ANDREW MOBUS

ROUGH\_ELEVENS DEVELOPMENT DOC

[03-14-2024][000]

The software development cycle has a lot going on; much like coding or programming itself, there's often more than one way to go about accomplishing the same objective. As with code itself, software is ultimately for humans; code is read far more frequently than it is written. While one could conceivably mean or include "executed by a computer" when saying "read" in the previous statement, the intent is scoped, and true, for humans; humans write code in high level languages because it is easier to figure out what’s going on, and keep track of all the myriad moving parts, than attempting to describe machine code, or assembly, or even raw binaries. In fact, raw binary information doesn't encode in of itself to any particular set of actions or operations; different paradigms and schema are used to interpret what different chunks of "raw data" mean in the context of the execution. Therefore, programming is an explicitly contextual pursuit.

Within programming, within software development, there is an inherent relationship between how much context is described for an operation and the resource cost of performing that operation. This is an unavoidable tradeoff, in some senses; given a data set and an operation to perform on that data set, the cost of performing that operation is proportional to the size of the dataset, all other factors being equal. Our primary mandate as developers is to construct data sets that, when processed, produce the desired output; our secondary mandate is to do so as efficiently as possible. That is, provide only as much context is necessary. No more, no less.

So, how much is too much, and how much is too little? This is one of the core questions developers analyze. Raw data has, in some respect, a constant quanta (size), and thus figuring out what data is required is the abstract answer to how much data. How verbose or terse a given dataset might be is dependent on who, or what, is reading, or processing, the data set. Usually, the issue is how much information is needed to tell apart one outcome or instruction from another -- quite literally, distinguishing 'this' from 'that'. Humans vision has rods and cones, which process different qualities of light or vision; how much light (measured in lumens) a human requires to distinguish one shape from another is different than distinguishing one color from another.

How much light one needs to 'see' (process) the object is a function of what information one needs for the goal in mind. If it takes more light to discern color than shape, and you don't care about the color of the object, then you need less light than an application where you care about the color. You can afford to skimp on the warehouse lighting if each parcel has a unique shape identifier without worry about parcel collision. Similarly, if you identify parcels by shape and color, or indeed just color, then you really ought to spring for the more powerful lighting, despite the upfront cost.

The described dynamic applies to the data we provide to computers, but equally, and often more, this dynamic holds true for communication between humans. This report is an example of such -- what information do I need to include in order to communicate to you, the reader, that I have fulfilled (and in fact exceeded) the given objectives. Some of that information is contained in the code itself. I could tell you that I internalized the importance of test driven development by changing the data set size from (9) to the entire deck, which therefore allowed me to determine whether or not the processing algorithm could correctly match all states, or I can just write a test that carries out that operation -- or do I need to do both?

The question of why I spent so much time creating an automated processing schema (an algorithm, in other words) when that wasn't explicitly part of the assignment has a couple answers. The simplest is that creating such an algorithm was a good exercise to improve my abilities as a programmer, and to develop a specific tool or skillset associated with projects I want to create in the future. However, that algorithm also allowed me to effectively and efficiently debug my code for control leaks -- for unexpected behavior -- by stress testing the constraints I gave to the algorithm. In doing so, I could be exhaustive in my approach, like running the program 100,000 times to figure out if the pattern I saw emerging was real or just a passing blip.

However, there’s a lot of information from the development process not present in the code; everything that was backtracked and changed, all the bugs that were fixed, the changes in implementation. Just providing the reader with what amounts to a core dump of information (i.e., all the text I produced in the course of development) would not be helpful -- I'd have to have that information chronologically sorted to be even remotely processable. This is a practice I used to do, and have reinstated, when writing code - whenever I make major changes to a chunk of code, or remove something that, when I wrote it, was a solution to a problem I had solved for the first time, I would copy that code and paste it into a text document for safe keeping. The purpose was, in part, so that if I decided that what I ended up doing was not a great idea, I had state information that I could, if needed, manually sift through. Setting up the infrastructure for this kind of bulk dump is not necessarily difficult, but requires consistency and evaluating the question "is this code worth keeping around?".

Part of why I do this is because otherwise, I just comment out code and keep it around so I know what my other options where -- it provides a kind of context, much as human language comments do. Often, these would be mixed with human language descriptions too. On the one hand, this means that the information is near (what seems to be) its context, but on the other hand, it isn't, because that code was only in context at that point in time, and the rest of the system may change. As a result, the 'documentation' provides conflicting & inaccurate context, while also creating noise for me to process when trying to keep track of what my code does and what's current code. Its far better for me to just dump as much context (the actual state of the code, i.e., the code itself) from that moment, provide a datecode, and move on. If I need something from that code, it’s better for me to 'rediscover' and excavate the code from the bulk file than mixing current and potentially out of date code.

By the same token, any documentation on the document should be reserved for context clues anchored in the system as would be found by someone retrieving the system 'from disc' with no context cached. By reading the code, I can figure out what the code does, and the comments can help by providing non code information ("I don't like this implementation but it works") ("this code has a scaling issue") ("works but buggy"). This abuts against the concept of documentation itself; code is, in some ways, its own documentation, because code says what it does; that's literally what code is. Code doesn't necessarily tell you why it does what it does, or the decisions reflecting outside circumstance ("this code required using an external library which sucks but is needed to meet the project requirements, even if I don’t think it’s really needed to meet the objective"). Documentation isn't free, however; like code, there is a cost associated with its generation.

Perhaps uniquely, as developers we can address this problem with the same tool we're creating: code. There's a cost associated with creating automatic or procedural documentation paradigms, which may be offset by the ease of information processing going forward. Maybe every time a function has its body code change, but not the name or parameters, the system logs this information under that key (the function signature); that way, to find all changes to that function, we could just query the function signature as the key. That's within range of, say, my implementation ability, and something (roughly) that I began work on in the course of this project, but here’s the catch -- that documentation system is not part of the local, near term objective requirements. Based on my experiences in my own projects, and my experience programming in general, I can recognize that in general this tool is worth the investment, but it may not be appropriate or urgent to do now. The proper prioritization of objectives is therefore as important as the objectives themselves.

Identifying objectives, and then identifying those objective's dependencies, allows working backwards from the desired state -- the goal, the final product and destination. We repeat this process to break down requirements into more manageable chunks, both for distributing workload and for scaling the information scope so that the developer doesn't have to be cognizant of the entire system operation when creating a solution. On a human team level, this equates to effective task definition and scheduling. By the same token, however, if everyone is only figuring out how to solve the same problem right in front of them, while we might get the best particular solution to that problem, we'll have to discard a lot of work effort. If contributors work on solving different parts of the problem, a workable solution might be reached with less gross expenditure of effort, and subpar components can be retrofit as required. Where does this all relate to our simple console game project?

As you have pointed out, CSC350H is not a programming or coding class -- it is a software development course. Software development can be a solo venture, but usually, most developers are working with other developers, within a system that dictates the objective and resource parameters. Learning to develop software means learning how to work as part of a team, where when the team succeeds, everyone succeeds; the rising tide raises all ships. If the team fails, it is a question of who gets a lifeboat, who gets the flotsam, and who is left to drown. We are rarely called to rebuild the entire ship, nor generally are we permitted to do so. Working within the system constraints -- using the tools provided or "speaking in their language" -- is a critical skill. Many of these lessons are much like reading code; we see the system as it works, and from it deduce conclusions as to the why. When one knows what problems one is trying to solve, one can more easily identify solutions, even if they may not be immediately required.

For myself, understanding logging and documentation is pertinent for me both to working with teams but also keeping my life, in general, manageable. I learned last semester just how damaging what seemed like small documentation inefficiencies can be. I scattered my physics notes through the rest of my note after filling the physics section, rather than immediately getting a new physics notebook. As a result, I never referred back to my notes because they were hard to process and find. I was worried I would fail physics, and there came a day where I ripped out all of the physics pages to consolidate them; you can be sure I was thankful I had at least dated all my notes so I could put them in order.

So, I've extolled the virtues of good documentation and logging, of having long term objectives against which to orient oneself in the development process, the necessity and problems with working in a collaborative team environment, and a whole lot of stuff on the abstract concept of data and process abstraction; it feels like I'm missing something really simple, though. Oh, I know -- given that I had the auto solver working & validated (meaning the rest of the code worked & was validated as well) about two weeks before the assignment was due, what could possibly explain the fact that I'm turning this is a day late. I've vastly exceeded the requirement in many ways -- including the depth and breadth of this writeup -- but I haven't submitted or completed a number of the basic unity labs, nor did I submit the midway framework portion of the assignment at the end of the (official, given) first week.

While there are meaning asterisks -- I've done some amount of informal checking in, and feel confident in my system understanding of how things like Unity work -- I feel these issues are a good indication of a larger issue I struggle with. Call it "missing the trees for the forest" or "premature abstraction & optimization" or just getting sidetracked, I feel there's a clear pattern of stepping in the pitfall directly in front of me because I'm too focused on addressing an obstacle I perceive further down the road. Sometimes it feels like the worst of both depth & breadth first traversal search algorithms; I spend too long exploring down an extended node chain, or I spend too much time performing lateral queries.

These are both issues of prioritization, especially in an order of operation sense. I do think the inspired project I started (and made serious development headway into) yesterday and today is a good use of my time, and both strengthens the lessons I've learned in this process while continuing to build further. However, that assessment is largely in the abstract, and the conditional clause is I have a bunch of other work that needs to be done sooner. I feel confident that some of the issue really just is a schedule and organization problem, which compounds with some maladaptive stress responses. That's part of why I focused on working so hard on something I ostensibly didn't need to, because it reflected one of the first times I've been genuinely ahead of the curve in terms of getting work done well before the deadline. I didn't explicitly set out to do so, so part of what I've been trying to do is figure out what factors contributed to my increased productivity, both in terms of raw output and in terms of accuracy to the provided benchmark.

There's always more that could be done -- improvements and optimizations, future proofing, the kitchen sink -- but, as with any journey, stopping every once in a while to get your bearings (and make sure you aren't lost) is a good idea. I've often had overly ambitious goals in the past when it's come to projects, and let my desire for something more meaningful or sophisticated get in the way of meeting the core requirements -- that’s a lesson which took a long time for me to learn. The grand vision rarely justifies the deviation from protocol or the inevitable delay, in terms of what that vision eventually yields. Here, in this case, I think it does - as long as I can stick the landing and turn this in.