History of Computers, Electronic Commerce and Agile Methods

DAVID F. RICO

HASAN H. SAYANI

Graduate School of Management and Technology
University of Maryland University College

RALPH F. FIELD

Graduate School of Management and Technology
University of Maryland University College

Abstract

The purpose of this chapter is to present a literature review relevant to a study of using agile methods to manage the development of Internet websites and their subsequent quality. This chapter places website quality within the context of the $2.4 trillion U.S. electronic commerce industry. Thus, this chapter provides a history of electronic computers, electronic commerce, software methods, software quality metrics, agile methods and studies on agile methods. None of these histories are without controversy. For instance, some scholars begin the study of the electronic computer by mentioning the emergence of the Sumerian text, Hammurabi code or the abacus. We, however, will align our history with the emergence of the modern electronic computer at the beginning of World War II. The history of electronic commerce also has poorly defined beginnings. Some studies of electronic commerce begin with the widespread use of the Internet in the early 1990s. However, electronic commerce cannot be appreciated without establishing a deeper context. Few scholarly studies, if any, have been performed on agile methods, which is the basic purpose of this literature review. That is, to establish the context to conduct scholarly research within the fields of agile methods and electronic commerce.

ADVANCES IN COMPUTERS, VOL. 73
ISSN: 0065-2458/DOI: 10.1016/S0065-2458(08)00401-4

1. Introduction

Agile methods are an approach for managing the development of new products based on principles of flexible manufacturing and lean development. The use of agile methods for Internet software was a reaction to the emergence of traditional software development methods, which were too cumbersome, expensive, rigid and fraught with failure. Downsizing was the norm and traditional methods were being used by large corporations in decline, rather than by young, energetic firms on the rise. Millions of websites were created overnight by anyone with a computer and a modicum of curiosity. Agile methods marked the end of traditional methods in the minds of their creators.

Traditional methods for managing software development were created when the first commercial computers began emerging in the 1950s. Scientists and engineers began creating increasingly more powerful and complex computer systems, and inordinately complex computer programs beyond the comprehension of a single human. These early computer programs had millions of components to perform the simplest of operations, giving rise to traditional methods. The rise of traditional methods is also linked to the debut of the commercial software industry in the 1960s. Traditional methods consisted of formal project plans, well-documented customer requirements, detailed engineering processes, hundreds of documents and rigorous testing.

Agile methods emerged with a focus on iterative development, customer feedback, well-structured teams and flexibility. Internet technologies such as HTML and Java were powerful new prototyping languages, enabling smaller teams to build bigger software products in less time. Because they could be built faster, customers could begin to see finished software sooner and provide earlier feedback, and developers could rapidly refine their software. This gave rise to closed-loop, circular, highly recursive and tightly knit processes for rapidly creating Internet software, leading to improvements in website quality for electronic commerce.

2. History of Computers and Software

2.1 Electronic Computers

Electronic computers are simply machines that perform useful functions such as mathematical calculations or inputting, processing and outputting data and information in meaningful forms [1]. As shown in Figure 1, modern electronic computers are characterized by four major generations: first-generation vacuum tube computers from 1940 to 1950, second-generation transistorized computers from 1950 to 1964, third-generation integrated circuit computers from 1964 to 1980 and fourth-generation microprocessor computers from 1980 to the present [1]. First-generation or vacuum tube computers consisted of the electronic numerical integrator and calculator or ENIAC; electronic discrete variable computer or EDVAC; universal automatic computer or UNIVAC; and Mark I, II and III computers [1]. Second-generation or transistorized computers consisted of Philco's TRANSAC S-1000, Control Data Corporation's 3600 and International Business Machine's 7090 [1]. Third-generation or integrated-circuit-based computers consisted of International Business Machine's System/360, Radio Corporation of America's Spectra 70 and

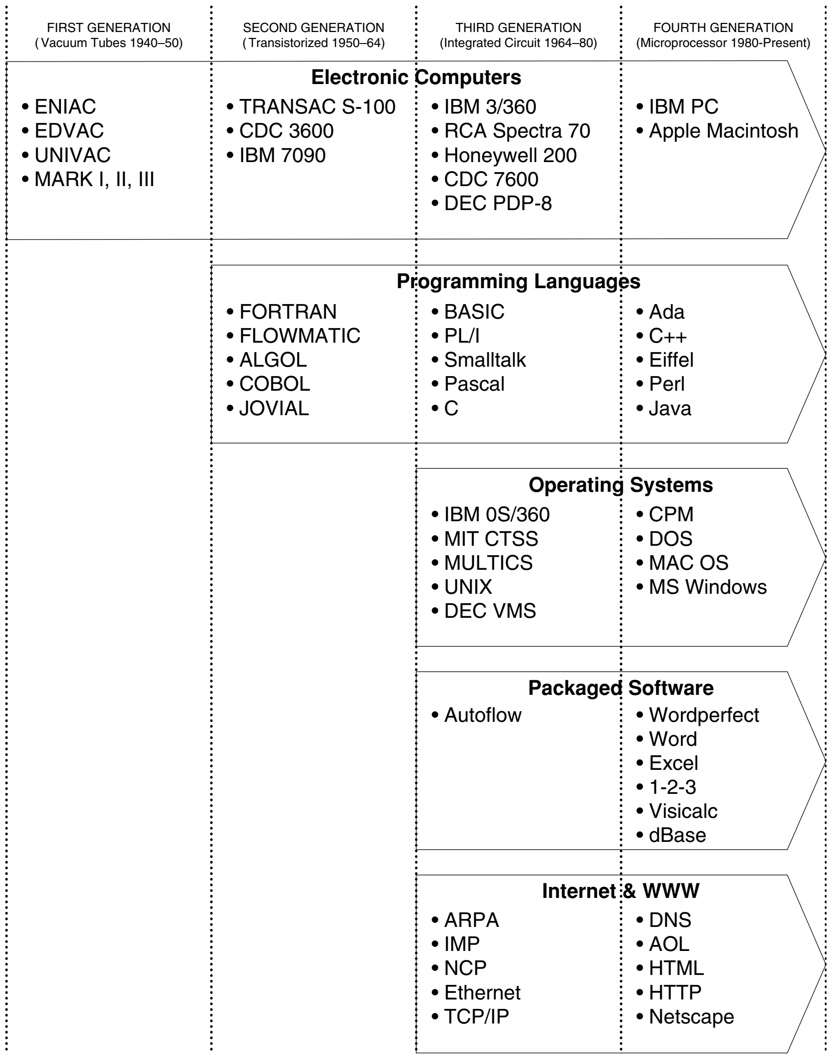

FIG. 1. Timeline and history of computers and software.

Honeywell's 200 [1]. Late third-generation computers included Cray's CDC 7600 as well as Digital Equipment Corporation's PDP-8, VAX 11/750 and VAX 11/780 [4]. Fourth-generation or microprocessor-based computers included the International Business Machine's Personal Computer or PC and Apple's Macintosh [3].

2.2 Programming Languages

Programming languages are defined as ‘any of various languages for expressing a set of detailed instructions for a digital computer’ [5]. By 1972, there were 170 programming languages in the U.S. alone [6] and today there are over 8500 programming languages worldwide [7]. First-generation or vacuum tube computers did not have any programming languages [6]. Second-generation or transistorized computers were characterized by an explosion of programming languages, the most notable of which included formula translation or FORTRAN, flowchart automatic translator or FLOWMATIC, algorithmic language or ALGOL, common business-oriented language or COBOL, Jules own version of the international algorithmic language or JOVIAL and the list processing language or LISP [6]. Third-generation or integrated-circuit-based computers likewise experienced a rapid increase in programming languages, the most notable of which were the beginner’s all-purpose symbolic instructional code or BASIC, programming language one or PL/1, Smalltalk, Pascal and C [8]. Fourth-generation or microprocessor-based computers continued the trend of introducing new programming languages, such as Ada, C++, Eiffel, Perl, Java and C#.

2.3 Operating Systems

Operating systems are simply a layer of software between the computer hardware and end-user applications used for controlling hardware peripherals such as keyboards, displays and printers [2]. First-generation or vacuum tube computers did not have any operating systems and ‘all programming was done in absolute machine language, often by wiring up plugboards’ [3]. Second-generation or transistorized computers did not have any operating systems per se, but were programmed in assembly languages and even using the early computer programming language called formula translation or FORTRAN [3]. Third-generation or integrated-circuit-based computers consisted of the first formalized multi-programming operating systems and performed useful functions such as spooling and timesharing [3]. Examples of third-generation operating systems included IBM’s Operating System/360, the Massachusetts Institute of Technology’s compatible time-sharing system or CTSS, the multiplexed information and computing service or MULTICS, the uniplexed information and computer system or UNICS, which became UNIX, and Digital Equipment Corporation’s virtual memory system or VMS [3]. Fourth-generation or microprocessor-based computers consisted of the control program for microcomputers or CPM, disk operating system or DOS, Apple’s Macintosh operating system or MAC OS and Microsoft’s Windows [3].

2.4 Packaged Software

Software is defined as 'instructions required to operate programmable computers, first introduced commercially during the 1950s' [9]. The international software industry grew slowly in revenues for commercially shrink-wrapped software from about zero in 1964, to $2 billion per year in 1979, and $50 billion by 1990 [10]. It is important to note that the custom, non-commercially available software industry was already gaining billions of dollars in revenue by 1964 [10]. First-generation or vacuum tube computers, much like programming languages and operating systems, did not have any software and 'all programming was done in absolute machine language' [3]. Second-generation or transistorized computers were characterized by bundled software, e.g., software shipped free with custom computer systems, and customized software such as International Business Machine's SABRE airline reservation system and the RAND Corporation's SAGE air defense system [10]. Third-generation or integrated-circuit-based computers saw the first commercialized shrink-wrapped software such as Applied Data Research's Autoflow flowcharting software [12] and the total annual sales for commercial software were only $70 million in 1970 compared with over $1 billion for custom software [10]. In part due to the U.S. Justice Department's anti-trust lawsuit against IBM around 1969, commercial software applications reached over 175 packages for the insurance industry in 1972 and an estimated $2 billion in annual sales by 1980 [10]. Fourth-generation or microprocessor-based computers represented the golden age of shrink-wrapped computer software and were characterized by Microsoft's Word and Excel, WordPerfect's word processor, Lotus' 1-2-3 and Visicorp's Visicalc spreadsheets, and Ashton Tate's dBase database [13]. By 1990, there were over 20 000 commercial shrink-wrapped software packages in the market [14]. And, the international software industry grew to more than $90 billion for pre-packaged software and $330 billion for all software-related products and services by 2002 [15] and is projected to reach $10.7 billion for the software as a service or SAAS market by 2009 [16].

2.5 Internet and WWW

The Internet is defined as a network of millions of computers, a network of networks, or an internetwork [17]. First-generation or vacuum tube computers were not known to have been networked. Late second-generation or transistorized computers gave rise to the Internet as it is known today [18]. Second-generation computers of the 1960s gave rise to packet switching theory, the first networked computers, the U.S. military's advanced research project's agency or ARPA and the first interface message processor or IMP [18], [19]. An MIT researcher published the first paper on packet switching theory, devising what was known as the 'Galactic Network' [18], [19]. This same researcher was appointed head of ARPA's Behavioural Sciences and Command and Control Programs [18]. The ARPANET was developed to see if machines could be networked and many machines, such as second-generation IBM 7090s were on the early ARPANET, even as third-generation computers began to emerge [18]. Third-generation or integrated-circuit-based computers took the early networking concepts devised during the second generation and formalized them into the ARPANET and Internet concepts as they are known today [18] All this came together when the Bolt, Beranek and Newman (BBN) Corporation installed the first IMP at UCLA in 1969 and the first host computer was connected [18]. The Network Working Group (NWG) completed the initial ARPANET host-to-host protocol called the Network Control Protocol (NCP) in 1970, network users began developing applications from 1971 to 1972 and the first public demonstration of the ARPANET took place in 1972 [18]. In summary, late third-generation computers of the 1970s gave rise to the network control protocol or NCP, email, open architecture networking, ethernet, transmission control protocol, Internet protocol and one of the first bulletin boards by Compuserve [18]. Late third-generation computers gave rise to the hyper text markup language or HTML. Tim Berners-Lee, a British physicist working for the Conseil Européen pour la Recherche Nucléaire or CERN (European Organization for Nuclear Research) created an early HTML prototype called Enquire, which ran on a Norwegian minicomputer called the Norsk Data Machine running the NORD Time Sharing System or NORD-TSS. HTML was first proposed to the Internet Engineering Task Force (IETF) in 1991 to 1993 and became an official standard in the 1995 to 1996 timeframe. Early fourth-generation or microprocessor-based computers gave rise to the domain name system or DNS and Prodigy and AOL were created [18]. Using middle fourth-generation computers, Tim Berners-Lee of CERN had created the first web server on a NeXTcube running the NeXTstep operating system, which was a UNIX variant, and is credited with the creation of the hyper text transfer protocol or HTTP in 1989. Middle fourth-generation computers of the Internet era were adapted to the formalized IETF HTML and HTTP standards and gave rise to Mosaic and Netscape, which caused the number of computers on the Internet to reach one million by 1992 and 110 million by 2001 [19]. Using ideas from Tim Berners-Lee, Marc Andreessen and Eric Bina, students at the National Centre for Supercomputing Applications (NCSA) at University of Illinois at Urbana-Champaign created the first popular WWW browser called Mosaic in 1992. Marc Andreessen formed Mosaic Communications Corporation to commercialize his WWW browser, which was renamed Netscape to deconflict with the NCSA's intellectual property claims. Netscape is credited with popularizing the WWW and Internet as it is known today.

3. History of Electronic Commerce

3.1 Electronic Commerce

The purpose of this section is to give a brief overview of the history of electronic commerce. Though electronic commerce seemed to enter into mainstream public consciousness in the 1990s, the electronic commerce industry is as old as the computer and software industries themselves. This section attempts to give readers a small appreciation of the earliest beginnings of the electronic commerce industry, as we believe electronic commerce is the key for the convergence of electronic computers, operating systems, programming languages, packaged software and the Internet and WWW (e.g., Apple iPhones). From a simple perspective, electronic commerce is defined as sharing of business information, maintaining business relationships or conducting business transactions using the Internet. However, there are at least four comprehensive definitions of electronic commerce [20]:

1. Communications perspective. Electronic commerce is the delivery of information, products, services or payments via telephones or computer networks.
2. Business process perspective. Electronic commerce is the application of technology to the automation of business transactions and workflows.
3. Service perspective. Electronic commerce is a tool that helps firms, consumers and managers cut service costs, improve quality and speed delivery.
4. Online perspective. Electronic commerce provides the capability of buying and selling products and information on the Internet and other online services.

Electronic commerce is one of the most misunderstood information technologies [20]. For instance, there is a tendency to categorize electronic commerce in terms of two or three major types, such as electronic retailing or online shopping [21]. However, as shown in Figure 2, electronic commerce is as old as the computer and software industries themselves and predates the Internet era of the 1990s [20]. There is no standard taxonomy of electronic commerce technologies, but they do include major categories such as magnetic ink character recognition, automatic teller machines, electronic funds transfer, stock market automation, facsimiles, email, point of sale systems, Internet service providers and electronic data interchange, as well as electronic retail trade and shopping websites [20].

3.2 Second-Generation Electronic Commerce

Second-generation or transistorized computers were associated with electronic commerce technologies such as magnetic ink character recognition or MICR

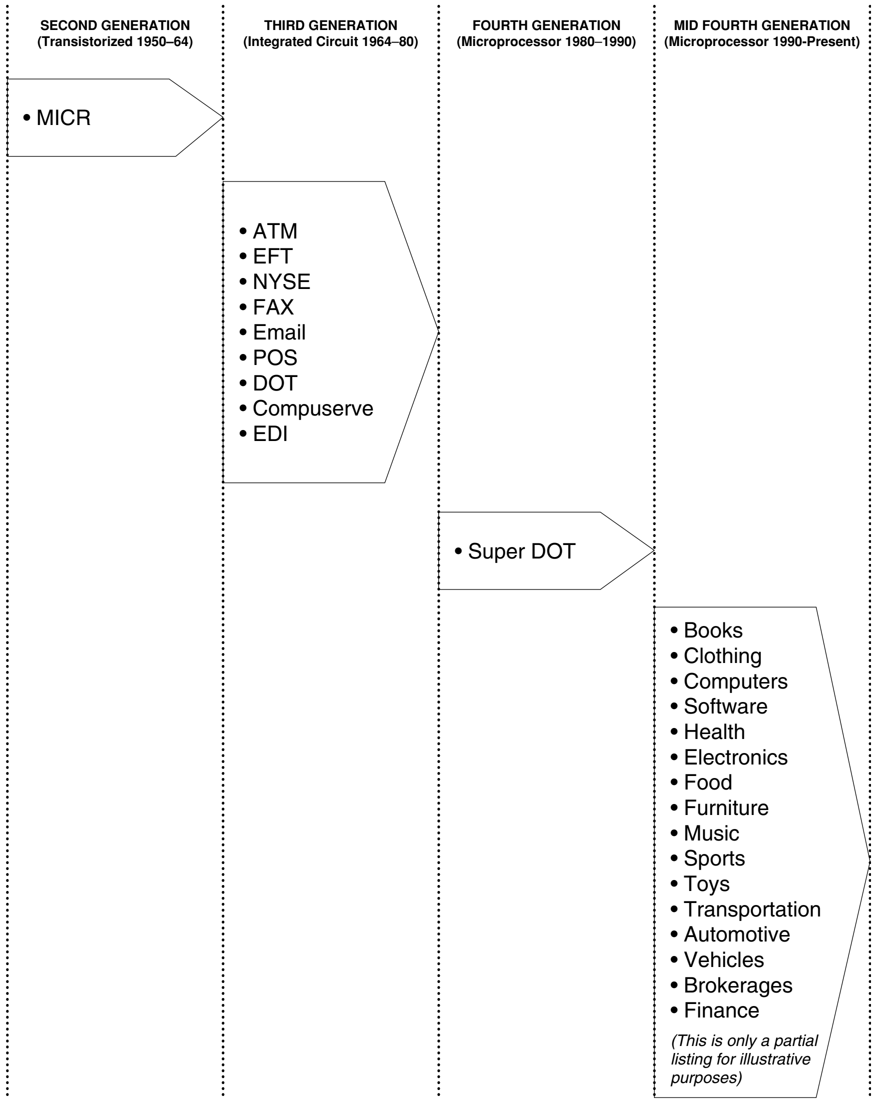

FIG. 2. Timeline and history of electronic commerce.

created in 1956, which was a method of ‘encoding checks and enabling them to be sorted and processed automatically’ [22].

3.3 Third-Generation Electronic Commerce

Third-generation or integrated-circuit-based computers were associated with electronic commerce technologies such as automatic teller machines, electronic funds transfer, stock market automation, facsimiles, email, point of sale systems, electronic bulletin boards and electronic data interchange. In 1965, automated teller machines were created [23], which were electronic machines or computers that automatically dispense money or cash [22]. In 1966, electronic funds transfer or EFT was created [24], which was ‘a set of processes that substitutes electronic messages for checks and other tangible payment mechanisms’ [22]. Also in 1966, the New York Stock Exchange or NYSE was first automated [25]. In 1971, facsimiles were created [26]. In 1973, email was created [19]. In 1975, electronic point of sale systems were created [27], which involved ‘the collection in real-time at the point of sale, and storing in a computer file, of sales and other related data by means of a number of electronic devices’ [28]. In 1976, the designated order turn-around or DOT was created, which automated small-volume individual trades [25]. In 1979, Compuserve launched one of the first electronic bulletin boards [19]. Also in 1979, electronic data interchange was created [29], which is the ‘electronic movement of information, such as payments and invoices, between buyers and sellers’ [30].

3.4 Fourth-Generation Electronic Commerce

Fourth-generation or microprocessor-based computers were associated with electronic commerce technologies such as the vast automation of the stock market. In 1984, the super designated order transfer 250 was launched to enable large-scale automatic program trading [25].

3.5 Mid-Fourth-Generation Electronic Commerce

Mid-fourth-generation computers were associated with electronic commerce technologies such as selected electronic services, electronic retail trade and electronic shopping and mail order houses. Selected electronic services consisted of industry sectors such as transportation and warehousing; information, finance, rental and leasing services; professional, scientific and technical services; administrative and support services; waste management and remediation services; health care and social assistance services; arts, entertainment and recreation services; accommodation and food services; and other services [21]. Electronic retail trade consisted of industry sectors such as motor vehicles and parts dealers; furniture and home furnishing stores; electronics and appliance stores; building materials, garden equipment and supplies stores; food and beverage stores; health and personal services; gasoline stations; clothing and accessories stores; sporting goods, hobby, book and music stores; general merchandise stores; miscellaneous store retailers; and non-store retailers [21]. And, electronic shopping and mail order houses consisted of industry sectors such as books and magazines; clothing and clothing accessories; computer hardware; computer software; drugs, health aids and beauty aids; electronics and appliances; food, beer and wine; furniture; music and videos; office equipment; sporting goods, toys, hobby goods, and games; other merchandise; and non-merchandise receipts [21].

Today, the U.S. electronic commerce industry garners revenues in excess of $2.4 trillion per year [31]. About $2.2 trillion is acquired from business-to-business or B2B commerce, also known as electronic data interchange or EDI. A good example of B2B is Wal-Mart computers, which automatically initiates an order and shipment from a wholesaler such as Proctor and Gamble when supplies run low. About $136 billion to $189 billion worth of U.S. electronic commerce comes from business-to-consumer (B2C), also known as online retail sales or Internet retailers. The best example of B2C is a consumer shopping for and ordering a textbook from Amazon. In 2007, 147 million Internet shoppers conducted 632.5 million transactions worth $136 billion to $189 billion. In total, there are 1.25 billion Internet users, the number of websites has reached 136 million and the number of web hosts has reached 470 million. Information technology, primarily in the form of the Internet contributes to more than 50% of total labour productivity growth in the top 10 industrialized nations and nearly 100% in China and India.

In 2006, the top 100 Internet retailers grew at an average rate of 19%, the bottom 100 from the top 500 grew at an average rate of 23%, startups grew at 55%, the fastest Internet retailers grew at 200%, and one firm grew at a rate of 400%. As many as 45% of U.S. Internet retailers are considered ‘pure-plays’; that is, non-brick-and-mortar retailers such as Amazon. Traditional retailers such as Wal-Mart, Sears and others lag behind pure-plays in growth, conversion rates, customer satisfaction, website satisfaction and website quality. (Conversion rates refer to the percentage of Internet shoppers who make a purchase after visiting an electronic commerce website.) It’s important to note that Internet retailing only garners about 3% to 4% of all retail sales in the U.S. That is, for every dollar spent by the Americans, only four cents is spent making online purchases. As the number of world-wide Internet users, shoppers and sales increase, this will result in unprecedented demands on the number of websites that need to be produced.

4. History of Software Methods

4.1 Database Design

One of the earliest software methods that emerged in the mainframe era of the 1960s was database design. As shown in Figures 3 and 4, database design is a process of developing the structure and organization of an information repository [32]. And, the U.S. formed a standard information resource dictionary system or IRDS [33], which is a 'logically centralized repository of data about all relevant information resources within an organization, often referred to as metadata' [34]. The use of flat files for the design of information repositories was one of the earliest forms of database design [35]. Network databases were one of the earliest forms of industrial-strength information repositories consisting of many-to-many relationships between entities or data records, examples of which include IBM's Information Management System or IMS/360 and UNIVAC's Data Management System or DMS 1100 [36]. Hierarchical databases soon emerged with a focus on organizing data into tree-like structures, which were believed to mimic the natural order of data in the real world [37]. Relational database design was introduced to create more reliable and less redundant information repositories based on the mathematical theory of sets [38].

4.2 Automatic Programming

Another of the earliest software methods that emerged in the mainframe era of the 1960s was automatic programming, which is also known as fourth-generation programming languages or 4GLs. Automatic programming is defined as the 'process and technology for obtaining an operational program and associated data structures automatically or semi-automatically, starting with only a high-level user-oriented specification of the environment and the tasks to be performed by the computer' [39]. Decision tables were one of the first automatic programming methods, which provided a simple format enabling both users and analysts to design computer software without any programming knowledge [40]. Programming questionnaires were also one of the first automatic programming methods, which provided an English-like yes or no questionnaire enabling non-computer programmers to answer a few discrete questions about their needs, leading to the automatic production of computer software [41]. The next evolution in automatic programming methods that emerged in the early midrange era was problem statement languages, characterized by the information system design and optimization system or ISDOS, which provided a means for users and other non-programmers to specify their needs and requirements and 'what' to do, without specifying 'how' the computer programs would perform their functions [42]. Special purpose languages also began emerging in the early midrange area, which

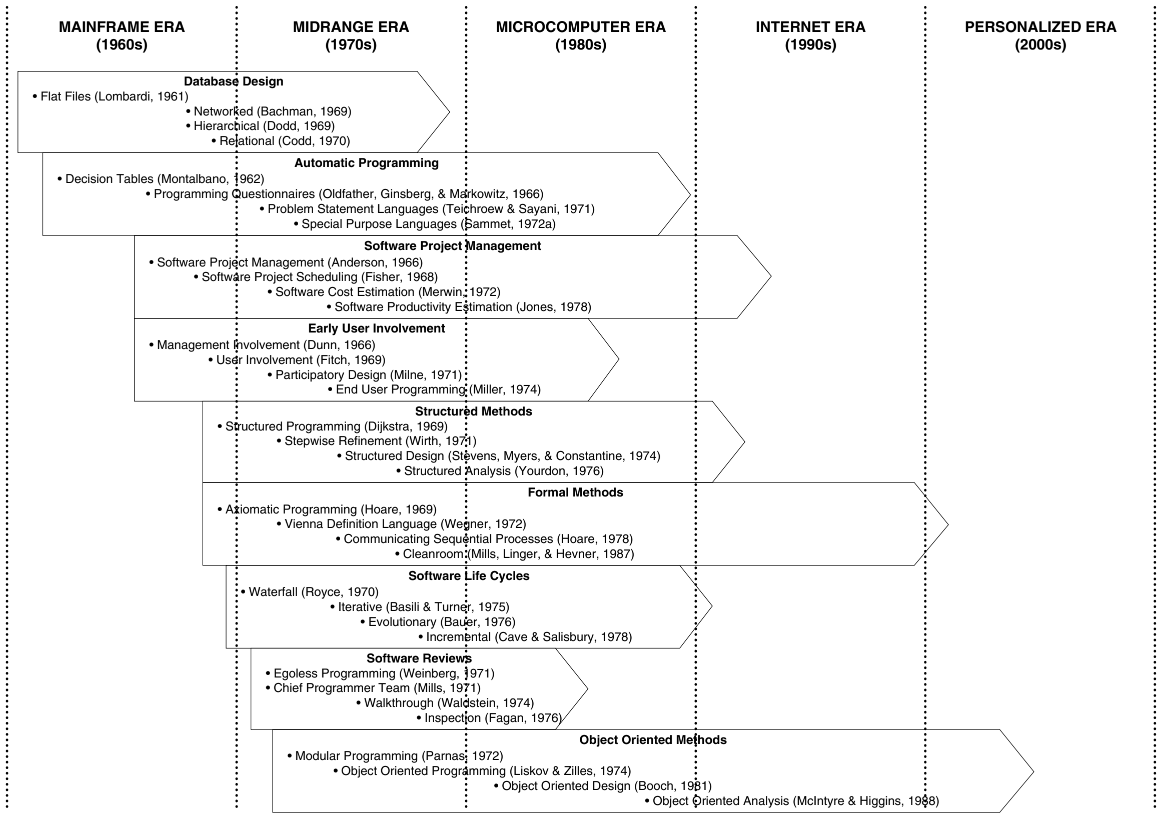
MAINFRAME ERA (1960s)
MIDRANGE ERA (1970s)
MICROCOMPUTER ERA (1980s)
INTERNET ERA (1990s)
PERSONALIZED ERA (2000s)
Database Design
• Flat Files (Lombardi, 1961)
• Networked (Bachman, 1969)
• Hierarchical (Dodd, 1969)
• Relational (Codd, 1970)
Automatic Programming
• Decision Tables (Montalbano, 1962)
• Programming Questionnaires (Oldfather, Ginsberg, & Markowitz, 1966)
• Problem Statement Languages (Teichroew & Sayani, 1971)
• Special Purpose Languages (Sammet, 1972a)
Software Project Management
• Software Project Management (Anderson, 1966)
• Software Project Scheduling (Fisher, 1968)
• Software Cost Estimation (Merwin, 1972)
• Software Productivity Estimation (Jones, 1978)
Early User Involvement
• Management Involvement (Dunn, 1966)
• User Involvement (Fitch, 1969)
• Participatory Design (Milne, 1971)
• End User Programming (Miller, 1974)
Structured Methods
• Structured Programming (Dijkstra, 1969)
• Stepwise Refinement (Wirth, 1971)
• Structured Design (Stevens, Myers, & Constantine, 1974)
• Structured Analysis (Yourdon, 1976)
Formal Methods
• Axiomatic Programming (Hoare, 1969)
• Vienna Definition Language (Wegner, 1972)
• Communicating Sequential Processes (Hoare, 1978)
• Cleanroom (Mills, Linger, & Hevner, 1987)
Software Life Cycles
• Waterfall (Royce, 1970)
• Iterative (Basili & Turner, 1975)
• Evolutionary (Bauer, 1976)
• Incremental (Cave & Salisbury, 1978)
Software Reviews
• Egoless Programming (Weinberg, 1971)
• Chief Programmer Team (Mills, 1971)
• Walkthrough (Waldstein, 1974)
• Inspection (Fagan, 1976)
Object Oriented Methods
• Modular Programming (Parnas, 1972)
• Object Oriented Programming (Liskov & Zilles, 1974)
• Object Oriented Design (Booch, 1981)
• Object Oriented Analysis (McIntyre & Higgins, 1988)

FIG. 3. Timeline and history of software methods.

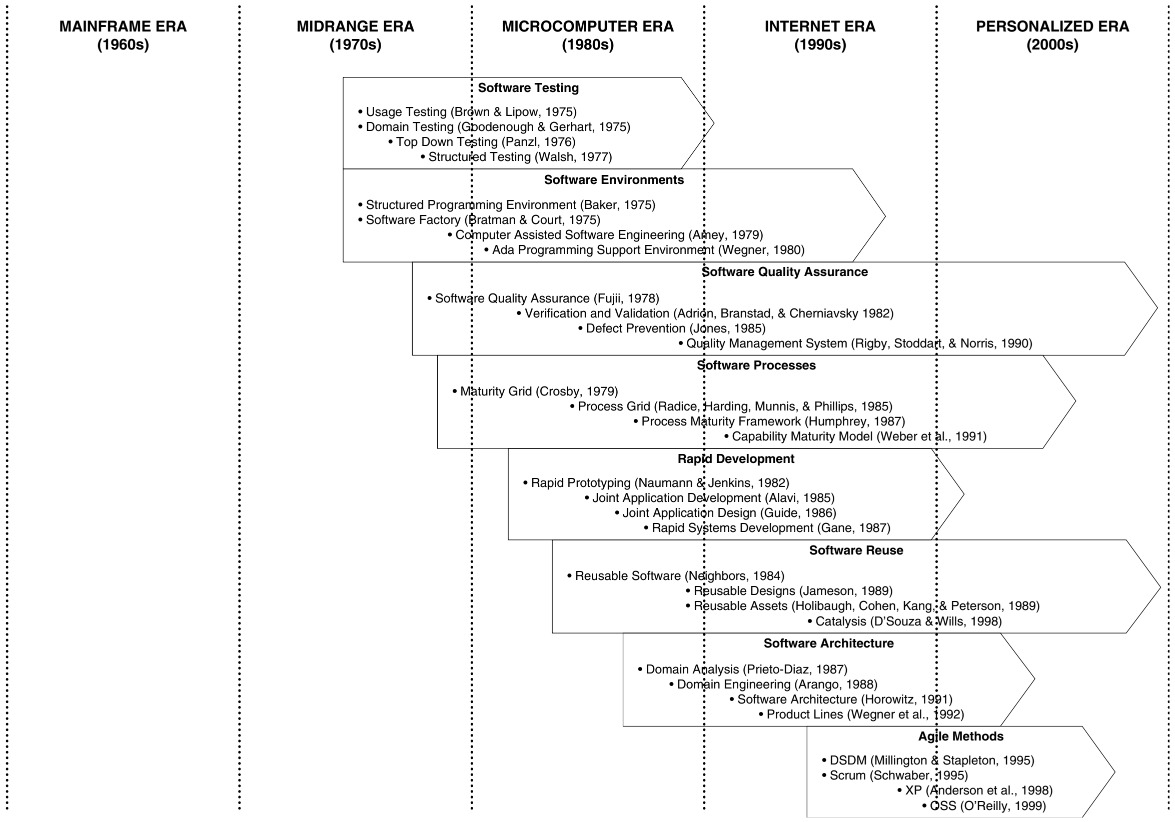
MAINFRAME ERA (1960s)
MIDRANGE ERA (1970s)
MICROCOMPUTER ERA (1980s)
INTERNET ERA (1990s)
PERSONALIZED ERA (2000s)
Software Testing
• Usage Testing (Brown & Lipow, 1975)
• Domain Testing (Goodenough & Gerhart, 1975)
• Top Down Testing (Panzl, 1976)
• Structured Testing (Walsh, 1977)
Software Environments
• Structured Programming Environment (Baker, 1975)
• Software Factory (Bratman & Court, 1975)
• Computer Assisted Software Engineering (Amey, 1979)
• Ada Programming Support Environment (Wegner, 1980)
Software Quality Assurance
• Software Quality Assurance (Fujii, 1978)
• Verification and Validation (Adrion, Branstad, & Cherniavsky 1982)
• Defect Prevention (Jones, 1985)
• Quality Management System (Rigby, Stoddart, & Norris, 1990)
Software Processes
• Maturity Grid (Crosby, 1979)
• Process Grid (Radice, Harding, Munnis, & Phillips, 1985)
• Process Maturity Framework (Humphrey, 1987)
• Capability Maturity Model (Weber et al., 1991)
Rapid Development
• Rapid Prototyping (Naumann & Jenkins, 1982)
• Joint Application Development (Alavi, 1985)
• Joint Application Design (Guide, 1986)
• Rapid Systems Development (Gane, 1987)
Software Reuse
• Reusable Software (Neighbors, 1984)
• Reusable Designs (Jameson, 1989)
• Reusable Assets (Holibaugh, Cohen, Kang, & Peterson, 1989)
• Catalysis (D'Souza & Wills, 1998)
Software Architecture
• Domain Analysis (Prieto-Diaz, 1987)
• Domain Engineering (Arango, 1988)
• Software Architecture (Horowitz, 1991)
• Product Lines (Wegner et al., 1992)
Agile Methods
• DSDM (Millington & Stapleton, 1995)
• Scrum (Schwaber, 1995)
• XP (Anderson et al., 1998)
• OSS (O'Reilly, 1999)

FIG. 4. Timeline and history of software methods (continued).

were regarded as very high level, English-like computer programming languages used for rapid prototyping and quick software composition, major examples of which include statistical analysis packages, mathematical programming packages, simplified database query languages and report generators [43].

4.3 Software Project Management

The earliest notions of software project management also emerged in the mainframe era of the 1960s. An early definition of software project management was the 'skillful integration of software technology, economics, and human relations' [44]. The project evaluation and scheduling technique or PEST was one of the first complete approaches to software project management emerging in this era [45]. Project network diagrams in the form of the program evaluation review technique or PERT and the critical path method or CPM, though not originating in computer programming, were soon applied for planning, scheduling and managing resources associated with software projects [46]. Cost-estimation techniques were soon added to the repertoire of software project management, especially for managing large U.S. military projects [47]. The framework for software project management was finally in place for years to come when basic measures of software productivity and quality were devised [48].

4.4 Early User Involvement

Early user involvement has long been recognized as a critical success factor in software projects since the earliest days of the mainframe era. Early user involvement is defined as 'participation in the system development process by representatives of the target user group' [49]. While project overruns were considered a normal part of the early computer age, scholars began calling for management participation to stem project overruns, which are now regarded as a 'management information crisis' [50]. By the late 1960s, 'user involvement' began to supplant management participation as a key for successfully designing software systems [51]. In the early 1970s, end users were asked to help design software systems themselves in what was known as 'participatory design' [52]. End user development quickly evolved from these concepts, which asked the end users to develop the applications themselves to help address the productivity paradox [53].

4.5 Structured Methods

The late mainframe period gave rise to structured methods as some of the earliest principles of software engineering to help overcome the software crisis. Structured methods are approaches for functionally decomposing software designs, e.g., expressing software designs in high-level components, which are further refined in terms of lower level components [54]. Structured programming emerged in this timeframe to help programmers create well-structured computer programs [55]. The next innovation in structured methods was called 'top down stepwise refinement', which consisted of the hierarchical design and decomposition of computer programs [56]. Structured design quickly followed suit, which is defined as 'a set of proposed general program design considerations and techniques for making coding, debugging, and modification easier, faster, and less expensive by reducing complexity' [57]. Structured analysis methods rounded out this family of methods by suggesting the use of graphs for depicting the decomposition of software functions and requirements [58].

4.6 Formal Methods

The late mainframe period also gave rise to formal methods, which would be used as the theoretical basis for software engineering for the next two decades. Formal methods are 'mathematically based languages, techniques, and tools for specifying and verifying' reliable and complex software systems [59]. Axiomatic programming is one of the first recognized formal methods, which uses mathematical set theories to design functionally correct software [60]. An entire set of formal semantics was soon devised to serve as a basis for creating mathematically correct computer programming languages called the Vienna definition language [61]. The communicating sequential processes method was then created by Tony Hoare to help design mathematically correct multi-tasking software systems [62]. The cleanroom or box-structured methodology was created to serve as a stepwise refinement and verification process for creating software designs [63]. Formal methods, primarily due the difficulty associated with their mathematical rigor, never enjoyed widespread adoption by the growing community of computer programmers [64].

4.7 Software Life Cycles

One of the first methods to come out of the early midrange era was the notion of software life cycles. A software life cycle is a 'collection of tools, techniques, and methods, which provide roles and guidelines for ordering and controlling the actions and decisions of project participants' [65]. The waterfall is one of the first recognized software life cycles consisting of seven stages: system requirements, software requirements, analysis, program design, coding, testing and operations [66] as popularized by Barry Boehm. The iterative software lifecycle appeared around the middle of the decade, which consisted of using a planned sequence of programming enhancements until computer software was complete [67]. The evolutionary software life cycle soon formed with notions of gradually enhancing computer programs rather than developing them in phases or iterations [68]. The incremental software life cycle followed next, recommending project assessments at each major milestone in order to identify and reduce risk [69]. The spiral model called for risk analysis between major milestones and prototypes as well [70].

4.8 Software Reviews

Software reviews emerged as a methodology in the very earliest stages of the midrange era. Software reviews are meetings held by groups of software engineers to review software products to identify and remove their defects [73]. Egoless programming was introduced in the early midrange era as a method of transforming software development from an individual craft into a loosely structured group activity [74]. Chief programmer teams emerged shortly thereafter to formalize the notion of egoless programming with one small difference; the team would have a clear leader [75]. Structured walkthroughs were quickly formed to once again place the responsibility for maintaining the overall program quality in the hands of the team, rather than a in the hands of a single individual [76]. Software inspections crystallized the concept of structured walkthroughs with a rigid meeting protocol for group reviews in order to optimize team performance [77]. In the same year, the U.S. military formed a standard with system design reviews, software specification reviews, preliminary design reviews, critical design reviews, test readiness reviews, functional configuration audits, physical configuration audits, formal qualification reviews and production readiness reviews [78].

4.9 Object-Oriented Methods

Object-oriented methods emerged in the midrange era as a direct response to calls for a software engineering discipline to mitigate the software crisis. Object-oriented methods are processes that 'allow a designer to generate an initial design model in the context of the problem itself, rather than requiring a mapping onto traditional computer science constructs' [79]. The Simula programming language actually emerged in the late 1960s, which was recognized by some as having modular and object-oriented programming features and capabilities [80]. Smalltalk soon followed as an offshoot of the flex programming language and reactive engine, which also had many early modular and object-oriented programming features [81]. However, the principles of modular and object-oriented programming – information hiding, self-contained data structures, co-located subroutines, and well-defined interfaces – were first formalized just after the emergence of Simula and Smalltalk programming languages, the principles of which were claimed to improve efficiency, flexibility and maintainability [84].

Object-oriented design emerged in the early microcomputer era to demonstrate one of the first graphical notations for describing object-oriented programming languages [85]. Finally, object-oriented analysis methods emerged, often reusing the tools of structured analysis to begin constructing specifications of software systems prior to devising their object oriented design [86].

4.10 Software Testing

Software testing gained recognition in the middle of the midrange era though system and hardware component testing had been the norm for at least two decades. Software testing is defined as 'the process of executing a software system to determine whether it matches its specification and executes in its intended environment' [87]. Software usage testing was developed based on the notion that software reliability could be improved by specifying how computer programs will be used, devising tests to model how users operate programs and then measuring the outcome of the testing [88]. Domain testing emerged at the same time with its principles of identifying test cases from program requirements, specifying a complete set of inputs using mathematical set theory and using set theory itself to prove program correctness when necessary [89]. Soon thereafter, top down testing was introduced, which recommended a unique test procedure for each software subroutine [90]. Finally, structured testing emerged as an approach to encapsulate best practices in software testing for novice computer programmers [91].

4.11 Software Environments

Software environments emerged in the middle of the midrange era as a means of improving software quality and productivity through automation. A software environment may be described as an 'operating system environment and a collection of tools or subroutines' [92]. A slightly better definition of software environment is a 'coordinated collection of software tools organized to support some approach to software development or conform to some software process model' [93], where software tools are defined as 'computer programs that assist engineers with the design and development of computer-based systems' [94]. Structured programming environments were created as a means of improving software reliability and productivity using guidelines, code libraries, structured coding, top down development, chief programmer teams, standards, procedures, documentation, education and metrics [95]. Software factories were soon created to introduce discipline and repeatability, software visualization tools, the capture of customer needs or requirements, automated software testing and software reuse [96]. Computer-assisted software engineering or CASE was also created to enhance software productivity and reliability by automating document production, diagram design, code compilation, software testing, configuration management, management reporting and sharing of data by multiple developers [97]. The Ada programming support environment or APSE was suggested as a core set of programming tools consisting of editors, compilers, debuggers, linkers, command languages and configuration management utilities [98]. Computer-aided software engineering was created to automate the tasks of documenting customer requirements, creating software architectures and designs, maintaining requirements traceability and configuration management [99]. Integrated computer-aided software engineering or I-CASE tools emerged, merging analysis and code generation tools [100]. However, I-CASE was a concept readily adopted by the U.S. DoD that was soon abandoned since it was never delivered and never worked well [101].

4.12 Software Quality Assurance

The modern day tenets of software quality assurance began to assume their current form in the late midrange era. Software quality assurance is defined as a 'planned and systematic pattern of all actions necessary to provide adequate confidence that the software conforms to established technical requirements' [102]. Software quality assurance was created to establish 'adherence to coding standards and conventions, compliance with documentation requirements and standards, and successful completion of activities' [103]. Software verification and validation was created to determine the adequacy of software requirements, software designs, software source code and regression testing during software maintenance [104]. Defect prevention was a structured process of determining the root causes of software defects and then institutionalizing measures to prevent their recurrence [105]. Quality management systems consisted of a set of organizational policies and procedures to ensure that the software satisfied its requirements [106].

4.13 Software Processes

Software processes were formed in the microcomputer era, though they were rooted in the traditions of software engineering, structured methods, software life cycles and software environments dating back to the late mainframe and early midrange eras. A software process is the 'collection of related activities seen as a coherent process subject to reasoning involved in the production of a software system' [107]. The maturity grid (e.g., uncertainty, awakening, enlightenment, wisdom and certainty), though not for software, inspired the software process modelling movement of the microcomputer era [108]. IBM then created its own process grid (e.g., traditional, awareness, knowledge, skill and wisdom and integrated management system) for conducting site studies of computer programming laboratories [109]. The process maturity framework was directly adapted from IBM's process grid [110], which was finally turned into the capability maturity model for U.S. military use [111].

4.14 Rapid Development

Rapid development was formalized in the microcomputer era though its tenets can be traced back to early notions of structured methods and prototyping. Rapid development is defined as a 'methodology and class of tools for speedy object development, graphical user interfaces, and reusable code for client-server applications' [112]. Rapid development leveraged productivity-enhancing technologies to build early models of information systems and involved end users in the definition of system requirements and designs [112]. Rapid development sought to control cost and schedule overruns; improve user acceptance, customer satisfaction, system quality and system success; and ultimately reduce information systems backlogs [112]. Rapid prototyping was defined as a process of quickly creating an informal model of a software system, soliciting user feedback and then evolving the model until it satisfied the complete set of customer requirements [113]. Joint application development was a process of having professional software developers assist end users with the development of their applications by evaluating their prototypes [114]. Joint application design, on the other hand, was a structured meeting between end users and software developers, with the objective of developing software designs that satisfied their needs [115]. Rapid systems development was an extension of the joint application design method, which advocated specific technological solutions such as relational databases and the completion of the software system, not just its design [116]. In a close adaptation, rapid application development recommended iterative rapid system development cycles in 60- to 120-day intervals [117].

4.15 Software Reuse

Software reuse assumed its current form in the early microcomputer era, though its earliest tenets can clearly be seen in literature throughout the 1950s, 1960s and 1970s. Software reuse is the 'use of existing software or software knowledge to construct new software' [118]. Software reuse was proposed as early as 1968 in order to help alleviate the 'software crisis' characterized by an explosion in computers and software complexity through the production of mass-produced software components [119]. The purpose of software reuse has evolved over the years to include improvements in productivity [120], reliability [121], quality [122] and cost efficiency [123]. Reusable software became synonymous with the Ada programming language in the

1980s though it was prophesied as a major strategy in 1968 and was a central management facet of Japanese software factories in the 1970s [124]. Toward the end of the microcomputer era, reusable software designs were considered just as important as reusable software source code [125]. In the same year, reusability was expanded to include requirements, designs, code, tests, and documents and dubbed 'reusable assets' [126]. Toward the end of the Internet era, catalysis was formed based on composing new applications from existing ones [127]. Though several studies chronicled software reuse success stories from the 1970s, 1980s and 1990s, [9, 128], scholars have concluded that software reuse has only had marginal success since 1968 [131].

4.16 Software Architecture

Software architecture began to assume a strategic role for managing the development of software systems near the end of the microcomputer era. Software architecture is defined as 'the structure and organization by which modern system components and subsystems interact to form systems and the properties of systems that can best be designed and analysed at the system level' [133]. Domain analysis was the discipline of identifying, capturing and organizing all of the information necessary to create a new software system [134]. Domain engineering was a process of managing reusable information about specific types of software systems, gathering architectural data and gathering data about the computer programs themselves [135]. Software architecture was a discipline of creating flexible software designs that were adaptable to multiple computer systems in order to respond to the rapidly changing military threats [136]. Software product lines soon emerged with an emphasis on evolving software architectures to reduce costs and risks associated with changes in design [137], along with software product families [138].

4.17 Agile Methods

Agile methods gained prominence in the late Internet and early personalized eras in part to accommodate the uniquely flexible nature of Internet technologies [139]. More to the point, the use of agile methods for Internet software was a reaction to the rise of traditional software development methods, which were too cumbersome, expensive, rigid and fraught with failure [139]. Downsizing was the norm and traditional methods were being used by large corporations in decline [139]. Agile methods are an approach for managing the development of software, which are based upon obtaining early customer feedback on a large number of frequent software releases [140]. In 2001, the 'agile manifesto' was created to outline the values and principles of agile methods and how they differed from traditional ones [141]. A council of 17 experts in agile methods met in order to find an ‘alternative to documentation-driven, heavyweight software development processes’. They believed that ‘in order to succeed in the new economy, to move aggressively into the era of e-business, e-commerce, and the web, companies have to rid themselves of their Dilbert manifestations of make-work and arcane policies’. Once the ground rules and assumptions of agile methods were established, they were able to get on with the business of writing the agile manifesto itself and publish it on the Internet. The agile manifesto began with the following statement: ‘we are uncovering better ways of developing software by doing it and helping others do it’. Then the agile manifesto laid out four broad values: (a) ‘working software over comprehensive documentation’, (b) ‘customer collaboration over contract negotiation’, (c) ‘individuals and interactions over processes and tools’, and (d) ‘responding to change over following a plan’. The agile manifesto itself was derived [139] from the dynamic system development methodology [142], scrum [143], extreme programming [144], open-source software development [145], crystal methods [146], feature-driven development [147], rational unified process [148], adaptive software development [149] and lean development [150]. Other approaches also influenced the development of agile methods such as the new product development game, new-product development rhythm, synch-n-stabilize, judo strategy and Internet time, which will be described later [139].

The dynamic system-development methodology or DSDM has three broad phases, which consist of requirement prototypes, design prototypes and an implementation or production phase [142]. Scrum is a light-weight software-development process consisting of implementing a small number of customer requirements in two- to four-week sprint cycles [143]. Extreme programming or XP consists of collecting informal requirements from on-site customers, organizing teams of pair programmers, developing simple designs, conducting rigorous unit testing, and delivering small and simple software packages in short two-week intervals [144]. Open-source software development involves freely sharing, peer reviewing, and rapidly evolving software source code for the purpose of increasing its quality and reliability [145]. Crystal methods involve frequent delivery; reflective improvement; close communication; personal safety; focus; easy access to expert users; and a technical environment with automated testing, configuration management and frequent integration [146]. Feature-driven development involves developing an overall model, building a features list, planning by feature, designing by feature and building by feature [147]. The rational unified process involves a project management, business modelling, requirements, analysis and design, implementation, test, configuration management, environment and deployment workflow [148]. Adaptive software development involves product initiation, adaptive cycle planning, concurrent feature development, quality review and final quality assurance and release [149]. And, lean development involves eliminating waste, amplifying the learning process, making decisions as late as possible, delivering products as fast as possible, empowering teams, building integrity in, and seeing systems as a whole [150]. Agile methods such as extreme programming adapted customer feedback, iterative development, well-structured teams, flexibility from the new product development game, new product development rhythm, synch-n-stabilize, judo strategy and Internet time [139].

5. History of Software Quality Measurement

5.1 Software Size

One of the earliest known measures used to describe computer programs was software size [151]. Software size is a measure of the volume, length, quantity, amount and overall magnitude of a computer program [152]. In the mid 1960s, lines of code or LOC was one of the first known measures of software size, which referred to the number of computer instructions or source statements comprising a computer program and is usually expressed as thousands of lines of code [153]. Almost a decade later in the mid to late 1970s, more sophisticated measures of software size emerged such as token count, volume, function count and function points [152]. Recognizing that individual lines of code had variable lengths, token count was created to distinguish between unequally sized lines of code, which was technically defined as 'basic syntactic units distinguishable by a compiler' [154]. In yet another attempt to accurately gauge the size of an individual line of code, volume was created to measure the actual size of a line of code in bits, otherwise known as binary zeros and ones [154]. Shortly thereafter, function count was created to measure software size in terms of the number of modules or subroutines [155]. Function points was another major measure of software size, which was based on estimating the number of inputs, outputs, master files, inquiries and interfaces [156]. Though software size is not a measure of software quality itself, it formed the basis of many measures or ratios of software quality right on through the modern era (e.g., number of defects, faults, or failures per line of code or function point). Furthermore, some treatises on software metrics consider software size to be one of the most basic measures of software complexity [157]. Thus, a history of software quality measurement may not be complete without the introduction of an elementary discussion of software size.

5.2 Software Errors

One of the earliest approaches for measuring software quality was the practice of counting software errors dating back to the 1950s when digital computers emerged. Software errors are human actions resulting in defects, defects sometimes manifest

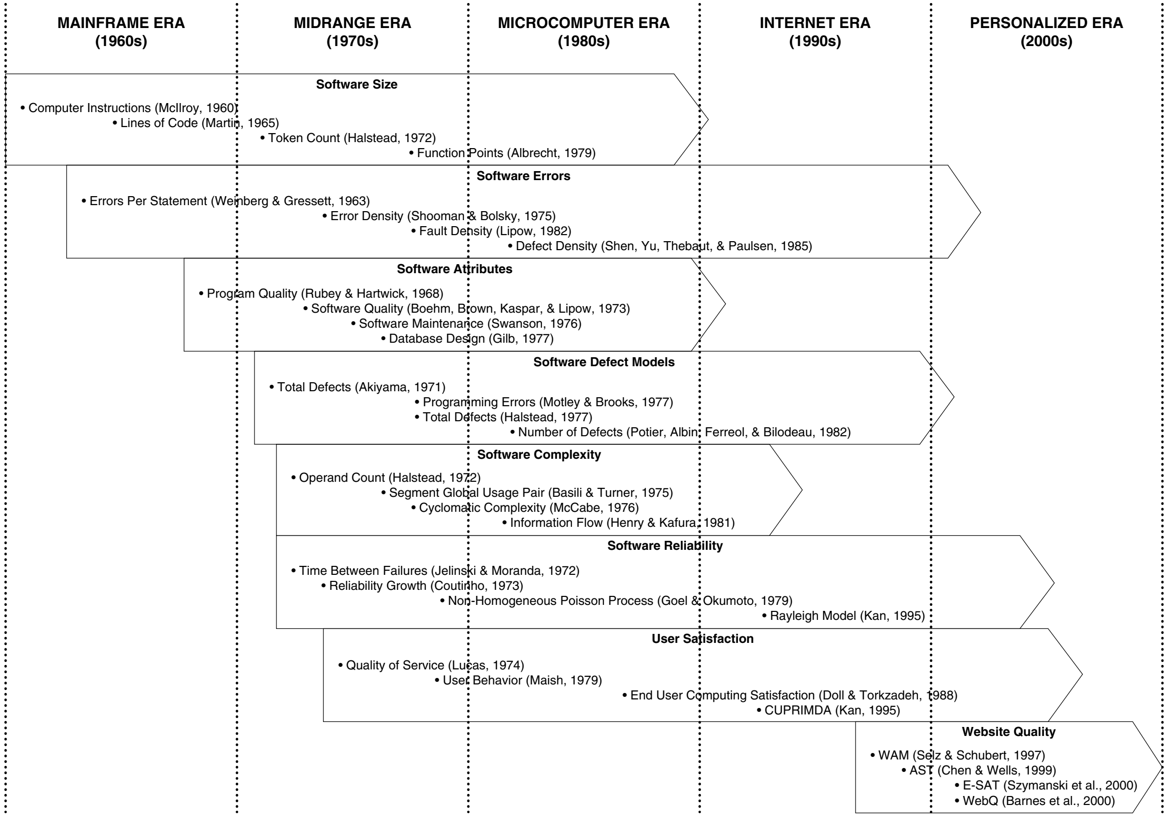
MAINFRAME ERA (1960s)
MIDRANGE ERA (1970s)
MICROCOMPUTER ERA (1980s)
INTERNET ERA (1990s)
PERSONALIZED ERA (2000s)
Software Size
• Computer Instructions (McIlroy, 1960)
• Lines of Code (Martin, 1965)
• Token Count (Halstead, 1972)
• Function Points (Albrecht, 1979)
Software Errors
• Errors Per Statement (Weinberg & Gressett, 1963)
• Error Density (Shooman & Bolsky, 1975)
• Fault Density (Lipow, 1982)
• Defect Density (Shen, Yu, Thebaut, & Paulsen, 1985)
Software Attributes
• Program Quality (Rubey & Hartwick, 1968)
• Software Quality (Boehm, Brown, Kaspar, & Lipow, 1973)
• Software Maintenance (Swanson, 1976)
• Database Design (Gilb, 1977)
Software Defect Models
• Total Defects (Akiyama, 1971)
• Programming Errors (Motley & Brooks, 1977)
• Total Defects (Halstead, 1977)
• Number of Defects (Potier, Albin, Ferreol, & Bilodeau, 1982)
Software Complexity
• Operand Count (Halstead, 1972)
• Segment Global Usage Pair (Basili & Turner, 1975)
• Cyclomatic Complexity (McCabe, 1976)
• Information Flow (Henry & Kafura, 1981)
Software Reliability
• Time Between Failures (Jelinski & Moranda, 1972)
• Reliability Growth (Coutinho, 1973)
• Non-Homogeneous Poisson Process (Goel & Okumoto, 1979)
• Rayleigh Model (Kan, 1995)
User Satisfaction
• Quality of Service (Lucas, 1974)
• User Behavior (Maish, 1979)
• End User Computing Satisfaction (Doll & Torkzadeh, 1988)
• CUPRIMDA (Kan, 1995)
Website Quality
• WAM (Selz & Schubert, 1997)
• AST (Chen & Wells, 1999)
• E-SAT (Szymanski et al., 2000)
• WebQ (Barnes et al., 2000)

FIG. 5. Timeline and history of software quality measures.

themselves as faults, and faults lead to failures, which are often referred to as software crashes [157]. The concept of 'errors per statement' first appeared in the early 1960s [158] and studies of 'error proneness' intensified towards the end of the decade [159]. The term error density was coined in the mid 1970s, which referred to the simple ratio of errors to software size [160]. Fault density was also a measure of software quality, which referred to the ratio of anomaly-causing faults to software size [161]. The term defect density subsumed the measure of error and fault density in the mid 1980s, which referred to the ratio of software errors to software size [162]. Many unique types of errors were counted, such as number of requirement, design, coding, testing and maintenance errors, along with number of changes and number of changed lines of code [152]. Even the term problem density emerged in the early 1990s, which referred to the number of problems encountered by customers to measure and track software quality [157]. The practice of counting errors, defects, faults and failures as a means of measuring software quality enjoyed widespread popularity for more than five decades.

5.3 Software Attributes

Another of the earliest approaches for measuring software quality was the practice of quantifying and assessing attributes or characteristics of computer programs. Software attributes are an 'inherent, possibly accidental trait, quality or property' such as functionality, performance or usability [163]. Logicon designed a model to measure software attributes such as correctness, logicality, non-interference, optimizability, intelligibility, modifiability and usability [164]. Next, TRW identified software attributes such as portability, reliability, efficiency, modifiability, testability, human engineering and understandability [165]. These early works led to numerous specialized spin-offs such as a framework for measuring the attributes of software maintenance [166] and even a database's design [167]. Spin-offs continued to emerge with an increasing focus on operationalizing these attributes with real software metrics [168]. By the mid 1980s, this practice reached Japan [171] and a comprehensive framework emerged replete with detailed software measures [172]. Use of software attributes to measure software quality was exemplified by the functionality, usability, reliability, performance, and supportability, which was named the FURPS model [173]. Software attributes enjoyed widespread use among practitioners throughout the 1970s and 1980s because of their simplicity, though scientists favoured statistical models.

5.4 Static Defect Models

One of the earliest approaches for predicting software quality was the use of statistical models referred to as static reliability or static software defect models.

'A static model uses other attributes of the project or program modules to estimate the number of defects in software' [157], ignoring 'rate of change' [152]. One of the earliest software defect models predicted the number of defects in a computer program as a function of size, decision count or number of subroutine calls [174]. Multi-linear models were created with up to 10 inputs for the various types of statements found in software code such as comments, data and executable instructions [175]. The theory of software science was extended to include defect models by using volume as an input, which itself was a function of program language and statement length [154]. Research on software defect models continued with more extensions based on software science, cyclomatic complexity, path and reachability metrics [176]. More defect models were created by mixing defects, problems and software science measures such as vocabulary, length, volume, difficulty and effort [162]. Later, IBM developed models for predicting problems, fielded defects, arrival rate of problems and backlog projection, which were used to design midrange operating systems [157]. Static linear or multi-linear statistical models to predict defects continue to be useful tools well into modern times, though older dynamic statistical reliability models are overtaking them.

5.5 Software Complexity

With its emergence in the early 1970s, the study of software complexity became one of the most common approaches for measuring the quality of computer programs. Software complexity is defined as 'looking into the internal dynamics of the design and code of software from the program or module level to provide clues about program quality' [157]. Software complexity sprang from fervor among research scientists eager to transform computer programming from an art into a mathematically based engineering discipline [177]. Many technological breakthroughs in the two decades prior to mid 1970s led to the formation of software complexity measures. These included the advent of digital computers in the 1950s, discovery of high-level computer programming languages and the formation of compiler theory. Furthermore, flowcharting was routinely automated, axiomatic theorems were used for designing new computer languages and analysis of numerical computer algorithms became commonplace. As a result, three major classes of software complexity metrics arose for measuring the quality of software: (a) data structure, (b) logic structure and (c) composite metrics [178]. One of the first data structure metrics was the count of operands, which measured the number of variables, constants and labels in a computer program versus measuring logic [177]. The segment-global-usage-pair metric determined complexity by counting references to global variables, a high number of which was considered bad among coders [67]. Another data structure metric was the span between variables, which measured how many logic structure statements existed between variables where a higher number was poor [179]. A unique data structure metric for measuring software quality was the number of live variables within a procedure or subroutine as a sign of undue complexity [180]. One data structure metric surviving to modern times is the information flow, or fan in-fan out metric, which measures the number of modules that exchange data [181]. Logic structure metrics were cyclomatic complexity or paths [182], minimum paths [183] and gotos or knots [184]. Also included were nesting [185], reachability [186], nest depth [187] and decisions [162]. Composite metrics combined cyclomatic complexity with other attributes of computer programs to achieve an accurate estimate of software quality [188]. They also included system complexity [191] and syntactic construct [192]. Finally, it's important to note that most complexity metrics are now defunct, though cyclomatic complexity, which arose out of this era, is still used as a measure of software quality today.

5.6 Software Reliability

Software reliability emerged in the early 1970s and was created to predict the number of defects or faults in software as a method of measuring software quality. Software reliability is the 'probability that the software will execute for a particular period of time without failure, weighted by the cost to the user of each failure encountered' [193]. Major types of reliability models include: (a) finite versus infinite failure models [194], (b) static versus dynamic [152] and (c) deterministic versus probabilistic [195]. Major types of dynamic reliability models include: life cycle versus reliability growth [157] and failure rate, curve fitting, reliability growth, non-homogeneous Poisson process and Markov structure [195]. One of the first and most basic failure rate models estimated the mean time between failures [196]. A slightly more sophisticated failure rate model was created based on the notion that software became more reliable with the repair of each successive code failure [197]. The next failure rate model assumed that the failure rate was initially constant and then began to decrease [198]. Multiple failure rate models appeared throughout the 1970s to round out this family of reliability models [199]. Reliability or 'exponential' growth models followed the emergence of failure rate models, which measured the reliability of computer programs during testing as a function of time or the number of tests [202, 203]. Another major family of reliability models is the non-homogeneous Poisson process models, which estimate the mean number of cumulative failures up to a certain point in time [205]. Reliability models estimate the number of software failures after development based on failures encountered during testing and operation. Though rarely mentioned, the Rayleigh life cycle reliability model accurately estimates defects inserted and removed throughout the software lifecycle [157]. Some researchers believed that the use of software reliability models offered the best hope for transforming computer programming from a craft industry into a true engineering discipline.

5.7 User Satisfaction

User satisfaction gradually became a measure of software quality during the 1950s, 1960s and 1970s [208] User satisfaction is defined as 'the sum of one's feelings or attitudes toward a variety of factors affecting that situation', e.g., computer use and adoption by end users [212]. Though not the first, one study of user satisfaction analysed attitudes towards quality of service, management support, user participation, communication and computer potential [213]. A more complex study of user satisfaction considered the feelings about staff, management support, preparation for its use, access to system, usefulness, ease of use and flexibility [214]. Most studies until 1980 focused on the end user's satisfaction with regards to software developers; but one study squarely focused on the end user's satisfaction with regards to the software itself [215]. One of the first studies to address a variety of software attributes such as software accuracy, timeliness, precision, reliability, currency and flexibility appeared [216]. Studies throughout the 1980s addressed user satisfaction with both designers and software [212, 217]. The late 1980s marked a turning point, with studies focusing entirely on user satisfaction with the software itself and attributes such as content, accuracy, format, ease of use and timeliness of the software [221]. A study of user satisfaction at IBM was based on reliability, capability, usability, installability, maintainability, performance and documentation factors [222]. Throughout the 1990s, IBM used a family of user satisfaction models called UPRIMD, UPRIMDA, CUPRIMDA and CUPRIMDSO, which referred differently to factors of capability, usability, performance, reliability, installability, maintainability, documentation, availability, service and overall satisfaction [157]. User satisfaction, now commonly referred to as customer satisfaction, is undoubtedly related to earlier measures of software attributes, usability or user friendliness of software and more recently web quality.

5.8 Website Quality

With their emergence in the late 1990s, following the user satisfaction movement, models of website quality appeared as important measures of software quality [223]. One of the first models of website quality identified background, image size, sound file display and celebrity endorsement as important factors of software quality [224]. The web assessment method or WAM quickly followed with quality factors of external bundling, generic services, customer-specific services and emotional experience [225]. In what promised to be the most prominent web quality model, attitude towards the site or AST had quality factors of entertainment, informativeness, and organization [226]. The next major model was the e-satisfaction model with its five factors of convenience, product offerings, product information, website design and financial security [227]. The website quality model or WebQual for business school portals was based on factors of ease of use, experience, information and communication and integration [228]. An adaptation of the service quality or ServQual model, WebQual 2.0 measured quality factors such as tangibles, reliability, responsiveness, assurance and empathy [229]. The electronic commerce user consumer satisfaction index or ECUSI consisted of 10 factors such as product information, consumer service, purchase result and delivery, site design, purchasing process, product merchandising, delivery time and charge, payment methods, ease of use and additional information services [230]. On the basis of nine factors, the website quality or SiteQual model consisted of aesthetic design, competitive value, ease of use, clarity of ordering, corporate and brand equity, security, processing speed, product uniqueness and product quality assurance [231]. In what promised to be exclusively for websites, the Internet retail service quality or IRSQ model was based on nine factors of performance, access, security, sensation, information, satisfaction, word of mouth, likelihood of future purchases and likelihood of complaining [232]. In a rather complex approach, the expectation-disconfirmation effects on web-customer satisfaction or EDEWS model consists of three broad factors (e.g., information quality, system quality and web satisfaction) and nine sub-factors [233]. In one of the smallest and most reliable website quality models to date, the electronic commerce retail quality or EtailQ model consists of only four major factors (e.g., fulfillment and reliability, website design, privacy and security, and customer service) and only 14 instrument items [234]. On the basis of techniques for measuring software quality dating back to the late 1960s, more data have been collected and validated using models of website quality than any other measure.

6. History of Agile Methods

6.1 New Product Development Game

As shown in Figure 6, two management scholars from the School of International Corporate Strategy at Hitotsubashi University in Tokyo, Japan, published a management approach called the 'new product development game' in the Harvard Business Review in early 1986 [235]. In their article, they argued that Japanese 'companies are increasingly realizing that the old sequential approach to developing new products simply will not get the job done'. They cited the sport of Rugby as the inspiration for the principles of their new product development game – In particular, Rugby's

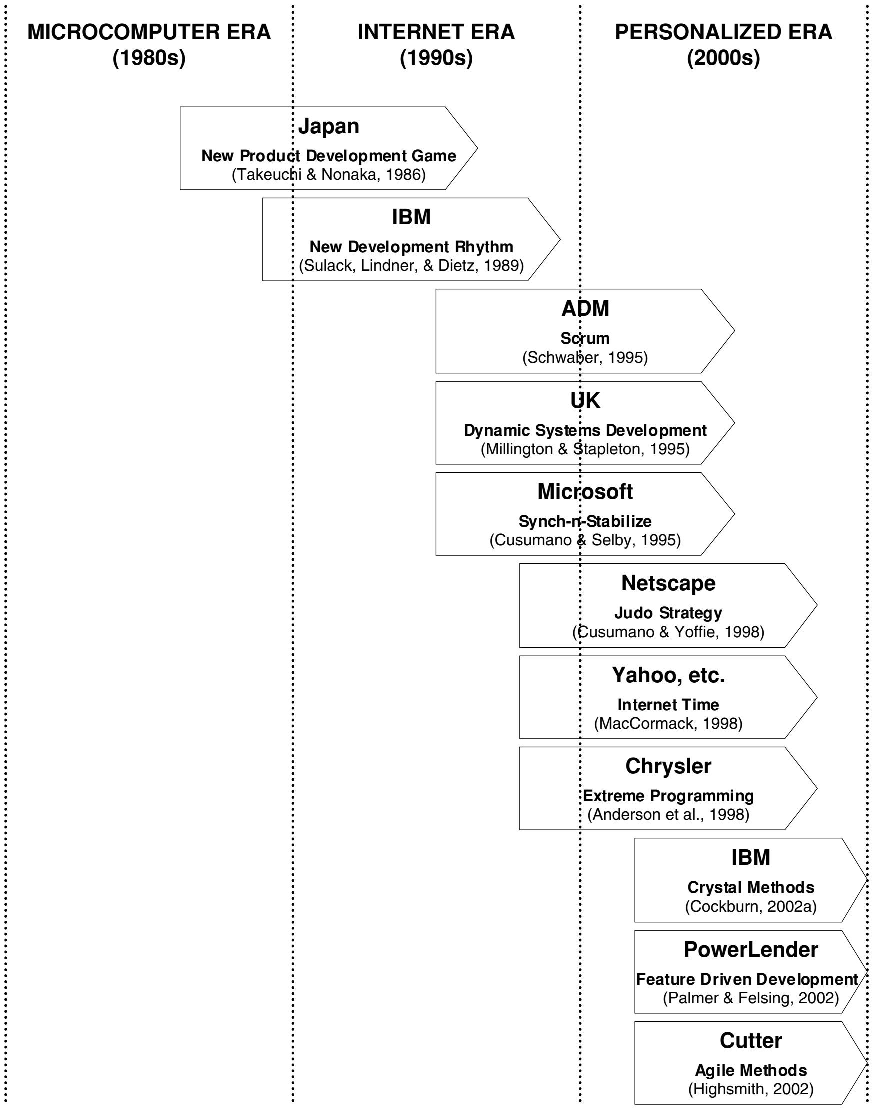

FIG. 6. Timeline and history of agile methods.

special play called the Scrum, when the players interlock themselves together as a tightly bound group to gain possession of the ball. The new product development game consisted of six major factors: (a) built-in instability, (b) self-organizing project teams, (c) overlapping development phases, (d) multi-learning, (e) subtle control and (f) organizational transfer of learning. They went on to demonstrate how four Japanese firms, e.g., Fuji-Xerox, Canon, Honda and NEC, applied the six factors of the new product development game to develop six major products, which became market successes. The six major factors of the new product development game were not unlike the total quality management and concurrent engineering movements that were popular in the U.S. during that timeframe, and their work inspired the development of agile methods for the next 20 years.

6.2 New Development Rhythm

In 1989, three managers from IBM in Rochester, Minnesota, published an article on how IBM devised a management approach called the 'new development rhythm', to bring the AS/400 midrange computer to market in only two years [236]. In their article, they stated that 'user involvement programs yielded a product offering that met the user requirements with a significantly reduced development cycle'. The new development rhythm consisted of six major factors: (a) modularized software designs, (b) software reuse, (c) rigorous software reviews and software testing, (d) iterative development, (e) overlapped software releases and (f) early user involvement and feedback. IBM's new development rhythm was a remarkable feat of management science and boasted a long list of accomplishments: (a) time-to-market improvement of 40%, (b) development of seven million lines of operating system software in 26 months, (c) compatibility with 30 billion lines of commercial applications, (d) $14 billion in revenues and (e) the IBM corporation's first Malcolm Baldrige National Quality Award. While there was nothing innovative about IBM's new development rhythm, it was IBM's audacity to apply these academic textbook approaches to commercial product development that was unique.

6.3 Scrum

In 1993, Jeff Sutherland of the Easel Corporation adapted the principles from the 'new product development game' [235] to the field of computer programming management, explicitly calling it 'scrum' [143]. In particular, scrum assumes that the 'systems development process is an unpredictable and complicated process that can only be roughly described as an overall progression'. Furthermore, scrum's creators believed 'the stated philosophy that systems development is a well-understood approach that can be planned, estimated, and successfully completed has proven incorrect in practice'. Therefore, scrum's creators set out to define a process as a 'loose set of activities that combines known, workable tools and techniques with the best that a development team can devise to build systems'. Today, scrum is composed of three broad phases: (a) pre-sprint planning, (b) sprint and (c) post-sprint meeting. During the pre-sprint planning phase, computer programmers gather to prioritize customer needs. During the sprint phase, computer programmers do whatever it takes to complete a working version of software that meets a small set of high-priority customer needs. Finally, during the post-sprint meeting, computer programmers demonstrate working software to their customers, adjust their priorities and repeat the cycle.

6.4 Dynamic Systems Development Method

In 1993, 16 academic and industry organizations in the United Kingdom banded together to create a management approach for commercial software called the 'dynamic systems development method' or simply DSDM [142]. Their goal was to 'develop and continuously evolve a public domain method for rapid application development' in an era dominated by proprietary methods. Initially, DSDM emphasized three success factors: (a) 'the end user community must have a committed senior staff that allows developers easy access to end users', (b) 'the development team must be stable and have well established skills' and (c) 'the application area must be commercial with flexible initial requirements and a clearly defined user group'. These success factors would later be expanded to include functionality versus quality, product versus process, rigorous configuration management, a focus on business objectives, rigorous software testing, risk management and flexible software requirements. DSDM consists of five major stages: (a) feasibility study, (b) business study, (c) functional model iteration, (d) design and build iteration and (e) implementation. The goal of DSDM is to explore customer requirements by building at least two full-scale prototypes before the final system is implemented.

6.5 Synch-N-Stabilize

In 1995, management scholars from MIT's Sloan School of Management and the University of California at Irvine published a textbook on how Microsoft managed the development of software for personal computers, dubbed as the 'sync-n-stabilize' approach [237]. The scholars were experts on software management approaches for the mainframe market and their two-year case study from 1993 to 1995 was a grounded theory or emergent research design, which led them to some startling conclusions. At one point in their textbook, they stated that 'during this initial research, it became clear why Microsoft was able to remain on top in its industry while most contemporaries from the founding years of the 1970s disappeared'. The synch-n-stabilize approach consisted of six major factors: (a) parallel programming and testing, (b) flexible software requirements, (c) daily operational builds, (d) iterative development, (e) early customer feedback and (f) use of small programming teams. Microsoft's success was indeed remarkable, and their synch-n-stabilize approach did indeed help them create more than 20 million lines of code for Windows and Office 95, achieve customer satisfaction levels of 95% and maintain annual profit margins of approximately 36%.

6.6 Judo Strategy

In 1998, two management scholars from both the Harvard Business School and MIT's Sloan School of Management published a textbook on how Netscape managed the development of software for the Internet, dubbed as the 'judo strategy' [238]. The scholars were experts on software management approaches for the personal computer market and their one year case study from 1997 to 1998 was a grounded theory or emergent research design, which prophetically led them to be critical of Netscape's future. Whereas Microsoft's strategic advantage was its immense intellectual capital, Netscape's only advantage seemed to be its first-mover status, which was quickly eroding to Microsoft's market share for browsers at the time their book was published. In fact, the authors criticized Netscape for not having a technical CEO in the fast-moving Internet market, which was a very unconventional view among management scholars. Some of the more notable factors characteristic of Netscape's judo strategy included: (a) design products with modularized architectures; (b) use parallel development; (c) rapidly adapt to changing market priorities; (d) apply as much rigorous testing as possible and (e) use beta testing and open source strategies to solicit early market feedback on features, capabilities, quality and architecture.

6.7 Internet Time

In 1998, a management scholar from the Harvard Business School conducted a study on how U.S. firms manage the development of websites, referring to his approach as 'Internet time' [239]. His study states that 'constructs that support a more flexible development process are associated with better performing projects'. Basically, what he did was survey 29 software projects from 15 Internet firms such as Microsoft, Netscape, Yahoo, Intuit and Altavista. He set out to test the theory that website quality was associated with three major factors: (a) greater investments in architectural design, (b) early market feedback and (c) greater amounts of generational experience. Harvard Business School scholars believed that firms must spend a significant amount of resources to create flexible software designs, they must incorporate customer feedback on working 'beta' versions of software into evolving software designs, and higher website quality will be associated with more experience among computer programmers. After determining the extent to which the 29 website software projects complied with these ‘Internet time’ factors through a process of interviews and surveys, he then assembled a panel of 14 industry experts to objectively evaluate the associated website quality. Statistical analysis supported two of the hypotheses, e.g., greater architectural resources and early market feedback were associated with higher website quality, but not the third, e.g., greater experience among computer programmers is associated with higher website quality. This was one of the first studies to offer evidence in support of agile methods.

6.8 Extreme Programming

In 1998, 20 software managers working for the Chrysler Corporation published an article on how they devised a management approached called ‘extreme programming’ or XP to turn around a failing software project that would provide payroll services for 86 000 Chrysler employees [144]. Today, extreme programming is synonymous with agile methods or agile programming and is one of the most widely used agile methods although its market lead is eroding. In their article, they stated that ‘extreme programming rests on the values of simplicity, communication, testing, and aggressiveness’. They also stated that the ‘project had been declared a failure and all code thrown away, but using the extreme programming methodology, Chrysler started over from scratch and delivered a very successful result’. Extreme programming consists of 13 factors: (a) planning game, (b) small releases, (c) metaphor, (d) simple design, (e) tests, (f) refactoring, (g) pair programming, (h) continuous integration, (i) collective ownership, (j) onsite customer, (k) 40 hour workweek, (l) open workspace and (m) just rules. The planning game consists of estimation of the scope and timing of releases. Small releases consist of groups of iterations that will be put into production when complete. Metaphors are a common nomenclature for objects and classes. Simple design is self-evident; that is keeping the software architecture and design as simple as possible without adding unnecessary bells and whistles. Tests are unit tests that must be written before the code is written and run after the code is complete. Refactoring is defined as the continuous refining of the designs and code for simplicity and efficiency. Pair programming means a team of two programmers responsible for writing a software module or group of modules. Continuous integration is also self-evident; all code must be integrated with the system soon after it is written and unit tested as a form of validation. Collective ownership means that anyone has the authority to redesign and recode any portion of the system. Onsite customer means that a customer is always present with the software team. Open workspace refers to collocated teams with few walls to optimize communication. And, just rules means programmers must agree to a common set of flexible rules. What these 20 software managers did was start over, get an informal statement of customer needs, gradually evolve a simple system design using iterative development, apply rigorous testing, use small teams of programmers, and get early customer feedback on their evolving design. In the end, Chrysler was able to deploy an operational payroll system serving more than 86 000 employees.

6.9 Crystal Methods

In 1991, a software manager with IBM was asked to create an approach for managing the development of object-oriented systems called ‘crystal methods’ [146]. Crystal methods were piloted on a ‘$15 million firm, fixed-price project consisting of 45 people’. Crystal methods are a ‘family of methods with a common genetic code, one that emphasizes frequent delivery, close communication and reflective improvement’. Crystal methods are a family of 16 unique approaches for project teams ranging from 1 to 1000 people and project criticality ranging from loss of comfort to loss of life. The seven properties of crystal methods are: (a) frequent delivery; (b) reflective improvement; (c) close communication; (d) personal safety; (e) focus; (f) easy access to expert users and (g) a technical environment with automated testing, configuration management and frequent integration. The five strategies of crystal methods are: (a) exploratory 360, (b) early victory, (c) walking skeleton, (d) incremental re-architecture and (e) information radiators. The nine techniques of crystal methods are: (a) methodology shaping, (b) reflection workshop, (c) blitz planning, (d) Delphi estimation, (e) daily stand-ups, (f) agile interaction design, (g) process miniature, (h) side-by-side programming and (i) burn charts. The eight roles of crystal methods are: (a) sponsor, (b) team member, (c) coordinator, (d) business expert, (e) lead designer, (f) designer-programmer, (g) tester and (h) writer. The work products include a mission statement, team structure and conventions, reflection workshop results, project map, release plan, project status, risk list, iteration plan and status, viewing schedule, actor-goal list, use cases and requirements file, user role model, architecture description, screen drafts, common domain model, design sketches and notes, source code, migration code, tests, packaged system, bug reports and user help text.

6.10 Feature-Driven Development

In 1997, three software managers and five software developers created a software development approach called ‘feature driven development’ to help save a failed project for an international bank in Singapore [147]. In their textbook, they stated that ‘the bank had already made one attempt at the project and failed, and the project had inherited a skeptical user community, wary upper management, and a demoralized

TABLE I
SUMMARY OF PRACTICES AND PROCESSES OF AGILE METHODS

Feature	FDD	Extreme programming	DSDM	Scrum
Practice	• Domain object modeling	• Planning game	• Active user involvement	• Product backlog
	• Developing by feature	• Small releases	• Empowered teams	• Burndown chart
	• Class (code) ownership	• Metaphor	• Frequent delivery	• Sprint backlog
	• Feature teams	• Simple design	• Fitness (simplicity)	• Iterations and increments
	• Inspections	• Tests	• Iterations and increments	• Self managed teams
	• Regular build schedule	• Refactoring	• Reversible changes	• Daily scrums
	• Configuration management	• Pair programming	• Baselined requirements	
	• Reporting/visibility of results	• Continuous integration	• Integrated testing	
		• Collective ownership	• Stakeholder collaboration	
		• On-site customer		
		• 40-hour weeks		
		• Open workspace		
		• Just rules		
Process	**Develop an Overall Model**	**User Stories**	**Feasibility Study**	**Iteration (1)**
	Form the Modeling Team	*Requirements*	Feasibility Report	Sprint Planning Meeting
	Conduct a Domain Walkthrough	Acceptance Tests	Feasibility Prototype (optional)	Product Backlog
	Study Documents	**Architectural Spike**	Outline Plan	Sprint Backlog
	Develop Small Group Models	System Metaphor	Risk Log	Sprint
	Develop a Team Model	**Release (1)**	**Business Study**	Daily Scrum
	Refine the Overall Object Model	Release Planning	Business Area Definition	Shippable Code
	Write Model Notes	*Release Plan*	Prioritized Requirements List	Sprint Review Meeting
	Internal and External Assessment	Iteration (1)	Development Plan	Shippable Code
	Build a Features List	Iteration Planning	System Architecture Definition	Sprint Retrospective Meeting
	Form the Features List Team	*Iteration Plan*	Updated Risk Log	**Iteration (2)**
	Build the Features List	Daily Standup	**Functional Model Iteration**	Sprint Planning Meeting
	Internal and External Assessment	Collective Code Ownership	Functional Model	Product Backlog

continued

TABLE I
continued

Feature	FDD	Extreme programming	DSDM	Scrum
Process	**Plan by Feature**	Create Unit Tests	Functional Prototype (1)	Sprint Backlog
	Form the Planning Team	*Unit Tests*	Functional Prototype	Sprint
	Determine Development Sequence	Pair Programming	Functional Prototype Records	Daily Scrum
	Assign Features to Chief Coders	Move People Around	Functional Prototype (2)	Shippable Code
	Assign Classes to Developers	Refactor Mercilessly	Functional Prototype	Sprint Review Meeting
	Self Assessment	Continuous Integration	Functional Prototype Records	Shippable Code
	Iteration (1)	Acceptance Testing	Functional Prototype (n)	Sprint Retrospective Meeting
	Design by Feature	Iteration (2)	Functional Prototype	**Iteration (n)**
	Form a Feature Team	Iteration Planning	Functional Prototype Records	Sprint Planning Meeting
	Conduct Domain Walkthrough	*Iteration Plan*	Non-functional Requirements List	Product Backlog
	Study Referenced Documents	Daily Standup	Functional Model Review Records	Sprint Backlog
	Develop Sequence Diagrams	Collective Code Ownership	Implementation Plan	Sprint
	Refine the Object Model	Create Unit Tests	Timebox Plans	Daily Scrum
	Write Class/Method Prologue	*Unit Tests*	Updated Risk Log	Shippable Code
	Design Inspection	Pair Programming	**Design and Build Iteration**	Sprint Review Meeting
	Build by Feature	Move People Around	Timebox Plans	Shippable Code
	Implement Classes/Methods	Refactor Mercilessly	Design Prototype (1)	Sprint Retrospective Meeting
	Conduct Code Inspection	Continuous Integration	Design Prototype	
	Unit Test	Acceptance Testing	Design Prototype Records	
	Promote to the Build	Iteration (n)	Design Prototype (2)	
	Iteration (2)	Iteration Planning	Design Prototype	
	Design by Feature	*Iteration Plan*	Design Prototype Records	
	Form a Feature Team	Daily Standup	Design Prototype (n)	
	Conduct Domain Walkthrough	Collective Code Ownership	Design Prototype	
	Study Referenced Documents	Create Unit Tests	Design Prototype Records	
	Develop Sequence Diagrams	*Unit Tests*	Tested System	
	Refine the Object Model	Pair Programming	Test Records	
	Write Class/Method Prologue	Move People Around	**Implementation**	
	Design Inspection	Refactor Mercilessly	User Documentation	

	Build by Feature	Continuous Integration	Trained User Population
	Implement Classes/Methods	Acceptance Testing	Delivered System
	Conduct Code Inspection	**Release (2)**	Increment Review Document
	Unit Test	Iteration (1)	
	Promote to the Build	Iteration (2)	
	Iteration (n)	Iteration (n)	
	Design by Feature	**Release (n)**	
	Form a Feature Team	Iteration (1)	
	Conduct Domain Walkthrough	Iteration (2)	
Process	Study Referenced Documents	Iteration (n)	
	Develop Sequence Diagrams		
	Refine the Object Model		
	Write Class/Method Prologue		
	Design Inspection		
	Build by Feature		
	Implement Classes/Methods		
	Conduct Code Inspection		
	Unit Test		
	Promote to the Build		

development team'. Furthermore, they stated that 'the project was very ambitious, with a highly complex problem domain spanning three lines of business, from front office automation to backend legacy system integration'. In order to address this highly complex problem domain that had already experienced severe setbacks, they created an agile and adaptive software development process that is 'highly iterative, emphasizes quality at each step, delivers frequent tangible working results, provides accurate and meaningful progress, and is liked by clients, managers, and developers'. As shown in Table I, feature- driven development consists of five overall phases or processes: (a) develop an overall model, (b) build a features list, (c) plan by feature, (d) design by feature and (e) build by feature. Feature driven development also consists of other best practices in software management and development such as domain object modeling, developing by feature, individual class ownership, feature teams, inspections, regular builds, configuration management and reporting and visibility of results.

7. History of Studies on Agile Methods

7.1 Harvard Business School I

In 1998, two management scholars from the Harvard Business School conducted a survey of 391 respondents to test the effects of flexible versus inflexible product technologies, as shown in Figure 7 and Table II [240]. What they found was that projects using inflexible product technologies required over two times as much engineering effort as flexible product technologies (e.g., 17.94 vs. 8.15 months).

7.2 Harvard Business School II

In 1998, another management scholar from the Harvard Business School conducted a survey of 29 projects from 15 U.S. Internet firms to test the effects of flexible software development management approaches on website quality [239]. What he found was that flexible product architectures and customer feedback on early beta releases were correlated to higher levels of website quality.

7.3 Boston College Carroll School of Management

In 1999, two management scholars from Boston College's Carroll School of Management conducted a case study of 28 software projects to determine the effects of iterative development on project success [241]. What they found was that software projects that use iterative development deliver working software 38% sooner, complete their projects twice as fast, and satisfy over twice as many software requirements.

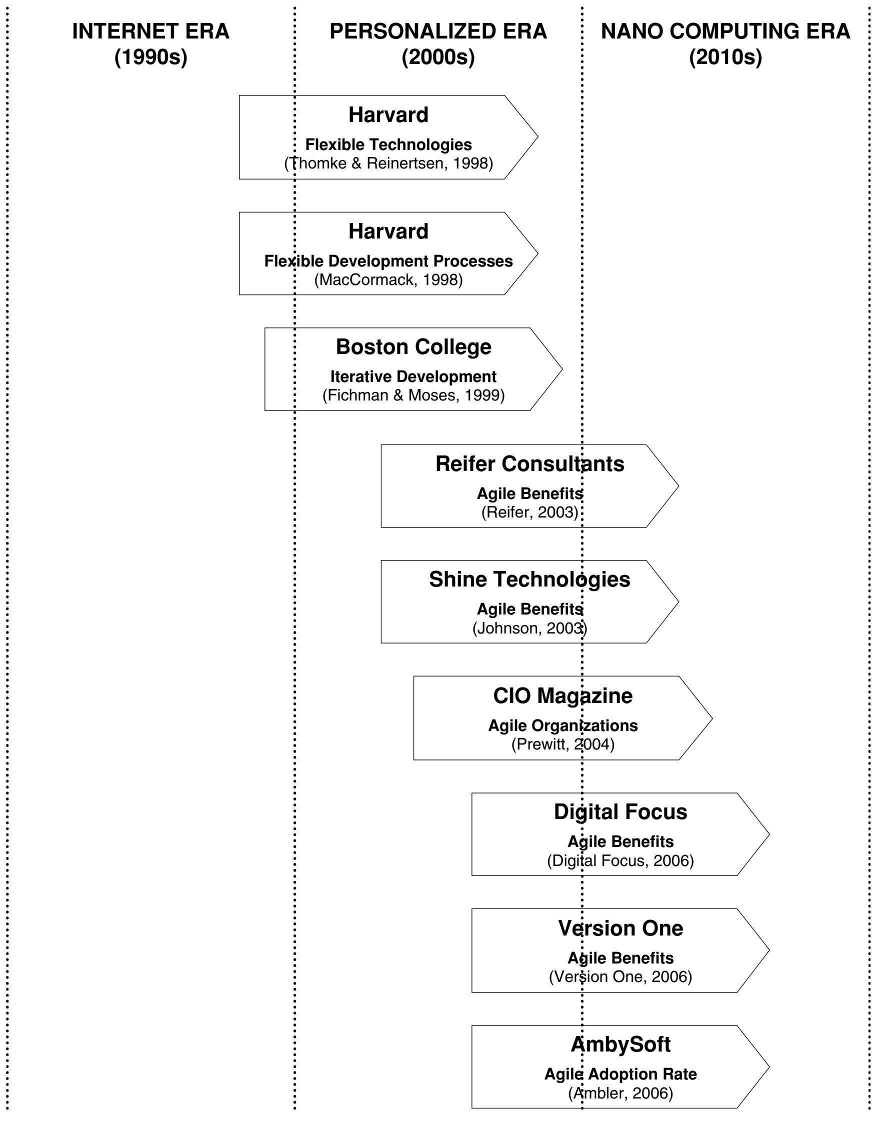
INTERNET ERA (1990s)
PERSONALIZED ERA (2000s)
NANO COMPUTING ERA (2010s)
Harvard Flexible Technologies (Thomke & Reinertsen, 1998)
Harvard Flexible Development Processes (MacCormack, 1998)
Boston College Iterative Development (Fichman & Moses, 1999)
Reifer Consultants Agile Benefits (Reifer, 2003)
Shine Technologies Agile Benefits (Johnson, 2003)
CIO Magazine Agile Organizations (Prewitt, 2004)
Digital Focus Agile Benefits (Digital Focus, 2006)
Version One Agile Benefits (Version One, 2006)
AmbySoft Agile Adoption Rate (Ambler, 2006)

FIG. 7. Timeline and history of studies on agile methods.

7.4 Reifer Consultants

In 2003, Reifer Consultants conducted a survey of 78 projects from 18 firms to determine the effects of using agile methods to manage the development of software [242]. What they found was that 14% to 25% of respondents experienced productivity gains, 7% to 12% reported cost reductions and 25% to 80% reported time-to-market improvements.

7.5 Shine Technologies

In 2003, Shine Technologies conducted an international survey of 131 respondents to determine the effects of using agile methods to manage the development of software [243]. What they found was that 49% of the respondents experienced cost reductions, 93% of the respondents experienced productivity increases, 88% of the respondents experienced quality increases and 83% experienced customer satisfaction improvements.

7.6 CIO Magazine

In 2004, CIO Magazine conducted a survey of 100 information technology executives with an average annual budget of $270 million to determine the effects of agile management on organizational effectiveness [244]. What they found was that 28% of respondents had been using agile management methods since 2001, 85% of the respondents were undergoing enterprise-wide agile management initiatives, 43% of the respondents were using agile management to improve organizational growth and market share, and 85% said agile management was a core part of their organizational strategy.

7.7 Digital Focus

In 2006, Digital Focus conducted a survey of 136 respondents to determine the effects of using agile methods to manage the development of software [245]. What they found was that 27% of the respondents were adopting agile methods for a project, 23% of the respondents were adopting agile methods company wide, 51% of the respondents wanted to use agile methods to speed up the development process, 51% of the respondents said they lacked the skills necessary to implement agile methods at the project level, 62% of the respondents said they lacked the skills necessary to implement agile methods at the organization level and 60% planned on teaching themselves how to use agile methods.

7.8 Version One

In 2006, Version One conducted an international survey of 722 respondents to determine the effects of using agile methods to manage the development of software [246]. What they found was that 86% of the respondents reported time-to-market improvements, 87% of the respondents reported productivity improvements, 86% of the respondents reported quality improvements, 63% of the respondents reported cost reductions, 92% of the respondents reported the ability to manage changing priorities, 74% of the respondents reported improved morale, 72% of the respondents reported risk reductions, 66% of the respondents reported satisfaction of business goals and 40% were using the scrum method.

7.9 AmbySoft 2006

In 2006, Ambysoft conducted an international survey of 4232 respondents to determine the effects of using agile methods to manage the development of software [247]. What they found was that 41% of organizations were using agile methods; 65% used more than one type of agile method; 44% reported improvements in productivity, quality and cost reductions; and 38% reported improvements in customer satisfaction.

7.10 AmbySoft 2007

In 2007, Ambysoft conducted another international survey of 781 respondents to further determine the effects of using agile methods to manage the development of software [248]. What they found was that 69% of organizations had adopted agile methods, 89% of agile projects had a success rate of 50% or greater, and 99% of organizations are now using iterative development.

7.11 UMUC

In 2007, a student at the University of Maryland University College (UMUC) conducted a survey of 250 respondents to determine the effects of using agile methods on website quality [249]. What he found was that: (a) 70% of all developers are using many if not all aspects of agile methods; (b) 79% of all developers using agile methods have more than 10 years of experience; (c) 83% of all developers using agile methods are from small- to medium-sized firms; (d) 26% of all developers using agile methods have had improvements of 50% or greater; and (e) developers using all aspects of agile methods produced better e-commerce websites.

Table II
Summary of Recent Studies and Surveys of Agile Methods

Year	Source	Findings	Responses
1998	Harvard (Thomke and Reinertsen, 1998)	50% reduction in engineering effort 55% improvement in time to market 925% improvement in number of changes allowed	391
1998	Harvard (MacCormack, 1998)	48% productivity increase over traditional methods 38% higher quality associated with more design effort 50% higher quality associated with iterative development	29
1999	Boston College (Fichman and Moses, 1999)	38% reduction in time to produce working software 50% time to market improvement 50% more capabilities delivered to customers	28
2003	Reifer Consultants (Reifer, 2003)	20% reported productivity gains 10% reported cost reductions 53% reported time-to-market improvements	78
2003	Shine Technologies (Johnson, 2003)	49% experienced cost reductions 93% experienced productivity increases 88% experienced customer satisfaction improvements	131
2004	CIO Magazine (Prewitt, 2004)	28% had been using agile methods since 2001 85% initiated enterprise-wide agile methods initiatives 43% used agile methods to improve growth and marketshare	100
2006	Digital Focus (Digital Focus, 2006)	27% of software projects used agile methods 23% had enterprise-wide agile methods initiatives 51% used agile methods to speed-up development	136
2006	Version One (Version One, 2006)	86% reported time-to-market improvements 87% reported productivity improvements 92% reported ability to dynamically change priorities	722
2006	AmbySoft (Ambler, 2006)	41% of organizations used agile methods 44% reported improved productivity, quality, and costs 38% reported improvements in customer satisfaction levels	4,232
2007	AmbySoft (Ambler, 2007)	69% of organizations had adopted agile methods 89% of agile projects had a success rate of 50% or greater 99% of organizations are now using iterative development	781
2007	UMUC (Rico, 2007)	70% of developers using most aspects of agile methods 26% of developers had improvements of 50% or greater Agile methods are linked to improved website quality	250

8. Conclusions

The gaps and problem areas in the literature associated with agile methods and website quality are numerous. First, there are few scholarly studies of agile methods.

That is, this author has been unable to locate and identify very many scholarly studies containing theoretical conceptual models of agile methods. Furthermore, few of the articles in the literature review were based on systematic qualitative or quantitative studies of agile methods. The literature review only mentions textbooks and articles with notional concepts in agile methods. Most of the quantitative survey research mentioned in the literature review was of a rudimentary attitudinal nature. In addition, few of the articles mentioned in the literature review addressed all four of the factors associated with agile methods (e.g., iterative development, customer feedback, well-structured teams and flexibility). And, few of them were systematically linked to scholarly models of website quality. So, the gaps are quite clear, a dearth holistic scholarship on agile methods and scholarly outcomes such as information systems quality.

There is a clear need for new studies on agile methods. We hope to inspire the creation of a long line of scholarly studies of agile methods. Furthermore, we hope to inspire more studies that attempt to link the factors of agile methods to scholarly models of information systems quality. First and foremost, there is a need for a systematic analysis of scholarly literature associated with the factors of agile methods. Then there is a need for a scholarly theoretical model of agile methods, depicting the factors, variables and hypotheses associated with using agile methods. In addition, there is a need for an analysis of scholarly literature to identify the factors and variables associated with website quality. Finally, there is a need to identify, survey, select or develop scholarly measures and instrument items for both agile methods and information systems quality, both of which together constitute new studies of agile methods.

References

[1] Rosen S., 1969. Electronic computers: a historical survey. *ACM Computing Surveys*, **1**(1):7–36.

[2] Denning J., 1971. Third generation computer systems. *ACM Computing Surveys*, **3**(4):175–216.

[3] Tanenbaum A., 2001. *Modern Operating Systems*. Englewood Cliffs, NJ: Prentice Hall.

[4] Carlson B., Burgess A., and Miller C., 1996. *Timeline of Computing History*. Retrieved on October 21, 2006, from http://www.computer.org/portal/cms_docs_ieeecs/ieeecs/about/history/timeline.pdf.

[5] Nerlove M., 2004. Programming languages: A short history for economists. *Journal of Economic and Social Measurement*, **29**(1–3):189–203.

[6] Sammet J. E., 1972b. Programming languages: history and future. *Communications of the ACM*, **15**(7):601–610.

[7] Pigott D., 2006. *HOPL: An Interactive Roster of Programming Languages*. Retrieved on October 21, 2006, from http://hopl.murdoch.edu.au.

[8] Chen Y., Dios R., Mili A., Wu L., and Wang K., 2005. An empirical study of programming language trends. *IEEE Software*, **22**(3):72–78.

[9] Cusumano M. A., 1991. *Japan's Software Factories: A Challenge to U.S. Management*. New York, NY: Oxford University Press.

[10] Campbell-Kelly M., 1995. Development and structure of the international software industry: 1950–1990. *Business and Economic History*, **24**(2):73–110.

[11] Steinmueller W. E., 1996. The U.S. software industry: an analysis and interpretive history. In Mowery D. C. (Ed.), *The International Computer Software Industry* (pp. 25–52). New York, NY: Oxford University Press.

[12] Johnson L., 1998. A view from the 1960s: how the software industry began. *IEEE Annals of the History of Computing*, **20**(1):36–42.

[13] Campbell-Kelly M., 2001. Not only microsoft: the maturing of the personal computer software industry: 1982–1995. *Business History Review*, **75**(1):103–146.

[14] Middleton R., and Wardley P., 1990. Information technology in economic and social history: the computer as philosopher's stone or pandora's box? *Economic History Review*, **43**(4):667–696.

[15] U.S. Department of Commerce. 2003. *Digital Economy*. Washington, DC: Author.

[16] Borck J., and Knorr E., 2005. A field guide to hosted apps. *Infoworld*, **27**(16):38–44.

[17] Reid R. H., 1997. *Architects of the Web: 1,000 Days that Build the Future of Business*. New York, NY: John Wiley and Sons.

[18] Leiner B., Cerf V. G., Clark D. D., Kahn R. E., Kleinrock L., Lynch D. C., et al. 1997. The past and future history of the internet. *Communications of the ACM*, **40**(2):102–108.

[19] Mowery D. C., and Simcoe T., 2002. Is the internet a US invention? An economic and technological history of computer networking. *Research Policy*, **31**(8/9):1369–1387.

[20] Kalakota R., and Whinston A., 1996. *Electronic Commerce: A Manager's Guide*. Reading, MA: Addison Wesley.

[21] U.S. Census Bureau. 2006. *E-stats*. Washington, DC: Author.

[22] Mandell L., 1977. Diffusion of EFTS among national banks. *Journal of Money, Credit, and Banking*, **9**(2):341–348.

[23] Anonymous. 1965. A fascinating teller. *Banking*, **58**(3):95–95.

[24] Ellis G. H., 1967. The fed's paperless payments mechanism. *Banking*, **67**(60):100–100.

[25] New York Stock Exchange 2006. *NYSE Timeline of Technology*. Retrieved on October 23, 2006, from http://www.nyse.com/about/history/timeline_technology.html.

[26] Anonymous. 1971. Tela-fax takes dead aim on checks, credit cards. *Banking*, **64**(3):52–53.

[27] Anonymous. 1975. Bank patents its EFT system. *Banking*, **67**(1):88–88.

[28] Lynch J. E., 1990. The impact of electronic point of sale technology (EPOS) on marketing strategy and retailer-supplier relationships. *Journal of Marketing Management*, **6**(2):157–168.

[29] Accredited Standards Committee. 2006. *The Creation of ASC X12*. Retrieved on October 22, 2006, from http://www.x12.org/x12org/about/X12History.cfm.

[30] Smith A., 1988. EDI: will banks be odd man out? *ABA Banking Journal*, **80**(11):77–79.

[31] Internet Retailer. 2007. *Internet Retailer 2007 Edition Top 500 Guide: Profiles and Statistics of America's 500 Largest Retail Web Sites Ranked by Annual Sales*. Chicago, IL: Vertical Web Media, LLC.

[32] Teorey T. J., and Fry J. P., 1980. The logical record access approach to database design. *ACM Computing Surveys*, **12**(2):179–211.

[33] American National Standards Institute. 1988. *Information Resource Dictionary System* (ANSI X3.138–1988). New York, NY: Author.

[34] Dolk D. R., and Kirsch R. A., 1987. A relational information resource dictionary system. *Communications of the ACM*, **30**(1):48–61.

[35] Lombardi L., 1961. Theory of files. *Communications of the ACM*, **4**(7):324–324.

[36] Bachman C. W., 1969. Data structure diagrams. *ACM SIGMIS Database*, **1**(2):4–10.

[37] Dodd G. G., 1969. Elements of data management systems. *ACM Computing Surveys*, **1**(2):117–133.

[38] Codd E. F., 1970. A relational model of data for large shared data banks. *Communications of the ACM*, **13**(6):377–387.
[39] Cardenas A. F., 1977. Technology for automatic generation of application programs. *MIS Quarterly*, **1**(3):49–72.
[40] Montalbano M., 1962. Tables, flowcharts, and program logic. *IBM Systems Journal*, **1**(1):51–63.
[41] Oldfather P. M., Ginsberg A. S., and Markowitz H. M., 1966. *Programming by Questionnaire: How to Construct a Program Generator* (RM-5128-PR). Santa Monica, CA: The RAND Corporation.
[42] Teichroew D., and Sayani H., 1971. Automation of system building. *Datamation*, **17**(16):25–30.
[43] Sammet J. E., 1972a. An overview of programming languages for special application areas. *Proceedings of the Spring Joint American Federation of Information Processing Societies Conference (AFIPS 1972)*, Montvale, New Jersey, USA, 299–311.
[44] Boehm B. W., and Ross R., 1988. Theory W software project management: a case study. *Proceedings of the 10th International Conference on Software Engineering, Singapore*, 30–40.
[45] Anderson R. M., 1966. Management controls for effective and profitable use of EDP resources. *Proceedings of the 21st National Conference for the Association for Computing Machinery, New York, NY, USA*, 201–207.
[46] Fisher A. C., 1968. Computer construction of project networks. *Communications of the ACM*, **11**(7):493–497.
[47] Merwin R. E., 1972. Estimating software development schedules and costs. *Proceedings of the Ninth Annual ACM IEEE Conference on Design Automation*, New York, NY, USA, 1–4.
[48] Jones T. C., 1978. Measuring programming quality and productivity. *IBM Systems Journal*, **17**(1):39–63.
[49] Ives B., and Olson M. H., 1984. User involvement and MIS success: a review of research. *Management Science*, **30**(5):586–603.
[50] Dunn O. E., 1966. Information technology: a management problem. *Proceedings of the Third ACM IEEE Conference on Design Automation*, New York, NY, USA, 5.1–5.29.
[51] Fitch A. E., 1969. A user looks at DA: yesterday, today, and tomorrow. *Proceedings of the Sixth ACM IEEE Conference on Design Automation, New York, NY, USA*, 371–382.
[52] Milne M. A., 1971. CLUSTR: a program for structuring design problems. *Proceedings of the Eighth Annual ACM IEEE Design Automation Conference*, Atlantic City, New Jersey, USA, 242–249.
[53] Miller L. A., 1974. Programming by non-programmers. *International Journal of Man-Machine Studies*, **6**(2):237–260.
[54] Bechtolsheim A., 1978. Interactive specification of structured designs. *Proceedings of the 15th Annual ACM IEEE Design Automation Conference*, Las Vegas, Nevada, USA, 261–263.
[55] Dijkstra E. W., 1969. *Notes on structured programming* (T.H.-Report 70-WSK-03). Eindhoven, Netherlands: Technological University of Eindhoven.
[56] Wirth N., 1971. Program development by stepwise refinement. *Communications of the ACM*, **14**(4):221–227.
[57] Stevens W. P., Myers G. J., and Constantine L. L., 1974. Structured design. *IBM Systems Journal*, **13**(2):115–139.
[58] Yourdon E., 1976. The emergence of structured analysis. *Computer Decisions*, **8**(4):58–59.
[59] Clarke E. M., and Wing J. M., 1996. Formal methods: state of the art and future directions. *ACM Computing Surveys*, **28**(4):626–643.
[60] Hoare C. A. R., 1969. An axiomatic basis for computer programming. *Communications of the ACM*, **12**(10):576–583.
[61] Wegner P., 1972. The vienna definition language. *ACM Computing Surveys*, **4**(1):5–63.
[62] Hoare C. A. R., 1978. Communicating sequential processes. *Communications of the ACM*, **21**(8):666–677.

[63] Linger R. C., Mills H. D., and Witt B. I., 1979. *Structured programming: Theory and Practice*. Reading, MA: Addison-Wesley.

[64] Shapiro S., 1997. Splitting the difference: the historical necessity of synthesis in software engineering. *IEEE Annals of the History of Computing*, **19**(1):20–54.

[65] Van Den Bosch F., Ellis J. R., Freeman P., Johnson L., McClure C. L., Robinson D., et al. 1982. Evaluation of software development life cycle: methodology implementation. *ACM SIGSOFT Software Engineering Notes*, **7**(1):45–60.

[66] Royce W. W., 1970. Managing the development of large software systems. *Proceedings of the Western Electronic Show and Convention (WESCON 1970)*, Los Angeles, California, USA, 1–9.

[67] Basili V. R., and Turner J., 1975. Iterative enhancement: a practical technique for software development. *IEEE Transactions on Software Engineering*, **1**(4):390–396.

[68] Bauer F. L., 1976. Programming as an evolutionary process. *Proceedings of the Second International Conference on Software Engineering*, San Francisco, California, USA, 223–234.

[69] Cave W. C., and Salisbury A. B., 1978. Controlling the software life cycle: the project management task. *IEEE Transactions on Software Engineering*, **4**(4):326–337.

[70] Boehm B. W., 1986. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, **11**(4):14–24.

[71] Belz F. C., 1986. Applying the spiral model: observations on developing system software in ada. *Proceedings of the 4th Annual National Conference on Ada Technology*, Atlanta, Georgia, USA, 57–66.

[72] Iivari J., 1987. A hierarchical spiral model for the software process. *ACM SIGSOFT Software Engineering Notes*, **12**(1):35–37.

[73] Sauer C., Jeffery D. R., Land L., and Yetton P., 2000. The effectiveness of software development technical reviews: a behaviorally motivated program of research. *IEEE Transactions on Software Engineering*, **26**(1):1–15.

[74] Weinberg G. M., 1971. *The Psychology of Computer Programming*. New York, NY: Van Nostrand Reinhold.

[75] Mills H. D., 1971. *Chief Programmer Teams: Principles and Procedures* (IBM Rep. FSC 71–5108). Gaithersburg, MD: IBM Federal Systems Division.

[76] Waldstein N. S., 1974. *The Walk Thru: A Method of Specification Design and Review* (TR 00.2536). Poughkeepsie, NY: IBM Corporation.

[77] Fagan M. E., 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, **15**(3):182–211.

[78] U.S. Department of Defense. 1976. *Military Standard: Technical Reviews and Audits for Systems, Equipments, and Computer Software* (MIL-STD-1521A). Hanscom AFB, MA: Author.

[79] Rosson M. B., and Alpert S. R., 1990. The cognitive consequences of object oriented design. *Human Computer Interaction*, **5**(4):345–379.

[80] Dahl O. J., and Nygaard K., 1966. Simula: an algol based simulation language. *Communications of the ACM*, **9**(9):671–678.

[81] Kay A., 1968. *Flex: A Flexible Extensible Language*. Unpublished master's thesis, University of Utah, Salt Lake City, UT, United States.

[82] Kay A., 1969. *The Reactive Engine*. Unpublished doctoral dissertation, University of Utah, Salt Lake City, UT, United States.

[83] Kay A., 1974. *Smalltalk: A Communication Medium for Children of All Ages*. Palo Alto, CA: Xerox Palo Alto Research Center.

[84] Parnas D. L., 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, **15**(12):1053–1058.

[85] Booch G., 1981. Describing software design in ada. *SIGPLAN Notices*, **16**(9):42–47.

[86] McIntyre S. C., and Higgins L. F., 1988. Object oriented systems analysis and design: methodology and application. *Journal of Management Information Systems*, **5**(1):25–35.

[87] Whittaker J. A., 2000. What is software testing? And why is it so hard? *IEEE Software*, **17**(1):70–79.

[88] Brown J. R., and Lipow M., 1975. Testing for software reliability. *Proceedings of the First International Conference on Reliable Software, Los Angeles, California, USA*, 518–527.

[89] Goodenough J. B., and Gerhart S. L., 1975. Toward a theory of test data selection. *Proceedings of the First International Conference on Reliable Software*, Los Angeles, California, USA, 493–510.

[90] Panzl D. J., 1976. Test procedures: a new approach to software verification. *Proceedings of the Second International Conference on Reliable Software, San Francisco, California, USA*, 477–485.

[91] Walsh D. A., 1977. Structured testing. *Datamation*, **23**(7):111–111.

[92] Leblang D. B., and Chase R. P., 1984. Computer aided software engineering in a distributed environment. *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, Pennsylvania, USA, 104–112.

[93] Dowson M., and Wileden J. C., 1985. Panel discussion on the software process and software environments. *Proceedings of the 8th International Conference on Software Engineering, London, England*, 302–305.

[94] Nemchinova Y., 2007. *The Feasibility of Using Software Tools in Teaching Technical Courses*. Unpublished doctoral dissertation, University of Baltimore, Baltimore, MD.

[95] Baker F. T., 1975. Structured programming in a production programming environment. *Proceedings of the First International Conference on Reliable Software*, Los Angeles, California, USA, 172–185.

[96] Bratman H., and Court T., 1975. The software factory. *IEEE Computer*, **8**(5):28–37.

[97] Amey W. W., 1979. The computer assisted software engineering (CASE) system. *Proceedings of the Fourth International Conference on Software Engineering, Munich, Germany*, 111–115.

[98] Wegner P., 1980. The ada language and environment. *ACM SIGSOFT Software Engineering Notes*, **5**(2):8–14.

[99] Day F. W., 1983. Computer aided software engineering (CASE). *Proceedings of the 20th Conference on Design Automation*, Miami Beach, Florida, USA, 129–136.

[100] Banker R. D., and Kauffman R. J., 1991. Reuse and productivity in integrated computer aided software engineering: an empirical study. *MIS Quarterly*, **15**(3):375–401.

[101] Hsieh D., 1995. David hsieh of lbms: integrated case is dead. *VARBusiness*, **11**(17):136–136.

[102] Abdel-Hamid T. K., 1988. The economics of software quality assurance: a simulation based case study. *MIS Quarterly*, **12**(3):394–411.

[103] Fujii M. S., 1978. A comparison of software assurance methods. *Proceedings of the First Annual Software Quality Assurance Workshop on Functional and Performance Issues*, New York, NY, USA, 27–32.

[104] Adrion W. R., Branstad M. A., and Cherniavsky J. C., 1982. Validation, verification, and testing of computer software. *ACM Computing Surveys*, **14**(2):159–192.

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

[106] Rigby P. J., Stoddart A. G., and Norris M. T., 1990. Assuring quality in software: practical experiences in attaining ISO 9001. *British Telecommunications Engineering*, **8**(4):244–249.

[107] Notkin D., 1989. The relationship between software development environments and the software process. *Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Boston, Massachusetts, USA, 107–109.

[108] Crosby P. B., 1979. *Quality is Free*. New York, NY: McGraw-Hill.

[109] Radice R. A., Harding J. T., Munnis P. E., and Phillips R. W., 1985. A programming process study. *IBM Systems Journal*, **24**(2):91–101.

[110] Humphrey W. S., 1987. *Characterizing the Software Process: A Maturity Framewor*k (CMU/SEI-87-TR-011). Pittsburgh, PA: Software Engineering Institute.

[111] Weber C., Paulk M., Wise C., and Withey J., 1991. *Key Practices of the Capability Maturity Model* (CMU/SEI-91-TR-025). Pittsburgh, PA: Software Engineering Institute.

[112] Agarwal R., Prasad J., Tanniru M., and Lynch J., 2000. Risks of rapid application development. *Communications of the ACM*, **43**(11):177–188.

[113] Naumann J. D., and Jenkins A. M., 1982. Prototyping: the new paradigm for systems development. *MIS Quarterly*, **6**(3):29–44.

[114] Alavi M., 1985. Some thoughts on quality issues of end-user developed systems. *Proceedings of the 21st Annual Conference on Computer Personnel Research, Minneapolis, Minnesota, USA*, 200–207.

[115] Guide International, Inc. 1986. *Joint Application Design*. Chicago, IL: Author.

[116] Gane C., 1987. *Rapid Systems Development*. New York, NY: Rapid Systems Development, Inc.

[117] Martin J., 1991. *Rapid Application Development*. New York, NY: Macmillan.

[118] Frakes W. B., and Kang K., 2005. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, **31**(7):529–536.

[119] McIlroy M. D., 1968. Mass produced software components. *Proceedings of the NATO Software Engineering Conference*, Garmisch, Germany, 138–155.

[120] Pyster A., 1982. Software development productivity. *Proceedings of the National ACM Conference. Dallas, Texas, USA*, 94–94.

[121] Lubars M. D., 1982. Affording higher reliability through software reusability. *ACM SIGSOFT Software Engineering Notes*, **11**(5):39–42.

[122] Zychlinski B. Z., and Palomar M. A., 1984. A software quality assurance program through reusable code. *Proceedings of the 3rd Annual International Conference on Systems Documentation*, Mexico City, Mexico, 107–113.

[123] Lim W. C., 1994. Effects of reuse on quality, productivity, and economics. *IEEE Software*, **11**(5): 23–30.

[124] Neighbors J. M., 1984. The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, **10**(5):564–574.

[125] Jameson K. W., 1989. A model for the reuse of software design information. *Proceedings of the 11th International Conference on Software Engineering, Pittsburgh, Pennsylvania, USA*, 205–216.

[126] Holibaugh R., Cohen S., Kang K., and Peterson S., 1989. Reuse: where to begin and why. *Proceedings of the Conference on Tri-Ada*, Pittsburgh, Pennsylvania, USA, 266–277.

[127] D'Souza D. F., and Wills A. C., 1998. *Objects, Components, and Frameworks With UML: The Catalysis Approach*. Reading, MA: Addison Wesley.

[128] 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).

[129] Poulin J. S., 1997. *Measuring Software Reuse: Principles, Practices, and Economic Models*. Reading, MA: Addison Wesley.

[130] Lim W. C., 1998. *Managing Software Reuse: A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components*. Upper Saddle River, NJ: Prentice Hall.

[131] Edwards S. H., 1999. The state of reuse: perceptions of the reuse community. *ACM SIGSOFT Software Engineering Notes*, **24**(3):32–36.

[132] Sherif K., and Vinze A., 1999. A qualitative model for barriers to software reuse adoption. *Proceeding of the 20th International Conference on Information Systems, Charlotte, North Carolina, USA*, 47–64.

[133] Kruchten P., Obbink H., and Stafford J., 2006. The past, present, and future of software architecture. *IEEE Software*, **23**(2):22–30.

[134] Prieto-Diaz R., 1987. Domain analysis for reusability. *Proceedings of the 11th Annual International Computer Software and Applications Conference (COMPSAC 1987)*, Tokyo, Japan, 23–29.

[135] Arango G., 1988. *Domain Engineering for Software Reuse* (ICS-RTP-88–27). Irvine, CA: University of California Irvine, Department of Information and Computer Science.

[136] Horowitz B. B., 1991. *The Importance of Architecture in DoD Software* (Technical Report M91-35). Bedford, MA: The Mitre Corporation.

[137] Wegner P., Scherlis W., Purtilo J., Luckham D., and Johnson R., 1992. Object oriented megaprogramming. *Proceedings on Object Oriented Programming Systems, Languages, and Applications, Vancouver, British Columbia, Canada*, 392–396.

[138] Northrop L. M., 2002. SEI's software product line tenets. *IEEE Software*, **19**(4):32–40.

[139] Highsmith J. A., 2002. *Agile Software Development Ecosystems*. Boston, MA: Addison Wesley.

[140] Beck K., 1999. Embracing change with extreme programming. *IEEE Computer*, **32**(10):70–77.

[141] Agile Manifesto. 2001. *Manifesto for Agile Software Development*. Retrieved on November 29, 2006, from http://www.agilemanifesto.org.

[142] Millington D., and Stapleton J., 1995. Developing a RAD standard. *IEEE Software*, **12**(5):54–56.

[143] Schwaber K., 1995. Scrum development process. *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 1995)*, Austin, Texas, USA, 117–134.

[144] Anderson A., Beattie R., Beck K., Bryant D., DeArment M., Fowler M., et al. 1998. Chrysler goes to extremes. *Distributed Computing Magazine*, **1**(10):24–28.

[145] O'Reilly T., 1999. Lessons from open source software development. *Communications of the ACM*, **42**(4):32–37.

[146] Cockburn A., 2002a. *Agile Software Development*. Boston, MA: Addison Wesley.

[147] Palmer S. R., and Felsing J. M., 2002. *A Practical Guide to Feature Driven Development*. Upper Saddle River, NJ: Prentice Hall.

[148] Kruchten P., 2000. *The Rational Unified Process: An Introduction*. Reading, MA: Addison Wesley.

[149] Highsmith J. A., 2000. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York, NY: Dorset House.

[150] Poppendieck M., and Poppendieck T., 2003. *Lean Software Development: An Agile Toolkit for Software Development Managers*. Boston, MA: Addison Wesley.

[151] McIlroy M. D., 1960. Macro instruction extensions of compiler languages. *Communications of the ACM*, **3**(4):214–220.

[152] Conte S. D., Dunsmore H. E., and Shen V. Y., 1986. *Software Engineering Metrics and Models*. Menlo Park, CA: Benjamin Cummings.

[153] Martin J., 1965. *Programming Real-Time Computer Systems*. Englewood Cliffs, NJ: Prentice Hall.

[154] Halstead M. H., 1977. *Elements of Software Science*. New York, NY: Elsevier North Holland.

[155] Basili V. R., and Reiter R. W., 1979. An investigation of human factors in software development. *IEEE Computer*, **12**(12):21–38.

[156] Albrecht A. J., 1979. Measuring application development productivity. *Proceedings of the IBM Applications Development Joint SHARE/GUIDE Symposium, Monterrey, California, USA*, 83–92.

[157] Kan S. H., 1995. *Metrics and Models in Software Quality Engineering*. Reading, MA: Addison-Wesley.

[158] Weinberg G. M., and Gressett G. L., 1963. An experiment in automatic verification of programs. *Communications of the ACM*, **6**(10):610–613.

[159] Youngs E. A., 1970. *Error Proneness in Programming*. Unpublished doctoral dissertation. University of North Carolina at Chapel Hill, Chapel Hill, NC, United States.

[160] Shooman M. L., and Bolsky M. I., 1975. Types, distribution, and test and correction times for programming errors. *Proceedings of the International Conference on Reliable Software*, Los Angeles, California, USA, 347–357.

[161] Lipow M., 1982. Number of faults per line of code. *IEEE Transactions on Software Engineering*, **8**(4):437–439.

[162] Shen V. Y., Yu T. J., Thebaut S. M., and Paulsen L. R., 1985. Identifying error prone software: An empirical study. *IEEE Transactions on Software Engineering*, **11**(4):317–324.

[163] Institute of Electrical and Electronics Engineers. 1990. *IEEE Standard Glossary of Software Engineering Terminology* (IEEE Std 610.12–1990). New York, NY: Author.

[164] Rubey R. J., and Hartwick R. D., 1968. Quantitative measurement of program quality. *Proceedings of the 23rd ACM National Conference*, Washington, DC, USA, 671–677.

[165] Boehm B. W., Brown J. R., Kaspar H., and Lipow M., 1973. *Characteristics of Software Quality* (TRW-SS-73-09). Redondo Beach, CA: TRW Corporation.

[166] Swanson E. B., 1976. The dimensions of maintenance. *Proceedings of the Second International Conference on Software Engineering*, San Francisco, California, USA, 492–497.

[167] Gilb T., 1977. *Software Metrics*. Cambridge, MA: Winthrop Publishers.

[168] Cavano J. P., and McCall J. A., 1978. A framework for the measurement of software quality. *Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues*, San Diego, California, USA, 133–139.

[169] Dzida W., Herda S., and Itzfeldt W. D., 1978. User perceived quality of interactive systems. *Proceedings of the Third International Conference on Software Engineering, Atlanta, Georgia, USA*, 188–195.

[170] Gaffney J. E., 1981. Metrics in software quality assurance. *Proceedings of the ACM SIGMETRICS Workshop/Symposium on Measurement and Evaluation of Software Quality*, Las Vegas, Nevada, USA, 126–130.

[171] Sunazuka T., Azuma M., and Yamagishi N., 1985. Software quality assessment technology. *Proceedings of the Eighth International Conference on Software Engineering*, London, England, 142–148.

[172] Arthur L. J., 1985. *Measuring Programmer Productivity and Software Quality*. New York, NY: John Wiley and Sons.

[173] Grady R. B., and Caswell R. B., 1987. *Software Metrics: Establishing a Company Wide Program*. Englewood Cliffs, NJ: Prentice Hall.

[174] Akiyama F., 1971. An example of software system debugging. *Proceedings of the International Federation for Information Processing Congress*, Ljubljana, Yugoslavia, 353–379.

[175] Motley R. W., and Brooks W. D., 1977. *Statistical Prediction of Programming Errors* (RADC-TR-77-175). Griffis AFB, NY: Rome Air Development Center.

[176] Potier D., Albin J. L., Ferreol R., and Bilodeau A., 1982. Experiments with computer software complexity and reliability. *Proceedings of the Sixth International Conference on Software Engineering*, Tokyo, Japan, 94–103.

[177] Halstead M. H., 1972. Natural laws controlling algorithm structure? *ACM SIGPLAN Notices*, **7**(2):19–26.

[178] Weissman L., 1973. Psychological complexity of computer programs. *ACM SIGPLAN Notices*, **8**(6):92–95.

[179] Elshoff J. L., 1976. An analysis of some commercial PL/1 programs. *IEEE Transactions on Software Engineering*, **2**(2):113–120.

[180] Dunsmore H. E., and Gannon J. D., 1979. Data referencing: an empirical investigation. *IEEE Computer*, **12**(12):50–59.

[181] Henry S., and Kafura D., 1981, Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, **7**(5):510–518.

[182] McCabe T. J., 1976. A complexity measure. *IEEE Transactions on Software Engineering*, **2**(4): 308–320.

[183] Schneidewind N. F., and Hoffmann H., 1979. An experiment in software error data collection and analysis. *IEEE Transactions on Software Engineering*, **5**(3):276–286.

[184] Woodward M. R., Hennell M. A., and Hedley D., 1979. A measure of control flow complexity in program text. *IEEE Transactions on Software Engineering*, **5**(1):45–50.

[185] Dunsmore H. E., and Gannon J. D., 1980. Analysis of the effects of programming factors on programming effort. *Journal of Systems and Software*, **1**(2):141–153.

[186] Shooman M. L., 1983. *Software Engineering*. New York, NY: McGraw Hill.

[187] Zolnowski J. C., and Simmons D. B., 1981. Taking the measure of program complexity. *Proceedings of the AFIPS National Computer Conference*, Chicago, Illinois, USA, 329–336.

[188] Myers G. J., 1977. An extension to the cyclomatic measure of program complexity. *SIGPLAN Notices*, **12**(10):61–64.

[189] Hansen W. J., 1978. Measurement of program complexity by the pair (cyclomatic number, operator count). *ACM SIGPLAN Notices*, **13**(3):29–33.

[190] Oviedo E. I., 1980. Control flow, data flow, and program complexity. *Proceedings of the Fourth International IEEE Computer Software and Applications Conference (COMPSAC 1980)*, Chicago, Illinois, USA, 146–152.

[191] Card D. N., and Glass R. L., 1990. *Measuring Software Design Quality*. Englewood Cliffs, NJ: Prentice Hall.

[192] Lo B., 1992. *Syntactical Construct Based APAR Projection* (Technical Report). San Jose, CA: IBM Santa Teresa Research Laboratory.

[193] Myers G. J., 1976. *Software Reliability: Principles and Practices*. New York, NY: John Wiley and Sons.

[194] Musa J. D., 1999. *Software Reliability Engineering*. New York, NY: McGraw Hill.

[195] Pham H., 2000. *Software Reliability*. Singapore: Springer Verlag.

[196] Jelinski Z., and Moranda P. B., 1972. Software reliability research. In Freiberger W. (Ed.), *Statistical Computer Performance Evaluation* (pp. 465–484). New York, NY: Academic Press.

[197] Schick G. J., and Wolverton R. W., 1978. An analysis of competing software reliability analysis models. *IEEE Transactions on Software Engineering*, **4**(2):104–120.

[198] Moranda P. B., 1979. An error detection model for application during software development. *IEEE Transactions on Reliability*, **28**(5):325–329.

[199] Goel A. L., and Okumoto K., 1979. Time dependent error detection rate model for software and other performance measures. *IEEE Transactions on Reliability*, **28**(3):206–211.

[200] Littlewood B., 1979. Software reliability model for modular program structure. *IEEE Transactions on Reliability*, **28**(3):241–246.

[201] Sukert A. N., 1979. Empirical validation of three software error prediction models. *IEEE Transactions on Reliability*, **28**(3):199–205.

[202] Coutinho J. S., 1973. Software reliability growth. *Proceedings of the IEEE Symposium on Computer Software Reliability*, New York, NY, USA, 58–64.

[203] Wall J. K., and Ferguson P. A., 1977. Pragmatic software reliability prediction. *Proceedings of the Annual Reliability and Maintainability Symposium*, Piscataway, New Jersey, USA, 485–488.

[204] Huang X. Z., 1984. The hypergeometric distribution model for predicting the reliability of software. *Microelectronics and Reliability*, **24**(1):11–20.

[205] Musa J. D., Iannino A., and Okumoto K., 1987. *Software Reliability: Measurement, Prediction, and Application*. New York, NY: McGraw Hill.

[206] Ohba M., 1984. Software reliability analysis models. *IBM Journal of Research and Development*, **21**(4):428–443.

[207] Yamada S., Ohba M., and Osaki S., 1983. S shaped reliability growth modeling for software error prediction. *IEEE Transactions on Reliability*, **32**(5):475–478.

[208] Thayer C. H., 1958. Automation and the problems of management. *Vital Speeches of the Day*, **25**(4):121–125.

[209] Hardin K., 1960. Computer automation, work environment, and employee satisfaction: a case study. *Industrial and Labor Relations Review*, **13**(4):559–567.

[210] Kaufman S., 1966. The IBM information retrieval center (ITIRC): system techniques and applications. *Proceedings of the 21st National Conference for the Association for Computing Machinery*, New York, NY, USA, 505–512.

[211] Lucas H. C., 1973. User reactions and the management of information services. *Management Informatics*, **2**(4):165–162.

[212] Bailey J. E., and Pearson S. W., 1983. Development of a tool for measuring and analyzing computer user satisfaction. *Management Science*, **29**(5):530–545.

[213] Lucas H. C., 1974. Measuring employee reactions to computer operations. *Sloan Management Review*, **15**(3):59–67.

[214] Maish A. M., 1979. A user's behavior toward his MIS. *MIS Quarterly*, **3**(1):39–52.

[215] Lyons M. L., 1980. Measuring user satisfaction: the semantic differential technique. *Proceedings of the 17th Annual Conference on Computer Personnel Research*, Miami, Florida, USA, 79–87.

[216] Pearson S. W., and Bailey J. E., 1980. Measurement of computer user satisfaction. *ACM SIGMETRICS Performance Evaluation Review*, **9**(1):9–68.

[217] Walsh M. D., 1982. Evaluating user satisfaction. *Proceedings of the 10th Annual ACM SIGUCCS Conference on User Services*, Chicago, Illinois, USA, 87–95.

[218] Ives B., Olson M. H., and Baroudi J. J., 1983. The measurement of user information satisfaction. *Communications of the ACM*, **26**(10):785–793.

[219] Joshi K., Perkins W. C., and Bostrom R. P., 1986. Some new factors influencing user information satisfaction: implications for systems professionals. *Proceedings of the 22nd Annual Computer Personnel Research Conference*, Calgary, Canada, 27–42.

[220] Baroudi J. J., and Orlikowski W. J., 1988. A short form measure of user information satisfaction: a psychometric evaluation and notes on use. *Journal of Management Information Systems*, **4**(4):44–59.

[221] Doll W. J., and Torkzadeh G., 1988. The measurement of end user computing satisfaction. *MIS Quarterly*, **12**(2):258–274.

[222] Kekre S., Krishnan M. S., and Srinivasan K., 1995. Drivers of customer satisfaction in software products: implications for design and service support. *Management Science*, **41**(9):1456–1470.

[223] Lindroos K., 1997. Use quality and the world wide web. *Information and Software Technology*, **39**(12):827–836.

[224] Dreze X., and Zufryden F., 1997. Testing web site design and promotional content. *Journal of Advertising Research*, **37**(2):77–91.

[225] Selz D., and Schubert P., 1997. Web assessment: a model for the evaluation and the assessment of successful electronic commerce applications. *Electronic Markets*, **7**(3):46–48.

[226] Chen Q., and Wells W. D., 1999. Attitude toward the site. *Journal of Advertising Research*, **39**(5): 27–37.

[227] Szymanski D. M., and Hise R. T., 2000. E-satisfaction: an initial examination. *Journal of Retailing*, **76**(3):309–322.

[228] Barnes S. J., and Vidgen R. T., 2000. Webqual: an exploration of web site quality. *Proceedings of the Eighth European Conference on Information Systems, Vienna, Austria*, 298–305.

[229] Barnes S. J., and Vidgen R. T., 2001. An evaluation of cyber bookshops: the webqual method. *International Journal of Electronic Commerce*, **6**(1):11–30.

[230] Cho N., and Park S., 2001. Development of electronic commerce user consumer satisfaction index (ECUSI) for internet shopping. *Industrial Management and Data Systems*, **101**(8/9):400–405.

[231] Yoo B., and Donthu N., 2001. Developing a scale to measure the perceived quality of an internet shopping site (sitequal). *Quarterly Journal of Electronic Commerce*, **2**(1):31–45.

[232] Janda S., Trocchia P. J., and Gwinner K. P., 2002. Consumer perceptions of internet retail service quality. *International Journal of Service Industry Management*, **13**(5):412–433.

[233] McKinney V., Yoon K., and Zahedi F., 2002. The measurement of web customer satisfaction: an expectation and disconfirmation approach. *Information Systems Research*, **13**(3):296–315.

[234] Wolfinbarger M., and Gilly M. C., 2003. Etailq: dimensionalizing, measuring, and predicting etail quality. *Journal of Retailing*, **79**(3):183–198.

[235] Takeuchi H., and Nonaka I., 1986. The new product development game. *Harvard Business Review*, **64**(1):137–146.

[236] Sulack R. A., Lindner R. J., and Dietz D. N., 1989. A new development rhythm for AS/400 software. *IBM Systems Journal*, **28**(3):386–406.

[237] Cusumano M. A., and 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.

[238] Cusumano M. A., and Yoffie D. B., 1998. *Competing on Internet Time: Lessons from Netscape and Its Battle with Microsoft*. New York, NY: The Free Press.

[239] MacCormack A., 1998. *Managing Adaptation: An Empirical Study of Product Development in Rapidly Changing Environments*. Unpublished doctoral dissertation, Harvard University, Boston, MA, United States.

[240] Thomke S., and Reinertsen D., 1998. Agile product development: managing development flexibility in uncertain environments. *California Management Review*, **41**(1):8–30.

[241] Fichman R. G., and Moses S. A., 1999. An incremental process for software implementation. *Sloan Management Review*, **40**(2):39–52.

[242] Reifer D. J., 2003. The business case for agile methods/extreme programming (XP). *Proceedings of the Seventh Annual PSM Users Group Conference*, Keystone, Colorado, USA, 1–30.

[243] Johnson M., 2003. *Agile Methodologies: Survey Results*. Victoria, Australia: Shine Technologies.

[244] Prewitt E., 2004. The agile 100. *CIO Magazine*, **17**(21):4–7.

[245] Digital Focus. 2006. *Agile 2006 Survey: Results and Analysis*. Herndon, VA: Author.

[246] Version One. 2006. *The State of Agile Development*. Apharetta, GA: Author.

[247] Ambler S. W., 2006. *Agile Adoption Rate Survey: March 2006*. Retrieved on September 17, 2006, from http://www.ambysoft.com/downloads/surveys/AgileAdoptionRates.ppt.

[248] Ambler S. W., 2007. *Agile Adoption Survey: March 2007*. Retrieved on July 23, 2007, from http://www.ambysoft.com/downloads/surveys/AgileAdoption2007.ppt.

[249] Rico D. F., 2007. *Effects of Agile Methods on Website Quality for Electronic Commerce*. Unpublished doctoral dissertation, University of Maryland University College, Adelphi, MD, United States.

[250] Rico D. F., Sayani H. H., Stewart J. J., and Field R. F., 2007. A model for measuring agile methods and website quality. *TickIT International*, **9**(3):3–15.

[251] Rico D. F., 2007. Effects of agile methods on electronic commerce: Do they improve website quality? *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS 2007)*, Waikaloa, Big Island, Hawaii.

[252] Rico D. F., 2008. Effects of agile methods website quality for electronic commerce. *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*, Waikaloa, Big Island, Hawaii.