Clean Code by Robert C Martin

Mo D Jabeen

August 10, 2022

1 Variable Names

- Names should show their intention
- Avoid multiple names with small changes between them
- Avoid names with other known meanings
- Avoid meaningless names like data, var etc
- Check if names are pronounceable
- The length of the name should correspond to the scope of the var
- Test good naming scheme has no mental mapping of names
- Method names should be verbs
- Use known subject domain names
- Create names that give context : Classes are great way to link context

2 Functions

- Keep functions very small, should be < 20 lines long
- Should *do only one thing* (the name of the function)
- Condition statements (if, else, while) should only be one line long; call a function
- Keep to one level of abstraction in the function

- Switch statements should be used in a low level class creating new instances instead of jumping to functions
  - Use polymorphism to determine the function based on the object
  - Use design type : ABSTRACT FACTORY
- Long names for functions are acceptable
- Minimise number of arguments (use structs and objects)
- There are three forms of functions:
  - Questions, getting info about the arg
  - Operative, transform arg and return it
  - Event, Trigger change of state based on arg
- Avoid flag arguments
- A list of related variables can be considered one arg; (meh(int blah, int ...args))
- Function arguments should not be changed in the function (the output should be via return)
- Remove any duplication of calling functions !!

3 Comments

- **Good comments dont replace bad code !!**

- Good for showing intent (why you made this decision)
- Use comments to add extra clarification on confusing syntax
- Show warnings on important areas of code
- Can be used with IDEs for Todos
- Avoid statements that are not completely accurate
- Avoid commented banners instead use more effectively organized code.
- DONT COMMENT OUT CODE, delete it; rely on source control to retrieve old code.

4 Formatting

- Small files are preferred
- Related concepts should not be in different files - Test: Avoid needing to jump between many files to understand a file
- Instance variables are placed at the top
- Caller functions should be placed above the callee
- Dependencies should flow down through the code
- Avoid very wide lines of code

5 Objects and Data Structures

Abstraction is all about simplifying and detaching results (details) from an implementation.

- Good abstraction allows us to not care about the details of how to interact
- Your focus on the abstraction is how to manipulate the essence of data

- Interfaces give you **answers/results** not variables and data points.

Objects: hide their data behind abstractions and expose functions that operate on that data.
Data Structures: expose their data and have no meaningful functions.

Objects are essentially the opposite of data structures, they should facilitate results not be used for compute. Use procedural code for function based code and objects for data based code.

- Procedural code with data structures are good for new functions and bad for new data types
- OO code with objects are good for new data types and bad for new functions

5.1 Law of Demeter

- Dont access class variables, its only use should be via internal methods.
- Avoid chains of function calls ie meh().hem() etc
- Avoid hybrids of OO and procedural; vars should either be freely accessed or encapsulated.

6 Error Handling

Error handling should not be obscure, hidden within functional code.

- Best to scope out errors used in exceptions an not leave them broad.
- Good errors allow you to find the issue, they should state the operation that failed and the type of error
- Its best practice to wrap any third party APIs for better control and easier to read implementations
- Preferable to use SPECIAL CASE class or objects, instead of a null/error that is caught as an exception and dealt with.

7 Unit Testing

TDD : Tests Driven Development, tests should be written before production code. Tests need to be developed with code as, crappy tests or old tests are as good or worse than no tests.

- The most important aspect of tests are they are easily readable and therefore easy to maintain
- Use the BUILD OPERATE CHECK pattern
- Each test function should only test one concept

Test should follow FIRST:

- Fast (to run)
- Independent (not dependant on other tests)
- Repeatable
- Self Validating (Boolean output - Pass or fail)
- Timely (Written before production code)

8 Classes

- Always avoid public variables (accessible from anywhere) in a class
- Encapsulation is important to ensure clear separation of concepts
- Classes names should define the scope, they should always be small
- Classes should follow the SRP principle (only have one reason to change)
  - To create better abstractions, think about the reasons you would change the class
  - The methods inside the class should use many of its variables, to create cohesive classes. This is preferred so the class acts as a complete whole.

9 Systems

- Startup and runtime logic should be separated
- Major dependencies should be carefully considered and resolved with a consistent global strategy
- An option for separation, is to use main to setup and initalised before passing to run time logic
- Some items need to be made in an adhoc manner, these should be separate (not intwined in the logic) and their creation should be abstracted away.

9.1 Kent Becks 4 rules for simple design

- Run all tests
- Contain no duplication
- Express intent of programmer
- Use standard design patterns to easily show intent
- Minimal number of classes and functions

10 Concurrency

- Threads for I/O
- Processes for data sets
- Concurrency should be considered a separate issue to single thread programming
- Concurrency issues arise from the large number of possible execution paths for even simple code lines

10.1 Concurrency defense principles

- Single responsibility principle (single reason to change)
- Keep concurrency related code separate from other code
- Limit the data that might be shared
- Preference to use copies of data and later merge into single thread
- Create independent threads
- Wary of dependencies between threads
- Keep synchronized sections small
- Carefully manage graceful shutdown
- Tests have potential to expose problems, run them frequently with different configurations and dont ignore errors as one offs.
- Tests single threaded functions first

10.2 Concurrent execution models

Producer consumer

- Queue, with some threads writing to it and others reading

Readers writers

- Writers wait until all readers are done. Kill off readers if they are running for too long.

Dining philosophers

- Only act when required and there is enough capacity