Five decades of software development have come and gone, and defective software is the norm: high quality, the exception. From the Millennium Bug to holes in network security, through a litany of catastrophic software defects in between: You have to look back and wonder why we still don’t have general-purpose technologies that let all programmers write reliable software at a reasonable cost in a reasonable amount of time.

At the outset of the 21st century, it seems appropriate to assess the last 50 years. The first experiments in modern programming occurred during World War II, with the 1950s representing the first full decade in which software development took place. From that time up to and including the dawn of this new millennium, we have witnessed a procession of astonishing advances in the types of problems software can solve and the very form that software solutions take. Likewise, software developers’ attitudes and work habits have undergone remarkable changes. Technological advances in computer hardware, operating systems, and programming languages have helped shape the software development field. However, social and economic factors have played, perhaps, a larger role by determining how industry used these advances, who ended up using them, and what—if any—influence they’ve had on our ability to produce high-quality software.

Although a complete catalog of the past 50 years would be impossible in this short article, we can give a decade-by-decade synopsis of software development theory and practice, focusing particularly on attitudes and trends that have shaped current software development methods. Perhaps by examining past trends—successes and failures—we can uncover clues as to what avenues to explore in improving future software systems.

1950 TO 1959: GENESIS

The practice of programming computers arose in part from US military needs in World War II. From computing bomb trajectories to decrypting enemy communications, the war fed the need for better and faster ways to compute. The problems that required immediate solution were numerical and computationally intense: They drove the best and brightest to develop machinery to solve them.

The first “computers” were human beings, mostly women who hadn’t been accepted into the US armed forces in the 1940s. They cranked mechanical adding machines in great assembly lines. It’s intriguing to think of a program in those terms: a line of women standing at their stations, each performing her portion of a bomb trajectory calculation and handing the answer off to a colleague to compute the next step.
But under the urgency of war, speed was just as important as accuracy, and Presper Eckert’s and John Mauchley’s ENIAC computer offered both. Further, the war offered a funding opportunity that might not otherwise have come about. The ENIAC was more electrical than mechanical—it used electricity not just to drive the mechanical parts but to actually calculate. This machine could not only produce results, but it could automatically use those results in other calculations. However, the ENIAC wasn’t ready in time and did not make a single computation in aid of the war (Scott McCartney, ENIAC: The Triumphs and Tragedies of the World’s First Computer, Walker and Company, New York, 1999). It wasn’t until after the war that the world felt the impact of the first electrical computers.

Postwar uses for computers tended to follow the wartime themes. Computers existed to solve highly mathematical problems, and their first programmers were mostly the people who defined and derived the equations. These physicists and mathematicians worked the way you might expect. They designed algorithms carefully. They rigorously documented, peer-reviewed, and implemented mathematical proofs. Never in the history of software development have more meticulous minds addressed the task of programming.

However, by modern standards, the problems these talented pioneers solved were relatively straightforward. This is not to say that the algorithms and mathematics were simple—quite the contrary. But the 1950s era of computers and programming environments were capable of only the most basic instructions and operations. Modern operating systems with thousands of built-in functions and services did not exist. Hardware remained uncomplicated by flexible communication protocols and programmable peripherals and devices.

Thus, the 1950s saw extraordinarily talented people solving problems that they intimately understood in programming environments that had few complicated instructions. In other words, it was a decade of smart people solving well-understood problems: a recipe for success by all accounts and a perfect way to start the discipline of software development.

1960 TO 1969: EXODUS

By the late 1950s, computers had become quite a phenomenon, and at the dawn of the 1960s, the discipline of software development went public. Universities began offering degree programs in this new technology, and the number of hardware manufacturers grew rapidly. Suddenly, computer hardware and training were accessible to the general public—or at least to the subset that attended college.

At the same time, computers were undergoing big advances in usability and capability. The problems they could solve grew in scope and complexity. The programming languages designed to solve those problems were also becoming more powerful and easier to apply. The 1960s were a phenomenal growth period for computing technology and set the tone for the remainder of the century.

The 1960s also offered the industry’s first chance to go astray. Less rigorous minds were tackling harder problems. (Indeed, how could the industry have found more rigorous, meticulous developers than it had in the 1950s?) This was the perfect recipe for disaster—but the disaster never happened. Software written during the 1960s attained the same high quality as programs written in the previous decade.

This seeming paradox has a simple, though unobvious, explanation: The factor that kept programmers honest and program quality high in the 1960s was the unavailability of personal compilers.

Compilation in the 1960s was not an easy endeavor. For the most part, a company or university owned only a single, huge computer. This computer’s compiler was, first, located a long walk from the programmer’s office; second, so heavily booked that access required advance reservations; and, third, painfully unsympathetic to misuses of programming language syntax and constructs.

In other words, compiling a program that wasn’t near perfect was a significant waste of time and effort, and it could lead to substantial rework. Imagine walking a half mile across campus with a nine-inch stack of punched cards only to have the stack of cards rejected by a “type mismatch” error on card 30. This could result in days, even weeks, of delay before another session with the sole compiler became available.

This painful compilation process kept programmers at their office desks—checking, soliciting peer review, and reading, reading, reading their cards (the code) until they had exhausted every available avenue of review. No measures were too extreme, because the price for sloppiness was severe.

So, in the 1960s, as complexity grew and less rigorous-minded people tackled the problems, the discipline of software development had leaped its first serious hurdle—software’s debut to the wider public was a success.

1970 TO 1979: CHAOS

The 1970s were not good years for software quality advocates. The challenges of the 1960s—harder problems and less-trained practitioners—worsened. On the flip side, inaccessible and time-consuming compilation became a
thing of the past. The advent of the PC changed the rules of programming, removing the constraints that kept quality high in the 1960s.

Desktop computing made the computer a tool for truly all people—not just mathematicians, university researchers, and military strategists. No longer did anyone need to wait hours, days, or weeks for the privilege of compiler access, because every PC could have a built-in compiler. Compilation was available any time a programmer wanted a quick syntax check. Why bother with all that desk checking when you could consult a compiler to determine syntactical correctness?

Perhaps what drove programmers to such relative laziness was the fact that the type of problems software was tackling had also changed. No longer did programmers just code mathematical algorithms. They were building systems to let companies do business faster and more efficiently. They were building software that had never before been possible.

It was in this era that programmers began passing bugs off as features. Naïve, 1970s-era users willingly submitted to complicated workarounds if they believed it was “the only way to implement a feature.” Programmers went so far as to pass off bugs as configuration or operating-environment problems caused by users. Users readily shouldered the blame because they understood so little about what was actually happening under the covers.

Testing was another casualty of the chaos decade. In the 1960s, competent developers had performed all review and testing functions. But the 1970s-era rush to devise automated solutions to new problems—and new features for existing systems—created a huge demand for programmers. So everyone with software training flocked to programming, and testing was overlooked in the flurry.

Granted, software development organizations did not shun testing altogether, but many organizations turned that task over to unskilled personnel, essentially transforming administrative staff into testers. Testers today still endure the stigma associated with software testing that arose in those early years.

Code written in the 1970s is the bane of modern programming. It even has a special name: legacy code. Legacy code is feared, poorly understood, and worried over; most software professionals try to avoid making its maintenance part of their careers. After all, someone else’s code can be hard to understand, and one mistake made modifying code can cause undesirable side effects, no matter how much testing takes place.

Finally, the other significant arrival on the 1970s scene was metrics—those numbers that supposedly tell a story about code’s “goodness” but whose interpretation is often overly subjective. The chaos decade got metrics off to a bad start. The theory that evolved centered on quantifying aspects of the source code—the number of loops, branches, conditional statements, and so forth. Instead of trying to determine whether the software was functionally correct, developers could simply count particular elements in code to determine its complexity.

This was a tempting diversion in the 1970s, and perhaps it gave many developers a sense of satisfaction about their code. However, to this day the use of metrics remains the exception to the rule. Most developers ignore metrics, because they realize that good programmers can create very good code that rates poorly according to some metric, and poor programmers can write bad code that looks good according to a metric. So, unfortunately, the repute of metrics has suffered because of its initial misrepresentation of reality. Good, modern functional-correctness metrics still suffer from their association with code complexity metrics that originated in the 1970s.

The bottom line for chaos-decade software is that the focus was code-centric and not quality-centric. By the end of the 1970s, it became apparent that changes in the industry were necessary. And the first book on software testing (Glenford Myers, The Art of Software Testing, Wiley, 1979) arrived at this decade’s end. It was a clear signal that change was in the air.

1980 TO 1989: REPAIR

During the 1980s, several efforts arose to repair the common wisdom of software development. Two stand out as being particularly noteworthy.

CASE tools

The first came in the form of computer-aided software engineering, known as CASE. In more general terms, we could call the CASE repair effort “the advent of code development tools.” The CASE idea was that programmers would create better software if they had software tools assisting them. (Every craftsperson needs the right tools. Carpenters need hammers, for example. Try beating a nail into a board with a shoe.)

Like carpenters, software developers have their own set of basic trusted tools: the editor, compiler, and debugger. The CASE movement brought them access to more advanced tools, such as fourth-generation programming languages (which have their own acronym, 4GL). Unfortunately, 4GLs and most other high-end tools

In the 1970s, many organizations turned testing over to unskilled personnel, essentially transforming administrative staff into testers.
haven’t lived up to expectations. Take Visual Basic as an example: By all accounts, this is a useful, powerful, and popular 4GL tool, making programmers more productive and less error prone. However, the best-paying jobs are for C developers, and the majority of large applications are still primarily coded in C (Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978). Admittedly, 4GL and CASE tools have their place, but general-purpose tools such as editors, compilers, and debuggers are the staples of software developers today, despite the CASE hype of the early 1980s.

Another reason that tools have failed to capture large-scale attention is that their very nature as quality enhancers works against them. If a tool promises dramatic quality gains, shouldn’t the tool itself be of high quality? Have the tool’s developers applied the tool to itself? Buggy tools sold to enhance quality rarely endear themselves to software developers. Lastly, the quality of the person using the tool is also important. The saying “a fool with a tool is still a fool” still holds true and represents a situation that will certainly not engender higher-quality software.

### Formal methods

The second major solution proposed in the 1980s for producing higher-quality code was the use of formal methods. As with CASE, many touted formal methods as a software engineering silver bullet (Richard Linger, “Cleanroom Process Model,” *IEEE Software*, vol. 11, no. 2, Mar. 1994, pp. 50–58). And, just as with CASE, the bullet turned out to contain a bit of silver but remained mostly lead.

What was silver in formal methods consisted of techniques such as information hiding, structured programming, and stepwise refinement, as the “Silver Linings in Formal Methods” sidebar explains. That these techniques, which originated or became widespread in the 1980s, are still common practice for modern programmers provides evidence of their success. Structured programming and object orientation (which is firmly rooted in the principles of information hiding) are undeniably useful for producing high-quality code; they are now so widely used that they are the rule rather than the exception for modern development.

However, rigorous formal methods never caught on in mainstream software development organizations. Despite a smattering of organizations that claim to do Cleanroom (a variant of rigorous formal methods; see David P. Kelly and Robert S. Oshana, “Integrating Cleanroom Software Engineering Methods into an SEI Level 4–5 Program,” *Crosstalk*, Nov. 1996), the overwhelming majority of modern software development is still ad hoc. The reasons are cost and return on investment. Formal methods are difficult to use, time-consuming, and often nearly require the person applying them to have a PhD in computer science for proper deployment. What’s more, as in the 1980s, there’s still a serious lack of tools to assist developers in using formal methods.

Nevertheless, the repair decade produced valuable ideas for increasing developer productivity. In addition, the researchers who worked tirelessly to develop formal methods gave the scientific community techniques that more
systematically guide development practices. The 1980s ended with the universal recognition of the importance of software practices—and with a change in attitude toward the requirement to attain higher levels of quality.

1990 TO 1999: PROCESS

The next major “solution” to the software quality problem came in the 1990s under the phrase software process improvement. At the center of this movement was the much heralded, often derided, Capability Maturity Model or CMM; see “The Capability Maturity Model” sidebar for a short explanation. For brevity’s sake, we’ll oversimplify the software process improvement dogma: Software development is a management problem to which you can apply proper procedures for managing data, processes, and practices to good end. Controlling the way software is produced ensures better software.

In other words, because developers had failed to manage their projects appropriately (as evidenced historically by software’s poor track record for quality), managers must install organizational controls to manage for them. The problem with this belief is many-fold, because even the best processes in the world can be misapplied (Jeffrey Voas, “Can Clean Pipes Produce Dirty Water?” IEEE Software, vol. 14, no. 4, July 1997, pp. 93-95).

Although we’re being facetious, our point is serious: Despite the fact that good software development processes are usually necessary, the software process improvement movement sold its processes to developers in a way that established an adversarial relationship between management and technical personnel. To make matters worse, many managers who knew nothing about software suddenly found their skills in high demand in software companies keen on process improvement.

However, software development is fundamentally a technical task: Good developers can develop good software despite poor or no management. However, the converse is improbable: Poor technicians are unlikely to develop good software under even the best management. (For an alternative analysis but similar conclusion, mostly concerning management’s role in Y2K mitigation, see Robert Glass, “Y2K and Other Software Noncrises,” IEEE Software, vol. 17, no. 2, Mar. 2002, pp. 104-100.) Thus, the CMM has propagated slowly. In many large software companies, developers are still unaware of its very existence.

The CMM is not the only software process improvement idea that came out of the 1990s. In the decade’s later years, software development organizations began to apply a related theory to their processes—Six Sigma, a method originally devised for reducing manufacturing and design defects in hardware systems.

Six Sigma is a disciplined, data-driven approach and methodology for eliminating defects (driving towards six sigmas between lower and upper specification limits) in any process—from manufacturing to transactional and from product to service. To achieve Six Sigma, a process must not produce more than 3.4 defects per million opportunities. A Six Sigma defect is defined as anything outside of customer specifications. A Six Sigma opportunity is then the total quantity of chances for a defect (http://www.isixsigma.com/sixsigma/six_sigma.asp).
The problem with Six Sigma, however, is that it is not clear what one million opportunities to introduce defects into a software product means. Furthermore, how could that ever be properly measured?

To further widen the chasm dividing management and technical staff over how to develop software, the 1990s was also a decade of remarkable progress in computing infrastructure. New operating platforms eclipsed older operating systems in sophistication. Knowledge that once was useful became obsolete. New programming languages popped up and became overnight successes. Programming had to be learned and relearned. New APIs (application programming interfaces) for communication, security, distributed computing, and, of course, the Web turned developers’ lives upside down. Because developers were constantly addressing the crisis of staying current, they had little time to attend to the pressures of following particular software process standards.

In defense of the software process movement, we must recognize it as a new phenomenon. Like many new phenomena, it is not completely understood and is widely misapplied. To our minds, one lesson of the 1990s is that the current state of the practice in software process does not easily support new technologies. What worked for mainframe or desktop applications does not necessarily work on products that are built quickly and deployed hourly in today’s Internet-time workplace.

However, like its partially successful predecessors, the emphasis on software process produced some beneficial side effects. The fact that many more developers are aware of simple things like configuration management, defect tracking, and peer review is clearly positive. The 1990s began as a process revolution and ended with the realization that process is not something that you can force on people or that will catch on in a few years. Furthermore, process for the sake of process is not enough. Process improvement comes from better technical practices, plain and simple.

Finally, the 1990s marked the first real attempt to turn software development into engineering through the concepts of component-based software engineering (CBSE) and commercial off-the-shelf (COTS) components. The idea is to create small, high-quality parts and join them together. The problem, of course, is that high-quality parts joined together do not necessarily result in a high-quality composite system. The composite system might suffer from a flawed method of composition, or assumptions about the components’ behavior or environment might be flawed. Furthermore, commercial software components, which companies usually license as executables, can yield nasty
side effects unknown to the licensee. Such side effects might only manifest themselves when joined to other components and are virtually impossible to detect by testing the component in isolation. Therefore, although the divide-and-conquer paradigm works well for hardware and physical systems, it can actually be a disaster for logical systems. Only time will tell how CBSE will affect software quality’s future.

2000 TO 2009: ENGINEERING?

In the early years of yet another decade, we wonder what the future holds. Will this be the decade in which we solve the software quality problem? Will this be the decade in which developers and users view software failure with surprise and wonder? Or will we end this decade with the same outlook we had in 2000: All software fails, and everyone must accept it (Charles C. Mann, “Why Software Is So Bad, and What Is Being Done to Fix It?” MIT Technology Rev., 17 June 2002).

According to Les Hatton (“Does OO Sync With How We Think?” IEEE Software, vol. 15, no. 3, May 1998, pp. 46-54), “The industry standard for good commercial software is around six defects per KLOC [thousand lines of code] in an overall range of around six to 30 defects per KLOC.” Thus, the defect rate has held fairly constant for the last two decades, regardless of the shift to object-oriented technology, automated debuggers, better test tools, and stronger type safety in languages such as Java and Ada. Is there any reason to believe that this will change in this decade? Although the technical challenges are staggering, there’s motivation in the fact that the cost of poor software quality will also climb. According to a report published in 2002 for the National Institute of Standards and Technology, Estimates of the economic costs of faulty software in the US range in the tens of billions of dollars per year and have been estimated to represent approximately just under 1 percent of the nation’s gross domestic product (Research Triangle Institute, “The Economic Impacts of Inadequate Infrastructure for Software Testing,” NIST Planning Report 02-3, May 2002).

We are already seeing a backlash against many of the mainstream waterfall and iterative software development methods in favor of agile and Extreme Programming methods. If taken “to the extreme,” agile development is a completely unstructured, chaotic process that employs unrepeatable processes and bypasses much of the testing and design phases. Although agile development might decrease time-to-market delays and increase the rate at which programmers can write code, whether such an approach improves quality is uncertain at best.

The question of what this decade will offer sets the “crisis people” apart from those of us who believe that human ingenuity and engineering know-how will defeat the quality problem for software. After all, accountants have figured out quality, airplane manufacturers have figured it out, appliance makers have figured it out, plumbers have figured it out, and electricians have figured it out.
Software developers are at least as talented as those who work in these other professions, so we believe that higher quality software is in our future. We, as a community, can figure it out. In fact, it appears that even Bill Gates has now recognized the need to “crack the software quality nut,” according to the e-mail message that he is rumored to have sent out to all employees on 15 January 2002:

Every few years I have sent out a memo talking about the highest priority for Microsoft. Two years ago, it was the kickoff of our .NET strategy. Before that, it was several memos about the importance of the Internet to our future and the ways we could make the Internet truly useful for people. Over the last year it has become clear that ensuring .NET is a platform for Trustworthy Computing is more important than any other part of our work. If we don’t do this, people simply won’t be willing—or able—to take advantage of all the other great work we do. Trustworthy Computing is the highest priority for all the work we are doing. We must lead the industry to a whole new level of Trustworthiness in computing.

Still, though, the question is when. When will we achieve the ability to create reliably high-quality software? The answer depends heavily on whether and how fast we work from certain ideas originating in the past decades—the ideas surveyed in this article. Each decade provided valuable insights, and, although no decade produced a silver bullet, each provided an additional piece to the software quality puzzle.

Our community’s main problem has been that it has summarily dismissed many useful ideas only because no single one was a panacea. For decades, the mindset has been that even if a technique enhanced the possibility of better software, if it didn’t guarantee perfect software, it had no value. Clearly this is wrong. Until we work harder as a community of professionals to combine past proven techniques into new quality-enhancing methodologies, gearing them toward the problems we’re trying to solve with software today, we will continue to wait.

James A. Whittaker is a professor in the Department of Computer Sciences, Florida Institute of Technology. Contact him at jw@cs.fit.edu.

Jeffrey M. Voas is cofounder and chief scientist at Cigital in Dulles, Virginia. Contact him at voas@cigital.com.

For further information on this or any other computing topic, visit our Digital Library at http://computer.org/publications/dlib.