Clean Code Book Notes

December 20, 2024

Chapter 1

- **Leblanc's Law: Later = Never**
- Never delay writing good code, always write it starting from prototype stage, don't just do it to get something working, this will never end well and cause mess and never be refactored. As mess builds up productivity decreases with time.
- **Reducing NOISE will make the code look cleaner, don't show user the code that need intent from somewhere else, talk about what's happening now**
- Plan ahead and write good/clean code from start
- Only the BEST and BRIGHTEST are selected for the tiger team!
- Manager's want truth even if it's bitter, the schedule depends on the programmer at the end of the day, they will defend the schedule with customers, it's your job to plan ahead and have adequate time to do things correctly so thread carefully while planning. Don't be unprofessional, stick to quality of code.
- messy code will never make you meet the deadlines, the only way to meet deadlines, the only way to go fast, is to keep the code as clean as possible
- being able to recognize good vs bad code is not the same as being able to write clean code
- **Some of us are born with it, some of have to fight to acquire it** - this applies to anything in life
- clean code = elegant, easy to read, provides only way to do things, performant, has tests, makes it easier for others to extend it, orderly and simple and cared for, non-duplicated, does only one thing, smaller is better (lol)
- Duplication and expressiveness, do only one thing and simple abstractions - tools to improve code quality: Jon jeffres
- Tests force the code to be clean and reliable, add them early
- Boy scout rule: *leave the campground cleaner than you found it.*
- practice! practice! practice!

When hand-washing was first recommended to physicians by Ignaz Semmelweis in 1847, it was rejected on the basis that doctors were too busy and wouldn't have time to wash their hands between patient visits

Gracie Jiu Jistu, founded and taught by the Gracie family in Brazil. We see Hakkoryu Jiu Jistu, founded and taught by Okuyama Ryuho in Tokyo. We see Jeet Kune Do, founded and taught by Bruce Lee in the United States.

Chapter 2

- **Names should reveal intent -** let it be variable, function, classes, DS, package etc.
- The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used, of it requires a comment, you've failed at naming.
- Avoid leaving false clues that obscure the meaning of the code ex. AccountList variable must be a list.
- Use pronounceable and searchable names
- Method names should have verbs
- Don't let people use mental mapping to remember variable names etc. be clear about names and their intent, clarity is king
- When constructors are overloaded, use static factory methods with names that describe the arguments. Consider enforcing their use by making the corresponding constructors private. Ex. in RZUUID we have FromStrFactory, this is a wrapper over one of the constructors, don't let users directly access the constructors
- Pick one word for one abstract concept and stick with it, for ex. don't have release/destroy/freeResource for the same thing be consistent with the concept
- do what it means, if you call a functions "add", it must add something, don't call it that for the sake of consistency
- use solution domain names, math/design patterns/algos/CS terms etc.
- prefixes and suffixes add context to naming, they suggest they belong to a family, ex. `fsGetSize(), fsWriteToDisk` suggest they are from the filesystem API
- don't add gratuitous context ex. *RZRenderContext*, when using an IDE typing RZ will give a plethora of stuff, so keep the names simple

Chapter 3

- **First Rule: Keep the functions as small as possible**
- Second Rule: Keep the functions smaller that the first rule
- Functions shouldn't be 100 lines long, hardly 20 at max
- Functions should tell a story, be transparent and obvious
- Blocks within *if* statements, *else* statements, *while* statements, and so on should be one line long. Probably that line should be a function call
- **Do one thing! -** Functions should be small, do only one thing and do that well and do only that
- If a function does only those steps that are one level below the stated name of the function, then the function is doing one thing. If it does things on another layer below, split it into the next layer of abstraction
- All statements within the functions must be on the same level of abstraction for it do only one thing
- **Stepdown rule:** We want the code to be readable from a top-down structure and tell a story

- Problems with *switch* statement: it's big, does more that one thing, violates SRP (single responsibility principle), violates OCP (open closed principle)
- Use descriptive names and be consistent, use the same nouns, verbs in function names, stick to the convention initially agreed upon
- Use a phraseology to tell story about function names
- Keep the number of arguments to a minimum, more than 3 is a clutter, as Lois says "De-Clutter!" consider using a struct in that case
- Args are almost and always inputs, keep output args minimum and rely on return types instead
- Passing a boolean into a function is a truly terrible practice, avoid them! This means that the function does more than one thing
- Don't use dyadic form (2 args) unless they have a natural cohesion and ordering, ex. setCartesian(int, int) is fine but passing the fstream to a writeFile function along with path name is not a good function ex. writeFile(fstream, String) is Nah!
- Reducing the number of arguments by creating objects out of them may seem like cheating, but it's not, use PODs as function arguments increases
- Functions that take variable arguments can be monads, dyads, or even triads. But it would be a mistake to give them more arguments than that
- Function names should be verbs that explains their intent clearly
- **Have no side effects!** - don't promise to do one thing and modify something else, there should be no hidden drama! These lead to **temporal coupling!**
- **Anything that forces you to check the function signature is equivalent to a double-take. It's a cognitive break and should be avoided**
- Functions should either do something or answer something, but not both. *while this is good practice, I don't completely agree with it, consider a function that send data over the network using sockets, this will return the amount of data that was successfully transmitted while also telling sockets to send the given data, here while this principle is violated, it's okay to do so!, thread this rule with caution while applying in practice*
- **DRY principle:** Don't repeat yourself. Avoid duplication!
- *Caution!* **Structured Programming:** Dijkstra said that every function, and every block within a function, should have one entry and one exit. Following these rules means that there should only be one return statement in a function, no break or continue statements in a loop, and never, ever, any goto statements
- How the author write better functions? *When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code. But I also have a suite of unit tests that cover every one of those clumsy lines of code. So then I massage and refine that code, splitting out functions, changing names, eliminating duplication. I shrink the methods and reorder them. Sometimes I break out whole classes, all the while keeping the tests passing*
- Functions are VERBS, classes are NOUNS of the language to TELL A STORY!

Chapter 4

- Avoid comments unless you can't express yourself enough via code or need to provide a more deeper thought process/insights or architecture design decision or caution
- So when you find yourself in a position where you need to write a comment, think it through and see whether there isn't some way to turn the tables and express yourself in code. Every time you express yourself in code, you should pat yourself on the back
- Spend the energy in writing more expressive code rather writing comments instead
- Comments are hard to maintain and can quickly become obsolete and misleading as code evolves over time
- Comments are valid while trying to explain intent/warning and clarify some decision made that is beyond the logic and intent of the functions
- TODO comments are fine but don't use them as an excuse to leave bad code behind!
- Don't write redundant comments, ex. inline get/set functions doesn't certainly need the energy to write comments
- Usage related comments are fine, they act as docs in a larger scope, comment the design not the obvious code
- **Don't leave commented code!** - *Others who see that commented-out code won't have the courage to delete it. They'll think it is there for a reason and is too important to delete. So commented-out code gathers like dregs at the bottom of a bad bottle of wine*
-

Chapter 5

- Keep functions that are related in terms of a concept close together to avoid jumping all over and reduce cognitive strain on the reader, keep the vertical distance to minimum and respect conceptual affinity
- If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible
- Keep the horizontal limit to 80-100-120 lines max
- Horizontal alignment: controversial topic let's leave this to personal preferences
- As a team/individual stick to some rules and follow them consistently

Chapter 6

- A class does not simply push its variables out through getters and setters. Rather it exposes abstract interfaces that allow its users to manipulate the *essence of the data*, without having to know its implementation
- **Objects:** They hide their data under some abstractions provide functions to operate on them
- **Data Structures:** They expose their data and have no meaningful functions

- Objects and Data Structures are kinda of virtually opposite, **Procedural code (code using data structures) vs OOP**
- Procedural code makes it easy to add new functions without changing the Data Structures where as OO code makes it easier to add new classes without changing existing functions
- Procedural can't easily add new data structures while OO can't easily add new functions because all classes must change
- the things that are hard for OO are easy for procedures, and the things that are hard for procedures are easy for OO!
- In any complex system there are going to be times when we want to add new data types rather than new functions. **For these cases objects and OO are most appropriate**
- On the other hand, there will also be times when we'll want to add new functions as opposed to data types. **In that case procedural code and data structures will be more appropriate**
  For ex. in a rendering backend abstraction, it's better to have it as PODs as we add new functionalities as the no. of backend API classes are limited to 2 or 3 (DX12/VK)
- **Law Of Demeter:** TBD TBD TBD
- BASICALLY don't return too many different types objects as they lead to an explosion of method calls and knowledge, keep the abstraction simple over the bigger picture operation, your abstraEction should handle your end goal in 1-2 function calls at most

Chapter 7

- Exceptions helps maintain code aesthetics and separate error handling from function class. For ex. just throw a exception and handle it elsewhere
- keeps things de-coupled and doesn't crash the program, it provides an alternative path
- While they are perfect for languages like python scripts to handle builds etc. generic renderer cannot benefit from exceptions as a game shouldn't cause such behaviour
- idk just go with flow and intuition

Chapter 8

- Frequent type casting is not clean code
- Write wrappers over third party libs to restrict access and set boundaries for the API. Tailor the interface to meet the needs of the application over other's libraries/modules
- While using a boundary interface, keep it inside the class, or close family of classes, where it is used. Avoid returning it from, or accepting it as an argument to, public APIs
- **When trying out a new library, Instead of experimenting and trying out the new stuff in our production code, we could write some tests to explore our understanding of the third-party code -** This is called learning tests, use this to learn the API and see how it would be integrated with our code
- The learning tests end up costing nothing. We had to learn the API anyway, and writing those tests was an easy and isolated way to get that knowledge

- They also help track behavioral changes with new releases of third party libraries - a win-win situation
- **Use Adapter Pattern to fake API that will be designed in future**
- When we use code that is out of our control, special care must be taken to protect our investment and make sure future change is not too costly and defines tests that set the expectations clear
- Use them in very few places and that too wrap them over or use adapter to keep direct interaction to a minimum

Chapter 9

- **Unit Tests** are the corner stone of **Production Code**
- Write tests first and let your API evolve while keeping the tests working, **in fact production code and tests are written together**
- Good Tests will make the API ready for production environment and keep the product stable
- **First Law:** You may not write production code until you have written a failing unit test
- **Second Law:** You may not write more unit tests that are sufficient to failing, unless they can compile, write compilable code before any tests pass
- **Third Law:** You may not write more production code than sufficient to pass the current failing unit tests, don't get lost in writing new stuff, make the tests pass before adding new functionality
- If you don't keep your tests clean, you will lose them. And without them, you lose the very thing that keeps your production code STABLE
- Without tests you will be reluctant to make changes because of the fear that you will introduce undetected bugs and break other things
- What makes a clean test? Three things. Readability, readability, and readability - *Think of them like API usage docs and this thought process should help you write clean and good tests*
- Reduce noise while writing tests to keep intent clear, don't force people to read unnecessary details that won't serve the purpose at hand
- **BUILD-OPERATE-CHECK pattern:** Each of the tests is clearly split into three parts. The first part builds up the test data, the second part operates on that test data, and the third part checks that the operation yielded the expected results
- Programming language and Test API abstractions help write cleaner tests, for ex. GTest and noexcept stuff will give you a wrapper API to write tests and keep it simple
- Write only one assert per test and one concept should be tested per test
- **F.I.R.S.T** - Fast - meh, Independent - doesn't depend on other tests, Repeatable - run in every env and config, Self-Validating - either pass or fail, Timely - write just before production code

Chapter 10

- Classes should be small, same as functions keep them as short as possible - (*not literally thought, we use a a different metric like no. of responsibilities*)
- There's no hard limits like functions but keep the no. of responsibilities it does to minimum, a *ShaderLibrary* class should be relative small enough to just handle load/unloading shaders etc. it doesn't make sense for it to handle shader manipulation, like setting params etc. keep it's job to simple book keeping of shaders and be done with it
- **SRP - Single Responsibility Principle** - A class should have only one reason to change. okay wdym? - one way is to maintain a cognitive cohesion over it's functions, a PhysicsSystem can't do any rendering stuff etc. or a AssetClass can't contain Version, it must be a separate class called AssetVersion for decoupling and to respect SRP
- **ask yourself this -** Do you want your tools organized into toolboxes with many small drawers each containing well-defined and well-labeled components? Or do you want a few drawers that you just toss everything into?
- **At any given time while writing classes or functions, the reader should only have a mental map of the minimal no. of systems to understand the problem being solved at hand, unnecessary mind mapping of bloated information that goes all over the place is an indication of bad code**
-