Tejas Menon

Professor Karla Fant

CS202: Object Oriented Programming (Summer 2017)

Assignment: Reflections on OOP

26 August 2017

Reflections on OOP

Working in an object oriented style and environment is something I think that works most meaningfully and idealistically in a very large program or system, where several objects or packages already exist and one is required to extrapolate those for a personal class or agenda, and thereby use 'external' solutions for your benefit to the largest extent possible. This ideology was something I missed in the beginning (CS162 time!), where I had wondered why we are creating so many classes when we're working with the guts of a program anyways (deep copying char*'s for example). What I had initially missed was the notion that I was to create a program that could later be copiously expanded, and others could use my implementation for their purposes. Once this mindset began setting in, I could surmise the necessity for a particular class, and if it intuitively felt meaningful for the functions I needed to implement or for the imaginary 'someone else' in the lifetime of the program, I set apart goals for it. The exigence of a class was decided therefore by the extent of its functionality later in the program and not simply by what 'seemed' like an appropriate addition.

An example of this was my 'misc' class in program 2- it was to be a superclass of every other class except galaxy, and manipulate other passed in misc objects to synchronize it with the all misc objects for all bodies throughout the galaxy. This simple but profound addition made very

few operations affect several objects at once, for instance with the passage of time- depicted artificially by the movement of the spaceship (another derived object of misc)- I could update the position information of all other bodies affected by this movement. This in turn made personalized spaceship motion easy to implement without requiring several callers and setters throughout all the classes. I could simply pass the modified spaceship object to the galaxy, and it would cause a function invocation on each celestial body- this function being in the base misc class of each body. This therefore made the management of 'time' a very high level process- the client could choose which galaxies/other planets would be affected by the movement of the spaceship and therefore the passage of time. And this I believe is the telling feature of good OOP - good solutions are always ones that simplify/make doing something a very non-specialized process- I can update planetary positions for simulating various other obscure occurrences such as planet collisions/specialized orbits using the same method, providing essentially a future developer a fundamental method to interact with the positions of planets- but not necessarily doing all the work they would want. This idea is very much like the streaming libraries in java and C++ that you can use to stream several sources such as input, files and other character buffers using the same functions albeit in different self-similar classes. CS202 was a course that in this regard critically looked at efficient design and instilled few 'best practices' while coding with OOP.

One problem, or best termed 'mirage' I encountered throughout this course, and may have cleared up slightly with Java programming, was the setting of boundaries between the data structure and the data stored. I noticed that including data structure like functionality to the actual data rather than have containing relationships between a class node and the data largely produced succinct

code with meaningful and self-sustaining classes- for program 3 I didn't even need to return the next pointers for my LLL of meetings, they managed their own LLL by calling the same function on the next obj for e.g. next->add_meeting(a_meeting) that recursed until next was NULL to append a meeting to the end. This then brought about classes that were all autonomous, and each higher level class was strictly restricted to calling functions to produce more change than setting and getting data. For a while, this mentality of trying to constrict functions of a node and data structure to the data we were working with was very rewarding and tenable, but as the functionality for the data structure grew and it became difficult to dedicate data structure jobs to different classes, I had to revert back to containing relationships and returning next/left/right pointers to a class- which then produced inconsistent code with the two styles merged. It is through this straining in reconciliation that I came upon my second learning for this course- there's no perfect design and rather it is one that immediately feels intuitive and most likely, one that stands out only after several trial and error propositions. And even at this point, there may be better designs on the horizon, but any design that has been worked on sufficiently long enough depending on the size of the project can be regarded as somewhat closer to the best design than what would be an initial draft. I conjecture here that any OOP problem can be solved using the most general solution- having a hierarchy with the data and a data structure completely distinct from it (in nodes and a list/table class) - while one can make better this solution by specializing it for an individual program- carefully foregoing any extra classes that may not be as necessary. A risk of employing this strategy has to do with not considering every instance of its usage, which can then mean unnecessary and counterproductive workarounds. So there seems to be a thin line between the most general solution and a highly specific solution where the best solution lies, and visualizing this design may not be immediately possible. This could also be considerably

dependent on the number of classes that one requires to solve a problem- a large program with 50 classes would require more time to find a perfect solution for than a smaller program with 5 – maybe writing a program that finds the perfect design for a given problem might be something fun to do. But then again, how do we find the best design for *this* problem? A recursive solution?!

What interests me in OOP, and very much programming in general, is that it stands as entirely distinct from other subjects or fields as it applies all learning rather than requiring its regurgitation, and that it requires substantial creativity apart from traditional bookish knowledge. More specifically, simply learning about inheritance, data structures and its combination with several assignments may not produce better programmers, but learning to creatively think out of the box and spending ample time developing an intuition for thinking about a problem in the light of OOP- or modelling it through the perspective of the abilities of a computer, brings about a change not just in programming style, but the very nature we think about things. For example the next time I think about the way in which ‘time’ is simulated in games, or the search algorithm to find best results for a selected choice, I can make educated predictions on what could be the process, its efficiency in big O, and effectively learn to create and make better, simply by observing. Even at this very moment, as I view the cursor moving across the screen, I can predict that some form of a ‘list’ data structure was implemented, and each line could possibly be a data member in an LLL with each word simultaneously being passed to a balanced tree or being checked off on a dictionary hashtable- since searching is exceptionally quick- and some GUI class tracks the position of the mouse and cursor that when clicked on a certain line,

activates functions for that particular node. Since this is also very quick, we can imagine that each node address is maybe hashed additionally with the line numbers.

Regarding efficient algorithms, a very firm parallel I sometimes use to strike is to think about myself accomplishing that task. For example while sorting numbers in ascending order, I try and observe what exactly the mind does to get that order- it must also traverse the entire list at least once to find the smallest, and maybe perform some additional steps along the way to reduce the number of passes. For example if there is a relationship between the data such that they are in a certain range, I can store all the data smaller than the average of the data I have encountered and sort them as I go. This would only require two complete passes (one for below average and one above average) with single sorting steps in the middle but it is still many times better than the O(n^2) solution I would have initially. Solely observing the mind for a solution would be inadequate, but it can be used to safely predict whether a particular solution can be improved further or not.

In conclusion, programming in OO, or solving a problem in many ways involves observing its real world counterpart, breaking its features down into fragments of similar functions and finding the relationship between these fragments in a hierarchy/containment, and only then begin to consider its technicalities within the sphere of computing. Sometimes, I feel it is mistaken for a complex process involving special insight or a spark of brilliance, but mostly it is only a unique application of our learnings onto the realm of everyday life.