原版风暴·软件工程系列

PEARSON
Addison Wesley

人月神话（影印版）

The Mythical Man-Month

（美）弗雷德里克·布鲁克斯（Frederick P. Brooks Jr.） 著

· 20 余年畅销不衰
· 软件开发和管理人员必读经典
· 作者 1999 年荣获美国计算机领域最高奖图灵奖
· 原汁原味，零距离领悟大师思想精髓

Pearson Education

中国电力出版社
www.infopower.com.cn

原版风暴·软件工程系列

人月神话（影印版）

The Mythical Man-Month

（美）弗雷德里克·布鲁克斯（Frederick P. Brooks Jr.） 著

中国电力出版社

图书在版编目（CIP）数据
人月神话/（美）布鲁克斯 著.—影印本.—北京：中国电力出版社，2003
（原版风暴·软件工程系列）
ISBN 7-5083-1303-8
Ⅰ.人… Ⅱ.布… Ⅲ.软件开发—项目管理—经验—美国—英文 Ⅳ.TP311.52
中国版本图书馆 CIP 数据核字（2003）第 014375 号

责任编辑：杨伟国
丛 书 名：原版风暴·软件工程系列
书　　名：人月神话（影印版）
编　　著：（美）弗雷德里克·布鲁克斯（Frederick P. Brooks Jr.）
出 版 者：中国电力出版社
地址：北京市三里河路 6 号　邮政编码：100044
电话：（010）88515918　传真：（010）88423191
印　　刷：北京地矿印刷厂
发 行 者：新华书店总店北京发行所
开　　本：880×1230 1/32　**印　　张：**10.5　**字　　数：**288 千字
书　　号：ISBN 7-5083-1303-8
版　　次：2003 年 3 月第 1 版
印　　次：2003 年 5 月第 2 次印刷
印　　数：5001～7000
定　　价：25.00 元

Dedication of the 1975 edition

To two who especially enriched my IBM years:
Thomas J. Watson, Jr.,
whose deep concern for people still permeates his company,
and
Bob O. Evans,
whose bold leadership turned work into adventure.

Dedication of the 1995 edition

To Nancy,
God's gift to me.

Preface to the 20th Anniversary Edition

To my surprise and delight, *The Mythical Man-Month* continues to be popular after 20 years. Over 250,000 copies are in print. People often ask which of the opinions and recommendations set forth in 1975 I still hold, and which have changed, and how. Whereas I have from time to time addressed that question in lectures, I have long wanted to essay it in writing.

Peter Gordon, now a Publishing Partner at Addison-Wesley, has been working with me patiently and helpfully since 1980. He proposed that we prepare an Anniversary Edition. We decided not to revise the original, but to reprint it untouched (except for trivial corrections) and to augment it with more current thoughts.

Chapter 16 reprints "No Silver Bullet: Essence and Accidents of Software Engineering," a 1986 IFIPS paper that grew out of my experience chairing a Defense Science Board study on military software. My coauthors of that study, and our executive secretary, Robert L. Patrick, were invaluable in bringing me back into touch with real-world large software projects. The paper was reprinted in 1987 in the IEEE *Computer* magazine, which gave it wide circulation.

"No Silver Bullet" proved provocative. It predicted that a decade would not see any programming technique that would by itself bring an order-of-magnitude improvement in software productivity. The decade has a year to run; my prediction seems safe. "NSB" has stimulated more and more spirited discussion in the literature than has *The Mythical Man-Month*. Chapter 17, therefore, comments on some of the published critique and updates the opinions set forth in 1986.

In preparing my retrospective and update of *The Mythical Man-Month*, I was struck by how few of the propositions asserted in it have been critiqued, proven, or disproven by ongoing software engineering research and experience. It proved useful to me now to catalog those propositions in raw form, stripped of supporting arguments and data. In hopes that these bald statements will invite arguments and facts to prove, disprove, update, or refine those propositions, I have included this outline as Chapter 18.

Chapter 19 is the updating essay itself. The reader should be warned that the new opinions are not nearly so well informed by experience in the trenches as the original book was. I have been at work in a university, not industry, and on small-scale projects, not large ones. Since 1986, I have only taught software engineering, not done research in it at all. My research has rather been on virtual environments and their applications.

In preparing this retrospective, I have sought the current views of friends who are indeed at work in software engineering. For a wonderful willingness to share views, to comment thoughtfully on drafts, and to re-educate me, I am indebted to Barry Boehm, Ken Brooks, Dick Case, James Coggins, Tom DeMarco, Jim McCarthy, David Parnas, Earl Wheeler, and Edward Yourdon. Fay Ward has superbly handled the technical production of the new chapters.

I thank Gordon Bell, Bruce Buchanan, Rick Hayes-Roth, my colleagues on the Defense Science Board Task Force on Military Software, and, most especially, David Parnas for their insights and stimulating ideas for, and Rebekah Bierly for technical production of, the paper printed here as Chapter 16. Analyzing the software problem into the categories of *essence* and *accident* was inspired by Nancy Greenwood Brooks, who used such analysis in a paper on Suzuki violin pedagogy.

Addison-Wesley's house custom did not permit me to acknowledge in the preface to the 1975 edition the key roles played by their staff. Two persons' contributions should be especially cited: Norman Stanton, then Executive Editor, and Herbert Boes, then Art Director. Boes developed the elegant style, which one reviewer especially cited: "wide margins, [and] imaginative use of typeface and layout." More important, he also made the crucial recommendation that every chapter have an opening picture. (I had only the Tar Pit and Reims Cathedral at the time.) Finding the pictures occasioned an extra year's work for me, but I am eternally grateful for the counsel.

Soli Deo gloria—To God alone be glory.

Chapel Hill, N.C. F. P. B., Jr.
March 1995

Preface to the First Edition

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—in more ways than most professional managers expect.

The lore of the field is accumulating. There have been several conferences, sessions at AFIPS conferences, some books, and papers. But it is by no means yet in shape for any systematic textbook treatment. It seems appropriate, however, to offer this little book, reflecting essentially a personal view.

Although I originally grew up in the programming side of computer science, I was involved chiefly in hardware architecture during the years (1956–1963) that the autonomous control program and the high-level language compiler were developed. When in 1964 I became manager of Operating System/360, I found a programming world quite changed by the progress of the previous few years.

Managing OS/360 development was a very educational experience, albeit a very frustrating one. The team, including F. M. Trapnell who succeeded me as manager, has much to be proud of. The system contains many excellencies in design and execution, and it has been successful in achieving widespread use. Certain ideas, most noticeably device-independent input-output and external library management, were technical innovations now widely copied. It is now quite reliable, reasonably efficient, and very versatile.

The effort cannot be called wholly successful, however. Any OS/360 user is quickly aware of how much better it should be. The flaws in design and execution pervade especially the control program, as distinguished from the language compilers. Most of these flaws date from the 1964–65 design period and hence must be laid to my charge. Furthermore, the product was late, it took more memory than planned, the costs were several times the estimate, and it did not perform very well until several releases after the first.

After leaving IBM in 1965 to come to Chapel Hill as originally agreed when I took over OS/360, I began to analyze the OS/360 experience to see what management and technical lessons were to be learned. In particular, I wanted to explain the quite different management experiences encountered in System/360 hardware development and OS/360 software development. This book is a belated answer to Tom Watson's probing questions as to why programming is hard to manage.

In this quest I have profited from long conversations with R. P. Case, assistant manager 1964–65, and F. M. Trapnell, manager 1965–68. I have compared conclusions with other managers of jumbo programming projects, including F. J. Corbato of M.I.T., John Harr and V. Vyssotsky of Bell Telephone Laboratories, Charles Portman of International Computers Limited, A. P. Ershov of the Computation Laboratory of the Siberian Division, U.S.S.R. Academy of Sciences, and A. M. Pietrasanta of IBM.

My own conclusions are embodied in the essays that follow, which are intended for professional programmers, professional managers, and especially professional managers of programmers.

Although written as separable essays, there is a central argument contained especially in Chapters 2–7. Briefly, I believe that large programming projects suffer management problems different in kind from small ones, due to division of labor. I believe the critical need to be the preservation of the conceptual integrity of the product itself. These chapters explore both the difficulties of achieving this unity and methods for doing so. The later chapters explore other aspects of software engineering management.

The literature in this field is not abundant, but it is widely scattered. Hence I have tried to give references that will both illuminate particular points and guide the interested reader to other useful works. Many friends have read the manuscript, and some have prepared extensive helpful comments; where these seemed valuable but did not fit the flow of the text, I have included them in the notes.

Because this is a book of essays and not a text, all the references and notes have been banished to the end of the volume, and the reader is urged to ignore them on his first reading.

I am deeply indebted to Miss Sara Elizabeth Moore, Mr. David Wagner, and Mrs. Rebecca Burris for their help in preparing the manuscript, and to Professor Joseph C. Sloane for advice on illustration.

Chapel Hill, N.C. F. P. B., Jr
October 1974

1

The Tar Pit

1
The Tar Pit

Een schip op het strand is een baken in zee.
[*A ship on the beach is a lighthouse to the sea.*]

DUTCH PROVERB

C. R. Knight, Mural of La Brea Tar Pits
The George C. Page Museum of La Brea Discoveries,
The Natural History Museum of Los Angeles County

No scene from prehistory is quite so vivid as that of the mortal struggles of great beasts in the tar pits. In the mind's eye one sees dinosaurs, mammoths, and sabertoothed tigers struggling against the grip of the tar. The fiercer the struggle, the more entangling the tar, and no beast is so strong or so skillful but that he ultimately sinks.

Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems—few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty—any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion. Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it. But we must try to understand it if we are to solve it.

Therefore let us begin by identifying the craft of system programming and the joys and woes inherent in it.

The Programming Systems Product

One occasionally reads newspaper accounts of how two programmers in a remodeled garage have built an important program that surpasses the best efforts of large teams. And every programmer is prepared to believe such tales, for he knows that he could build *any* program much faster than the 1000 statements/year reported for industrial teams.

Why then have not all industrial programming teams been replaced by dedicated garage duos? One must look at *what* is being produced.

In the upper left of Fig. 1.1 is a *program*. It is complete in itself, ready to be run by the author on the system on which it was developed. *That* is the thing commonly produced in garages, and

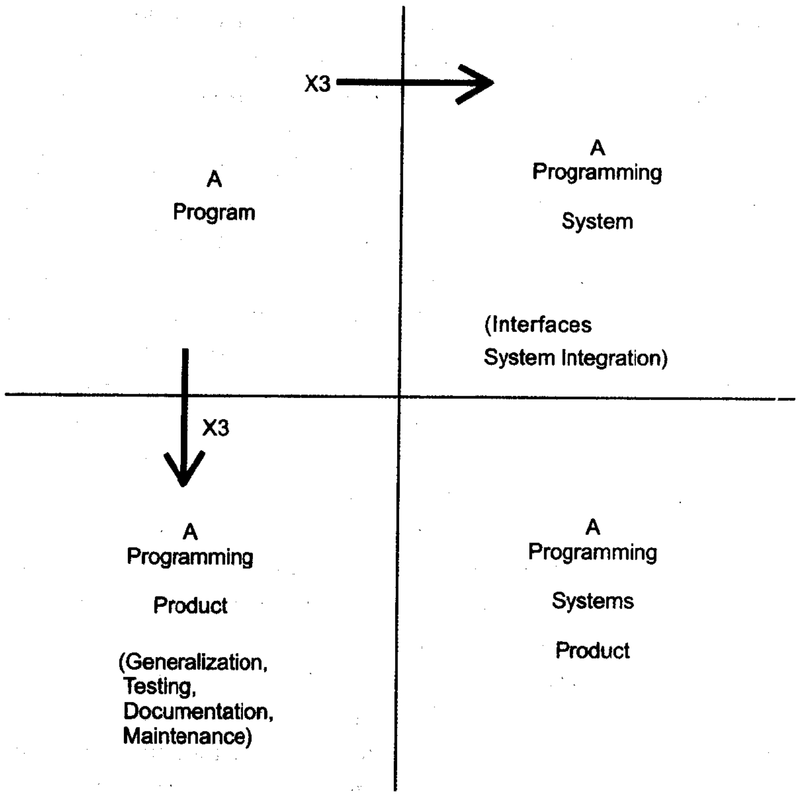

Fig. 1.1 Evolution of the programming systems product

that is the object the individual programmer uses in estimating productivity.

There are two ways a program can be converted into a more useful, but more costly, object. These two ways are represented by the boundaries in the diagram.

Moving down across the horizontal boundary, a program becomes a *programming product*. This is a program that can be run, tested, repaired, and extended by anybody. It is usable in many operating environments, for many sets of data. To become a generally usable programming product, a program must be written in a generalized fashion. In particular the range and form of inputs must be generalized as much as the basic algorithm will reasonably allow. Then the program must be thoroughly tested, so that it can be depended upon. This means that a substantial bank of test cases, exploring the input range and probing its boundaries, must be prepared, run, and recorded. Finally, promotion of a program to a programming product requires its thorough documentation, so that anyone may use it, fix it, and extend it. As a rule of thumb, I estimate that a programming product costs at least three times as much as a debugged program with the same function.

Moving across the vertical boundary, a program becomes a component in a *programming system.* This is a collection of interacting programs, coordinated in function and disciplined in format, so that the assemblage constitutes an entire facility for large tasks. To become a programming system component, a program must be written so that every input and output conforms in syntax and semantics with precisely defined interfaces. The program must also be designed so that it uses only a prescribed budget of resources—memory space, input-output devices, computer time. Finally, the program must be tested with other system components, in all expected combinations. This testing must be extensive, for the number of cases grows combinatorially. It is time-consuming, for subtle bugs arise from unexpected interactions of debugged components. A programming system component costs at least three times as much as a stand-alone program of the same function. The cost may be greater if the system has many components.

In the lower right-hand corner of Fig. 1.1 stands the *programming systems product.* This differs from the simple program in all of the above ways. It costs nine times as much. But it is the truly useful object, the intended product of most system programming efforts.

The Joys of the Craft

Why is programming fun? What delights may its practitioner expect as his reward?

First is the sheer joy of making things. As the child delights in his mud pie, so the adult enjoys building things, especially things of his own design. I think this delight must be an image of God's delight in making things, a delight shown in the distinctness and newness of each leaf and each snowflake.

Second is the pleasure of making things that are useful to other people. Deep within, we want others to use our work and to find it helpful. In this respect the programming system is not essentially different from the child's first clay pencil holder "for Daddy's office."

Third is the fascination of fashioning complex puzzle-like objects of interlocking moving parts and watching them work in subtle cycles, playing out the consequences of principles built in from the beginning. The programmed computer has all the fascination of the pinball machine or the jukebox mechanism, carried to the ultimate.

Fourth is the joy of always learning, which springs from the nonrepeating nature of the task. In one way or another the problem is ever new, and its solver learns something: sometimes practical, sometimes theoretical, and sometimes both.

Finally, there is the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures. (As we shall see later, this very tractability has its own problems.)

Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms. The magic of myth and legend has