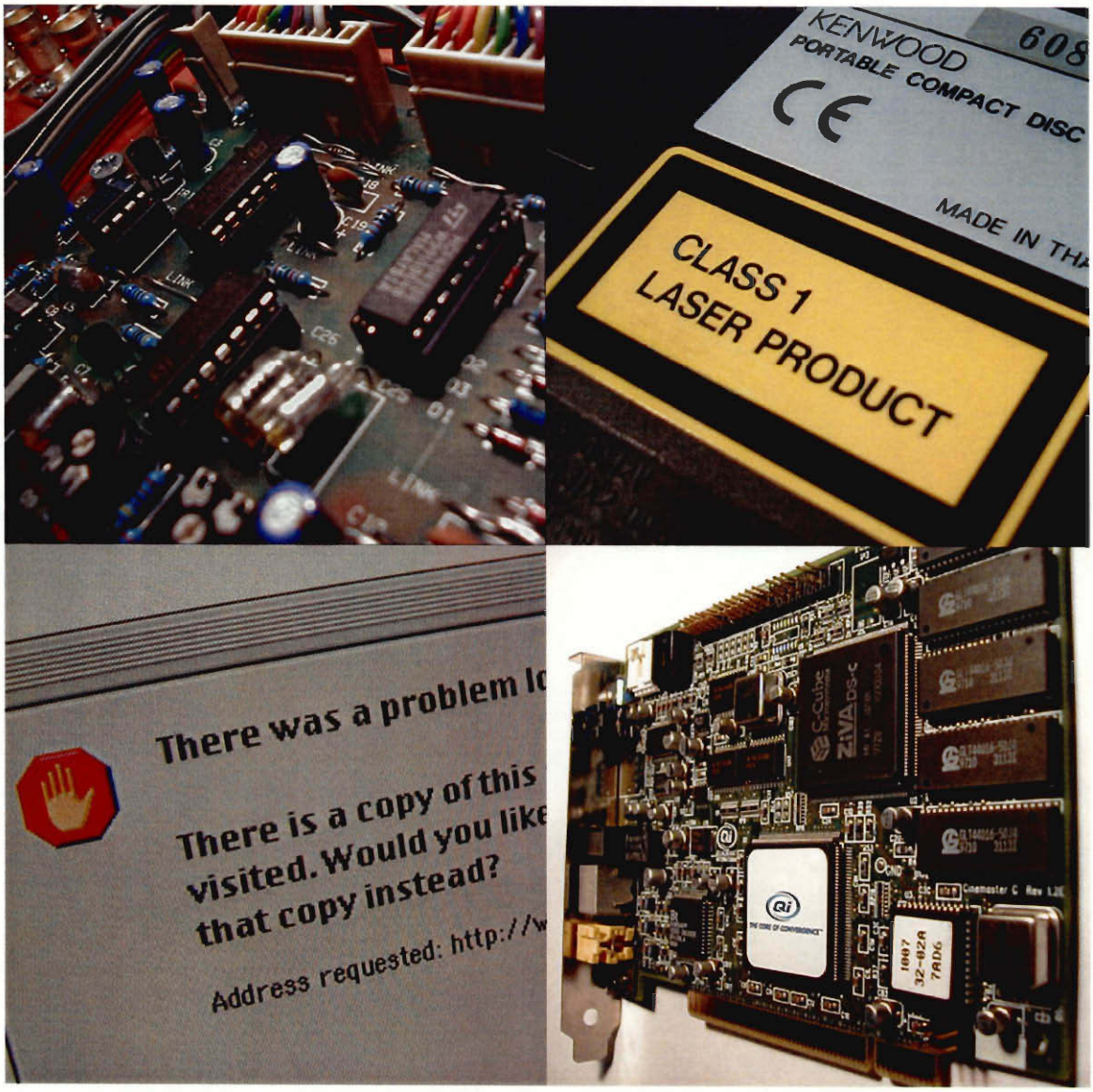
01D014I
GTH-1011-45

Trust and Component-Based Software Engineering



By Paul W. Calnan, III and Anthony J. Andrade, Jr.
Dr. George T. Heineman, Project Advisor

Project Number: GTH-I011

Trust and Component-Based Software Engineering

An Interactive Qualifying Project Report
submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science

By

Paul W. Calnan, III

Anthony J. Andrade, Jr.

Date: March 12, 2001

Approved:

Professor George T. Heineman
Project Advisor

CONTENTS

A B S T R A C T

Software productivity has been a problem since the 1960s. Projects are often delayed, over budget, or even canceled. Component-based software development, as a means of code reuse, has been viewed as a way to help improve the state of software engineering, but a marketplace for software components currently does not exist. It is our assertion that trust is a necessary but not a sufficient precondition for the development of a software component marketplace. To this end, we examine a variety of ways that trust can be engendered in components: creating a profession of software engineering, product certification, and process assessment.

ACKNOWLEDGEMENTS

We would like to thank Professor Heineman for his invaluable help and support throughout the course of this project, as well as allowing us to use portions of his book, *Component-Based Software Engineering*, prior to its publication.

Cover page photographs courtesy of Free Images (http://www.freeimages.co.uk/).

CHAPTER 1

Introduction

In 1968, the NATO Software Engineering Conference coined the term *software crisis* in recognition of the problems that the software industry faced at that time: high cost of development, low quality products, difficulty in scheduling, and management difficulties [CoxByte]. Since then, computers have developed into a worldwide mass-market. The cost of hardware has decreased dramatically while its speed has doubled almost every eighteen months. Gains in software productivity and reductions in costs have not kept pace. Software development continues to see problems similar to those discussed at the NATO conference thirty-three years ago: schedule pressure, a shortage of programmers, ever-increasing project sizes, and persistent managernent difficulties [McConnell, p.4]. The software crisis is not over.

Many in the industry complain that there is a lack of skilled personnel to fill the programming jobs that are available. It was estimated that thirty years ago, there were 50,000 jobs unfilled due to a shortage of labor in the software industry [McConnell, p.4]. Today, industry officials claim that at least 346,000 high-tech jobs remain unfilled [UTHCT]. This issue has become politically charged over the past few years as the Clinton administration and the Congress have disagreed on the issue of H1-B visas that would allow foreign nationals holding college degrees to work in the United States for up to six years [InternetNews].

Projects continue to be quite large, requiring massive amounts of effort to complete. The initial Windows NT development required 1,500 staff-years of effort. IBM's OS/360, which was completed in 1966, required more than three times as much effort [McConnell, p.4].

A study published in *Patterns of Software System Failure and Success* in 1995 (see Table 1) shows that almost one-quarter of all software projects are canceled. Other research by The Standish Group shows that 31.1% of all software projects will be canceled prior to completion and that 52.7% of projects will cost 189% of their original cost estimates. They estimate that in

1995, canceled software projects cost American companies and government agencies $81 billion. It is estimated that software projects that take longer than originally estimated will cost those same organizations an additional $59 billion [UQAM].

Size of Project	Early	On-Time	Delayed	Canceled	Sum
1 function point	14.68%	83.16%	1.92%	0.25%	100.00%
10 function points	11.08%	81.25%	5.67%	2.00%	100.00%
100 function points	6.06%	74.77%	11.83%	7.33%	100.00%
1,000 function points	1.24%	60.76%	17.67%	20.33%	100.00%
10,000 function points	0.14%	28.00%	23.83%	48.00%	100.00%
100,000 function points	0.00%	13.67%	21.33%	65.00%	100.00%
Average	5.53%	56.94%	13.71%	23.82%	100.00%

Table 1: Percentage of Software Projects Early, On-Time, Delayed, Canceled [Jones, Yourdon]

While this is just anecdotal evidence of the productivity problems seen in software development today, it is clear that something needs to change in order to increase the productivity, predictability, and manageability of software projects. Much has been written over the years suggesting changes and improvements to the software development process. In his 1986 paper "No Silver Bullet—Essence and Accident in Software Engineering," Fred Brooks compares the productivity problems that often surface in software projects to werewolves--monsters that transform "from the familiar into horrors." He explains that the software industry has long looked for a "silver bullet" solution that would kill the beast and make software development more productive. Brooks's thesis is that no single development—no silver bullet—would bring an order-of-magnitude improvement in productivity, reliability, or simplicity during the years 1986-1996 [Brooks, p.179]. By all measure, his thesis has been generally accepted.

Brooks begins by dividing the difficulties faced in software development into the essential and the accidental. Essential difficulties are those that are inherent in software development: data sets, relationships among data items, algorithms, and function invocations. They are abstract in that they are conceptually the same under different representations [Brooks, p.182]. Accidental difficulties come up in the production of software but are not inherent in it. They arise while representing the essential concepts under a particular representation. The central question raised by Brooks is, "What fraction of total software effort is now associated with the accurate and

orderly representation of the conceptual construct, and what fraction is the effort of mentally crafting the constructs?" [Brooks, p.209] He goes on to argue that the difficulties in software design come from the specification, design, and testing of the program, not the representing it and testing the fidelity of the representation [Brooks, p.182]—in other words, the essential not the accidental. The software industry must attack the essential problems in order to see order-of-magnitude improvements in productivity, because this is where most of the difficulties lie.

Many looked to object-oriented (OO) programming as a means of solving some of the essential difficulties that faced software development. OO programming offers software designers the ability to write modular code with well-defined interfaces. It also gives programmers more powerful design principles to apply to their craft: abstraction, encapsulation, inheritance, and polymorphism. The modularity and abstraction that comes with OO makes it possible for programmers to develop software with conceptually larger pieces, thus removing some of the fine-grained complexity that is common under other design methods. However, many would argue that OO programming has not improved productivity or software quality as much as originally promised. David Parnas, one of the originators of the object-oriented concept, sees the problem as follows:

> *[OO programming] has been tied to a variety of complex languages. Instead of teaching people that OO is a type of design, and giving them design principles, people have taught that OO is the use of a particular tool. We can write good or bad programs with any tool. Unless we teach people how to design, the languages matter very little. The result is that people do bad designs with these languages and get very little value from them* [Brooks, p.221].

To use Brooks's terminology, OO has been used in the realm of the accidental rather than in the realm of the essential. In order to make real improvements in the realm of productivity, we need to turn our efforts to the realm of the essential difficulties of software development.

There exists a gap between what software developers would like to create and what they are able to create. What is needed is a way to decompose a software project, breaking the complexity of the problem into manageable components. As projects increase in size, this becomes more apparent. In the past, large projects were written with thousands of lines of

code—today, large projects are written with millions of lines of code. If this trend continues, large projects may eventually be written with billions or trillions of lines of code. However, the complexity, cost, and time to develop a project that large will be unreasonable, to say the least. The problem of complexity is not a new one. Modern industry has seen this problem and has found a solution: standard, interchangeable components.

In his paper "What If There's A Silver Bullet... And the Competition Gets It First," Brad Cox sums up his view of the essential difficulties of software development in a cartoon caption. A plumber says to his client, "Don't waste your money on generic, 'reusable' components from the plumbing supply store. Every component in a proper plumbing system should be custom-designed right from the ground up" [CoxByte]. Consider for a moment how expensive and difficult it would be if your plumbing system was designed from the ground up. It is not designed that way because it would not be feasible. Likewise, electrical engineers do not design every capacitor and resistor in their systems, nor do mechanical engineers design every nut and bolt in their projects. Cox goes on to explain that the process consumes the software community: we build our software from first principles, using very fine-grained building blocks. While all other engineering disciplines have "defined standard products and allowed diverse processes to be used in making them, in software we do the reverse, defining standard languages and methodologies from which standard components are to magically ensue" [CoxPSIR].

All mature engineering disciplines have embraced the concepts of reuse and standard components. For some reason, software engineering has not. It has long been recognized that code reuse is a good way to improve programmers' productivity. Programmers often reuse fragments of their own code, as well as system libraries and GUI toolkits. The industry has not, however, embraced reuse to the extent that other engineering disciplines have. What better way to attack the essential difficulties of software development than not to write the software in the first place? Software components, "binary units of independent production, acquisition, and deployment that interact to form a functioning system," [Szyperski, p.xiii] would provide an excellent vehicle for such reuse. The underlying technologies that support component-based software (e.g. COM, CORBA, EJB) have been available for some time now. It is evident from

other industries that components are an essential cornerstone to the design and development of new products. But, for some reason, there has yet to develop a marketplace for these reusable components.

Much has already been written extolling the virtues of component-based software engineering: how it reduces time to market, improves productivity and software quality, etc. It is not our intent in this paper to discuss component-based software engineering itself, nor is it our intent to explain why a marketplace for reusable software components does not exist in the industry today. Rather, we intend to examine some of the issues surrounding the creation and the development of a component marketplace. In particular, we will examine the issue of trust, and how it affects the development of a component marketplace. It is our assertion that trust is a necessary but not a sufficient pre-condition for the development of a component marketplace. Trust will not form a component marketplace, but a component marketplace will not form without trust. Bearing that in mind, we will examine ways to engender trust in software components.

Trust is defined by Webster as the "assured resting of the mind on the integrity, veracity, justice, friendship, or other sound principle, of another person" [Webster]. It is the foundation of commerce [Keen, et al., p.1], and it plays a major role in all of our economic transactions: we trust banks to keep our money; we trust that goods paid for will be delivered and will perform as promised; we trust that the currency exchanged has real value; we trust that the issuer of a check has the actual funds promised; and so on. Trust is fostered by a variety of measures. Companies allow for consumers to return goods that are faulty and often provide money-back guarantees in case the goods provided do not perform as promised. A preexisting relationship or the personal reputation of the provider of the good can also provide trust. We trust certain brand names that we are familiar with to be of a certain quality. Often times, our trust is assured in the knowledge that there are legal repercussions for broken trust. There are contracts and regulations that provide the consumer with a means of redress when a product is faulty. Likewise, consumer protection laws provide us with a set of reasonable expectations for a product.

People often have a difficult time trusting software. This is no surprise considering the poor quality of software often seen in today's marketplace. Some in the software industry would have consumers believe that "the complexity of software products makes them inherently imperfect" [Kaner, footnote 64] and that "the idea of perfect software is a goal or aspiration not presently attainable, at least not without exorbitant costs that would drive many thousands of small companies out of the business" [Kaner, footnote 66]. Almost anyone who has used a computer knows from experience that this is true—programs do unpredictable things, computers often "lock up" or crash, and so on. The Y2K "bug" caused many people to panic; in the weeks and months leading up to January 1, 2000, some people were so convinced that computers the world over would crash that they hoarded food and water, built bomb shelters, canceled travel plans, and withdrew extra money from the bank in case disaster struck.

People within the software industry are equally aware of potential problems with software; they have a first-hand view of the difficulties of development and the damage that defects can cause. A leap of faith is required for a company to use components developed by a third-party. Trust in the component producer's product is essential. When a company licenses a component from a third-party, they receive binary code to incorporate into their product along with documentation on how to use the component. No source code is provided. The company that licenses the component can trust the component producer's word that the software performs to specifications. The company can also engage in black box testing of the component to assure that it performs acceptably. Either way, a certain level of trust is required on the part of the licensing company before they can incorporate the component into their product.

The mechanisms that provide legal redress for broken trust mentioned above do not necessarily apply to the realm of software. A close examination of any license agreement that comes with a piece of software shows how careful companies are to limit their liabilities for defects. For example:

> ***Note on Java support.*** *The software product contains support for programs written in Java. Java technology is not fault tolerant and is not designed, manufactured, or intended for use or resale as on-line control equipment in hazardous environments requiring fail-safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines, or weapons*

systems, in which the failure of Java technology could lead directly to death, personal injury, or severe physical or environmental damage.

No other warranties. *To the maximum extent permitted by applicable law, Microsoft and its suppliers disclaim all other warranties, either express or implied, including, but not limited to, implied warranties of merchantability and fitness for a particular purpose, with regard to the software product, and any accompanying hardware.*

Customer remedies. *Microsoft's and its suppliers' entire liability and your exclusive remedy shall be, at Microsoft's option, either (a) return of the price paid, or (b) repair or replacement of the software product or hardware that does not meet Microsoft's Limited Warranty and which is returned to Microsoft with a copy of your receipt. This Limited Warranty is void if failure of the software product or hardware has resulted from accident, abuse, or misapplication.*

No liability for consequential damages. *To the maximum extent permitted by applicable law, in no event shall Microsoft or its suppliers be liable for any special, incidental, indirect, or consequential damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising out of the use of or inability to use the software product, even if Microsoft has been advised of the possibility of such damages.*

[Taken from the Microsoft Windows NT Workstation End-User License Agreement]

A new law called the Uniform Computer Information Transaction Act (UCITA) has been proposed by the National Conference of Commissioners on Uniform State Laws that would further limit the legal remedies for broken trust in a software product. It would extend producers' ability to disclaim all warranties and liability for consequential damages that may arise from using their software. By allowing software producers to avoid facing any legal consequences for defective software, this new law would take away most means for legal redress for broken trust.

Trust is important for component-based software development and trusting software is very difficult, especially considering the state of software today and the legal outs that license agreements and UCITA provide. Therefore, in order for component-based software development to blossom and for a component marketplace to develop, the industry must focus on ways to engender trust in software. We will examine three mechanisms that the software industry can use to engender trust in their products: the formation of a profession of software engineering, product certification, and process assessment. Each will be examined in turn, their benefits and drawbacks will be discussed, and a proposal for implementing each will be put forth.

References

[Brooks] Brooks, Frederick P., Jr. *The Mythical Man-Month* (Anniversary Edition). Reading, MA: Addison-Wesley, 1995.

[CoxByte] Cox, Brad. "What If There's a Silver Bullet… And the Competition Gets It First?" <http://www.virtualschool.edu/cox/CoxByte.html>.

[CoxPSIR] Cox, Brad. "Planning the Software Industrial Revolution." <http://www.virtualschool.edu/cox/CoxPSIR.html>.

[InternetNews] Mark, Roy. "Clinton Proposes New Version of H1-B Bill." dc.internet.com 12 May 2000. <http://dc.internet.com/news/article/0,1934,2101_360741,00.html>.

[Jones] Jones, Capers. *Patterns of Software Failure and Success*. Referenced by [Yourdon].

[Kaner] Kaner, Cem. "Software Engineering and UCITA." <http://www.badsoftware.com/engr2000.htm>.

[Keen, et al.] Keen, Peter, et al. *Electronic Commerce Relationships*. Upper Saddle River, New Jersey: Prentice Hall PTR, 2000.

[McConnell] McConnell, Steve. *After the Gold Rush: Creating a True Profession of Software Engineering*. Redmond, Washington: Microsoft Press, 1999.

[Szyperski] Szyperski, Clemens. *Component Software: Beyond Object-Oriented Programming*. New York: ACM Press, 1999.

[UTHCT] Black, Shaun. "4 Articles on U.S. Senate Vote to Expand H1-B Visa Program." 19 May 1998. Online posting. <http://psyche.uthct.edu/nes/wwwboard/messages/144.html>.

[UQAM] Netscape Component Developer's Kit. <http://www.er.uqam.ca/nobel/m202230/reference.html>.

[Webster] Webster's Revised Unabridged Dictionary, 1996, 1998. <http://www.dictionary.com>.

[Yourdon] Ed Yourdon's Y2K Chronicles, Chapter 11: "Y2K Projects: déjà vu all over again." <http://www.yourdon.com/books/y2k2020/11.dejavu.html>.

CHAPTER 2

Licensing, Certification, and the Software Engineering Profession

Introduction

As we enter the 21st century, the modern world has come to rely upon software. Personal computers and the Internet continue to affect people in ever growing numbers. Computers and software systems are part of our national power grid, air traffic control, and even the anti-lock braking systems in automobiles. Our increased dependence on software was best illustrated by the recent Year 2000 (Y2K) crisis. The idea that computers the world over could stop functioning when the date changed from 1999 to 2000 caused many people to panic. The possibility that financial institutions, food distribution, water supplies, electrical power, air traffic control systems, missile control systems, and other computer systems could fail provided a sense of how dependent we are on software and computers. Y2K also showed us how important good software engineering is—and how something as simple as storing the date using two digits instead of four could have such a colossal effect.

High-profile software failures frequently make national headlines. The Melissa Virus, caused by a vulnerability in the scripting capabilities of Microsoft Outlook, cost companies billions of dollars in lost productivity due to e-mail outages [Busch]. Problems with the Denver International Airport baggage system software delayed the opening of the new airport, eventually costing the airport's planners $1.1 million a day in interest and operating costs while BAE Automated Systems struggled to debug their system [CoxNSBR].

Software delays and failures continue to cost us untold billions each year. A Government Accounting Office (GAO) report on major software challenges stated, "We have repeatedly reported on cost rising by millions of dollars, schedule delays of not months but years, and multi-billion-dollar systems that don't perform as envisioned" [SEI-CMM]. Tables 1 and 2 illustrate the situation:

Size of Project	Early	On Time	Delayed	Cancelled
1 function point	14.68%	83.16%	1.92%	0.25%
10 function points	11.08%	81.25%	5.67%	2.00%
100 function points	6.06%	74.77%	11.83%	7.33%
1,000 function points	1.24%	60.76%	17.67%	20.33%
10,000 function points	0.14%	28.00%	23.83%	48.00%
100,000 function points	0.00%	13.67%	21.33%	65.00%
Average	5.53%	56.94%	13.71%	23.82%

Table 1: Percentage of software projects early, on time, late, canceled [Jones]

Project Size (function points)	Minimum Duration (months)	Actual Duration (months)	Maximum Duration (months)	Estimate (months)	Variance from Estimate	Percent
1	0.06	0.16	0.40	0.15	0.01	6.25%
10	0.35	1.07	2.36	1.00	0.07	6.54%
100	3.60	10.00	19.00	9.00	1.00	10.00%
1,000	12.24	27.20	43.52	22.00	5.20	19.12%
10,000	24.90	49.80	84.66	36.00	13.80	27.71%
100,000	44.28	73.80	132.84	48.00	25.80	34.96%
Average	14.24	27.01	47.13	19.36	7.65	17.43%

Table 2: Extent of project delays [Jones]

23.8% of all software projects are canceled and 13.7% are delayed by more than half a year on average. It is obvious that something must be done to fix this problem.

Researchers and practitioners in industry and academia have long pondered the problem of productivity and predictability. Some have proposed technical solutions such as high-level languages, object-oriented programming, artificial intelligence, expert systems, graphical programming, and a variety of CASE tools [NSB]. Others have proposed process-based improvements such as code reuse, rapid prototyping, incremental development [NSB], the Software Engineering Institute's Capability Maturity Model [SEI-CMM], and the ISO 9000 series of standards [ISO 9000-3]. These potential solutions have each had varying degrees of success, yet the productivity problem persists. This chapter will address a different solution to the problem: the only way to ensure software projects are completed within schedule and budget is to improve the skills of the people who develop the software and their managers. Rather than focusing on improving the way software is developed, we focus on raising the level of practice in the field by forming a software engineering profession.

The Concept of a Profession

The concept of a profession and a professional is carefully defined and explained in the law:

> *The term "professional" is not restricted to the traditional professions of law, medicine, and theology. It includes those professions which have a recognized status and which are based on the acquirement of professional knowledge through prolonged study*
> ***Code of Federal Regulations, Title 29, Subpart B, Section 541.300***
>
> *The first element in the requirement is that the knowledge be of an advanced type. Thus, generally speaking, it must be knowledge which cannot be attained at the high school level. Second, it must be knowledge in a field of science or learning. This serves to distinguish the professions from the mechanical arts where in some instances the knowledge is of a fairly advanced type, but not in a field of science or learning. The requisite knowledge, in the third place, must be customarily acquired by a prolonged course of specialized intellectual instruction and study.*
>
> *The typical symbol of the professional training and the best prima facie evidence of its possession is, of course, the appropriate academic degree, and in these professions an advanced academic degree is a standard (if not universal) [sic] prequisite.*
> ***Code of Federal Regulations, Title 29, Subpart B, Section 541.301***

Legal precedence provides five hallmarks of a profession [Kaner]:

- *The requirement of extensive learning and training.*
- *A code of ethics imposing standards above those normally tolerated in the marketplace.*
- *A disciplinary system for members who breach the code.*
- *A primary emphasis on social responsibility over strictly individual gain, and the corresponding duty of its members to behave as members of a disciplined and honorable profession.*
- *The prerequisite of a license prior to admission to practice.*

According to those definitions, software engineering is not a profession. The Code of Federal Regulations explains this further, stating that there are too many variations in the academic requirements and the standards for employment for software engineering to be considered a formal profession [CFR, 29-541.302h]. We believe that there would be many benefits if software engineering were a profession. Engineering professions help non-expert members of the public determine which engineers are qualified to build technical products [McConnell, p.40]. By making software engineering a profession, standards for employment would be raised by requiring all software engineers to have a college degree in computer science or software engineering. A code of ethics, similar to those in medicine and law, would be in effect for all practitioners to follow. Furthermore, breaches in the code of ethics could carry severe consequences, including

the revocation of the license to practice. This would be similar again to medicine and law where unethical conduct can result in loss of license or disbarment. Also, requiring a license prior to entering the profession raises the minimum level of knowledge in the field. Typically, a license is given to an individual who passes a standardized examination, such as the bar exam in the legal profession. Over time, this would eliminate the least skilled software engineers from the employment pool.

Becoming a professional engineer would have benefits for the individual as well. First, professional engineers would show a commitment to their future by making a large investment in their level of education. Many employers see this as a desirable trait. Also, many employers in other engineering fields require a license to be promoted into senior engineering positions. In other engineering fields, only professional engineers can consult in private practice and to serve as expert witnesses in court. Finally, as laws become stricter in their requirements for entry into a field, having a license may improve job security. [IEEE-PE]

Organizations such as the IEEE Computer Society and the ACM have been actively promoting software engineering as a profession since 1993 [SWEBOK]. However, there are opponents to making a profession out of software engineering. Many feel that software engineering is too immature a field, with a body of knowledge that is not well defined [ACM]. Others are concerned that licensing would put the licensing bodies in a position to unfairly exclude otherwise qualified developers from employment [DeMarco]. These issues, and many others, must be addressed before there will be widespread support for making software engineering.

There are two ways in which software engineering can take a major step towards becoming a formal profession: certification and licensing. Each can be mapped to the software engineering profession, and each has issues surrounding them. We begin by looking at certification.

Certification

Certification is a voluntary process that is administered by a profession [SEI-MPSE] to ensure the public will be served by professionals qualified to perform certain kinds of work [McConnell, p. 102]. The accounting profession has the most widely known form of professional certification, where practitioners are known as Certified Public Accountants (CPAs) [SEI-MPSE]. Anyone can refer to themselves as accountants and can maintain financial records for companies or prepare tax forms. Only those accountants who meet certain educational and experience requirements can refer to themselves as CPAs. Additionally, only CPAs can perform the mandatory Securities and Exchange Commission (SEC) audits of publicly traded companies in the United States [AICPA].

The SEC was established in 1934 and is appointed by the Congress. It has the authority, among other things, to establish financial accounting and reporting standards. Traditionally, they have relied on the private sector for aid in this area through the help of the Financial Accounting Standards Board (FASB), a private sector organization that makes and authorizes standard accounting rules and practices [FASB]. The American Institute of Certified Public Accountants (AICPA) also provides technical support, standards, and guidelines in conjunction with FASB. Among these standards is the set of Generally Accepted Accounting Principles (GAAP). GAAP—developed by FASB, AICPA, and the Governmental Accounting Standards Board (GASB)—specifies uniform standards and guidelines for financial accounting and reporting [AICPA]. State legislatures appoint state accounting boards and grant them the authority to certify CPAs. To become a CPA, most states require an accounting degree from a college or university. Some states also require a certain amount of professional work experience before becoming a CPA. From there, candidates must pass the Uniform CPA Examination, written and graded by AICPA. [AICPA] The accounting rules and practices, like GAAP, are among the knowledge required of CPA candidates to become certified.

There are currently a variety of certification programs in the computer and software industry today. Some are run by a professional society, such as the American Society for Quality Control (ASQC). They offer a program to become certified as a Software Quality Engineer—

described as a professional who understands the standards and principles of software quality. The requirements for this certification include eight years of professional experience, with three years in a decision making position. A bachelor's degree counts as four years of experience and a graduate degree counts as five years. The ASQC also requires proof of professionalism: either membership in a professional society, a professional engineer's license, or statements from two professional colleagues. Finally, the ASQC requires a written examination and an agreement to abide by the ASQC's code of ethics [SEI-MPSE].

The Institute for the Certification of Computing Professionals, Inc. (ICCP), a non-profit organization, offers two certification programs: Associate Computing Professional (ACP) and Certified Computing Professional (CCP). Requirements for ACP certification include receiving a score of 50% or higher two exams, one on core topics in computing, and one on the basics of a chosen programming language. Requirements for CCP certification include 48 months of experience and a score of 70% or higher on three exams, one on core topics in computing, and two from the following categories: management, procedural programming, systems development, business information systems, communications, office information systems, systems security, software engineering, and systems programming. [SEI-MPSE]

Commercial organizations also offer certification programs, although these are not usually considered to be professional certification. Novell offers a Certified Network Engineer certification [http://www.novell.com/education/cne/]. Microsoft offers Microsoft Certified Systems Engineer certification [http://www.microsoft.com/mcp/certstep/mcse.htm]. Learning Tree International [http://www.learningtree.com/us/cert/index.htm] offers certification in the following areas: PC Service and Support, LANs, WANs, Internetworking, Open Systems, Client/Server Systems, Oracle7 Database Administration, Oracle7 Application Development, Netware 3.x, Netware 4.x, UNIX Programming, UNIX Systems, C/C++ Programming, and Software Development. However, none of these have a broad enough range of knowledge to be applicable to software engineers. [SEI-MPSE]

Certification can ensure that workers have a minimum level of competency in the field. However, because certification is a voluntary, industry-based solution, there needs to be some

substantial reasons to become certified. Certification can be made a condition of employment, but this may be difficult to achieve under existing equal employment opportunity laws [SEI-MPSE]. Also, the high demand for developers in today's market would cause many to oppose any move to restrict entry of people into the market. Also some have observed that "there seems to be almost universal opposition to certification among practitioners ... partly because there is not yet any evidence that certification will solve any existing problem in the software engineering profession" [SEI-MPSE].

Society's growing dependence on software illustrates the need for some mechanism to protect the public welfare. If the industry fails to produce such mechanisms, the government would likely step in and impose such a mechanism. Each time there is a publicized software failure, especially one that results in loss of life or substantial loss of property, it becomes more likely that the government will become involved, especially by licensing software engineers [SEI-MPSE], as we now describe.

Licensing

Licensing is a mandatory process, typically administered by state legislatures [SEI-MPSE], designed to protect the public health and welfare. Consequently, it is an absolute requirement for entering into certain fields. In Minnesota, for example, professionals such as architects, professional engineers, land surveyors, landscape architects, and interior designers are required to hold a license before entering the field [Minnesota Statue, 326.02-1]. Some occupations that are generally not considered to be professions also have a licensing requirement. For instance, in California, barbers, locksmiths, private investigators, embalmers, automotive lamp and brake adjusters, professional and amateur boxers, custom upholsterers, and jockeys are among the occupations that are required to hold a license before entering the field. Each state has different lists of occupations that require a license. The common aspect is that each occupation listed affects the public. However, no occupation affects the public more than software development; yet software engineers remain unlicensed [McConnell, p. 103].

Public safety issues have motivated professional engineering since its inception. In the 1860s, American bridges were falling at a rate of more than twenty-five per year [McConnell, pp.56]. The loss of life and property associated with failures brought about more strict engineering approaches. In Canada, the Quebec City bridge collapse in 1907 had similar effects for Canadian engineering. To this day, Canadian engineering graduates receive an iron ring, made from the iron of a bridge that collapsed, to symbolize an engineer's responsibility to society and public safety [McConnell, pp.56]. In 1937, a boiler exploded in an elementary school in Texas, killing 300 children. In response to that disaster, Texas began to require the licensing of professional engineers [McConnell, p. ix].

Today, all state legislatures have laws to safeguard the public health and welfare by requiring a license to be a professional engineer. The National Council of Examiners in Engineering and Surveying (NCEES) helps write model laws for the use of the state legislatures. The state legislature typically creates a licensing board and delegates to them the authority of implementing the licensing statutes. The board has the authority to make rules and to determine the qualifications of applicants based on experience and examinations [NCBELS]. NCEES and the National Society of Professional Engineers (NSPE) provide advice in this process, thus ensuring some level of uniformity from state-to-state.

Most state boards require candidates to have a bachelor's degree from an Accreditation Board for Engineering and Technology (ABET) accredited program and pass a Fundamentals of Engineering examination. At that point, a candidate is considered to be an Engineering Intern or an Engineer In Training, depending on the state's terminology. The candidate must then complete at least four years of engineering experience, often under the guidance of another professional engineer. After the four years, the candidate must then pass the Principles and Practice of Engineering examination. Once the candidate passes that exam, he or she is considered to be a professional engineer [IEEE-PE]. The two exams are prepared by NCEES, again to ensure a degree of uniformity from state-to-state. Professional societies often aid in the preparation of an accepted body of knowledge or body of practice for use in the examination.

If software engineering were to become a licensed profession, processes similar to those found in other engineering disciplines would be needed. NCEES, with the help of the ACM and the IEEE Computer Society, would need to formulate examinations to test candidates' knowledge of the software engineering discipline. This would require a well-defined body of knowledge (discussed later in this chapter).

Laws differ from state-to-state as to what kind of work can be only be done by professional engineers. Engineers working for an industrial corporation are exempt from the licensing requirements in most states [SEI-MPSE]. For example, the State of Washington has the following provision in their law: [SEI-MPSE]

> *The work of a person rendering engineering or land surveying services to a corporation, as an employee of such corporation, when such services are rendered in carrying on the general business of the corporation and such general business does not consist, either wholly or in part, of the rendering of engineering services to the general public: Provided, that such corporation employs at least one person holding a certificate of registration under this chapter or practicing law-fully under the provisions of this chapter.*

Most states have similar exemptions. Consequently, the majority of engineers do not get licensed. Table 3 illustrates the effect of exemptions on the four main engineering fields.

Discipline	**Licensed**
Civil	44%
Mechanical	23%
Electrical	9%
Chemical	8%
All Engineers	18%

Table 3: Percentage of Licensed Engineering Graduates in the US [SEI-MPSE]

The difference between the various engineering disciplines lies in nature of the engineered product and how much impact that product has on the public safety. Products that are reproduced in large numbers can be tested before being manufactured and sold. This is the case with electrical engineers—their products are manufactured in such large quantities and they have the ability to be tested before being sold, thus reducing the risk to the public safety. Civil engineers, on the other hand, often produce unique products—like bridges and roads—that are safety-critical. These products often directly impact the public safety. These differences in the nature of their products are reflected in the percentages of licensed engineers in each discipline. [McConnell, p. 103-4]

Software engineers produce a variety of products. Some are safety-critical, like air traffic control systems and embedded software for medical devices. Others are unique, but not safety critical, as is the case with custom business software. However, software engineers also create mass-produced software, like operating systems and word processors [McConnell, p. 104]. Each type of software has a different level of impact on the public safety. If state legislatures enact similar exemptions for software engineers, different software fields and products would require a variety of different percentages of licensed software engineers.

Some would argue that licensing might not apply to software engineers since the majority of their products do not affect the public welfare. However, the government has in the past stepped in to regulate industries that present a public nuisance. This was seen, for example, with the enactment of lemon laws in the automobile industry. As businesses and private citizens come to rely more on software, the loss in productivity and general nuisance created by software that is unstable or that does not work as promised becomes an issue. The government is in a position to step in and require software engineers to become licensed in an effort to protect consumers.

Issues to Be Considered

As the examples given above for other professional fields would suggest, a certifying body must formulate a set of necessary knowledge required of software engineering professionals. As discussed in the introduction, software development knowledge can be divided into essential and accidental properties. Any effort in development of a body of knowledge should reflect the essential rather than the accidental knowledge [McConnell, p. 80]. There is no currently accepted body of knowledge for software engineering. Some have sought to provide such bodies of knowledge by forming the Software Engineering Body of Knowledge, or SWEBOK initiative [SWEBOK]. SWEBOK was started with the following goals: to characterize the contents of and to provide topical access to the Software Engineering Body of Knowledge; to promote a consistent view of software engineering worldwide; to clarify the place of, and set the boundary

of, software engineering with respect to other disciplines such as computer science, project management, computer engineering, and mathematics; and to provide a foundation for curriculum development and individual certification and licensing material [SWEBOK]. The effort has categorized knowledge and core competencies into ten areas [list from McConnell, pp. 86-7]: software requirements engineering, software design, software construction, software testing, software evolution and maintenance, software configuration management, software quality engineering, software engineering management, software engineering infrastructure, and software engineering process. This body of knowledge can be used as a basis in defining the core areas of expertise required for certification or licensing.

Also, there are certain infrastructural components that need to be addressed before software engineering is considered to be a profession. The Software Engineering Institute discusses these in the publication *A Mature Profession of Software Engineering* (CMU/SEI-96-TR-004). The components are: initial professional education, accreditation, skills development, certification, licensing (both discussed above), professional development, a code of ethics, and a professional society. We will examine each of these in turn to determine how software engineering compares to other professions, and what needs to be done in that area before software engineering is to be considered a profession.

Initial Professional Education

From the definition of a profession in the Code of Federal Regulations, a profession requires "knowledge of an advanced type in a field of science or learning customarily acquired by a prolonged course of specialized intellectual instruction and study" [CFR]. This is requirement is usually met through a college degree. Different levels of degree are required for different professions. Doctors and lawyers are required to have a graduate level degree in order to enter the field. Professional engineers, on the other hand, typically enter the field with bachelor's degrees in their particular area of interest.

Most people entering the field of software development, however, have a degree in computer science or computer engineering [SEI-MPSE]. As of 1996, no college or university in

the United States offered a bachelor's degree in software engineering, and only 20 offered a master's degree in software engineering [SEI-MPSE]. Essentially, the field is filled with individuals performing engineering tasks without formal engineering training. Furthermore, many point out that computer science and software engineering are not equivalent. In most fields, there is a definite division between science and engineering. Fred Brooks describes the difference as follows: "A scientist builds in order to learn; an engineer learns in order to build" [McConnellG]. Put another way, a scientist learns how to test hypotheses in order to extend knowledge in the field while an engineer learns how to apply well-understood knowledge in order to solve practical problems [McConnell, p. 38-9]. Even the definitions of computer science and software engineering differ. The ACM/IEEE Computer Society Task Force on the Core of Computer Science defines computer science as "the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application." Software engineering is defined as "the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates" [Fairley85 – referenced in SEI-MPSE].

Accreditation

For software engineering degree programs to become widespread, the Accreditation Board for Engineering and Technology (ABET) must develop accreditation guidelines for programs in software engineering. Accreditation assures the quality of educational programs. A variety of accreditation bodies exist in the United States to accredit entire colleges and universities, as well as to accredit individual programs [SEI-MPSE]. ABET accredits engineering programs, as well as computer science programs after merging with the Computing Sciences Accreditation Board (CSAB). ABET has historically turned to professional societies for help in the development of accreditation and curriculum guidelines [SEI-MPSE]. The ACM and the IEEE Computer Society, along with the body of knowledge produced by the SWEBOK initiative, can all be used by ABET in the formation of these guidelines.

Skills Development

There are certain skills expected of people entering a profession. These skills are used in the application of the knowledge learned during the initial professional education. The concept of skill development is seen in many professions. Apprenticeships have traditionally taught students the necessary skills to perform in a particular field. In modern times, lab courses, projects, and internships have been used to develop skills during initial professional education. Interning doctors, law clerks, and engineer-in-training programs are all examples of skill development at an early stage in a person's professional career [SEI-MPSE]. The SEI document mentioned above lists some examples of the skills necessary for software engineers: general communication skills, specialized communication skills, tool skills, procedural skills, and programming skills. In order to ensure that all software engineers have the necessary skill level in each of these areas, a list of necessary skills should be agreed upon. From there, the skills can be developed during undergraduate education and through professional development.

Professional Development

Professional development is intended to improve the skills and the currency of knowledge that a professional has after entering the profession [SEI-MPSE]. It is important for people in professions that have a rapidly changing body of knowledge to engage in periodic professional development. This is seen in medicine, when doctors study to stay abreast of new diseases, treatments, and procedures [SEI-MPSE]. This is also seen in law, where lawyers need to keep up to date on new legal precedences and laws. In the engineering field, professional development is typically on a project-by-project basis—learning new skills for a particular project rather than on long-term career development [SEI-MPSE]. All forms of professional development represent a corporate investment in their personnel, typically with the goal of improving productivity and increasing the quality of their products and services [SEI-MPSE].

Professional development is already found in wide use throughout the software industry. However, these activities are typically engaged in on an ad hoc basis—only when employers see

a need and are willing to pay the costs [SEI-MPSE]. Also, since there is no generally accepted body of knowledge for software engineers, the materials covered in professional development are not well defined. As the body of knowledge is formed, especially in conjunction with certification or licensing, guidelines for professional development will be refined.

Code of Ethics

A code of ethics is adopted by a profession to ensure that its members behave in a responsible manner. Members of the profession are bound to follow the code of ethics; those who breach the code may face disciplinary measures or loss of license. The gravity and importance of the code reflects the profession's responsibility to the public. Other professions, notably those of medicine and law, have well-established codes of ethics. The medical profession has the Oath of Hippocrates, which dates back to 400 B.C. [SEI-MPSE].

The ACM and the IEEE Computer Society Joint Task Force on Software Engineering Ethics and Professional Practices have approved a software engineering code of ethics [Ethics]. The code reflects the responsibility that all formal professions have to the public:

> *Software engineers shall commit themselves to making the analysis, specification, design, development, testing, and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety, and welfare of the public, software engineers shall adhere to the following eight principles:*
>
> *1. Public. Software engineers shall act consistently with the public interest.*
> *2. Client and employer. Software engineers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest.*
> *3. Product. Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.*
> *4. Judgment. Software engineers shall maintain integrity and independence in their professional judgment.*
> *5. Management. Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.*
> *6. Profession. Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.*
> *7. Colleagues. Software engineers shall be fair to and supportive of their colleagues.*
> *8. Self. Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.*
> **Software Engineering Code of Ethics and Professional Practice**
> [http://computer.org/computer/Code-of-Ethics.pdf]

This represents yet another step in establishing software engineering as a formal profession.

Professional Society

Professional associations are often found in mature professions. They typically work to exchange knowledge to support the profession. They are also usually involved in educational activities and aid in the defining of certification criteria. The American Medical Association and the American Bar Association are the professional societies for the medical and legal professions, respectively. The computing field has two professional societies, the Association for Computing Machinery (ACM), and the Institute of Electrical and Electronics Engineers (IEEE) Computing Society. These two societies represent the computing field at large—there is no professional society specifically for software engineers. However, the two computing professional societies provide services for software engineers, so it is unlikely that another professional society dedicated solely to software engineers is necessary [SEI-MPSE].

Computer Malpractice

There are other issues surrounding the creation of a software engineering profession beyond the infrastructural issues mentioned above. Perhaps the most important of these issues is the concept of computer malpractice. Malpractice is the professional form of negligence. Negligence is defined as "the omission of the care usual under the circumstances. A specialist is bound to higher skill and diligence in his specialty than one who is not a specialist, and liability for negligence varies accordingly" [Webster]. Anyone can be sued for ordinary negligence. In such a situation, your actions are compared to those of any reasonable person under the given circumstances. Only a professional can be sued for malpractice. If you are sued for malpractice, your actions are compared to those of any reasonable member of your profession under the given circumstances. Malpractice suits are more serious than suits for ordinary negligence or for breach of contract. While a non-professional service provider's contract may limit the damages that must be paid to the customer in the case of negligence, such limits are rejected in

malpractice suits because they are deemed to violate public policy. Consequently, malpractice plaintiffs are more likely to collect punitive damages when cases are ruled in their favor. [Kaner]

Courts in the past have been unwilling to allow computer malpractice lawsuits because software engineering is still not viewed as a formal profession [Kaner]. Forming a software engineering profession would change this. Software engineers that are liable for malpractice would require some form of malpractice insurance, typically on the part of the employing company [McConnell, p.107]. In order for this to happen, the insurance company would need some way of limiting its risks. Essentially, the insurance company must trust the software engineer seeking insurance, as well as the software product being produced.

Industry Opinions

The actions that developers choose to take will ultimately decide the direction that software engineering will go. Therefore, it is important to understand the positions of industry experts. Influential forces in the software industry have been divided in their endeavors to raise the quality of software. Some believe, often vehemently, that allowing the true professionals of the field to step forward and distinguish themselves from ordinary computer programmers will raise the standards for quality in software. The evolution of software engineering into a profession will ultimately happen, aided by or in spite of our efforts. There are also those who argue that defining a profession will only serve to inhibit access to the vast knowledge and varied areas that make up the current spectrum of computer science.

The majority of those in favor of improving the status of software engineering, either by licensing or certification, are committed to the concept that a process that separates out the most able software developers will improve the level of quality of software being developed. They agree that licensing or certification will help to achieve a higher level of competency in the field, thus protecting the public from bad developers, while contributing to a standardization of practices, creating more dependable software, and improving the education and skills of software developers through revisions and standardizations of college curricula [Hawthorne]. The major

concerns of proponents, such as John Speed, involve creating an environment that will foster the growth of software engineering into a mature engineering profession [Speed]. They are trying to solve what is commonly referred to as a minimax problem [Brooks]. The main focus is on obtaining the maximum amount of quality in software and integrity in its designers, while having a minimal amount of negative effects on the software industry. Questions such as "What areas of the software industry require licensing?" "What percent of developers should be licensed?" and "What criteria does a software engineer have to meet?" are the problems that these people now face. Looking for a starting point, many argue that a logical plan of attack would be to start small and then gradually increase the domain of the professional software engineer. The design of software for safety critical systems has provided a potential starting point for professional software engineers. The situation is optimal because the software being developed will have a large impact on public safety. Industry experts such as Barry Boehm argue that success will provide a foundation for further growth, and failure will point out our mistakes, but taking no action may force us into use a poorly conceived "quick-fix" [Boehm]. But even this optimal situation has its drawbacks. The ACM disapproves of licensing at this time because "the SWEBOK effort ... will have little relevance for safety-critical systems, and it dangerously excludes the most important knowledge required to build these systems." [ACM]. No situation in software engineering is perfect, so licensing and certification proponents must find a pragmatic solution.

Some opponents of licensing and certification are not so willing to accept the idea that defining a profession will improve the quality of software engineering or even computing as a whole. Other opponents believe that the licensing and certification will bring improvement to the quality of software, but the effort that these steps entail far outweigh the benefits. W. A. Wulf argues that defining an engineering profession will only put limits on us as computer scientists, saying:

> *Defining ourselves by what we are not is common in CS and it has, in my opinion, damaged us. It has made us inhospitable to pragmatically and intellectually fertile areas that should be part of our discipline. Indeed, beyond being inhospitable, we have expelled whole areas - numerical methods, libraries, and MIS, for example. If we continue this expulsion of the practical we will leave the field a barren husk.* [Wulf]

He also explains that being a software engineer means that we will be engineers in all respects, including our curriculum [Wulf]. He points out that important computer science courses, such as discrete mathematics, may not be as important to engineers and may be pushed back in computer science curricula to make way for fundamental engineering courses [Wulf]. Ken Kennedy, a faculty member of Rice University, questions whether schools will be able to be innovative if required to follow a standard engineering curriculum [Kennedy]. Illustrating these effects from first hand experience he says:

> *At Rice, a secondary byproduct of accreditation has been increasing pressure to require more hours in engineering, and consequently fewer hours in disciplines outside science and engineering. As a result, all too many of our engineering graduates have difficulty communicating their ideas and are almost completely out of touch with the intellectual, social, political, and ethical issues that define our society.* [Kennedy]

The effects of creating a profession of software engineering discussed above have not only shown how a profession limits computer science, but also begs the question "Is it worth it?". Even for those who do not agree that these actions will have such a detrimental effect on computer science as a whole, there are those who think that licensure or certification will simply not be worth the effort. Fred Brooks, Jim Gray, and Ken Kennedy were among those who wrote position papers on this subject and were not particularly impressed by the current results of the licensing movement in Texas [Brooks, Gray, Kennedy]. Some of those in line with this opinion feel that licensure or certification does not effectively address the issues relating to software quality [Gray]. Tom DeMarco explains that he "wants instead to see the community 'switch gears' and focus on the 'essential' problems of software engineering", as opposed to the 'accidental' ones [Easterbrook]. Or as Ken Kennedy said, "Rather than focusing on licensing, we should instead be focusing on understanding and promulgating good curricula in computer science. Once we have achieved that, we should reconsider licensing." [Kennedy]. Some feel that the computer science is not ready for licensing or certification; others feel it never will be. Regardless of their differences, opponents of licensure and certification have valid logical reasons not to support this route toward a profession of software engineering at this time.

Proposal

It is our opinion that licensing offers greater promise than certification for improving the state of the software industry. It is unlikely that the industry will begin voluntary certification, especially considering the high demand for software developers in today's market. It is more likely that the government will intervene, as it has with other engineering disciplines, and establish professional software engineering licensure. In 1998, the Texas Board of Professional Engineers began the process of making software engineering a licensable engineering discipline. This is important because Texas has a history of introducing changes before nationwide acceptance [McConnell, p.104-5]. We will examine what is going on in Texas as an example of how to begin the licensing process.

The Texas Board of Professional Engineers is working with the ACM and the IEEE Computing Society to produce a Principles of Practice Examination for Software Engineering. The examination has not yet been written and no date has been set for the first exam. The three bodies are also considering drafting a Fundamentals of Engineering Examination geared towards software engineers. Until the exam has been written, it is possible to request an exam waiver. The requirements for an exam waiver are:

- an engineering degree from a college or university accredited by ABET and twelve years of engineering experience, or
- a non-accredited degree and sixteen years of engineering experience, or
- a Ph.D. in engineering from a college or university with an undergraduate or master's program that is accredited by ABET, and to have taught in an ABET accredited program for at least six years, or to have at least six years of combined experience in the industry and teaching in an ABET accredited program

Also, all engineers applying for exam waivers are required to have a total of nine references, with five of them coming from licensed engineers [TBPE].

As is the case with other engineering disciplines, the state legislatures would give authority to the current licensing boards to begin licensing software engineers. As the Texas situation indicated, the ACM and IEEE Computer Society are still doing work to create the

necessary examinations for software engineering licensure. Until those exams are available, states can follow a similar course of action to those in Texas—allowing an exam waiver that is sufficiently exclusive as to not lessen the value of being a licensed software engineer. A state-based movement to make software engineering a profession would be supported by curriculum development in public colleges and universities that operate under state mandate. The ACM, the IEEE Computer Society, and the Software Engineering Institute would support this by supplying curricular recommendations and sample undergraduate programs.

Licensing has been applied to many other engineering disciplines. It has been a success across the board—there are no engineering fields that began licensure and then stopped it because it did not work. That alone is sufficient cause to consider the licensure of software engineering. The benefits listed at length above would only serve to improve the current state of the software industry. However, licensure is not without its faults. Licensed software engineers are not necessarily good developers—they just meet the requirements for practice set out by the licensing board. Licensing acts as a filter, generally improving the labor pool by excluding the worst in the industry [McConnell, p.110]. It is an imperfect mechanism; some good engineers may be excluded from the labor pool, just as some bad engineers may be included. However, without licensing, the public is at the mercy of all software engineers, both good and bad. Licensing eliminates the worst of the group and limits the choice to the better engineers.

Forming a software engineering profession helps to improve trust in software products. A software company that hires professional software engineers can say that they are hiring the people with the best credentials [McConnell, p. 106]. The company can then argue that their software is more trustworthy than software developed by a company that does not hire professional engineers. Also, licensed engineers in a company will have a greater say in products that they might be held liable for [McConnell, p. 107]. The potential repercussions for poorly designed or implemented software raise the stakes for the engineers involved. A professional software engineer is less likely to sign off on a poor design or on a product that is not yet ready for the market.

Having professional software engineers on staff can help in other aspects of corporate operations. For instance, when seeking venture capital, the number of professional engineers on a company's staff can act as a discriminant, making that company more attractive to potential investors. In the next chapter we will examine product certification. We feel that professional software engineers are very important for a credible product certifying process. These two topics together are essential in developing trust in software components. If software is not shared between companies, there is no need for one company to know that much about another company's software. However, when software is exchanged and shared between companies—when a supply chain is formed between software companies—it becomes essential that there are standards for the design, development, and testing of the software produced. To that end, software components need to be certified, and the best people to perform such certification would be professional software engineers.

References

[ACM] ACM: Position of the Licensing of Software Engineers. <http://www.acm.org/serving/se_policy/papers.html>.

[AICPA] The American Institute of Certified Public Accountants Website. <http://www.aicpa.org/>.

[Boehm] ACM: Position of the Licensing of Software Engineers. <http://www.acm.org/serving/se_policy/papers.html#boehm>.

[Brooks] ACM: Position of the Licensing of Software Engineers. <http://www.acm.org/serving/se_policy/papers.html#brooks>.

[Busch] Busch Consulting, Inc. Virus Protection. <http://www.buschconsulting.com/virus.html>

[CFR] Code of Federal Regulations. < http://www.access.gpo.gov/nara/cfr/cfr-table-search.html>.

[CoxNSBR] Cox, Brad. "No Silver Bullet Reconsidered." <http://www.virtualschool.edu/cox/AmProTTEF.html>.

[DeMarco] DeMarco, Tom. "DeMarco on the Certification and Licensing of Software Engineers." <http://www.systemsguild.com/GuildSite/TDM/certification.html>.

[Easterbrook] Easterbrook, Steve. "DeMarco: Process Considered Harmful?" <http://www.cis.cs.tu-berlin.de/~icsewow/v2n2/v2n2-2.html>.

[Ethics] Software Engineering Code of Ethics. <http://computer.org/computer/connection/CSNews_2.htm>.

[FASB] Federal Accounting Standards Board Website. <http://www.fasb.org/>.

[Gray] ACM: Position of the Licensing of Software Engineers. <http://www.acm.org/serving/se_policy/papers.html#gray>.

[Hawthorne] ACM: Position of the Licensing of Software Engineers. <http://www.acm.org/serving/se_policy/papers.html#hawthorne>.

[IEEE-PE] "Become a Professional Engineer." IEEE Website. <http://www.ieee.org/organizations/eab/pelicens.html>.

[ISO 9000-3] Kehoe, Raymond, et al. *ISO 9000-3: A Tool for Software Product and Process Improvement*. New York: Springer, 1996.

[Jones] Jones, Capers. *Patterns of Software Failure and Success*. Referenced by [Yourdon] (See Chapter 1 References).

[Kaner] Kaner, Cem. "Computer Malpractice." <http://www.badsoftware.com/malprac.htm>.

[Kennedy] ACM: Position of the Licensing of Software Engineers. <http://www.acm.org/serving/se_policy/papers.html#kennedy>.

[McConnell] McConnell, Steve. *After the Gold Rush: Creating a True Profession of Software Engineering*. Redmond, Washington: Microsoft Press, 1999.

[McConnellG] McConnell, Steve. "Software Engineering Is Not Computer Science." <http://www.gamasutra.com/features/19991216/mcconnell_pfv.htm>.

[NCBELS] North Carolina Board of Engineers and Surveyors Website. <http://www.ncbels.org/>.

[NSB] Brooks, Frederick P., Jr. "No Silver Bullet." *The Mythical Man-Month* (Anniversary Edition). Reading, MA: Addison-Wesley, 1995.

[SEI-CMM] Paulk, Mark C., et al. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, Massachusetts: Addison-Wesley, 1994.

[SEI-MPSE] Ford, Gary, et al. "A Mature Profession of Software Engineering." CMU/SEI-96-TR-004. <http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.004.html>.

[Speed] Heineman, George T., et al. *Component-Based Software Engineering*. Addison-Wesley, 2001.

[SWEBOK] The Software Engineering Body of Knowledge (SWEBOK) Website. <http://www.swebok.org/>.

[TBPE] "Board Establishes Software Engineering Discipline." Texas Board of Professional Engineers Website. <http://www.tbpe.state.tx.us/sofupdt.htm>.

[Webster] Webster's Revised Unabridged Dictionary, 1996, 1998. <http://www.dictionary.com/>.

[Wulf] ACM: Position of the Licensing of Software Engineers. <http://www.acm.org/serving/se_policy/papers.html#wulf>.

CHAPTER 3

Product Certification

Introduction

When software developers intend to use components in their product, it is necessary that they trust the components that they are going to use. They need high quality components and they need to trust that those components will perform as specified. This chapter will examine product certification as a way of improving that trust. The IEEE Standard Glossary of Software Engineering Terminology defines certification as a written guarantee, a formal demonstration, or the process of confirming "that a system or component complies with its specified requirements and is acceptable for operational use" [CBSE-UL]. The fact that certified components demonstrably meet pre-determined and well-established criteria reduces the risk of system failure and increases the likelihood that the system will comply with specified design standards [CBSE-UL]. Some would argue that this is not technically feasible. It is difficult to prove that a component is acceptable for operational use. Similarly, verifying that a component meets pre-determined and well-established criteria raises some issues—namely, defining the criteria and performing the verification. Bearing this in mind, for the purposes of this discussion, we will assume that certification is a feasible goal.

Electrical devices have been certified for more than a century by Underwriters Laboratories (UL) [CBSE-UL]. Examine almost any electrical device and you will find a UL marking on it, signifying that the device has been certified as safe to use. UL's history can be traced back to the 1893 Chicago World's Fair. A large exhibit on the virtues of electrical power was set up. However, in setting up the exhibit, complicated and untested wiring was placed near flammable materials. Since the wiring was untested and seemed unsafe, the fire insurance underwriters were not willing to risk insuring the exhibit. This almost stopped the exhibition

entirely. However, the underwriters contacted noted Boston electrician W. H. Merrill and hired him to come to Chicago and examine the wiring. Merrill reviewed the wiring, noted the safeguards present, and the underwriters extended fire insurance to the exhibit. Merrill's work at the World's Fair exposed him to many electrical producers who later contacted him to verify the safety of their products. With their backing, as well as support from fire insurance underwriters, Merrill started Underwriters Laboratories in 1894 [NFPA]. It is doubtful that this kind of action would take place today with software—for example, the organizers of COMDEX are not likely to threaten to cancel an exhibition due to uncertified software. However, the need for insurance is a powerful motivating force. As seen in the last chapter, a move to make software engineering a formal profession would bring about the need for malpractice insurance. That would mean that insurance companies would need some way of mitigating their risk in insuring software. Certification of components would be a reasonable way of accomplishing this.

Background and Certification Examples

Certification has two distinct facets: technical claims are made about a product and an unbiased authority stands behind those claims [SEI-TCCBSE]. This illustrates the two purposes of component certification. First, the producer establishes facts about the component. Second, the authority establishes trust in the facts by standing behind them [SEI-TCCBSE]. Obviously, the more trusted and impartial the certifying authority is, the more trusted their judgment about the component would be.

As stated earlier, electrical components have been certified for over a century by UL. This provides objective evidence from an impartial third-party that a product meets safety standards specified through industry consensus, regulation, or law. The National Electrical Code [CSSInfo] contains safety requirements for electrical components to be used in buildings and requires third party certification. If the electrical component in question contains software, that software must meet the electrical code as well. When the component is certified it receives a UL

marking, providing confidence that the component has met the code's safety standards [CBSE-UL].

Software certification has been around for quite some time as well. The Department of Defense developed a set of standards for operating systems called the Trusted Computer System Evaluation Criteria (TCSEC) [TCSEC]. TCSEC is a set of criteria with the following purpose:

- *To provide a standard to manufacturers as to what security features to build into their new and planned commercial products in order to provide widely available systems that satisfy trust requirements (with particular emphasis on preventing the disclosure of data) for sensitive applications.*
- *To provide DoD components with a metric with which to evaluate the degree of trust that can be placed in computer systems for the secure processing of classified and other sensitive information.*
- *To provide a basis for specifying security requirements in acquisition specifications.* ***[TCSEC]***

Evaluated systems are ranked according to their security capabilities. There are four security classes: D, C, B, and A (in increasing order). Each class has four major criteria sets: Security Policy, Accountability, Assurance, and Documentation [TCSEC]. The details of the evaluation procedure and of the different security classes are unimportant for this discussion, but there are some important issues to notice. First, note the purpose of the TCSEC—it provides manufacturers a standard to design to and it provides the DoD a standard to evaluate new products with, as well as to specify requirements for new purchases. Second, note that the criteria is not limited to functionality; documentation is required as well.

Both of these examples show some important aspects of certification. The need for standards is illustrated: the producer needs standards to design to and the certifying body needs to use those standards to ensure that the product being certified is acceptable. Also, the certification provides a manner of evaluating a product's fitness for a given task. For example, an electrical component may only be certified for indoor use, or the operating system certified by TCSEC may not be secure enough for certain high-security operations. Finally, the certification provides a way for specifying requirements when a consumer is making a purchasing decision.

Note also that certification is not merely testing a product. TCSEC requires adequate documentation in the form of user guides, manuals, and test and design documentation [TCSEC].

The ANSI/UL 1998 Standard for Software in Programmable Components addresses the entire life-cycle of a product, emphasizing risk-based analysis and design, consideration of provisions for hardware malfunctions, test planning and coverage, usability considerations, comprehensive documentation, processes for handling software changes, qualifications for off-the-shelf software, labeling that uniquely identifies the specifics of the product interface, the intended hardware platform, and the intended software configuration [CBSE-UL].

Some industry programs and proposals are called certification, but go no further than testing. For example, Microsoft has started the Certified for Windows Program [MSCert1]. Microsoft, its customers, and third party developers created guidelines for developing manageable and reliable applications. There are two versions of the specification: a core specification for desktop applications and a comprehensive specification for distributed applications. Software that complies with either version is eligible for the "Certified for Windows Logo." Microsoft provides test plans, test frameworks, and test tools for a variety of platforms, as well as general functionality and stability test procedures [MsCert2]. The software producer evaluates how well the application meets the specifications and then submits the software, along with a fee, to VeriTest, an independent testing lab located in Santa Monica, Paris, and Tokyo [MSCert3]. VeriTest verifies that the application meets the application specification mentioned above. They perform over 700 pages of tests, including 32-bit capability, core application stability, long filename support, proper install and uninstall procedures, and user interface fundamentals [MSCert4]. This is really a form of platform testing rather than certification. An application is submitted with the claim that it runs under Windows 2000 and VeriTest certifies that this is true. Unfortunately, this does not map well to components. Typically, it is not known ahead of time what a component will be used for. Also, it is not usually known what type of platform or configuration the component will be run on—this is especially an issue with Java components that could run on any of a number of hardware platforms. While the testing used in the Certified for Windows Program is valuable, it fails to cover many certification issues necessary for components.

Another form of certification, proposed by Jeffrey Voas, is a usage-based certification process [Voas00]. A stable version of the product is supplied to a Software Certification Laboratory (SCL). Linked to the software are "residual testing capabilities" which collects data and reports back to the SCL any failures that occur. The software is then released to a small group of users who use the software normally. The testing capabilities linked in gather information and occasionally report back to the SCL about any failures that occur. "When testing collects enough data from the field to affirm that a component works properly in a particular market sector, the SCL will provide software warranties specific to that sector. For example, an SCL warranty might read as 'Software product X is warranted to perform with a reliability of 99.9 in the Windows NT environment'" [Voas00]. As was the case with the Microsoft Certified for Windows Program, this is really another form of testing. The usage-based certification program resembles a beta-testing environment. It provides a convenient way for discovering otherwise unknown defects, but it does not really certify the component. Again, this proposal fails to cover many of the process and documentation aspects necessary for the certification of software components.

Component Certification

Most of the discussion to this point focused on software in general. We now turn our attention to software components. Specifically, we examine second party testing for a company's use of a component and third party certification for the general use of a component. Central to all forms of component certification is the concept of compositional reasoning [SEI-TCCBSE]. The idea behind compositional reasoning is that there is a causal link between the properties of a component and the properties of a complete system that uses that component. Those properties that have the greatest effect on the end system are the properties that need to be certified in the component. The economic value of component certification comes from the strength of the compositional reasoning in predicting end-system properties [SEI-TCCBSE]. Furthermore, strong compositional reasoning tells us which properties of components are necessary to certify.

We begin by looking at second party testing. In this situation, the purchaser must determine that a component is sufficient for the task at hand. Essentially, the purchaser is certifying that a component is acceptable for use in his or her end product. Consequently, a number of issues must be examined: that the component meets the needs of the end system, that the component is of sufficient quality and reliability, the impact that the component will have on the end system, and whether the end system will tolerate the component [Voas98]. To accomplish this, the purchaser must first identify which properties of the component are important to certify—memory usage, latency, performance, stability, and so on. From there, the purchaser must test the component to see how it meets with his or her requirements. There are a number of testing methods that the purchaser can perform to verify the component's fitness [test list and issues taken from Voas98]. First, black box testing can be performed, determining whether the component is of high enough quality. This requires documentation on how the component is to be used and what functions are available as well as a driver program to generate input and test output. Since the component is not certified, the purchaser cannot determine how much testing was done before it was released. So, by performing his or her own range of tests, the purchaser is able to gauge how well the component meets its documentation and specifications. Bearing that in mind, it is difficult to thoroughly test a component, especially one of any significant size. The cost of testing may be significant, especially considering that the purchaser must generate test cases, an input generator, and an program to test the output. However, some would argue that the cost of testing is minor compared to the cost and time savings of using the component rather than writing new code [Voas98]. From there, system-level fault injection can be used to determine how well the system would tolerate a failing component. Input and output would be perturbed to generate errors and see how the underlying system handles those errors. Finally, operational system testing would take place, testing how well the system tolerates a functioning component. This ensures that the system tolerates the component and that it is a good fit. After performing the three types of testing outlined above, the purchaser would then decide whether or not to certify the component for use in the end system. The purchaser would only certify a component that fits the requirements of and has a positive impact on the end system. However,

he or she can choose to certify a component that is not of high enough quality if the failures are infrequent or can be handled [Voas98].

Perhaps the use of the term certification in this scenario is a misnomer. What is actually taking place is integration testing. The producer checks how well the component fits into the end system and then checks how well the system behaves with the new component. It is not as wide reaching as, for example, the ANSI/UL 1998 certification mentioned above. But, for the purposes of a developer creating a component-based application, it may be sufficient. The value for the overall component marketplace is limited. While it may be possible to state that component X has been certified for use by company Y, the component is still not certified for use in all systems. Technical claims about the component can be supported by this scenario, but they are supported by the customer, not by an impartial third party, as is the case with UL-type certification mentioned above. The additional costs to the component purchaser may be discouraging—test scenarios and fault-injection means more time and money spent. Similarly, testing is never exhaustive—some problems can always slip by. The true value of this form of second party testing would come when many consumers certify a component for use in their systems. If a sufficiently large number of component consumers certify a given component, it may not be necessary for another consumer to engage in such an in-depth evaluation of the component. Similarly, if a sufficiently large group of component consumers feel the need for some form of certification, it would likely be more cost effective to join forces and form an independent body to perform third party certification on the candidate components.

Third party certification fits the UL-type model discussed above. A laboratory, independent of the producer, supplier, seller, buyer, or government would provide objective evidence as to which components have been produced according to specifications, standards, and any other criteria deemed necessary [CBSE-UL]. As stated above, certification requires standards for the producer to design to and for the certifying body to certify against. Certification also requires extensive specifications and documentation about the component. This aids in the certification process by defining intended use, expected behavior, interface information, and so on. Additionally, the specification and documentation helps in identifying the important properties

for compositional reasoning, thus providing the component consumers the ability to predict the properties of their end system.

Component producers would pay a fee to the certifying body to examine their component. The component would be submitted, along with all necessary specifications and documentation, and the certifying body would verify that it complies with the appropriate standards. Testing may be performed, following a regimen similar to the one discussed for second party testing. However, certification is not limited to testing the component. As we saw above, the ANSI/UL 1998 Standard for Software in Programmable Components contains provisions that address the entire product life-cycle [CBSE-UL]. Similar provisions would be necessary for a successful certification scheme.

We saw earlier examples of third party certification of software systems: the DoD TCSEC standard and the Microsoft Certified for Windows. As a means of example, we will examine the component certification program offered by the Compuware Corporation on behalf of the Component Vendor Consortium (CVC). The CVC Certification requirements [CVC-CERT] state what is necessary for certification. However, it is mostly limited to memory issues. Compuware is responsible for ensuring that components have no unexplained memory corruption errors, no memory leaks, and no unhandled errors or exceptions. The vendor links in libraries that monitor resource usage and source code line coverage. From there, the vendor performs multiple runs on the component until 80% coverage is reached. The resulting log files, and a $160 fee, are submitted to Compuware for analysis and review. If there are no "unexplained" memory corruption errors, no memory leaks in the component or its dependencies, and 80% of the source code was covered, the component will be certified [CVC-CERT]. The real value of such certification is questionable. Obviously memory corruption and leaks account for some portion of component failures. However, there is so much overlooked by this scheme that it is doubtful to be of much use. Furthermore, Compuware admits that there are things missing from the final certification. They do not attempt to enforce versioning—vendors may attempt to mislead the certifying body by submitting another version of their component. There is no checking for logic errors. The component is not tested under multiple environments. Also, there is no way to

ensure that all errors are handled or that all control paths are tested [CVC-CERT]. Certainly a more thorough scheme is necessary for certification to become a viable course of action.

Legal Issues

Software has often been treated as a special case because of its unique production process. This special status severely limits certification and is an issue that must be resolved in order for certification to succeed. Software is treated as an "inherently buggy" product [UCITA], which has traditionally allowed software companies to avoid responsibility for any problems with their code. Software components will not be trusted if its producer sees no difference between a fully and partially functional product. This begs the question, "Why bother testing and certifying a piece of software, if the defects may not be fixed?" Liability regarding software creation and certification is another issue that must be addressed before trustable code can be produced through certification. Software components can be trusted only after software developers take responsibility for faults in their product.

Software is often a complex product. Engineers are trained to reduce complexity by dividing a complicated system into more manageable components. More complex products can then be built by relying on generic off-the-shelf components. Currently, software developers cannot do this. They are also well trained in modularizing complex systems, but they cannot necessarily trust off-the-shelf components.

To trust a component, without thoroughly testing it, the consumer needs some form of approval from a trusted authority in that field. Currently, there is no such body in the field of software. The developer must rely on the claims of the component producer. Standards can give some amount of trust to a component by ensuring it does or does not do certain things.

Defining boundaries that software is expected to adhere to is a necessary first step towards producing trustable code, but it also entails added liability for software developers. Since certification is not mandatory, software companies must voluntarily accept the chance of being sued for a defective product. In return for this, a software company would produce a better product and gain the trust of its consumers. Theoretically, the overall quality of software would

improve by weeding out developers who sell faulty products. However, this is not guaranteed. Good software developers sometimes make mistakes or a problem with the software may arise out of environments that did not exist when it was released. Even software leaders, like Microsoft, who have devoted extensive resources to software development of and with components, make software that contains defects. Current legal movements, like UCITA, indicate mixed progress in software liability. With the intent of making developers responsible for their product and improving the quality of software, members of the industry are proposing UCITA. However, this act has many opponents because of the large loopholes that it leaves for software developers to completely avoid responsibility. Another liability problem is deciding who is at fault when a defect is found in certified software. The certifying body is responsible to test the product, but at what point are defects considered to be outside of the certifier's responsibility? Accepting product liability entails potentially large risks and adds new problems for software companies. However, this is needed to ensure trust in software components. Finding the best way of accomplishing this will be a benefit to the software industry.

There is legal precedent to hold a certifying body liable for a faulty product. If a claim is made that the public should trust a product because a certain body tested it, every assurance should be made that that claim is true. Otherwise the certifying body may be sued [Kaner]. In the case of Hanberry v. Hearst Corporation, a consumer sued *Good Housekeeping* for negligent endorsement of a shoe that was defectively designed [Kaner]. The court ruled that when an association endorses a product for economic gain and encourages the public to buy it, the association could be held liable if a consumer buys the product based on the endorsement and is then injured because it is defective [HKLaw]. In the case of Hempstead v. General Fire Extinguisher Corporation, a worker that was injured by a fire extinguisher that exploded sued Underwriters Laboratories for negligence in its testing and approving of the product [Kaner]. The court held that liability for a faulty product did not rest just with the manufacturers and sellers of the product. It ruled that UL knew, or should have known, of safety precautions necessary to avoid such accidents [DKSLaw].

Proposal

As more software companies use components in their products, the need for some form of certification will grow. It should be expected that companies using components would engage in second party testing to certify that a component is acceptable for use. In time, some duplication of effort would be found—many companies would be evaluating the same component to similar yet independently developed standards. At some point, a critical mass will be reached, and it will be more cost effective for an independent third party component certification laboratory to be formed.

The formation of an independent third party certifying body would require standards agreed upon by the industry. These would be used for developers to design to and for the certifying body to certify against. These standards need to be sufficiently wide reaching to make certification worthwhile. This may include standard interfaces for a given type of component. The issue of standard interfaces is seen in the electrical engineering field. For example, all resistors have an agreed upon color-coding scheme for easy identification. The Socket 7 processor socket designed by Intel has a well-defined interface allowing other chip producers like AMD or Cyrix to design processors that work on those motherboards. Other properties of components that will be certified will likely come from research in the area of compositional reasoning. Component properties that most affect the properties of the end product need to be identified. Other issues need to be taken into consideration as well. We saw in the ANSI/UL 1998 Standard for Programmable Components consideration given to the entire product life-cycle: proper emphasis on risk-based analysis and design, consideration of provisions for hardware malfunctions, test planning and coverage, usability considerations, comprehensive documentation, processes for handling software changes, qualifications for off-the-shelf software, labeling that uniquely identifies the specifics of the product interface, the intended hardware platform, and the intended software configuration [CBSE-UL]. Similar issues not based solely on the performance of the component must be taken in to account when forming the standards for certification.

Third party certification would have many benefits. A certified component will have met rigorous standards agreed upon by the industry. A certain level of stability testing will be

necessary; this will help ensure the safety and reliability of the component. As certification becomes more widespread, name recognition of the certifying body may provide a competitive advantage for certified components over non-certified components. Also, the presence of documentation and specifications that the component was certified against will reduce the performance uncertainty—the proper use guidelines and performance specifications will allow component consumers to make educated decisions based on standard information.

Certification has drawbacks. For example, thoroughly testing a component is very difficult. It is quite possible that components will be certified even though they still have defects in them. However, if issues similar to the ones mentioned above for ANSI/UL 1998 certification are used, the presence of these defects may be diminished. Also, certification is not instantaneous. It requires time for the certifying body to examine the component and make a decision as to whether to certify or not. This translates into a longer time to market that in a marketplace without certification. Similarly, in new markets it may not be beneficial for components to be certified. For instance, the lag time to perform certification on a component produced by a company trying to enter the market may give other companies time to generate competitive components. Also, as is often seen in today's software industry, companies creating new software are often bought out and their products are integrated into a larger product. There are also legal issues to be taken into consideration, as seen above. Finally, there are technical issues. For instance, it is unclear whether patches to existing software components need to be certified.

Certification also highlights the need for professional software engineers. In the case of second party testing, professional software engineers are best suited to determine the fitness of a component. In the case of third party certification, there would be a need for qualified individuals to formulate the standards for certification as well as to certify the components.

A complimentary approach to product certification is to certify the processes used to produce the artifacts [SEI-TCCBSE]. Conway's law states, "The organization of the software and the organization of the software team will be congruent" [Jargon File]. Again, we refer back to the ANSI/UL 1998 standard—many of the issues discussed in the standard apply solely to the

process of developing the software. The next chapter will discuss the issues surrounding process certification.

References

[CBSE-UL] Heineman, George T., et al. *Component-Based Software Engineering.* Addison-Wesley, 2001.

[CSSInfo] Website for the 1999 National Electric Code. <http://www.cssinfo.com/1999NEC.html>.

[CVC-CERT] Compuware NuMega, Developer Support Group. "CVC Certification Requirements." <http://www.components.org/cvccert.doc>.

[DKSLaw] MacDonnell, Janet L. "Guilt by Association: Product Liability for Non-Manufacturers." <http://www.dkslaw.com/papers/toxicpapers/guiltjm.html>.

[HKLaw] *Product Liability Newsletter*, June 1999, Volume 1, Issue 2. <http://www.hklaw.com/newsletters.asp?ID=55>.

[Jargon File] The Jargon File. <http://www.tuxedo.org/~esr/jargon/html/The-Jargon-Lexicon-framed.html>.

[Kaner] Kaner, Cem. "Software Negligence and Testing Coverage." <http://www.kaner.com/coverage.htm>.

[MSCert1] Microsoft Corporation Website. <http://msdn.microsoft.com/certification/default.asp>.

[MSCert2] Microsoft Corporation Website. <http://msdn.microsoft.com/certification/download.asp>

[MSCert3] Microsoft Corporation Website. <http://msdn.microsoft.com/certification/test.asp>.

[MSCert4] Microsoft Corporation Website. <http://www.microsoft.com/windows2000/upgrade/compat/certified.asp>.

[NFPA] Grant, Casey Cavanaugh, "The Birth of NFPA." <http://www.nfpa.org/About_NFPA/An_Overview_of_NFPA/The_Birth_of_NFPA/the_birth_of_nfpa.html>.

[SEI-TCCBSE] Bachman, Felix, et al. "Technical Concepts of Component-Based Software Engineering." CMU/SEI-2000-TR008. <http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr008.pdf>.

[TCSEC] Department of Defense Trusted Computer System Evaluation Criteria, DOD 5200.28-STD. <http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html>.

[UCITA] Uniform Computer Information Transactions Act. <http://www.law.upenn.edu/bll/ulc/ucita/ucitanc.htm>.

[Voas00] Voas, Jeffrey M. "Developing a Usage-Based Software Certification Process." *Computer*, August 2000, Volume 33, Number 8.

[Voas98] Voas, Jeffrey M. "Certifying Off-the-Shelf Software Components." <http://www.cnsoftware.org/studies/trustworthy/Voas-98-CertCOTS.pdf>.

CHAPTER 4

Process Assessment

Introduction

As we saw in the previous chapter, the certification of a product relies on more than simply testing and evaluating a piece of software. Effective certification examines the processes used to develop software as well. Reports of software management problems are ubiquitous in today's industry. Typically, software producers would prefer to deliver software with defects than to delay release [CMM, p.4]. The General Accounting Office (GAO) released a report in 1993 outlining some of the problems that the government has faced in procuring software. The report states "we have repeatedly reported on cost rising by millions of dollars, schedule delays of not months but years, and multi-billion-dollar systems that don't perform as envisioned" [GAO-93-13, CMM p. 4]. The report attributes these problems to the fact that knowledge of the software development process is not keeping pace with the ever-increasing size and complexity of software products [CMM, p.4].

There exists a gap between the best practice and the average current practice in software development in the industry. A Department of Defense task force set up to investigate the "software crisis" went so far as to say that "few fields have so large a gap between best current practice and average current practice" [CMM, p.4]. Studies show that about 75% of software project teams begin their projects by coding rather than planning and designing [McConnell, p.11]. This is important because 40 to 80% of development projects' budgets are dedicated to fixing defects introduced earlier on in the development process [McConnell, p.11]. Studies have also shown that projects focusing on short schedules are more likely to have budget and schedule overruns. Projects focusing on low defect counts had the best schedule and productivity [Jones, McConnell p.16]. It is common practice in the industry to try to trade quality

for time and cost. Considering that projects that remove 95% of defects before release are the most productive [Jones, McConnell p.16], this tradeoff might not exist.

All of the issues mentioned above are process-based. Unlike the previous two chapters, the process used to develop software probably has the least direct effect on the consumer. So, why bother to discuss process assessment? Software processes can have a direct affect on the time to develop, the budget needed, and the quality of the end product. A typical software organization with an immature process is described in the Software Engineering Institute's Capability Maturity Model (SW-CMM) as follows: there is no objective basis for judging product quality or for solving product or process problems, there is little understanding of how the steps of the software process affect quality, the quality of the end product is difficult to predict, reviews and testing are often cut short or are eliminated when projects fall behind schedule, and the customer has little insight into the product until delivery [CMM, p.7]. Compare that to the SEI-CMM description of a software organization with a mature process: there exists an organization-wide ability for managing software development and maintenance processes; the processes are documented, usable, and consistent with the way work gets done in the organization; managers monitor the quality of software products and the process that produces them; there is an objective, quantitative basis for judging product quality and analyzing problems with the product and process; schedules and budgets are based on past performance and are realistic; and cost, schedule, functionality, and quality are usually as good as expected [CMM, p.7]. Conway's law states that "the organization of the software and the organization of the software team will be congruent" [Jargon File]. Improving the organization of and the process followed by the software producer can only help to improve the software that they produce.

Schedule and budget considerations have little effect on the end consumer. Product quality, on the other hand, has a direct effect on the consumer. From quality comes trust—consumers are more likely to trust a high-quality product than a low-quality product. There are many avenues to trust in software components. It is our hypothesis that improving the process used to develop software can help improve the quality of the end product. This is a difficult theory to test—a company would not set up two development teams to produce the same piece of

software, having one team produce software using a process and having the other produce software without a process, and then try to compare the quality of the two products. The purpose of this chapter is to examine the current mechanisms for process assessment and to see how these mechanisms can improve trust in the end product.

ISO 9000 Series

The International Standards Organization (ISO) developed the 9000 series of standards in order to provide a common standard for quality management and assurance [ISO 9000-3, p.3]. The standards, which can be applied to a company of any size or complexity, provide guidelines for an independent audit process [ISO 9000-3, p.3]. Quality-auditors check to see that a company follows commonly accepted procedures and practices when manufacturing or developing a product or providing a service [ISO 9000-3, p.3]. The ISO 9000 series of standards do not guarantee quality products. Rather, they are based on the notion that "organizations that follow accepted practices and procedures are more likely to create reliable products in a consistent manner that meets the customer's needs than those organizations that do not follow accepted practices and procedures" [ISO 9000-3, p.3].

The ISO 9000 series of standards are comprised of a set of standards, listed below [from ISO 9000-3, pp.3-4]:

- ISO 9000 – Quality management and quality assurance standards: guideline for selection and use
- ISO 9000-1 – Revision of ISO 9000
- ISO 9000-3 – Guideline for the application of ISO 9001 to the development, supply, and maintenance of software
- ISO 9001 – Quality Systems: Model for quality assurance in design and development, production, installation and servicing
- ISO 9002 – Quality Systems: Model for quality assurance in production, installation, and servicing

- ISO 9003 – Quality Systems: Model for quality assurance in final inspection and test
- ISO 9004 – Quality management and quality system elements
- ISO 9004-2 – Quality management and quality system elements (Part 2)

ISO 9000-3 expands the ISO 9000-1 standard, providing guidelines for applying ISO 9001 to the specification, development, production, installation, and support of software [ISO 9000-3, p.4]. This is intended to provide guidance where a contract requires that a supplier demonstrate its capabilities to develop, supply, and maintain software products [ISO 9000-3, p.26].

The theory behind these standards is that "well-managed organizations with defined engineering processes are more likely to produce products that consistently meet the purchaser's requirements, within schedule and budget, than poorly managed organizations that lack an engineering process" [ISO 9000-3, p.19]. To that end, the standard focuses on a number of areas. The supplier has the responsibility to create an organization that has an engineering process and policy guidelines to ensure quality products. [ISO 9000-3, p.20]. The purchaser has the responsibility to explicitly state all requirements for the product, to properly authorize changes to those requirements, and to be prepared to assume the ownership of the product after testing it to see that it is acceptable [ISO 9000-3, p.20]. The engineering process followed by the supplier must have the following phases: purchaser requirement analysis, design, implementation, test, and maintenance [ISO 9000-3, p.20]. The supplier must also provide the following supporting activities: configuration management, document control, product and process quality measurement, and training [ISO 9000-3, p.20]. A supplier must apply for ISO 9000 registration. In doing so, it selects an independent certification body to perform the necessary audits. The auditors check for the presence of a layered, documented engineering process and for evidence that it is being used [ISO 9000-3, p.157-8].

Software Engineering Institute's Capability Maturity Model (SW-CMM)

The Capability Maturity Model (SW-CMM) began as research between the Software Engineering Institute (SEI) and the MITRE Corporation to "develop a process maturity framework that would help organizations improve their software process" [CMM, p.5]. The framework, which is publicly available, is based on actual practices and reflects the best state of the practice [CMM, p.5].

The SW-CMM provides a model for improving the capability of software organizations. It categorizes processes into five maturity levels with the following characteristics [list taken from CMM, pp.15-20]:

- Level 1: Initial
    - The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.
    - The organization typically does not provide a stable environment for developing and maintaining software.
    - Over-commitment leads to staff shortages which lead to dropping planned procedures and going back to code-and-test development.
    - Can produce products that work, although often over budget and late.
- Level 2: Repeatable
    - Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
    - Policies for managing a software project and procedures to implement those policies exist.
    - Success in planning and managing new projects is based on experience from similar projects.
    - Projects implement effective control of a project management system, following realistic plans based on the performance of previous projects.

- Level 3: Defined
    - The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.
    - The organization has a standard process for developing and maintaining software that includes both software engineering and management processes and integrates them into a coherent whole.
    - Projects tailor the organization's standard software process to develop their own software process.
    - Capability is standard and consistent because both software engineering and management activities are stable and repeatable.
- Level 4: Managed
    - Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.
    - The organization sets quantitative quality goals for both software products and processes.
    - Productivity and quality are measured for important software process activities across all projects as part of an organizational measurement program.
    - The risks involved in entering a new application domain are known and carefully managed.
    - Capability is quantifiable and predictable because the process is measured and operates within quantitative limits.
- Level 5: Optimizing

- Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.
- Capability is continuously improving due to fine-tuning of the process.

Industry Opinions

It is logical to think that improving any one aspect of a product will have a positive impact on the overall quality of that product. This line of reasoning would suggest that improving the software development process would lead to a higher-quality, more trustable program. Improving the process leads to fewer bugs in the final product and reduces development time and costs. The major issue in accepting software process improvement is the impact it will have relative to the effort it requires. The reduction in development time and resources must not be overshadowed by the costs of improving the process. However, some benefits are not easy to assess quantitatively. Also, an improved process may limit a company—the organization may focus on specific projects that are well suited its existing processes and avoid newer and potentially more profitable projects.

Alan Koch, a 23-year veteran of the software development industry, argues that companies cannot afford not to improve software process [Koch]. He points out that many of the problems that an improved process addresses are visible only to the people directly involved in the development. The further a person is from the actual implementation of the project, the impact of problems and related improvements fades [Koch]. A senior manager may not see a large benefit when a process has been improved, but a project manager may notice the absence of many of the usual problems. Koch says that software engineers may even overlook the problems by assuming that they are just part of the job.

According to Koch, one of the most vital obstacles in improving the software process is productivity. He says that time is the most important resource that a company has, and many managers assume that the more time you spend implementing the project, the better the time is used. Consequently, managers view taking time and people away from projects to work on

process as a waste. Koch argues that this is not a waste at all, because the people working on the projects are spending time working through recurring problems related to the software process.

There are also more specific problems whose effects are difficult to measure exactly, but can be fixed through process improvement. Problems defining and communicating the customer's requirements between different departments can be a cause of major problems in the project. Koch argues:

> *A fine-tuned requirements management process will assure that the requirements specification says everything it needs to, and that all parties have the same understanding of what it says. A few additional hours of attention to the requirements process can avoid days or weeks of rework later in the project* [Koch].

Consequently some design problems are avoided. Koch points out "surprises" that a project team can encounter during the course of a project, including things not working as expected, or difficulties combining code from different team members. He says, "Well thought-out architecture, design, and review processes coupled with judicious use of prototyping, spiral development and other non-traditional methods will assure that the only surprise we encounter is that the whole thing came together so easily." Project management issues, including dividing the workload efficiently throughout the entire software development cycle, and configuration management problems, ensuring a smooth integration of everyone's code, are problems that upper management may not see, but their solutions can be largely beneficial to the company. The final category of problems addressed by process improvement is testing. Koch argues that this a long process because programmers spend a large amount of time fixing bugs that could have been addressed or even prevented in previous steps. He says that ideally, testing should be a quick check through the program to be sure the implementation went correctly.

Some members of the software development community, like Koch, feel that it is extremely important to refine the software creation process. Others do not directly oppose the idea of software process improvement, but simply feel that the benefits achieved by software process improvement do not justify its costs. Some, like Tom DeMarco, feel that software process improvement may actually be harmful.

Improving the process would not have an adverse effect on the quality of the end product, but may be bad for the company developing the software. Demarco points out four paradoxes that show how process improvement can hurt a company [Easterbrook]. The first paradox states "the result of process improvement is that developing software gets harder." As Steve Easterbrook wrote in an interview with DeMarco,

> *In the event, his theme was not so much that process technology is bad, but rather that it does not address the really difficult problems of software engineering. In Fred Brooks' terms, process improvement only addresses the 'accidental' problems of developing software; the harder, 'essential' problems remain* [Easterbrook].

DeMarco feels that process improvement refines the "easy" parts software engineering but does not address the more difficult problems. The second paradox DeMarco presented was that companies that do invest in process improvements often become averse to taking on riskier projects. A company that has a well-developed process may avoid change and potentially more profitable projects. Easterbrook points to a company experimenting with a new technology, such as Java, and having to discard its old prccess as an example of this paradox. The idea that humans are able to react quickly to rapid change, but tend to ignore slow change is DeMarco's third paradox [Easterbrook]. He states that "we may be getting better and better at doing the things that are less and less worth doing." Process refinement may make a company avoid getting out of a declining industry trend. DeMarco's final paradox is that we do not feel we are reaping the benefits of the effort put into reuse, when in fact we are. DeMarco points to examples of this in software such as PowerBuilder and Visual Basic, which allow us to reuse past experience [Easterbrook]. DeMarco also says that "this is, in part, because reuse is only possible if you invest heavily in the thing you want to reuse." DeMarco points out how software process improvement may improve quality of the software product, but harm the software company. The real problems, in DeMarco's opinion, are risk management and conflict resolution.

Proposal

The process plays a key role in determining the overall quality of the end product. Other engineering fields have used process refinement as an effective quality control device that

benefits both producers and consumers. Process refinement is also helpful in producing better software. Refining processes gives software a faster development time, a lower cost, and higher quality. Certification brings trust into the equation. By allowing an independent third party to test and verify a company's development process, the software industry is providing a reference point to evaluate the quality of the process, and indirectly the quality of the product.

Current standards such as the ISO-9000 series and the SW-CMM both promote the idea of process refinement in software development. The main focus of the ISO-9000 series is to ensure that a company has an open and well-defined process that is followed during production. The SW-CMM goes a step further by actually rating the process. SW-CMM has a set scale that can be used to rank different producers according to the maturity of their processes. However, both of these process assessment methods do not set or test standards for the quality of the product; rather, they assume that a better process will lead to a better product. The main goal of this project is to find a way to engender trust in software in order to form a component marketplace. If the quality of a product is sufficient, it may not be trusted. While process refinement improves the overall quality of the product, the current process assessment methods are not sufficient to ensure an acceptable level of trust between a producer and consumer.

Clearly, process improvement would have a positive impact on the quality of software, but the nature of the relationship between the process and the product make it difficult to define the effects of process improvement in quantitative manner. This applies to the positive effects that Koch points out as well as the paradoxes illustrated by DeMarco. Measuring the affect that process assessment will have on trust is difficult. While process improvement would be beneficial to software quality, there are more direct and more effective methods to engender trust in software. It is our belief that process assessment can improve trust. However, considering the cost of implementing such a scheme compared to the perceived benefits, the other methods proposed above would likely be more beneficial in the formation of a component marketplace.

References

[CMM] Paulk, Mark C., et al. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, Massachusetts: Addison-Wesley, 1994.

[Easterbrook] Easterbrook, Steve. "DeMarco: Process Considered Harmful?" <http://www.cis.cs.tu-berlin.de/~icsewow/v2n2/v2n2-2.html>.

[ISO 9000-3] Kehoe, Raymond, et al. *ISO 9000-3: A Tool for Software Product and Process Improvement*. New York: Springer, 1996.

[Jones] Referenced in [McConnell], p.16

[Jargon File] The Jargon File. <http://www.tuxedo.org/~esr/jargon/html/The-Jargon-Lexicon-framed.html>.

[Koch] Koch, Alan S. "Can We Afford Software Process?" <http://www.askprocess.com/WhitePapers/AffordProcess.html>.

[McConnell] McConnell, Steve. *After the Gold Rush: Creating a True Profession of Software Engineering*. Redmond, Washington: Microsoft Press, 1999.

CHAPTER 5

Conclusion

We believe that trust is a necessary but not a sufficient pre-condition for the development of a component marketplace. In other words, trust will not form a component marketplace, but a component marketplace will not form without trust. Therefore, in order to form a component marketplace, we need to examine ways in which the software industry can engender trust in its software components. This paper has focused on three main ways in which trust in components, and software in general, can be improved: forming a software engineering profession, certifying software components, and assessing the process used to develop software components. To conclude, we will compare each of the three mechanisms for improving trust that we discussed previously, and suggest a possible policy that can be enacted by the software industry and the government.

Bear in mind that none of the suggested courses of action previously discussed are easy or without cost. However, as discussed in the introduction, something needs to be done in order to improve the software industry's relationship with its customers. We live in a time when the government is not willing to steer a course that will potentially slow down the economy. Computers and software are a cornerstone of today's economy. Given the recent economic slowdown and the accompanying "dot-com crash," the government will not likely adopt a policy that will further slow the software based portions of the economy. However, while the costs of our proposal may be high, the long-term benefits will far outweigh the initial difficulties.

Our first proposed method of improving trust in software components was to form a profession of software engineering by licensing software engineers. Recall that licensing is a mandatory process administered at the state-level. Certification, on the other hand, is a voluntary process. It is our opinion that the industry is not likely to begin a voluntary certification regimen, especially considering the high demand for developers in today's market. Licensing has been

successful in other engineering disciplines. As stated before, licensing acts as a filter, albeit an imperfect one. Essentially, the worst software engineers would be excluded from licensing, while the best software engineers would become licensed. The process does not guarantee that all of the bad software engineers will be excluded and all of the good software engineers will be included. However, the worst of the lot will likely be excluded, thus improving the labor pool as a whole.

Licensing will improve the state of the industry by providing a metric that potential software engineers can be compared to. This will in turn improve trust. A software producer that hires licensed software engineers can say that it hires people with the best credentials [McConnell, p.106]. Consequently, the producer can then argue that their software is more trustworthy than software developed by a company that does not hire professional software engineers. Licensed engineers will have a greater say in products that they might be held liable for [McConnell, p.107]. Furthermore, a licensed engineer is less likely to sign off on a poor design or product, especially considering that a poor product or design may result in a malpractice suit.

Forming a profession of software engineering would require the formation of a new tort of computer malpractice. Professionals are held to much higher standards than non-professionals. Professional software engineers and the companies that employ them would require malpractice insurance. For that to happen, the insurer would need to have some level of trust in the product being produced. We expect that a software engineering profession would have the greatest positive impact on trust. However, considering the costs of agreeing upon a body of knowledge, licensing the potential software engineers, potentially higher salaries for professional software engineers, and malpractice insurance, we believe that formal licensure is too costly to implement at this time.

Our second proposed method to improve trust is product certification. Electrical components have been certified for more than a century by Underwriters Laboratories (UL) to ensure compliance with safety and performance standards [CBSE-UL]. We believe that a similar scheme would be beneficial in the formation of a component marketplace. Implementing a

product certification regimen would require standards that are agreed upon by the industry. Such standards must be sufficiently wide reaching to make certification a worthwhile endeavor. Issues such as standard interfaces for different component types and component performance and stability should be agreed upon by the industry. However, effective product certification should address issues beyond the actual performance of the component. As seen with the ANSI/UL 1998 Standard for Programmable Components, consideration must be given to the entire product life-cycle, including the process used to develop the component [CBSE-UL].

Component certification will improve trust by ensuring that the component in question has met rigorous, industry-based standards. Claims made about a component are verified and supported by an independent third-party. Requirements for documentation and specifications will help to reduce performance uncertainty and help component consumers in the selection and evaluation of components prior to purchase. Also, as seen with UL certification of electrical components, name recognition may provide a competitive advantage for certified components over non-certified components. The presence of professional software engineers will benefit the certification process—any third-party certification laboratory would require highly qualified individuals to formulate the standards for certification, as well as to certify the actual components.

Certification has some drawbacks. Like licensing software engineers, it is an imperfect mechanism. It is quite difficult, if not impossible, to thoroughly test a product to determine that it will run without faults on any hardware configuration. Also, certifying a component will add to the cost of production in the form of fees to the certifying body and ensuring that the component is ready to be certified. Certification is not an instantaneous process—it takes time for the certifying body to check the component and determine whether to certify it. This means a longer time to market. In new markets, it may not be beneficial for components to be certified. The time it takes to certify a component that is produced by a company trying to enter the market may give other companies time to create competitive components. Also, companies creating new software are often bought out and their products are eventually integrated into another larger product. However, even with these drawbacks, it is our opinion that product certification will have a positive impact on trust. Further, considering the costs of starting a product certification regimen

in the industry, as compared to the costs of the other mechanisms discussed, we believe that this is the best way to engender trust in components.

As mentioned earlier, an effective product certification scheme would include issues beyond the testing and evaluating of a software component. The process used to develop software is important as well. This brings us to our third and final mechanism for improving trust: process assessment. The process used to develop a piece of software has very little direct effect on the consumer. However, product quality has a direct effect on the consumer. Consumers are more likely to trust a high-quality component than a low-quality component. Process improvements are expensive for a company to implement, and the perceived benefits for the company and for the consumer are difficult to measure. Therefore, we believe that process assessment should only be used in support of product certification. The time, money, and effort that would be required for an effective process assessment regimen would be better spent in other means of improving trust.

Bearing all of this in mind, we present our policy proposal. Since product certification has the highest benefits relative to costs, it is the foundation for our proposal. Second party testing is already present in the industry today. We believe that as more companies begin to assess components for their own use, eventually a critical mass will be reached, and it will be more cost effective to form an independent third party component certification laboratory. This laboratory would best follow a model similar to that of Underwriters Laboratories. This would require widespread voluntary support from the industry to fund the formation of such an entity, as well as industry-wide agreement on standards for the components that will be certified. Such standards may include standard interfaces for components. This is already seen in other engineering fields. For instance, almost all computer hardware components have a standard interface, such as IDE hard drives; PCMCIA cards; PCI and ISA cards; standard 30-, 72-, and 168-pin RAM chips; Socket 7 processors; and power supplies providing a well-specified voltage and current on specific leads. We mentioned the ANSI/UL 1998 Standard for Programmable Components previously as a model for other standards for software components. Such a standard would take

into consideration all phases of a component's life-cycle, including risk-based analysis and design, consideration of provisions for hardware malfunctions, test planning and coverage, usability considerations, comprehensive documentation, processes for handling software changes, qualifications for off-the-shelf software, labeling that uniquely identifies the specifics of the product interface, the intended hardware platform, and the intended software configuration [CBSE-UL]. Note again that any standards for evaluating components must go beyond simply testing that a component is stable and does not have any memory leaks. Sufficient emphasis must be placed on the process used to develop the software.

We stated above that process improvements and assessments are expensive for a company to implement, and that the time and effort spent improving a company's process is better spent on other means of improving trust. However, we still believe that the process used to create a piece of software affects the quality of the software, and thus indirectly affects trust. Therefore, some form of process assessment, similar to that specified in the ANSI/UL 1998 standard described above, is necessary for a worthwhile product certification framework.

In the long run, however, we believe that creating a profession of software engineering will have the greatest positive impact on trust. Whether we like it or not, software engineering will eventually be considered a formal profession. As a society, our dependence on software grows continuously. As that dependency grows, the impact of software on the public welfare grows as well. Each time there is a highly publicized software failure, it becomes more likely that the government will become involved and require that software engineers be licensed [SEI-MPSE]. As we become more dependent on software, it becomes more likely that a software failure will result in loss of life or property. Historically, when a form of engineering puts the public at risk, the government has stepped in and required licensure.

However, government-regulated licensure is not the only way to form a profession. Efforts are currently under way to define a software engineering body of knowledge (SWEBOK). The ACM and the IEEE Computer Society have approved a software engineering code of ethics. Many of the other hallmarks of a profession, discussed in chapter 2, are already in place or are

being developed. So, whether the government steps in or not, efforts are being made to create a profession out of software engineering.

In the end, trust in software depends upon the people who write the software, the software products produced, and the processes used to create the software. All three areas must be addressed in order for consumers to trust software. All three areas are also intertwined. For instance, to effectively certify the product, the process used to create the product must be examined. To effectively create a good product, highly skilled individuals are needed, and a well-designed process must be in place for them to follow.

The business case for these actions may not be very strong. The industry may not be willing to form a profession of software engineers, to certify products, or to improve their development processes. However, some investments do not require a strong business case for them to be useful in the long run. There is no business case for immunizing children, but we do it anyway because it is in the prevailing national interest for people not to get sick. Our national interstate highway system and ARPANET, which would eventually grow to become what we know now as the Internet, were not created with a prevailing business case in mind. Rather, they were created as a means of infrastructural investment. The returns that those large investments have provided are far beyond what anyone could have expected at the time the investments were made. So, while there may not be a prevailing business case for improving the software industry, increasing trust in software, or forming a component marketplace, we can still reap the benefits of those actions, and work on closing the gap between what we would like to accomplish and what we can accomplish as software engineers.

References

[McConnell] McConnell, Steve. *After the Gold Rush: Creating a True Profession of Software Engineering*. Redmond, Washington: Microsoft Press, 1999.

[CBSE-UL] Heineman, George T., et al. *Component-Based Software Engineering*. Addison-Wesley, 2001.

[SEI-MPSE] Ford, Gary, et al. "A Mature Profession of Software Engineering." CMU/SEI-96-TR-004. <http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.004.html>.