Saaif Ahmed
saaifza2

The Mythical Man Month: Essays on Software Engineering
Book Report

Intro to the Report: This report will be based on interesting chapters that directly relate to my experience as a software engineer. Not every chapter will be covered, nonetheless there will be a breadth of topics discussed.

Chapter 1: The Tar Pit

Summary: The book begins by introducing the concept of software engineering as a whole. Primarily it discusses the tall tale, or fairy tale that is what software engineering is perceived to be. The author brings up stories of developers building magnificent tools in their garages, and those developers moving on to become extremely rich and successful. Frederick Brooks however, breaks down this idea into its reality. The small programs that people are most likely able to create in their garages are not true software engineering projects. Large projects are built upon wide and varied input data, a generalized and expansive algorithm, and supported by an abundance of documentation. The author remarks that this is approximately 3 times the work as compared to a developer doing all of the above but for a single use case. Large projects are likely to end up in the tar pit, which is the namesake of the chapter.

Brooks mentions that an animal caught in a tar pit cannot escape and therefore relates that a software engineering project can also get caught in a tar pit. The goal of this book, therefore, is for software engineers to overcome these challenges that can put the project into the metaphorical tar pit. From management problems to development issues, there are many causes for a large project to be ensnared in the tar pit, but no single culprit. The reality of a software engineering project is also discussed. If attempting to market a system as a product it will end up costing 9 times more than a single program on its own. This is the truth of software engineering, as systems are much more complex orchestrations of programs that do so much more.

At the end the author then discusses the reasons why people become software engineers. Programming has distinct joys to it that attract many to the field. There is the joy of being an engineer and creatively building things. The joy of being an aid to others with an individual set of skills. Programming is often described as a series of puzzles, and that puzzle solving is somewhat fun to other people. Finally the author discusses the extreme challenges of software engineering, and not those discussed typically. Cognitively for humans, programming is a different beast. The author remarks that programming demands perfection, as nothing will work unless it is absolutely perfect. The programmer is a separate identity from the management who controls the access and resources. Lastly, the fun of programming is met by its equal in tedious hours of bug finding.

Experience: I work as a software engineer for McDonald's Corporation at the headquarters in Chicago. I have been there for about 1.5 years and I have come to realize many things that this book has reiterated and have learned other things from this book that my job did not. With regards to this chapter I would like to applaud the author on the breakdown of the software engineer fantasy that people seem to have regarding this profession. The world as it is today, is hardly a world where garage based development will get anyone anywhere. The world of software engineering today should concern itself with the development of systems that link many different small items together to make a large whole. At McDonald's I work as a backend developer for the flagship application. This application is called the McDonald's Global Mobile Application, or GMA for short. The first thing that new developers learn at McDonald's is that the app is a system of various different workflows, and tools that coalesce into something greater than the sum of its parts. In the backend we use an assortment of API Gateways,

Lambda functions, SQS queues, Dynamo DB tables, and much more in order to make GMA work. The mark of a good developer is one who can navigate through these different technologies, rather than being one who is able to develop single use case solutions. This is in line with the teachings of the book.

Through my experience I have not been in many scenarios where a project has been caught in the metaphorical tar pit. Because McDonald's is an extraordinarily large company, it is difficult for anything to get caught in development cycles. The company has been developing the engineering department for a while now and has not shown signs of slowing down. I should consider myself lucky that there is an abundance of resources, funds, and help for developers at the company.

Lastly, the ending remarks of the author were interesting. Working as a software engineer has made me realize that, at least at the level I am currently at, I am not typically doing the joyous tasks mentioned by the author. In reality, handling tickets that typically include investigation, and learning about a new technology are not fun. An application of knowledge is how I tend to retain any information at all. Without that I am hard pressed to say my work is fun. I don't think I picked up this career with the immense joy of solving problems, designing programs, or wanting my work to be limited by managerial resources. I became a software engineer because it was a fruitful career financially and I was able to learn it to a decent level. The book had made me realize that if I want my career to advance I will encounter difficult problems, and I had better have a personal justification for the challenges I will face.

Chapter 3: The Surgical Team

Summary: In this chapter the author attempts to describe what an effective and efficient software engineering team looks like. To begin, Brooks remarks that a high performing individual is 10 times more effective than a low performing individual. The author then goes on to describe what a small team looks like which they describe as a max of 10 people. Big teams are unproductive and unfocused as compared to small teams. However, big teams are required to make large projects, otherwise they take too long. The programs and data managed by the team should be owned by the team rather than any individual. The author then begins to describe what the ideal team, that he calls the surgical team, is. The roles in this structure included architect, advisor, tester, documenter, toolmaker, and admin. With everyone only knowing exactly what they need to know, this style can work really well.

Experience: The lessons in this chapter are present within the team design at McDonald's. I was on a team that had 12 members when I first joined the company. While I did well on that team it was not as efficient as it could have been. I was moved to a smaller team of 6 developers at that time and it was much better. The author's indication of high performers being 10 times productive is correct. While not exactly 10 times more productive, my senior engineers far exceeded my knowledge level by a factor of 10. They had years of more experience than I did and thus were leagues more effective.

Given the teams in McDonald's are not necessarily focused on doing everything at once the surgical design doesn't necessarily fit in. However it does make sense in theory. There are elements of the author's surgical team design throughout the organization as a whole rather than it being in an individual team. This is because teams have to work on large systems and many parts that are similar in theory but different when zoomed into. For example, an architect is not building the architecture for one service, they are building it for many services in one workflow. A backend team is not servicing one repository, they are supporting many different backend services that are built. This is why the author's idea is expanded into an organizational surgical team at McDonald's. The scale of the work that McDonald's handles is too large for it to be confined to one person. Like how the author mentions it would be impossible for any one person to be the architect, or tester, or documenter for the GMA app engineering teams.

Chapter 6: Passing the Word

Summary: The author mentions that a system when being designed should not have everything laid out at the beginning. Obviously there is a need for formal definitions and they should be used when applicable, but some room for flexibility should be maintained. During implementation this flexibility will ease the development process and make the whole project smoother.

Another crucial part of development is the communication during the design. During implementation and design people would want to propose changes and brainstorm solutions to problems. These can be solved by incorporating meetings into the schedule where people are focused on the creativity aspect over the decision making. The author notes that it is important for these meetings to be somewhat frequent such as weekly, and that the same people need to attend the meeting. This is to maintain context from meeting to meeting. During votes it is important to have a tie breaker which can be the chief architect.

Lastly the author remarks on the testing regiment that software engineering products should have. The author mentions that a test team should be included into the project. This team should be independent of the other developers or managers. The testing team should be a proxy for the customer and act as the customer would. The testing regiment should be incredibly harsh in order to maintain the thresholds and parameters set by the software engineering project.

Experience: While this chapter was short in content it has a lot of good information. With regards to the first topic of the chapter, the specifications of design being open, there is some relevance to the work of a software engineer. In McDonald's there are certainly liberties taken by the architects when designing the systems we use. The flexibility allows the engineers to make concessions where necessary or convenient while still reaching the overall goal set forth by the architects. This is exactly in line with the teachings of the author. Many times, I have reached out to my senior engineers who are vastly more familiar with the architecture of the specific backend system we work on. I have asked them if I needed to worry about certain elements, and they have stated that it can be left justifiably ambiguous and not contain a proper formal definition.

As far as communication during the design that the author mentions, there is something that can be related from the author's work to the day to day workings of a software engineer. Within McDonald's most teams work on the Agile development cycle. Within these sprints we have daily standups, also known as morning meetings, and we report on what we did the previous day. These allow engineers to be up to date on the inner workings of the sprint, but this still can lead to an information overload. With a decently large team of 8 developers each holding onto 2 tasks, there is a lot to be said about how much any one person can retain. To solve that problem we have backlog review sessions. As our team overlooks what will be worked on in the next sprint, there is an opportunity to discuss the direction of the team's work and propose changes to tasks. These changes can be marked down on the ticket and thus be as up to date for the sprint as possible. This system works in the exact manner as Brooks has described in this chapter.

For the testing methodology that the author has described in this chapter, McDonald's has taken that into consideration. At McDonald's while developers are able to test like a customer, there is no guarantee that we will fulfill that role efficiently. Instead we employ a quality assurance team for every market we service. The backend is mostly universal across each market, but the configurations that apply to the backend are not. These must be tested as different markets have different laws and regulations that McDonald's must adhere to. These testing teams being independent from the developers is a critical role to fill. They must act as a given market customer in order to truly test the GMA app. We have 2 dedicated environments for quality assurance testing. Developers in general cannot interact with these environments to ensure the testing is as harsh as possible.

Chapter 9: 10 Pounds in a 5 Pound Sack

Summary: This chapter covers the size problems related to developing software engineering projects. The definition of size in this context is primarily the size of the program in terms of bytes. The author dictates that the builders of a programming system should start off by setting size budgets. There should be a way to control the size, a size goal should be instantiated, and the developers should be aware of size reduction methodology. The importance of a performance simulator is crucial to catch problems that could force a redesign. If at the end of the project development cycle, there is a major performance failure, the programming project may require lots of changes wasting time and money.

Throughout the implementation of the software engineering project it is critical that a dialogue is open between architects and engineers. This is to ensure that the intended design is made and that the engineers do not lose focus on the broader picture. When optimizing a software project this situation can arise and it is best avoided. The representation of the data is another important part of the optimization of an engineering project. The author makes an interesting statement, that from the analysis of the data structures a flowchart would not be needed. It is from the structure of the data that the author states more information can be retrieved. Brooks then goes on to describe a project that recognized the patterns in the data. For a console interpreter for an IBM 650 the developer made an interpreter for the interpreter. This saves on space for the program and is an example of how representation is the essence of programming. The interpreter recognized the common patterns and overcame the slow human interaction component.

The last topic discussed in this chapter is the concept of scalability and the preparations for that. Scalable systems are what large software systems are designed to accommodate. However, the author notes that scalability is only possible within a limit. In most cases the better idea is to design a large system straight from the beginning. With that mentality in mind it is important for managers to get their workers trained. The knowledge transfer of time-space saving methods should be common.

Experience: I think this chapter is interesting because it is both a relic of a time long surpassed by modern technology, yet still maintains principles about the external costs to software engineering. In the modern era, the size of a given program is not really something developers take into consideration. For an individual, doubling their storage on their devices is nothing but an online order away. Developers of large studios release games and software that are hundreds of gigabytes regularly. There is no need for optimization around the storage systems of computers. For roughly $60 one can procure another terabyte of storage which reduces the need for developers to save on size. In my line of work, we hardly care about the size of the code we write. Any additional file is roughly a dozen kilobytes which makes our large apps a couple hundred megabytes at absolute maximum. These sizes are inconsequential to the everyday user who has access to roughly 256 gigabytes of storage at any given time. The very literal message of this chapter is outdated, however there are other teachings from the author that are very useful.

While developers don't necessarily need to worry about hardware limitations for the most part, there is something to be said about the resources developers may take for granted. For example, when building web applications or anything using cloud based resources the price associated with these items must be accounted for. Professional level resources in AWS for any company can reach the tens of thousands easily. For McDonald's, each development environment costs roughly $30,000 USD to maintain per month. This is because we have millions of requests every day and our usage is at an astronomically large number. Therefore at McDonald's we have to make precautions when developing. This is in line with the remarks of the author within this chapter. The unnecessary costs, even for the sake of development, should be avoided. Developers are not typically allowed to generate new AWS resources or Kubernetes pods because they incur costs that other quality assurance teams may need for

testing. Our testing is extremely rigorous as we have 3 different testing environments before anything goes live in production.

One of those testing environments is the performance environment. In this chapter the author notes that performance testing must be conducted as the implementation is occurring. This is due to the fact that performance failures, unlike most bugs, will almost surely cause a redesign, and these waste time and money. Therefore at McDonald's we have performance testing environments to ensure that our code is fast enough and can handle exceedingly large volumes. Just like how the author mentioned, we do not want to redesign anything late into the development cycle, therefore before even getting to quality assurance our code has to meet rigorous performance standards. The environment is set up to reproduce 10 times the expected load of input and output and the applications have to have a 99% success rate within our thresholds. We can ensure our apps are ready for anything by doing this.

The last topic discussed by the author is scalability of developing projects. Throughout the past 10 to 15 years many different coding standards have emerged. As the minimum scale for a project has increased, so too have the standards been adjusted to accommodate this fact. In the day to day I find myself working and developing in a manner that is already conducive to large scale design, and I feel that most of the work any developer would do nowadays is much the same. Due to the fact that we no longer need to worry about size limitations, the algorithms we use are optimized to be as efficient as possible by utilizing a lot more memory. However, the author is correct in the philosophy of design. A large system should be designed to be large from the start, rather than trying to scale up a smaller one. At McDonald's we definitely take ideas and principles from smaller ideas but the GMA app was designed to be the large system that it is from the start. There is no other way to go about it.

Chapter 15: The Other Face

Summary: The main topic of this chapter is documentation and the author highlights the importance of it, in addition to the different types of documentation that are necessary. Prior to even talking about what documentation should be, Brooks starts by encouraging the teaching of documentation to new programmers. As mentioned by the author, documentation needs to begin with a high level conceptual overview. One method of doing this is akin to the black box style of design, where inputs and outputs are highlighted in addition to the purpose of the program.

The author remarks on the 3 different types of users for whom documentation should be written for and in their different manners. These are the casual users, users who depend on the program, and those who will adjust and make modifications to the software. Beyond writing narrative prose on the inputs and outputs of a program, for users that depend on a program there must be documentation on the testing procedures, and test cases such that the correct behavior can be verified. Flowcharts are a useful item, as mentioned by the author but they are double-edged. Lengthy and complicated flowcharts are not helpful for most developers. The author remarks that most programs do not need a flowchart, much less those that exceed one page in length. Therefore, Brooks suggests that the flowchart design be limited to one page.

The maintenance of documentation is also highlighted and touched on in this chapter. Syncing issues arise when the documentation is made outside of the code. If the code base is worked upon frequently the documentation also has to be intentionally updated to accommodate that change. Documentation should be included in the code where possible. Rigid coding standards however lead people to over-document and sometimes in the wrong locations. A great detail of a low level idea for a program is not as useful as an extremely detailed high level write up on a program. Lastly the code should be documented while it is being created, rather than afterwards. The author mentions that machines are made for people and not the opposite. From this we can deduce that the documentation

should be written for a person to read and understand how to use the tool that is the software or software system.

Experience: I have a lot of experience with the concepts presented in this chapter. My current position at McDonald's is my first foray into the industry of software engineering. The overarching goal of the work we do at McDonald's is to transfer the outsourced work done by consulting companies and extraneous engineering firms into in-house McDonald's engineering teams. From the very first day I started, I have heard an abundance of laments from my colleagues on the abysmal work done by the other companies. Because McDonald's had no real engineering's team as these tools and systems were being developed, there were no standards on the documentation principles. We have spent a lot of man hours trying to search through and understand what documentation is present. Afterwards when testing the code we received, the documentation does not tend to line up exactly to what is documented. Not to mention that much of the code itself does not have readable, usable documentation. The teachings within this chapter would have saved a great amount of time and money if the older teams knew of them.

Documentation is a very interesting thing at McDonald's. Due to the problems that I have mentioned many teams have started taking their own approach to documentation. Because software engineering teams at McDonald's work on the Agile development cycle and use JIRA software to manage the sprints there is a defined term for documentation tickets. They are called Spike tickets, and working on spike tickets are generally well defined and useful. Brooks mentioned in this chapter that code should have documentation written within it, but in reality there are many items that cannot account for this. While we can do the bare minimum of good coding practices, such as readable and coherent names, the vast majority of systems that a modern developer could interact with cannot hold documentation. Take AWS cloud services, there is no way to access that code base of functions. In the Terraform scripts that generate the resources there is a way to write some prose and descriptions. But the detailed workings are still left to AWS to handle. That is where the Spike tickets come in. For a given Spike ticket, assuming it is well written, relays the exact information that the team needs to know about a given technology, functionality, or workflow. We can estimate in advance how much effort we want to put in the research so that nothing is over-documented or under-documented. Across teams at McDonald's however there is something to mention about the lack of overall cohesion in the documentation as each team does it a bit differently. However using organization charts, we can determine easily who to ask on a given topic and have them point us in the right direction. Collaboration is an encouraged part of the culture at McDonald's and has made the overall transfer to in house operations much more bearable.

That isn't to say that the author is mistaken everywhere, far from it. As a new developer to McDonald's I was not familiar with the tools for documentation that everyone else was using. Therefore, I began to search through comments of the code in order to understand what I could. But, I was disappointed at the very minimal usage of comments in the different repositories. The author mentions that rigid coding standards often drive over documentation. I would add that it can lead to inefficient documentation. The objective of good code is to make it as readable as possible, but I believe that developers must concede that some operations are not immediately readable. Testable code is broken up into many blocks that span multiple files, and it gets very difficult very quickly to keep up with a logical flow in a program. No amount of good naming conventions and code structure will solve that problem. Instead, developers should write a simple comment that explains what's happening at each step of the method. This makes it much easier to follow along as new developers are drawn to comments for insight. This idea is described by the author and that idea of documenting alongside the code is something I have tried to incorporate into my team's work at McDonald's.