CSE 115A - Introduction to Software Engineering

Elijah Hantman

Reflecting on the development process I have two main takeaways.

The most important thing I learned is the importance of rapid iteration and communication when it comes to getting things done quickly. The project I worked on was Executable Spreadsheet. The project was written in C and was based around two main applications. The first was a front end spreadsheet editor written in Ncurses. The second was a library for storing Spreadsheets in an optimized lookup format, and for taking a small program written in a custom language and executing it.

Over the course of development we ran into several core issues. The most impactful issue was a lack of time, this was caused by a combination of issues, first we had a lack of clarity on when the project began which caused us to start off behind. The second main issue was a lack of time to resolve dependencies between tasks. The very first task was setting up a build system, it was a single person task since the project is small enough that we don't need to come up with a large scale agreement, and that we just need something which works, which meant 4 people were waiting on 1 person to finish. Luckily Quincy Strange who wrote the Makefile was great and on top of things and got everything done as soon as he could. However later in the project we needed to write a Tokenizer before we could really settle on how the Parser should work, however the person who agreed to write the tokenizer was out of town and didn't finish until the next sprint. So we ended up delayed by several days.

There were also issues regarding code quality, early on I provided a utils file with custom logging and assertions, however the usage of these utilities was fairly spotty outside of my own code. Things improved as the project went on but even at the end people were still mixing in the standard library functions, calling exit with a failure code rather than our custom solution.

I think most of these issues could be resolved with one of two things. If we had additional time we could have resolved dependencies via coming up with agreed upon interface boundaries, as well as improved the quality of code by gaining time to refactor and rewrite. We could have also fixed things if more of the group was in close communication more of the time. Often we had group members who were difficult to contact due to work or family obligations, and at the beginning we didn't have any contact information and had to email everyone to start work which ended up taking the time that should have gone into sprint 1.

My second big takeaway is the importance of cumulative tooling. Over the course of this project I wrote three hashmaps which, in theory, work nearly identically. I also wrote something like 15 growable arrays. A lot of time could have been saved if I had prewritten macros for creating and using growable arrays and hashmaps. While this could have been solved using a third party dependency I still think that overall our product is improved for having few dependencies, almost every line of code compiled is useful and important for the task we are doing which is not necessarily true of third party code. Our build system also ended up being fairly simple in concept, being just a single makefile. Philosophically I think that having as few dependencies as possible comes with a kind of software longevity and independence which is valuable in and of itself. Things like google might disappear but as long as you have a computer which runs linux, and a compiler, you can compile and use our application. However, in order to regain some of the lost feature velocity that comes with making applications which are fundamentally composed of other libraries and technologies you need to develop your own libraries and technologies.

Some of the earlier mentioned issues such as depending on single people to write core components could be resolved by either making good implementations of useful components beforehand and saving them, or by custom tools which are faster to setup. For example, the Makefile is more then 350 lines of code however much of it is shuffling lists of source files around in order to get lists of dependencies. A custom build solution could reduce the amount of code required to build a project.

Overall, this project has been a great opportunity to work together with other programmers on a small but complicated piece of software. And it has reinforced the importance of several key values, namely communication and preexisting tooling.