# Clean Coding

A collection of clean coding principles we should abide to, to the best of our abilities. Writing code is hard, writing clean code is harder, but it’s no excuse to write illegible code that cannot be maintained. Code that cannot be maintained cannot be scaled, and hence we will fall prey either to our successes or to mounting irremovable bugs.

It is therefore everyone’s duty to ensure our code stays clean, and we should call each other out in the face of dirty code or bad code smells. If you spot dirty code and stay silent about it for whatever reason, then you’re as much a culprit as the transgressor. There should only be two types of code in our codebases, clean code and code that hasn’t yet been spotted to be dirty. There shouldn’t be dirty code that’s left on its own. There is no reason to have dirty code.

What is Clean Code? I take reference from Clean Code by Robert C. Martin, and start with programmatic practices I think will result in clean code first, before moving on to Dart specific style guides and best practices. This isn’t meant to replace reading the source materials, but instead to provide a reference to the code of coding conduct we promise to abide to. Just as with spotting dirty code, it is everyone’s responsibility to update the conventions listed here, and ensure that this document does not develop ‘smells of irrelevancy’ as time goes by.

## Names

**Choose Descriptive Names**

Make sure all chosen names are descriptive and give intuition to readers with enough context. Remember meanings tend to drift as software evolves, so frequently reevaluate the appropriateness of names chosen. Names in software are mostly what makes software readable.

**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 that you are working in. Often a variable is named at too low a level, and opportunities should be taken to change those names when they are found. Making code readable requires dedication to continuous improvement.

**Use Standard Nomenclature Where Possible**

Names are easier to understand if they are based on existing convention or usage. Patterns are one kind of standard. Teams will often invent their own standard system of names for a particular project. The more names are overloaded with special meanings that are relevant to the project, the easier it is for readers with context to intuit meaning and function.

**Use Long Names for Long Scopes**

Length of a name should reflect or be related to the length of the scope. For bigger scopes, longer names should be used. Variables and functions with short names lose their meaning over long distances. So the longer the scope of the name, the longer and more precise the name should be.

**Avoid Encodings**

Names should not be encoded with type or scope information. Today’s environments provide enough information without having to mangle names.

**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. A function that does more than one simple action should not be lazily described as a single verb.

## Comments

**Inappropriate Information**

It is inappropriate for a comment to hold information better held in a different kind of system such as VCS (Git), issue tracking (Jira), scrum (Monday) or other record keeping systems. Comments should be reserved about technical notes about code and design. If there is an external platform better served to hold that comment, and we’re not using it, we should be.

**Obsolete Comments**

A comment that has gotten old, irrelevant or incorrect has become obsolete. Delete it or update it immediately. Obsolete comments quickly become islands of irrelevance and misdirection. Don’t be afraid of deleting comments because ‘it might be important to someone else’. We have version controls for a reason.

**Redundant Comments**

Redundant comments are comments that are redundant because the code adequately describes itself. Like this description.

**Poorly Written Comments**

A comment worth writing should be written well. Make sure it is the best comment you can write to describe the technical aspects of the code in a brief, succinct but elaborate manner.

**Commented-Out Code**

Delete it. Do not leave old code in the code base. If it’s important it’ll be retrieved in the VCS. Commented out code gets more and more obsolete with each passing day.

## Testing

**Tests Require More Than One Step**

Running all the unit tests should be doable in one command. Being able to run all the tests is so fundamental and important that it should be quick, easy and obvious to do so.

**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.

**Use a Coverage Tools**

Coverage tools report gaps in testing strategies, and make it easy to find modules, classes and functions are insufficiently tested.

**Don’t Skip Trivial Tests**

They are easy to write and their documentary value is higher than the cost to produce them.

**Test Boundary Conditions**

Take special care to test boundary conditions. We often misjudge boundaries.

**Exhaustively Test Near Bugs**

Bugs tend to congregate. Where a bug is found, do an exhaustive test of the function.

**Patterns of Failure are Revealing**

Looking at code that is or is not executed by passing tests give clues to why the failing tests are failing.

**Test Should Be Fast**

A slow test is a test that won’t get run. When things get tights, its slow tests that are dropped.

## Functions

**Too Many Arguments**

Functions should have a small number of arguments, with having no arguments being the best. Three is the most, and should be avoided with extreme prejudice.

**Output Arguments**

Output arguments are counterintuitive. Arguments should be an input, not as an output. If the function should change the state of something, have it change the state of the object it is called on.

**Flag Arguments**

Boolean arguments are confusing and should be eliminated.

**Dead Functions**

Same as dead code. Anything that isn’t called should be discarded. Don’t be afraid to delete dead code. VCS, remember?

## General

**Multiple Languages** **in One Source File**

The ideal for a source to contain one, and only one language. If there must be more than one, efforts should be taken to minimize both the number and extent of extra languages in our source files.

**Obvious Behavior Is Unimplemented**

Every class and function should implement the behaviors that another programmer could reasonably expect. Classes and functions, like names, should be implemented in a way that a good programmer could intuit its behavior by its name without needing to read the details of the code. If anyone but the author reads the function and is surprised by what it does based on its name, the obvious behavior is unimplemented.

**Incorrect Behavior at the Boundaries**

Go the extra effort to prove that your code works in all the corners and boundary cases (writing and passing unit tests). Correct behavior is often complicated, and it is earned with due diligence. Don’t rely on your intuition and instead earn the correct behavior by testing around boundaries.

**Do Not Override Safeties**

Turning off failing tests and telling yourself you’ll pass them later is as bad as pretending credit cards are free money. Do not sacrifice your longevity of success with convenience now.

**Duplication**

Every time there is duplication in the code, there is a missed opportunity for abstraction. Often the duplication could be another function or class, and by folding duplication into abstraction, the vocabulary of the design is increased. Coding becomes faster and less error prone for all programmers because the abstraction level has been raised.

**Code at Wrong Level of Abstraction**

Abstractions should separate higher level general concepts from lower level detailed concepts. Good software design requires that we separate concepts at different levels and place them in different containers. Isolating abstractions is one of the hardest things software developers do, and there is no quick fix when it’s gotten wrong.

**Base Classes Depending On Derivatives**

The point of portioning concepts is so that there can be independence of different levels of concepts. When base classes reference the name of derivatives, there is a lack of encapsulation. In general, base classes should know nothing about their derivatives.

**Too Much Information**

Well-defined modules have very small interfaces that allow users to do a lot with very little. Poorly designed modules have wide and deep interfaces that forces many different gestures for simple things. Keep coupling low and concentrate on keeping interfaces tight and small by hiding data, utility functions, constants, temporaries and instance variables.

**Dead Code**

Like dead functions, dead code is code that isn’t executed. Dead code should be removed or updated so that it plays a purpose. Don’t ignore dead code, because as time passes, it starts to rot and smells. In the face of dead code, give it a decent burial and delete it.

**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. Private functions should be declared just below their first usage. Limit the vertical distance between the invocations and definitions.

**Inconsistency**

Once a certain way has been done, do all similar things in the same way, unless it is explicitly dirty code. This is to prevent surprises that detract from intuition. This emphasis the importance of choosing conventions.

**Clutter**

Remove all redundancies. If redundancies weren’t such a problem, this would become clutter in itself. No meaningless artifacts, no variables that aren’t used, functions that aren’t called, comments that add no information or even if statements that check for thing that never happen.

**Artificial Coupling**

Things that don’t depend on each other should not be artificially coupled. Take time to figure out where functions, constants and variables ought to be declared.

**Feature Envy**

Methods of a class should be interested in the variables and the functions of the class they belong to, and not the variables and the functions of other classes.

**Selector Arguments**Purposes of selector arguments are difficult to remember and each selector argument combines many functions into one. Selector arguments are lazy ways to avoid splitting a large function into several small functions.

**Obscured Intent**

Code should be expressive. Run-on expressions, Hungarian notation and magic numbers are things that obscure the author’s intent. Code should be intuitive to other programmers in your team who have the appropriate context.

**Misplaced Responsibility**

One of the most important decisions a software developer can make is where to place code. This is where the principle of least surprise comes into play; and code should be placed where a reader (with the appropriate context) would naturally expect it to be.

**Use Explanatory Variables**

One of the more powerful ways to make a program readable is to break up the calculations into intermediate values that are held in variables with meaningful names. More explanatory variables are generally better than fewer.

**Functions Names Should Do What They Say**

If a reader with the appropriate context has to look at an implementation or documentation of a function to know what it does, then work should be done to find a better name or rearrange functionality so that it can be placed in functions with better names.

**Understand The Algorithm**

Lots of very funny code is written because people don’t take the time to understand the algorithm. Before you consider yourself done with a function, make sure there is understanding of how the function works and it’s not enough that it passes the tests, but the author must also *know* the solution is correct. One of the ways to gain this explicit knowledge is to refactor the function into something so clean and expressive that it is obvious it works.

**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. The team should not need a document to describe these conventions because their code provides the examples.

**Replace Magic Numbers with Named Constants**

In general, it is a bad idea to have raw numbers in code. They should be hidden behind well-named constants. Some constants are so easy to recognize that they don’t always need a named constant as long as they are used in conjunction with very self-explanatory code.

**Be Precise**

When decisions are made in code, make sure to make that decision precisely. Know why the decision was made and how to deal with exceptions around that decision. Don’t be lazy about the precision about your decisions. Ambiguities and imprecision in code are either a result of disagreement or laziness. In either case, they should be eliminated.

**Use Structure Over Convention**

Enforce design decisions with structure over convention. Naming conventions are good, but they are inferior to structures that force compliance. Conventions that result in clean code should be enforced by structure that forces compliances whenever possible.

**Encapsulate Conditionals**

Extract functions that explain the intent of the conditional.

**Avoid Negative Conditionals**

Negatives are just a bit harder to understand than positives. Conditionals should therefore be expressed as positives.

**Functions Should Do One Thing**

Functions that do more than one thing such as a series of operations, should be converted into many smaller functions, each of which does one thing.

**Don’t be Arbitrary**

Have a reason for the structure of code, and make sure that reason is communicated by the structure of the code. If a structure appears arbitrary, others will feel empowered to change it. A consistent structure encourages preservation and discourages defiance.

**Encapsulate Boundary Conditions**

Boundary conditions are hard to keep track of, hence put the processing for them in one place.

**Functions Should Descend Only One Level of Abstraction**

The statements within a function should all be written at the same level of abstraction, which is one level below the operation described by the name of the function. Though the idea is plain enough, humans are far too good at mixing levels of abstraction.

**Keep Configurable Data At High Levels**

For constants such as defaults or configuration values that are known and expected at a high level of abstraction, do not bury them in low level functions. Expose those arguments to low level functions called from the high-level function.

**Avoid Transitive Navigations**

We want to make sure modules only know about their immediate collaborators and not the navigation map of the whole system. Transitive navigations are how architectures become rigid. Too many modules know too much about the architecture.