Nicole P. Satiembre **April 9, 2024**

Cebu Technological University-Main Campus

Clean Code

Chapter 17 - Smells and Heuristics

Here is a list of things that smells like bad code:

Comments

C1: Inappropriate Information. In general, meta-data such as authors, last modified-date, SPR number, and so on should not appear in comments. Comments should be reserved for technical notes about the code and design.

C2: Obsolete Comment. A comment that has gotten old, irrelevant, and incorrect is obsolete. Comments get old quickly. It is best not to write a comment that will become obsolete.

C3: Redundant Comment. A comment is redundant if it describes something that adequately describes itself.

C4: Poorly Written Comment. If you are going to write a comment, take the time to make sure it is the best comment you can write.

C5: Commented-Out Code. That code sits there and rots, getting less and less relevant with every passing day.

Environment

E1: Build Requires More than One Step. Building a project should be a single trivial operation. You should not have to check many little pieces out from source code control. You should be able to check out the system with one simple command and then issue one other simple command to build it.

E2: Tests Require More than One Step. You should be able to run all the unit tests with just one command. In the best case, you can run all the tests by clicking on one button in your IDE.

Functions

F1: Too Many Arguments. Functions should have a small number of arguments. No argument is best, max would be three.

F2: Output Arguments. Output arguments are counterintuitive. Readers expect arguments to be inputs, not outputs.

F3: Flag Arguments. Boolean arguments loudly declare that the function does more than one thing.

F4: Dead Function. Methods that are never called should be discarded. Keeping dead code around is wasteful.

General

G1: Multiple Languages in One Source File. The ideal is for a source file to contain one, and only one, language. Realistically, we will probably have to use more than one. But we should take pains to minimize both the number and extent of extra languages in our source files

G2: Obvious Behavior is Unimplemented. When an obvious behavior is not implemented, readers and users of the code can no longer depend on their intuition about function names. They lose their trust in the original author and must fall back on reading the details of the code.

G3: Incorrect Behavior at the Boundaries. Every boundary condition, every corner case, every quirk and exception represents something that can confound an elegant and intuitive algorithm. Don't rely on your intuition. Look for every boundary condition and write a test for it.

G4: Overridden Safeties. Chernobyl melted down because the plant manager overrode each of the safety mechanisms one by one. The safeties were making it inconvenient to run an experiment. The result was that the experiment did not get run, and the world saw it's first major civilian nuclear catastrophe.

G5: Duplication. One important rule we must alway follow is the DRY principle. Do Not Repeat Yourself. Every time you see duplication in the code, it represents a missed opportunity for abstraction.

G6: Code at Wrong Level of Abstraction. Good software design requires that we separate concepts at different levels and place them in different containers. We don't want lower and higher-level concepts mixed together.

G7: Base Classes Depending on their Derivatives. Base classes should know nothing about their derivatives. Deploying derivatives and bases in different jar files and making sure the base jar files know nothing about the contents of the derivative jar files allow us to deploy our systems in discrete and independent components.

G8: Too Much Information. Good software developers learn to limit what they expose at the interfaces of their classes and modules. The fewer methods a class has, the better. The fewer variables a function knows about, the better. The fewer instance variables a class has, the better.

G9: Dead Code. Dead code is code that isn't executed. Dead code is not completely updated when designs change. It still compiles, but it does not follow newer conventions or rules.

G10: Vertical Separation. Variables and functions should be defined close to where they are used. Local variables should be declared just above their first usage and should have a small vertical scope.

G11: Inconsistency. Be careful with the conventions you choose, and once chosen, be careful to continue to follow them.

G12: Clutter. Variables that aren't used, functions that are never called, comments that add no information, and so forth. All these things are clutter and should be removed. Keep your source files clean, well-organized, and free of clutter.

G13: Artificial Coupling. Things that don't depend upon each other should not be artificially coupled. In general, an artificial coupling is a coupling between two modules that serves no direct purpose.

G14: Feature Entry. The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes.

G15: Selector Arguments. Selector arguments are just a lazy way to avoid splitting a large function into several smaller functions.

G16: Obscured Intent. We want code to be as expressive as possible. Run-on expressions, Hungarian notation, and magic numbers all obscure the author's intent.

G17: Misplaced Responsibility. One of the most important decisions a software developer can make is where to put code.

G18. Inappropriate Static. In general, you should prefer nonstatic methods to static methods. When in doubt, make the function nonstatic. If you really want a function to be static, make sure that there is no chance that you'll want it to behave polymorphically.

G19. Use Explanatory Variables. More explanatory variables are generally better than fewer. It is remarkable how an opaque module can suddenly become transparent simply by breaking the calculations up into well-named intermediate values.

G20. Function Names Should Say What They Do. If you have to look at the implementation (or documentation) of the function to know what it does, then you should work to find a better name or rearrange the functionality so that it can be placed in functions with better names.

G21. Understand the Algorithm. Before you consider yourself to be done with a function, make sure you understand how it works. Often the best way to gain this knowledge and understanding is to refactor the function into something that is so clean and expressive that it is obvious how it works.

G22. Make Logical Dependencies Physical. If one module depends upon another, that dependency should be physical, not just logical. The dependent module should not make assumptions about the module it depends upon.

G23. Prefer Polymorphism to If/Else or Switch/Case. There may be no more than one switch statement for a given type of selection. The cases in that switch statement must create polymorphic objects that take the place of other such switch statements in the rest of the system.

G24: Follow Standard Conventions. Every team should follow a coding standard based on common industry norms. This coding standard should specify things like where to declare instance variables; how to name classes, methods, and variables; where to put braces; and so on.

G25: Replace Magic Numbers with Named Constants. In general it is a bad idea to have raw numbers in your code. You should hide them behind well-named constants.

G26: Be Precise. Ambiguities and imprecision in code are either a result of disagreements or laziness. In either case they should be eliminated.

G27: Structure over Convention. Enforce design decisions with structure over convention.

G28: Encapsulate Conditionals. Extract functions that explain the intent of the conditional.

G29: Avoid Negative Conditionals. Negatives are just a bit harder to understand than positives. So, when possible, conditionals should be expressed as positives.

G30: Functions Should do One Thing. It is often tempting to create functions that have multiple sections that perform a series of operations. Functions of this kind do more than one thing, and should be converted into many smaller functions, each of which does one thing.

G31: Hidden Temporal Couplings. Temporal couplings are often necessary, but you should not hide the coupling. Structure the arguments of your functions such that the order in which they should be called is obvious.

G32: Don’t be Arbitrary. Have a reason for the way you structure your code, and make sure that reason is communicated by the structure of the code.

G33: Encapsulate Boundary Conditions. Boundary conditions are hard to keep track of. Put the processing for them in one place.

G34: Functions Should Descend Only One Level of Abstraction. The statements within a function should all be written at the same level of abstraction, which should be one level below the operation described by the name of the function.

G35: Keep Configurable Data at High Levels. If you have a constant such as a default or configuration value that is known and expected at a high level of abstraction, do not bury it in a low-level function. Expose it as an argument to that low-level function called from the high-level function.

G36: Avoid Transitive Navigation. In general we don't want a single module to know much about its collaborators. Make sure that modules know only about their immediate collaborators and do not know the navigation map of the whole system.

Java

J1: Avoid Long Import Lists by Using Wildcards. Making sure that modules know only about their immediate collaborators and do not know the navigation map of the whole System. Specific imports are hard dependencies, whereas wildcard imports are not.

J2: Don't Inherit Constants. Don't use inheritance as a way to cheat the scoping rules of the language. Use a static import instead.

J3: Constants versus Enums. The meaning of ints can get lost. The meaning of enums cannot, because they belong to an enumeration that is named.

Names

N1: Choose Descriptive Names. Don't be too quick to choose a name. Make sure the name is descriptive.

N2: Choose Names at the Appropriate Level of Abstraction. Don't pick names that communicate implementation; choose names that reflect the level of abstraction of the class or function you are working in.

N3: Use Standard Nomenclature Where Possible. Names are easier to understand if they are based on existing convention or usage. Patterns are just one kind of standard.

N4: Unambiguous Names. Choose names that make the workings of a function or variable unambiguous.

N5: Use Long Names for Long Scopes. The length of a name should be related to the length of the scope. You can use very short variable names for tiny scopes, but for big scopes you should use longer names.

N6: Avoid Encodings. Names should not be encoded with type or scope information. Prefixes such as m_ or f are useless in today's environments.

N7: Names Should Describe Side-Effects. Names should describe everything that a function, variable, or class is or does. Don't hide side effects with a name.

Tests

T1: Insufficient Tests. A test suite should test everything that could possibly break. The tests are insufficient so long as there are conditions that have not been explored by the tests or calculations that have not been validated.

T2: Use a Coverage Tool! Coverage tools report gaps in your testing strategy. They make it easy to find modules, classes, and functions that are insufficiently tested.

T3: Don't Skip Trivial Tests. They are easy to write and their documentary value is higher than the cost to produce them.

T4: An Ignored Test is a Question about an Ambiguity. Sometimes we are uncertain about a behavioral detail because the requirements are unclear.

T5: Test Boundary Conditions. Take special care to test boundary conditions. We often get the middle of an algorithm right but misjudge the boundaries.

T6: Exhaustively Test Near Bugs. Bugs tend to congregate. When you find a bug in a function, it is wise to do an exhaustive test of that function. You'll probably find that the bug was not alone.

T7: Patterns of Failure are Revealing. Sometimes you can diagnose a problem by finding patterns in the way the test cases fail. This is another argument for making the test cases as complete as possible. Complete test cases ordered reasonably, expose patterns.

T8: Test Coverage Patterns Can Be Revealing. Looking at the code that is or is not executed by the passing tests gives clues to why the failing tests fail.

T9: Tests Should Be Fast. A slow test is a test that won't get run. When things get tight, it's the slow tests that will be dropped from the suite. So do what you must to keep your tests fast.