# 

# 

# 

12 January 2016

**A Very Brief History of Computing,  
 1948-2015**

Professor Martyn Thomas

In my first lecture, *Should We Trust Computers?,* I described the critically important role of software based systems today. I quoted some research results that showed that software is typically put into service with more than ten defects in every thousand lines of program source code (*KLoC*), and explained that some of the systems we all depend on in various ways contain many million lines, implying many thousands of defects.

I explained that *testing* a software system can only ever find a tiny fraction of these errors, and gave an example to show why that is inevitably true. Testing generally finds the most obvious errors and therefore the more testing that you do, the more obscure will be the errors that remain. That is the fundamental explanation of how it can be true that most of the software that we depend on does actually work well enough almost all of the time, and yet none of it seems to be secure against even teenage hackers – as TalkTalk appears to have found out recently[[1]](#endnote-1).

In this second lecture, I want to show how we got into a situation where the hardware that we use is mostly superbly engineered and ultrareliable, whereas the software is often neither. I shall do that by sketching the history of computing at lightning speed, drawing out some things that seem to me to be important to remember when we come to consider the future – because although computing has changed most of society beyond recognition in its first seven decades, there are far more changes to come, and the more we understand the past, the better we shall be able to manage the future.

A lightening sketch of the history of hardware and software has to be selective. I therefore apologise in advance for omitting many major developments and people who have made seminal contributions to computer science and software or hardware engineering (some of whom may be in the hall tonight, as our history is remarkably recent!). I will, however, draw attention to the role of UK universities, UK companies and UK Government research establishments because I constantly find that people talk and write as if modern computing was born and raised in the USA, which is far from the truth. The UK’s role is often understated – even and in my view most regrettably in the Science Museum in London. UK computer scientists and engineers have been world class and often world leading throughout the history of computing although, too often, UK institutions have not had the vision and the ambition to make the investments that could have led to major commercial success.

**Beginnings**

The modern computer age began a little after 11 am on Monday, June 21 1948, in Manchester, England. I shall explain shortly why I have taken that moment as the true beginning.

Gresham College was founded 350 years earlier that this, and has played its own part in advancing the science and art of calculation. The first Gresham Professor of Geometry, Henry Briggs, is credited with changing John Napier’s original logarithms into common (base 10) logarithms to simplify their use.

Base 10 logarithms were still in common use in the 1960s, in schools and in engineering offices, alongside their mechanical equivalent, the slide rule (which was invented in 1630 by Richard Delamain and then, independently reinvented in 1632 by William Oughtred).

Calculation has been very important since the invention of arithmetic, particularly for building, for trade and for several centuries, for navigation at sea.

There have been very many mechanical calculators. In 1623, Wilhelm Schickard built a machine that added and subtracted automatically (and multiplied and divided partly automatically). In the 1640s, Blaise Pascal built a small machine that could add and subtract. In 1673, Gottfried Wilhelm Leibniz invented the Leibniz Wheel that could add, subtract, multiply and divide automatically. And most famously perhaps, in the 19th Century, Charles Babbage invented his Difference Engine and then the Analytical Engine with its input device inspired by the punched cards that the weaver and engineer Joseph Marie Jacquard had recently invented to control his remarkable automatic loom[[2]](#endnote-2).

The current Gresham Professor of Geometry, Raymond Flood, will present a lecture, on Charles Babbage and his friend and colleague, the mathematician Ada, Countess of Lovelace, next Tuesday, January 19 2016 at 1pm in Barnards’s Inn Hall. 2015 was the 200th anniversary of Ada Lovelace’s birth, and a recent lecture on *The Scientific Life of Ada Lovelace,* by Professor Ursula Martin, is on the Gresham College website.

In the early 20th Century, the term “computer” meant a person who performed calculations, and there were thousands of them, employed in commerce and in industry, often using mechanical calculating machines. Mechanical calculating machines were in common use until surprisingly recently; when I first worked at Bath University in 1976, the Department of Mathematics had a room[[3]](#endnote-3) fitted out with Brunsviga mechanical calculators that were used to teach classes of undergraduates, even though the university had owned a computer for several years. (At this time, one computer would typically provide the main computing services for a whole university; departmental computers had only appeared recently, and the personal computer arrived a few years later).

But I get ahead of myself.

In the 1930s, two powerful ideas came together. The first of these ideas was that electronic circuits could be used to replace mechanical calculators, with a great increase in speed. The second idea was the mathematical theory of computation, and most particularly the work of the British mathematician Alan Turing who showed that a machine that can obey a very small set of instructions and that has a memory is sufficient to perform all possible calculations. (Professor Flood will lecture on Alan Turing on Tuesday, 19 April 2016 at 1:00pm).P

Together, these ideas led to the goal of a universal electronic computer – a machine that could be programmed to perform any possible computation and to do so at high speed.

Many teams worked towards this goal, in America, the UK and elsewhere, accelerated by the priorities of World War II and the intellectual, physical and financial resources that could be committed to the task as a key part of the war effort.

As Ada Lovelace already understood, any universal computer must have a memory – it is not enough to just have circuits for performing calculations. There were several competing approaches to building an electronic memory and one of them was invented in 1944 by Professor F C Williams and a young graduate mathematician, Tom Kilburn, (who had been instructed in 1942 to take a crash course on electronics and to report to Williams at the Telecommunications Research Establishment at Malvern, where wartime work on radar was being carried out). This used a well-known characteristic of a cathode-ray tube or CRT (the sort of tube that was used in radars and in televisions before the flat screen was developed). The inside front of a CRT has a fluorescent coating which glows when it is energised by a beam of electrons, and this glow persists until the charge dissipates, over a period of a second or so. Williams invented a way of reading this charge by fixing a collector plate to the outside of the tube. It was then possible to create a data store by writing a pattern of dots on the screen, reading them back, and using the output to drive a circuit that re-energised the pattern of dots for as long as required.

Williams returned to Manchester University in 1946 and Tom Kilburn went with him, to help him to build a computer, based, at least in part, on a design by the Hungarian, John von Neumann (which, in turn was based on Turing’s ideas). The new computer was intended as a test bed for the Williams Storage Tube and it was known as the Small Scale Experimental Machine (SSEM) and later as the “Baby”. Its circuits were constructed from about 500 thermionic valves, and the Storage Tube had a capacity of 1024 binary digits, arranged as 32 words each of 32 bits of Random Access Memory. A second storage tube provided a 32-bit register called A (for “accumulator”) and a third (called C for “Control) held the address of the current instruction. Data was input through a panel of switches and the computer logic could obey 7 different instructions (negate and store in A, subtract from A, store, skip the next instruction if A is negative, branch to an address, branch to a relative address, halt). These instructions can be shown to be sufficient to implement any possible program! [Because they are sufficient to implement a Universal Turing Machine].

In 1974, Williams recalled the first successful program run on the Baby.

*“A program was laboriously inserted and the start switch pressed. Immediately the spots on the display tube entered a mad dance. In early trials it was a dance of death leading to no useful result, and what was even worse, without yielding any clue as to what was wrong. But one day it stopped, and there, shining brightly in the expected place, was the expected answer. It was a moment to remember. This was in June 1948, and nothing was ever the same again.”[[4]](#endnote-4)*

*Manchester SSEM.tiff*

There had been calculators before this, both mechanical and electronic (e.g. Howard Aiken’s ASCC). There had been computers (notably Max Newman’s COLOSSUS at Bletchley Park and Eckert and Mauchley’s ENIAC), but the SSEM, The Baby, was the first that was truly universal in that it could perform any calculation simply by entering a new program into the store. This why I say that the modern computer age began a little after 11 am on Monday, June 21 1948, in Manchester, England.

Things moved astonishingly quickly. The UK Government (through Sir Benjamin Lockspeiser, Chief Scientist at the Ministry of Supply) instructed Ferranti Limited in October 1948 to *“construct an electronic calculating machine to the instructions of Professor F C Williams”.* This commercial machine was completed in February 1951. By the end of 1956, Manchester University had filed 81 computer patents.

At the same time, Alan Turing was working on the development of ACE (Automatic Computing Engine) at the National Physical Laboratory. Turing had proposed this in 1946, giving NPL a full description including circuit diagrams. He estimated the cost at £11,200 and the Alan Turing Archive contains a letter dated September 1950, from Sir Benjamin Lockspeiser, recording that the Treasury had agreed the previous year that English Electric Limited should be awarded a contract for £15,000 per annum to assist with developing the machine. The letter says that the “first stage (or pilot)” ACE is now substantially complete and requests that the contract is extended at £15,000 a year to allow the machine to be further developed and tested. Sir Benjamin writes that

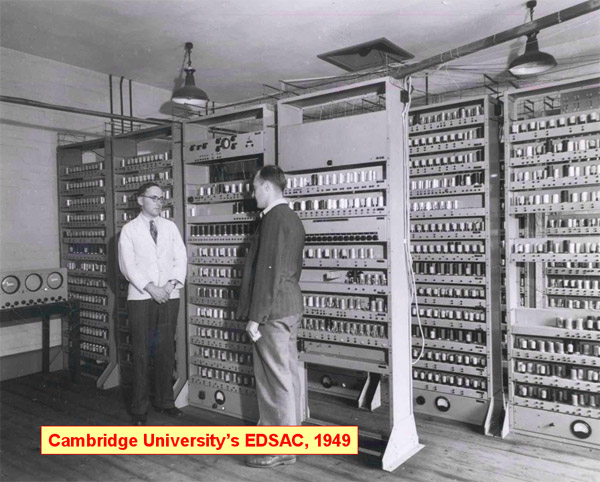
*… work on the development of high-speed calculating machines is also proceeding at other centres (Government establishments and universities). It [unreadable] considerable effort and the machines are of such [unreadable] there is not likely to be a need for more than a few of them in the country.*

In fact, more than 50 of the production version of ACE (named DEUCE) were produced[[5]](#endnote-5).

Meanwhile, Professor Maurice Wilkes had been developing EDSAC (Electronic Delay Storage Automatic Calculator) in the Cambridge University Mathematical Laboratory. In early 1947, Wilkes had attended a talk by Turing about his design for ACE (Kilburn was also present) and the Work on EDSAC started around the same time. According to Wilkes,

*“the object of the EDSAC project was “to construct a workmanlike computer that would give a taste of what a stored-program computer was like to work with, and so to allow work on program writing to start”*.[[6]](#endnote-6)

EDSAC ran its first program in May 1949 – it was the world’s first stored-program computer with fully integrated input and output. EDSAC also introduced the bootstrap loader and libraries of subroutines on paper tape.



Just as Wilkes was starting work on EDSAC, a group of managers from the catering company Joe Lyons was touring the USA looking for new business ideas. They heard about computers and discovered that pioneering work was going on in Cambridge. On their return to the UK, they visited Wilkes and decided that they should build their own computer, with a design based closely on EDSAC, to run their payroll and the scheduling and administration of their bakery. They provided some additional funding to Wilkes’s team, lent them a technician (Ernest Lenaerts) and when EDSAC was seen to work, they hired as their Chief Engineer, John Pinkerton, who had worked on radar in World War II and who was just completing a PhD in the Cavendish Laboratory. The new computer was called Lyons Electronic Office (LEO).

Pinkerton’s challenge was to improve on EDSAC in many ways. LEO had to have much faster input output, to meet the time constraints on the payroll. It had to be reliable enough to meet the needs of a business. And it needed to handle commercial calculations efficiently, such as converting between binary and decimal for calculations involving pounds, shillings and pence. That John Pinkerton achieved these objectives at all is remarkable, yet towards the end of 1953, LEO took over part of Lyons’ payroll and it was so successful that Lyons set up a subsidiary company to build computers for sale to other companies. By 1962 Pinkerton had designed LEO III, a completely new machine that incorporated Wilkes’ latest research invention of microprogramming, an interrupt system that allowed multiprogramming, and special instructions for number conversion and commercial arithmetic. 61 LEO IIIs were sold and there is little doubt that if LEO had had access to sufficient investment capital, it could have become a leading supplier of commercial computers. Unfortunately, Lyons agreed a merger of LEO Computers Ltd with English Electric and by the end of the 1960s, with strong Government encouragement, all the significant English computer manufacturers had merged into International Computers Limited, which was developing a new range of computers (the 2900 series) based on the latest Manchester University research (MU5) and the LEO computer range had vanished.

LEO.tiff

In the 1950s, there were many groups developing computing equipment, in the UK and overseas. Andrew Booth and Kathleen Britton, at Birkbeck University, London, developed magnetic drum storage, the floppy disk and several computers (APEXC, MAC and M2).

In 1951, in the USA, Eckert and Mauchley built the first Univac computer, Univac 1, for business use. The US Government gave IBM a $350 million military contract to develop computers. IBM’s 701 data processing system appeared in 1952 and IBM’s 650 computer in 1953.

In 1953, Richard Grimsdale and Douglas Webb (part of Tom Kilburn’s group at Manchester University, produced a prototype Transistor Computer, replacing unreliable thermionic valves with much more reliable semiconductors. A commercial version of the transistor computer was built by Metropolitan-Vickers as the MV950.

The technology for producing single transistors and diodes quickly led to integrated circuits, where many components could be packed on to a single silicon chip. Advances in physics and manufacturing continually increased the number of components on a chip and, as a result, increased the speed of circuits whilst reducing their cost.

In 1965, Gordon Moore, of Fairchild Semiconductors (and later of Intel) published a paper[[7]](#endnote-7) called “Cramming More Components onto Integrated Circuits” in which he predicted that the number of transistors on a single chip would continue to double every year, which would dramatically reduce the cost of components.

Moore's Law fig 3.tiff

His prediction, which was later called Moore’s Law, proved remarkably accurate:

Moore's Law actual.tiff

Other developments in computer hardware – such as associated memory, array processors, Multics, the Manchester ATLAS computer, Ethernet and the Cambridge Ring – must be left for other lectures, as must the development of networking from ARPANET to the World Wide Web.

Internet history.tiff

**Software before *the First Software Crisis***

Computers were of little use without programs, of course.

The Manchester Baby was programmed in binary, using the switches on the front panel, but programs for later machines were punched on paper tape using standard tele-printer codes. This had the advantage that programs could be prepared offline, tested on the computer, and then amended on the tape without having to re-type the entire program. Early computers were programmed as sequences of the binary instructions that the hardware could execute (“machine code”) but programming in machine code was error-prone so instructions were assigned mnemonics (such as “ST” for “Store this value at the following address”). A small program would be entered into the computer to read the paper tape, decode the mnemonics into machine code and to assemble the program in the computer memory. These programs became called assemblers, and the mnemonic machine code was referred to as “assembly language”.

An important innovation of EDSAC was a library of subroutines. David Wheeler, a mathematics student at Cambridge University, invented the closed subroutine in 1949 and the technique of jumping to another program which would perform some calculation and then return to the instruction following the jump became known as the “Wheeler Jump”.

Subroutines saved a lot of programming time, and libraries of subroutines grew into “Operating Systems” that provided all the basic functions that most programmers would need again and again (such as input-output, managing disks and tapes, and signalling to the computer operator).

Features were added to assembly languages to simplify programming and these enhanced languages were called “autocodes” and the programs that translated autocodes became known as “compilers”. The first autocode compiler, for the Manchester Mark 1, was developed by Alick Glennie.

I have already mentioned the Lyons Electronic Office computer developed by John Pinkerton and based on EDSAC. The software to manage the bakery was written by a team led by David Caminer that included Mary Coombs (probably the world’s first woman to write business software) and Frank Land.

The A-0 system compiler was written by US Naval Captain Grace Hopper in 1951 and 1952 for the UNIVAC I. Grace Hopper played a significant part in the development of COBOL.

It was quickly recognised that programming in assembly language and autocodes took too long and led to too many errors; programmers needed languages that focused on the problem to be solved, rather than on the detailed design of a computer’s hardware. One of the first “higher-level languages” was FORTRAN (FORmula TRANslator) developed by John Backus in 1957 for IBM (the compiler took 17 person-years of effort). FORTRAN was a major advance over assembler, and became very widely used; it could be compiled into programs that ran very efficiently, although not as efficiently as assembler, and this was very important. However, FORTRAN lacked the features that would prove necessary for structured programming and the secure development of large systems.

The programming language that has arguably had the greatest influence on language design and programming is Algol 60[[8]](#endnote-8). It is a small, elegant language, whose syntax was defined in the referenced report, which I urge you to read and admire. The Backus Naur notation (BNF) used for the syntax is itself a work of beauty and very influential.

Algol included several seminal concepts, foremost of which are *recursion* (the ability of a function or procedure to call itself), and *strong data types* (requiring that variables should have a stated data type, such as integer, boolean or character, and that only operations that are defined on this data type should be permitted in the program). P

The computer science that supports the specification, design and analysis of programming languages advanced rapidly during the 1960s, with notable work by Donald Knuth[[9]](#endnote-9), Tony Brooker and J M Foster (at RRE). Hundreds of new computer languages were designed in the 1960s and 1970s.

By 1960, computer use was already growing very rapidly worldwide and by 1968 there were at least 10,000 computers installed in Europe alone. The new applications needed much more powerful software, and software systems became much larger. The Operating System that was designed for the new 360 range of IBM computers, OS/360, cost IBM over $50m per year during development and at least 5000 person years of effort. The development of OS/360 was led by Fred Brooks, who described his “million dollar mistake” of letting the developers design the system architecture, in his classic book *The Mythical Man Month[[10]](#endnote-10).*

OS/360 was far from the only software project to suffer failures, cost overruns and delays and the NATO Science Council decided to organise two expert conferences (in Garmisch, Germany, 7-11 October 1968 and in Rome, Italy 27-31 October 1969) to address the emerging software crisis. The two conference proceedings were published in *Software Engineering,* edited by Peter Naur and Brian Randell and *Software Engineering Techniques,* edited by John Buxton and Brian Randell. Both reports are still extremely interesting and Brian Randell (who is now an Emeritus Professor at Newcastle University) has made them available online.[[11]](#endnote-11)

The experts’ diagnoses of the problems were accurate but largely ignored, as were their proposed solutions. For example, E.S Lowry, from IBM, is quoted as saying:

*“Any significant advance in the programming art is sure to involve very extensive automated analyses of programs. … … Doing thorough analyses of programs is a big job. … It requires a programming language which is susceptible to analysis. I think other programming languages will head either to the junk pile or to the repair shop for overhaul, or they will not be effective tools for the production of large programs.”*

Tony Hoare was at the 1969 Rome conference and the report shows that he understood the limitations of testing that I illustrated in my first Gresham lecture. He is quoted as saying:

*“One can construct convincing proofs quite readily of the ultimate futility of exhaustive testing of a program and even of testing by sampling. So how can one proceed? The role of testing, in theory, is to establish the base propositions of an inductive proof. You should convince yourself, or other people, as firmly as possible that if the program works a certain number of times on specified data, then it will always work on any data. This can be done by an inductive approach to the proof. Testing of the base cases could sometimes be automated. At present, this is mainly theory; note that the tests have to be designed at the same time as the program and the associated proof is a vital part of the documentation. This area of theoretical work seems to show a possibility of practical results, though proving correctness is a laborious and expensive process. Perhaps it is not a luxury for certain crucial areas of a program.*

Following a comment by Perlis in defence of testing, Dijkstra remarked: *“Testing shows the presence, not the absence of bugs”.* This truth remains unrecognised by most programmers, even though the intervening 46 years have demonstrated it again and again. Dijkstra’s many writings for his students are online and are entertaining, insightful and certainly still repay study[[12]](#endnote-12).

Alan Turing had recognised that program analysis was essential as long ago as 1949, saying *“How can one check a routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions that can be checked individually, and from which the correctness of the whole program easily follows”.*

By the 1970s, the need for greater rigour in software development was widely recognised.

* IBM were preparing a mathematically formal definition of their programming language, PL/I (in VDM)
* Edsger Dijkstra had introduced “Structured Programming”, based on a theorem by Jacopini and Bohm and published his famous letter “*Go-to considered harmful*”.
* Tony Hoare had published *An Axiomatic Basis for Computer Programming*, introducing the practical use of preconditions, postconditions, invariants and formal proof.
* Ole-Johan Dahl and Kristen Nygaard had invented object-oriented programming in their language SIMULA.

The best summary of the state of knowledge in 1970 is *Structured Programming*, by Dahl, Dijkstra and Hoare (1972), which should still be part of any professional programmer’s education.

**Complexity** is the main problem faced by software developers. In his 1972 Turing Award lecture, *The Humble Programmer,* Dijkstra said:

*“we [must] confine ourselves to the design and implementation of intellectually manageable programs. … If someone fears that this restriction is so severe that we cannot live with it, I can reassure him: the class of intellectually manageable programs is still sufficiently rich to contain very many realistic programs for any problem capable of algorithmic solution.”*

During the 1960s, another project took shape that had an enormous influence, though it was not the commercial success that had been hoped. This was the collaboration between MIT, General Electric and Bell Labs to develop a highly reliable and secure computer hardware and software system, *Multics.* The many innovative features of Multics deserve a lecture on their own, and they influenced many subsequent systems, but there is one particular legacy that I have to include here.

Dennis Richie and Ken Thompson were part of the Bell Labs team on Multics and, frustrated by his lack of access to computer facilities, Thompson found an underutilised machine and wrote a very simple operating system for it in assembler. His system aimed to provide many of the facilities of Multics but for a single user and far more efficiently. He called his system *Unix*.

Richie was a language designer who developed an Algol-like language for efficient systems programming. This was the programming language *C,* and in 1972 Richie and Thompson re-implemented Unix in C so that they could move it to other Digital PDP computers easily. The original Unix is a model of elegance and architectural simplicity. It is worth downloading the early source[[13]](#endnote-13) and studying it. Unix, of course, has become the model for our most important operating systems, most significantly represented by Linux and Apple’s OSX.

There were many advances in software engineering throughout the 1970s and 1980s, from which I would highlight

*Advances in Structured methods*

* Top-down functional design, stepwise refinement
* Data-led design (in particular Jackson Structured Programming)

*Wider use of methods based on computer science*

* VDM (Jones), Z (Abrial)

*Advances in software development processes*

* Mythical Man Month (Fred Brooks), Harlan Mills[[14]](#endnote-14)
* Software Engineering Economics (Barry Boehm)
* Strategies for Software Engineering[[15]](#endnote-15) (Martyn Ould)

In 1973, researchers at the Xerox research centre in Palo Alto (Xerox PARC) developed a model for human/computer interfaces based on windows, icons, the mouse and pointers (WIMP). This was introduced into the mainstream of computing in 1984 with the Apple Macintosh and has become the standard method for using computers.

In 1982, the Japanese Industry Ministry MITI launched an ambitious *Fifth Generation Computer Programme* (FGPS). The first four generations of computers were based on valves, then transistors, ICs and VLSI microprocessors. The fifth was to have massively parallel hardware and Artificial intelligence software – one project was to build a hand-held device that you could take under your car when you had a problem, to discuss the symptoms you could see and receive advice on what to do next.

This FGPS was open to all countries and attracted visits from around the world. It frightened the UK, US and EU into competitive research: in the UK it led directly to the £350m *Alvey* research programme into software engineering, AI, HCI and VLSI design. My own company, *Praxis,* worked with International Computers Limited on computer-aided software engineering tools and workflow modelling and with MoD on VLSI design tools for the Electronic Logic Language, ELLA. FGPS was ended after 10 years, having greatly increased the number of skilled staff in the Japanese computer industry.

The increase in the ratio of computer performance to price continued to double every one to two years throughout the 1980s and 1990s, just as Moore’s Law had predicted. This drove computers into more and more application areas, with exponential growth in the use of personal computers and increasing numbers of real-time control systems. Once again, software engineering failed to keep up with hardware engineering: more and more programmers were recruited to work on the new applications, but personal computers lacked even the support tools that had existed on mainframes (and the new generation of programmers did not have the experience that mainframe and minicomputer programmers had acquired through years of successes and failures). Unsurprisingly, projects continued to overrun and to fail.

In the UK, the main public sector purchasers of software (the Public Purchasers’ Group, PPG) collaborated to establish some standards, initially published as *Software Tools for Application to Real Time Systems* (STARTS) and the use of these methods and tools and the quality management standard BS 5750 were introduced into PPG purchasing contracts. The National Computing Centre then led a project to develop a similar guide for business systems (IT STARTS) and BS 5750 became an ISO standard (ISO 9001) and compliance was required by more and more UK customers. By the early 1990s, most UK software houses were certified to comply with ISO 9001.

In the USA, the failure of IT projects for the Department of Defense led to the setting up of a Software Engineering Institute (SEI) at Carnegie-Mellon University (CMU); the SEI was commissioned to develop a method that would enable DoD to assess the competence of defense contractors. Watts Humphrey led the SEI development of the *Capability Maturity Model* (CMM)[[16]](#endnote-16).

The CMM assessed organisations against five levels of maturity of their software development capability:

*Level 1: Initial*

Software development processes are ad hoc and unstable

*Level 2: Repeatable*

The organisation can (usually) repeat processes successfully once they have worked once

*Level 3: Defined*

The development processes are a documented company standard. Staff are trained in the processes.

*Level 4: Managed*

Processes are measured and the measurements are used in project mgt

*Level 5: Optimising*

Continuous improvement of the development processes has become routine.

Most Defense Contractors were found to be at Level 1, with ad-hoc processes.

A few companies in the USA, UK and elsewhere adopted or continued to use mathematically formal methods but this was rare. Almost all customers were content to issue software contracts with statements of requirements that were informal, incomplete, contradictory and largely unenforceable, and most software companies were happy to bid for these contracts and to make profits from the inevitable “change requests” that arose when the deficiencies in the requirements became clear. Unsurprisingly, customer dissatisfaction with the software industry grew, but there were great benefits to be gained from using computers in the new application areas that opened up as hardware prices fell and computing power increased, even if the software was late, expensive or unreliable, so software companies continued to flourish without adopting better software engineering methods.

Except in a few safety-critical areas such as air traffic control, the nuclear industry and railway signalling, speed to market was considered far more important than good software engineering.

In 1995, a US consultancy, Standish Group, published their first survey and report on software projects. In a survey of 8,380 application projects, **31.1%** were cancelled before delivery and only **16.2%** were on time, on budget and met the customer’s stated requirements. The average cost overrun was 189%, the average time overrun was 222%, and the average percentage of the required features that were actually delivered was 61% of those originally specified. For every 100 projects that started, there were 94 restarts, costing extra time and money (some projects had to be restarted several times). The report of this survey, which Standish Group called *The Chaos Report* can be found online[[17]](#endnote-17).

Unfortunately there is not time in this lecture to cover the developments in computer communications systems, from ARPANET to the World-Wide Web: these will have to wait for a future lecture.

By the end of the 1990s, software development was often good enough for routine projects, but it was mainly a practical craft that depended on the skills and experience of individuals rather than an engineering profession that could be relied on to develop systems, to provide strong evidence that they would be fit for their intended purpose, and to accept liability for defects.

2000 came and went, with billions of pounds spent on repairing the Y2K date-related defects that programmers had left in their software. There is a myth that the “millennium bug” was never a problem, but the truth is that many thousands of critical errors were found and corrected, and that many systems did fail (and some failures led to the demise of organisations). Many companies discovered that they did not know what their critical systems were, or where to find their latest complete source code. Many suppliers defrauded their customers by insisting on wholly unnecessary upgrades before they would supply the certification of Y2K compliance that auditors, insurers and supply chain customers required.

Moore’s Law continued to predict the falling cost of computing, which led to tablet computing, smart phones, apps, and systems embedded in everything from cars to televisions, washing machines and light-bulbs. Increasingly, these embedded systems were also online, leading to the growing Internet of Things that I shall discuss in a later lecture.

Of course, all this required many more programmers, and there were plentiful jobs for people with little training in software engineering, writing software with little concern for cybersecurity and almost entirely dependent on testing to show that their work was good enough – decades after computer scientists and software engineers had shown that testing alone would always be inadequate.

Today we have millions of programmers worldwide[[18]](#endnote-18):

Developer numbers from IDC.tiff

So now we have a third software crisis. The first was in the 1960s – mainframe software errors and overruns – and it led to the NATO conferences and the increased use of structured methods. The second was in the 1980s – overruns and failures in real-time systems, military systems, and large corporate IT systems and it led to the increased use of quality management systems, CASE tools, and (for critical systems) mathematically formal methods.

The third software crisis is with us today – represented by problems of Cybersecurity, vulnerabilities in critical infrastructure, failures in our increasingly complex banking systems, increased online crime and fraud, and overruns and cancellation of major IT projects in Government, industry and commerce.

The solution has to be that software engineering replaces test-and-fix, but this remains unlikely to happen quickly enough.

Tony Hoare was once asked why software development had not become an engineering discipline in the way that other professions had. He replied:

*“We are like the barber-surgeons of earlier ages, who prided themselves on the sharpness of their knives and the speed with which they dispatched their duty -- either shaving a beard or amputating a limb.*

*Imagine the dismay with which they greeted some ivory-towered academic who told them that the practice of surgery should be based on a long and detailed study of human anatomy, on familiarity with surgical procedures pioneered by great doctors of the past, and that it should be carried out only in a strictly controlled bug-free environment, far removed from the hair and dust of the normal barber's shop*.”

Now we have a Livery Company, although centuries later than the Barber-Surgeons created theirs, perhaps we might yet become an engineering profession.

© Professor Martyn Thomas, 2016

1. http://www.theinquirer.net/inquirer/news/2431728/talktalk-ddos-hack-leaves-four-million-customers-at-risk *accessed 21 december 2015* [↑](#endnote-ref-1)
2. B V Bowden (ed), *Faster Than Thought,* Pitman, London 1953. [↑](#endnote-ref-2)
3. It was on the first floor of Building 1 West. [↑](#endnote-ref-3)
4. *The birth of The Baby,* Briefing Note 1 researched by Ian Cottam for the 50th anniversary of the Manchester Mark I computer, University of Manchester Department of Computer Science [↑](#endnote-ref-4)
5. http://www.alanturing.net/turing\_archive/pages/Reference%20Articles/BriefHistofComp.html#MUC *accessed 16 December 2015*. [↑](#endnote-ref-5)
6. M V Wilkes, in the IEE Pinkerton Lecture, December 2000. [↑](#endnote-ref-6)
7. G E Moore, *Cramming More Components onto Integrated Circuits*, Electronics, pp. 114–117, April 19, 1965. [↑](#endnote-ref-7)
8. http://web.eecs.umich.edu/~bchandra/courses/papers/Naure\_Algol60.pdf [↑](#endnote-ref-8)
9. *On the Translation of Languages from Left to Right.* Information and Control, 8, 607-639 (1965). [↑](#endnote-ref-9)
10. Frederick Brooks Jr, *The Mythical Man Month*, Addison Wesley 1975 (Anniversary Edition 1995) ISBN 0201835959. [↑](#endnote-ref-10)
11. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/ [↑](#endnote-ref-11)
12. http://www.cs.utexas.edu/users/EWD/ [↑](#endnote-ref-12)
13. http://minnie.tuhs.org/cgi-bin/utree.pl [↑](#endnote-ref-13)
14. http://trace.tennessee.edu/utk\_harlan/ [↑](#endnote-ref-14)
15. Martyn Ould, *Strategies for Software Engineering,* Wiley 1990, ISBN 0471926280 [↑](#endnote-ref-15)
16. http://www.sei.cmu.edu/reports/87tr011.pdf [↑](#endnote-ref-16)
17. https://www.projectsmart.co.uk/white-papers/chaos-report.pdf or https://net.educause.edu/ir/library/pdf/NCP08083B.pdf [↑](#endnote-ref-17)
18. : http://www.infoq.com/news/2014/01/IDC-software-developers

    Gresham College

    Barnard’s Inn Hall

    Holborn

    London

    EC1N 2HH

    www.gresham.ac.uk [↑](#endnote-ref-18)