Chocoholics Anonymous: a Retrospective

Ethan Fleming, Long Vo, Tommy Ngo, Alex Staley, Trent Wilson, Carl Bunt

December 6, 2019

Chocolate addiction is a harrowing experience. It can destroy families, careers, and the very minds of those unfortunate enough to suffer from it. Fortunately, there are organizations that can help. Unfortunately, those organizations depend on reliable software, and developing that software can be as harrowing as chocolate addiction itself.

One factor we considered when first deciding how to go about development was the fact that we didn't know each other. That is, a pair or two of us had been in classes together in the past, and worked on projects together, but by and large each of us was an unknown quantity to the rest of us. Considering that, we elected to adopt an informal structure of responsibilities. No one developer would be "the UI guy," there would not be a "backend" and a "frontend" team, and if some piece of the software lagged behind the rest of the system, we would have no one but the entire team to blame. This way, we wouldn't shoot ourselves in the feet by overlooking each unknown team member's mystery skill set and assigning them jobs that other team members might have been better at.

A week or so into the project we amended this "free love" style to an informal set of job titles, which we left in a spreadsheet to be claimed on a first come, first served basis. One of us would serve as the Project Manager, one as the System Architect, one as the System Designer, and the remaining three as Programmers. Of course, it was stressed, we would all take part in all necessary tasks, and the title one ended up with would not burden that person with (or relieve that person of) any specific responsibility. Nominal roles, to be sure, but it felt good to organize our group into some sort of sensical structure.

Armed with our feels-good-but-doesn’t-pigeonhole-us-into-any-scary-responsibilities hierarchy, we attacked the project with vigor. Of course, coordinating times to work together proved very difficult (read: impossible), being that busy students such as we have lives, even jobs in some cases. The requirements document came together smoothly enough despite this, and despite a slightly uncomfortable awakening when we put our slightly-structured-free-love model into practice. The design phase of the project went smoothly as well. Perhaps… *too* smoothly.

Backing up a bit, we chose the waterfall development model as the ideal methodology because we noticed a “methodology” chapter in the design document template that looked like it needed to be filled, and agile still sounded too much like witchcraft (or at least, like a technique that works best when one knows what the hell one is doing). With the waterfall method, we could relax in our confidence at the rules having been neatly laid out for us.

This confidence was the primary culprit in the unsettling smoothness of our design phase. When one designs a software system on paper, one’s design can take whatever shape that pleases one. One can frolic in an object-oriented paradise, blissfully unaware of the behavior of one’s design when it becomes implemented. One can freely and wildly speculate about the benefits of taking advantage of the project to get to know an unfamiliar database library. One can set all sorts of traps for oneself. Our system architect is now convinced that the main benefit of waterfall-style development is that it allows the implementation phase to begin with confidence and momentum acquired during a happily ignorant design phase.

In a nutshell, the problem was naming. Having an object-oriented design filled with objects whose names are identical to the objects’ real-world counterparts is all well and good.

But when the objects themselves don't do quite the same things as their real-world counterparts, matching up in some ways but subtly differing in others, you have a problem. We had a problem. Providers, members, services, reports, managers, users… all were elements of our system just as they were real-world users of the system who did things unrelated to their same-named virtual counterparts. Before we had even opened an editor, our design doomed us to fail by virtue of pure confusion. By the time we realized the conceptual swamp we'd gotten stuck in, it was too late.

Witchcraft though it may be, agile may have been a better choice for our development model. If we had felt free to begin implementation while the design was still in progress, we could have used the reality of the system taking shape to inform our design, and changed our naming system to something less easily confused with reality. As it happened, implementation and its associated rude awakenings hit our team pretty hard, and not just in the naming department. It was already week 7 or 8 when we realized we were operating in several different development environments, using several different compilers. Issues associated with this setup dogged us from the second we created our first file, and in fact continue to dog us. This document really should more properly be referred to as a "contemprospective". In fact, the author of this contemprospective is only writing this contemprospective because he is the only group member currently unable to get the project to compile on his machine.

Several other factors contrived to bedevil us throughout development. One was the wide variation of opinions on the value of clearly and comprehensively commented code. This was a boon, since we all are now much more experienced in deciphering code we didn't ourselves write. Another problem resulted from our hasty language selection -- namely, the amount of time we had to spend fixing bugs that are C++ specific (segmentation faults and the like) rather than working on actual logic relevant to the design. Given the chance again, we might choose a language that allows for quicker and simpler development, especially for cross-platform solutions, like Java or C#. Additionally, coordinating a uniform development environment for everyone -- say, a single IDE rather than two different ones and the command line -- could have been simpler in a different language. If only we could have focused on perfecting the actual features of the software we were developing rather than spending hours trying to get it to compile, this project might have turned out differently.

Haphazard though the development process has been, it hasn't been all bad. We all increased our Git and GitHub chops several dozenfold. Some of us got their first taste of team-based software development, such as it was. A couple of us got to learn SQLite, after a hasty decision to use it to implement our so-simple-csv-files-would-have-been-overkill database (this particular ambition ushered in much woe and folly). But the biggest achievement we can claim is that we find ourselves still in a relatively positive frame of mind, and none of us feel any overwhelming urge to wring the other group members' necks, even after ten weeks of sharing a barrel as it plummeted down our waterfall. We don't hate each other, at least not openly. In fact, there's even talk of going out for a celebratory beverage after the product has been delivered. If nothing else, we're all richer in experience and friendship.

Our main takeaways from this harrowing experience are the aforementioned riches, together with the memories of the consequences of our questionable decisions, collective and otherwise. Had we implemented a more rigid delegation of duties, used a different development methodology, reconsidered our naming scheme, or made efforts to simplify our toolset, we might've had a less stressful, easier time working -- both individually and remotely. Keeping in mind the as-yet-unfinished status of this project, the author of this contemprospective still feels safe in asserting that we're all of us better software developers than we were three months ago.