Summary: The Mythical Man-Month

Cory Nezin

February 18, 2018

1 The Author

The author of "The Mythical Man-Month" is Frederick P. Brooks. Brooks worked as a project manager for IBM and is known as the father of the "IBM system/360" which is a family of mainframe computers produced between 1965 and 1978. Brooks founded the Department of Computer Science at the University of North Carolina, Chapel Hill, where he currently teaches computer architecture, molecular graphics, and virtual environments. Since originally authoring the book in 1975, Brooks chaired the Defense Sicence Board, where his experience was later incorporated into a new edition of the book.

2 Introduction

> "In many ways, managing a large computer programming project is like managing any other large undertaking–in more ways than most programmers believe. But in many other ways it is different–more ways than most professional managers expect."

As Brooks says, management of software projects is very similar to management of any other project, but not so similar that management decisions should be identical. Therefore the book, and this summary will not focus on the aspects of management that are similar, but rather those that are unique to the field.

The book is not scientific in nature, as it simply reflects the personal veiws of the author, which were developed during his time at IBM. Brooks explains that the main project he worked on, OS/360, had both successes and failures in terms of its final design. The product was late, bloated, and much more expensive than estimated. After several iterations, however, it was reliable, efficient, and versatile.

The author says that the book is "a belated answer to Tom Watson's probing questions as to why programming is hard to manage." The book does not contain the knowledge and experience of Brooks alone, but also the thoughts and ideas from several other managers in the field with whom Brooks has conversed with.

Brooks compares Large-system programming to a tar pit. The harder you struggle, and the more resources you devote to getting out, the more you are enveloped. It seems counterintuitive that some of most important work, for example the origins of Apple and Microsoft, were completed in garages by small teams. However, this is a common theme throughout the book, that efficiency decreases with man power. Yet large teams dedicated

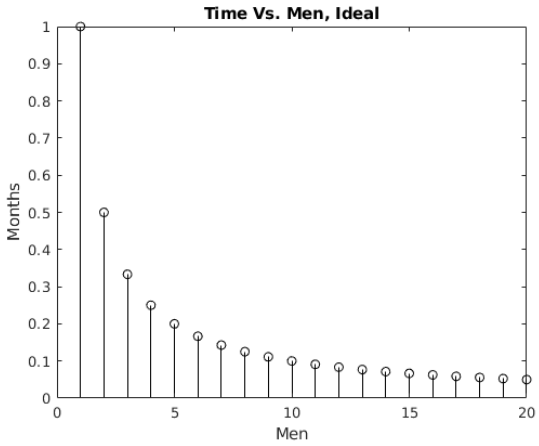

Figure 1: Time to produce a given quantity versus number of men, perfectly partitionable. Plot adapted from "The Mythical Man-Month"

to increasingly large products are necessary because there is a giant cost differential between small, important programs that solve some specific landmark problem, and large general programs which work across the board for many consumers.

3 The Mythical Man-Month

The titular chapter explains the guiding principle, and most difficult trap of large-system programming. the man-month is a unit of measurement for productivity. In perfectly partitionable tasks, like reaping crops, one can estimate that the quantity produced is the number of men times the number of months they work times some constant. That is, $Quantity(Units) = N(Men) \times M(Months) \times C\frac{Units}{Man-Months}$, or efficiency remains constant with an increasing number of men. A graph demonstrating this principle is shown in figure 1 In some cases like assembling cars on a factory line, we may even expect efficiency to increase with number of men because of specialization.

On the opposite end of the spectrum some tasks are not partionable at all, e.g. the bearing of a child will take nine months no matter how many workers are assigned. This type of task follows the formula $Quantity(Units) = M(months) \times C\frac{Units}{Month}$ A graph demonstrating this principle is shown in figure 2.

Finally, for complex tasks like large-system programming, the heavy burden of intercommunication is added. If we assume that every worker must communicate with every other worker in order to complete their tasks, the time cost grows quadratically with the number of men, or $M(months) = C\frac{Man-Months}{Unit} \times \frac{Quanitity(Units)}{N(men)} + B\frac{Months}{Men^2} \times (N(Men))^2$ A graph demonstrating this principle is shown in figure 3

Brooks suggests the following rule of thumb for completing a software task:

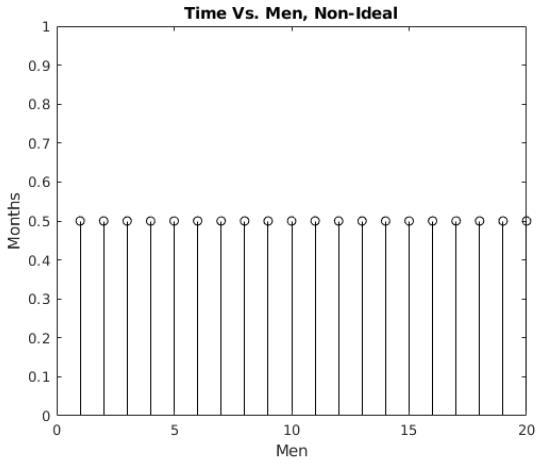

Figure 2: Time to produce a given quantity versus number of men, unpartitionable. Plot adapted from "The Mythical Man-Month"

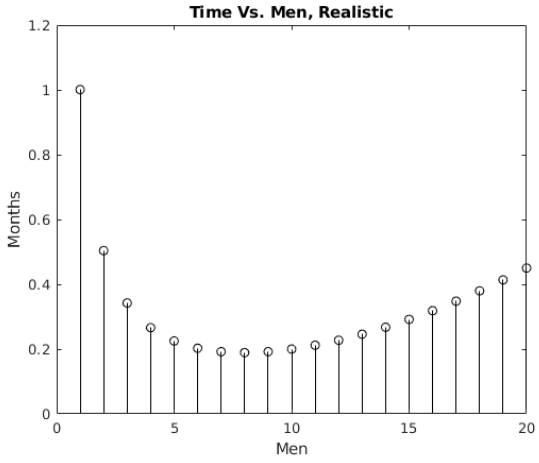

Figure 3: Time to produce a given quantity versus number of men, semi-partitionable. Plot adapted from "The Mythical Man-Month"

1/4 planning

1/6 coding

1/4 component test and early system test

1/4 system test, all components in hand

Brooks claims that, compared to conventional scheduling, more time is devoted to planning, much more time is devoted to testing, and much less time is devoted to actual coding, or implementation. Brooks says that failure to allow enough time for system testing is particularly disasterous since it is at the end of the schedule, and the software mmay be crucial in supporting other business efforts like shipping new computers. All of this gives rise to the infamous Brooks's Law: "Adding manpower to a late software project makes it later."

4 The Surgical Team

Given the observation that efficiency decreases with the number of workers on a team, large system design may seem hopeless. One proposed solution is the idea of a surgical team. The proposed team is surgical in two ways: it is designed to tackle a particular issue and it is composed of a small amount of people who each have a particular role. The motivation for such a team is not only a result of Brooks's Law, but also the insight of a study by Sackman, Erikson, and Grant. It was found that on an average programming team, the average ratio between the highest and lowest productivity between members was 10:1, an extraordinarily large number. Moreover, the program quality (measured in terms of speed and resource usage) was on average 5:1. Therefore it seams, programming work should not be distribued equally. Harlan Mills proposed the surgical team as a solution, Brooks created the following 9 roles to describe one possible realization.

The Surgeon
Also known as the chief programmer, he designs the program, codes it, tests it, and writes the documentation. He should have great talent and at least ten years of experience.

The Copilot
The copilot is able to do anything that the surgeon can do, but is less experienced and therefore less efficient. He may on occasion write code, but is primarily useful in thinking of new general strategies, representing his team to others, and giving advice to the surgeon.

The Administrator
The surgeon has final say on all oersonnel, budget, and space issues however he cannot afford to spend much time on these. Thus the administrator is necessary to handle and track these issues, and provide necessary information to the surgeon strictly when needed.

The Editor
The editor is strictly in charge of documentation, taking the surgeon's first draft, editing it, and producing a final standalone product.

The Secretaries
Both the administrator and the surgeon each need a secretary for handling day to day activities, scheduling, and non-product files.

The Program Clerk
The program clerk is in charge of maintaining technical records including documentation, code modules, and test outputs and evaluations. This member is in charge of making all computer runs visible to all teams members in a useful way.

The Toolsmith
The surgeon requires a set of tools to do his core work, programming. There are many software libraries built for building more software including tasks like version control, text editing, and interactive debugging. The toolsmith is in charge of assuring that all of these functionalities are provided reliably.

The Tester
The tester is an extremely important role, given that half of the recommended schedule is devoted to testing. The tester is in charge of generating tests and generating testing code so tests can be run quickly and systematically.

The Language Lawyer
The language laywer is in charge of having very specialized and deep knowledge of languages. He is useful for finding very efficient ways of performing a specific task, or hacks to perform obscure actions. He may serve multiple teams.

5 Software Specific Techniques

Brooks lays forward several specific schemes for testing and developing software programs. Many of these schemes are specific to software – some are made easier by being in a software environment, and some are necessary only because we are in a software environemnt.

Brooks says that the most subtle bugs "are system bugs arising from mismatched assumptions made by the authors of various components". That is, if one tries to separate a task into smaller portions, they must make sure that the portions match at the boundary. Therefore careful attention must be paid by managers when defining what each component is, what its inputs and outputs are, how fast it must run, and how much space it must use. This can be ensured by "testing the specification", handing the specification to an outside group of people who won't be developing it, and having them audit it. Brooks claims that developers will happily fill in the gaps of the specification rather than pointing them out.

Top-down design is the widely used practice of taking an abstract construct and breaking it down into smaller and smaller pieces. This is already popular in all fields of engineering, however it is especially useful and necessary in software engineering because of customizability. In other fields of engineering, there is often the constraint of using already existing chips

and materials simply because it would cost too much to manufacture them. Therefore bottom up engineering is sometimes more useful. However in software, it is very easy to make small components, or modify already existing components to fit your purpose, espcially since their is no physical cost associated with it.

Brooks also says to "build plenty of scaffolding", or code which is not intended to be in the final product, but help future debugging. He says "it is not unreasonable for there to be half as much code in scaffolding as there is in product. For example, dummy components which reads and tests input data format and returns meaningless, but correctly formatted results.

6 Unified System Design

Just as a painting would not look right if painted by multiply artists, a system would not work right if designed by multiple architects. Brooks claims that "conceptual integrity" is the most important consideration in system design, and that it is better for a system to reflect on set of design ideas rather than many good, uncoordinated ideas.

The ultimate test of system design, Brooks says, is the ratio of function to conceptual complexity. Function means being able to do complicated things quickly, conceptual complexity is how difficult it is to master. Neither high function nor low complexity is enough by themselves.

There are two fundamental levels to any system, the architecture and implementation. A careful separation of these two things will aid in the creation of a unified design while allowing for partitioning. While the architecture must be created by one person, or a small group of agreeable people, implementation is more free, and so it can be separated into large groups. Architecture says what happens while implementation says how it happens. It is important that architecture is consistent because it is "user facing" meaning it is the only part that the consumer interacts with. The implementation may change vastly while the architecture remains the same at no cost to the user.

Communication, as in any organization, plays an extremely important role for software engineers. It is required not only for development and communication between teams, but for communication to the user in the form of a manual. Because manuals must be precise, formal definitions and feature descriptions are required to avoid confusion. One must be careful however to maintain separation of architecture and implementation – while stating the implementation of a feature would be a valid definition, this would cross the boundary and therefore be uncalled for even being more precise.

Meetings are of course necessary. Brooks suggests weekly half-day meetings where representatives of the hardware and software implementers, as well as market planners can propose problems or changes, and discuss possible solutions. The other type of meeting suggested is a yearly two-week long meeting designed to empty the backlog of issues which where initially rejected at the weekly meetings but have caused issues since.