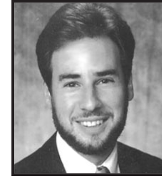
From the Editor



Steve McConnell

Software Engineering Principles

Many software practitioners think of software engineering knowledge almost exclusively as knowledge of specific technologies: Java, Perl, HTML, C++, Linux, Windows NT, and so on. Knowledge of specific technology details is necessary to perform computer programming. If someone assigns you to write a program in C++, you have to know something about C++ to get your program to work.

You often hear people say that software development knowledge has a three-year half-life: half of what you need to know today will be obsolete within three years. In the domain of technology-related knowledge, that's probably about right. But there is another kind of software development knowledge that is likely to serve a professional programmer throughout his or her career.

Essence and Accident

In 1987, Fred Brooks published an influential article, "No Silver Bullets—Essence and Accident in Software Engineering" (*Computer*, April 1987). Its main contention was that no single tool or methodology—no "silver bullet"—portended a 10-to-1 improvement in productivity over the next decade. The reasoning behind that claim is highly relevant to the distinction between technology knowledge and software engineering principles.

In using the words "essence" and "accident," Brooks drew on an ancient philosophical tradition of distinguishing between "essential" and "accidental" properties. Essential properties are those properties that a thing must have to be that thing: A car must have an engine, wheels, and a transmission in order to be a car. These are essential properties. A car might or might not have a V8 or a V6, studded snow tires or racing slicks, an automatic or a manual transmission. These are "accidental" properties, the properties that arise by happenstance and do not affect the basic "car-ness" of the car.

According to Brooks, the most difficult work of software development is not that of representing the concepts faithfully in a specific computer programming language (coding) or checking the fidelity of that representation (testing). That is the accidental part of software engineering.

The essence of software engineering, Brooks argued, consists of working out the specification, design, and verification of a highly precise and richly detailed set of interlocking concepts. What makes software development difficult is its essential complexity, conformity, changeability, and invisibility. Computer programs are complex by nature. Even if you could invent a programming language that operated at the level of the problem domain, programming would still be a complicated activity because you would still need to define relationships between real-world entities precisely, identify exception cases, anticipate all possible state transitions, and so on. Strip away the accidental work involved in representing these factors in a specific programming language within a specific operating system, and you still have the essential difficulty of developing and debugging the underlying real-world concepts.

Additional complexity arises from the fact that software cannot be created in isolation, but must conform to real-world constraints such as pre-existing hardware, third-party components, government regulations, and legacy data formats. The software designer often faces inflexible, external factors that limit the extent to which complexity can be reduced.

Another source of complexity is software's changeability. The more successful a program is, the more uses people will find for it, and the more it will be adapted beyond the domain for which it was originally intended.

EDITOR-IN-CHIEF: Steve McConnell • Construx Software • software@construx.com

Steven C. McConnell. All Rights Reserved @ 1999

A final source of software complexity arises from software's inherent invisibility. Software can't be represented with geometric models. Attempts to geometrically represent even simple, static logic flow quickly become absurdly complicated, as anyone who has ever tried to draw a full flow chart for even a relatively simple program will attest. Adding to the problem, software doesn't exist meaningfully in static form; it only exists meaningfully when it's executed. So even absurdly complicated geometrical representations show software's structure as simpler than it really is because they ignore the time dimension.

Brooks argued that we've already made the major gains in the accidental elements of software engineering, such as the invention of high-level languages, movement to interactive computing from batch processing, and development of powerful integrated environments. Any further order-of-magnitude improvements can be made only by addressing software's essential difficulties: the complexity, conformity, changeability, and invisibility inherent to software development.

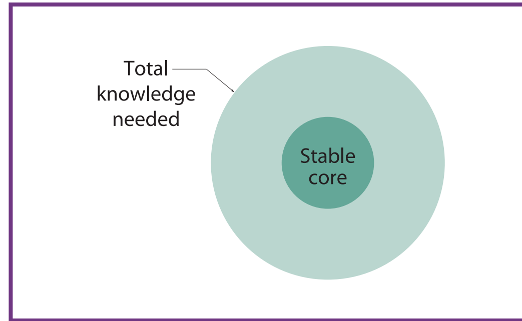


Figure 1. Of the total body of knowledge needed to create a complex software system in 1968, the stable core of knowledge—that part still in use today—comprised only about 10 percent.

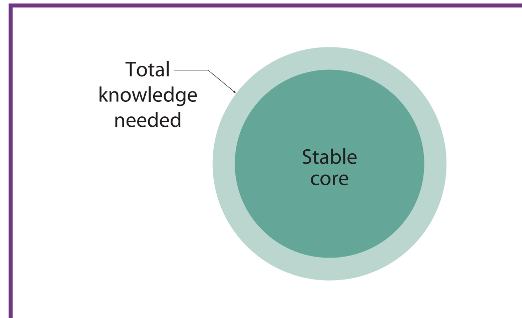


Figure 2. Today, circa 2000, the stable core has grown to about 75 percent of the total.

Software Engineering Principles

Knowledge that addresses what Brooks calls the essential difficulty of software engineering is what I think of as "software engineering principles." During the past 30 years, since the first NATO conference on software engineering in 1968, the software industry has come a long way in identifying the essential knowledge that a software engineer needs in order to develop software effectively.

How small was this body of knowledge in 1968? Consider that the first fully correct binary search algorithm was published just six years before the NATO conference. C. Böhm and G. Jacopini published the paper that made elimination of the goto and structured programming possible only two years before the conference ("Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, May 1966, pp. 366-371). Edsger Dijkstra wrote his famous "GoTo Statement Considered Harmful" letter to the editor the same year as the conference (*Communications of the ACM*, Vol. 11, 1968, pp. 147-148). Larry Constantine, Glenford Myers, and Wayne Stevens didn't publish the first paper on structured design until six years after the conference ("Structured Design," *IBM Systems Journal*, No. 2, 1974, pp. 115-139). Tom Gilb published the first book on software metrics three years later (*Software Metrics*, Winthrop Publishers, 1977), and Tom DeMarco's landmark book on structured analysis didn't appear until two years after that (*Structured Analysis and System Specification*, Prentice Hall, 1979). Anyone who tried to identify a stable core of knowledge in 1968 would have had their work cut out for them. Moreover, it's hard to find much knowledge that would have been considered part of a "stable core" of software engineering knowledge in 1968 that is still in use today—perhaps only about 10 percent.

Think of this in terms of the total body of knowledge needed to create a complex software system, and the proportion of that body of knowledge that is relatively stable—the part that is not going to be obsolete within three years. As Figure 1 illustrates, the stable core of software engineering knowledge at the time of the 1968 NATO conference was relatively small. Today, we still do not have perfect knowledge of what is needed to develop software

effectively. As Figure 2 shows, I estimate that the stable core now occupies perhaps 75 percent of the knowledge needed to develop a software system. But we probably don't ever want a core that makes up 100 percent of the body of knowledge; that would indicate that software engineering had reached a steady state and was immune to future improvements.

Considered from the body-of-knowledge viewpoint, I think we've made significant progress in the areas Brooks refers to as the "essential difficulties" of software development. We now have adequate or good reference books on requirements development, design, construction, testing, reviews, quality assurance, software project management, algorithms, and user interface design, just to name a few topics, and new and better books that codify the knowledge needed to be a software engineer are appearing regularly. An investment in learning software engineering principles is a particularly good investment for a software professional to make because that knowledge will last a whole career.

As a software development professional, you need knowledge of specific technologies to do your job. But you need knowledge of software engineering principles to do your job well. A continuing pursuit of such knowledge is one mark of a true professional. ❖

IEEE SOFTWARE

Editorial Board

Ted Biggerstaff (Microsoft), Maarten Boasson (Hollandse Signaalapparaten), Terry Bollinger (MITRE), Andy Bytheway (Univ. of the Western Cape), David Card (Software Productivity Consortium), Carl Chang (Univ. of Ill., Chicago), Larry Constantine (Constantine & Lockwood), Christof Ebert (Alcatel Telecom), Robert Glass (Computing Trends), Lawrence D. Graham (Christensen, O'Connor, Johnson, & Kindness), Natalia Juristo (Universidad Politécnica de Madrid), Barbara Kitchenham (Univ. of Keele), Tomoo Matsubara (Matsubara Consulting), Nancy Mead (Software Eng. Inst.), Stephen Mellor (Project Technology), Pradip Srimani (Colorado State Univ.), Wolfgang Strigel (Software Productivity Centre), Jeffrey M. Voas (Reliable Software Technologies Corporation), Karl E. Wiegers (Process Impact)

Industry Advisory Board

Robert Cochran (Catalyst Software), Annie Kuntzmann-Combelles (Objectif Technologie), Alan Davis (Omni-Vista), Enrique Draier (Netsystem SA), William Griffin (GTE Labs), Arthur Hersh (Hersh Group), Eric Horvitz (Microsoft), Dehua Ju (ASTI Shanghai), Donna Kasperson (Science Applications Int'l), Günter Koch (Austrian Research Centers), Wojtek Kozaczynski (Rational Software Corp.), Karen Mackey (Lockheed Martin), Masao Matsumoto (Univ. of Tsukuba), Susan Mickel (Rational Univ.), Deependra Moitra (Lucent Technologies, India), Melissa Murphy (Sandia), Kiyoh Nakamura (Fujitsu), Grant Rule (Guild of Independent Function Point Analysts), Chandra Shekaran (Microsoft), Martyn Thomas (Praxis), Sadakazu Watanabe (Fukui Univ.)

Contributing Editors

Ware Myers, Roger Pressman, Ellen Ullman, Mike Yacci

Magazine Operations Committee

Carl Chang (chair), William Everett (vice chair), James Aylor, Jean Bacon, Wushow Chou, George Cybenko, William Grosky, Steve McConnell, Daniel E. O'Leary, Ken Sakamura, Munindar P. Singh, James J. Thomas, Yervant Zorian

Publications Board

Benjamin Wah (chair), Jake Aggarwal, Gul Agha, Jon Butler, Alberto del Bimbo, Sorel Reisman, Ron Williams, Zhiwei Xu

Editor-in-Chief: **Steve McConnell**
10662 Los Vaqueros Circle
Los Alamitos, CA 90720-1314
software@construx.com

Editors-in-Chief Emeritus:
Carl Chang and Alan M. Davis

Managing Editor: **Dale C. Strok**
dstrok@computer.org
Staff Editor: **Anne C. Lear**
Assistant Editor: **Cheryl Baltes**
Magazine Assistant: **Robin Martin**
rmartin@computer.org

Art Director: **Jill Boyer**
Cover Illustration: **Dirk Hagner**
Technical Illustrator: **Alex Torres**
Production Artist: **Jill Boyer**

Publisher: **Matt Loeb**
Membership/Circulation
Marketing Manager: **Georgann Carter**
Advertising Manager: **Patricia Garvey**
Advertising Coordinator:
Marian Anderson

Editorial: Send 2 electronic versions (1 word-processed and 1 postscript) to Managing Editor, *IEEE Software*, 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1314; software@computer.org. Articles must be original and not exceed 5,400 words including figures and tables, which count for 200 words each. All submissions are subject to editing for clarity, style, and space. Unless otherwise stated, bylined articles and departments, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *IEEE Software* does not necessarily constitute endorsement by the IEEE or the IEEE Computer Society.

Copyright and reprint permission: Copyright © 1999 by the Institute of Electrical and Electronics Engineers, Inc. All rights reserved. Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limits of US copyright law for private use of patrons those post-1977 articles that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Dr., Danvers, MA 01923. For copying, reprint, or republication permission, write to Copyright and Permissions Dept., IEEE Publications Admin., 445 Hoes Ln., Piscataway, NJ 08855-1331.

Circulation: *IEEE Software* (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: 345 E. 47th St., New York, NY 10017-2394. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1314; (714) 821-8380; fax (714) 821-4010. IEEE Computer Society headquarters: 1730 Massachusetts Ave. NW, Washington, DC 20036-1903. Annual electronic/paper/combo subscription rates for 1999: $27/34/44 in addition to any IEEE Computer Society dues, $49 in addition to any IEEE dues; $93 for members of other technical organizations. Nonmember subscription rates available on request. Back issues: $10 for members, $20 for nonmembers. This magazine is available on microfiche.

Postmaster: Send undelivered copies and address changes to Circulation Dept., *IEEE Software*, PO Box 3014, Los Alamitos, CA 90720-1314. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Product (Canadian Distribution) Sales Agreement Number 0487805. Printed in the USA.