Jingkang Zhang

Prof. P. N. Hilfinger

CS197

10 August 2020

Internship Report

During the summer of 2020, I worked as a Software Development Engineer Intern at Audible, with a focus on full-stack web development. Through this report, I intend to illustrate my experiences, describe my responsibilities, summarize what I have learned during this internship, and compare them to knowledge learned in school.

The project that I worked on was an internal-facing console designed to help engineers and business team with their monthly and quarterly routines of generating reports. More specifically, It is a one-stop-shop for viewing, managing, and tracking the up-to-date status of my team's Period Close activities. Finance team, On-Call Engineers and Support Engineers can all leverage this tool during closing periods to help the process go more smoothly, clarify priorities and responsibilities, and reduce unnecessary communication overhead.

There are different phases in my intern project. First, I went through the design of the project with my mentor and manager. This is an important step in that pinning down the need, designing the project, and having a development plan is crucial to industrial software development. Contrary to what we do in school for class projects, that we mostly do them for the sake of learning only, when engineers get onto a project that is funded by a company, real impacts need to

be made to realize value. Thus, while the company might still want to give engineers a university-like atmosphere of learning and steady self-improvement, in the end it will expect employees show value equal to what they are paid for - they need to ship products. The first step of guaranteeing the successful shipping of a valuable product is a well-calculated and carefully-justified plan. Towards this, I discussed with my manager and mentor extensively about the purpose, value, and priority of the project. "Why do we need this product in the first place? What pain points does it address? Who are our customers? How do we measure the success of the product?" These are the questions I needed to propose over and over to both myself and all the other relevant stakeholders in the project.

After pinning down a development plan, it's important to design the architecture and choose the tech stack with which I'd like to implement the project. In school, the tech stack is usually specified by the course staff. They even provide students with a large amount of skeleton code, which greatly abstracts away the underlying complexity to actually get the project working from scratch. With such babysitting, students develop the tendency to ignore the overheads in establishing a project by vastly underestimating the time and energy required to merely build the project to the point where people think of it as the start, before anything related to business logic is put in place. Without revealing too much about confidential contents, I would say I used a pretty standard set of Amazon internal tools to build my project. On the backend, I used Java with Spring, agreeing with my team's legacy code base. For database, I used AWS DynamoDB to store the data related to my own microservice. I also used AWS Lambda in Node.js for some utility scripts regarding database initialization and manipulation. On the frontend, I used JavaScript with React. Admittedly, a lot of my frustration in setting up the project came from the steep learning

curve of many of the Amazon internal tools - I found myself taking too long to get hold onto these tools, partly because I don't learn fast enough, and partly because the tools are not well documented and don't come with useful error messages. The lesson learned is that in industry, the world is no longer as ideal as in school; you no longer have best-practice-codebase by Professor Hilfinger, you no longer have TAs who care about doing things in the right way more than anything else, and you can no longer turn to course staff whenever something is broken. In the company, you need to learn to unblock yourself, you need to respect other engineers' time, and you need to step out of your comfort zone frequently.

Then it comes to development. The biggest challenge for a beginner engineer to develop in a corporate environment is that their efforts put in and the output don't match up. While in school, super complex things can be accomplished with so minimal effort because Prof. DeNero will have already written 80% of the code for you to train your AI or create your Ants VS SomeBees game, Prof. Hilfinger provides you with a thoughtful skeleton to a solid software engineering project which you can never write from scratch as an average sophomore, and 95% of code in PintOS is given to you. The false sense of accomplishment is that you think *you* did great and that *you* made the project possible. In company, it's exactly the opposite. The things that you are supposed to do are so simple! There are not many arcane CS concepts: no recursion, no textbook algorithms, no esoteric data structures. All you need to do is some CRUD operation in DB and "linking" together everything. The only complexity seems to come from the long call stacks - there are just too many layers of call frames for a human to make a clear sense of everything that happens. This is frustrating for a college student in that they find themselves bogged down by something that is far

from concept-wise difficult. It is the fact that things won't work out as expected and their lack of tribal knowledge that causes the imbalance in effort and result, making them feel unproductive.

Specifically, I find myself constantly stuck on company-adopted frameworks or boilerplate technologies. These technologies are often used to solve huge pain points which will arise when the scale of the problem will grow big. They weren't there in the beginning, so engineers *were* able to use their ad hoc approaches to write unorganized code. However, as they begin to encounter challenges never emerged before and suffer from tech debts in their careless code, they start to develop opinionated frameworks and boilerplates. This is a trade-off: while it's true that these frameworks can eliminate many problems in the long run, they pose substantial barriers to beginners whose intents are very simple, giving them steep learning curve, especially when the libraries are not well-documented, do not give useful error logs, or feature implicit behaviors. Some examples include Guice, the dependency injection tool, which gives implicit behaviors; the Amazon internal build and deployment system, which can be frustrating at times; etc.

Writing functional and elegant code is the last step. I did fine in this step because I have had previous internship experience and was accustomed to writing readable and maintainable code. Thus, other engineers didn't find substantial problems in my code during code reviews. Main difficulties in writing the code also come from lack of library-specific knowledge. There's no major surprise computer-science-wise: in the end, everything will make sense. It's only the process of making sense of the random errors that takes more effort than one would imagine. Implementation is always one of the easier steps, thus I'm also giving it shorter length in this report.

Above shall constitute my internship report for the summer of 2020 at Audible.