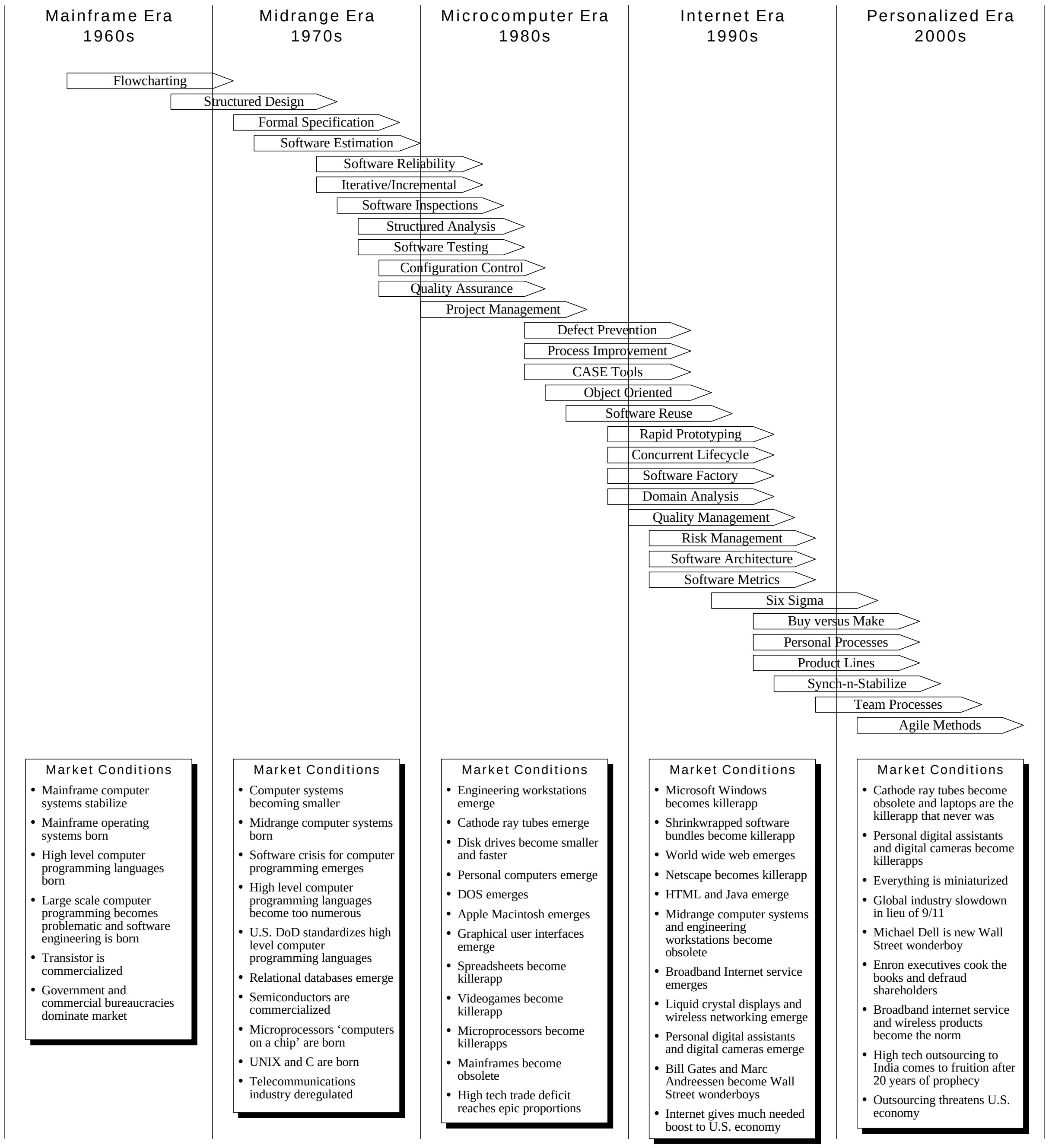
SHORT HISTORY OF SOFTWARE METHODS

by David F. Rico

This short history identifies 32 major classes of software methods that have emerged over the last 50 years. There are many variations of each major class of software method, which renders the number of software methods in the hundreds. This short history contains a brief synopsis of each of the 32 major classes of software methods, identifying the decade and year they appeared, their purpose, their major tenets, their strengths, and their weaknesses. The year each software method appeared corresponds to the seminal work that introduced the method based on extensive bibliographic research and the strengths and weaknesses were based on scholarly and empirical works to provide an objective capstone for each method.



Market Conditions (1960s)
- Mainframe computer systems stabilize
- Mainframe operating systems born
- High level computer programming languages born
- Large scale computer programming becomes problematic and software engineering is born
- Transistor is commercialized
- Government and commercial bureaucracies dominate market

Market Conditions (1970s)
- Computer systems becoming smaller
- Midrange computer systems born
- Software crisis for computer programming emerges
- High level computer programming languages become too numerous
- U.S. DoD standardizes high level computer programming languages
- Relational databases emerge
- Semiconductors are commercialized
- Microprocessors 'computers on a chip' are born
- UNIX and C are born
- Telecommunications industry deregulated

Market Conditions (1980s)
- Engineering workstations emerge
- Cathode ray tubes emerge
- Disk drives become smaller and faster
- Personal computers emerge
- DOS emerges
- Apple Macintosh emerges
- Graphical user interfaces emerge
- Spreadsheets become killerapp
- Videogames become killerapp
- Microprocessors become killerapps
- Mainframes become obsolete
- High tech trade deficit reaches epic proportions

Market Conditions (1990s)
- Microsoft Windows becomes killerapp
- Shrinkwrapped software bundles become killerapp
- World wide web emerges
- Netscape becomes killerapp
- HTML and Java emerge
- Midrange computer systems and engineering workstations become obsolete
- Broadband Internet service emerges
- Liquid crystal displays and wireless networking emerge
- Personal digital assistants and digital cameras emerge
- Bill Gates and Marc Andreessen become Wall Street wonderboys
- Internet gives much needed boost to U.S. economy

Market Conditions (2000s)
- Cathode ray tubes become obsolete and laptops are the killerapp that never was
- Personal digital assistants and digital cameras become killerapps
- Everything is miniaturized
- Global industry slowdown in lieu of 9/11
- Michael Dell is new Wall Street wonderboy
- Enron executives cook the books and defraud shareholders
- Broadband internet service and wireless products become the norm
- High tech outsourcing to India comes to fruition after 20 years of prophecy
- Outsourcing threatens U.S. economy

Figure 1. Timeline of Market Conditions and Software Methods

Table 1

Summary of relationships between market conditions, software methods, and business objectives

Era	Market Conditions	Problem	Method	Goal	Savings
Mainframe	▪ Bureaucracies use computers ▪ Early programs in binary	Unreadable programs	Flowcharting	Document programs	Maintenance
	▪ Programmers are Math PhDs	Complex programs	Structured Design	Program using subroutines	Development
Midrange	▪ Telecoms, engineering firms, and academia use computers	Error prone programs	Formal Specification	Verify programs early	Maintenance
	▪ Number of projects explode	Costly programs	Software Estimation	Control costs	Development
	▪ Productivity, cost, and quality are pervasive issues	Failure prone programs	Software Reliability	Test important programs	Maintenance
	▪ Large projects are broken into smaller pieces	Long schedules	Iterative/Incremental	Program using subprojects	Development
	▪ Most computer programs inoperable	Poor code reviews	Software Inspections	Remove most defects	Maintenance
	▪ Software maintenance costs are astronomical	Unwritten requirements	Structured Analysis	Document requirements	Development
	▪ Programmers are engineers and physicists	Inoperable programs	Software Testing	Test all programs	Maintenance
		Lost programs	Configuration Control	Inventory programs	Development
		Unmet requirements	Quality Assurance	Evaluate requirements	Maintenance
Microcomputer	▪ Medium and small-sized businesses use computers	Overrun schedules	Project Management	Control schedules	Development
	▪ Fast computers and graphical user interfaces ideal for automated software tools to document customer requirements and designs	Expensive testing	Defect prevention	Reduce appraisal cost	Maintenance
		Unpredictable projects	Process Improvement	Standardize management	Development
		Manual documentation	CASE Tools	Automate documentation	Development
	▪ Automated scheduling tools abound but discipline of project management doesn't take hold	Numerous subroutines	Object Oriented	Group subroutines	Development
		Redundant subroutines	Software Reuse	Standardize subroutines	Development
	▪ Firms try and solve productivity and quality problems with reusable computer programs	Undefined requirements	Rapid Prototyping	Solicit early feedback	Development
		Long phases or stages	Concurrent Lifecycle	Use overlapping stages	Development
	▪ Programmers are computer scientists	Manual lifecycles	Software Factory	Automate lifecycles	Development
		Slow learning curves	Domain Analysis	Use program specialists	Development
Internet	▪ Computers are fast, cheap, easy-to-use and reliable, and one or more computers per household are the norm	Late verification	Quality Management	Verify requirements early	Maintenance
		Numerous problems	Risk Management	Reduce problems early	Development
	▪ Internet technologies like Netscape and HTML obsolete all prior computer languages	Redundant designs	Software Architecture	Standardize designs	Development
		Inconsistent measures	Software Metrics	Standardize measures	Maintenance
	▪ Productivity, quality, and automation take a back seat	Inconsistent quality	Six Sigma	Achieve high quality	Maintenance
	▪ Project management attempts to make a last stand, though it's rejected as too inflexible	Redundant applications	Buy versus Make	Buy commercial programs	Development
	▪ Most software can be purchased now and doesn't have to be built anew	Poor management	Personal Processes	Teach management skills	Development
		Redundant products	Product Lines	Standardize products	Development
	▪ Millions of websites are created by anyone with a computer and some curiosity	Late programming	Synch-n-Stabilize	Begin programming early	Development
		Poor team management	Team Processes	Teach team management	Development
Personalized	▪ Computers small and wireless	Inflexible programs	Agile Methods	Create flexible programs	Development

MAINFRAME ERA

The 1960s were a defining period for the world of computers giving rise to what we now know as mainframe computers (Solomon, 1966). Think of mainframe computers as building-sized calculators, most of which can now fit in your shirt pocket and are versatile enough to run on sunlight. Of course, these mainframe computers gave rise to large scale operating systems requiring hundreds of expert programmers to produce over many years (Needham and Hartley, 1969). More importantly, high-level computer programming languages such as the Common Business Oriented Language or COBOL were created to help humans communicate with these building-sized calculators and instruct them to perform useful functions more easily (Sammet, 1962). The creation of these mainframes, their operating systems, and their high-level COBOL computer programming languages caused the North Atlantic Treaty Organization or NATO to form a new technical discipline called software engineering to help manage the explosion of computer programming projects (Naur & Randell, 1969). But, this was only the tip of the iceberg, because transistors were being commercialized, which would soon give rise to an explosion of new computers, operating systems, programming languages, and software projects that NATO could never have anticipated (Smith, 1965). However, suffice it to say that mainframes met the needs of the government and corporate bureaucracies that dominated the market and era of the 1960s rather well (Niskanen, 1968).

Flowcharting. The first major software method to emerge during the mainframe era of the 1960s was flowcharting (Knuth, 1963). The purpose of flowcharting was to document computer programs written in primitive computer programming languages, which used complex symbolic codes or pseudonyms, so that other computer programmers could understand their design, maintain them (e.g., correct, enhance, or upgrade), and prepare computer program user guides for customers (Knuth). Flowcharting was a seven step process of dividing the computer program into logical sections, describing the overall computer program at the beginning, documenting the functions and assumptions, delineating each section with a special code, delineating each instruction with a special code, placing special markings next to each instruction, inserting a special symbol to show the relationships between instructions, and automatically generating a flowchart from the results (Knuth). From these codes within codes, diagrams were automatically generated to help computer programmers visualize the primitive computer languages using flowcharts (Knuth). The strength of flowcharting was that it provided a two dimensional graphical representation of the flow and control of computer programs that was easily understood, enabled other computer programmers to easily understand it, and it provided customers with an explanation of computer programs (Knuth). The weakness of flowcharting was that it generated numerous complex graphical symbols and arrows because computer programs were organized into logical abstractions called subroutines (Dijkstra, 1968). The use of flowcharting has never proven to have any appreciable quantitative benefits (Shneiderman, Mayer, McKay, & Heller, 1977), yet flowcharting remains one of the most ubiquitous tools for business process reengineering and quality improvement (Stinson, 1996; Wingo, 1998).

Structured design. The second major software method to emerge during the mainframe era of the 1960s was structured design (Dijkstra, 1968). The purpose of structured design was to organize computer programs into a functionally decomposed hierarchy of larger algorithmic abstractions called subroutines versus individual computer statements, much like an organizational chart of a large and complex bureaucracy (Dijkstra). Structured design was a top down approach of identifying the main purpose of the computer program, dividing the top level function into smaller parts, dividing the smaller parts into even smaller parts, and then iterating over this process until the computer program was a functionally decomposed hierarchy of subroutines rather than a conglomeration of primitive computer instructions (Dijkstra). The strength of structured design was that computer programs organized into a hierarchy of subroutines were easier to design, implement, test, and maintain (Wirth, 1971; Stevens, Myers, & Constantine, 1974). The weakness of structured design was that no one could agree on the optimal size, number, and complexity of the hierarchy of subroutines leading to wide variations in structured design practices (Endres, 1975; Basili & Perricone, 1984).

MIDRANGE ERA

If the 1960s were a defining era for the world of computers, the 1970s completely redefined the face of technology resulting in the creation of what we now know as midrange or minicomputer systems (Eckhouse, 1975). Midrange systems were simply smaller, better, faster, and cheaper computers created by the likes of Gene Amdahl, an IBM engineer, whose back-of-the-napkin innovations fell on deaf ears at Big Blue, but was more than welcome news to the Japanese (Amdahl, 1967). Amdahl simply told the Japanese that he could build a better mousetrap, if they'd just foot-the-bill, and the likes of NASA would roll out the red carpet for his new machines, which they did in order to fuel the information processing needs of the both the Apollo and Skylab programs (Yamamoto, 1992). But, trouble was on the horizon, the innovations in computers, operating systems, programming languages, and software methods were resulting in too much software, too fast, for the U.S. government, its military, and its corporations to manage successfully, causing the Dutch inventor of the structured design methodology to describe this period as the software crisis (Dijkstra, 1972). The software crisis transcended the invention of the microcomputer and only recently began subsiding with the advent of the Internet age, more than 25 years later (Humphrey, 1989). To add insult to injury, over 170 high-level computer languages emerged by the early stages of the midrange era (Sammet, 1972). The U.S. DoD, for whom computers had been created, demanded an end to this Tower of Babel and set out to create a single, standard high-level computer programming language that embodied the best features, ideas, and innovations in computer programming (Carlson, Druffel, Fisher, and Whitaker, 1980). A slew of advanced information technologies exploded onto the global scene, which would not only fulfill the promise of computers for technologists, but for the U.S. military as well as the worldwide commercial marketplace. These technologies included the relational database for creating large repositories of reliable information that were portable across a wide variety of computers and were cost effective to maintain (Codd, 1970). Semiconductors were also commercialized, opening the door for smaller and faster midrange systems, more sophisticated operating systems, sleek and efficient computer programming languages, and more importantly, cost effective computers that would serve as a catalyst for enhancing organizational performance and profitability (Monk, 1980). The microprocessor also emerged in the midrange era, which was simply an entire computer on a chip (Brennan, 1975). Whereas, a mainframe was the size of a building and early midrange computers filled up entire rooms, microprocessors could fit in a lunch box (Brennan). The UNIX operating system was created, which was the smallest, most efficient, and most useful computer operating system ever created (Ritchie & Thompson, 1974), and by only two people using their own private computer programming language called C (Kernighan & Ritchie, 1978), whereas early mainframe operating systems required thousands of people to create over periods as long as a decade (Brooks, 1975).

The midrange era saw the deregulation of the telecommunications industry and breakup of Ma Bell into many baby bells, which consumed the plethora of midrange systems, commercial semiconductors, microprocessors, UNIX operating systems, and the C programming language to create switching systems for handling consumer long-distance calling, consisting of tens of millions of lines of code (Stanley, 1973; Kernighan & Ritchie; Ritchie & Thompson).

Formal specification. The first major software method to emerge during the midrange era of the 1970s was formal specification (Hoare, 1971). The purpose of formal specification was to provide a method for designing computer programs that were free of errors and defects before they were created as an alternative to flowcharting and structured design (Hoare). Formal specification was a four stage process of applying the language and semantics of logic and mathematics to defining requirements or problem statements, translating the problem into a mathematical or logical formula, simplifying the formula, translating the formula into a computer program, and then refining the computer program further (Knuth). The strength of formal specification was that it reduced time to market and productivity by four times, it increased computer program quality by 100 times, and it had a return on investment of 33 to 1 (McGibbon, 1996). The weakness of formal specification was that it didn't have a mechanism for capturing customer needs, the use of logic and mathematics was difficult to link to customer requirements, and the use of logic and mathematics was ambiguous and subject to wide interpretation (Liu & Adams, 1995).

Software estimation. The second major software method to emerge during the midrange era was software estimation (Merwin, 1972). The purpose of software estimation was to establish an empirical framework for project management of computer programming based on effort, costs, productivity, quality, schedules and risks (Walston & Felix, 1977). Software estimation was a seven step process of defining the customer requirements, producing a conceptual design, estimating the computer program size from historical data, estimating the effort and costs from historical data, producing a schedule, developing the product, and analyzing the results (Flaherty, 1985). The strength of early software estimation methods was that they established the basis for empirical project and quality management of computer programming for several decades (McGibbon, 1997). However, in spite of their pervasive use well into the 21st century, the reliability and validity of software estimation has proven to be a continuing issue hindering their wide spread usage and acceptance (Auer & Biffl, 2004).

Software reliability. The third major software method to emerge from the midrange era was software reliability (Littlewood, 1975). The purpose of software reliability was to improve customer satisfaction by reducing the number of times a system crashed by operating computer programs the way customers would, removing the failures, and measuring the mean time between failures with various mathematical or statistical models (Littlewood). Software reliability was a five step process of defining the customer's reliability needs, applying a software development method to meet those needs, integrating in commercial or reusable software that met the reliability requirements, operationally testing computer programs by using them the way customers would, and verifying that computer programs satisfied mean time to failure requirements through the application of specialized statistical or mathematical models (Everett, 1995). The strengths of software reliability were that it was an inexpensive way of maximizing customer satisfaction and statistical reliability models had proven to be accurate (Kan, 1991, 1995). The weaknesses of software reliability were that some reliability models were not focused on testing that models the customer's behavior, reliability models varied widely in their designs, the underlying metrics and measures of reliability models varied widely, software failures were inconsistently classified, few practitioners applied more than one type of reliability model, computer programs were rarely classified according to type and kind, reliability models were difficult to calibrate and verify, and there was evidence of bias among reliability studies (Lanubile, 1996).

Iterative and incremental. The fourth major software development method to emerge during the midrange era was iterative and incremental development (Basili & Turner, 1975). The purpose of iterative and incremental development was to maximize customer satisfaction by designing computer programs one component at a time and soliciting early market feedback (Basili & Turner). Iterative and incremental development was a six step process of identifying as many customer or market needs as possible, defining a product specification consisting of prioritized customer requirements, planning a series of autonomous subprojects to implement the customer requirements from highest to lowest priority, rapidly designing computer programs to meet individual requirements, soliciting early market feedback by releasing rapidly produced computer programs, and repeating the entire process by leveraging the early market feedback, benefiting from the lessons learned in doing so and gaining momentum from the experiences obtained through this iterative process (Basili & Turner). The strengths of iterative and incremental development were that it could result in 50% increases in productivity, a time to market improvement of 66%, and an overall computer programming cost reduction of 20% (Woodward, 1999). The weakness of the iterative and incremental method was lack of sound project management and quality assurance practices (Humphrey, 1996).

Software inspections. The fifth major software development method to emerge during the midrange era was software inspections (Fagan, 1976). The purpose of software inspections was to maximize customer satisfaction, software quality, and software reliability of computer programs by using small teams to identify and remove software defects from computer programs using group reviews (Fagan). Software inspections were a six step process of planning the number of inspections based on the number of computer programs to be inspected, holding a preliminary overview meeting to orient programmers on the computer programs they were about to review, individual self study and review of computer programs and their associated documentation, the team review itself to analyze computer programs one instruction at a time to identify software defects, a rework stage for the computer programmer to repair defects, and a follow-up stage to verify the defects were repaired and to report the status of the computer program quality (Fagan). The strengths of software inspections were that they were 10 times more efficient at eliminating defects than software testing, minimized software maintenance costs, resulted in high levels of computer program reliability, and had a return on investment of up to 30 times over software testing alone (Fagan, 1986). The weaknesses of software inspections were inadequate training, poor facilitation, poor execution, organizational resistance, lack of management support, lack of focus, bloated checklists, manual data collection, pursuit of trivial issues, inexperience, inadequate preparation, and lack of measurement (Lee, 1997).

Structured analysis. The sixth major software development method to emerge during the midrange era was structured analysis (Ross & Schoman, 1977). The purpose of structured analysis was to improve the success of software projects and computer program design by documenting customer requirements in a functionally decomposed graphical hierarchy (Ross & Schoman). Structured analysis was a seven step process of defining the top level function of the system with its inputs and outputs, decomposing the top level function into subfunctions with their inputs and outputs, continuing this process until all of the customer requirements had been represented on the hierarchy chart, internally verifying the functional decomposition, soliciting customer feedback about the functional decomposition, repairing and refining the functional decomposition based on the customer feedback, and then developing one or more computer subroutines corresponding to each requirement on the functional decomposition (Ross & Schoman). The strength of structured analysis was that it resulted in substantial improvements in productivity, estimation accuracy, computer program quality, requirements quality, and satisfaction of customer requirements at the system and software level (Hardy, Thompson, & Edwards, 1994).

Software testing. The seventh major software development method to emerge during the midrange era was software testing (Hennell, Hedley, & Woodward, 1977). The purpose of software testing was to ensure a computer program satisfied its customer needs and requirements, performed its intended functions, and operated properly and failure free. Software testing was an eight step process of defining a test strategy based on customer requirements, a project plan for conducting testing, an overall design for the tests, individual test cases to verify a range of requirements, test procedures corresponding to a particular set of conditions, a test item transmittal for communications, a test log for recording testing actions, and a test summary to report the findings of the testing process (Institute of Electrical and Electronics Engineers, 1993). The strengths of software testing was that it improved time to market, productivity, quality, and return on investment by more than 10 times (Asada & Yan, 1998). The weakness of testing was that it was 10 times less efficient than other methods (Russell, 1991).

Configuration control. The eighth major software development method to emerge during the midrange era was configuration control (Bersoff, Henderson, & Siegel, 1978). The purpose of configuration control was to reduce the cost and risk of developing computer programs by maintaining an inventory of computer programs and their associated documents (e.g., requirements, designs, subroutines, and tests) as they were being developed (Bersoff, Henderson, & Siegel). Configuration control was an eight step process of identifying configuration items (e.g., computer programs and their documentation), naming configuration items, gathering configuration items as they were developed, requesting changes to configuration items, evaluating those changes, approving or disapproving changes, implementing the changes, producing status reports about the state of configuration items, and auditing the configuration items against inventory lists (Institute of Electrical and Electronics Engineers, 1990). The strengths of configuration control were that it enhanced communication among computer programmers, it established a clear inventory of all computer programs and associated documentation, and it acted as a solid foundation to enhance decision-making for project managers and computer programmers (Cugola et al., 1997). The weakness of configuration control was that it cost more than it saved computer programmers (Cugola et al.).

Quality assurance. The ninth major software development method to emerge during the midrange era was quality assurance (Benson & Saib, 1978). The purpose of quality assurance was to ensure that a computer program satisfied its customer needs and requirements by ensuring that management and technical activities were properly used (Benson & Saib). Quality assurance was an 11 step process of ensuring minimum documentation was produced; standards, practices, conventions, and metrics were utilized; reviews and audits were conducted; software testing was performed; problem reporting and corrective action took place; software tools, techniques, and methodologies were applied; computer programs were controlled; media were tracked; suppliers were managed; records were kept; training was conducted; and risk management was performed (Institute of Electrical and Electronics Engineers, 1989). The strengths of quality assurance were that it ensured customer needs and requirements were satisfied early in the lifecycle of computer programs when it was less expensive to do so and it enhanced quality and reliability of computer programs (Rubey, Browning, & Roberts, 1989). The weaknesses of quality assurance were that it was often confused with many other software methods such as software testing and configuration control and was is often omitted for this reason (Runeson & Isacsson, 1998).

MICROCOMPUTER ERA

The microcomputer era of the 1980s ushered in revolutionary changes in information technology, and, as if all at once, threatened to make the mainframes of the 1960s and midrange systems of the 1970s suddenly obsolete, in favor of much faster, cheaper, and more reliable workstations (Bewley, Roberts, Schroit, & Verplank, 1983) and personal computers (Ahlers, 1981). Cathode ray tubes or inexpensive television screens combined with keyboards also seemed to magically appear, replacing the punch cards that were developed in the 1960s and 1970s to program computers (Randles, 1983). Computer disk drives, which were one of the most significant innovations to emerge from the midrange era, became faster and cheaper during the microcomputer era (Murray, 1989). While UNIX was the star of the 1970s (Ritchie & Thompson, 1974), the disk operating system or DOS for personal computers became the killerapp of the 1980s, which was a computer program not unlike its big mainframe or midrange operating system cousins (Pechura, 1983). Apple's Macintosh, a premium-priced personal computer, appeared as an oddity replete with one of the first graphical user interfaces (Broadhead, Gregory, Pelkie, Main, & Close, 1985), which due to its price was only a promise of things to come for the average user (Ives, 1982). Spreadsheets emerged as a killerapps because they made easy the task of budgeting, financial analysis, and could be used to design simple economic what-if scenarios for strategic planning (Carlsson, 1988). The microprocessor that emerged in the 1970s was now getting faster and cheaper in two-year intervals and itself was considered a killerapp (Lockwood, 1988). Microcomputers and their powerful computer programs were functionally indistinguishable from mainframes (Collins & Stem, 1986; Harrison, 1985). The loss of the mainframe revenues for the U.S. computer industry exacerbated the broadening technologically-based trade deficit between East and West and quickly became a national crisis (Tyson, 1992).

Project management. The first major software development method to emerge during the microcomputer era was project management (Mills, 1980). The purpose of project management was to estimate the costs, develop a schedule, and manage the costs and schedule for developing complex computer programs (Mills). One of the goals of project management was to oversee the implementation of a consistent set of practices or lifecycle across all computer programming projects that were known to reduce costs and improve quality (O'Neill, 1980). Standardized design and programming practices were considered a key part of the project management life cycle (Linger, 1980). The project management lifecycle also consisted of standardized configuration control and automated software testing tools (Dyer, 1980). Early project management also consisted of a standard suite of lifecycle documents, such as project planning, requirements, design, and testing documentation (Quinnan, 1980). The strength of project management and its standard lifecycle, activities, and documents was that it acted as a basis for bottom up estimation of project costs and provided a framework for cost control (Mills).

However, this early project management model failed to incorporate best practices in iterative and incremental development, which were instrumental project management practices for dramatically reducing costs and risks (Basili & Turner, 1975).

Defect prevention. The second major software development method to emerge during the microcomputer era was defect prevention (Jones, 1985). The purpose of defect prevention was to improve customer satisfaction of computer programs along with quality and reliability by preventing software defects more cost effectively than software inspections and testing (Jones). Defect prevention was a six step process of holding kickoff meetings to teach computer programmers about the common causes of errors and how to avoid them, holding causal analysis meetings after computer programs were complete to analyze the software errors and determine what to do about them, populating an action database with all organizational action plans to prevent common causes of error in the future, holding action team meetings to begin implementing organizational controls to prevent errors, and populating a defect repository with defect prevention results to feed back into the kickoff meetings (Jones). The strengths of defect prevention were that it was hundreds of times more effective at achieving customer satisfaction, software reliability, and software quality than software inspections and testing alone (Mays, Jones, Holloway, & Studinski, 1990). The weaknesses of defect prevention were that its process was subjective and occurred too late, and it was a manual process subject to intense organizational resistance (Chillarege et al., 1992).

Process improvement. The third major software development method to emerge during the microcomputer era was process improvement (Radice, Harding, Munnis, & Phillips, 1985). The purpose of process improvement was to standardize project management and quality assurance practices across an organization in order to enhance productivity and quality (Radice, Harding, Munnis, & Phillips). Process improvement was a four-step process of auditing an organization’s computer programming processes, identifying their strengths and weaknesses, standardizing the strongest ones, and improving the weakest practices (Radice, Harding, Munnis, & Phillips). The strength of this approach was that it enabled organizations to standardize only the strongest best practices that were tailored for each business unit, serving as a simultaneous bottom up and top down approach (Radice, Harding, Munnis, & Phillips). And, some organizations experienced ten-fold increases in productivity and quality using process improvement (Diaz & Sligo, 1997). However, more organizations failed to show any improvements at all and frequently deceived the auditors in order to get favorable evaluations (O'Connell & Saiedian, 2000).

CASE tools. The fourth major software development method to emerge during the microcomputer era was computer aided software engineering or CASE tools (Hoffnagle & Beregi, 1985). The purpose of CASE tools was to serve as an broad ranging, but integrated set of automated tools to support project management and development of computer programs (Hoffnagle & Beregi). The ultimate goal of using CASE tools was to automate manually intensive tasks, stimulate productivity, and ultimately yield market and economic advantages (Hoffnagle & Beregi). CASE tools automated important activities such as project management, documentation of customer requirements, design of computer programs, software testing, and configuration control (Mercurio, Meyers, Nisbet, and Radin, 1990). CASE tools cost millions of dollars to purchase and deploy (Huff, 1992) and failed to live up to their economic benefits (Guinan, Cooprider, and Sawyer, 1997).

Object oriented. The fifth major software development method to emerge during the microcomputer era was object oriented design (Booch, 1986). The purpose of object oriented design was to logically organize computer programs into larger abstractions called objects that mirrored the real world, which resulted in computer programming modules called radios, engines, missiles, or automobiles (Booch). Structured analysis and design organized computer programs into mathematical formulas, such as sine, cosine, or tangent, and resulted in thousands of subroutines that were difficult to design, program, and maintain (Booch). By grouping computer programs into real world abstractions, computer programs would contain fewer modules, and thus would be easier to design, program, and maintain (Booch). More importantly, object oriented programs would be faster to produce, easier to understand, and easier to adapt to rapidly changing customer needs (Booch). Combined, these benefits reduced the costs and improved the reliability of computer programs (Booch). The productivity-enhancing benefits of object oriented design never came to fruition (Reifer, Craver, Ellis, & Strickland, 2000).

Software reuse. The sixth major software development method to emerge during the microcomputer era was software reuse (Owen, Gagliano, and Honkanen, 1987). The purpose of software reuse was to create libraries of high quality object oriented computer programs that could be used many times, eliminate reinventing the wheel, and reduce development and maintenance costs (Owen, Gagliano, and Honkanen). Software reuse was a four-step process of creating an organizational environment for software reuse (e.g., organizational policies and procedures), creating the reusable computer programs themselves, managing an organizational library of these reusable compute programs, and building new systems from these computer programs (Lim, 1998). The strengths of software reuse were its 50% increases in quality, 60% improvements in productivity, and 40% reductions in time to market (Lim, 1994). However, software reuse proved to be difficult to implement due to intense cultural resistance and the effort of achieving its economic benefits was too great (Lim).

Rapid prototyping. The seventh major software development method to emerge during the microcomputer era was rapid prototyping (Luqi, 1989). The purpose of rapid prototyping was to solicit a complete set of customer requirements and optimize customer satisfaction by developing early prototypes of computer programs, which they could evaluate (Luqi). Rapid prototyping was a four-step process of soliciting preliminary customer requirements, creating an early model of computer programs, allowing customers to evaluate the models, refining the models until they met the customer's needs, documenting the customer requirements, and designing the final computer program based on the customer requirements (Luqi). The strengths of rapid prototyping were that customers were involved early, a complete set of customer requirements was obtained, and the final products met needs (Luqi). The weaknesses of rapid prototyping were that project management was ignored and poorly operating prototypes that didn't meet quality assurance standards were delivered to customers (Luqi).

Concurrent lifecycle. The eighth major software development method to emerge during the microcomputer era was concurrent lifecycle development (Sulack, Lindner, & Dietz, 1989). The purpose of the concurrent lifecycle was to produce large computer programs as quickly as possible in an era of highly competitive market conditions (Sulack, Lindner, & Dietz). Concurrent lifecycle development was a five part approach consisting of breaking large software projects into multiple iterations; overlapping activities such as design, programming, and test; soliciting early feedback from customers; standardizing software development activities across a large organization; and rigorously verifying the computer programs. The strengths of this approach were cycle time reductions of 60% (Sulack, Lindner, & Dietz) and early user involvement (Pine, 1989), which led to 100% customer satisfaction and quality levels (Kan, Dull, Amundson, Lindner, & Hedger, 1994). However, use of concurrent lifecycles simply could not result in a level of competitiveness necessary to keep pace with rapid innovations in personal computer and internet software (Liu, 1995; Bethony, 1996; Baker, 1994; Heck, 1995; Bouvier, 1995).

Software factory. The ninth major software development method to emerge during the microcomputer era was the software factory (Cusumano, 1989). The purpose of the software factory was to reduce the cost and improve the quality of producing computer programs in order to achieve massive economies of scale comparable to manufacturing (Cusumano). The software factory consisted of centralizing resources, establishing standards, automating configuration control and testing, applying quality assurance, and using computer aided software engineering or CASE tools and software reuse (Cusumano). The strength of the software factory was that it helped corporations improve sales by 10 times, reduce time to market by five times, improve software quality by five times, and achieve software reuse levels of more than 50% (Cusumano, 1991). The primary weakness of the software factory was its inflexibility and inability to help firms keep pace with rapid innovations in personal computer and internet software (Liu, 1995; Bethony, 1996; Baker, 1994; Heck, 1995; Bouvier, 1995).

Domain analysis. The tenth major software development method to emerge during the microcomputer era was domain analysis (Arango, 1989). The purpose of domain analysis for computer programming was to systematically capture organizational learning and knowledge, reduce the learning curve, and prevent reinventing the wheel many times (Arango). Domain analysis was a five part process of characterizing the requirements for a class of computer programs, collecting data, analyzing data, classifying data, and evaluating the data for the various classes (Schafer, Prieto-diaz, & Matsumoto, 1994). The strength of domain analysis was its ability to implement 60% more design features per unit time than traditional methods, reduce development costs by 75%, and reduce the number of computer programmers by over 66% (Siy & Mockus, 1999). The weakness of domain analysis was that it propagated the use of older technologies, which were not as powerful and easy to use as internet tools, requiring the re-calibration of the economic models of domain analysis (Siy & Mockus).

INTERNET ERA

The internet era of the 1990s was the most profound decade for the field of computer programming since computers emerged in the 1950s as information technology was transformed from a caterpillar into butterfly in the form of Microsoft Windows (Liu, 1995) and the world wide web (Baker, 1994). The 1990s started off rather slowly with only incremental and inconsequential improvements in Microsoft Windows (Udell, 1993), which almost lulled a few computer programmers into a false sense of security about mainframe, midrange, and microcomputer era software methods (Humphrey, Snyder, & Willis, 1991). However, about the middle of the decade, the 1990s took off and never looked back, as the world wide web exploded onto the global scene (Heck, 1995; Bouvier, 1995; Singhal & Nguyen; 1998), Windows 95 became an instant media sensation (Liu). The coffin had been nailed shut on the mainframe era (Gould, 1999) and its software methods characterized by Humphrey (1989) were suddenly overtaken by events. A few more amazing technological wonders materialized in the 1990s, namely high speed internet service (Sheehan, 1999), crystal clear and colorfully brilliant flat screen displays (Bergquist, 1999), wireless networks (Choy, 1999), palm sized computers (Sakakibara, Lindholm, & Ainamo, 1995), and digital cameras (Gerard, 1999). The creator of Microsoft Windows was now the richest person on the planet with $90 billion (Schulz, 1998) and a 21 year old college student who created Netscape had a net worth of $2 billion (Tetzeli & Puri, 1996).

The U.S. jumped from last to first among technological innovators within the top 10 industrialized nations and internet technologies contributed a much needed breath of life to the once sputtering U.S. economy (Goss, 2001).

Quality management. The first major software development method to emerge during the internet era was quality management (Rigby, Stoddart, & Norris, 1990). The purpose of quality management was to embody the best features of modern quality assurance practices in an international standard and encourage organizations to create and enforce standards for management, administration, finance, product development, and even training (Tingey, 1997). Implementing quality management was a four part process of characterizing the as-is approach to product development, creating an organizational culture of quality, designing a broad ranging program of quality assurance, and conducting formal audits to verify compliance with ISO 9001 (Rigby, Stoddart, & Norris). Productivity and cycle time benefits for quality management were on the order of 13%, quality improvements were on the order of 12 times, and cost savings were on the order of 20% (Naveh, Marcus, Allen, & Moon, 1999). The weakness of quality management was the sheer difficult of designing, deploying, and verifying the use of a set of standard practices for everything from management through training (Rigby, Stoddart, & Norris).

Risk management. The second major software development method to emerge during the internet era was risk management (Boehm, 1991). The purpose of risk management for computer programming was to identify serious or high impact threats to software project success and to mitigate or eliminate negative impacts to the software project (Boehm). Risk management was a seven step process of identifying the risk factors, assessing the risk probabilities and effects on the project, developing risk mitigation strategies, monitoring the risk factors, invoking a contingency plan if necessary, managing a crisis if it occurred, and recovering from the crisis (Boehm). The strengths of risk management were that it cost as little as one hour to identify four risks for computer programming projects and cost savings ranged as high as $6,000,000, while its weaknesses wee that very few projects applied risk management (Freimut, Hartkopf, Kaiser, Kontio, & Kobitzsch, 2001).

Software architecture. The third major software development method to emerge during the internet era was software architecture (Lange & Schwanke, 1991). The purpose of software architecture is to improve the function and performance of computer programs by analyzing them for the purposes of exploiting the economic benefits of satisfying customer needs (Lange & Schanke). Software architecture is a six step process of specifying the structure, analyzing the scope, evaluating the requirements, recording the structural data, evaluating the modularity, and improving the structure of computer programs. The strength of software architecture is that it costs only 10% of total project costs, project savings outweigh its costs, and customer satisfaction levels reach 80% (Clements, 1998). The weakness of software architecture is that software technologies along with their design structure quickly become obsolete and the process was manually intensive (Clements).

Software metrics. The fourth major software development method to emerge during the microcomputer era was software metrics (Kan, 1991). The purpose of using software metrics was to quantitatively measure, predict, and control the degree to which computer programming projects and computer programs themselves satisfied customer requirements (Kan). Examples of software metrics were software cost models (e.g., number of hours to produce a computer program), quality models (e.g., number of defects in computer programs), reliability (e.g., time between system crashes), and customer satisfaction (Kan, 1995). Implementing software metrics was an eight step process of preparing the measurement program, identifying and defining the goals, preparing and conducting interviews, developing the measurement plan, developing a data collection plan, collecting data, analyzing the data, and recording the measurement activities for future reference (Birk, Van Solingen, & Jarvinan, 1998). The strengths of software measurement were that it resulted in cost estimation accuracy of more than 90%, quality and reliability estimation accuracy of more than 99%, and customer satisfaction of nearly 100% (Kan). The weakness of software measurement was that it wsa inconsistently practiced and highly error prone (Johnson & Disney, 1998).

Six sigma. The fifth major software development method to emerge during the internet era was six sigma (Ebenau, 1994). The purpose of six sigma was to meet or exceed all customer needs and expectations by reducing the process variation associated with producing computer programs (Ebenau). Six sigma was a 10 step process of collecting data about processes like costs, defects, and resources; selecting a type of two dimensional graph in which to plot the data for visualization purposes; recording the process data points on the two dimensional graph; estimating the average values and standard deviation of the process data points to establish operating boundaries or control limits; drawing boundaries on the two dimensional graph; labeling the X and Y axes with appropriate labels showing the type of process characteristics and time measures; plotting an initial sample of process data; searching for potential problems as indicated by the plotted process data; and recalculating the control limits to adjust for any problem areas in the process data (Ebenau). The strengths of six sigma were that it resulted in 100% quality of computer programs, high levels of customer satisfaction, 100% increases in schedule estimation accuracy, faster time to market, and achieved cost savings in the millions of dollars (Murugappan & Keeni, 2003). The weakness of six sigma was that humans couldn’t be controlled like machines (Binder, 1997).

Buy versus make. The sixth major software development method to emerge during the internet era was buy versus make (Kontio, 1996). The purpose of buy versus make was to purchase commercial computer programs instead of building custom computer programs as a means of reducing the high costs and risks associated with software development (Kontio). Buy versus make consisted of searching for one or more commercial computer programs that may have met your needs, screening the initial candidates to narrow down the choices, evaluating the final set of alternatives using criteria, applying a system of weighting to distinguish among the alternatives, and applying a process of bottoms up scoring of the alternatives (Kontio). The strengths of buy versus make were that it had a return on investment of nearly six times over traditional software development methods, it reduced software development costs by as much as 85%, and it was up to 70% faster (Ochs, Pfahl, Chrobok-Diening, & Nothhelfer-Kolb, 2001). The weakness of buy versus make was accentuated by the internet age because web products had too many bugs translating into security flaws, insecure purchasing and downloading procedures, incompatible security schemes with other computer programs, poor operating security while users interfaced to the internet, weak security measures that were unknown by ordinary users, and insecure internet maintenance and updating procedures such as virus file updates from popular anti-virus computer programs (Lindqvist & Jonsson, 1998).

Personal processes. The seventh major software development method to emerge during the internet era was personal processes (Humphrey, 1996). The purpose of personal processes were to help computer programmers achieve high levels of programming productivity and software quality by teaching them how to apply project management, software testing, and software metrics (Humphrey). Personal processes were a four step approach to learning how to apply software estimation for project planning, earned value management for tracking time and effort, quality assurance for removing bugs from designs and computer programs, and incremental and iterative development for reducing the costs and risks of long duration computer programming projects (Humphrey). The strengths of personal processes were that they improved productivity by up to 25 times, increased the precision of software estimates, and resulted in nearly zero-defect and failure free computer programs (Ferguson, Humphrey, Khajenoori, Macke & Matvya, 1997). The weaknesses of personal processes were their manual data collection procedures, which resulted in poor data validity and reliability (Johnson & Disney, 1998).

Product lines. The eighth major software development method to emerge during the internet era was product lines (Klingler & Creps, 1996). The purpose of product lines was to reduce the risks of computer programming for software development firms by creating an systematic organizational environment for developing reusable software for specific families or types of computer programs (Klingler & Creps). Products lines had three broad classes of processes: core asset development, product development, and management (Northrop, 2002). Core asset development consisted of defining software architectures, evaluating them, developing reusable software modules, buying as much software as possible, gathering reusable software from external sources, and continuously gaining experience with the various computer programming application areas or domains (Northrop). Product development consisted of developing requirements for the final customer computer programs, building the final products from core assets, and then testing them (Northrop). Management consisted of technical activities (e.g., risk management) and organizational activities (e.g., technology forecasting), according to Northrop. The strengths of product lines were that it could reduce the cost and risk of computer programming by up to 95% (Poulin, 1999). The weakness of product lines was the lack of customer involvement, rapid prototyping, early market feedback, and iterative overlapping development (MacCormack, Verganti, & Iansiti, 2001).

Synch-n-stabilize. The ninth major software development method to emerge during the internet era was synch-n-stabilize (Cusumano, 1997). The purpose of the synch-n-stabilize approach was to get large computer programs to market quickly using many small parallel teams with a maximum amount of entrepreneurial flexibility, freedom, creativity, and autonomy (Cusumano). Synch-n-stabilize was a seven step approach consisting of parallel programming and testing, use of vision statements as requirements, projects consisting of three or four rapid iterations, daily programming and integration testing, schedules with hard deadlines, early customer feedback, and use of small computer programming teams (Cusumano). The strengths of the synch-n-stabilize approach was it helped Microsoft achieve a 34 million percent increase in revenues from 1975 to 1995 with profits of nearly $2 billion (Cusumano & Selby, 1995, p. 5) and customer satisfaction of 90% across its suite of products (Cusumano & Selby, p. 377). The weakness of synch-n-stabilize was that it resulted in poor quality with 63,000 known defects in the Windows 2000 operating system (Dugan, 2000) and numerous security flaws (Johnston, 2002), which translated into 32,000 to 57,000 security flaws (Davis, Humphrey, Redwine, Zibulski, & McGraw, 2004).

Team processes. The tenth major software development method to emerge during the internet era was team processes (Webb & Humphrey, 1999). The purpose of team processes was to improve the likelihood of successfully developing larger computer programs that required teams of computer programmers to produce by teaching them how to apply disciplined project management and quality assurance practices (Webb & Humphrey). The 10 steps of team processes included defining product goals, establishing individual roles such as project manager, producing a development strategy, building incremental release plans, developing quality assurance plans, refining the incremental release plans, applying risk management, performing management reporting, holding management reviews, and conducting post mortem reviews (Webb & Humphrey). The strength of team processes included improved size estimation, effort estimation, schedule estimation, defect density, process yield, productivity, data accuracy, process fidelity, cost reductions, and reliability for larger computer programs (Webb & Humphrey). The weakness of team processes were high training costs (Webb & Humphrey), lack of prototyping, early customer involvement, and overlapping iterations (MacCormack, Verganti, & Iansiti, 2001).

PERSONALIZED ERA

The personalized era of the new millennium continued to bring a series of profound structural changes to the computer and software industry most notably characterized by the obsolescence of the cathode ray tube (Olenich, 2001). The demand for laptop computers began to erode (Plott, 2001) in favor of wireless personal digital assistants (Williams, 2003). Digital cameras continued to gain ground as a must-have killerapp (Scoblete, 2004), and computer technologies such as microprocessors, memories, displays, wireless network interfaces, and integrated circuits continued to retreat in size (Pinkerton, 2002). The U.S. budget deficit of the 1980s returned with a vengeance and itself contributed to the erosion of the global economy (Pelagidis & Desli, 2004) and Enron didn't help matters much by its financial ruin (Barlev & Haddad, 2004). Michael Dell continued to be the darling of Wall Street, not so much for his personal computers or laptop computers, but his vision to extend his product line into personal digital assistants (Kraemer, Dedrick, & Yamashiro, 2000). The elusive high speed broadband internet service of the 1990s finally became the global standard and rumors of wireless broadband service abounded (Farrell, 2004; Rubin, 2004). India finally became a threat to the U.S. computer programming industry as predicted in the early 1990s (Yourdon, 1992) and the politics of outsourcing U.S. jobs to India and other places became a central issue for the U.S. economy (Morrison-Paul & Siegel, 2001).

Agile methods. The first major software development method to emerge during the personalized era was agile methods (MacCormack, Verganti, & Iansiti, 2001). The purpose of agile methods was to improve customer satisfaction with computer programs by creating flexibly malleable software designs, soliciting early market feedback about those designs, and bringing improved computer programs to market as rapidly as possible (MacCormack, Verganti, & Iansiti). Agile methods consisted of implementing critical market needs first in the form of rapid prototypes, releasing those prototypes to customers for evaluation, implementing the customer changes, conducting early system testing, and quickly releasing beta versions in an overlapping and iterative fashion, while concurrently starting the next iteration in parallel (MacCormack, Verganti, & Iansiti). The strength of agile methods was its correlation between internet firms judged to have high quality websites and computer programming practices designed to solicit early market feedback with rapid prototypes and beta releases, while implementing overlapping development stages (MacCormack, Verganti, & Iansiti). The weakness of agile methods was its lack of project management and quality assurance practices (Humphrey, 1996), which could lead to a high number of defects (Dugan, 2000) and security issues for internet products and services (Johnston, 2002; Davis, Humphrey, Redwine, Zibulski, & McGraw, 2004; Ochs, Pfahl, Chrobok-Diening, & Nothhelfer-Kolb, 2001).

REFERENCES

Ahlers, D. M. (1981). Personal computers: You can't afford to ignore them, but be cautious. *ABA Banking Journal, 73*(10), 101-103.

Amdahl, G. M. (1967). Validity of the single-processor approach to achieving large scale computing capabilities. *Proceedings of the American Federation of Information Processing Societies Conference (AFIPS 1967), Atlantic City, New Jersey, USA*, 483-485.

Arango, G. (1989). Domain analysis: From art form to engineering discipline. *Proceedings of the Fifth international Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, USA*, 152-159.

Asada, M., & Yan, P. M. (1998). Strengthening software quality assurance. *Hewlett Packard Journal, 49*(2), 89-97.

Auer, M., & Biffl, S. (2004). Increasing the accuracy and reliability of analogy based cost estimation with extensive project feature dimension weighting. Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2004), Redondo Beach, California, USA, 147-155.

Baker, S. (1994). Mosaic surfing at home and abroad. *Proceedings of the 22nd Annual ACM SIGUCCS Conference on User Services, Ypsilanti, Michigan, USA*, 159-163.

Barlev, B., & Haddad, J. R. (2004). Dual accounting and the enron control crisis. *Journal of Accounting, Auditing & Finance, 19*(3), 343-359.

Basili, V. R., & Perricone, B. T. (1984). Software errors and complexity: An empirical investigation. *Communications of the ACM, 27*(1), 42-52.

Basili, V. R., & Turner, J. (1975). Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering, 1*(4), 390-396.

Benson, J. P., & Saib, S. H. (1978). A software quality assurance experiment. *Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues, New York, New York, USA*, 87-91.

Bergquist, C. J. (1999). Liquid crystal displays: The easy way. *Popular Electronics, 16*(2), 31-37.

Bersoff, E. H., Henderson, V. D., & Siegel, S. G. (1978). Software configuration management. *Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues, New York, New York, USA*, 9-17.

Bethoney, H. (1996). Microsoft's office 97 suite: Together at last. *PC Week, 13*(50), 104-104.

Bewley, W. L., Roberts, T. L., Schroit, D., & Verplank, W. L. (1983). Human factors testing in the design of xerox's 8010 star office workstation. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Boston, Massachusetts, USA*, 72-77.

Binder, R. V. (1997). Can a manufacturing quality model work for software? *IEEE Software, 14*(5), 101-102, 105.

Birk, A., Van Solingen, R., & Jarvinan, J. (1998). *Business impact, benefit, and cost of applying GQM in industry: An in-depth, long-term investigation at schlumberger RPS* (IESE-Report 040.98/E). Kaiserslautern, Germany: University of Kaiserslautern, Fraunhofer-Institute for Experimental Software Engineering.

Boehm, B. W. (1991). Software risk management: Principles and practices. *IEEE Software, 8*(1), 32-41.

Booch, G. (1986). Object-oriented development. *IEEE Transactions on Software Engineering, 12*(2), 211-221.

Bouvier, D. J. (1995). Versions and standards of HTML. *ACM SIGAPP Applied Computing Review, 3*(2), 9-15.

Brennan, P. J. (1975). Here comes the microprocessor. *Banking, 67*(10), 100-105.

Broadhead, N., Gregory, J., Pelkie, C., Main, S, & Close, R. T. (1985). The unique advantages of the macintosh. *Proceedings of the 13th Annual ACM SIGUCCS Conference on User Services, Toledo, Ohio, USA*, 87-90.

Brooks, F. P. (1975). *The mythical man month: Essays on software engineering*. Reading, MA: Addison Wesley.

Carlson. W. E., Druffel, L. E., Fisher, D. A., & Whitaker, W. A. (1980). Introducing Ada. *Proceedings of the ACM Annual Conference, New York, New York, USA*, 263-271.

Carlsson, S. A. (1988). A longitudinal study of spreadsheet program use. *Journal of Management Information Systems, 5*(1), 82-100.

Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K., & Wong, M. Y. (1992). Orthogonal defect classification: A concept for in-process measurements. *IEEE Transactions on Software Engineering, 18*(11), 943-956.

Choy, K. C. (1999). Reusable management frameworks for third generation wireless networks. *Bell Labs Technical Journal, 4*(4), 171-189.

world auto industry. Boston, MA: Harvard Business School.

Clements, P. C. (1998). Software architecture in practice. *Proceedings of the Fifth Systems and Software Technology Conference (STC 1998), Salt Lake City, Utah, USA*, 1-66.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM, 13*(6), 377-387.

Collins, R. H., & Stem, D. E. (1986). Mainframe statistics on a micro? *Journal of Marketing Research*, *23*(2), 191-194.

Cugola, G., Fugetta, A., Fusaro, P., Von Wangenheim, C. G., Lavazza, L., Manca, S., Pagone, M. R., Ruhe, G., & Soro, R. (1997). A case study of evaluating configuration management practices with goal oriented measurement. *Proceedings of the Fourth International Software Metrics Symposium (METRICS 1997), Albuquerque, New Mexico, USA*, 144-151.

Cusumano, M. A. (1989). The software factory: A historical interpretation. *IEEE Software, 6*(2), 23-30.

Cusumano, M. A. (1991). Factory concepts and practices in software development. *IEEE Annals of the History of Computing, 13*(1), 3-32.

Cusumano, M. A., & Selby, R. W. (1995). *Microsoft secrets: How the world's most powerful software company creates technology, shapes markets, and manages people*. New York, NY: The Free Press.

Cusumano, M. A., & Selby, R. W. (1997). How microsoft builds software. *Communications of the ACM, 40*(6), 53-61.

Davis, N., Humphrey W., Redwine, S. T., Zibulski, G., & McGraw, G. (2004). Processes for producing secure software: Summary of US national cybersecurity summit subgroup report. *IEEE Security and Privacy, 2*(3), 18-25.

Diaz, M., & Sligo, J. (1997). How software process improvement helped motorola. *IEEE Software, 14*(5), 75-81.

Dijkstra, E. H. (1972). The humble programmer. *Communications of the ACM, 15*(10), 859-866.

Dijkstra, E. W. (1968). The structure of THE: Multiprogramming system. *Communications of the ACM, 11*(5), 341-346.

Dugan, S. M. (2000). Windows 63,000: When does a bug become a point of collaboration? *InfoWorld*, *22*(8), 94-94.

Dyer, M. (1980). The management of software engineering part IV: Software engineering development practices. *IBM systems Journal, 19*(4), 451-465.

Ebenau, R. G. (1994). Predictive quality control with software inspections. *Crosstalk, 7*(6), 9-16.

Eckhouse, R. H. (1975). *Minicomputer systems: Organization and programming (PDP-11)*. Upper Saddle River, NJ: Prentice Hall.

Endres, A. (1975). An analysis of errors and their causes in system programs. *Proceedings of the International Conference on Reliable Software, Los Angeles, California, USA*, 327-336.

Everett, W. W. (1995). The software reliability engineering process. *Proceedings of the Second IEEE Software Engineering Standards Symposium (ISESS 1995), Montreal, Quebec, Canada*, 248-248.

Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal, 12*(7), 744-751.

Fagan, M. E. (1986). Advances in software inspections. *IEEE Transactions on Software Engineering, 15*(3), 182-211.

Farrell, M. (2004). Broadband boom. *Multichannel News, 25*(44), 1-2.

Ferguson, P., Humphrey, W. S., Khajenoori, S., Macke, S., & Matvya, A. (1997). Results of applying the personal software process. *IEEE Computer, 30*(5), 24-31.

Flaherty, M. J. (1985). Programming process productivity measurement system for system/370. *IBM Systems Journal, 24*(3/4), 168-175.

Freimut, B., Hartkopf, S., Kaiser, P., Kontio, J., & Kobitzsch, W. (2001). *An industrial case study of implementing software risk management* (IESE-Report 016.01/E). Kaiserslautern, Germany: University of Kaiserslautern, Fraunhofer-Institute for Experimental Software Engineering.

Gerard, A. (1999). Digital cameras are a go for the mass market. *Electronic Publishing, 23*(12), 16-16.

Goss, E. (2001). The internet's contribution to U.S. productivity growth. *Business Economics, 36*(4), 32-42.

Gould, L. S. (1999). Where have all the mainframes and minicomputers gone? *Automotive Manufacturing and Production, 111*(3), 56-59.

Guinan, P. J., Cooprider, J. G., & Sawyer, S. (1997). The effective use of automated application development tools. *IBM Systems Journal, 36*(1), 124-139.

Hardy, C., Thompson, B., & Edwards, H. (1994). *A preliminary survey of method use in the UK* (Occasional Paper No: 94-12). Sunderland, UK: University of Sunderland, School of Computing and Information Systems, Commercial Software Engineering Group.

Harrison, T. P. (1985). Micro versus mainframe performance for a selected class of mathematical programming problems. *Interfaces, 15*(4), 14-19.

Heck, M. (1995). Dominant netscape navigator sets its own standards. *InfoWorld, 17*(48), 125-126.

Hennell, M. A., Hedley, D., & Woodward , M. R. (1977). Quantifying the test effectiveness of algol 68 programs. *Proceedings of the Strathclyde ALGOL 68 Conference, Glasgow, Scotland*, 36-41.

Hoare, C. A. R. (1971). Proof of a program: FIND. *Communications of the ACM, 14*(1), 39-45.

Hoffnagle, G. F., & Beregi, W. E. (1985). Automating the software development process. *IBM Systems Journal, 24*(2), 102-120.

Huff, C. C. (1988). Elements of a realistic CASE tool adoption budget. *Communications of the ACM, 35*(4), 45-54.

Humphrey, W. S. (1989). *Managing the software process*. Reading, MA: Addison Wesley.

Humphrey, W. S. (1996). Using a defined and measured personal software process. *IEEE Software, 13*(3), 77-88.

Humphrey, W. S., Snyder, T. R., & Willis, R. R. (1991). Software process improvement at hughes aircraft. *IEEE Software, 8*(4), 11-23.

Institute of Electrical and Electronics Engineers. (1989). *IEEE standard for software quality assurance plans* (IEEE Std 730-1989). New York, NY: Author.

Institute of Electrical and Electronics Engineers. (1990). *IEEE standard for software configuration management plans* (IEEE Std 828-1990). New York, NY: Author.

Institute of Electrical and Electronics Engineers. (1993). *IEEE guide for software verification and validation plans* (IEEE Std 1059-1993). New York, NY: Author.

Ives, B. (1982). Graphical user interfaces for business information systems. *MIS Quarterly, 6*(4), 15-47.

Johnson, P. M., & Disney, A. M. (1998). The personal software process: A cautionary case study. *IEEE Software, 15*(6), 85-88.

Johnston, S. J. (2002). Serious security holes in internet explorer. *PC World, 20*(8), 49-49.

Jones, C. L. (1985). A process-integrated approach to defect prevention. *IBM Systems Journal, 24*(2), 150-165.

Kan, S. H. (1991). Modeling and software development quality. *IBM Systems Journal. 30*(3), 351-362.

Kan, S. H. (1995). *Metrics and models in software quality engineering*. Reading, MA: Addison Wesley.

Kan, S. H., Dull, S. D., Amundson, D. N., Lindner, R. J., & Hedger, R. J. (1994). AS/400 software quality management. *IBM Systems Journal, 33*(1), 62-88.

Kernighan, B., & Ritchie, D. M. (1978). *The C programming language*. Upper Saddle River, NJ: Prentice Hall.

Klingler, C. D., & Creps, R. (1996). Integrating and applying processes and methods for product line management. *Proceedings of the 10th International Software Process Workshop (ISPW 1996), Dijon, France*, 97-100.

Knuth, D. E. (1963). Computer-drawn flowcharts. *Communications of the ACM, 6*(9), 555-563.

Kontio, J. (1996). A case study in applying a systematic method for COTS selection. *Proceedings of the 18th International Conference on Software Engineering (ICSE 1996), Berlin, Germany*, 201-209.

Kraemer, K. L., Dedrick, J., & Yamashiro, S. (2000). Refining and extending the business model with information technology: Dell computer corporation. *Information Society, 16*(1), 5-21.

Lange, R., & Schwanke, R. W. (1991). Software architecture analysis: A case study. *Proceedings of the Third international Workshop on Software Configuration Management, Trondheim, Norway*, 19-28.

Lanubile, F. (1996). Why software reliability predictions fail. *IEEE Software, 13*(4), 131-132, 137.

Lee, E. (1997). Software inspections: How to diagnose problems and improve the odds of organizational acceptance. *Crosstalk, 10*(8), 10-13.

Lim, W. C. (1994). Effects of reuse on quality, productivity, and economics. *IEEE Software, 11*(5), 23-30.

Lim, W. C. (1998). *Managing software reuse: A comprehensive guide to strategically reengineering the organization for reusable components*. Upper Saddle River, NJ: Prentice Hall.

Lindqvist, U., & Jonsson, E. (1998). A map of security risks associated with using COTS. *IEEE Computer, 31*(6), 60-66.

Linger, R. C. (1980). The management of software engineering part III: Software engineering design practices. *IBM systems Journal, 19*(4), 432-450.

Littlewood, B. (1975). A reliability model for Markov structured software. *Proceedings of the International Conference on Reliable Software, Los Angeles, California, USA*, 204-207.

Liu, D. (1995). Windows 95: Should you take the plunge? *Business Forum, 21*(1/2), 28-32.

Liu, S., & Adams, R. (1995). Limitations of formal methods and an approach to improvement. *Proceedings Asia Pacific Software Engineering Conference, Brisbane, Australia*, 498-417.

Lockwood, R. (1988). The premium 386: Updating a technology. *Personal Computing, 12*(6), 93-97.

Luqi. (1989). Software evolution through rapid prototyping. *IEEE Computer, 22*(5), 13-25.

MacCormack, A., Verganti, R., & Iansiti, M. (2001). Developing products on internet time: The anatomy of a flexible development process. *Management Science, 47*(1), 133-150.

Mays, R. G., Jones, C. L., Holloway, G. J., & Studinski, D. P. (1990). Experiences with defect prevention. *IBM Systems Journal, 29*(1), 4-32.

McGibbon, T. (1996). *A business case for software process improvement* (Contract Number F30602-92-C-0158). Rome, NY: Air Force Research Laboratory—Information Directorate (AFRL/IF), Data and Analysis Center for Software (DACS).

McGibbon, T. (1997). *Modern empirical cost and schedule estimation* (Contract Number F30602-89-C-0082). Rome, NY: Air Force Research Laboratory—Information Directorate (AFRL/IF), Data and Analysis Center for Software (DACS).

Mercurio, V. J., Meyers, B. F., Nisbet, A. M, & Radin, G. (1990). AD/cycle strategy and architecture. *IBM Systems Journal, 29*(2), 170-188.

Merwin, R. E. (1972). Estimating software development schedules and costs. *Proceedings of the Ninth Workshop on Design Automation, New York, New York, USA*, 1-4.

Mills, H. D. (1980). The management of software engineering part I: Principles of software engineering. *IBM Systems Journal, 19*(4), 415-420.

Monk, J. (1980). Developments in microelectronics technology and the economics of the semiconductor industry. *International Journal of Social Economics, 7*(1), 13-23.

Morrison-Paul, C. J., & Siegel, D. S. (2001). The impacts of technology, trade, and outsourcing on employment and labor composition. *Scandinavian Journal of Economics, 103*(2), 241-264.

Murray, C. J. (1989). Disk drive spins faster for higher performance. *Design News, 45*(21), 112-113.

Murugappan, M., & Keeni, G. (2003). Blending CMM and six sigma to meet business goals. *IEEE Software, 20*(2), 42-48.

Naur, P., & Randell, B. (1969). Software engineering. *NATO Software Engineering Conference, Garmish, Germany,* 1-136.

Naveh, E., Marcus, A., Allen, G., & Moon, H. K. (1999). *ISO 9000 survey '99: An analytical tool to assess the costs, benefits, and savings of ISO 9000 registration*. New York, NY: McGraw Hill.

Needham, R. M., & Hartley, D. F. (1969). Theory and practice in operating system design. *Proceedings of the Second Symposium on Operating Systems Principles, Princeton, New Jersey, USA*, 8-12.

Niskanen, W. A. (1968). The peculiar economics of bureaucracy. *American Economic Review, 58*(2), 293-305.

Northrop, L. M. (2002). SEI's software product line tenets. *IEEE Software, 19*(4), 32-40.

Ochs, M., Pfahl, D., Chrobok-Diening, G., & Nothhelfer-Kolb, B. (2001). A method for efficient measurement-based COTS assessment and selection: Method description and evaluation results. *Proceedings of the Seventh International Software Metrics Symposium, London, England*, 285-296.

O'Connell, E., & Saiedian, H. (2000). Can you trust software capability evaluations? *IEEE Computer, 33*(2), 28-35.

Olenich, D. (2001). Apple cans CRT for LCD monitors. *TWICE: This Week in Consumer Electronics, 16*(3), 9-9.

O'Neill, D. (1980). The management of software engineering part II: Software engineering program. *IBM Systems Journal, 19*(4), 421-431.

Owen, G. S., Gagliano, R., & Honkanen, P. (1987). Functional specifications of reusable MIS software in ada. *Proceedings of the Conference on Tri-Ada, Arlington, Virginia, USA*, 19-26.

Pechura, M. A. (1983). Comparing two microcomputer operating systems: CP/M amd HDOS. *Communications of the ACM, 26*(3), 188-195.

Pelagidis, T., & Desli, E. (2004). Deficits, growth, and the current slowdown: What role for fiscal policy? Journal of *Post Keynesian Economics, 26*(3), 461-469.

Pine, B. J. (1989). Design, test, and validation of the Application System/400 through early user involvement. *IBM System Journal, 28*(3), 376-385.

Pinkerton, G. (2002). Many technologies contribute to miniaturization. *Electronic Design, 50*(27), 36-36.

Plott, D. (2001). Palming off laptops. *Far Eastern Economic Review, 164*(22), 42-42.

Poulin, J. S. (1999). The economics of software product lines. *International Journal of Applied Software Technology, 3*(1), 20-34.

Quinnan, R. E. (1980). The management of software engineering part V: Software engineering management practices. *IBM systems Journal, 19*(4), 466-477.

Radice, R. A., Harding, J. T., Munnis, P. E., & Phillips, R. W. (1985). A programming process study. *IBM Systems Journal, 24*(2), 91-101.

Randles, F. (1983). On the diffusion of computer terminals in an established engineering environment. *Management Science, 29*(4), 465-476.

Reifer, D., Craver, J., Ellis, M., & Strickland, D. (2000). Is ada dead or alive within the weapon system world. *Crosstalk, 13*(12), 22-25.

Rigby, P. J., Stoddart, A. G., & Norris, M. T. (1990). Assuring quality in software: Practical experiences in attaining ISO 9001. *British Telecommunications Engineering, 8*(4), 244-249.

Ritchie, D. M., & Thompson, K. (1974). The UNIX time-sharing system. *Communications of the ACM, 17*(7), 365-375.

Ross, D. T., & Schoman, K. E. (1977). Structured analysis for requirements definition. *IEEE Transactions on Software Engineering, 3*(1), 6-15.

Rubey, R. J., Browning, L. A., & Roberts, A. R. (1989). Cost effectiveness of software quality assurance. *Proceedings of the IEEE National Aerospace and Electronics Conference (NAECON 1989), Dayton, Ohio, USA*, 1614-1620.

Rubin, D. K. (2004). Designers follow the wireless boom across the U.S. and around the world. *Engineering News Record, 252*(26), 66-67.

Runeson, P., & Isacsson, P. (1998). Software quality assurance: Concepts and misconceptions. *Proceedings of the 24th Euromicro Conference (EUROMICRO 1998), Vesteras, Sweden*, 20853-20859.

Russell, G. W. (1991). Experience with inspection in ultralarge-scale developments. *IEEE Software, 8*(1), 25-31.

Sakakibara, K., Lindholm, C., & Ainamo, A. (1995). Product development strategies in emerging markets: The case of personal digital assistants. *Business Strategy Review, 6*(4), 23-38.

Sammet, J. E. (1962). Basic elements of COBOL 61. *Communications of the ACM, 5*(5), 237-253.

Sammet, J. E. (1972). Programming languages: History and future. *Communications of the ACM, 15*(7), 601-610.

Schafer, W., Prieto-diaz, R., & Matsumoto, M. (1980). *Software reusability*. New York, NY: Ellis Horwood.

Schulz, Y. (1998). Bill gates: Robber baron or productivity savior? *Computing Canada, 24*(18), 20-20.

Scoblete, G. (2004). Ahead software targets digital cameras. *TWICE: This Week in Consumer Electronics, 19*(14), 32-32.

Sheehan, M. (1999). Faster, faster: Broadband access to the internet. *Online, 23*(4), 18-24.

Shneiderman, B., Mayer, R., McKay, D., & Heller, P. (1977). Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM, 20*(6), 373-381.

Singhal, S., & Nguyen, B. (1998). The Java factor. *Communications of the ACM, 41*(6), 34-37.

Siy, H., & Mockus, A. (1999). Measuring domain engineering effects on software change cost. *Proceedings of the Sixth IEEE International Symposium on Software Metrics, Boca Raton, Florida, USA*, 304-311.

Smith, S. B. (1965). Planning transistor production by linear programming. *Operations Research, 13*(1), 132-139.

Solomon, M. B. (1966). Economies of scale and the IBM system/360. *Communications of the ACM, 9*(6), 435-440.

Stanley, K. B. (1973). International telecommunications industry: Interdependence of market structure and performance under regulation. *Land Economics, 49*(4), 391-403.

Stevens, W. P., Myers, G. L., Constantine, L. L. (1974). Structured design. *IBM Systems Journal, 13*(2), 115-139.

Stinson, C. (1996). Clear process: Flowcharts + analysis = business process reengineering. *PC Magazine, 15*(2), 1-1.

Sulack, R. A., Lindner, R. J., & Dietz, D. N. (1989). A new development rhythm for AS/400 software. *IBM Systems Journal, 28*(3), 386-406.

Tezeli, R., & Puri, S. (1996). What it's really like to be marc andreessen. *Fortune, 134*(11), 136-144.

Tingey, M. O. (1997). *Comparing ISO 9000, malcolm baldrige, and the SEI CMM for software: A reference and selection guide*. Upper Saddle River, NJ: Prentice Hall.

Tyson, L. D. (1992). *Who's bashing whom: Trade conflict in high technology industries*. Washington, DC: Institute for International Economics.

Udell, J. (1993). Is there a better windows 3.1 than windows 3.1? *Byte Magazine, 18*(12), 85-90.

Walston, C. E., & Felix, C. P. (1977). A method of programming measurement and estimation. *IBM Systems Journal, 16*(1), 54-73.

Webb, D., & Humphrey W. S. (1999). Using the TSP on the taskview project. *Crosstalk, 12*(2), 3-10.

Williams, J. (2003). Taming the wireless frontier: PDAs, tablets, and laptops at home on the range. *Computers in Libraries, 23*(3), 10-15.

Wingo, W. (1998). New flowcharting software helps meet ISO management rules. *Design News, 54*(6), 24-24.

Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM, 14*(4), 221-227.

Woodward, S. (1999). Evolutionary project management. *IEEE Computer, 32*(10), 49-57.

Yamamoto, T. (1992). *Fujitsu: What mankind can dream, technology can achieve*. Tokyo, Japan: Toyo Keizai.

Yourdon, E. (1992). *Decline and fall of the american programmer*. Englewood Cliffs, NJ: Prentice Hall.