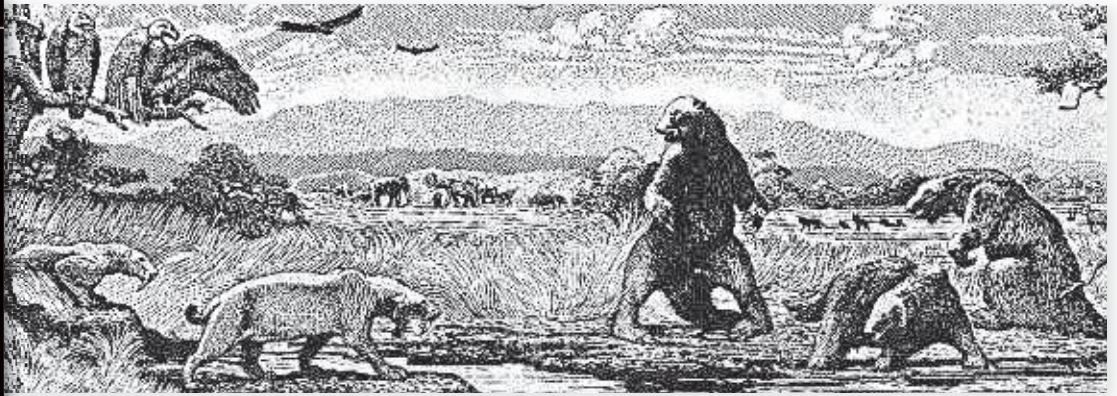
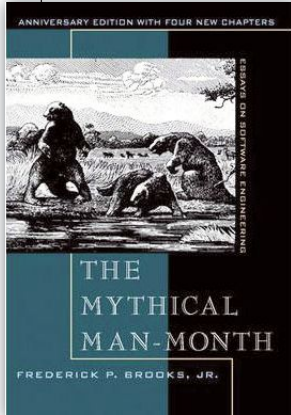


# BOOKreview

by John Baldwin



## *The Mythical Man-Month* and Its Relevance to the FreeBSD Project

### *The Mythical Man-Month*

.....By Frederick P. Brooks Jr.

**Publisher** .....Addison-Wesley Professional;  
Anniversary edition (1995)

**Print List Price** .....\$42.99

**Digital List Price** .....\$34.99

**ISBN-10** .....0201835959

**ISBN-13** .....978-020185953

**Pages** .....336

**T**he *Mythical Man-Month* is a classic work on software development. It is structured as a series of essays that use anecdotal evidence as the basis for practical advice. Rather than dealing with specific technical details, these essays focus on how people develop software.

I first read *TMM* (as it is otherwise known) less than a year removed from college graduation on the recommendation of FreeBSD developers. While some aspects of programming practice have changed since *TMM* was originally written in 1975 (such as widespread use of higher-level programming languages and interactive debugging), many of the concerns the author, Fred Brooks, notes are still relevant today.

One of the first topics Brooks broaches is distinguishing a simple computer program from a product. A developer who bangs out a program to accomplish a task for himself or herself has written a computer program. A product, on the other hand, must be usable by other people. It must handle arbitrary inputs (including erroneous inputs) correctly. It must have documentation that explains both how to use the program and which

tasks a program is suitable (or not suitable) for. Developing a product requires far more effort than the time to implement the initial computer program. In FreeBSD, programs should have a manual page at a minimum. However, supplemental documentation in the Handbook is often required (and not always present).

Similarly, designing a computer program to be a part of a computer system requires significant work. Programs belonging to a system should provide consistent interfaces to users. They should also provide consistent methods for cooperation. One of the ways UNIX-like systems achieve this is through the use of pipes to join the input and output streams of processes. However, there are several other ways to achieve consistency, such as reusing the same option letters for common command line flags or using consistent language in manual pages.

Chapter 2 discusses the scheduling of software projects. In particular, it seeks to destroy the notion that adding developers to a software project increases productivity (and reduces schedule length) linearly. Instead, Brooks highlights the communication overhead imposed when adding developers to a project as well as the additional time required for training. Brooks also notes that many software projects contain a critical path of tasks that must be completed in sequence rather than in parallel. These hard truths are used to warn against overly optimistic responses to schedule slippage, and in particular, the knee-jerk reaction of adding additional developers to a software project that is behind schedule.

FreeBSD releases do not have a combination of a hard deadline along with a required set of fea-

tures in quite the same way Brooks's examples imagine. However, several of Brooks's observations are still applicable. Even with the ability to drop features and focus on just shipping a release on a given date, FreeBSD releases (especially the X.0 releases on new branches) are often delayed. These delays usually stem from critical path tasks such as waiting for a pending security advisory scheduled for release near the time of the release. FreeBSD also contains a large developer base, which results in a large portion of time spent on communication.

Chapter 3 proposes a model of organizing developers into teams similar to the organization of teams used for medical surgeries. Each team consists of a single "surgeon" who is tasked with writing the code for the end product as well as additional team members to assist the surgeon. FreeBSD developers are not generally organized in teams as proposed by Brooks. However, Brooks's model does highlight tasks that are important to development beyond simply writing code. Two of the team members he calls for are dedicated to tools and testing.

The FreeBSD community has expended a significant level of effort on improving the tools available on FreeBSD in the past decade. These include adopting more modern replacements for toolchain components such as Clang, as well as entirely new tools such as DTrace. This work continues with improvements to debuggers and performance analysis utilities.

FreeBSD developers have also increased the testing coverage. For several years, FreeBSD's source tree included a hodgepodge of tests that were not run regularly. Some tests used common frameworks, but many other tests were stand-alone. Through the work of several developers, FreeBSD adopted the kyua testing framework and first shipped a set of kyua-based tests in 10.1-RELEASE. Many existing tests from NetBSD were imported as part of the process, and existing tests were converted to the new framework. These tests can now be easily run as automated jobs. There is still a lot of work to do on this front, not only with adding tests to fill in gaps in existing test suites, but new classes of tests such as tests of the installer.

Chapter 4 highlights the importance of conceptual integrity and offers a plan for achieving it in large systems. Brooks asserts that "conceptual integrity is the most important consideration in system design." He states that the only way to achieve this is to limit the number of minds involved in the design. Brooks proposes splitting architecture from implementation. This concentrates the design in the minds of the architects

while spreading the workload of implementation across a larger pool of developers. FreeBSD does not place a sharp distinction between these roles and does not have any formal, system-wide architects. Informally, FreeBSD developers do seek review from peers, and there are individuals whose review is sought for changes to specific portions of the system. In some cases, these individuals are de-facto architects for those portions of the system, but not formally named. FreeBSD has attempted to make this process more formal in the past, but these attempts have not succeeded.

The Second-System Effect is the topic of Chapter 5. Here, Brooks documents developers' tendencies to pare down features in the first version of a system only to swing to the opposite extreme in the second system. Having successfully completed the initial version, the developer then proceeds to add every conceivable feature to version 2.0. Brooks states this system "is the most dangerous system a man ever designs." The result is a bloated system with poor conceptual integrity. FreeBSD is not a perfect system and certainly has had portions of the system that have suffered from this tendency. Our developers are continually learning from each other as well as from our own past mistakes. Sometimes our peer review permits us to fix mistakes sooner rather than later, but not always. However, FreeBSD's developer community provides a place where developers can be mentored and warned about trends such as the Second-System Effect.

Chapters 6 and 7 highlight the importance of effective communication and organization as requirements for effective development involving a group of individuals. FreeBSD developers and community members communicate over various venues including email, web forums, and real-time chat such as IRC. However, our community also realizes the value of in-person communications and places a premium on "face time" at conferences, developer summits, and user group meetings. The community actively participates in these meetings not just as attendees, but as organizers, speakers, and volunteers. Organization remains a challenge for FreeBSD. The FreeBSD Project's formal membership is very developer-heavy, but is in need of a variety of skill sets apart from pure coding. This is not to say that the Project is completely disorganized, but there are certainly gaps in the coverage of many of the non-coding tasks the Project requires for maximum effectiveness.

Estimating the time and resources needed for systems projects is the subject of Chapter 8. Brooks cites results from various studies analyzing some of the factors that affect scheduling of systems projects. One of the first studies highlights


the fact that a significant portion of programmers' working days are lost to factors such as meetings, working on tasks for other projects such as a high-priority bugfix, or hardware failures. Other studies note that the lines of code produced over time vary with the complexity of the program. Of the projects surveyed, operating systems were the most complex (and thus the slowest to develop).

Chapter 9 deals with managing memory allocation between portions of a large program. From a technical standpoint, much of this chapter has been made obsolete by virtual memory. However, the chapter still contains a few nuggets. First, Brooks relates an anecdote from the OS/360 development that resulted in the system performing very poorly. Each module was assigned a size constraint and the developers of each module resulted to workarounds such as overlays and "borrowing" space from neighboring modules without considering the effect on the system at large. While each module functioned in isolation within its assigned space, the system as a whole suffered from excessive disk

and apply their own design requirements to FreeBSD. In addition, the Project actively seeks feedback from our customers to determine their requirements. While we are not always able to satisfy every requirement, we do use these to set Project direction. Vendor summits are one of the tools the Project uses to engage with our customers. These are often held in conjunction with an existing conference or developer summit. If you are a FreeBSD customer looking to engage with the Project, please contact the author or another FreeBSD developer. We want to hear from you.

Chapter 11 focuses on change. Requirements of a software product change over time as does the hardware software runs on. Brooks encourages developers to embrace change and plan for it rather than fighting against it. For new software projects (or even new subsystems), one often needs to build an initial prototype to better understand the problem being solved. Just as deriving specifications forces consideration of several design decisions, implementing a prototype also uncovers a host of unforeseen issues. Trouble arises when one assumes that this initial version must ship as the final product rather than having the freedom to re-architect the design of the actual product. Just as it is important to plan for change in the design of software, it is also important to plan for design in the organization building the software. For FreeBSD, this means building and maintaining a community, not just source code. Over its lifetime, the FreeBSD Project has benefited from the contributions of many individuals. Often individuals move on to other interests or tasks after working on FreeBSD for a period of time. FreeBSD has survived these fluctuations in our community and has continued to grow. However, we must continue to actively welcome and recruit new individuals to our community.

The importance of tools is the subject of Chapter 12. While the tools-smith from Chapter 3 is mentioned, this chapter focuses on other topics than programming tools. In particular, Brooks discusses the trade-offs of sharing access to new hardware among multiple developers, the importance of simulators relative to actual hardware, and the benefits of interactive debugging. Interactive debugging is something that developers now take for granted, but the other two topics remain prescient today. Bringing up FreeBSD on the arm64 platform has encountered constraints with target machine scheduling due to the limited number of machines available. Simulators such as QEMU also permit developers to test FreeBSD on a broader range



**Just as it is important to plan for change in the design of software, it is also important to plan for design in the organization building the software. For FreeBSD, this means building and maintaining a community, not just source code.**

I/O to satisfy overlay requests. Among several lessons from this story is the point that one must keep system-wide effects in mind and not focus exclusively on one module. A second point made in this chapter is the importance of data representation on performance. The data representation often drives the algorithms constraining the range of performance. In those cases, large performance changes will come through altering the data representation rather than optimizing the existing code flow.

In Chapter 10, Brooks discusses the importance of formal documents and their use in setting down design. In particular, deriving specifications requires one to consider a host of design decisions that would otherwise not be realized until deep in the bowels of a coding session. This is certainly an area where the FreeBSD Project is not very strong. As an open-source project, FreeBSD's developers are not handed a set of specifications from external customers to build a design against. In many cases, FreeBSD's developers are themselves FreeBSD consumers

of platforms from the comfort of their desktops.

Brooks next turns to system debugging in Chapter 13. The chapter opens with an emphasis on architecture and design. Skimping on the planning and design stages sows the seeds of an unstable and buggy system. Brooks favors a top-down design approach using iterative refinement to bring detail to the design. Much of the chapter warns against various shortcuts during system test (now referred to as integration testing). First, test and debug individual components in isolation rather than attempt to test multiple components together. The first approach does require more “scaffolding,” but in the latter approach, bugs in components can interact in surprising ways resulting in breakage that is much harder to debug. In some cases, multiple bugs across components may cancel out each other in a larger test, giving a false sense of correctness. Second, don’t skimp on the extra code and tools that are needed for testing, but not in the final product. These enable more detailed component testing which saves time and frustration later in integration testing. Third, maintain a standard copy of the current system that new and in-progress components are tested against. Changes to this system should be rolled out periodically in a schedule that provides periods of stability for in-progress components to be tested against. Finally, new components should be added to the system one at a time, and the new system should run a full complement of regression tests as each component is added.

While Chapter 8 covered the topic of planning and estimating for a software project, Chapter 14 focuses on managing a project once it is in progress. Brooks’s first recommendation is to have a schedule that contains concrete milestones. Vague milestones result in ambiguous communication between layers of management (or between managers and developers). Brooks also warns against micromanaging. When managers jump in to correct issues the managees are capable of solving, the managees resort to hiding reports of issues as a means of self-defense. To foster clear communication, a manager must be willing to accept bad news without immediately barking orders.

Chapter 15 addresses the subject of program documentation. Brooks begins by noting three levels of documentation required for a program: how to use a program, how to test a program, and how to modify a program. Brooks states that one of the root problems at the time of writing was that documentation was kept separate from source and inevitably became stale. He proposes self-documenting source code as the most viable

solution to this problem. Self-documenting source is now widely practiced, and in particular, the use of high-level languages has made this more reasonable. In-source documentation systems such as Javadoc and Doxygen have been used by many software projects to store documentation alongside the source that can be easily translated into a more human-readable format. While certain portions of FreeBSD’s kernel source do use Doxygen comments, most of FreeBSD’s documentation for the first two levels is stored separately from the source. Some documents are not well suited to storing in source, but API descriptions may be better served by a system like Doxygen rather than stand-alone manual pages.

One of the documentation topics Brooks mentioned while describing the last level of documentation was to explain “why” design decisions were made. While these decisions are sometimes discussed in comments in the source code, these decisions are often explained in the source code repository logs of modern software projects. FreeBSD (and BSD) have a long history of using source code control, and the FreeBSD Project culture expects and encourages thoughtful log messages that explain the “why” of changes, not just the “what.”

The 20th-anniversary edition of *TMM* includes an essay titled “No Silver Bullet” as Chapter 16. The essay was originally published in 1987 and discusses software productivity. In particular, Brooks asserts that in the following decade (1987–1996), no “silver bullet” would arise to improve the productivity of software developers by an order of magnitude. Brooks begins by dividing the difficulties of software development into two classes: essence and accidents. The essence of software is its abstract design. It includes the representation of data and algorithms used. Accidents deal with difficulties in expressing this abstract design. These can include the limitations of hardware and the languages and forms used to express an idea.

Brooks claims that the three large steps in improving software productivity prior to 1987 all attached accidental difficulties. High-level languages permit programmers to more concisely express concepts while deferring many of the mundane details to the compiler. Time-sharing systems facilitate quick turn-around time during development, allowing developers to sustain concentration for long periods of time. Unified programming environments provide a standard way to connect existing programs together that can be used to solve larger tasks, such as using an I/O pipeline in UNIX.

The essence of software development remains hard. Software is complex. Brooks notes that in many other objects people create, repeated elements are common, but in software, a repeated element is consolidated into a subroutine. Newer versions of software projects are not formed solely by duplicating existing modules. Instead, they contain entirely new components that must interact with the existing modules. At the same time, software is invisible and its structure defies detailed visualizations. When visualizations are used, they are forced to constrain the information, by focusing on control-flow rather than data-flow, for example.

Brooks asserts that difficulties arising from this essence are inherent in software. He did not see any techniques proposed or in practice in 1987 that would attack these difficulties in a substantial way over the ensuing decade.

In Chapter 17 Brooks responds to some of the criticisms of “No Silver Bullet” and critiques additional silver bullet candidates nine years after the publication of the original paper.

To encourage more vigorous debate of the various propositions in the original *TMM*, Chapter 18 includes a bulleted list.

Finally, in Chapter 19 Brooks revisits the con-

tent of the first 15 chapters 20 years after they were first written. In some ways, this is the most interesting chapter of the book. Brooks notes topics made obsolete by technological advances while affirming the propositions he believes are still valid. Certainly the power of *The Mythical Man-Month* stems from its continued relevance decades after it was first written. In part, this is due to the essence of software described in “No Silver Bullet.” Technological advances have not altered the fundamental construct of software. Secondly, *TMM* is largely about people and the interactions of people who work with software. While new technology does affect our lifestyle and the “accidents” of our task, people are still people. ●

---

**JOHN BALDWIN** ([jhb@freebsd.org](mailto:jhb@freebsd.org)) joined the FreeBSD Project as a committer in 1999. He has worked in several areas of the system including SMP infrastructure, the network stack, virtual memory, and device driver support. John has served on the Core and Release Engineering teams and organizes an annual FreeBSD developer summit each spring.

# freeBSD JOURNAL™

## DID YOU MISS?

- Nov/Dec 2015 / **Olivier Cochard-Labbé**, **The BSD Router Project**
- Jan/Feb 2016 / **Peter Holm**, **Using Fuzzy Testing to Build Industrial-Strength Systems** • March/April 2016 / **Brooks Davis**, **Cheri**
- May/June 2016 / **Andy Waafa**, **ARMv8** • July/Aug 2016 / **Chris Johns et al**, **FreeBSD and RTEMS** • July/Aug 2016 / **Michael Lucas**, **Tuning ZFS**

## Get caught up today

Order Back Issues @ [www.freebsd.foundation.org/journal](http://www.freebsd.foundation.org/journal)