Code Smells

Code smells are the many warning signs that may present themselves within your code as indicators that you should consider refactoring. Developing a code nose is necessary to spot indicators that may be preventing you from having clean, concise, and easily readable code.

Code smells within classes:

Comments

There's a fine line between comments that illuminate and comments that obscure. Are the comments necessary? Do they explain *why* and not *what*? Can you refactor your code so that comments aren't required? Just remember that you're writing comments for people and not computers.

Long Method

All other things being equal, a shorter method is easier to read, easier to understand, and easier to troubleshoot. Refactor long methods into smaller methods if you can.

Long Parameter List

The more parameters a method has, the more complex it is. Limit the number of parameters you need in a given method or use an object to combine the parameters.

Duplicated Code

Duplicated code is the bane of software development. Stamp out duplication whenever possible. You should always be on the lookout for more subtle cases of near-duplication too. *Don't Repeat Yourself* is commonly referred to as the DRY Principle.

Conditional Complexity

Watch out for large conditional logic blocks, particularly blocks that tend to grow larger or change significantly over time. Consider alternative object-oriented approaches such as decorator, strategy, or state.

Combinatorial Explosion

You have lots of code that does almost the same thing, but with tiny variations in data or behavior. This can be difficult to refactor, and you could perhaps try using generics or an interpreter.

Large Class

Large classes, like long methods, are difficult to read, understand, and troubleshoot. Does the class contain too many responsibilities? Can the large class be restructured or broken into smaller classes?

Type Embedded In Name

Avoid placing types in method names; it's not only redundant, but it forces you to change the name if the type changes.

Uncommunicative Name

Does the name of the method succinctly describe what that method does? Could you read the method's name to another developer and have them explain to you what it does? If not, rename it or rewrite it.

Inconsistent Names

Pick a set of standard terminology and stick to it throughout your methods. For example, if you have Open(), you should probably have Close().

Dead Code

Ruthlessly delete code that isn't being used. That's the benefit of using source control systems.

Speculative Generality

Write code to solve today's problems, and worry about tomorrow's problems when they actually materialize. *You (Probably) Aren't Gonna Need It* is known as the YAGNI Principle.

Oddball Solution

There should only be one way of solving the same problem in your code. If you find an oddball solution, it could be a case of poorly duplicated code or it could be an argument for the adapter model, if you really need multiple solutions to the same problem.

Temporary Field

Watch out for objects that contain a lot of optional or unnecessary fields. If you're passing an object as a parameter to a method, make sure that you're using all of it and not singling out specific fields.

Code smells between classes:

Alternative Classes With Different Interfaces

If two classes are similar on the inside, but different on the outside, perhaps they can be modified to share a common interface.

Primitive Obsession

Don't use a bunch of primitive data type variables as a substitute for a class. If your data type is sufficiently complex, write a class to represent it.

Data Class

Avoid classes that passively store data. Classes should contain data and methods to operate on that data.

Data Clumps

If you always see the same data hanging around together, maybe it belongs together. Consider rolling the related data up into a larger class.

Refused Bequest

If you inherit from a class, but never use any of the inherited functionality, should you really be using inheritance?

Inappropriate Intimacy

Watch out for classes that spend too much time together, or classes that interface in inappropriate ways. Classes should know as little as possible about each other.

Indecent Exposure

Beware of classes that unnecessarily expose their internals. Aggressively refactor classes to minimize their public surface. You should have a compelling reason for every item you make public. If you don't, then hide it.

Feature Envy

Methods that make extensive use of another class may belong in another class. Consider moving this method to the class it is so envious of.

Lazy Class

Classes should pull their weight. Every additional class increases the complexity of a project. If you have a class that isn't doing enough to pay for itself, can it be collapsed or combined into another class?

Message Chains

Watch out for long sequences of method calls or temporary variables to get routine data. Intermediaries are dependencies in disguise.

Middleman

If a class is delegating all its work, why does it exist? Cut out the middleman. Beware classes that are merely wrappers over other classes or existing functionality in the framework.

Divergent Change

If, over time, you make changes to a class that touch completely different parts of the class, it may contain too much unrelated functionality. Consider isolating the parts that changed in another class.

Shotgun Surgery

If a change in one class requires cascading changes in several related classes, consider refactoring so that the changes are limited to a single class.

Parallel Inheritance Hierarchies

Every time you make a subclass of one class, you must also make a subclass of another. Consider folding the hierarchy into a single class.

Incomplete Library Class

We need a method that's missing from the library, but we're unwilling or unable to change the library to include the method. The method ends up tacked on to some other class. If you can't modify the library, consider isolating the method.

Solution Sprawl

If it takes five classes to do anything useful, you might have solution sprawl. Consider simplifying and consolidating your design.

Author: Trevor Morin