A View of 20th and 21st Century Software Engineering

Barry Boehm
University of Southern California
University Park Campus, Los Angeles
boehm@cse.usc.edu

ABSTRACT

George Santayana's statement, "Those who cannot remember the past are condemned to repeat it," is only half true. The past also includes successful histories. If you haven't been made aware of them, you're often condemned not to repeat their successes.

In a rapidly expanding field such as software engineering, this happens a lot. Extensive studies of many software projects such as the Standish Reports offer convincing evidence that many projects fail to repeat past successes.

This paper tries to identify at least some of the major past software experiences that were well worth repeating, and some that were not. It also tries to identify underlying phenomena influencing the evolution of software engineering practices that have at least helped the author appreciate how our field has gotten to where it has been and where it is.

A counterpart Santayana-like statement about the past and future might say, "In an era of rapid change, those who repeat the past are condemned to a bleak future." (Think about the dinosaurs, and think carefully about software engineering maturity models that emphasize repeatability.)

This paper also tries to identify some of the major sources of change that will affect software engineering practices in the next couple of decades, and identifies some strategies for assessing and adapting to these sources of change. It also makes some first steps towards distinguishing relatively timeless software engineering principles that are risky not to repeat, and conditions of change under which aging practices will become increasingly risky to repeat.

Categories and Subject Descriptors

D.2.9 [**Management**]: Cost estimation, life cycle, productivity, software configuration management, software process models.

General Terms

Management, Economics, Human Factors.

Keywords

Software engineering, software history, software futures

ICSE'06, May 20–28, 2006, Shanghai, China.

1. INTRODUCTION

One has to be a bit presumptuous to try to characterize both the past and future of software engineering in a few pages. For one thing, there are many types of software engineering: large or small; commodity or custom; embedded or user-intensive; greenfield or legacy/COTS/reuse-driven; homebrew, outsourced, or both; casual-use or mission-critical. For another thing, unlike the engineering of electrons, materials, or chemicals, the basic software elements we engineer tend to change significantly from one decade to the next.

Fortunately, I've been able to work on many types and generations of software engineering since starting as a programmer in 1955. I've made a good many mistakes in developing, managing, and acquiring software, and hopefully learned from them. I've been able to learn from many insightful and experienced software engineers, and to interact with many thoughtful people who have analyzed trends and practices in software engineering. These learning experiences have helped me a good deal in trying to understand how software engineering got to where it is and where it is likely to go. They have also helped in my trying to distinguish between timeless principles and obsolete practices for developing successful software-intensive systems.

In this regard, I am adapting the [147] definition of "engineering" to define engineering as "the application of science and mathematics by which the properties of software are made useful to people." The phrase "useful to people" implies that the relevant sciences include the behavioral sciences, management sciences, and economics, as well as computer science.

In this paper, I'll begin with a simple hypothesis: software people don't like to see software engineering done unsuccessfully, and try to make things better. I'll try to elaborate this into a high-level decade-by-decade explanation of software engineering's past. I'll then identify some trends affecting future software engineering practices, and summarize some implications for future software engineering researchers, practitioners, and educators.

2. A Hegelian View of Software Engineering's Past

The philosopher Hegel hypothesized that increased human understanding follows a path of thesis (this is why things happen the way they do); antithesis (the thesis fails in some important ways; here is a better explanation); and synthesis (the antithesis rejected too much of the original thesis; here is a hybrid that captures the best of both while avoiding their defects). Below I'll try to apply this hypothesis to explaining the evolution of software engineering from the 1950's to the present.

2.1 1950's Thesis: Software Engineering Is Like Hardware Engineering

When I entered the software field in 1955 at General Dynamics, the prevailing thesis was, "Engineer software like you engineer hardware." Everyone in the GD software organization was either a hardware engineer or a mathematician, and the software being developed was supporting aircraft or rocket engineering. People kept engineering notebooks and practiced such hardware precepts as "measure twice, cut once," before running their code on the computer.

This behavior was also consistent with 1950's computing economics. On my first day on the job, my supervisor showed me the GD ERA 1103 computer, which filled a large room. He said, "Now listen. We are paying $600 an hour for this computer and $2 an hour for you, and I want you to act accordingly." This instilled in me a number of good practices such as desk checking, buddy checking, and manually executing my programs before running them. But it also left me with a bias toward saving microseconds when the economic balance started going the other way.

The most ambitious information processing project of the 1950's was the development of the Semi-Automated Ground Environment (SAGE) for U.S. and Canadian air defense. It brought together leading radar engineers, communications engineers, computer engineers, and nascent software engineers to develop a system that would detect, track, and prevent enemy aircraft from bombing the U.S. and Canadian homelands.

Figure 1 shows the software development process developed by the hardware engineers for use in SAGE [1]. It shows that sequential waterfall-type models have been used in software development for a long time. Further, if one arranges the steps in a V form with Coding at the bottom, this 1956 process is equivalent to the V-model for software development. SAGE also developed the Lincoln Labs Utility System to aid the thousands of programmers participating in SAGE software development. It included an assembler, a library and build management system, a number of utility programs, and aids to testing and debugging. The resulting SAGE system successfully met its specifications with about a one-year schedule slip. Benington's bottom-line comment on the success was "It is easy for me to single out the one factor that led to our relative success: we were all engineers and had been trained to organize our efforts along engineering lines."

Another indication of the hardware engineering orientation of the 1950's is in the names of the leading professional societies for software professionals: the Association for Computing Machinery and the IEEE Computer Society.

2.2 1960's Antithesis: Software Crafting

By the 1960's, however, people were finding out that software phenomenology differed from hardware phenomenology in significant ways. First, software was much easier to modify than was hardware, and it did not require expensive production lines to make product copies. One changed the program once, and then reloaded the same bit pattern onto another computer, rather than having to individually change the configuration of each copy of the hardware. This ease of modification led many people and organizations to adopt a "code and fix" approach to software development, as compared to the exhaustive Critical Design Reviews that hardware engineers performed before committing to production lines and bending metal (measure twice, cut once). Many software applications became more people-intensive than hardware-intensive; even SAGE became more dominated by psychologists addressing human-computer interaction issues than by radar engineers.

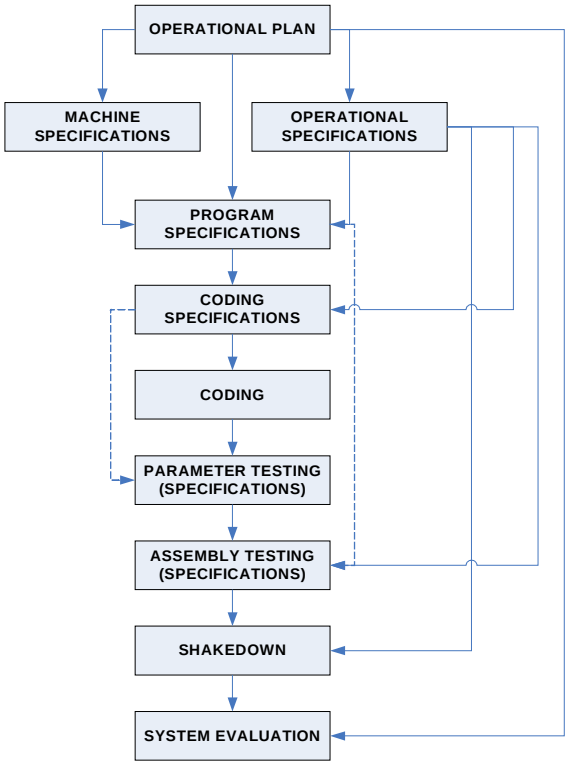

Figure 1. The SAGE Software Development Process (1956)

Another software difference was that software did not wear out. Thus, software reliability could only imperfectly be estimated by hardware reliability models, and "software maintenance" was a much different activity than hardware maintenance. Software was invisible, it didn't weigh anything, but it cost a lot. It was hard to tell whether it was on schedule or not, and if you added more people to bring it back on schedule, it just got later, as Fred Brooks explained in the Mythical Man-Month [42]. Software generally had many more states, modes, and paths to test, making its specifications much more difficult. Winston Royce, in his classic 1970 paper, said, "In order to procure a $5 million hardware device, I would expect a 30-page specification would provide adequate detail to control the procurement. In order to procure $5 million worth of software, a 1500 page specification is about right in order to achieve comparable control."[132].

Another problem with the hardware engineering approach was that the rapid expansion of demand for software outstripped the supply of engineers and mathematicians. The SAGE program began hiring and training humanities, social sciences, foreign language, and fine arts majors to develop software. Similar non-engineering people flooded into software development positions for business, government, and services data processing.

These people were much more comfortable with the code-and-fix approach. They were often very creative, but their fixes often led to heavily patched spaghetti code. Many of them were heavily influenced by 1960's "question authority" attitudes and tended to march to their own drummers rather than those of the organization employing them. A significant subculture in this regard was the

"hacker culture" of very bright free spirits clustering around major university computer science departments [83]. Frequent role models were the "cowboy programmers" who could pull all-nighters to hastily patch faulty code to meet deadlines, and would then be rewarded as heroes.

Not all of the 1960's succumbed to the code-and-fix approach, IBM's OS-360 family of programs, although expensive, late, and initially awkward to use, provided more reliable and comprehensive services than its predecessors and most contemporaries, leading to a dominant marketplace position. NASA's Mercury, Gemini, and Apollo manned spacecraft and ground control software kept pace with the ambitious "man on the moon by the end of the decade" schedule at a high level of reliability.

Other trends in the 1960's were:

- Much better infrastructure. Powerful mainframe operating systems, utilities, and mature higher-order languages such as Fortran and COBOL made it easier for non-mathematicians to enter the field.
- Generally manageable small applications, although those often resulted in hard-to-maintain spaghetti code.
- The establishment of computer science and informatics departments of universities, with increasing emphasis on software.
- The beginning of for-profit software development and product companies.
- More and more large, mission-oriented applications. Some were successful as with OS/360 and Apollo above, but many more were unsuccessful, requiring near-complete rework to get an adequate system.
- Larger gaps between the needs of these systems and the capabilities for realizing them.

This situation led the NATO Science Committee to convene two landmark "Software Engineering" conferences in 1968 and 1969, attended by many of the leading researcher and practitioners in the field [107][44]. These conferences provided a strong baseline of understanding of the software engineering state of the practice that industry and government organizations could use as a basis for determining and developing improvements. It was clear that better organized methods and more disciplined practices were needed to scale up to the increasingly large projects and products that were being commissioned.

2.3 1970's Synthesis and Antithesis: Formality and Waterfall Processes

The main reaction to the 1960's code-and-fix approach involved processes in which coding was more carefully organized and was preceded by design, and design was preceded by requirements engineering. Figure 2 summarizes the major 1970's initiatives to synthesize the best of 1950's hardware engineering techniques with improved software-oriented techniques.

More careful organization of code was exemplified by Dijkstra's famous letter to ACM Communications, "Go To Statement Considered Harmful" [56]. The Bohm-Jacopini result [40] showing that sequential programs could always be constructed without go-to's led to the Structured Programming movement.

This movement had two primary branches. One was a "formal methods" branch that focused on program correctness, either by mathematical proof [72][70], or by construction via a "programming calculus" [56]. The other branch was a less formal mix of technical and management methods, "top-down structured programming with chief programmer teams," pioneered by Mills and highlighted by the successful New York Times application led by Baker [7].

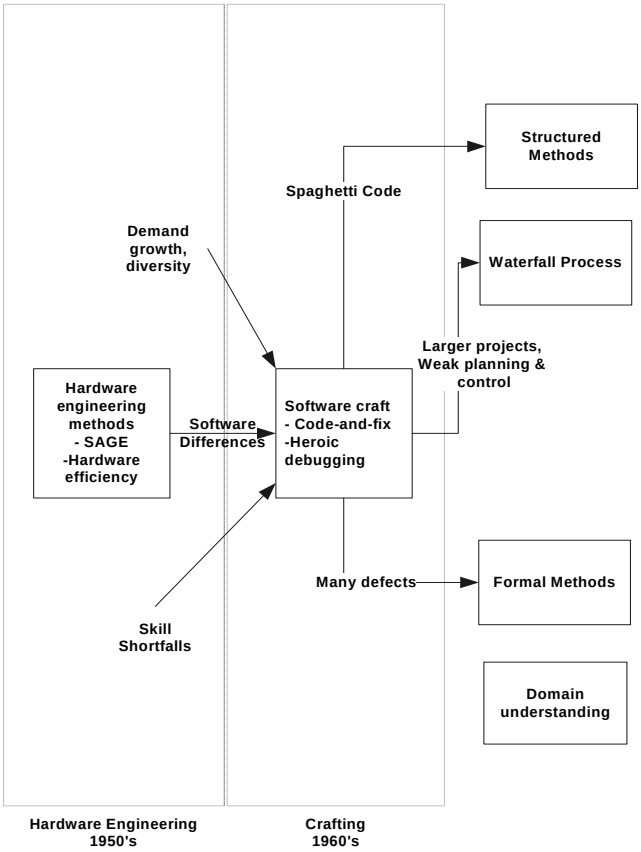

Figure2. Software Engineering Trends Through the 1970's

The success of structured programming led to many other "structured" approaches applied to software design. Principles of modularity were strengthened by Constantine's concepts of coupling (to be minimized between modules) and cohesion (to be maximized within modules) [48], by Parnas's increasingly strong techniques of information hiding [116][117][118], and by abstract data types [92][75][151]. A number of tools and methods employing structured concepts were developed, such as structured design [106][55][154]; Jackson's structured design and programming [82], emphasizing data considerations; and Structured Program Design Language [45].

Requirements-driven processes were well established in the 1956 SAGE process model in Figure 1, but a stronger synthesis of the 1950's paradigm and the 1960's crafting paradigm was provided by Royce's version of the "waterfall" model shown in Figure 3 [132].

It added the concepts of confining iterations to successive phases, and a "build it twice" prototyping activity before committing to full-scale development. A subsequent version emphasized verification and validation of the artifacts in each phase before proceeding to the next phase in order to contain defect finding and fixing within the same phase whenever possible. This was based on the data from

TRW, IBM, GTE, and safeguard on the relative cost of finding defects early vs. late [24].

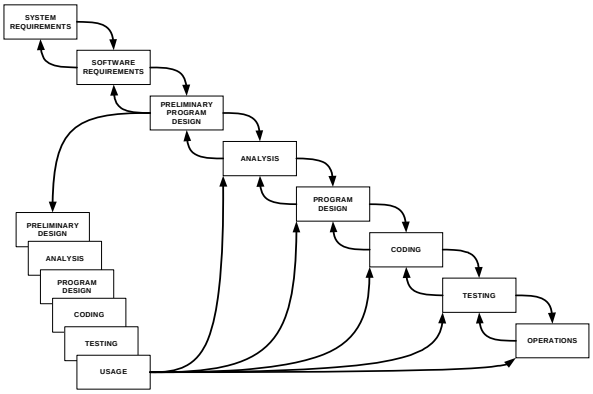

Figure 3. The Royce Waterfall Model (1970)

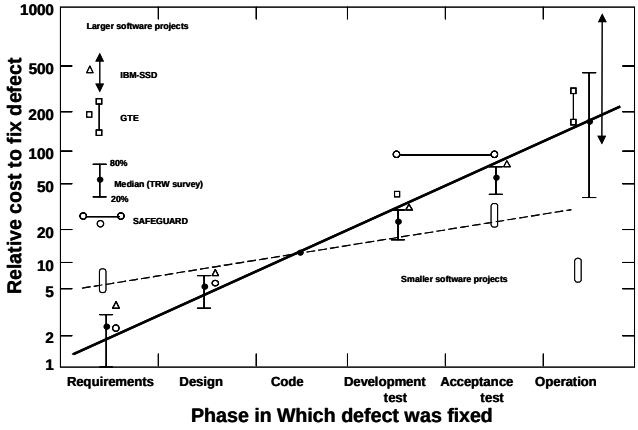

Figure 4. Increase in Software Cost-to-fix vs. Phase (1976)

Unfortunately, partly due to convenience in contracting for software acquisition, the waterfall model was most frequently interpreted as a purely sequential process, in which design did not start until there was a complete set of requirements, and coding did not start until completion of an exhaustive critical design review. These misinterpretations were reinforced by government process standards emphasizing a pure sequential interpretation of the waterfall model.

Quantitative Approaches

One good effect of stronger process models was the stimulation of stronger quantitative approaches to software engineering. Some good work had been done in the 1960's such as System Development Corp's software productivity data [110] and experimental data showing 26:1 productivity differences among programmers [66]; IBM's data presented in the 1960 NATO report [5]; and early data on distributions of software defects by phase and type. Partly stimulated by the 1973 Datamation article, "Software and its Impact: A Quantitative Assessment" [22], and the Air Force CCIP-85 study on which it was based, more management attention and support was given to quantitative software analysis. Considerable progress was made in the 1970's on complexity metrics that helped identify defect-prone modules [95][76]; software reliability estimation models [135][94]; quantitative approaches to software quality [23][101]; software cost and schedule estimation models [121][73][26]; and sustained quantitative laboratories such as the NASA/UMaryland/CSC Software Engineering Laboratory [11].

Some other significant contributions in the 1970's were the in-depth analysis of people factors in Weinberg's Psychology of Computer Programming [144]; Brooks' Mythical Man Month [42], which captured many lessons learned on incompressibility of software schedules, the 9:1 cost difference between a piece of demonstration software and a software system product, and many others; Wirth's Pascal [149] and Modula-2 [150] programming languages; Fagan's inspection techniques [61]; Toshiba's reusable product line of industrial process control software [96]; and Lehman and Belady's studies of software evolution dynamics [12]. Others will be covered below as precursors to 1980's contributions.

However, by the end of the 1970's, problems were cropping up with formality and sequential waterfall processes. Formal methods had difficulties with scalability and usability by the majority of less-expert programmers (a 1975 survey found that the average coder in 14 large organizations had two years of college education and two years of software experience; was familiar with two programming languages and software products; and was generally sloppy, inflexible, "in over his head", and undermanaged [50]. The sequential waterfall model was heavily document-intensive, slow-paced, and expensive to use.

Since much of this documentation preceded coding, many impatient managers would rush their teams into coding with only minimal effort in requirements and design. Many used variants of the self-fulfilling prophecy, "We'd better hurry up and start coding, because we'll have a lot of debugging to do." A 1979 survey indicated that about 50% of the respondents were not using good software requirements and design practices [80] resulting from 1950's SAGE experience [25]. Many organizations were finding that their software costs were exceeding their hardware costs, tracking the 1973 prediction in Figure 5 [22], and were concerned about significantly improving software productivity and use of well-known best practices, leading to the 1980's trends to be discussed next.

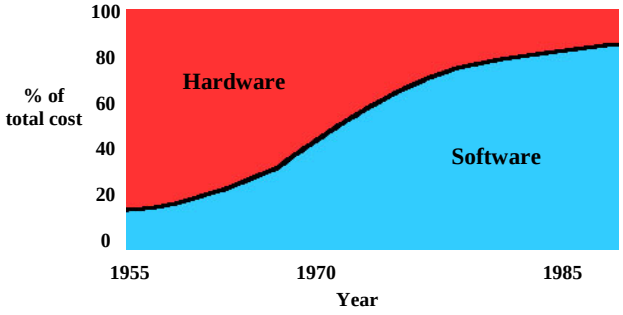

Figure 5. Large-Organization Hardware-Software Cost Trends (1973)

2.4 1980's Synthesis: Productivity and Scalability

Along with some early best practices developed in the 1970's, the 1980's led to a number of initiatives to address the 1970's problems, and to improve software engineering productivity and scalability. Figure 6 shows the extension of the timeline in Figure 2 through the rest of the decades through the 2010's addressed in the paper.

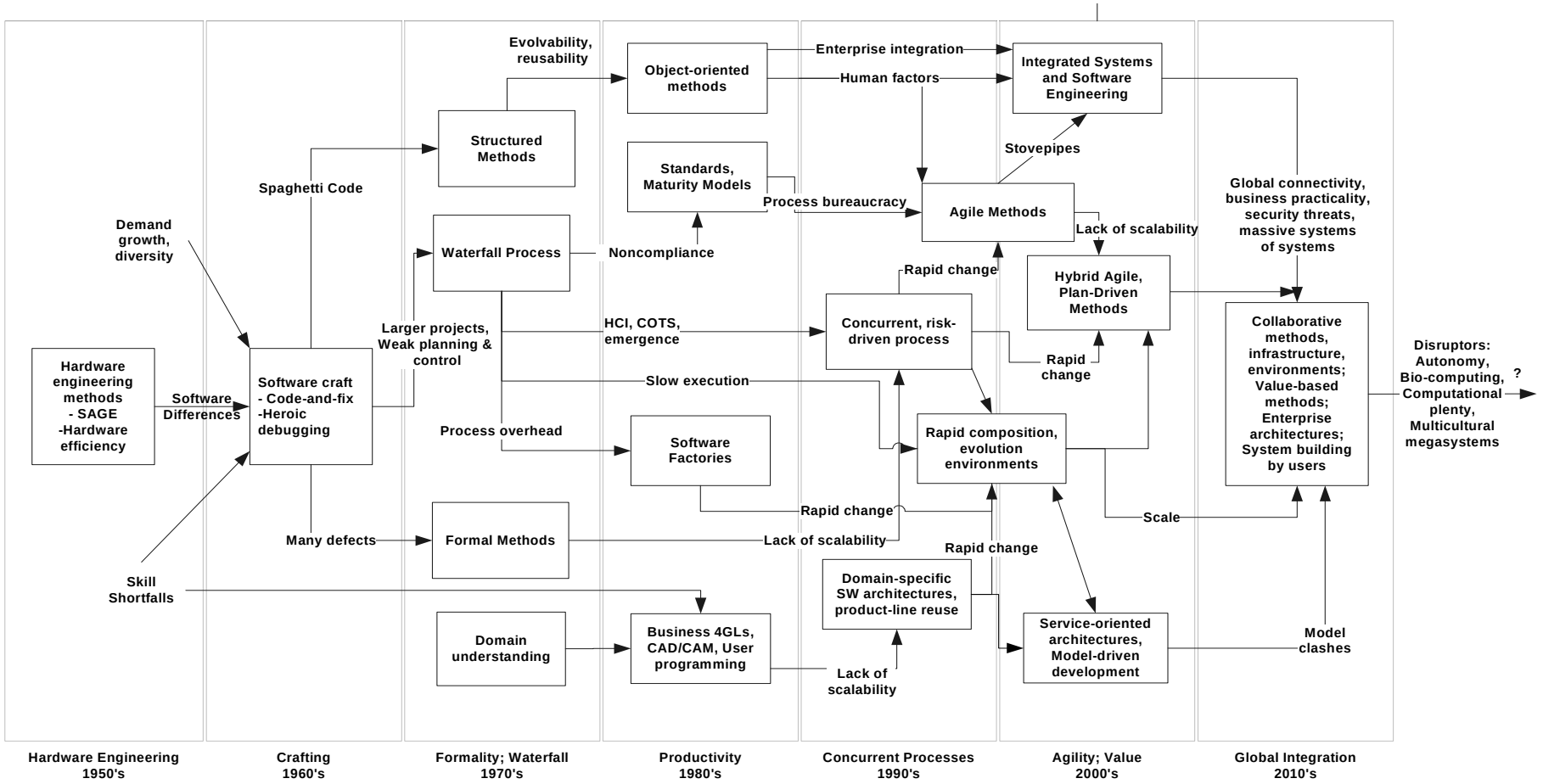

Figure 6. A Full Range of Software Engineering Trends

The rise in quantitative methods in the late 1970's helped identify the major leverage points for improving software productivity. Distributions of effort and defects by phase and activity enabled better prioritization of improvement areas. For example, organizations spending 60% of their effort in the test phase found that 70% of the "test" activity was actually rework that could be done much less expensively if avoided or done earlier, as indicated by Figure 4. The cost drivers in estimation models identified management controllables that could reduce costs through investments in better staffing training, processes, methods, tools, and asset reuse.

The problems with process noncompliance were dealt with initially by more thorough contractual standards, such as the 1985 U.S. Department of Defense (DoD) Standards DoD-STD-2167 and MIL-STD-1521B, which strongly reinforced the waterfall model by tying its milestones to management reviews, progress payments, and award fees. When these often failed to discriminate between capable software developers and persuasive proposal developers, the DoD commissioned the newly-formed (1984) CMU Software Engineering Institute to develop a software capability maturity model (SW-CMM) and associated methods for assessing an organization's software process maturity. Based extensively on IBM's highly disciplined software practices and Deming-Juran-Crosby quality practices and maturity levels, the resulting SW-CMM provided a highly effective framework for both capability assessment and improvement [81] The SW-CMM content was largely method-independent, although some strong sequential waterfall-model reinforcement remained. For example, the first Ability to Perform in the first Key Process Area, Requirements Management, states, "Analysis and allocation of the system requirements is not the responsibility of the software engineering group but is a prerequisite for their work." [114]. A similar International Standards Organization ISO-9001 standard for quality practices applicable to software was concurrently developed, largely under European leadership.

The threat of being disqualified from bids caused most software contractors to invest in SW-CMM and ISO-9001 compliance. Most reported good returns on investment due to reduced software rework. These results spread the use of the maturity models to internal software organizations, and led to a new round of refining and developing new standards and maturity models, to be discussed under the 1990's.

Software Tools

In the software tools area, besides the requirements and design tools discussed under the 1970's, significant tool progress had been mode in the 1970's in such areas as test tools (path and test coverage analyzers, automated test case generators, unit test tools, test traceability tools, test data analysis tools, test simulator-stimulators and operational test aids) and configuration management tools. An excellent record of progress in the configuration management (CM) area has been developed by the NSF ACM/IEE(UK)–sponsored IMPACT project [62]. It traces the mutual impact that academic research and industrial research and practice have had in evolving CM from a manual bookkeeping practice to powerful automated aids for version and release management, asynchronous checkin/checkout, change tracking, and integration and test support. A counterpart IMPACT paper has been published on modern programming languages [134]; other are underway on Requirements, Design, Resource Estimation, Middleware, Reviews and Walkthroughs, and Analysis and Testing [113].

The major emphasis in the 1980's was on integrating tools into support environments. There were initially overfocused on Integrated Programming Support Environments (IPSE's), but eventually broadened their scope to Computer-Aided Software Engineering (CASE) or Software Factories. These were pursued extensively in the U.S. and Europe, but employed most effectively in Japan [50].

A significant effort to improve the productivity of formal software development was the RAISE environment [21]. A major effort to develop a standard tool interoperability framework was the HP/NIST/ECMA Toaster Model [107]. Research on advanced software development environments included knowledge-based support, integrated project databases [119], advanced tools interoperability architecture, and tool/environment configuration and execution languages such as Odin [46].

Software Processes

Such languages led to the vision of process-supported software environments and Osterweil's influential "Software Processes are Software Too" keynote address and paper at ICSE 9 [111]. Besides reorienting the focus of software environments, this concept exposed a rich duality between practices that are good for developing products and practices that are good for developing processes. Initially, this focus was primarily on process programming languages and tools, but the concept was broadened to yield highly useful insights on software process requirements, process architectures, process change management, process families, and process asset libraries with reusable and composable process components, enabling more cost-effective realization of higher software process maturity levels.

Improved software processes contributed to significant increases in productivity by reducing rework, but prospects of even greater productivity improvement were envisioned via work avoidance. In the early 1980's, both revolutionary and evolutionary approaches to work avoidance were addressed in the U.S. DoD STARS program [57]. The revolutionary approach emphasized formal specifications and automated transformational approaches to generating code from specifications, going back to early–1970's "automatic programming" research [9][10], and was pursued via the Knowledge-Based Software Assistant (KBSA) program The evolutionary approach emphasized a mixed strategy of staffing, reuse, process, tools, and management, supported by integrated environments [27]. The DoD software program also emphasized accelerating technology transition, based on the [128] study indicating that an average of 18 years was needed to transition software engineering technology from concept to practice. This led to the technology-transition focus of the DoD-sponsored CMU Software Engineering Institute (SEI) in 1984. Similar initiatives were pursued in the European Community and Japan, eventually leading to SEI-like organizations in Europe and Japan.

2.4.1 No Silver Bullet

The 1980's saw other potential productivity improvement approaches such as expert systems, very high level languages, object orientation, powerful workstations, and visual programming. All of these were put into perspective by Brooks' famous "No Silver Bullet" paper presented at IFIP 1986 [43]. It distinguished the "accidental" repetitive tasks that could be avoided or streamlined via

automation, from the "essential" tasks unavoidably requiring syntheses of human expertise, judgment, and collaboration. The essential tasks involve four major challenges for productivity solutions: high levels of software complexity, conformity, changeability, and invisibility. Addressing these challenges raised the bar significantly for techniques claiming to be "silver bullet" software solutions. Brooks' primary candidates for addressing the essential challenges included great designers, rapid prototyping, evolutionary development (growing vs. building software systems) and work avoidance via reuse.

Software Reuse

The biggest productivity payoffs during the 1980's turned out to involve work avoidance and streamlining through various forms of reuse. Commercial infrastructure software reuse (more powerful operating systems, database management systems, GUI builders, distributed middleware, and office automation on interactive personal workstations) both avoided much programming and long turnaround times. Engelbart's 1968 vision and demonstration was reduced to scalable practice via a remarkable desktop-metaphor, mouse and windows interactive GUI, what you see is what you get (WYSIWYG) editing, and networking/middleware support system developed at Xerox PARC in the 1970's reduced to affordable use by Apple's Lisa(1983) and Macintosh(1984), and implemented eventually on the IBM PC family by Microsoft's Windows 3.1 (198x).

Better domain architecting and engineering enabled much more effective reuse of application components, supported both by reuse frameworks such as Draco [109] and by domain-specific business fourth-generation-language (4GL's) such as FOCUS and NOMAD [102]. Object-oriented methods tracing back to Simula-67 [53] enabled even stronger software reuse and evolvability via structures and relations (classes, objects, methods, inheritance) that provided more natural support for domain applications. They also provided better abstract data type modularization support for high-cohesion modules and low inter-module coupling. This was particularly valuable for improving the productivity of software maintenance, which by the 1980's was consuming about 50-75% of most organizations' software effort [91][26]. Object-oriented programming languages and environments such as Smalltalk, Eiffel [102], C++ [140], and Java [69] stimulated the rapid growth of object-oriented development, as did a proliferation of object-oriented design and development methods eventually converging via the Unified Modeling Language (UML) in the 1990's [41].

2.5 1990's Antithesis: Concurrent vs. Sequential Processes

The strong momentum of object-oriented methods continued into the 1990's. Object-oriented methods were strengthened through such advances as design patterns [67]; software architectures and architecture description languages [121][137][12]; and the development of UML. The continued expansion of the Internet and emergence of the World Wide Web [17] strengthened both OO methods and the criticality of software in the competitive marketplace.

Emphasis on Time-To-Market

The increased importance of software as a competitive discriminator and the need to reduce software time-to-market caused a major shift away from the sequential waterfall model to models emphasizing concurrent engineering of requirements, design, and code; of product and process; and of software and systems. For example, in the late 1980's Hewlett Packard found that several of its market sectors had product lifetimes of about 2.75 years, while its waterfall process was taking 4 years for software development. As seen in Figure 7, its investment in a product line architecture and reusable components increased development time for the first three products in 1986-87, but had reduced development time to one year by 1991-92 [92]. The late 1990's saw the publication of several influential books on software reuse [83][128][125][146].

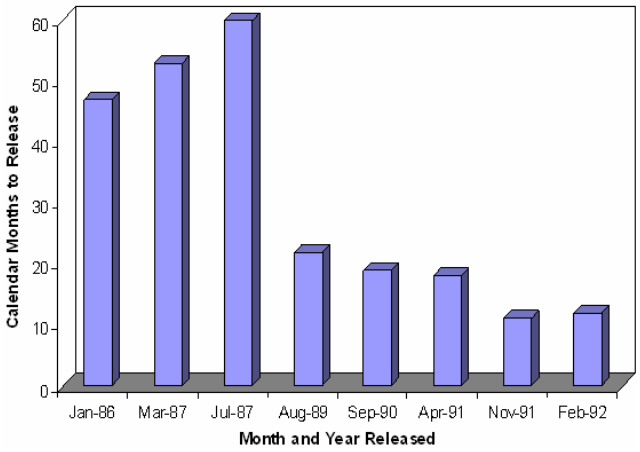

Figure 7. HP Product Line Reuse Investment and Payoff

Besides time-to market, another factor causing organizations to depart from waterfall processes was the shift to user-interactive products with emergent rather than prespecifiable requirements. Most users asked for their GUI requirements would answer, "I'm not sure, but I'll know it when I see it" (IKIWISI). Also, reuse-intensive and COTS-intensive software development tended to follow a bottom-up capabilities-to-requirements process rather than a top-down requirements-to capabilities process.

Controlling Concurrency

The risk-driven spiral model [28] was intended as a process to support concurrent engineering, with the project's primary risks used to determine how much concurrent requirements engineering, architecting, prototyping, and critical-component development was enough. However, the original model contained insufficient guidance on how to keep all of these concurrent activities synchronized and stabilized. Some guidance was provided by the elaboration of software risk management activities [28][46] and the use of the stakeholder win-win Theory W [31] as milestone criteria. But the most significant addition was a set of common industry-coordinated stakeholder commitment milestones that serve as a basis for synchronizing and stabilizing concurrent spiral (or other) processes.

These anchor point milestones-- Life Cycle Objectives (LCO), Life Cycle Architecture(LCA), and Initial Operational Capability (IOC) – have pass-fail criteria based on the compatibility and feasibility of the concurrently-engineered requirements, prototypes, architecture, plans, and business case [33]. They turned out to be compatible with major government acquisition milestones and the AT&T Architecture Review Board milestones [19][97]. They were also

adopted by Rational/IBM as the phase gates in the Rational Unified Process [87][133][84], and as such have been used on many successful projects. They are similar to the process milestones used by Microsoft to synchronize and stabilize its concurrent software processes [53]. Other notable forms of concurrent, incremental and evolutionary development include the Scandinavian Participatory Design approach [62], various forms of Rapid Application Development [103][98], and agile methods, to be discussed under the 2000's below. [87] is an excellent source for iterative and evolutionary development methods.

Open Source Development

Another significant form of concurrent engineering making strong contribution in the 1990's was open source software development. From its roots in the hacker culture of the 1960's, it established an institutional presence in 1985 with Stallman's establishment of the Free Software Foundation and the GNU General Public License [140]. This established the conditions of free use and evolution of a number of highly useful software packages such as the GCC C-Language compiler and the emacs editor. Major 1990's milestones in the open source movement were Torvalds' Linux (1991), Berners-Lee's World Wide Web consortium (1994), Raymond's "The Cathedral and the Bazaar" book [128], and the O'Reilly Open Source Summit (1998), including leaders of such products as Linux , Apache, TCL, Python, Perl, and Mozilla [144].

Usability and Human-Computer Interaction

As mentioned above, another major 1990's emphasis was on increased usability of software products by non-programmers. This required reinterpreting an almost universal principle, the Golden Rule, "Do unto others as you would have others do unto you", To literal-minded programmers and computer science students, this meant developing programmer-friendly user interfaces. These are often not acceptable to doctors, pilots, or the general public, leading to preferable alternatives such as the Platinum Rule, "Do unto others as they would be done unto."

Serious research in human-computer interaction (HCI) was going on as early as the second phase of the SAGE project at Rand Corp in the 1950's, whose research team included Turing Award winner Allen Newell. Subsequent significant advances have included experimental artifacts such as Sketchpad and the Engelbert and Xerox PARC interactive environments discussed above. They have also included the rapid prototyping and Scandinavian Participatory Design work discussed above, and sets of HCI guidelines such as [138] and [13]. The late 1980's and 1990's also saw the HCI field expand its focus from computer support of individual performance to include group support systems [96][111].

2.6 2000's Antithesis and Partial Synthesis: Agility and Value

So far, the 2000's have seen a continuation of the trend toward rapid application development, and an acceleration of the pace of change in information technology (Google, Web-based collaboration support), in organizations (mergers, acquisitions, startups), in competitive countermeasures (corporate judo, national security), and in the environment (globalization, consumer demand patterns). This rapid pace of change has caused increasing frustration with the heavyweight plans, specifications, and other documentation imposed by contractual inertia and maturity model compliance criteria. One organization recently presented a picture of its CMM Level 4 Memorial Library: 99 thick spiral binders of documentation used only to pass a CMM assessment.

Agile Methods

The late 1990's saw the emergence of a number of agile methods such as Adaptive Software Development, Crystal, Dynamic Systems Development, eXtreme Programming (XP), Feature Driven Development, and Scrum. Its major method proprietors met in 2001 and issued the Agile Manifesto, putting forth four main value preferences:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation
- Responding to change over following a plan.

The most widely adopted agile method has been XP, whose major technical premise in [14] was that its combination of customer collocation, short development increments, simple design, pair programming, refactoring, and continuous integration would flatten the cost-of change-vs.-time curve in Figure 4. However, data reported so far indicate that this flattening does not take place for larger projects. A good example was provided by a large Thought Works Lease Management system presented at ICSE 2002 [62]. When the size of the project reached over 1000 stories, 500,000 lines of code, and 50 people, with some changes touching over 100 objects, the cost of change inevitably increased. This required the project to add some more explicit plans, controls, and high-level architecture representations.

Analysis of the relative "home grounds" of agile and plan-driven methods found that agile methods were most workable on small projects with relatively low at-risk outcomes, highly capable personnel, rapidly changing requirements, and a culture of thriving on chaos vs. order. As shown in Figure 8 [36], the agile home ground is at the center of the diagram, the plan-driven home ground is at the periphery, and projects in the middle such as the lease management project needed to add some plan-driven practices to XP to stay successful.

Value-Based Software Engineering

Agile methods' emphasis on usability improvement via short increments and value-prioritized increment content are also responsive to trends in software customer preferences. A recent Computerworld panel on "The Future of Information Technology (IT)" indicated that usability and total ownership cost-benefits, including user inefficiency and ineffectiveness costs, are becoming IT user organizations' top priorities [5]. A representative quote from panelist W. Brian Arthur was "Computers are working about as fast as we need. The bottleneck is making it all usable." A recurring user-organization desire is to have technology that adapts to people rather than vice versa. This is increasingly reflected in users' product selection activities, with evaluation criteria increasingly emphasizing product usability and value added vs. a previous heavy emphasis on product features and purchase costs. Such trends ultimately will affect producers' product and process priorities, marketing strategies, and competitive survival.

Some technology trends strongly affecting software engineering for usability and cost-effectiveness are increasingly powerful enterprise support packages, data access and mining tools, and Personal Digital Assistant (PDA) capabilities. Such products have tremendous

potential for user value, but determining how they will be best configured will involve a lot of product experimentation, shakeout, and emergence of superior combinations of system capabilities.

In terms of future software process implications, the fact that the capability requirements for these products are emergent rather than prespecifiable has become the primary challenge. Not only do the users exhibit the IKIWISI (I'll know it when I see it) syndrome, but their priorities change with time. These changes often follow a Maslow need hierarchy, in which unsatisfied lower-level needs are top priority, but become lower priorities once the needs are satisfied [96]. Thus, users will initially be motivated by survival in terms of capabilities to process new work-loads, followed by security once the workload-processing needs are satisfied, followed by self-actualization in terms of capabilities for analyzing the workload content for self-improvement and market trend insights once the security needs are satisfied.

It is clear that requirements emergence is incompatible with past process practices such as requirements-driven sequential waterfall process models and formal programming calculi; and with process maturity models emphasizing repeatability and optimization [114]. In their place, more adaptive [74] and risk-driven [32] models are needed. More fundamentally, the theory underlying software process models needs to evolve from purely reductionist "modern" world views (universal, general, timeless, written) to a synthesis of these and situational "postmodern" world views (particular, local, timely, oral) as discussed in [144]. A recent theory of value-based software engineering (VBSE) and its associated software processes [37] provide a starting point for addressing these challenges, and for extending them to systems engineering processes. The associated VBSE book [17] contains further insights and emerging directions for VBSE processes.

The value-based approach also provides a framework for determining which low-risk, dynamic parts of a project are better addressed by more lightweight agile methods and which high-risk, more stabilized parts are better addressed by plan-driven methods. Such syntheses are becoming more important as software becomes more product-critical or mission-critical while software organizations continue to optimize on time-to-market.

Software Criticality and Dependability

Although people's, systems', and organizations' dependency on software is becoming increasingly critical, de-pendability is generally not the top priority for software producers. In the words of the 1999 PITAC Report, "The IT industry spends the bulk of its resources, both financial and human, on rapidly bringing products to market." [123].

Recognition of the problem is increasing. ACM President David Patterson has called for the formation of a top-priority Security/Privacy, Usability, and Reliability (SPUR) initiative [119]. Several of the Computerworld "Future of IT" panelists in [5] indicated increasing customer pressure for higher quality and vendor warranties, but others did not yet see significant changes happening among software product vendors.

This situation will likely continue until a major software-induced systems catastrophe similar in impact on world consciousness to the 9/11 World Trade Center catastrophe stimulates action toward establishing account-ability for software dependability. Given the high and increasing software vulnerabilities of the world's current financial, transportation, communications, energy distribution, medical, and emergency services infrastructures, it is highly likely that such a software-induced catastrophe will occur between now and 2025.

Some good progress in high-assurance software technology continues to be made, including Hoare and others' scalable use of assertions in Microsoft products [71], Scherlis' tools for detecting Java concurrency problems, Holtzmann and others' model-checking capabilities [78] Poore and others' model-based testing capabilities [124] and Leveson and others' contributions to software and system safety.

COTS, Open Source, and Legacy Software

A source of both significant benefits and challenges to simultaneously adopting to change and achieving high dependability is the increasing availability of commercial-off-the-shelf (COTS) systems and components. These enable rapid development of products with significant capabilities in a short time. They are also continually evolved by the COTS vendors to fix defects found by many users and to competitively keep pace with changes in technology. However this continuing change is a source of new streams of defects; the lack of access to COTS source code inhibits users' ability to improve their applications' dependability; and vendor-controlled evolution adds risks and constraints to users' evolution planning.

Overall, though, the availability and wide distribution of mass-produced COTS products makes software productivity curves look about as good as hardware productivity curves showing exponential growth in numbers of transistors produced and Internet packets shipped per year. Instead of counting the number of new source lines of code (SLOC) produced per year and getting a relatively flat software productivity curve, a curve more comparable to the hardware curve should count the number of executable machine instructions or lines of code in service (LOCS) on the computers owned by an organization.

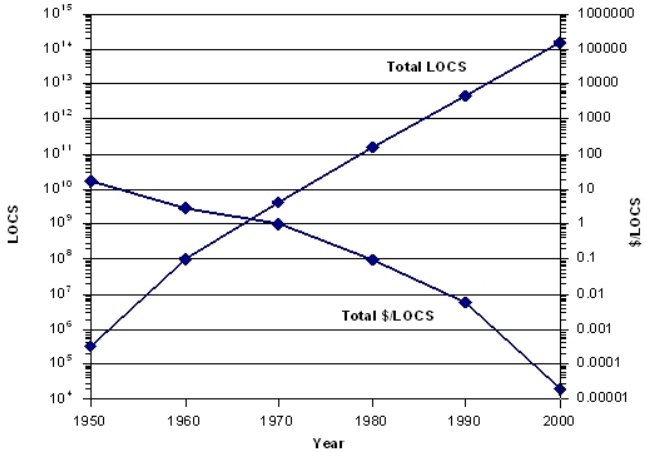

Figure 8. U.S. DoD Lines of Code in Service and Cost/LOCS

Figure 8 shows the results of roughly counting the LOCS owned by the U.S. Department of Defense (DoD) and the DoD cost in dollars per LOCS between 1950 and 2000 [28]. It conservatively estimated the figures for 2000 by multiplying 2 million DoD computers by 100 million executable machine instructions per computer, which gives 200 trillion LOCS. Based on a conservative $40 billion-per-year DoD software cost, the cost per LOCS is $0.0002. These cost improvements come largely from software reuse. One might object

that not all these LOCS add value for their customers. But one could raise the same objections for all transistors being added to chips each year and all the data packets transmitted across the internet. All three commodities pass similar market tests.

COTS components are also reprioritizing the skills needed by software engineers. Although a 2001 ACM Communications editorial stated, "In the end – and at the beginning – it's all about programming." [49], future trends are making this decreasingly true. Although infrastructure software developers will continue to spend most of their time programming, most application software developers are spending more and more of their time assessing, tailoring, and integrating COTS products. COTS hardware products are also becoming more pervasive, but they are generally easier to assess and integrate.

Figure 9 illustrates these trends for a longitudinal sample of small e-services applications, going from 28% COTS-intensive in 1996-97 to 70% COTS-intensive in 2001-2002, plus an additional industry-wide 54% COTS-based applications (CBAs) in the 2000 Standish Group survey [140][152]. COTS software products are particularly challenging to integrate. They are opaque and hard to debug. They are often incompatible with each other due to the need for competitive differentiation. They are uncontrollably evolving, averaging about to 10 months between new releases, and generally unsupported by their vendors after 3 subsequent releases. These latter statistics are a caution to organizations outsourcing applications with long gestation periods. In one case, an out-sourced application included 120 COTS products, 46% of which were delivered in a vendor-unsupported state [153].

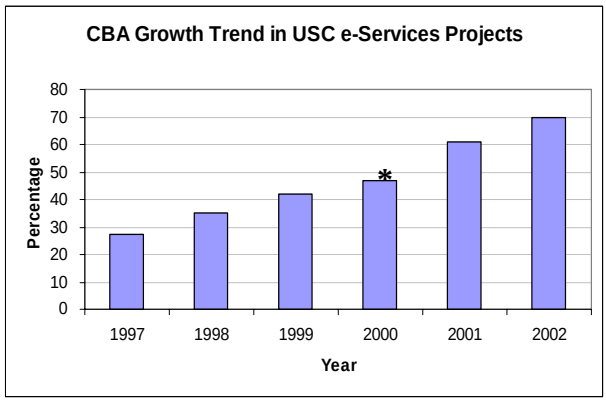

Figure 9. CBA Growth in USC E-Service Projects — *Standish Group, Extreme Chaos (2000)

Open source software, or an organization's reused or legacy software, is less opaque and less likely to go unsupported. But these can also have problems with interoperability and continuing evolution. In addition, they often place constraints on a new application's incremental development, as the existing software needs to be decomposable to fit the new increments' content and interfaces. Across the maintenance life cycle, synchronized refresh of a large number of continually evolving COTS, open source, reused, and legacy software and hardware components becomes a major additional challenge.

In terms of the trends discussed above, COTS, open source, reused, and legacy software and hardware will often have shortfalls in usability, dependability, interoperability, and localizability to different countries and cultures. As discussed above, increasing customer pressures for COTS usability, dependability, and interoperability, along with enterprise architecture initiatives, will reduce these shortfalls to some extent.

Model-Driven Development

Although COTS vendors' needs to competitively differentiate their products will increase future COTS integration challenges, the emergence of enterprise architectures and model-driven development (MDD) offer prospects of improving compatibility. When large global organizations such as WalMart and General Motors develop enterprise architectures defining supply chain protocols and interfaces [66], and similar initiatives such as the U.S. Federal Enterprise Architecture Framework are pursued by government organizations, there is significant pressure for COTS vendors to align with them and participate in their evolution.

MDD capitalizes on the prospect of developing domain models (of banks, automobiles, supply chains, etc.) whose domain structure leads to architectures with high module cohesion and low inter-module coupling, enabling rapid and dependable application development and evolvability within the domain. Successful MDD approaches were being developed as early as the 1950's, in which engineers would use domain models of rocket vehicles, civil engineering structures, or electrical circuits and Fortran infrastructure to enable user engineers to develop and execute domain applications [29]. This thread continues through business 4GL's and product line reuse to MDD in the lower part of Figure 6.

The additional challenge for current and future MDD approaches is to cope with the continuing changes in software infrastructure (massive distribution, mobile computing, evolving Web objects) and domain restructuring that are going on. Object–oriented models and meta-models, and service-oriented architectures using event-based publish-subscribe concepts of operation provide attractive approaches for dealing with these, although it is easy to inflate expectations on how rapidly capabilities will mature. Figure 10 shows the Gartner Associates assessment of MDA technology maturity as of 2003, using their "history of a silver bullet" rollercoaster curve. But substantive progress is being made on many fronts, such as Fowler's Patterns of Enterprise Applications Architecture book and the articles in two excellent MDD special issues in Software [102] and Computer [136].

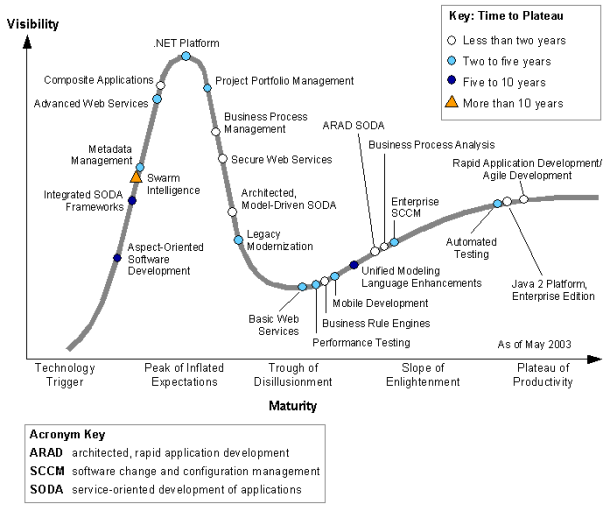

Figure 10. MDA Adoption Thermometer – Gartner Associates, 2003

Interacting software and Systems Engineering

The push to integrate application-domain models and software-domain models in MDD reflects the trend in the 2000's toward integration of software and systems engineering. Another driver in recognition from surveys such as [140] that the majority of software project failures stem from systems engineering shortfalls (65% due to lack of user input, incomplete and changing requirements, unrealistic expectations and schedules, unclear objectives, and lack of executive support). Further, systems engineers are belatedly discovering that they need access to more software skills as their systems become more software-intensive. In the U.S., this has caused many software institutions and artifacts to expand their scope to include systems, such as the Air Force Systems and Software Technology Center, the Practical Systems and Software Measurement Program and the Integrated (Systems and Software) Capability Maturity Model.

The importance of integrating systems and software engineering has also been highlighted in the experience reports of large organizations trying to scale up agile methods by using teams of teams [35]. They find that without up-front systems engineering and teambuilding, two common failure modes occur. One is that agile teams are used to making their own team's architecture or refactoring decisions, and there is a scarcity of team leaders that can satisfice both the team's preferences and the constraints desired or imposed by the other teams. The other is that agile teams tend to focus on easiest-first low hanging fruit in the early increments, to treat system-level quality requirements (scalability, security) as features to be incorporated in later increments, and to become unpleasantly surprised when no amount of refactoring will compensate for the early choice of an unscalable or unsecurable off-the-shelf component.

3. A View of the 2010's and Beyond

A related paper on the future of systems and software engineering process [38] identified eight surprise-tree trends and two 'wild-card' trends that would strongly influence future software and systems engineering directions. Five of the eight surprise-tree trends were just discussed under the 2000's: rapid change and the need for agility; increased emphasis on usability and value; software criticality and the need for dependability; increasing needs for COTS, reuse, and legacy software integration; and the increasing integration of software and systems engineering. Two of the eight surprise-tree trends will be covered next under the 2010's: global connectivity and massive systems of systems. Surprise-free computational plenty and the two wild-card trends (increasing software autonomy and combinations of biology and computing) will be covered under "2020 and beyond".

3.1 2010's Antitheses and Partial Syntheses: Globalization and Systems of Systems

The global connectivity provided by the Internet and low-cost, high-bandwidth global communications provides major economies of scale and network economies [7] that drive both an organization's product and process strategies. Location-independent distribution and mobility services create both rich new bases for synergetic collaboration and challenges in synchronizing activities. Differential salaries provide opportunities for cost savings through global outsourcing, although lack of careful preparation can easily turn the savings into overruns. The ability to develop across multiple time zones creates the prospect of very rapid development via three-shift operations, although again there are significant challenges in management visibility and control, communication semantics, and building shared values and trust.

On balance, though, the Computerworld "Future of IT" panelists felt that global collaboration would be commonplace in the future. An even stronger view is taken by the bestselling [66] book, The World is Flat: A Brief History of the 21st Century. It is based on extensive Friedman interviews and site visits with manufacturers and service organizations in the U.S., Europe, Japan, China and Taiwan; call centers in India; data entry shops in several developing nations; and software houses in India, China, Russia and other developing nations. It paints a picture of the future world that has been "flattened" by global communications and overnight delivery services that enable pieces of work to be cost-effectively outsourced anywhere in the world.

The competitive victors in this flattened world are these who focus on their core competencies; proactively innovative within and at the emerging extensions of their core competencies; and efficiently deconstruct their operations to enable outsourcing of non-core tasks to lowest-cost acceptable suppliers. Descriptions in the book of how this works at Wal-Mart and Dell provide convincing cases that this strategy is likely to prevail in the future. The book makes it clear that software and its engineering will be essential to success, but that new skills integrating software engineering with systems engineering, marketing, finance, and core domain skills will be critical.

This competition will be increasingly multinational, with outsourcees trying to master the skills necessary to become outsourcers, as their internal competition for talent drives salaries upward, as is happening in India, China, and Russia, for example. One thing that is unclear, though is the degree to which this dynamic will create a homogeneous global culture. There are also strong pressures for countries to preserve their national cultures and values.

Thus, it is likely that the nature of products and processes would also evolve toward the complementarity of skills in such areas as culture-matching and localization [49]. Some key culture-matching dimensions are provided in [77]: power distance, individualism/collectivism, masculinity/femininity, uncertainty avoidance, and long-term time orientation. These often explain low software product and process adoption rates across cultures. One example is the low adoption rate (17 out of 380 experimental users) of the more individual/masculine/short-term U.S. culture's Software CMM by organizations in the more collective/feminine/long-term Thai culture [121]. Another example was a much higher Chinese acceptance level of a workstation desktop organized around people, relations, and knowledge as compared to Western desktop organized around tools, folders, and documents [proprietary communication].

As with railroads and telecommunications, a standards-based infrastructure is essential for effective global collaboration. The Computerworld panelists envision that standards-based infrastructure will become increasingly commoditized and move further up the protocol stack toward applications.

A lot of work needs to be done to establish robust success patterns for global collaborative processes. Key challenges as discussed above include cross-cultural bridging; establishment of common shared vision and trust; contracting mechanisms and incentives; handovers and change synchronization in multi-timezone development; and culture-sensitive collaboration-oriented

groupware. Most software packages are oriented around individual use; just determining how best to support groups will take a good deal of research and experimentation.

Software-Intensive Systems of Systems

Concurrently between now and into the 2010's, the ability of organizations and their products, systems, and services to compete, adapt, and survive will depend increasingly on software and on the ability to integrate related software-intensive systems into systems of systems (SOS). Historically (and even recently for some forms of agile methods), systems and software development processes and maturity models have been recipes for developing standalone "stovepipe" systems with high risks of inadequate interoperability with other stovepipe systems. Experience has shown that such collections of stovepipe systems cause unacceptable delays in service, uncoordinated and conflicting plans, ineffective or dangerous decisions, and an inability to cope with rapid change.

During the 1990's and early 2000's, standards such as the International Organization for Standarization (ISO)/International Electrotechnical Commission (IEC) 12207 [1] and ISO/IEC 15288 [2] began to emerge that situated systems and software project processes within an enterprise framework. Concurrently, enterprise architectures such as the International Business Machines (IBM) Zachman Framework [155], Reference Model for Open Distributed Processing (RM-ODP) [127] and the U.S. Federal Enterprise Architecture Framework [3], have been developing and evolving, along with a number of commercial Enterprise Resource Planning (ERP) packages.

These frameworks and support packages are making it possible for organizations to reinvent themselves around transformational, network-centric systems of systems. As discussed in [77], these are necessarily software-intensive systems of systems (SISOS), and have tremendous opportunities for success and equally tremendous risks of failure. There are many definitions of "systems of systems" [83]. For this paper, the distinguishing features of a SOS are not only that it integrates multiple independently-developed systems, but also that it is very large, dynamically evolving, and unprecedented, with emergent requirements and behaviors and complex socio-technical issues to address. Table 1 provides some additional characteristics of SISOSs.

Characteristic	Range of Values
Size	10-100 million lines of code
Number of External Interfaces	30-300
Number of "Coopetitive" Suppliers	20-200
Depth of Supplier Hierarchy	6-12 levels
Number of Coordination Groups	20-200

Table 1. Complexity of SISOS Solution Spaces.

There is often a Lead System Integrator that is responsible for the development of the SOS architecture, identifying the suppliers and vendors to provide the various SOS components, adapting the architecture to meet evolving requirements and selected vendor limitations or constraints, then overseeing the implementation efforts and planning and executing the SOS level integration and test activities. Key to successful SOS development is the ability to achieve timely decisions with a potentially diverse set of stakeholders, quickly resolve conflicting needs, and coordinate the activities of multiple vendors who are currently working together to provided capabilities for the SOS, but are often competitors on other system development efforts (sometimes referred to as "coopetitive relationships").

In trying to help some early SISOS programs apply the risk-sriven spiral model, I've found that that spiral model and other process, product, cost, and schedule models need considerable rethinking, particularly when rapid change needs to be coordinated among as many stakeholders as are shown in Table 1. Space limitations make it infeasible to discuss these in detail, but the best approach evolved so far involves, in Rational Unified Process terms:

1. Longer-than-usual Inception and Elaboration phases, to concurrently engineer and validate the consistency and feasibility of the operation, requirements, architecture, prototypes, and plans; to select and harmonize the suppliers; and to develop validated baseline specifications and plans for each validated increment.
2. Short, highly stabilized Construction-phase increments developed by a dedicated team of people who are good at efficiently and effectively building software to a given set of specifications and plans.
3. A dedicated, continuous verification and validation (V&V) effort during the Construction phase by people who are good at V&V, providing rapid defect feedback to the developers.
4. A concurrent agile-rebaselining effort by people who are good at rapidly adapting to change and renegotiating the specifications and plans for the next Construction increment.

Further elaboration of the top SISOS risks and the process above are in [35] and [39]. Other good SISOS references are [95], [135], and [50].

3.2 2020 and Beyond

Computational Plenty Trends

Assuming that Moore's Law holds, another 20 years of doubling computing element performance every 18 months will lead to a performance improvement factor of 220/1.5 = 213.33 = 10,000 by 2025. Similar factors will apply to the size and power consumption of the competing elements.

This computational plenty will spawn new types of platforms (smart dust, smart paint, smart materials, nanotechnology, micro electrical-mechanical systems: MEMS), and new types of applications (sensor networks, conformable or adaptive materials, human prosthetics). These will present software engineering challenges for specifying their configurations and behavior; generating the resulting applications; verifying and validating their capabilities, performance, and dependability; and integrating them into even more complex systems of systems.

Besides new challenges, though, computational plenty will enable new and more powerful software engineering approaches. It will enable new and more powerful self-monitoring software and computing via on-chip co-processors for assertion checking, trend analysis, intrusion detection, or verifying proof-carrying code. It will enable higher levels of abstraction, such as pattern-oriented programming, multi-aspect oriented programming, domain-oriented visual component assembly, and programming by example with

expert feedback on missing portions. It will enable simpler brute-force solutions such as exhaustive case evaluation vs. complex logic.

It will also enable more powerful software and systems engineering tools that provide feedback to developers based on domain knowledge, programming knowledge, systems engineering knowledge, or management knowledge. It will enable the equivalent of seat belts and air bags for user-programmers. It will support show-and-tell documentation and much more powerful system query and data mining techniques. It will support realistic virtual game-oriented systems and software engineering education and training. On balance, the added benefits of computational plenty should significantly outweigh the added challenges.

Wild Cards: Autonomy and Bio-Computing

"Autonomy" covers technology advancements that use computational plenty to enable computers and software to autonomously evaluate situations and determine best-possible courses of action. Examples include:

- Cooperative intelligent agents that assess situations, analyze trends, and cooperatively negotiate to determine best available courses of action.
- Autonomic software, that uses adaptive control techniques to reconfigure itself to cope with changing situations.
- Machine learning techniques, that construct and test alternative situation models and converge on versions of models that will best guide system behavior.
- Extensions of robots at conventional-to-nanotechnology scales empowered with autonomy capabilities such as the above.

Combinations of biology and computing include:

- Biology-based computing, that uses biological or molecular phenomena to solve computational problems beyond the reach of silicon-based technology.
- Computing-based enhancement of human physical or mental capabilities, perhaps embedded in or attached to human bodies or serving as alternate robotic hosts for (portions of) human bodies.

Examples of books describing these capabilities are Kurzweil's The Age of Spiritual Machines [86] and Drexler's books Engines of Creation and Unbounding the Future: The Nanotechnology Revolution [57][58]. They identify major benefits that can potentially be derived from such capabilities, such as artificial labor, human shortfall compensation (the five senses, healing, life span, and new capabilities for enjoyment or self-actualization), adaptive control of the environment, or redesigning the world to avoid current problems and create new opportunities.

On the other hand, these books and other sources such as Dyson's Darwin Among the Machines: The Evolution of Global Intelligence [61] and Joy's article, "Why the Future Doesn't Need Us" [83], identify major failure modes that can result from attempts to redesign the world, such as loss of human primacy over computers, over-empowerment of humans, and irreversible effects such as plagues or biological dominance of artificial species. From a software process standpoint, processes will be needed to cope with autonomy software failure modes such as undebuggable self-modified software, adaptive control instability, interacting agent commitments with unintended consequences, and commonsense reasoning failures.

As discussed in Dreyfus and Dreyfus' Mind Over Machine [59], the track record of artificial intelligence predictions shows that it is easy to overestimate the rate of AI progress. But a good deal of AI technology is usefully at work today and, as we have seen with the Internet and World Wide Web, it is also easy to underestimate rates of IT progress as well. It is likely that the more ambitious predictions above will not take place by 2020, but it is more important to keep both the positive and negative potentials in mind in risk-driven experimentation with emerging capabilities in these wild-card areas between now and 2020.

4. Conclusions

4.1 Timeless Principles and Aging Practices

For each decade, I've tried to identify two timeless principles headed by plus signs; and one aging practice, headed by a minus sign.

From the 1950's

+ Don't neglect the sciences. This is the first part of the definition of "engineering". It should not include just mathematics and computer science, but also behavioral sciences, economics, and management science. It should also include using the scientific method to learn through experience.
+ Look before you leap. Premature commitments can be disastrous (Marry in haste; repent at leisure – when any leisure is available).
– Avoid using a rigorous sequential process. The world is getting too tangeable and unpredictable for this, and it's usually slower.

From the 1960's

+ Think outside the box. Repetitive engineering would never have created the Arpanet or Engelbart's mouse-and-windows GUI. Have some fun prototyping; it's generally low-risk and frequently high reward.
+ Respect software's differences. You can't speed up its development indefinitely. Since it's invisible, you need to find good ways to make it visible and meaningful to different stakeholders.
– Avoid cowboy programming. The last-minute all-nighter frequently doesn't work, and the patches get ugly fast.

From the 1970's

+ Eliminate errors early. Even better, prevent them in the future via root cause analysis.
+ Determine the system's purpose. Without a clear shared vision, you're likely to get chaos and disappointment. Goal-question-metric is another version of this.
– Avoid Top-down development and reductionism. COTS, reuse, IKIWISI, rapid changes and emergent requirements make this increasingly unrealistic for most applications.

From the 1980's

+ These are many roads to increased productivity, including staffing, training, tools, reuse, process improvement, prototyping, and others.
+ What's good for products is good for process, including architecture, reusability, composability, and adaptability.

– Be skeptical about silver bullets, and one-size-fits-all solutions.

From the 1990's

+ Time is money. People generally invest in software to get a positive return. The sooner the software is fielded, the sooner the returns come – if it has satisfactory quality.
+ Make software useful to people. This is the other part of the definition of "engineering."
– Be quick, but don't hurry. Overambitious early milestones usually result in incomplete and incompatible specifications and lots of rework.

From the 2000s

+ If change is rapid, adaptability trumps repeatability.
+ Consider and satisfice all of the stakeholders' value propositions. If success-critical stakeholders are neglected or exploited, they will generally counterattack or refuse to participate, making everyone a loser.
– Avoid falling in love with your slogans. YAGNI (you aren't going to need it) is not always true.

For the 2010's

+ Keep your reach within your grasp. Some systems of systems may just be too big and complex.
+ Have an exit strategy. Manage expectations, so that if things go wrong, there's an acceptable fallback.
– Don't believe everything you read. Take a look at the downslope of the Gartner rollercoaster in Figure 10.

4.2 Some Conclusions for Software Engineering Education

The students learning software engineering over the next two decades will be participating their profession well into the 2040's, 2050's, and probably 2060's. The increased pace of change continues to accelerate, as does the complexity of the software-intensive systems or systems of systems that need to be perceptively engineered. This presents many serious but exciting challenges to software engineering education, including:

- Keeping courses and courseware continually refreshed and up-to-date;
- Anticipating future trends and preparing students to deal with them;
- Monitoring current principles and practices and separating timeless principles from out-of-date practices;
- Packaging smaller-scale educational experiences in ways that apply to large-scale projects;
- Participating in leading-edge software engineering research and practice, and incorporating the results into the curriculum;
- Helping students learn how to learn, through state-of-the-art analyses, future-oriented educational games and exercises, and participation in research; and
- Offering lifelong learning opportunities as software engineers continue to practice their profession.

Acknowledgements and Apologies

This work was partly supported by NSF grants CCF-0137766 for the IMPACT project, my NSF "Value-Based Science of Design" grant, and the Affiliates of the USC Center for Software Engineering.

I also benefited significantly from interactions with software pioneers and historians at the 1996 Dagstuhl History of Software Engineering seminar, the SD&M Software Pioneers conference, and the NSF/ACM/IEE (UK) IMPACT project.

Having said this, I am amazed at how little I learned at those events and other interactions that has made it into this paper. And just glancing at my bookshelves, I am embarrassed at how many significant contributions are not represented here, and how much the paper is skewed toward the U.S. and my own experiences. My deepest apologies to those of you that I have neglected.

5. REFERENCES

[1] ISO. Standard for Information Technology – Software Life Cycle Processes. ISO/IEC 12207, 1995.

[2] ISO. Systems Engineering – System Life Cycle Processes. ISO/IEC 15288, 2002.

[3] FCIO (Federal CIO Council), A Practical Guide to Federal Enterprise Architecture, Version 1.0, FCIO, Washington, DC, February 2001. zaqi4.

[4] Alford, M.W., Software Requirements Engineering Methodology (SREM) at the Age of Four, *COMPSAC 80* Proceedings, (October 1980) pp 366-374.

[5] Anthes, G., The Future of IT. *Computerworld*, (March 7, 2005) 27-36

[6] Aron, J. *Software Engineering*, NATO Science Committee Report, January 1969.

[7] Arthur, W. B., Increasing Returns and the New World of Business. *Harvard Business Review* (July/August, 1996) 100-109

[8] Baker, F. T. Structured programming in a production programming environment. *In Proceedings of the international Conference on Reliable Software.* Los Angeles, California, April 21 - 23, 1975.

[9] Balzer, R.M. A Global View Of Automatic Programming, *Proc. 3rd Joint Conference On Artificial Intelligence*, August, 1973, pp. 494-499.

[10] Balzer, R. M., T. E. Cheatham, and C. Green, Software Technology in the 1990's: Using a New Paradigm, *Computer*, Nov. 1983, pp. 3945.

[11] Basili V. R. and M. V. Zelkowitz, Analyzing medium scale software development, *Third International Conf. On Software Engineering*, Atlanta, Ga. (May, 1978) 116-123.

[12] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, Addison-Wesley, 1998.

[13] Bass, L. and Coutaz, J., *Developing Software for the User Interface*, Addison Wesley, 1991

[14] Beck, K. Extreme Programming Explained, Addison-Wesley, 2000

[15] Belady, L. A. and Lehman, M. M., Characteristics of large systems, in *Research Directions in Software Technology*, P.Wegner (editor), MIT-Press, Cambridge, Massachusetts, 1979

[16] Benington, H. D., Production of Large Computer Programs, Proceedings of the *ONR Symposium on Advanced Program Methods for Digital Computers*, June 1956, pp. 15 - 27. Also available in Annals of the History of Computing, October 1983, pp. 350 - 361.

[17] Berners-Lee, T., *World Wide Web Seminar*. http://www.w3.org/Talks/General.html (1991)

[18] Berners-Lee, T., Cailliau, R., Luotonen, A., Frystyk , H., Nielsen, F., and Secret, A.. The World-Wide Web, in *Comm. ACM* (CACM), 37(8), 76-82, 1994.

[19] Best Current Practices: Software Architecture Validation, AT&T Bell Labs, 1990.

[20] Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., Gruenbacher, P. (eds.). *Value-Based Software Engineering*. Springer Verlag (2005)

[21] Bjorner, D., On the use of formal methods in software development. In Proceedings of the 9th International Conference on Software Engineering (Monterey, California, United States), 1987.

[22] Boehm, B., Software and its impact: a quantitative assessment. *Datamation*, pages 48-59, May 1973.

[23] Boehm, B. W., J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod, M. J. Merritt, *Characteristics of Software Quality*, TRW Software Series TRW-SS-73-09, December 1973.

[24] Boehm, B., Software engineering. *IEEE Trans. Computers*, 100(25):1226-1241, 1976.

[25] Boehm, B., Software Engineering: As it is. ICSE 1979: 11-21.

[26] Boehm, B., *Software Engineering Economics*, Prentice-Hall 1981

[27] Boehm, B., Standish, T. A. Software Technology in the 1990's: Using an Evolutionary Paradigm. *IEEE Computer* 16(11): 30-37 (1983)

[28] Boehm, B., A Spiral Model of Software Development and Enhancement, *Computer*, May 1988, pp. 61-72.

[29] Boehm B., An Early Application generator and Other Recollections, in Glass, R. L., *In The Beginning: Recollections of Software Pioneers*, IEEE CS Press, 1988

[30] Boehm, B., *Software Risk Management*, CS Press, Los Alamitos, Calif., 1989.

[31] Boehm, B. and Ross, R., Theory-W Software Project Management: Principles and Examples, *IEEE Trans. SW Engineering*., July 1989, pp. 902-916.

[32] Boehm, B., A Spiral Model for Software Development and Enhancement, *Computer*, vol. 21, May 1988, pp. 61-72.

[33] Boehm, B., Anchoring the Software Process, *IEEE Software*, Vol. 13, No. 14, July 1996

[34] Boehm, B., Managing Software Productivity and Reuse. *Computer* 32, 9 (Sep. 1999), 111-113

[35] Boehm, B., Brown, A.W, Basili, V. and Turner R., Spiral Acquisition of Software-Intensive Systems. *CrossTalk*, May 2004, pp. 4-9.

[36] Boehm, B., Turner, R., *Balancing Agility and Discipline*, Addison Wesley (2004)

[37] Boehm, B., Jain, A. An Initial Theory of Value-Based Software Engineering. In: Aurum, A., Biffl, S., Boehm, B., Er-dogmus, H., Gruenbacher, P. (eds.): *Value-Based Software Engineering*, Springer Verlag (2005)

[38] Boehm, B., Some Future Trends and Implications for Systems and Software Engineering Processes, *Systems Engineering*, vol. 9, No. 1, 2006, pp 1-19.

[39] Boehm, B. and Lane, J., 21st Century Processes for Acquiring 21st Century Software Intensive Systems of Systems, *Cross Talk*, May 2006 (to appear)

[40] Böhm, C. and Jacopini, G. Flow diagrams, turing machines and languages with only two formation rules. *Comm. ACM* 9, 5 (May. 1966), 366-371

[41] Booch, G., Rumbaugh, J. and Jacobson, L. *The Unified Modeling Language User Guide*. Addison-Wesley Longman Inc., 1999.

[42] Brooks, F. P., The Mythical Man-Month, Addison Wesley, 1975.

[43] Brooks, F. P., No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10-19, April 1987.

[44] Buxton J. N. and Randell B. (Eds.) *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*, Rome, Italy, 27-31 Oct. 1969. Scientific Affairs Division, NATO, Brussels (May 1970).

[45] PDL/74 Program Design Language Reference Guide (Processor Version 3), Caine Farber Gordon Inc., 1977

[46] Charette, R. N., Software Engineering Risk Analysis and Management, McGraw-Hill, 1989.

[47] Clemm, G., The Workshop System - A Practical Knowledge-Based Software Environment, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM SIGPLAN Notices, 13, 5, pp. 55-64 (Nov 1988).

[48] Constantine, L. L. The Programming Profession, Programming Theory, and Programming Education. *Computers and Automation* 17,2 (Feb. 1968) pp. 14-19.

[49] Crawford, D. Editorial Pointers. *Comm. ACM* (October, 2001) 5.

[50] Cooper, J. D., *Characteristics of the Average Coder*, personal communication, May 1975.

[51] Conrow, E. H., Risk Management for Systems of Systems, *CrossTalk*, v. 18, n. 2 (February 2005), pages 8-12.

[52] Cusumano, M. A. The Software Factory: A Historical Interpretation. *IEEE Softw.* 6, 2 (Mar. 1989), 23-30.

[53] Cusumano, M., Selby, R.: Microsoft's Secrets. HarperCollins (1996)

[54] Dahl, O., and Nygaard, K. Simula: a language for programming and description of discrete event systems. Tech. rep., Norwegian Computing Center, 1967.

[55] DeMarco, T., Structured analysis and system specification. Prentice Hall, 1978.

[56] Dijkstra, E. *Cooperating Sequential Processes*. Academic Press, 1968

[57] Drexler, E.K.: Engines of Creation. Anchor Press (1986)

[58] Drexler, K.E., Peterson, C., Pergamit, G.: Unbounding the Future: The Nanotechnology Revolution. William Morrow & Co. (1991)

[59] Dreyfus, H., Dreyfus, S.: Mind over Machine. Macmillan (1986)

[60] Druffel, L.E.. and Buxton, J.N. Requirements for an Ada programming support environment: Rationale for STONEMAN. *In Proceedings of COMPSAC 80. IEEE Computer Society*, (Oct. 1980), 66-72.

[61] Dyson, G. B.: Darwin Among the Machines: The Evolution of Global Intelligence, Helix Books/Addison Wesley (1997)

[62] Ehn, P. (ed.): *Work-Oriented Design of Computer Artifacts*, Lawrence Earlbaum Assoc. (1990)

[63] Elssamadisy, A. and Schalliol, G., Recognizing and Responding to 'Bad Smells' in Extreme Programming, *Proceedings, ICSE 2002*, pp. 617-622

[64] Estublier, J., Leblang, D., Hoek, A., Conradi, R., Clemm, G., Tichy, W., and Wiborg-Weber, D. Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol.* 14, 4 (Oct. 2005), 383-430.

[65] Fagan, M.E., Design and Code inspections to reduce errors in program development, 1976, *IBM Systems Journal*, Vol. 15, No 3, Page 258-287

[66] Friedman, T. L., The World Is Flat: A Brief History of the Twenty-First Century. Farrar, Straus & Giroux. New York. 2005

[67] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, MA. (1994)

[68] Grant, E., and H. Sackman., An Exploratory Investigation of Programmer Performance Under On-Lineand Off-Line Conditions. Report SP-2581, System Development Corp., September 1966.

[69] Gosling, J., Joy, B., Steele, G., The Java Language Specification, Sun Microsystems, Inc. (1989)

[70] Hoare, C. A. R., An axiomatic basis for computer programming. *Comm. ACM*, 12:576--583, 1969.

[71] Hoare, C. A. R., Assertions: A Personal Perspective, *IEEE Annals of the History of Computing*, v.25 n.2, p.14-25, April 2003

[72] Floyd, C. *Records and References in Algol-W*, Computer Science Department, Stanford University, Stanford, California, 1969

[73] Freiman, F. R. and Park, R. E., *The PRICE Software Cost Model*, RCA Price Systems, Cherry Hill, NJ, Feb. 1979.

[74] Highsmith, J. *Adaptive Software Development*. Dorset House (2000)

[75] Guttag, J. V., *The Specification and Application to Programming of Abstract Data Types*, Ph.D. dissertation, Computer Science Department, University of Toronto, Canada, 1975.

[76] Halstead, M., *Elements of Software Science*, North Holland, 1977

[77] Harned, D., Lundquist, J. "What Transformation Means for the Defense Industry". The McKinsey Quarterly, November 3, 2003: 57-63.

[78] Holtzmann, G, *The SPIN Model Checker*, Addison Wesley, 2004

[79] Hofstede, G., *Culture and Organizations*. McGraw Hill (1997)

[80] Hosier, W. A., Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming, *IRE Trans. Engineering Management*, EM-8, June, 1961.

[81] Humphrey, W. S., *Managing the Software Process*. Reading, MA: Addison-Wesley, 1989

[82] Jackson M. A., *Principle of Program Design*, Acad. Press, 1975

[83] Jacobson, I., Griss, M., Jonsson, P., Software Reuse: Architecture, Process and Organization for Business Success, Addison Wesley, 1997

[84] Jacobson, I., Booch, G. and Rumbaugh, J., *Unified Software Development Process*, Addison-Wesley, 1999

[85] Joy, B.: Why the Future Doesn't Need Us: Wired (April, 2000)

[86] Kurzweil, R., *The Age of Spiritual Machines*. Penguin Books (1999)

[87] Kruchten, P. B., The Rational Unified Process (An Introduction). Addison Wesley 1999

[88] Larman, C., Agile and Iterative Development: A Manager's Guide, Addison-Wesley, 2004.

[89] Lane, J., Valerdi, R. "Synthesizing System-of-Systems Concepts for Use in Cost Estimation," IEEE SMC, 2005.

[90] Levy, S., Hackers: Heroes of the Computer Revolution, Anchor Press/Doubleday, 1984

[91] Lientz, B.P. and E.B. Swanson, *Software Maintenance Management*, Addison-Wesley, Reading, Mass., 1980

[92] Lim, W.C., *Managing Software Reuse*, Prentice Hall, 1998

[93] Liskov, B. and Zilles, S. N., Programming with abstract data types. In Proceedings of *Symposium on Very High Level Programming Languages*, 1974

[94] Littlewood, B. and Verrall, J. L., A Bayesian Reliability Growth Model for Computer Software, *Appl. Statist.*, Vol. 22, pp. 332-346 (1973)

[95] Maier, M., Architecting Principles for Systems of Systems, Proceedings of the Sixth Annual International Symposium, International Council on Systems Engineering, Boston, 1996, pages 567-574.

[96] Marca, D. and Bock, G. eds., *Groupware*, IEEE CS Press, 1992

[97] Maranzano, J. F., Sandra A. Rozsypal, Gus H. Zimmerman, Guy W. Warnken, Patricia E. Wirth, David M. Weiss, Architecture Reviews: Practice and Experience, *IEEE Software*, vol. 22, no. 2, pp. 34-43, Mar/Apr, 2005.

[98] Martin, J., Rapid Applications Development, Macmillan, 1991

[99] Maslow, A., *Motivation and Personality*, Harper and Row (1954)

[100] Matsumoto, Y., Evaluation of the digital prediction filter applied to control a class of sevomotor by microcomputers, *IEEE Transactions on IECI*, Vol.IECI-23,No.4,pp.359-363(Nov. 1976)

[101] McCabe, T., A Software Complexity Measure, *IEEE Trans. Software Engineering* Vol 2, No 12, 1976.

[102] McCracken, D.D. A maverick approach to systems analysis and design. In System Analysis and Design: A Foundation for the 1980's, W.W. Cotterman, J.D. Gouger, N.L. Enger, and F. Harold, Eds. North-Holland, Amsterdam, 1981, pp. 446-451.

[103] McConnell, S. *Rapid Development*. Microsoft Press, 1996

[104] Meller, S. J., Clark, A. N., and Futagami, T., Model-Driven Development, *IEEE Software*, (Special Issue on Model-Driven Development) Vol. 20, No. 5, 2003, 14-18

[105] Meyer, B. *Object-Oriented Software Construction*, Second Edition. Prentice Hall, 1997

[106] Myers, G., *Reliable Systems through Composite Design*. New York: Petrocelli/Charter, 1975.

[107] National Institute of Standards and Technology, Gaithersberg. *Reference Model for Frameworks of Software Engineering Environments*, draft version 1.5 edition, 1991.

[108] Naur, P. and Randell, B. (Eds.). *Software Engineering: Report of a conference sponsored by the NATO Science Committee* (7–11 Oct. 1968), Garmisch, Germany. Brussels, Scientific Affairs Division, NATO, 1969.

[109] Neighbors, J.M., The Draco Approach to constructing Software from reusable components, *IEEE Trans. Software Engineering*, vol.SE-10, No.5, pp.564-574, September, 1984

[110] Nelson, E., Management Handbook for the Estimation of Computer Programming Costs, Systems Development Corporation, Oct. 1966

[111] Nunamaker, J. et. al., "Lessons from a Dozen years of Group Support Systems Research: A Discussion of lab and Field Findings," *J.MIS*, vol. B, no.3 (1996-1997), pp. 163-207

[112] Osterweil, L., Software Processes are Software Too. *Proceedings, In Ninth International Conference on Software Engineering*, (Monterey, CA, 1987), ACM, IEEE, LosAlamitos, CA, 2-13.

[113] Osterweil, L., Ghezzi, C., Kramer, J., and Wolf, A. 2005. Editorial. *ACM Trans. Softw. Eng. Methodol.* 14, 4 (Oct. 2005), 381-382.

[114] Paulk, M., Weber, C., Curtis, B., Chrissis, M., *The Capability Maturity Model*. Addison Wesley (1994)

[115] Paulk, M. C., Weber, C. V., Curtis B, Chrissis M. B. et al., *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison Wesley, 1995.

[116] Parnas, D.L. On the Criteria To Be Used in Decomposing Systems Into Modules, *Comm. ACM*, Vol. 5, No. 12, December 1972, pp. 1053-1058.

[117] Parnas, D. L. On the design and development of program families, *IEEE Trans. Software Engineering* 2(1), 1-9, 1976.

[118] Parnas, D. L. Designing software for ease of extension and contraction, *IEEE Trans. Software Engineering* SE-5(2), 128-138, 1979.

[119] Patterson, D.: 20th Century vs. 21st Century C&C: The SPUR Manifesto. *ACM Comm*. (March, 2005) 15-16

[120] Penedo, M. H. and Stuckle, E. D., PMDB--A Project Master Database for Software Engineering Environments. In *Proceedings of the 8th International Conference on Software Engineering*, pages 150--157. IEEE Computer Society, August 1985.

[121] Perry, D. E. and Wolf, A. L., Foundations for the study of software architecture. *Software Engineering Notes*, vol 17, no 4, October 1992

[122] Phongpaibul, M., Boehm, B.: Improving Quality Through Software Process Improvement in Thailand: Initial Analysis. *Proceedings, ICSE 2005 Workshop on Software Quality* (May, 2005)

[123] PITAC (President's Information Technology Advisory Committee), Report to the President: Information Technology Research: Investing in Our Future (1999)

[124] Poore, J. H. and Carmen J. Trammell, Bringing Respect to Testing Through Statistical Science, *American Programmer*, Vol. 10, No. 8, August, 1997.

[125] Poulin, J.S., *Measuring Software Reuse*, Addison Wesley, 1997.

[126] Putnam, L. H., A General Empirical Solution to the Macro Software Sizing and Estimating Problem, *IEEE Trans.* Software Engineering, Vol 4, No 4, 1978

[127] Putman, J. Architecting with RM-ODP. Prentice Hall, 2001.

[128] Raymond, E.S, *The Cathedral and the Bazaar*, O'Reilly, 1999

[129] Reifer, D.J., *Practical Software Reuse*, Wiley, 1997

[130] Redwine, S. and Riddle, W., Software Technology maturation, *Proceedings of the 8th ICSE*, 1985.

[131] Ross, D. Structured Analysis (SA): A Language for Communicating Ideas. *IEEE Trans. Software Engineering*, SE-3, 1 (Jan. 1977). 16-34.

[132] Royce, W. W., Managing the Development of Large Software Systems: Concepts and Techniques, Proceedings of *WESCON*, August 1970

[133] Royce, W., Software Project Management - A Unified Framework, Addison Wesley 1998

[134] Ryder, B. G., Soffa, M. L., and Burnett, M. 2005. The impact of software engineering research on modern progamming languages. *ACM Trans. Softw. Eng. Methodol.* 14, 4 (Oct. 2005)

[135] Sage, A., Cuppan, C., On the Systems Engineering and Management of Systems of Systems and Federations of Systems, *Information, Knowledge, and Systems Management*, v. 2 (2001), pages 325-345.

[136] Schmidt, D. C., Model-Driven Engineering, *IEEE Computer*, 39(2), February 2006.

[137] Shaw, M. and Garlan, D., *Software Architecture. Perspectives on an emerging discipline*, Prentice Hall, Upper Saddle River, NJ 07458, (1996).

[138] Shneiderman, B., Software psychology: human factors in computer and information systems. 1980: Cambridge, Mass, Winthrop Publishers. 320.

[139] Shooman, M.L., Probabilistic Models for Software Reliability Prediction. *Statistical Computer Performance Evaluation*, Academic Press, New York, pp. 485-502 (1972)

[140] Stallman, R. M., Free Software Free Society: Selected Essays of Richard M. Stallman, GNU Press, 2002

[141] Standish Group, *Extreme Chaos*, http://www.standishgroup.com (2001)

[142] Stroustrup, B., The C++ Programming Language. Addison-Wesley, 1986

[143] Teichroew, D. and Sayani, H. Automation of system building. Datamation 17,16 (August 15, 1971), 25-30.

[144] The Open Road: A History of Free Software, http://www.sdmagazine.com/opensourcetimeline/sources.htm

[145] Toulmin, S., *Cosmopolis*, University of Chicago Press (1992)

[146] Tracz, W., Test and analysis of Software Architectures, *In Proceedings of the international Symposium on software Testing and Analysis (ISSTA '96)*, ACM press, New York, NY, pp 1-3.

[147] Webster's Collegiate Dictionary, Merriam-Webster, 2002.

[148] Weinberg, G.M., The Psychology of Computer Programming, New York: Van Nostrand Reinhold (1971)

[149] Wirth, N. The programming language Pascal. Acta Infformatica, 1:35--63, 1971.

[150] Wirth, N. Programming in Modula-2. Springer, 1974.

[151] Wulf, W. A., London, R., and Shaw, M., An Introduction to the Construction and Verification of Alphard Program. IEEE Trans. Software Engineering, SE-2, 1976, pp. 253-264

[152] Yang, Y., Bhuta, J., Port, D., and Boehm, B.: Value-Based Processes for COTS-Based Applications. IEEE Software (2005)

[153] Yang, Y., Boehm, B., and Port, D. A Contextualized Study of COTS-Based E-Service Projects. Proceedings, Software Process Workshop 2005 (May, 2005)

[154] Yourdon E. and Constantine, L. L., Structured Design. Yourdon Press, 1975.

[155] Zachman, J. "A Framework for Information Systems Architecture." IBM Systems Journal, 1987.