CHAPTER FIVE

Body of Knowledge

Truth will sooner come out of error than from confusion.

FRANCIS BACON

A person needs to know about 50,000 chunks of information to be an expert in a field, where a chunk is any piece of knowledge that can be remembered rather than derived.[1] In mature fields, it typically takes at least ten years for a world-class expert to acquire that much knowledge. Some people have argued that software-related knowledge isn't stable enough to be codified into a well-defined body of knowledge. They say that half of what a person needs to know to develop software today will be obsolete within three years. If the half-life claim is true, in the 10 years it would take an expert to learn 50,000 chunks of information, 30,000 of those chunks would become obsolete. Would-be software engineers would be like Sisyphus, pushing a boulder up a mountain only to have the boulder roll down the mountain as soon as it reaches the top.

What are the half-lives of Java, Perl, C++, Linux, and Microsoft Windows? All these technologies are highly relevant as I write this, but will they still be relevant by the time you read this? The half-life claim might well be true for technology-related knowledge. But there is another kind of software development knowledge that is likely to serve a professional programmer throughout his or her career, and that knowledge is not subject to these same limitations.

Essence and Accident

In 1987, Fred Brooks published an influential article, "No Silver Bullets—Essence and Accidents of Software Engineering."[2] Its main contention is that no single tool or methodology—no "silver bullet"—portended a 10 to 1 improvement in productivity over the next decade. The reasoning behind this claim helps in identifying software development knowledge not subject to the three-year half-life.

In using the words "essence" and "accident," Brooks draws 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 to be a car. These are essential properties. A car might have a V8 or an in-line six, studded snow tires or racing slicks, an automatic transmission or a stick shift. These are "accidental" properties, the properties that arise by happenstance and do not affect the basic "car-ness" of the car. The term "accidental" can be confusing, but it just means nonessential or optional.

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 development. The essence of software development, Brooks argues, consists of working out the specification, design, and verification of a highly precise and richly detailed set of interlocking concepts. He says that software is difficult because of its essential complexity, conformity, changeability, and invisibility.

Computer programs are *complex* by nature. Even if you could invent a programming language that operated exactly at the level of the problem domain, programming would still be a complicated activity because you would still need to precisely define relationships between real-world entities, 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 and in a specific computing environment, and you will still have the essential difficulty of defining the underlying real-world concepts and debugging your understanding of them.

Another essential difficulty arises from the need for software to *conform* to messy real-world constraints such as pre-existing hardware, third-party components, government regulations, business rules, and legacy data formats. The software designer often faces inflexible external considerations that limit the extent to which complexity can be reduced.

Software's *changeability* presents another essential difficulty. 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. As the software grows, it will become more complex and will need to conform to additional constraints. The more software is adapted, the more involved the adaptations become.

A final source of essential difficulty arises from software's inherent *invisibility.* Software can't be visualized with 2-D or 3-D geometric models. Attempts to visually represent even simple logic flow quickly become absurdly complicated, as anyone who has ever tried to draw a flow chart for even a simple program will attest.

Brooks argues that software development has already made all possible major gains in the accidental elements. These gains include the invention of high-level languages, adoption of interactive computing, and development of powerful integrated environments. Any further order-of-magnitude productivity improvements, he says, can be made only by addressing software's essential difficulties: the complexity, conformity, changeability, and invisibility inherent in software development.

Defining a Stable Core

Knowledge that helps developers deal with what Brooks calls the "essential difficulties" of software development is what I think of as "software engineering principles," which make up software engineering's core body of knowledge. In 1968, NATO held the first conference on software engineering. Using the term "software engineering" to describe the body of knowledge that existed at that time was premature, and it was intended to be provocative.

Exactly how small was the stable core in 1968? Consider that the first fully correct binary search algorithm was published in 1962, only six years before the NATO conference.[3] C. Böhm and G. Jacopini published the paper that established the theoretical foundation for eliminating the *goto* and the creation of structured programming only two years before the conference.[4] Edsger Dijkstra wrote his famous letter to the editor, "GoTo Statement Considered Harmful," in 1968.[5] At the time the conference was held, subroutines were a fairly new idea, and programmers regularly debated whether they were really useful. Larry Constantine, Glenford Myers, and Wayne Stevens didn't publish the first paper on structured design until six years after the conference in 1974.[6] Tom Gilb published the first book on software metrics in 1977,[7] and Tom DeMarco published the first book on software requirements analysis in 1979.[8] Anyone who tried to identify a stable core of knowledge in 1968 would have had their work cut out for them.

From an analysis of the SWEBOK project's Body of Knowledge areas (which I'll discuss later in this chapter), I estimate that the half-life of software engineering's body of knowledge in 1968 was only about 10 years. As Figure 5-1 illustrates, the stable core was relatively small, and I estimate that only about 10 to 20 percent of software engineering knowledge from 1968 is still in use today.

Software engineering has made significant progress since 1968. Hundreds of thousands of pages have been written about software engineering. Professional societies host hundreds of conferences and workshops every year. Knowledge has been codified into more than 2,000 pages of IEEE software engineering standards. Dozens of universities across North America offer graduate education in software engineering, and dozens more have recently begun to offer undergraduate programs.

The fact that we do not have perfectly stable knowledge of software engineering practices hardly makes software engineering unique. In the 1930s, the medical profession did not yet know about penicillin, the structure of DNA, or the genetic basis of many diseases, and it did not have technologies such as heart-lung machines and magnetic resonance imaging. And yet there was a profession of medicine.

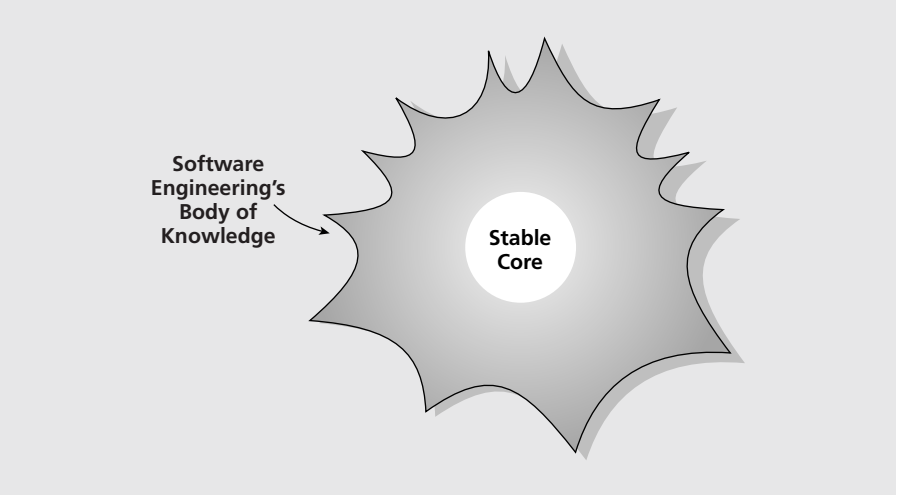

FIGURE 5-1

As of the 1968 NATO Conference on Software Engineering, only about 10 to 20 percent of the software engineering body of knowledge was stable (i.e., would still be relevant 30 years later). The half-life of the software engineering body of knowledge at that time was about 10 years.

As Figure 5-2 suggests, based on my SWEBOK knowledge-area analysis, I estimate that as of 2003 the stable core now makes up about 50 percent of the knowledge needed to develop a software system. That might not seem like a dramatic change from the 10 to 20 percent of 1968, but it implies that the body of knowledge's half-life has improved from about 10 years to about 30 years. That means that the educational investment a person makes at the beginning of a career in software will remain largely relevant throughout that person's career.

Stabilization of software engineering's body of knowledge puts software engineering on an educational footing similar to other engineering disciplines. As David Parnas points out, the content of a physics class can remain unchanged even if the lab gets a new oscilloscope. Most of the content of software engineering courses can be independent of specific, relatively short-lived technologies like C++ and Java. Students will have to

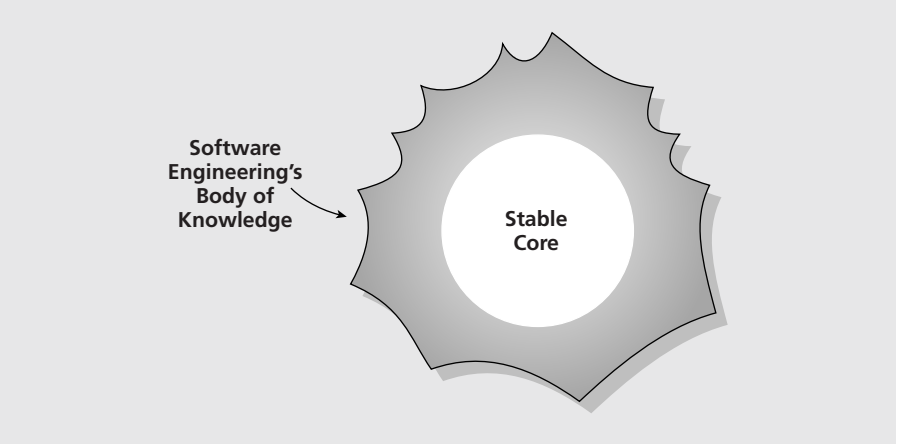

FIGURE 5-2
As of 2003, about 50 percent of the software engineering body of knowledge is stable and will still be relevant 30 years from now.

learn those technologies in the lab, but in the classroom they can focus on more lasting knowledge.

Software Engineering's Body of Knowledge

In Chapter 4, I argued that software engineering is not the same as computer science, but if it isn't computer science, what is it? Those of us working in software development now have an exciting opportunity to see a new field being born. For more established fields like mathematics, physics, and psychology, we tend to take the contents of the field for granted, assuming that the definition of what is in and what is out of the field has always been the way it is and has to be that way. But at some point people working in each field developed textbooks and university curriculums that required them to decide what knowledge was in and what was out. For hundreds of years, people didn't differentiate between mathematics, physics, psychology, and philosophy. Mathematics began to be treated as separate from philosophy about 300 A.D.. Physics began to be treated as

separate from philosophy about 1600. Psychology wasn't distinguished from philosophy until about 1900.

In defining what knowledge is in and what is out of the field of software engineering, experts have recommended that the focus should be on generally accepted knowledge and practices. As Figure 5-3 suggests, "generally accepted" refers to the knowledge and practices that are applicable to most projects most of the time—practices that most experts would agree are valuable and useful. Generally accepted does not mean that the knowledge and practices should be applied uniformly to all projects. Project leaders will still be responsible for determining the most appropriate practices for any particular project.[9]

Since 1968, we've made significant progress in the areas Brooks referred to as the "essential difficulties" of software development. We now have adequate or good reference books on requirements engineering, design,

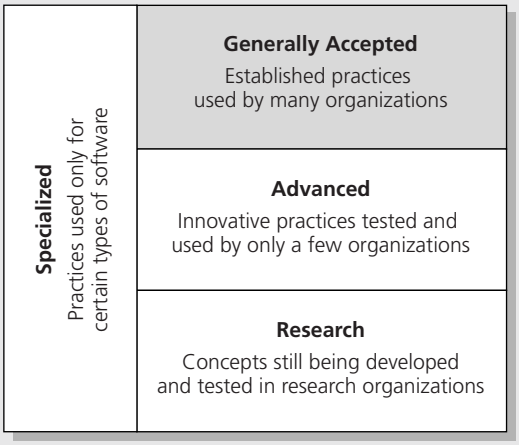

FIGURE 5-3

Categories of possible knowledge in the software engineering body of knowledge. The focus of defining the knowledge needed by a professional software engineer is on generally accepted knowledge.

Source: Adapted from "Professionalism of Software Engineering: Next Steps"[10]

construction, testing, reviews, quality assurance, software project management, algorithms, and user interface design, just to name a few topics. New and better books that codify software engineering knowledge are appearing regularly. Some core elements have not yet been brought together in practical textbooks or courses, and in that sense our body of knowledge is still under construction. But the basic knowledge about how to perform each of these practices is available—in journal articles, conference papers, and seminars as well as books. (These books are listed on the software engineering professional Web site described at the back of the book.) The pioneers of software engineering have already blazed the trails and surveyed the land. Now the software engineering settlers need to build the roads and develop the rest of the education and accreditation infrastructure.

Researchers at the Université du Québec à Montréal have spearheaded an effort to identify the generally accepted elements of software engineering. This effort has been coordinated by the ACM and the IEEE Computer Society and involves both academic and industrial participants. This effort is called the Software Engineering Body of Knowledge project, or SWEBOK.[11]

As Figure 5-4 suggests, software engineering draws from computer science, mathematics, cognitive sciences (psychology and sociology), project management, and various engineering disciplines.

From this starting point, SWEBOK has identified knowledge areas that make up the core competencies for a professional software engineer.[12]

- *Software Requirements.* The discovery, documentation, and analysis of the functions to be implemented in software.
- *Software Design.* Definition of the basic structure of the system at the architectural and detailed levels, division into modules, definition of interfaces for modules, and choice of algorithms within modules.
- *Software Construction.* Implementation of the software including detailed design, coding, debugging, unit testing, technical reviews, and performance optimization. This area overlaps somewhat with Software Design and Software Testing.

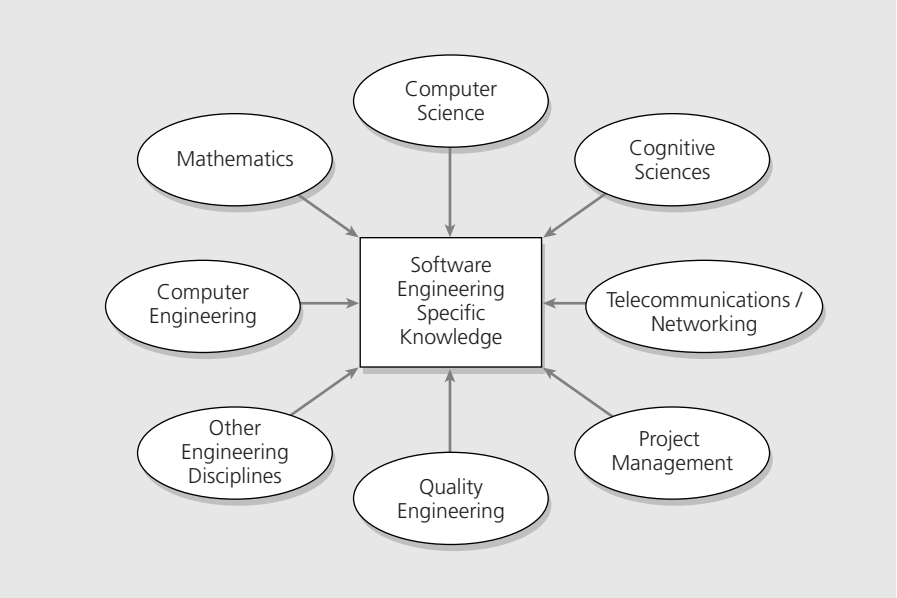

FIGURE 5-4
Sources of knowledge in software engineering's body of knowledge.
Source: Adapted from "Professionalism of Software Engineering: Next Steps"[13]

- *Software Testing.* All activities associated with executing software to detect defects and evaluate features. Testing includes test planning, test case design, and specific kinds of tests including development tests, unit tests, component tests, integration tests, system tests, regression tests, stress tests, and acceptance tests.
- *Software Maintenance.* Revision and enhancement of existing software, related documentation, and tests.
- *Software Configuration Management.* Identification, documentation, and change control of all intellectual property generated on a software project including source code, content (graphics, sound, text, and video), requirements, designs, test materials, estimates, plans, and user documentation.
- *Software Quality.* All activities associated with providing confidence that a software item conforms or will conform to technical

requirements. Quality engineering includes quality assurance planning, quality measurement, reliability, testing, technical reviews, audits, and verification and validation.

- *Software Engineering Management.* Planning, tracking, and controlling of a software project, software work, or a software organization.
- *Software Engineering Tools and Methods.* Tool and methodology support, such as CASE tools, reusable code libraries, and formal methods, including practices for adopting and disseminating tools and methods within an organization.
- *Software Engineering Process.* Activities related to improving software development quality, timeliness, productivity, and other project and product characteristics.

The extent of this list might surprise some people. Many practicing programmers work as though software construction is the only knowledge area that matters. As important as that area is, it is only one of ten areas that a professional software engineer should know.

Other practicing programmers might be surprised at the complete absence of specific languages and programming environments—Java, C++, Visual Basic, Linux, and so on. That's because the body of knowledge emphasizes software engineering principles rather than technology knowledge.

A few people's reaction to these knowledge areas will be, "That's a lot to expect someone to learn just to write computer programs." It is a lot to expect someone to learn, and historically we've been expecting them to learn it implicitly, through on-the-job exposure to new information. The result is that most practicing computer programmers have pretty good knowledge of Software Construction and Software Maintenance; marginal knowledge of Software Requirements, Software Design, Software Testing, and Software Engineering Tools and Methods; and virtually no knowledge of Software Configuration Management, Software Quality, Software Engineering Management, or Software Engineering Process.

I don't expect software engineers to achieve mastery in each of these areas, but a professional software engineer should at least acquire introductory knowledge of all areas, competence in most, and mastery of some.

As I described in Chapter 4, one of the differences between a scientist and an engineer is that scientists can afford to have knowledge that is narrow and deep, but engineers need a broad understanding of all the factors that affect the products they create.

Planting a Stake

Is this definition of software engineering's body of knowledge the final answer? No. The field of medicine has continued to evolve, and the field of software engineering will continue to evolve too. But there is great value in planting a stake in the ground and saying, "This is what constitutes the software engineering body of knowledge at this time."

As Francis Bacon pointed out when he laid the foundation for modern science 350 years ago, errors are a better basis for progress than confusion is. Bacon knew that when his approach was used many of the initial conclusions—the "first vintages"—would be mistaken, but that was part of his plan. Major elements of our current definition of the software engineering body of knowledge will undoubtedly turn out to be mistaken, but an imperfect, clear definition will give us a baseline upon which we can improve. Exchanging the current confused muddle for a clearly defined body of knowledge is a good trade, errors and all.

Notes

1. Shaw, Mary, "Prospects for an Engineering Discipline of Software," *IEEE Software,* November 1990, pp. 15ff.
2. Brooks, Frederick P., Jr., "No Silver Bullets—Essence and Accidents of Software Engineering," *Computer,* April 1987, pp. 10–19.
3. Knuth, Donald, *The Art of Computer Programming, Volume 3: Sorting and Searching,* Reading, MA: Addison-Wesley, 1973, p. 419.
4. "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM,* May 1966, pp. 366–371.
5. *Communications of the ACM,* Vol. 11, 1968, pp. 148ff. Also available from www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF.

6. "Structured Design," *IBM Systems Journal,* No. 2, 1974, pp. 115–139.
7. *Software Metrics,* Cambridge, MA: Winthrop Publishers, 1977.
8. *Structured Analysis and System Specification,* NJ: Prentice Hall, 1979.
9. Duncan, W. R., "A Guide to the Project Management Body of Knowledge," Upper Darby, PA: Project Management Institute, 1996.
10. Tripp, Leonard, "Professionalism of Software Engineering: Next Steps," Keynote Address at *12th Conference on Software Engineering Education and Training,* March 22, 1999.
11. Additional information is available on the SWEBOK Web site at www.swebok.org.
12. "Guide to the Software Engineering Body of Knowledge: Trial Version (Version 1.00)," Alain Abran, et al., IEEE Computer Society, 2001.
13. Tripp, Leonard, "Professionalism of Software Engineering: Next Steps," Keynote Address at *12th Conference on Software Engineering Education and Training,* March 22, 1999.