. A sequence of statements needs to happen in a particular order, but nothing in the code forces this to happen.
  Remedy: Make each statement depend on some output from the previous statement.
- Transitive Navigation / Train wrecks. A sequence of getting calls, like: `o.getA().getB().getC().doSomething()`. The object `o` needs to know too much about the system that way.
  Remedy: Hide the chain behind a method that expresses more directly what you are doing, and/or make some of the classes in the middle do the same.
- Divergent Change. When a module changes for many different reasons. It is a sign that this module is doing too many disparate things.
  Remedy: Find a subset of the methods that have common functionality, and extract them as a delegate or a superclass.
- Shotgun Surgery. When we have to change many modules to effect a single behavioral change. This means that functionality that changes for the same reasons has been needlessly spread across many classes.
  Remedy: Move the relevant functionality around to where it should be.
- Data Clumps. Data that tends to stick together should be in a class.
  Remedy: Extract a new class out of those common elements.
- Primitive Obsession. The overuse of primitive types.
  Remedy: Create a new class to host the functionality related to those primitive types.

- Speculative Generality. Unneeded abstractions that make the code harder to read.

  Remedy: Eliminate with Extreme Prejudice! YAGNI!

Form

- Vertical Separation. Variables and functions should be close to their use.

  Remedy: Move them closer to each other.

- Inconsistency. Similar tasks are performed differently, or similar variables are named differently.

  Remedy: Use Rename, Extract Method and other refactorings to do things in similar ways when possible.

- Obscured Intent. Various aspects of the code make it hard to discern the code's intent. Lack of Explanatory Variables.

  Remedy: Extract Methods/Variables to give names to things, and renaming to give them better names.

- Magic constants. Used as is in the text, without clear meaning.

  Remedy: Extract Constant. Choose a good name for it.

- Complex conditional tests.

  Remedy: Extract Methods for them to make them more readable. And find ways to simplify the code.

- Temporary field. A field that is not always set/used.

  Remedy: It may belong to a different class.

- Null checks all over the code.

  Remedy: Separate your public methods from your internal methods, and only allow null inputs from outside calls.

  Consider creating a "Null Object" class, to represent meaningful functionality for a "null object" and pass it around.