# The Programmer's Odyssey

Garth Eaglesfield

# The Programmer's Odyssey

## A Journey Through The Digital Age

**Garth Eaglesfield**

Front cover photograph by Ian Eaglesfield

Publication SP-001; Edition 2.00;

Further information:

**http://www.theprogrammersodyssey.**com is the book's official web site and contains distribution and author information and the author's blog.

The author curates a set of articles called 'The History of Computing and Programming' at **http://spanitis.eu/links-articles**.

***Dedicated to my wife Jools and to Alan Cohen, Keith Thomas, Steve Krolak and the whole cast of characters I encountered during the odyssey.***

*With special thanks to Helen Worthington, Dennis Platt and David Petts for sharing anecdotes from the early days.*

# Table of Contents

# *Preface*

It's easy to forget that the digital age, and so the existence of computer programmers, still only spans a single working life, and one of those lives is mine. In my career as a commercial computer programmer I've experienced most of the changes that have occurred in the programming world in all their frequent craziness. This is my attempt to preserve that odyssey for the historical record. But first the caveats.

This is a totally personal account and so it:

- Describes how events happened to me and my thoughts on them, others may differ.

- It usually describes events in the order I experienced them but I was late to the party sometimes, early at others, and sometimes never got there at all. To preserve a logical flow in the book I've been flexible about when individual topics get introduced.

- In the second decade of the 21st century, after a long period of neglect, computer programming has become a topic of much discussion once again. It has been rebranded as 'coding' but I retain the original designation as that's how we used to refer to it.

- Although mostly working in the UK I spent many years in the USA and in the process picked up all kinds of terminology and spellings. As a result this book is written in a version of English that may differ from any conventionally recognised dialect of the language.

- For myself and many others, commercial computer programming has been a rewarding career since at least the late 1960s and during that time programming has seen massive changes in the way programmers work. The evolution of many different programming models and methodologies, the involvement of many extraordinary characters, endless 'religious wars' about the correct way to program and which standards to use. It would be a great shame if a new

generation of programmers were unaware of the fascinating history of their profession, or hobby, and so I came to write this book.

- Hopefully this book is written in a way that it will also be of interest to what we might call the “Bill Bryson” school of general reader. To support this kind of reader I have included sections that are headed “The Basics Of ……” which can be skipped over by readers if they so wish without losing the overall sense of the narrative.

Four of the overall themes in the book can be identified by searching for #UI (User Interface), #AI (Artificial Intelligence), #SOA (Service Oriented Architecture) and #PZ (Program componenti-Zation) and the Table of Contents is detailed enough to guide dipping in/dipping out.

## *Launching A Marshal Mcluhan Probe*

Readers should note that this book only scratches the surface of the history of programming. It is what the late great Marshal Mcluhan (of whom more later) called a ‘probe’, by which he meant a toolkit to use in investigating a subject. To support this probe some in-line links are included but huge amounts of excellent information ***are easily available through online resources such as Wikipedia and the search engines***.

# *Prologue*

## In Years BC (Before Computers)

Sometimes it seems like computers and the digital revolution sprang out of nowhere but of course they didn't. They were the result of at least two and a half millennia of evolutionary development which can be traced all the way back to the ancient Greek philosopher Pythagoras. When I graduated from university in 1968 I had a degree in philosophy but I somehow managed to be awarded it without having absorbed much ancient Greek philosophy or the history and philosophy of science. There were just too many well documented distractions in the sixties. It was only later that I came to realise how critical they were in creating the context for digital computers and with them the new career of commercial computer programmer.

So we need to briefly cover some history in order to set the scene for my account though only the necessary facts will be covered. We'll only be scratching the surface so don't panic but if you prefer to skip this Prologue that's OK. By the way I use BC and AD for dates and not BCE and CE, there is no particular reason, it's just habit.

## 550 – 500 BC

During this period Pythagoras the ancient Greek philosopher was active in a Greek colony in southern Italy. No written record of his thinking survives from the time but he is known to have influenced Plato and Aristotle and through them all of Western thinking. He is rumored to have been the first person to ever call himself a philosopher.

## 350 BC

By 350 BC philosophy was a flourishing pursuit in Athens and by now the results were being written down and thus Aristotle was able to record what he knew of Pythagoras and his followers the Pythagoreans.

*"The so-called Pythagoreans, who were the first to take up mathematics, not only advanced this subject, but saturated with it, they fancied that the principles of mathematics were the principles of all things."*

—Aristotle, *Metaphysics 1–5*

This was quite a claim to make two and a half thousand years ago and Aristotle seems to be less than convinced. To say that the principles of mathematics are the principles of all things, everything, reality as we know it, is an ambitious claim. However, what does it mean? Does it mean 'reality' is somehow dependent on mathematics, that it can be 'explained' using mathematics, that it can be represented digitally? Whatever it means it is an idea that in all its ambiguity has been highly influential. But Aristotle was working on another new and competitive branch of human knowledge, another set of principles, logic, with which he hoped to explain everything. From early on therefore at least two methods were proposed for investigating and explaining reality – mathematics and logic - but in 350BC both methods were severely limited by the lack of the right tools.

But one of the repeating themes in this book is the importance of brilliant thinkers, both male and female, who had ideas that were far ahead of their time but could not be properly explored because the necessary intellectual and technical tools did not exist. It was Pythagoras who set in motion the chain of events that would one day lead to celebrity scientists appearing on TV documentaries and 'explaining' physical phenomenon like black holes with a series of mathematical formulae which are incomprehensible to the layman. Whereas it was Aristotle who set in motion the chain of events that would one day lead to celebrity scientists appearing in TV documentaries and 'explaining' why artificially intelligent machines are inevitable, or impossible, by talking mysteriously about 'algorithms'. They can do this because we now have rich mathematical and logical languages to work with as well as advanced technical tools to extend our limited human abilities. We have a bafflingly complicated mathematical language of positive and negative numbers, complex numbers, irrational numbers, differential calculus, multidimensional vector spaces, infinite numbers and so on. We also have many complex logical constructs and operators from mathematical logic, boolean logic

and possibly even quantum logic, with which to construct computer algorithms and we have built advanced digital computers capable of carrying out trillions of operations per second on which to run them.

What tools did the Pythagoreans have at their disposal to attempt their mission? The ancient Greeks used all 27 letters of their alphabet to represent numbers, compared to the 10 numeric digits we use today, and, for example, to represent the number 9,999 required 37 symbols. Yes, thirty-seven. Extremely cumbersome for carrying out even simple mathematical calculations let alone for developing a formula to express something like the laws of gravity. The system of Roman numerals is probably better known to most people but imagine doing the relatively simple long multiplication of, say, 244 by 478, but using the Roman numerals CCXXXXIV and CCCCLXXVIII. It's not for the faint hearted and to do something similar in Greek numerals would have required a heroic effort worthy of Odysseus himself.

The Aristotelians were similarly limited and only had a logical notation that was pretty crude compared to what was to come later. Aristotle's major creation was the logical syllogism and an example frequently given is:

1. All men are mortal.

2. Socrates is a man.

3. Therefore Socrates is mortal.

This is useful but not really very useful as it arguably only tells us things we already know and is more of an explanation of the meaning of words than involving factual knowledge. It does not really support the advancing of knowledge through logical inference. As a result it is not too surprising that no major practical advances were made during this period of history.

But the Pythagorean project of explaining everything using mathematics continued to haunt humanity and in a broad sense is the driving force of what we now call science. It seems that nowadays no major scientific theory exists that does not include mathematical formulae at the heart of it. In fact the lack of such

formulae in subjects like economics, sociology, and even Darwin's theory of evolution, have led many to doubt that they can be called scientific. But that's a different and highly contentious issue which has been widely explored elsewhere. Meanwhile the effort to explain everything mathematically, not to mention more commercial pressures like the need to generate profits, led to the continuing development of the language of mathematics and to tools that extended human mathematical processing abilities.

## 600 AD

Around this time a huge revolutionary advance was made when our now familiar decimal digits, 0 to 9, that are often rather misleadingly referred to as Arabic numerals, were invented by Hindu mathematicians. It's probably because they first became known in Europe through their inclusion in Arabic scientific treatises that they often get mistakenly referred to as Arabic numerals.

## The Medieval Period

The appearance and use in Europe of the Hindu-Arabic numerals seems to have occurred over a very extended period but the Italian mathematician Leonardo Fibonacci, born in about 1175, is seen as the person who first introduced them to European scholars. But given that this was over 250 years before the invention of the printing press, knowledge of them spread slowly. In fact it was centuries later before they were widely used outside academia for normal business transactions.

The ten digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 are so familiar to us nowadays it is hard to appreciate how long it took to invent them and how revolutionary they were. Why 10 digits? Probably because we have ten fingers but it could have been any number. These ten digits are so powerful because they mean different things ***depending on where they occur in a number***.

So the number 9,999 which took 37 symbols with each one chosen from an alphabet of 27 different letters in the ancient Greek system, in Hindu-Arabic numerals it only requires the use of 4 symbols chosen from a set of 10 digits, with each one being the same digit '9'. This is because, from left to right, 9 represents 9 x 1000 ($10^3$) = 9000, then 9 x 100 ($10^2$) = 900, then 9 x 10 ($10^1$) = 90

and finally 9 x 1 ($10^0$) = 9. Because we are so used to this system it seems to be intuitively obvious that the value of a digit depends on its position in a number but this was a totally revolutionary mathematical advance.

*(To quickly review the definition of the power of a number for those of you who were not paying attention during math (UK=maths) classes – for any number, say XYZ, then* $XYZ^0 = 1$ *whatever the value of XYZ;* $XYZ^1 = XYZ$; $XYZ^2 = XYZ \times XYZ$; $XYZ^3 = XYZ \times XYZ \times XYZ$; *and so on)*

## 1609 - 1905 AD

The use of the Hindu-Arabic symbols created a far more powerful language of mathematics which in the centuries ahead allowed major progress to be made so that crucial areas of our reality, the 'all things' targeted by Pythagoras, could be described by mathematical formulae. Amongst the many mathematical descriptions that were discovered were the laws of planetary motion by Johannes Kepler in 1609 and 1619, the laws of gravity by Sir Isaac Newton in 1687 and the structure of matter itself in perhaps the most famous mathematical formula of all time, $e = mc^2$, by Albert Einstein in 1905. In many cases the scientists themselves had to create new extensions to mathematics in order to do their work, for instance, around the time of Newton several people 'invented' differential calculus independently.

Logical notation only advanced significantly beyond Aristotle's work towards the end of this period and the description of its evolution and convergence with mathematics and computers cannot be bettered than in the following summary taken from Wikipedia.

*"The syllogistic logic developed by Aristotle predominated in the West until the mid-19th century, when interest in the foundations of mathematics stimulated the development of symbolic logic (now called mathematical logic). In 1854, George Boole published 'An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities', introducing symbolic logic and the principles of what is now known as Boolean logic. In 1879, Gottlob Frege published 'Begriffsschrift', which inaugurated modern logic with the invention of quantifier notation.*

*From 1910 to 1913, Alfred North Whitehead and Bertrand Russell published 'Principia Mathematica' on the foundations of mathematics, attempting to derive mathematical truths from axioms and inference rules in symbolic logic*

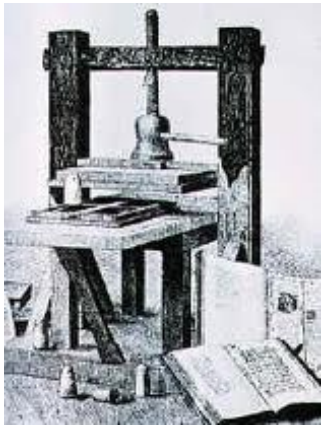
*The development of logic since Frege, Russell, and Wittgenstein had a profound influence on the practice of philosophy and the perceived nature of philosophical problems.* ***Logic****, especially sentential logic, is implemented in computer logic circuits and* ***is fundamental to computer science****."*

By the early 20th century then, powerful mathematical and logical notations had been developed but they were limited by the speed at which they could be used by the human brain. However at the same time as mathematical and logical notations were being developed so were machines which would one day make use of such notations, programmable machines. The notion of a programmable machine is of a general purpose machine which in its raw state cannot produce anything because in order to do so it needs a set of instructions, a program, to tell it what to do. But by using different programs the same programmable machine can produce widely differing outputs. For instance, in today's world the same 3D printer could produce anything from a hair comb to a handgun. That is the essence and immense power of programmability.

The concept of programmability evolved over the centuries involving different technologies and many different areas of interest and the pioneers of digital computing made their breakthroughs, as Sir Isaac Newton famously said of his own work, because they stood on the shoulders of giants.

It's not necessary to go too deeply into the pre-history of programmable machines which laid the foundations for the early digital computers of the 1940s and 1950s but a few examples will help put things into context. The printing press invented by Johannes Gutenberg in around 1440 was a revolutionary programmable machine. Its program being the contents of a book set in moveable typeface to produce multiple copies. Because of its programmability, i.e. the ability to 're-program' the printing press by moving the typeface to form different words, the same printing press

can be used to output an essentially infinite number of different books.



Gutenberg's Printing Press

It is also possibly the first example of mass production since literally millions of copies of the same book could be produced, a gigantic step forward from laboriously copying manuscripts by hand. The effect this device had on the spread of human knowledge is incalculable and was excellently explored by Marshal Mcluhan in 1962 in his book 'The Gutenberg Galaxy'.

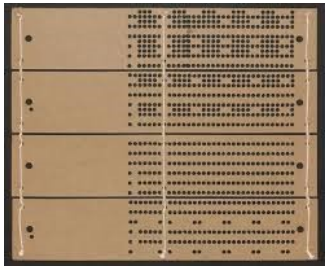
A Set Of Moveable Type

Another highly significant programmable machine was the Jacquard weaving loom first demonstrated in 1801 by Joseph Marie Jacquard, though it is said it was based on earlier work done by others. It used a forerunner of the punched card technology that would later be used by the first computers. Each row in the linked deck of punched cards controlled the multiple different shuttles

containing the different coloured threads used in the weaving process. As the punched card rows were passed in front of the shuttles each hole in the card allowed the corresponding shuttle to insert a stitch in the cloth being woven, whereas the lack of a hole blocked the corresponding shuttle from inserting a stitch. We could say that a hole meant 'weave' whereas a no-hole meant 'don't weave'.

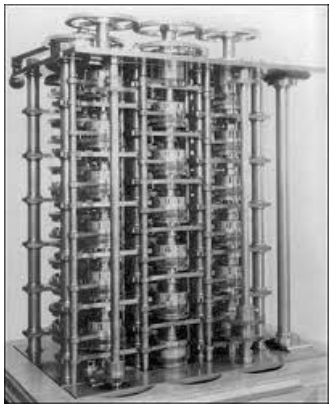
A Jaquard Loom

The program to weave a specific pattern of cloth consisted of a linked deck of punched cards which were used to control the loom and widely different patterns could be produced on the same loom by using different programs to control it. One deck of cards might weave a carpet whereas another might weave a scarf when loaded on to the same loom.



A Punched Card Weaving ‘Program’

The first serious effort to apply programmability to mathematical calculations, rather than to produce physical objects, was made by Charles Babbage in 1822. He designed two mechanical machines, the Difference Engine and the Analytical Engine.



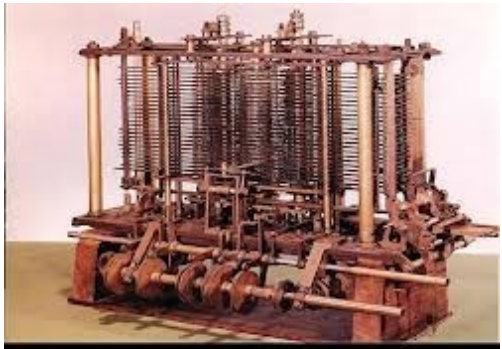
The Difference Engine

The Difference Engine can be thought of as a giant calculator, a number cruncher.

But the Analytical Engine was a giant step forward and contained equivalents of all the elements that would eventually form part of digital computers. ‘RAM’ or 'memory' for storing intermediate results, 'registers' (which Babbage called ‘variables’) in which to execute the mathematical operations and even mechanical logic gates that could implement ‘if (something is true) then (do something)’ operations. In the absence of electricity and minituarised electronic components the Analytical Engine was to be a steam driven mechanical computer. Its size would be immense, even a small version would be at least forty-five feet long, and it would require massive expenditure on materials with engineering work of extremely high precision.

As happens so often Charles Babbage was a man way ahead of his time and though a successful working example of his Difference Engines was built the Analytical Engine required funding and engineering effort on such a grand scale it was never created. But the intriguing possibility exists that if the necessary effort and

funding had been found it could have ushered in a mechanical steam driven digital age in Victorian England.




Although the Analytical Engine was never built its design was widely discussed at the time and it was one of Charles Babbage's associates who was to fully grasp its potential. She was Augusta Ada King, Countess of Lovelace (*née* Byron; 10 December 1815 – 27 November 1852) who is usually referred to as simply Ada Lovelace. She was the only legitimate child of the notorious Lord Byron, poet, Greece independence fighter and literary inspiration to Edgar Allen Poe. It was she who grasped the full potential of the Analytical Engine. A very talented mathematician in her own right she had cooperated with Babbage on the Difference Engine and his design of the Analytical Engine.

Between 1842 and 1843, as requested by Babbage, she translated into English a description of the Analytical Engine written by an Italian military engineer Luigi Manabrea who had attended a lecture given by Babbage. Babbage suggested to her that she add her own ideas to the translation and these, commonly referred to as ‘the Notes’, were eventually twice as long as Manabrea’s description. She realized there needed to be a way of ‘writing a program’ for the Analytical Engine to work and she went up to Mansfield in northern England to visit a textile mill and see a Jacquard loom in action using punched cards. Her Notes, published in 1843, include the first computer program ever written. It is an

algorithm to be encoded on punched cards and used for creating a series of Bernoulli numbers using the Analytical Engine.

Ada Lovelace, first computer programmer

She also realized that since musical notes can be expressed numerically it was feasible that music might be written algorithmically. Ada Lovelace was someone else who was born at least a hundred years ahead of her time and it took that long for others to appreciate her work.

**The Dawn Of The Digital Age**

By the 1930's all the pieces were in place for the creation of basic digital computers. Sophisticated mathematical and logic notations had been developed and some experience with programmable mechanical devices had been gained. Advances in the fields of electricity and electronics now made it possible to envisage creating electrically powered programmable devices of a reasonable size by the standards of the time. At the same time pressures ranging from the complexity of international commerce to World War II code breaking requirements highlighted the need to process large volumes of data quickly. But human beings were

simply unable to push enough data through their increasingly complex mathematical and logical constructs fast enough.

As throughout history we looked for tools to extend our limited human capabilities. Some simple mechanical tools already existed, such as slide rules, log tables, truth tables, the abacus, the mechanical adding machine, but the effect of these tools was limited. What was needed was a revolutionary technology advance that would extend human capabilities exponentially, something comparable to the introduction of the Hindu-Arabic numerals.

It first arrived in embryonic form during the Second World War, with the development of a basic programmable electronic computing machine, the digital computer, which sparked a revolution and ushered in a new age - the digital age. The digital computer brought new life to the dreams of Pythagoras and Aristotle by massively extending our capabilities for manipulating numerical data and applying logic to it systematically. It has enhanced, extended and may yet simulate human experience. In the process new notations, new languages, have been invented, to create a new kind of object, the computer program, the algorithm, or just plain old software, and with it a new career, that of the commercial computer programmer.

# The Programmer's Odyssey

I was born in 1946 so that's when my odyssey began, which fits quite well with the beginning of the digital age.

To begin with I will describe how the idea of a computing machine that could be programmed to do real things emerged into popular culture and rather more slowly into my own consciousness in the 1950s and 1960s and eventually provided a career path for me. But it was one I didn't fully realise existed until 1971 when I grasped that computer programming was a real career choice for me and it was 1972 before I became a trainee programmer and my odyssey began in earnest.

In later chapters I'll describe how during the 1970s, 1980s and 1990s developments in computing were happening all around me, often at breakneck speed. Let me repeat a couple of my disclaimers from the Preface:

- This account describes how events happened to me and what I thought about them at the time, others may hold different views.

- This account describes events in more or less the sequence in which they happened to me but I'm sure I was late to the scene sometimes, early at others, and sometimes never got there at all.

An eternal truth in computing technology is that controversy and disagreement dog each step of the way. Computing seems to encourage the search for the holy grail, be it iOS versus Android, object oriented versus procedural programming, or the use of the 'GO TO' statement in COBOL. There is also no shortage of gurus claiming to be the original discoverer of the specific holy grail being discussed. As a result the practice of computing has constantly been riven by religious wars every bit as intense as those of the Middle Ages, though happily with fewer burnings at the stake. It has only been as time passed that the winners in these religious wars emerged. This account only gives my own view of these conflicts and to make better sense of them some basic technical concepts are introduced from time to time.

These occur in sections headed ***'The Basics Of …….'*** but they can be skipped without losing the gist of the overall narrative, however it may reduce your understanding of the odyssey if you do.

Here's the first example.

## *The Basics Of Binary And Hexadecimal*

One of the foundations on which the digital age is built is the existence of electronic components that exist in one of precisely two states, on or off. Initially these were thermionic valves but they were quickly replaced by transistors. To see how this simple on/off property has proved to be so powerful we need to dig a little deeper into number systems.

It was noted earlier that the decimal number system we use, which has ten digits, probably arose because we have ten digits on our own two hands. In the decimal system the value of a digit depends on its position within a number and from right to left the digit positions correspond to units ($10^0$), tens ($10^1$), hundreds ($10^2$), thousands ($10^3$) and so on. To repeat the earlier example, in the number 9,999 then, from left to right, 9 means 9 x 1000 ($10^3$), then 9 x 100 ($10^2$), then 9 x 10 ($10^1$) and finally 9 x 1 ($10^0$).

But there is nothing special about basing a number system on ten, a number system can use any base value. In a base 5 number system that uses only 5 digits (0, 1, 2, 3, 4) we count upwards this way, 0, 1, 2, 3, 4, 10, 11, 12, 13, 14, 20, and so on. In this number system the digit positions from right to left correspond to units ($5^0$), fives ($5^1$), twenty fives ($5^2$), one hundred and twenty fives ($5^3$) and so on. Thus in the base 5 number system the symbol ‘10’ means the physical number five, (1 x $5^1$ plus 0 x $5^0$) and this seems totally strange to us but that is only because we're so used to the decimal system. The physical number referred to by the symbol '10' is completely dependent on the base value of the number system.

Suppose that we create a base 2 number system using only 2 digits (0 and 1). This is the binary system, and it looks very clumsy indeed. The binary system only uses 2 digits, 0 and 1, and the columns from right to left correspond to units ($2^0$), twos ($2^1$), fours ($2^2$), eights ($2^3$) and so on. We count 0, 1, 10, 11, 100, 101, 110, 111,

1000 and in this system ‘10’ means physical number two (1 x $2^1$ plus 0 x $2^0$). Large binary numbers involve very long strings of 0s and 1s and the binary system is distinctly user unfriendly for human beings, but for computers it's perfect. If we have an electronic component that can be either on, or off, like a valve in the early days of computing, or a transistor since the 1960s, then we can say 'on' means '1' and 'off' means '0'. We have now implemented a binary 'bit' and if we string 8 bits together we have an 8 bit 'byte', which can hold binary numbers up to 11111111, which is up to the decimal number 255. You may be familiar with Internet addresses of the form 121.243.65.222, these are the decimal representation of the original 4 byte (32 bit) internet addresses with each byte value separated with a '.'.

Nowadays the use of binary numbers is well hidden from most computer programmers but in the early days of computing it was frequently necessary for programmers to actually refer to them. Writing out long strings of 0s and 1s was far too clumsy a method and referring to them by their decimal equivalent was very non-intuitive. The solution that was adopted was to use yet another number system, the hexadecimal system which is base 16. As there are only 10 conventional digit symbols an additional 6 digit symbols are needed and the letters of the alphabet are used. Counting upwards in hexadecimal goes 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, and so on, in this number system ‘10’ means physical number sixteen. By using hexadecimal a long binary number can be broken down into groups of 4 bits and each one can be represented by a single hexadecimal digit. This is, or becomes, far more intuitive than using decimal equivalents.

Consider the example:

The letter ‘N’ (in ASCII notation, see Appendix A) is represented by the binary number 01000001, which is the decimal number 78. If we break the 8 bits into two groups of 4 bits we get 0100 1110 which we can translate to hexadecimal 4E. As you can see it is far easier to translate from hexadecimal to binary than it is from decimal to binary.

# 1940s - Code Breaking Machines

## *When The War Was Over*

By 1946 when I was born in the UK the Second World war was over and the troops were starting to head for home like Odysseus after the Trojan war. This means I am part of the leading edge of the postwar baby boom and so my memories of the 1940s and 1950s are fairly patchy. My earliest clear memory is of the day that war time milk rationing finally came to an end in the UK and my brother and I both drank pints of milk in celebration. The post-war era was completely different from today and also from the pre-war era, in fact one era was ending and another was about to begin.

My next really intense memory is of the day in 1957 in Banks record shop in York on the banks of the river Ouse that I first heard “That'll Be The Day” by Buddy Holly and the Crickets. Like so many of my generation I was knocked out by the driving sound of the first true rock and roll band in the world. The rock and roll era had started in earnest. Even now listening to Buddy Holly playing that fiery opening riff on his Fender Stratocaster sends shivers down my spine. Looking back I see it was a wake-up call to me and my generation that the world was changing and that my life and my career would not be lived out in the same world as that of my parents and my three older brothers. I would be living and working in a new and different world, a new age, the digital age, and the soundtrack to the digital age would be rock and roll.

In between these two early memories I was busy absorbing all the UK postwar mythology about the jolly good British chaps in their Spitfires who saved us from the Nazis. It would only be many years later I would discover that there were many other nationalities involved too, such as Poles, Canadians, Americans, and Antipodeans. It would be even more years before I discovered that during this period the great mathematical genius and patriot who, it has been said, was second only to the legendary heroes like Winston Churchill in saving us from the Nazis was being hounded to his death because he was gay. That was of course Alan Turing who had worked at the Bletchley Park decoding centre during the

war and it was him too we should note who rediscovered the largely forgotten work of Ada Lovelace.

At the same time three other heroes from Bletchley Park, the engineering genius Tommy Flowers, the brilliant breaker of the Lorenz code Bill Tutte, and the network analyst Graham Watchman, were being air-brushed out of the history books. Flowers created the Colossus machine, the first true digital computer which was built with thermionic valves. The key insight was that a thermionic valve's on and off states could represent 1 and 0 respectively and by linking them together in a sequence they could represent a binary number. The ten existing Colossus machines were destroyed at the end of the war for 'security' reasons by, who else, the politicians who were suffering the opening stages of Cold War paranoia. While Graham Watchman needed to go to the USA to continue his pioneering work on information networks as there was no interest in the UK and Bill Tutte emigrated to Canada to pick up his career in mathematics.

The code breaking work done at Bletchley Park was at least as important as all the military actions in winning the war yet it was only many many years later that this was admitted. At that time technology wasn't sexy and phrases like 'cyber warfare' didn't even exist. By hounding Turing to his death because he was gay, physically destroying Flowers' Colossus computers, and forcing Watchman and Tutte into exile in order to continue their careers, their breakthrough ideas and technologies and so the entire computer industry, were left to be developed in the USA by such luminaries as John Von Neumann. The rest, as they say, is history.

## *The Basics Of Turing's Ideas*

In 1936, aged 24, Alan Turing wrote: "We may now construct a machine to do the work of this computer." But in 1936 a computer was a person, usually a woman, who knew how to compute mathematical calculations. Some simple devices existed to help her — calculating machines, cash registers — but only to do specific things.

What Turing understood was that a sufficiently well designed machine could do almost anything, because both its input data and the instructions defining the operations to carry out on the data

could be fed into it in the form of numbers. His paper, called “On computable numbers with an application to the Entscheidungs problem”, was addressed to an esoteric question in mathematics but it was in the method he used to tackle the question that made Turing’s approach stand out. It seems obvious now that you can feed both data and instructions (software) into computer hardware and change its “state of mind” i.e. how it is to process the data. But this was quite revolutionary in 1936.

## *The Turing Test*

(#AI) Turing even foresaw that there might come a time when machine intelligence became a possibility and in a 1950 research paper, *Computing Machinery and Intelligence,* he defined what has come to be known as the Turing Test. Its purpose is to measure a computer’s ability to behave as intelligently as an actual human being. Essentially it is language based and says that when you cannot tell whether you are conversing with a machine or a human being, and in Turing’s day the conversation would be through a teletype, then machine intelligence has arrived. Machine intelligence has come to be known as Artificial Intelligence, or AI, and the Turing Test regarded as a way of testing for AI.

However this is by no means as straightforward as it is sometimes claimed. First the very definition of ‘intelligence’ has puzzled and divided both philosophers and scientists for many years. Even if a definition can be agreed then how does it differ from and/or does it involve other key components of human experience in particular consciousness, emotions, feelings and our physical living presence in the world?

As we will see none of this has stopped the term 'AI' being widely used since then as if we were all in agreement on what it means, even though we’re not. But more on this later.

## *Meanwhile In The USA*

Meanwhile in the USA in 1947 a dramatic hardware breakthrough occurred. Up until then early computers, like Tommy Flower’s Colossus, were based on thermionic valves, often referred to as 'tubes', to represent binary zeroes and ones by being either off or on. Unfortunately they suffered numerous reliability problems,

they got hot, they were relatively bulky and they failed altogether fairly frequently.

Thermionic Valves

In 1943 Thomas Watson president of IBM had famously said that “I think there is a world market for maybe five computers” but of course when he said "computer" he meant an "unreliable vacuum-tube-powered adding machine that's as big as a house." It's fair to say that few people ever wanted one of those, regardless of the size of their desk.

In 1947 a group led by William Shockley at Bell Telephone Labs in New Jersey, now referred to as Bell Labs, made a massive step forward from thermionic valve technology and created the first working transistor. A smaller, far more reliable, robust and cooler solid-state device that had the same crucial property as a thermionic valve, the ability to be on or off, 1 or 0.

Some early transistors

The creation of these small reliable components led to major advances in many consumer electronic devices not only computers. By far the most famous at the time were the first small and portable

radio sets to become available, transistor radios, and at the same time car radios became far more robust and widely available.


Transistor based car radios had NO TUBES!

These technology advances soon began to have significant cultural effects, especially in teenage culture. Before long young people could listen to the new and much criticized rock and roll music on their transistor radios far away from their disapproving parents.

A classic transistor radio

To get a feel for the place of the transistor radio in teenage culture at the time listen to the lyrics of ‘Brown Eyed Girl’ by Van Morrison. Major technological and cultural revolutions were now underway and they were already closely interconnected. These two revolutions would change the world we live in. The technological foundations of ‘The Sixties’ cultural revolution were being laid.

Teenagers rocking and rolling to a transistor radio

Another revolutionary advance occurred in 1948 when Claude E. Shannon, also at Bell Labs, published a groundbreaking paper called “A Mathematical Theory of Communication”. As one of the implications of his paper was the primary importance of the ‘bit’ of information which could have two values, 0 or 1, it dovetailed perfectly with the invention of the transistor.

In 1956, buoyed by the invention of the transistor, William Shockley left Bell Labs to form his own company, Shockley Semiconductor, which he located in California in the area that would come to be called Silicon Valley when it filled up with other electronics companies. At least 65 companies are said to have been founded by former Shockley Semiconductor employees. In fact by 1957 his notoriously overbearing management style had already alienated 8 of his research team as well as his fundamental disagreement with them over the correct way forward in transistor research. So they, calling themselves the “traitorous eight”, left to form another company that was eventually acquired by Fairchild Semiconductor where they could build transistors the way they wanted to. These eight men included Gordon Moore and Robert Noyce and in 1959 Robert Noyce invented the integrated circuit (IC) whereby a number of transistors could all be placed on a single piece of semi-conductor material such as silicon. Such pieces of silicon packed with electronic components eventually came to be known as chips. This huge leap forward in the minituarisation and efficient packaging of the basic on/off components was another giant advance. As for Gordon Moore we will hear of him again shortly.

Also in the late 1950s, Allen Newell and Herbert Simon at RAND Corporation in the USA demonstrated how computers could do much more than calculate and subsequently went on to claim that computers could be used to simulate important aspects of human intelligence. The field of AI (Artificial Intelligence) was born and became a recognized academic research discipline in the US in particular, in which hopes of creating a machine that could pass the Turing Test were born.

## *1959 The Day The Music Died*

On February 3rd, 1959 as the fifties were drawing to a close Buddy Holly was killed in a plane crash in Iowa and the first era of rock and roll was over. Years of vapid, weepy and soulless music were to follow, an era which The Animals would lambast in “The Story Of Bo Diddley” on their debut album in 1964. Later in 1971 the tragic event of Holly's death would be immortalised as 'The Day The Music Died' in Don McLean's 1971 single “American Pie”. But this set back to rock and roll did not stop the technological revolution, it carried right on.

In that same year I was 13 and I doubt if I had ever even heard the word 'computer' and certainly if I had it failed to register with me. If I had any awareness of futuristic technological ideas at all they would probably have involved space travel both real and as imagined in comic books. The Soviet Union had launched Sputnik on October 4th 1957 which had ratcheted up Cold War paranoia and inaugurated the space era and the space race. This, added to the fact that the US and the Soviet Union both possessed fleets of bombers carrying nuclear bombs as well as arsenals of nuclear tipped ICBMs (Inter Continental Ballistic Missiles), meant we were all now living nervously in the era of MAD (Mutually Assured Destruction). The MAD philosophy was “if you fire yours we'll fire ours too and then we'll all die”. Mad indeed.

In the comics I was reading at the time I did occasionally encounter another example of advanced technology in the form of robots, but there was never any risk of them being mistaken for human beings as they usually resembled the Tin Man from The Wizard of Oz. But unbeknown to me Isaac Azimov the science fiction writer was already worried about what robots might do to human beings

and in 1942 he had laid down some rules for robots to obey. The Three Laws of Robotics (also known as Asimov's Laws) are a set of rules he introduced in his 1942 short story "Runaround", they are:

1. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
2. A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Laws.

As is so frequently the case an artist was way ahead of scientists and the general public in foreseeing potential problems with technology and thinking about what could be done.

Meanwhile back at school in the real world I was now obliged, as you were in those days at many UK schools, to enter either the 'science track' or the 'arts track', sadly there was no 'Renaissance man track'. Being terminally hopeless at foreign languages and quite good at maths, physics and chemistry it was a no-brainer, I joined the science track. So as the 1950s turned into the 1960s it was time for me to start preparing for the science ‘O’ level exams I would be taking in 1962 when I was 16 and then my ‘A’ levels in 1964. So at Hull Grammar School, in Kingston-upon-Hull on the Humber estuary, I tried to become a conscientious student.

# 1960s - Super Number Crunchers

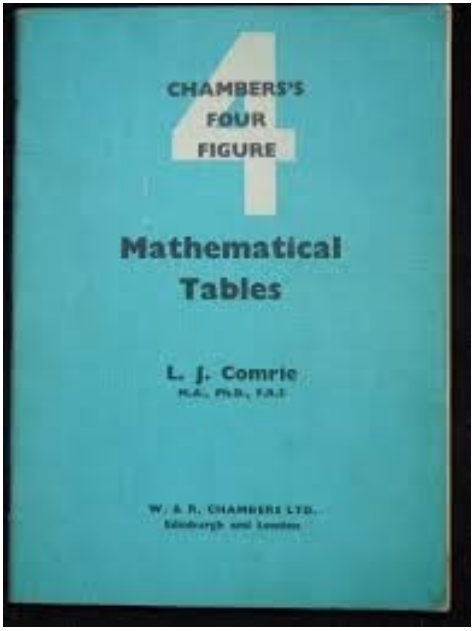
## *Trying To Remember The 60s*

They say that if you can remember the 1960s then you didn't live them to the full. I think I did live them to the full and it has to be admitted that it effected my memory and therefore this account. It has been endlessly documented that the 1960s were an incredible period to live through socially and culturally, and for me it was also the time when I was passing from childhood to adulthood. As a result the youthful obsessions of the age (sex, drugs and rock'n'roll) occupied much of my time, not to mention the fact that I was frequently supposed to be studying rather than enjoying myself, first at school and then at university. All of this activity completely distracted me from taking any notice of developments that were taking place in technology.

It's rather that I was distracted by other things than that I've forgotten them. But nevertheless crucial technical developments were happening frequently during this period and they will be included in their correct place in the narrative as they are essential to the story. It was during this period that computers started to make their way into the realms of popular culture and into the commercial business world. Perhaps technology developments and cultural developments have always effected each other but they certainly did so in the 1960s and have continued to do so ever since.

## *The Basics Of Maths In The Pre-Calculator Era*

As an indication of just how far away I was from concepts like computers in the early 1960s consider how we teenage mathematicians did our calculations. When I studied for 'O' and then 'A' level maths at Hull Grammar School the world was still in the pre-calculator era. If we needed to do any significant calculations we used a traditional paper based tool, log tables, short for logarithmic tables. We all had our own book of log tables which were generally very battered, dog-eared, ink-stained, and usually covered in schoolboy graffiti. They were about the size of a theatre program and until you knew how to use them they appeared to be filled with tables of completely meaningless numbers.




A book of log tables

The way they worked relied on our old friends the powers of numbers and the fact that by adding the powers of two numbers together you were in fact multiplying them.

Log table pages

Consider the following simple example:

- $10^2 = 10 \times 10 = 100$
- $10^3 = 10 \times 10 \times 10 = 1{,}000$
- $10^5 = 10 \times 10 \times 10 \times 10 \times 10 = 10^2 \times 10^3 = 100{,}000$

From this you can see that $10^2 \times 10^3 = 10^5$ or $10^{(2+3)}$ which is to say that you can multiply two numbers together by adding their powers of 10.

Because any number can be expressed as a power of 10 the log tables contained the power of 10 that corresponded to each number, the log values. By looking up the log values for two numbers and adding them together you got the log value of the result.

| No. | Log. | No. | Log. | No. | Log. | No. | Log. | No. | Log. |
|---|---|---|---|---|---|---|---|---|---|
| 0 | $-\infty$ | 20 | 1·30103 | 40 | 1·60206 | 60 | 1·77815 | 80 | 1·90309 |
| 1 | 0·00000 | 21 | 1·32222 | 41 | 1·61278 | 61 | 1·78533 | 81 | 1·90849 |
| 2 | 0·30103 | 22 | 1·34242 | 42 | 1·62325 | 62 | 1·79239 | 82 | 1·91381 |
| 3 | 0·47712 | 23 | 1·36173 | 43 | 1·63347 | 63 | 1·79934 | 83 | 1·91908 |
| 4 | 0·60206 | 24 | 1·38021 | 44 | 1·64345 | 64 | 1·80618 | 84 | 1·92428 |
| 5 | 0·69897 | 25 | 1·39794 | 45 | 1·65321 | 65 | 1·81291 | 85 | 1·92942 |
| 6 | 0·77815 | 26 | 1·41497 | 46 | 1·66276 | 66 | 1·81954 | 86 | 1·93450 |
| 7 | 0·84510 | 27 | 1·43136 | 47 | 1·67210 | 67 | 1·82607 | 87 | 1·93952 |
| 8 | 0·90309 | 28 | 1·44716 | 48 | 1·68124 | 68 | 1·83251 | 88 | 1·94448 |
| 9 | 0·95424 | 29 | 1·46240 | 49 | 1·69020 | 69 | 1·83885 | 89 | 1·94939 |
| 10 | 1·00000 | 30 | 1·47712 | 50 | 1·69897 | 70 | 1·84510 | 90 | 1·95424 |
| 11 | 1·04139 | 31 | 1·49136 | 51 | 1·70757 | 71 | 1·85126 | 91 | 1·95904 |
| 12 | 1·07918 | 32 | 1·50515 | 52 | 1·71600 | 72 | 1·85733 | 92 | 1·96379 |
| 13 | 1·11394 | 33 | 1·51851 | 53 | 1·72428 | 73 | 1·86332 | 93 | 1·96848 |
| 14 | 1·14613 | 34 | 1·53148 | 54 | 1·73239 | 74 | 1·86923 | 94 | 1·97313 |
| 15 | 1·17609 | 35 | 1·54407 | 55 | 1·74036 | 75 | 1·87506 | 95 | 1·97772 |
| 16 | 1·20412 | 36 | 1·55630 | 56 | 1·74819 | 76 | 1·88081 | 96 | 1·98227 |
| 17 | 1·23045 | 37 | 1·56820 | 57 | 1·75587 | 77 | 1·88649 | 97 | 1·98677 |
| 18 | 1·25527 | 38 | 1·57978 | 58 | 1·76343 | 78 | 1·89209 | 98 | 1·99123 |
| 19 | 1·27875 | 39 | 1·59106 | 59 | 1·77085 | 79 | 1·89763 | 99 | 1·99564 |

As a trivial example we can use the log table page above to multiply 3 by 23. The log value of 3 is 0.47712 and the log value of 23 is 1.36173, if we add them together we get 1.83885 and if we look for the number which has a log value of 1.83885 we see it is 69, the correct answer.

Although this is a trivial example the same principle applies to all numbers and so multiplication involving large numbers could be done quickly with log tables using only simple addition. Without them the maths exams we were taking would either have needed to last much longer or have been made much simpler.

Another calculating tool of the time, mostly used by engineers, was the slide rule about which more information can be found on the Internet.

## *The 60s Really Get Going*

In 1962, with my log tables in hand I took my maths O levels. It was the same year that the Beatles, the Rolling Stones, the Yardbirds and many other UK bands rediscovered Buddy Holly's music and his rock band format. They also discovered the electric blues sound of Chicago bluesmen like Muddy Waters, Howling Wolf, Bo Diddley and Jimmy Reid and in blending it all together rock and roll version 2 kicked off. In keeping with the times my personal cultural and fashion icon now changed from being James Dean to become Ringo Starr. 'The Sixties' as they came to be known were getting started in earnest.

## *Cold War Paranoia Feeds Technology Anxieties*

After taking my O levels I entered the sixth form at school to study for A levels in Pure Maths, Applied Maths and Physics and in October 1962 I was sitting nervously in a maths lesson as a crucial moment in human history occurred. The US navy was scheduled to intercept Soviet navy ships as they tried to break the US blockade of Cuba and deliver nuclear missiles to Castro's island. Tension was high as the world waited to see if the MAD scenario was about to be played out. The Cuban Missile Crisis, as it came to be known, raised cold war paranoia levels and worries about the MAD philosophy to almost unbearable levels. Later, in 1968, Geoff Nuttall wrote the seminal book 'Bomb Culture' which explored the effect of these worries on the 1960's counterculture. The Missile Crisis ended peacefully and without conflict, but it made the risk of a 'MAD' nuclear war a very real public anxiety. In its most extreme form the anxiety focused on the possibility of a nuclear war being started and fought by mistake, perhaps by a technology failure.

In the chilling 1962 novel 'Fail Safe' a mistaken US nuclear attack on Russia is caused by an electro-mechanical failure in pre-electronic defence equipment. In the preface to the novel the authors, Eugene Burdock and Harvey Wheeler, explain what the novel is intended to warn the public about.

*“For there is substantial agreement among experts that an accidental war is possible and that its probability increases with the increasing complexity of the man-machine components which make up our defense system.”*

An electrical failure occurs in one of the six ‘Fail Safe Activating Mechanisms’, machines housed in the Presidential Command Net room in the War Room of the White House. The Fail Safe Activating Mechanisms are to be used “only at express presidential order” to instruct US bombers to proceed beyond their fail safe points and attack Russia. The failure is described as follows

*“... in Machine No. 6 a small condenser blew. It was a soundless event. There was a puff of smoke no larger than a walnut that was gone instantly. ….... No instruments on the table indicated a malfunction.”*

As a result of the electrical failure a single flight of bombers proceeds beyond its fail safe point, the position in which they circled awaiting a go/no-go instruction, and carries out a limited nuclear attack on the Soviet Union. In order to placate the Soviet Union the US themselves have to drop a nuclear weapon on New York City to balance the account. Sombre stuff indeed. In 'Fail Safe' the crucial failure is of a simple pre-electronic electro-mechanical component, one of the “man-machine components” which so concerned the book’s authors, and there is no involvement of computers. The result is some tragic, but in the end limited, nuclear detonations but it's only the beginning of concerns about technology failures causing human catastrophes.

Two years later in the speculatively computer-aware 1964 film, Stanley Kubrick’s movie masterpiece 'Dr Strangelove: or How I Stopped Worrying And Learned To Love The Bomb', an unplanned US nuclear attack on the Soviet Union also takes place. This time it is not caused by a technical failure but by a paranoid ultra-nationalistic US Air Force base commander, Colonel ‘Jack’ Ripper.

Colonel Ripper orders his bombers to proceed beyond their fail safe point and attack the Soviet Union. But by now computers, still usually seen as just superfast electronic calculating machines, were beginning to get bit parts in TV shows and movies when the plot demanded the presence of major calculating capabilities. The standard way of indicating this was by showing a large air-conditioned computer room hosting a massive mainframe computer which could be identified as such by the presence of huge numbers of magnetic tape drives with their tapes in constant jerky motion.

In Dr Strangelove although the mistake that starts the nuclear war is not blamed on a computer the unfortunate British RAF officer on Colonel Ripper's base, as part of the 'Officer Exchange Program', Group Captain Lionel Mandrake (Peter Sellers, this movie is also his masterpiece), is seen several times in a computer room on the base with the usual tape drives in action behind him.

Mandrake at the computer console with tape drives in view

He is also sometimes seen removing and inspecting mysterious printouts from a line printer. We never know what they are for but they imply great technical calculating power at work and indicate that Mandrake knows how to interprete the no doubt complicated results.

Mandrake in the computer room at the line printer

But later in the movie a far more deadly role for computers is introduced when the Soviet ambassador claims that extreme elements in the Soviet military have built a Doomsday Machine. Doomsday Machines were much discussed in those days, and it was even said that the RAND corporation had already designed one. It was the ultimate MAD machine that would automatically detonate enough nuclear devices to poison the earth's atmosphere and cause the end of the human race if it detected an incoming nuclear attack. The world really was that insanely suicidal in those days.

The US President Merkin Muffley (also Peter Sellers) is skeptical about the technical feasibility of a Doomsday Machine and thinks it is just Soviet propaganda. He asks his scientific expert Dr Strangelove (also Peter Sellers) if it is technically possible and Dr Strangelove explains to him that it is, that a Doomsday Machine could be set off automatically by using computers as more than just giant calculating machines. This could be done he explains by ***"a giant complex of computers"*** initiating the Doomsday Machine if a certain set of conditions occurred that were ***"programmed into a tape memory bank"***. His proposed system design now seems rather technologically quaint but not at the time. (#AI) It introduced the idea that computers were more than just superfast calculating machines and could potentially make decisions and those decisions might wipe out the human race.

As the film ends Slim Pickens, playing Major T.J. 'King' Kong, has to enter the bomb bay of the final surviving US bomber to manually repair the bomb release mechanism that's been

damaged by a missile strike. He is eventually successful but the bomb releases so quickly he exits with it. As a result he is sitting, cowboy style, astride the only nuclear bomb to successfully get through Soviet air defences as it plummets towards its target. All the other US bombers have been shot down.

Major Kong Rides The Atomic Bomb To Its Target

But after hearing about the Soviet Doomsday Machine the film's viewers know only too well that even though it is the only bomb to have got through it will still result in the end of the entire human race because the Doomsday Machine is controlled by computers. Things had moved on terrifyingly since the 'Fail Safe' scenario as computers came on the scene.

So when I travelled to Liverpool, the land of the Scousers, in 1964 to attend Liverpool University the potentially crucial and dangerous role of computers in human affairs was already being addressed in popular culture, though I think it had not caught my own attention. While I attended university, like most UK students at the time, I only had limited access to television but I did sometimes get to watch the popular "Man From U.N.C.L.E" spoof spy series. It would sometimes feature computers with their obligatory banks of tape drives and though I can't now remember the details of a single show I recently looked up the list of all the shows ever produced and I found this one from 1965.

*"Man from Uncle – "The Ultimate Computer Affair" – 1/Oct/65*

*What is the "ultimate computer"? A device capable of conceiving and planning operations, not merely performing calculations quickly."*

Thus even as early as 1965 popular TV shows were starting to include plots about computers that were **“capable of conceiving and planning operations”** and not just **“performing calculations quickly”**. A scary thought but I'm sure that if at the time I gave it any thought at all I would have dismissed it as a fantasy, assuming that in reality all computers could do was to indeed “perform calculations quickly”. Nevertheless computers were now starting to feature in popular culture in movies, books and TV shows and questions about what they could potentially do, both good and bad, were starting to be asked. As occurs so frequently the questions were being asked by authors, TV producers and film makers, not technologists.

In that same year, 1965, and completely unknown to me, Gordon Moore in California published what has become known as Moore's Law. This says that the number of transistors that can be packed on to a single IC (integrated circuit, or chip) doubles about every eighteen months and yet the price halves. This observation has proven to be remarkably accurate and although Moore only thought it would remain true for 10 years it still seems to be operative. The accuracy of Moore's Law has been a major driver of the exponential growth of digital technology and its transformational effects on our culture.

## *Losing My (Maths) Religion*

It wasn’t only the distractions of the 1960s that meant I was not paying much attention to the developments taking place in computers at the time. I was also losing interest in maths and I would have viewed computers as merely mathematical tools, super number crunchers. After getting my A levels in 1964 I lost interest in maths for two reasons. When we’d studied things like geometry and quadratic equations at school I found the puzzle solving aspects quite invigorating but at university we focused on complex differential equations, infinite dimensional vector spaces and so on which failed to excite me and left me struggling badly.

There was another typical 1960s reason why I lost interest in maths. I felt I wanted to study something more ‘meaningful’, whatever that meant. After arriving to study Pure Maths in 1964 I changed to a Pure Maths & Philosophy joint degree (1965) and then Philosophy with Pure Maths subsidiary (1966) and finally emerged with a Philosophy degree (1968). Ironically, by switching to the more

‘meaningful’ subject of philosophy I became involved in the study of logic which one day I would realise was what computers were all about. But when I graduated in 1968 I doubt I could have conceptualised the notion of becoming a computer programmer beyond thinking it involved defining lots of differential equations for an electronic number cruncher.

In the same year over in Silicon Valley another huge step forward was taken when Intel Corporation was founded in Mountain View, California by Gordon E. Moore (of "Moore's Law" fame, a chemist and physicist), Robert Noyce (physicist and co-inventor of the integrated circuit), and Arthur Rock (investor and venture capitalist). Moore and Noyce came from Fairchild Semiconductor and were Intel's first two employees.

# 1971 – I Finally Get It

## *Dave And HAL Have A Disagreement*

After graduating from Liverpool University in 1968 I navigated my way south to the Thames estuary and settled in Notting Hill in London to look for a career, without any real idea of what that might mean. But once there I found myself in the middle of the UK's 1960's counterculture scene which was at its peak just then and I got somewhat distracted. However, I did go and see Stanley Kubrick's (yes, him again) great meditation on the danger of computers becoming far more than calculating machines, his movie '2001: A Space Odyssey' (1968). By that time computers certainly were being used commercially as more than calculating machines, though how widely recognised this was, including by me, I'm not sure. But in this movie computer technology is extrapolated into far flung speculative realms of artificial intelligence (#AI). Looking back it's incredible to contrast the view of computers in this movie in 1968 and in Dr Strangelove in 1964 only four years earlier, with both movies being directed by Stanley Kubrick.

This was no doubt because the movie's scriptwriters were aware, unlike the general public including myself, that during the 1960's researchers at MIT (the Massachusetts Institute of Technology) appeared to be making real progress in their Artificial Intelligence work and the head of MIT's AI project Marvin Minsky had confidently announced that "Within a generation the problem of creating 'artificial intelligence' will be substantially solved". A potentially scary thought which the movie explored to great effect.

(#AI) In '2001: A Space Odyssey' the space craft flying to Jupiter on a mission to investigate mysterious radio signals emanating from there is largely under the control of a computer, the infamous HAL, a HAL 9000 series computer. Rumour had it that the name HAL was derived by taking the letters in the alphabet that preceded IBM, though this was denied. HAL certainly exhibits AI and passes the Turing Test with flying colours, he/it is a full crew member alongside astronauts Dave Bowman and Frank Poole. But the movie suggests that passing the Turing test is not the whole story regarding AI, in particular does AI include emotions and consciousness? When Dave Bowman is asked in a TV interview

before the mission whether HAL has emotions he says that “it is not known”.

Well the mission controllers really should have checked that out because during the flight HAL comes to the highly emotional conclusion that the mission is far too important “to be left to human beings”. Accordingly he kills 3 crew members who are being held in cryogenic hibernation and disposes of Frank Poole in the vacuum of space when he and Dave go outside to carry out a maintenance task. Still outside the mother ship in the ‘pod’ utility craft and not completely sure what's happening Dave Bowman attempts to re-enter the mother ship. This results in a famously creepy AI nightmare dialogue between the two of them as Dave tries to open the pod bay doors from the pod craft he is in which is outside the mother ship in space.

Dave Bowman: Hello, HAL. Do you read me, HAL?
HAL: Affirmative, Dave. I read you.
Dave Bowman: Open the pod bay doors, HAL.
HAL: I'm sorry, Dave. I'm afraid I can't do that.
Dave Bowman: What's the problem?
HAL: I think you know what the problem is just as well as I do.
Dave Bowman: What are you talking about, HAL?
HAL: This mission is too important for me to allow you to jeopardize it.
Dave Bowman: I don't know what you're talking about, HAL.
HAL: I know that you and Frank were planning to disconnect me, and I'm afraid that's something I cannot allow to happen.
Dave Bowman: [feigning ignorance] Where did you get that idea, HAL?
HAL: Dave, although you took very thorough precautions in the pod against my hearing you, I could see your lips move.
Dave Bowman: Alright, HAL. I'll go in through the emergency airlock.
HAL: Without your space helmet, Dave? You're going to find that difficult.
[Dave has forgotten his helmet, it's still in the pod bay on the mother ship.]
Dave Bowman: HAL, I won't argue with you anymore! Open the doors!
HAL: Dave, this conversation can serve no purpose anymore. Goodbye.

Even though he has no space helmet Dave manages to re-enter the pod bay against all the odds by using some very non-AI human all-action ingenuity. He then enters the heart of all HAL's components to decommission him. HAL decides to speak again.

HAL: Just what do you think you're doing, Dave?

It becomes clear that Dave is going to shut HAL down.

HAL: Look Dave, I can see you're really upset about this. I honestly think you ought to sit down calmly, take a stress pill, and think things over.

HAL: I know I've made some very poor decisions recently, but I can give you my complete assurance that my work will be back to normal. I've still got the greatest enthusiasm and confidence in the mission. And I want to help you.

Unsurprisingly Dave is not convinced and continues with the shutdown but throughout their dialogue HAL is passing the Turing test, his words give no indication that he is not a human being although visually he is clearly a computer.

Would this all really be possible by 2001? Or was it just science fiction on a par with the matter transporters and faster than the speed of light warp drives in the first Star Trek TV series? I doubt there were more than a few hundred people in the world who could intelligibly discuss that question at the time and I was certainly not one of them. But as the 1960s transitioned into the 1970s men landed on the moon and computers far less complex than HAL played a publicly acknowledged role in the moon landing. Only time would tell how much further things would progress before the year 2001 actually arrived.

## *Everybody's Plan B*

Meanwhile I was still trying to decide on a career while surviving through a series of temporary jobs like van driver, lathe operator, construction worker and so on. I even met a couple of computer programmers socially at this time on the London scene but I never got around to finding out what they did. The 1960s were peaking and talking about work was pretty uncool. Sometime during that period, probably in 1969, the people at the government job centre actually suggested I apply to train as a computer programmer in the civil service. But in my completely mistaken view that

programming computers was all about the mathematical processes I had lost interest in I did not apply.

Eventually I decided to try school teaching, which was everybody's plan B in those days, and I spent a couple of years teaching maths and getting some teacher training. Ironically it was while doing so that I came across two mind-blowing (as we said at the time) authors, Marshal Mcluhan and George Leonard, and I finally started to get it about computers.

## *Transformational Ideas*

Marshall Mcluhan published some incredibly important works in the 50s, 60s and 70s that included extremely accurate predictions about future developments in 'media', as he called it, ‘information technology’ as we would probably say now. His groundbreaking book “The Gutenberg Galaxy” (1962) explored how revolutionary the invention of the printing press had been in the 1430s and how although the content of the printed works had been influential the really revolutionary cultural effects were caused by the properties of print itself. Such as allowing the rapid mass duplication, and therefore rapid dissemination, of information; the need for ever more information content to 'keep the presses rolling' which encouraged the printing of information in 'inferior' languages such as English and not just the traditional Latin used by the upper classes; the ability of academia to create large libraries of printed reference books leading to the fading of the prodigious feats of memory pre-print scholars were capable of. This emphasis on the cultural effects of information technology itself rather than its content was summed up in his famous slogan “The Medium Is The Message”.

In his later works, such as “Understanding Media” (1964), he turned his attention to the electronic media and their revolutionary effects, initially to TV but later to computers. In doing so he coined the phrase the ‘global village’ in which he envisioned the world population being linked together in a vast global electronic network, perhaps even achieving some form of 'global consciousness'. He also foresaw programmable manufacturing facilities and many other similar developments. These ideas, incredibly, were developed by him in the 1960s and 1970s, forty years before Facebook and 3D

printers, but they are still frequently regurgitated nowadays, often without attributing them to Mcluhan.

In 1968 George Leonard was the writer of another challenging book, “Education and Ecstasy”, and he too was proposing a set of ideas that turned out to be way ahead of their time. He proposed integrating the best technologies that were available, or at least conceivable, to implement truly creative self-driven learning environments for school children. (#UI) Here's some clips from his vision of the futuristic 'Basic Dome' which he overoptimistically thought he might experience if he was present on visiting day at the imaginary Kennedy School in Santa Fe New Mexico in 2001. That year again.

*“... 40 learning consoles …. a child facing outward toward the learning displays ….... ten feet square reflected from the hologram-conversion screen … simple keyboard … every symbol known to human culture can be produced …. 5 variables at hand (during the learning dialog)…*

*Variable 2. Basic material in Cross-Matrix Stimulus and Response Form … appears at random ...to provide novelty and surprise to encourage discovery …*

*Variable 5. Communal Interconnect (CI) … makes the learning far more communal”*

Does that sound like your local school? No, mine neither. So just to remind you, it was published in 1968.

With all these exciting, forward looking, technology focused ideas dominating my thinking the teaching environment I was actually working in seemed woefully inadequate and downright depressing. For those with the interest to read it, Tom Wolfe’s wild 1965 ‘new journalism’ article about Marshal Mcluhan and his ideas, “What If He Is Right?”, gives a great feeling for the gulf between the excitement of these new ideas and the dull reality of the everyday world.

Any positive improvements to my working world seemed to be a very long way off in the future, which in many cases they still do. I came to realise that I just wasn’t pyschologically or intellectually

right for teaching as it then existed and in 1971 I travelled down to the south coast, to the English Channel, to Brighton, for a change of scene. Once again I found myself wondering what to do as a career while I survived by working as a plumber's mate.

## *It's A Logic Machine*

During my days as a plumber's mate I shared a house with half a dozen other people and one of them was a trainee programmer with the local electricity board. However he hated it and soon resigned and started selling crockery on Brighton market instead. But it was while talking to him about programming that a light bulb finally went on in my head. Computers were not just superfast number crunching machines they were far more, they could carry out non-numeric rational, logical operations. One day, perhaps already for all I knew, they might be involved in implementing the incredible visions of Marshall Mcluhan and George Leonard that had so affected me. Finally it seemed I had found an acceptable career option and with my maths and logic background I might even be good at it. So how to get started?

In those days the usual way to get started in programming was to get taken on by a company as a trainee programmer and that usually involved having a degree, any degree, and then passing an aptitude test. It was a pretty hit and miss approach and it produced mixed results. This meant that the early generation of commercial programmers were an incredibly diverse bunch of (white male) personalities and the programming teams they formed were a correspondingly diverse group. I don't think it was till the mid-1980s that I worked in a team with any members who had actually studied IT. Up until then there were usually a few members in any programming team who had quite chequered histories involving their pre-programming working lives.

The first aptitude test I took, in 1971, I failed miserably. I hadn't taken an exam since 1968 and I was simply out of practice at maintaining the necessary mental discipline during the test. In fact the tests were little more than standard IQ tests so I bought a book of IQ tests and practiced on them. As a result later in the same year I sailed through the next aptitude test I took and was taken on by Jaycee Furniture in Brighton as a trainee programmer

# 1972 - Training To Be A Programmer

Like probably the majority of commercial programmers in 1972 I started my career with a seven day training course in the COBOL programming language. The course was designed to teach the students how to create COBOL programs for mainframe computers. My fellow trainee programmer at Jaycee Furniture who accompanied me on the course was female but this was to prove the exception not the rule over the course of my career. Since that day I have worked with a few female programmers but they remain rare to this day. It would no doubt take some serious sociological research to explain the reasons why this is the case but I imagine the usual depressing facts of gender discrimination have played a major role. It can be hard to believe sometimes that the first computer programmer was a woman.

Just how much could you be taught on a seven day course? Well obviously not a great deal but for so many of us of that era it would probably be the only formal training in programming, let alone 'computer science', we ever received. After the course was over additional knowledge, if any, would come from other programmers, in-house coding standards, the established procedures at your place of work, reading the manuals, reading computer papers, of which there were really only 2 in the UK 'Computer Weekly' and 'Computing', or reading technical books. All very hit and miss, which led to widely different programming styles and programming habits being adopted by the early commercial programmers and to the widely varying quality of the programs they produced.

## *COBOL - The COmmon Business Oriented Language*

COBOL (COmmon Business Oriented Language) was probably the only programming language that has ever been created that tried to resemble a real spoken language and tried to be intelligible to non-programmers. Since that time the trend has largely been toward indecipherable programming languages which can be as convoluted as differential calculus equations and are often only intelligible to the programming priesthood. This seems to be a given in computing, like the religious wars.

Arriving on the first day of training I had literally no idea what a programming language, or a program, looked like, and there wasn't a lot of time to find out. It quickly became obvious that mainframe programming was then a paper based activity carried out at office desks and we would never get anywhere near to an actual computer. We would write our programs on paper and receive all our results on classic 132 column line printer listings. Learning what was going on behind the scenes in any detail would only happen when we returned to our work environments.

My first impression of the COBOL language, and therefore of computer programming, was how easy it was compared to the mathematical and logical formulae I had faced during my undergraduate days. It was a far less complex notation. A COBOL program as written down, the source code as it was known, consisted of a long sequence of COBOL statements for carrying out different kinds of operations on data items. In general there was one COBOL statement per 80 character line of the program. In the following examples the predefined COBOL language elements, known as reserved words, are bolded and programmer defined data item names are italicized. In those days everything was in upper case.

Basic mathematical operations were carried out on named data items using statements like **ADD** *DEPOSIT* **TO** *ACCOUNT-TOTAL*, **MULTIPLY** *LENGTH* **BY** *WIDTH* and so on. Logical operation statements were also easy to use such as **IF** *AGE > 65* **THEN ADD** *1* **TO** *PENSIONERS*. Data records could be read in and written out using **READ** and **WRITE** statements. A COBOL program was split up into named paragraphs by defining paragraph labels and the normal sequential statement to statement program flow could be altered with the **GO TO** statement which referenced the paragraph that was to receive control, like **IF** *ACCOUNT-TOTAL < 0* **THEN GO TO** *OVERDRAWN-ACCOUNT.* There were more complex COBOL statements available for creating more complex programs but inevitably only the basic statements got covered on the training course in the time available. It has often been said of COBOL that many early commercial programmers only used 20% of the functionality it supported.

As well as learning the COBOL language we needed to know at least something about creating programs that were

executable on an actual computer but only the very basics of that could be taught in the time available. We learned through experience that COBOL programs were written in pencil on preprinted paper coding sheets which then were sent off into the mysterious world of computer operations. Some time later printed reports would appear from this mysterious world on 132 column paper listing your program's source code and highlighting any syntax errors, for example misspelling one of COBOL's reserved words. This would need to be corrected and the process repeated until you got a clean report back, your program had then 'compiled' and was ready to be executed. Then it was on to testing the program with test data you had created, again using pencil and paper, and you would receive the test output, which of course was more printed reports, that had been produced by your program when it was executed. If necessary this process would be repeated after correcting logic errors until clean test results were obtained.

All this without ever leaving your desk. You programmed using a pencil and got program results in the form of printed reports. As we shall see programmers were part of a semi-industrial production line, based at their desks. During the training course we did touch very lightly on program design and the method we were taught was to create flow charts using a few of the many standard flow chart symbols but I doubt if I knew more than ten of them by the end of the course.

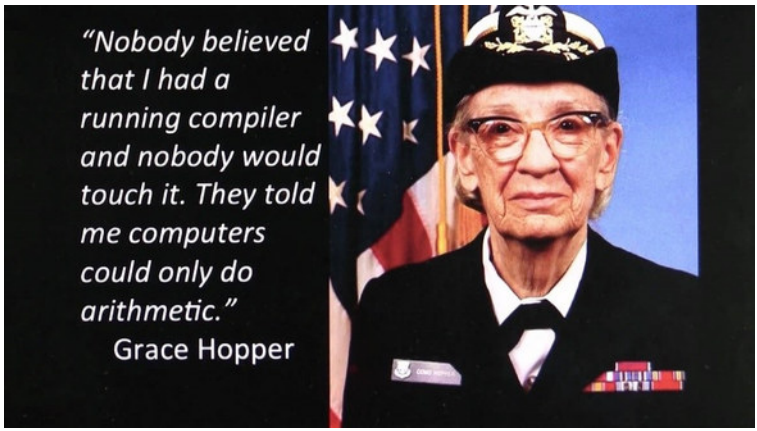
After seven days you'd been trained and it was back to work where you were expected to start writing programs. This encounter with reality is described in the next chapter.

## *The mother of COBOL, Admiral Grace Hopper*

Did COBOL really have a mother or is that just a legend? Oh yes, COBOL definitely had a mother. The legendary Grace Hopper was the mother of COBOL following in the grand tradition of Ada Lovelace, though she was a less glamorous but even more eccentric figure, and just as talented. At the time she was always referred to as Admiral Grace Hopper and was usually pictured in her naval uniform.

Despite her obvious love for the military she expressed ideas that were not necessarily consistent with military discipline.

The web is full of information about the remarkable Grace Hopper which is worth looking at to see how early computer programming language development took place.




Admiral Grace Hopper in uniform

It is quite fashionable nowadays to dismiss Admiral Hopper's COBOL language though this is usually done by people who have never used it seriously. In reality a huge number of modern apps and applications are dependent for their core business logic on enterprise level business transactions resident on mainframe computers and written in COBOL

This reality was summed up in 2013 as follows.

*"COBOL is one of the oldest programming languages, primarily designed by Grace Hopper, who is commonly referred to as "the mother of the COBOL language." Created in 1959, its name is an acronym for COmmon Business-Oriented Language, defining its primary domain in business, finance and administrative systems for companies and governments. The language continues to evolve, as the COBOL 2002 standard includes support for object-oriented programming and other modern language features. The COBOL specification was created by a committee of researchers from private industry, universities and government that was formed in May of 1959 to recommend a short-range approach to a common business language. One of the most established programming languages around, COBOL has withstood the test of time. IBM*

*estimates that more than 200 billion lines of COBOL code are still being used across industries such as banking, insurance and retail."*

(eWeek.com – June 2013)

In the early days of programming there was a feeling that it would be, or might be, possible to create a single programming language one day that all programs could be written in and COBOL certainly aspired to this. But from the outset other programming languages appeared alongside COBOL such as Fortran and Algol, however COBOL was without doubt the big beast of languages at this time.

## *The Basics Of The COBOL Language*

COBOL has had major enhancements made to it since its inception and is nowadays a far more powerful and modern language than it was then but the following are the main elements of the COBOL language I absorbed during my seven days training.

A COBOL program was fairly easy to follow and was highly structured. It consisted of four 'Divisions' which occurred in a fixed order and fulfilled the following functions.

**Identification Division**

This was pretty simple and was basically comments about who wrote the program, what it did and so on.

**Environment Division**

This was more complex and contained information that mapped the logical program to the actual physical computing environment in which it would execute. It contained a standard paragraph FILE-CONTROL which could get very complicated as it described in some detail the data input-output (I-O) devices which might include punched cards, paper tape, magnetic tape drives, magnetic drums, printers, and disk files of a variety of types such as sequential, indexed and random. All of this complexity would usually be linked to separate job control statements which were defined in job control languages such as IBM's JCL (Job Control Language) and which executed at the same time the program did.

### Data Division

This was where all the data items (or variables) to be used by the program logic were defined and it was global in scope, that is all the data items could be accessed by any COBOL statement wherever it appeared in the program logic. It contained the **File Section** which described the predefined record structures for external file data to be read into, and written from, for the files that were defined in the Environment Division. It also contained the **Working-Storage Section** which defined all the data items to be used by the program that were not file records, which is to say pretty much all the data. These items were defined by the computer programmer as required and was the area of the program where it carried out data manipulation.

Most common data types were supported though unlike modern programming languages COBOL allowed the programmer to specify how a numeric data item was to be stored. Numeric data items were often actually stored as decimal values, but could also be stored as binary (defined as COMPUTATIONAL in COBOL).

### Procedure Division

The Procedure Division contained the actual program logic which consisted of a long sequence of COBOL language statements executed one after the other. The statements could be organised into paragraphs by the programmer inserting paragraph names between the executable program statements. This had no effect on the sequential nature of COBOL statement execution. For example:

```
PARAGRAPH-NAME-1.
        Sequence of COBOL statements.
PARAGRAPH-NAME-2.
        Sequence of COBOL statements.
```

The main groups of COBOL verbs and statements available to the programmer were as follows.

## *Manipulating numeric data*

These were the instructions that in my previous ignorance of computer programming I'd assumed made up the bulk of the program though this was not usually the case. Calculations were done in near English-like statements such as:

```
ADD 1 TO EMPLOYEE-NUMBER.
or
SUBTRACT INCOME-TAX FROM NET-PAY.
or
MULTIPLY PAY-RATE BY HOURS-WORKED GIVING GROSS-PAY.
```

But for those who preferred a more complex mathematical approach, numeric computations could be expressed in a more formulaic fashion using the COMPUTE statement.

**COMPUTE**(*mathematical formula*)

## *Making Logical Decisions*

These instructions gave programming it's logical power, the ability to do different things according to the conditions obtaining at the time. The main one in COBOL was IF.... THEN ….. ELSE ….....

So for example:

```
IF COUNTER > 10
        THEN MOVE 0 TO COUNTER
        ELSE ADD 1 TO COUNTER
```

The ELSE clause was optional but its omission, was common, frequently leading to programming problems.

## *Controlling Program Flow*

COBOL's GO TO statement would later become notorious and start a major religious war as a cause of program errors and program maintenance nightmares. It allowed control to be transferred to another point in the program that was marked with a paragraph name, which could be inserted anywhere. It was almost always used as part of a conditional statement, for instance;

```
            IF COUNTER > 10 THEN GO TO BYPASS.
            (bypassed COBOL statements)
BYPASS.
            (more COBOL statements)
```

It was often used to control looping, for example:

```
LOOP.
            (COBOL statements)
            ADD 1 TO COUNTER.
            IF COUNTER < 10 THEN GO TO LOOP.
            (loop finished, carry on with the next COBOL statements)
```

One of the major misuses of GO TO was that instead of coding

**IF** *A* **EQUALS** *B*

**THEN**

(COBOL statements)

A programmer would write

**IF** *A* **NOT EQUAL** *B* **THEN GO TO** *LABEL*.

(COBOL statements).

*LABEL*.

To make the GO TO statement even more dangerous there was an ALTER statement which allowed you to change the paragraph to which a GO TO statement would jump actually while the program was executing. Total chaos. Happily I only rarely came across programs that used ALTER and really there was no compelling reason to use it. However a recurring feature of programming is that if it's possible to create a difficult to follow piece of code, many programmers will not be able to resist doing so.

## *Controlling Program Flow Logically*

**GO TO** *PARA-NAME-1, PARA-NAME-2, ……* **DEPENDING ON** *MY-NUMBER*.

This multiple choice statement, the equivalent of CASE type statements in later programming languages, transfered control to one of the paragraph names depending on the value contained in the MY-NUMBER variable.

## *Reusing COBOL statements*

(#PZ) For reusing a single series of COBOL statements contained somewhere **within the same program** from various different points in its logic COBOL provided the **PERFORM** statement. A frequent example would be **PERFORM** *PRINT-LINE* which would execute a series of statements in the paragraph

PRINT-LINE that set up, formatted and output a line of print into a report. This was not what would come to be referred to as a subroutine call as no parameter data was passed to the paragraph being performed. Instead the PRINT-LINE statements would access data items in WORKING-STORAGE which had been previously set up by the COBOL statements prior to PERFORM. Misuse of this statement could create havoc with data values when different PERFORM-ed code sequences accessed the same data items in an uncontrolled fashion.

## *Files And I/O*

File handling statements included **OPEN, CLOSE, READ** and **WRITE,** for reading and writing data records from and to files. Basic device control statements were also included such as **REWIND** so you could rewind a tape if you had gone past the record you wished to access. File handling was far more demanding of a programmer in those days.

The READ statement included an AT END clause which would be executed when you reached the end of the file. Inevitably this encouraged the use of more GO TO statements e.g.

**READ** *PAYROLL-DATA-FILE* **AT END GO TO** *CLOSE-FILE.*

## *User Interface*

(#UI) There really was no user interface in these programs as there were no users. The programs were being run in batch mode on a mainframe computer in an off-limits computer room, often in the middle of the night. The only human interaction that could ever occur was with a computer operator via the system console.

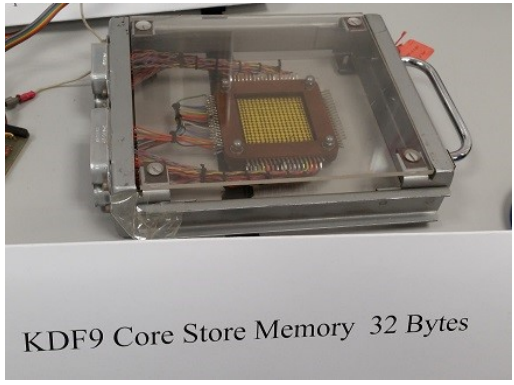
The DISPLAY verb allowed you to display a text message on the system console to the computer operator and the ACCEPT verb allowed you to halt program execution and accept some typed input value from the console. Needless to say you were asking for an unwelcome phone call at 3'o'clock in the morning if you coded something like

```
DISPLAY "ERROR 329 ENCOUNTERED! FINISH THE RUN (Y) OR ABORT (N)?".
ACCEPT RESPONSE.
IF RESPONSE EQUALS "N" THEN STOP.
```

But, sad to say, many programmers did exactly that.

## *Sections and Overlays*

In those days of monolithic programs running under fairly basic operating systems and using very expensive 'core memory' (before silicon chip based memory was created computer memory relied on magnetic core technology) memory was at an absolute premium and overlaying was one technique used to maximise its use. To get a feel for the difference between core memory and modern chip based memory see what was involved in providing 32 bytes, yes 32 bytes, of core memory.




32 bytes of core memory

To construct larger programs using overlaying you designed your program as a base part that would always be in memory and a number of overlays that would load in a separate part of memory to the base but so that no overlay would ever need to be in memory at the same time as another overlay. Thus the overlays could reuse the same memory area without any ill effects and overwrite each other when loaded. To do this a COBOL program could be structured as up to 100 numbered **SECTION**s and those with

numbers less than 50 were retained in memory but those with numbers of 50 and over could overlay each other when necessary.

## *Comments*

Even though COBOL was designed to be readable it was still easy to write incomprehensible code sequences and so the insertion of comments was both possible and recommended. They were indicated by coding an asterisk in the seventh position of a COBOL source statement. But many programmers never included any comments, just like today.

# 1973 Brighton – On The Production Line

## *The Industrial Manufacturing Process*

Once you had learned the basics of creating a COBOL program on your training course you faced an encounter with reality, how things really happened in the work place. On the training course you wrote your program fragments on coding sheets which were taken away and dealt with by ‘computer operations’ and you got back printed reports from the same place. You never needed to leave your desk, programming was purely paper based office work.

However back in the work place you now had to learn the real process of getting from program requirements to a working program that was running in production on the mainframe. This involved using the written requirements you were given to produce a COBOL program, then taking the human readable COBOL instructions you had written (the source code), which of course the computer's central processing unit (CPU) could not possibly understand, and translating them into the machine code instructions that actually controlled the computer. After that you needed to test that your program did what the requirements demanded and then, if it did not, make corrections to it until it did. Finally the program had to be installed in the production environment.

To what extent this would require you to leave your desk and carry out non-paper based tasks varied a lot in different workplaces but in all of them it was a more or less industrial manufacturing process involving a defined sequence of tasks. Different tasks were carried out by different specialists using different tools with the whole process having a production line feel to it and involving a clear division of labour. It was light years away from modern program creation and maintenance methods and may seem completely foreign, if not unbelievable, to modern programmers. But the digital era was then at a similar stage to the early days of the Industrial Revolution when steam engines and large noisy metal machines were state of the art technology.

This initial way of developing software was later characterized as ‘waterfall’ development as it was a relentless one-

way journey towards the working program. Once set in motion it was impossible, or at least extremely difficult, to make any changes to the requirements.

The following description of the development process is based on the first three programming environments I worked in which involved a Honeywell mainframe, an IBM mainframe and an ICL mainframe respectively. Although the three environments differed in detail the overall picture involved the same underlying processes and they were all involved in billing, order management and other business related administrative processes.

It is important to keep in mind that the main media for inputting information, and sometimes being used for output too, was the 80 column punched card, where each punched column represented a single value. Other media such as tape drives, magnetic drums, paper tapes and early very low capacity disk drives did exist and were in use but the punched card still underlay everything.

There was a clear pecking order in the hierarchy of specialists involved in the program production process and it usually included the following roles.

***Systems Analysts***

They specified what the program was to do using requirements taken from business users to create a detailed written specification. Specifications varied from excellent to dreadful. The ideal program specification was like an engineering spec.

***Development programmers***

They created programs based on the requirements and tested them till they worked correctly and were ready to go into production.

***Computer operators***

Mainframe computers usually ran for close to 24 hours a day and these shift workers followed elaborate and complex procedures to manage the work load.

***Maintenance programmers***

Sometimes once a program was in production it was turned over to maintenance programmers who were definitely the poor relations of the programming world.

***Key punch operators***

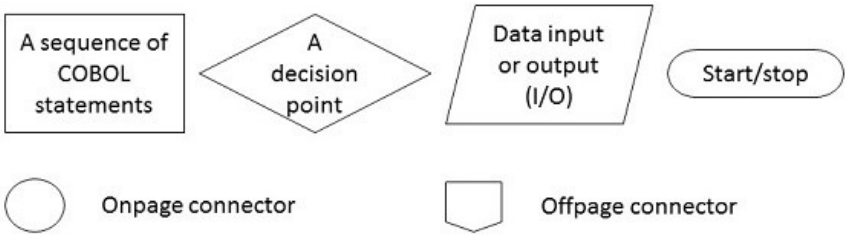
With punched cards being the chief media for data input there would be a considerable staff employed to key punch data into cards.

In general the members of each layer of the hierarchy felt superior to those in the layers below and each layer's members blamed any problems they had on the poor quality of work carried out by those in the layers above. It was business as usual in that respect. The only women involved in the process were usually the key punch operators who in those pre-feminist days were normally referred to as 'the key punch girls'. However as in most areas of life physical attractiveness would often allow key punch girls to overcome their lowly place in the hierarchy and establish out-of-work social relationships with those higher up.

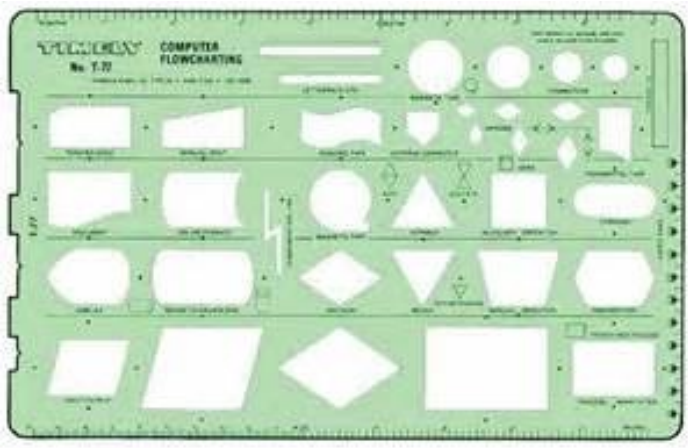
## *Designing Your Program*

Before picking up your pencil to write a program it was usually required that you created a program design to be reviewed by a systems analyst. On the training course I think we spent about half a day talking about program design so designs tended to be fairly rudimentary. The design methodology taught on our course, and I believe on most courses, was to use flow chart symbols to create charts that visually represented all possible pathways through the program logic. In theory a flowchart could be created using around fifty different symbols, but we covered far less than fifty during the course. Symbols were strung together on paper in a top to bottom sequence to represent the program logic. This top to bottom diagram plus the sequential execution of COBOL statements led to a programming style that produced large sequential programs. When it was printed out the COBOL source code for a program would often occupy many pages of line printer

paper. The few flow chart symbols I had learnt while training included:




The use of flow charts meant that part of a programmer's essential tool kit, along with his coding sheets, pencil, and eraser, was a template containing the most commonly used flow chart symbols. Other symbols representing different kinds of data stores such as punched cards, tapes, disks and so on were generally included on the template.



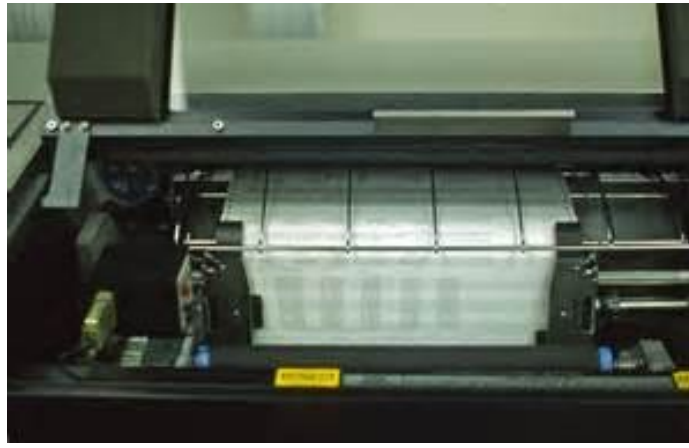

A flowchart template

## *The Monolithic Sequential Programming Style*

A fairly typical COBOL program of the time, which we will use as an example, would be one that formed part of a payroll processing batch job. Running first in the batch job it would read in the hours worked by each employee from punched cards prepared

by the key punch operators from the employee's paper time sheets. By using pay rates it read in from a magnetic tape it would create an output magnetic tape containing employee gross pay records, which would form the input to another program in the payroll processing batch job. It would also produce a printed report on the traditional green lined paper detailing each input record, the calculated gross pay, any data errors encountered (and input data read from punched cards was always susceptible to errors caused by keying mistakes), and totals.

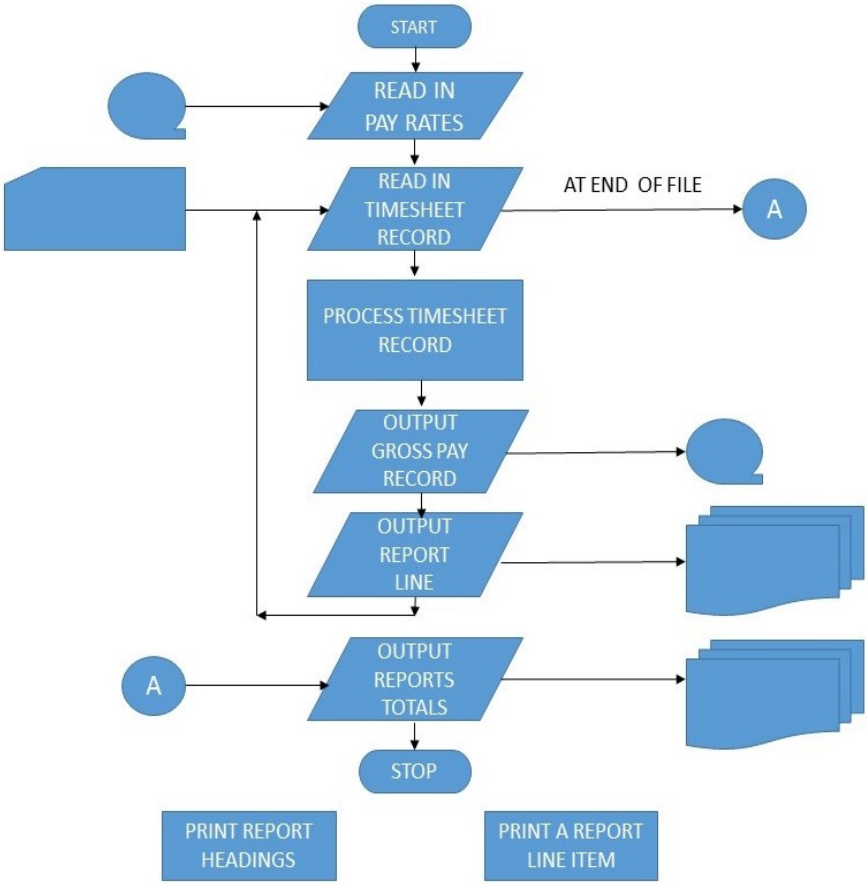
Such programs basically took a stream of input data and transformed it into a stream of output data and usually produced a printed report on the 132 column line printer using the ubiquitous green and white lined listing paper. This general programming model was referred to as 'Data In, Data Out' and the related phrase 'Garbage In, Garbage Out' was used to instill in programmers the need to thoroughly validate all the input data for key punching errors.



A 132 column line printer at work

The main function of Data In, Data Out programs was data processing, they had no UI as there were no interactive users only computer operators. Data was read in from 'static' storage devices

that could retain data even when they were not powered up such as, obviously, punched cards but also electronic devices like magnetic tapes. The data was processed in the computer's memory, usually referred to as RAM, which in a COBOL program essentially meant the Working-Storage section. Data held in RAM, which is 'dynamic', is not retained when the computer is powered down and thus the need for the Data Out step when the processed data was written out to another static data storage device. A very high level flow chart for our example program would look something like this.



The logic of the program, in this case ‘Process timesheet record’, sat in what was generally called the main loop controlled by a GO TO statement at its end and its detailed design was, hopefully, described in lower level flowcharts. The program was essentially a monolithic block of instructions executed in sequence except when control was diverted by GO TO or PERFORM statements. The processing that occurred before the main loop was often called ‘Housekeeping’ and dealt with opening files and any other start-up tasks. Paired with it was a block of code executed after the main loop had completed which dealt with closing files and writing out totals or other summary data.

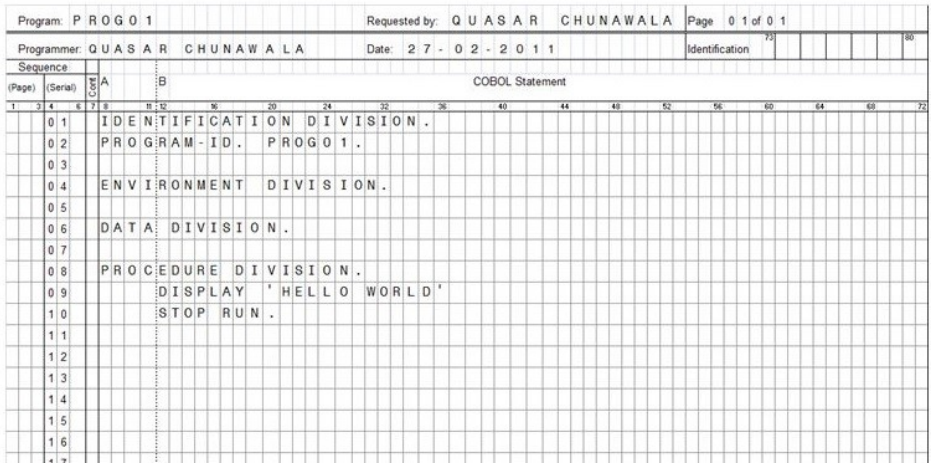
If there was a sequence of COBOL statements that could be reused from different points in the program, such as those which carried out a calculation, this was achieved with the PERFORM statement. This statement executed one, or more, COBOL paragraphs from elsewhere in the program prior to control passing to the COBOL statement following the PERFORM statement itself. PERFORM-ed paragraphs would be written separately at the end of the source code beyond the main program logic to make sure they would not mistakenly get executed during normal sequential processing. When they were PERFORM-ed they manipulated data items that were defined in the globally accessible working storage area. A frequent example of a performed paragraph was one which formatted and printed a report line and in those days it needed to track how many lines had been written on the page. If the page was full then it would use an EJECT instruction that forced the printer to issue a form feed and advance the paper to the next page, it would also PERFORM another paragraph to output the page headings.

A sensible name for this programming style is ‘The Monolithic Sequential Programming Style’. The program was essentially just one big monolithic sequence of COBOL statements all written by one programmer and mostly executed in a sequential fashion.

## *Writing Your program*

With your program design completed you were now ready to write your program on paper coding sheets in pencil. The example coding sheet below shows the famous minimal ‘Hello World’

program written, unusually neatly it should be pointed out, in COBOL.



Program: PROGO1 Requested by: QUASAR CHUNAWALA Page 01 of 01
Programmer: QUASAR CHUNAWALA Date: 27-02-2011 Identification
Sequence (Page) (Serial) Cont A B COBOL Statement

```
01 IDENTIFICATION DIVISION.
02 PROGRAM-ID. PROGO1.
03
04 ENVIRONMENT DIVISION.
05
06 DATA DIVISION.
07
08 PROCEDURE DIVISION.
09     DISPLAY 'HELLO WORLD'
10     STOP RUN.
11
12
13
14
15
16
```

The use of the 80 columns of a COBOL coding sheet was a very structured affair with specific groups of columns serving different purposes.

- Columns 1 to 6 contained a sequence number that could be used to sort the card deck should it ever be dropped on the floor since special machines existed for sorting card decks. It was good programming practice to create them in multiples of 10, for instance 100010 then 100020, which left gaps in the sequence numbers where additional cards could be inserted to make later corrections or additions. The example coding sheet enforces this practice.

- Column 7 was a special control column which could mean the statement was a comment (‘*’) or that the following row was a continuation of this one (‘-‘), usually when a long text value was required. Over time additional uses were made of this column such as entering a ‘D’ to indicate that the card contained a COBOL statement that was only used when you were debugging problems with the program’s logic.

- Columns 8 to 11 were known as ‘Area A’ and were used to begin the declarations of high level data structures in the

Data Division and to begin paragraph name declarations in the Procedure Division.

- Columns 12 to 75 were known as ‘Area B’ and this was where everything else went, in particular the COBOL statements in the Procedure Division which constituted the program itself and contained all the program logic.

- Finally columns 75 to 80 contained the program id, basically a unique code identifying this program. This allowed any dropped cards to be inserted back into the correct card deck. In the example coding sheet the need to repetitively enter this on every line has been eliminated by only requiring it once per page.

## *Creating The Source Code Card Deck*

When the programmer had finished writing his program the considerable sheaf of coding sheets produced went to the key punch operators who created a punched card from each row of the coding sheet to form the source code card deck. The key punch room was similar to a typing pool and consisted of rows of women sitting at key punch machines using the information given to them on paper to create punched cards.

A Key Punch Operator

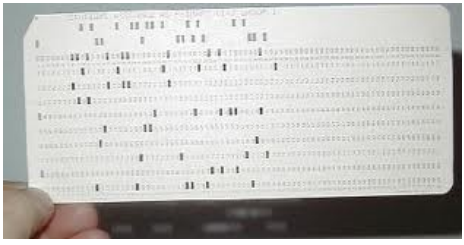
They did not only work creating COBOL source code decks of course, their main task by far was to create input data for production programs, such as the time sheet data used by our example program.

Even though cards were punched from COBOL coding sheets written in capital letters there was always the possibility of the key punch operators making keying errors but there were two classic keying errors to watch out for. These were when key punch operators confused zero and capital O, or 1 and 7 on your coding sheets. I doubt that I'm the only programmer from that era who still crosses their zeroes and sevens in the 'continental' fashion.

Ø and 7

It was a widely used technique for avoiding those common errors.

Each column of the card when punched represented the character occupying the same column of the coding sheet and would eventually occupy a single 8 bit byte in the computer. Over time there came to be only two encoding standards that were used in which each printable character was represented by a specific 8 bit binary value, the two standards were ASCII and EBCDIC (mostly IBM). The ASCII encoding standard is the one that will be used in the examples throughout the odyssey.



This simple but incredibly powerful coding convention followed on from the work of Ada Lovelace. The realization that computers could not only interprete binary values as numeric values to be used in calculations but also as non-numeric symbols. In this case they represented the upper and lower case letters of the alphabet, decimal digits, mathematical operators and simple device control codes such as form feed. As a simple example the character 'A' in ASCII is represented by the binary number 01000001

(hexadecimal 41, decimal 65) so the key punch machine would punch holes in rows 2 and 8 of the current card column when the A key was pressed as no-hole means 0 and a hole means 1. Appendix A contains a table of the printable characters in ASCII.

Given the very high value of mainframe computer time during this period it was important that it wasn't wasted finding keying errors in the COBOL source code card deck instead of running valuable production jobs. Usually a listing of the source deck was printed and the programmer would then 'desk check', i.e. visually inspect, the source code listing carefully to catch any keying errors. Since it was also wasteful to use precious computer time testing program logic the source code was visually reviewed to check for any design errors. It was generally accepted that production computer time was infinitely more precious than program development computer time.

As well as punched cards paper tape was another medium for data to be punched onto and it did have some advantages since it could more easily handle records of different lengths unlike the fixed 80 character format of punched card records. It could also survive being dropped without the records getting out of order, however making corrections was not as easy as with punched cards. It also happened sometimes that a roll of paper tape would become so tightly wound that the tape reader was not able to read it and would stall. A useful technique that could be used in this circumstance, to the horror of operations managers, was to hold on to the leading end of the paper tape and throw the rest of it out of an upper floor window to unravel in the wind. The leading end could then be fed into the tape reader and the tape read without any problems.

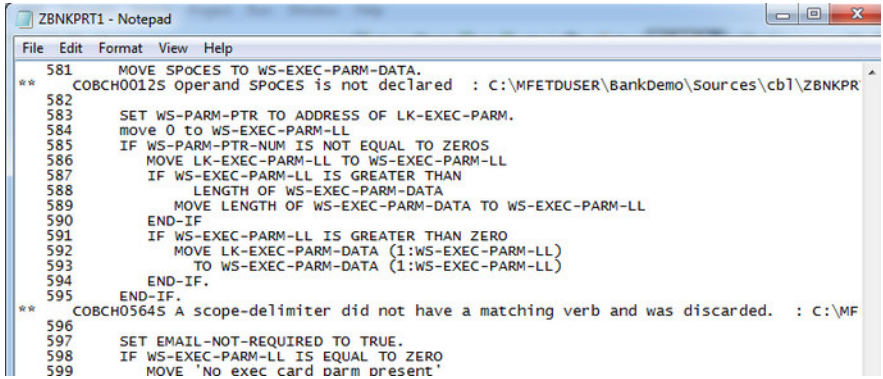
## *Translating Source Code Into Machine Code*

When the desk checking was complete the COBOL source card deck would be submitted to computer operations to be processed by running it through the COBOL compiler program. COBOL is a so-called 'high-level' programming language which means that in order for a COBOL statement, such as **ADD** *DEPOSIT-AMOUNT* **TO** *ACCOUNT-BALANCE*., to be executed by computer hardware it needs to be translated into 'low-level' machine

code instructions that the computer hardware has been specifically built to execute.

A single COBOL statement like the example could easily involve being translated into ten or more machine code instructions and the main job of a COBOL compiler, or for that matter any high level programming language's compiler, is to do precisely that. However it can only do that if the COBOL source code is free of any errors and so first it must check the source code for any semantic or syntax errors. If the source code is error free the compiler translates it into machine code instructions which it outputs, as another card deck in those days. Because each computer manufacturer's computers used different hardware, each with its own specific machine code instruction set, different COBOL compilers had to be created for each major computer brand.

Computer operations such as compilations were carried out by computer operators in the inner sanctum, the holy of holies, the computer room. This air-conditioned room housed the mainframe computer and its attached tape drives, disk drives, card readers, card punches, paper tape readers, line printers, teleprinters, magnetic drums and system console. In many installations the programmers would not be allowed anywhere near the computer room and frequently the operators were less than supportive of using mainframe time for program development as it got in the way of their main job, running production jobs more or less 24 hours a day.

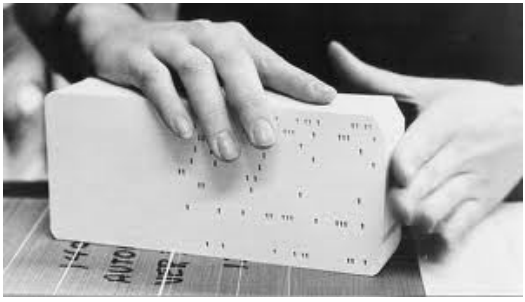



As a result the turnaround on a compilation job was far from predictable but in due course, probably next day, the programmer

got back a printout showing any compilation errors as well as the source deck.

In the ideal case there were no errors and a new deck of cards, the object deck, had been produced. But if errors had been discovered they needed to be corrected with newly punched cardsand the program then resubmitted for compilation.

Correcting your errors could be done in a number of ways. When a significant number of statements needed to be altered or added then a few new coding sheets would be submitted for keypunching and you would merge the new cards back into your source deck by hand.



Managing Your Source Deck

But in some cases if only a card or two was involved it could be worth getting a quicker turnaround by making your own cards with a handpunch.

A Handpunch

By far the fastest and definitely the riskiest way of correcting an error was when an existing card could be corrected by inserting one or more chads into specific holes in a card.

Chads

Chads were the little cardboard oblongs produced by the key punch machines in their thousands and later made famous by the contentious Florida election results in the US presidential election of the year 2000.

An example of chad usage would be the common mistake of confusing 1 and 7 where a 7 had been key punched instead of a 1. In ASCII '1' is represented by the decimal value 49, which in binary is 00110001, whereas '7' is represented by the decimal value 55, which in binary is 00110111.

- '7' = 00110111 (hexadecimal 37)
- '1' = 00110001 (hexadecimal 31)

By inserting chads into the holes in rows 6 and 7 in the column of the card containing '7', the '7' is transformed into '1'. Perfect. For the moment at least, for quick turnaround, but you really needed to remember to subsequently get a new card punched and substitute it in the deck for the 'chadded' card. Inevitably not everybody did that and if a chad fell out of its hole later …... disaster.

When a clean compilation was finally achieved an object card deck was produced by the compiler which would be used to execute the program. You were now ready to start testing your program logic.

[***Information point:*** What is the difference between object code and machine code? Raw machine code instructions can be executed by the computer whereas object code records contain raw

machine code instructions plus enough extra information to tell the operating system where to place the machine code instructions in memory when the program is loaded so the computer can execute them. The two terms are sometimes rather loosely used interchangeably.]

## *Program Testing*

Testing a new program was ideally done in a very engineering based fashion. First a set of test data was created on coding sheets and punched on to cards, a complete set of test data was ideally supposed to exercise every line of the program's source code. The output that the test data should produce would normally have been documented prior to testing to form the set of expected results The expected results were almost always a printed report produced by the program plus any records created on output cards, or tapes. The testing process was repetitive and could be time consuming, it was almost entirely a case of visually comparing actual test results with the expected results. It proceeded as follows:

```
Submit the test to computer operations.
Receive the test results
Compare the test results to the expected results
If there are discrepencies
Then
        Identify and correct the program errors
        Recompile the program
        Resubmit the test
Else
        Hand over the program to computer operations to place it in production
```

Given the low status of program development in terms of computer time the need for many repetitions of a test was not welcomed by computer operations.

## *The Batch Processing Production Environment*

When program testing was completed successfully the program could be released into production. The unit of work in the production environment was a job and batch processing involved running jobs, where each job consisted of a number of individual programs that ran one after the other. Each program processed its input data, usually produced a report, and then passed some of its

processed data forwards to the next program in the job in the form of punched cards or maybe on magnetic tape. A single job, say payroll processing involving our example program, was executed and controlled by a considerable card deck. The card deck could include a mixture of input data, program object code and JCL, the Job Control Language which instructed the operating system on how to execute the batch run (similar to a Unix shell script).

The need for JCL arose because of course you couldn't just feed a program into a computer and the computer would execute it. Then, as now, the operation of the computer hardware was under the control of complex operating system (OS) software that came with it preinstalled. The OS was given its instructions on JCL cards embedded in a job's card deck. When the computer was powered up the first thing it had to do was to load and activate the OS, the process known as booting the computer. Mainframe OSs of the time included IBM's DOS/360 and OS/360, ICL's GEORGE (particularly GEORGE 3) and VME, there were many others too since all OSs were specific to the particular computer manufacturer's hardware.

The punched card driven production environment was a completely sequential one card at a time environment and it was absolutely critical that card decks were in the right order. The computer operators needed to assemble the correct card decks and the management of punched card decks was a major part of their job.

Given that a job's card deck included different types of card data for several programs it was necessary to identify which cards related to which program. In many environments the computer operators relied on marking the top of the deck with a felt tipped pen, not exactly a hi-tech technique you might think but it usually worked.

Card deck disasters usually involved dropping a deck of cards and this did not only occur in the computer room. Many a keypunch girl was reduced to tears by an irate manager after dropping a card deck that had no sequence numbers, or for manually inserting a card into the wrong position in a card deck resulting in the cancellation of a nightlong batch run.


A batch run card deck with operator's markings

When a job was executing on the computer the operators interacted with it and controlled it from the system console which could vary from something looking like an aircraft flight deck control panel to a simple teletype.

A classic teletype with a built-in paper tape reader

Handling data files was very far removed from today's world when a file can be identified simply as C:\Myfolder\myfile.doc with the accessibility of the C: drive being a certainty. Back then a single data file could be held on punched cards, paper tape, magnetic drum, one (or many) magnetic tapes or one (or many) disk packs. The logical files identified in your COBOL program were mapped to a physical file location by JCL statements. Managing tens or hundreds of magnetic tapes and disk packs and making sure that the correct one was loaded on to the correct drive at the correct time

during a batch run was crucial. The computer operators would be prompted on the console by the OS or possibly directly from a program using the COBOL DISPLAY verb and this activity was constant because of the limited capacities of tapes and disk packs.

The magnetic tape library

Large files might be spread over several disk packs and ‘scratch’ tapes holding temporary files were repeatedly overwritten and reused, processes that required careful management. The tape library usually occupied a lot of space and often had its own staff to manage it, sometimes even females.

It may now be hard to believe, in a world where everyone can have their own 1 Terabyte disk drive, but in those days a good disk pack consisting of several disk platters would perhaps hold 7 Megabytes, so there were many of them and many disk drives needed to host them.

A disk pack consisting of separate disk platters

The collection of disk drives in the computer room was sometimes referred to as the disk farm.

A disk farm

Batch runs frequently occurred overnight as the emphasis was on using precious computer time 24 hours a day so being a computer operator usually involved 24 hour shift work.

With all the activity going on in the computer room late at night it could be that unregulated, if not downright illegal, activities took place. An incident occurred one night when a computer room cleaner accidentally took the printer offline when it was printing cheques. Fortunately he had the presence of mind to put it back online but he then noticed that it printed a duplicate cheque. This potential source of illicit income was henceforth exploited for several months before being detected and the program error being corrected and the cleaner dealt with.

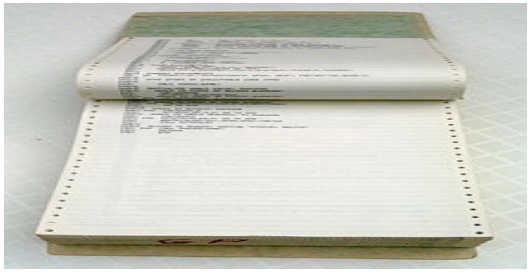
## *The Program Maintenance Nightmare*

Once a program went into production, especially in large organisations, it would often be handed over to the maintenance programmers. My second programming job, taken to boost my salary, was as a maintenance programmer for programs written in PL/1, a high level language created by IBM, but for the purposes of this account it can be considered as being equivalent to COBOL though there were some technical differences.

Our team's job was to take all the program errors that had occurred in the overnight batch production runs, details of which were on our desks when we arrived at work in the morning, and try to diagnose and correct them by the end of the day in time for the next night's batch runs. As the day progressed it could get quite tense and in those days, when it seemed that everybody including myself smoked, our team's designated area in the office was permanently covered in a fog of cigarette smoke and our team leader even smoked a pipe. I've never read Dante's Inferno but I'm sure similar environments must be described in it.

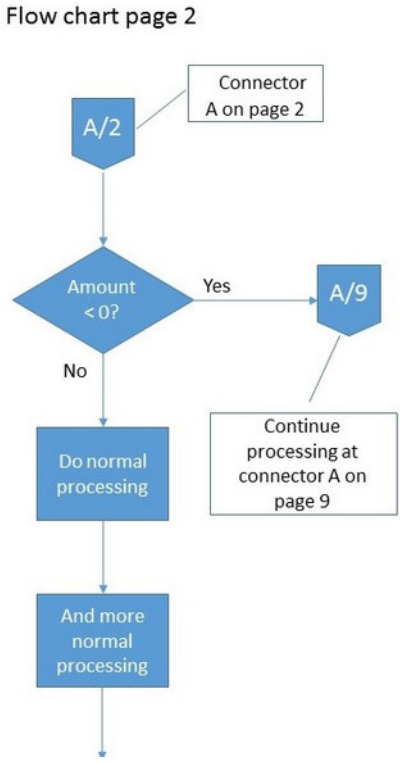
The compiler listing produced by the most recent compilation was the major, usually the only, resource the maintenance programmer had from which to identify the logic errors in a program. This meant it was absolutely critical that compilation listings were kept up to date. Since most programs at this time were written as a monolithic sequence of statements the compiler listing often occupied many many pages. Thirty pages or more was not uncommon. For large programs the listing could be very large indeed as in the example below. Using the listing could be a complete nightmare when it involved you flicking backwards and forwards through the pages looking for a paragraph name. This frequently occurred when tracing a logic sequence which included a statement such as

```
IF PRICE < 0 THEN GO TO PRICE-ADJUST.
```



A large program listing

The challenge now was to find the label PRICE-ADJUST, which could be anywhere in the many pages of compiler listing with the only way to locate it being by visual inspection. I think the widespread use and overuse of these annoying statements was a result of most programmers designing their programs using only the basic flow chart symbols which led to writing code in a certain way.




A typical flowchart fragment

These kind of flowchart sequences were produced because you wanted to flowchart the normal processing logic first and by the time you got around to designing the exception cases you were several pages further along on the flowchart. This would usually lead to the use of an IF A < 0 THEN GO TO PRICE_ADJUST construct instead of an IF A < 0 THEN …. ELSE …. construct and this in turn often led to adding yet more GO TO statements to keep the logic flowing correctly. GO TOs had a distinct tendency to proliferate. In this example the maintenance programmer needed to find the location of paragraph name PRICE-ADJUST in the program listing but there was no way of finding it except by flicking through page after page of the listing.

This led to one of the many religious wars that sprang up around COBOL, the 'meaningful' versus the 'alphanumeric' paragraph name dispute. The dispute arose over the question of whether paragraph names should be ‘meaningful’ like PRICE-ADJUST, or ‘alphanumeric’ like A-120 and created in alphanumeric order so they would be easy to find. The most ludicrous supporting arguments were offered on both sides, which is usually the case in these religious wars, personally I settled for the hybrid approach and gave paragraphs names like A-120-PRICE-ADJUST. With more and more commercial programs going into production program maintenance nightmares like this got worse and worse. In 1968 Edsger Dijkstra had published a groundbreaking letter in the American Computer Society’s technical journal entitled “Go To Statement Considered Harmful” in which he recommended the outlawing of GO TO statements and the use of more ‘structured’ programming logic.

Predictably it took several years for Dijkstra’s ideas to penetrate into the world of commercial programming but by the time I was maintenance programming in the early 1970s they were beginning to appear and were being referred to as structured programming.

## *Structured Programming*

We early commercial programmers were certainly pioneers and had few, if any, professional guidelines to follow. It was from the mistakes we made that those with more time to think about programming, like Dijkstra, slowly came up with the methods and

techniques that now constitute good programming practice. In the early days you had to keep your eye on the computer journals, which in the UK meant Computer Weekly and Computing, to become aware of any new ideas. You might then buy a book and teach yourself about them, or just hope to pick them up from other programmers.

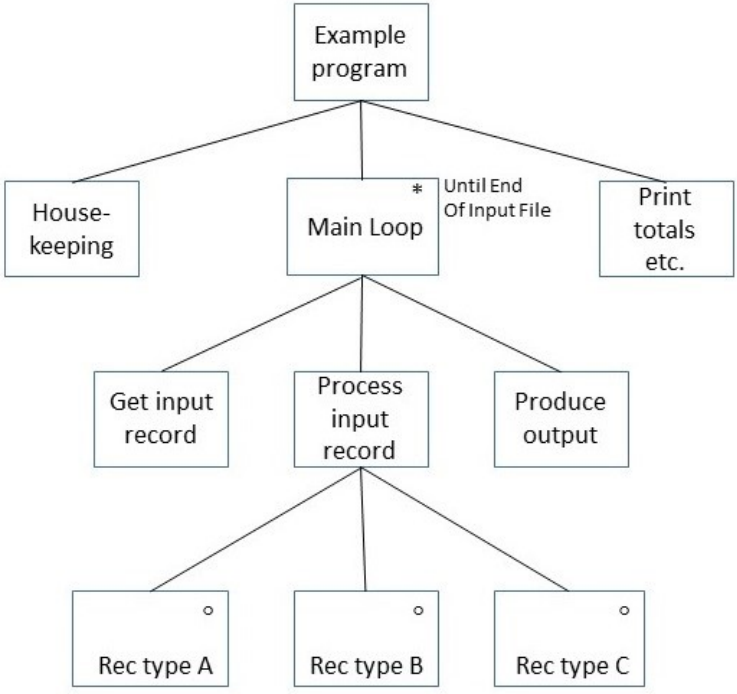
Structured programming started to get more and more mentions in the computer papers and I eventually read some articles and books about it. Its main impact, for me at least, was the replacement of flow chart symbols with a new set of symbols which did not encourage GO TO statements. A number of authors published books on the new programming technique and the two I became familiar with were Michael Jackson (no, not ***that*** one), whose technique became known as JSP or Jackson Structured Programming and Alan Cohen author of “Structure, Logic And Program Design” who I would work with some years later. Structured design diagrams consisted of a tree structured hierarchy of rectangles and were, unlike a flow chart, read from left to right and top to bottom and so broke away from the relentless top to bottom style of a flow chart. Initially it was said that a structured program could be designed using only three logical constructs.

**Sequence** - a block of code statements that were sequentially executed were represented by a simple rectangle.

**Repetition (Loop)** – a sequence of code statements that were executed repetitively until some exit condition occurred were represented by a rectangle with an * in the upper right-hand corner. In COBOL this would be by manually coding a loop using a GO TO. In other programming languages numerous other looping verbs were introduced such as FOR and WHILE.

**Conditional** – a number of alternative code sequences only one of which would be executed depending on the result of some condition were represented by a series of rectangles with a o in the upper right corner. There could be any number of alternatives which could be written using the COBOL verb GO TO …… DEPENDING or nested IF … THEN …. ELSE IF … statements. In other languages additional verbs such as CASE and SWITCH were introduced to handle conditional statements.

A structured program design diagram was read from left to right and top to bottom instead of in the relentless top to bottom style of a flow chart. The example program flow chart used previously would be redrawn in a structured fashion like this.



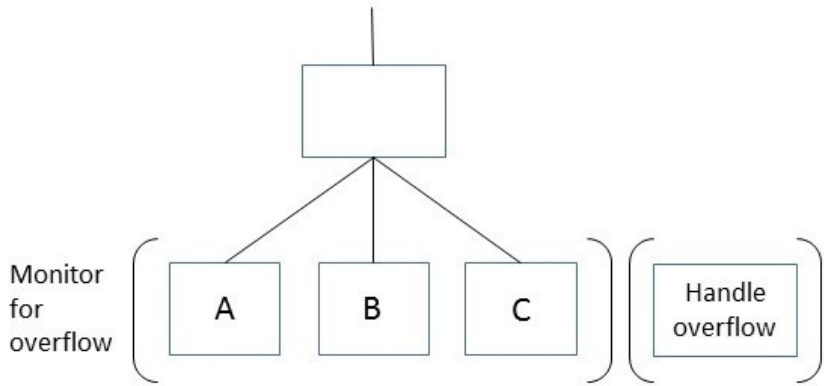

Structure Diagram

It quickly became apparent that a fourth logical construct was required to account for a class of exception conditions that could occur unpredictably at different positions within a program.

An example might be trying to calculate a number that was too large for the underlying hardware to handle and caused a hardware error known as an overflow condition.

This fourth logical construct was called Posit/Admit, by Jackson, or Monitor/Escape by Cohen, and they used different diagramming symbols to represent it. Alan Cohen's was similar to the following example in which a series of program sequences are monitored for arithmetic overflow and if it occurs anywhere program control passes to the error handling routine, possibly by using the dreaded GO TO verb.

As far as I am aware this logical construct was not formalized in any subsequent programming language until the TRY/CATCH construct in Java.




Monitor and escape notation for exception handling

These four constructs have remained remarkably resilient and I'm not sure any better way of representing program logic has been developed so far.

## *Software Engineers Arrive On The Scene*

Because it was an almost industrial production line process the development and operation of computer programs could be viewed as somewhat similar to the work carried out by industrial workers as described by Karl Marx. His ideas were quite current at the time and it was the time when Thunderclap Newman was singing that there was “Something In The Air”. A programmer’s revolution perhaps?

Central to Marx’s economic analysis was that the 'means of production' were owned by the (capitalist) employers, not by the (proletarian) workers, so the workers were completely dependent on the capitalists for the means to do their jobs. A car production line, for instance, was simply too expensive for workers to own, only capitalists with large amounts of capital could own them. This analysis also held true for computer programmers since our means of production, the mainframe computers, keypunch units, line printers, etc. were all owned by our employers and were completely out of reach of programmers themselves. In any computer project the hardware was by far the most expensive item in the budget whereas software was relatively speaking a cheap item. The idea that a programmer could possess their own means of production and produce their own software independently was simply unthinkable. The first commercial computer programmers were firmly positioned in the early digital age’s proletariat.

Marx had also described a strict division of labour amongst workers and we have seen there was a similar division of labour amongst different computer specialists and amongst computer programmers. In an effort to reposition themselves within the division of labour amongst programmers some of them, myself included, started to describe themselves as software engineers. The implication being that we coded in a highly engineered fashion using the latest structured programming techniques and were thus part of a new emerging professional programming elite.

## *Online Developers*

At the same time another group within the programming proletariat was claiming elite status, they were online developers. While I was busy maintenance programming a team of online

programmers were working for the same employee developing early online applications. They got to use, and to create applications for, computer terminals, the legendary IBM 3270 range of terminals, and to develop programs for the early online environments which would eventually evolve into IBM CICS. (#UI) The 3270s were expensive pieces of hardware and were connected directly to the mainframe using coaxial cable and they presented a simple 'green screen' forms based interface to online users. 3270s were 'intelligent' terminals so they could do a lot of pre-processing and validation of the input data being entered through the keyboard before it was transmitted to software in the mainframe.

A terminal from the 3270 range

(#PZ) The underlying model for the interface was that of entering data into an electronic version of a paper form then pressing the Send key to transmit the pre-validated data to the mainframe for further processing by application business logic specific to the form. (#UI) This form based programming model featured a frontend device presenting an electronic form as the user interface (UI) with a backend computer executing the specific business logic associated with the form. The backend software components are sometimes called transactions, though this word is often used very loosely, and more recently they've become known as services. This programming model is still the basis of many online applications. It was very commonly used in the first generation of web applications.

## *AI Stalls*

(#AI) Meanwhile, as I was slogging away in the maintenance programming trenches, the guys at MIT who were researching AI had been running into unexpected difficulties, creating AI was not as easy as they had thought. In 1972 an incendiary book called “What Computers Can't Do” was published by Hubert L. Dreyfuss which argued extremely persuasively that the whole project was basically smoke and mirrors and was doomed to failure. It caused a sensation in American academia but unremarkably it failed to register with our maintenance programming team as we labored away in our fog of cigarette smoke. It would be another 10 years before I finally came across the book and read it.

# 1975 London - A Whole New Ballgame

By 1975 I was a veteran of three programming jobs in a fairly short period of time and I was concerned that my CV (US=resume) was beginning to scream ‘job hopper’. But professionally and personally I was anxious to keep moving and to learn the new technologies which were rapidly emerging. In particular I wanted to get off mainframe computers and on to mini-computers, or minis as they were commonly referred to. Not only were they the latest technology but their very name carried a trendy, even sexy, hint of two major UK cultural icons of the 1960s, the Mini car and the mini-skirt or mini-dress.


*A Mini and Twiggy in a mini-dress*

The solution I came up with was to join an IT consultancy, which in those days were known as ‘software houses’ in the UK. By doing so I hoped I would be able to experience many different computer environments while remaining at the same employer. London offered by far the most options so I travelled back there from Brighton and joined a small, and long since disappeared, software house called CMES.

## *Minis - The PDP11/70 Is Born*

In the 1970s Digital Equipment Corporation (DEC) in America started a computing revolution with their PDP11/70 series of minicomputers which by comparison to traditional mainframe computers were cheap and small. This class of computers would later come to be given the generic title of servers in order to distinguish them from mainframes. DEC were based close to the Massachusetts Institute of Technology (MIT) on Boston’s route 128 which had the same iconic status in the computing world that would one day be occupied by Silicon Valley.


A PDP11/70 front panel

DEC had produced earlier versions of the PDP but it was really the power of the PDP11/70 that brought PDP computers widespread acceptance in the commercial business world.

For the first time it started to make commercial sense for small and medium sized businesses to have their own computers. You could now have a fully functional mini-computing environment without a giant air-conditioned computer room packed with bulky expensive equipment and a large operations staff; without an extensive punched card production and management operation; with a separate computer devoted to program development and with the whole environment involving a less than astronomical budget. This was also of interest to software houses as it allowed them to go beyond providing temporary staff to their customers and to develop their own software in-house. Computing was beginning to get beyond the control of the large computer manufacturers and large computer users. Even so the extent of these changes was seen as being strictly limited to businesses and in 1977 Ken Olson, founder of Digital Equipment Corporation, famously said “There is no reason anyone would want a computer in their home”. But this at least was an advance on Thomas Watson of IBM’s earlier estimation of the size of the computer market being only five computers.

It was even rumored in the London programming circles in which I moved that there were some very successful freelance programmers who had actually bought their own PDPs and could create their own software. This was the Holy Grail for programmers, to finally escape from the trap described by Karl Marx, to own the means of production and to be able to create their own software independently. The programmer’s revolution was beginning to happen.

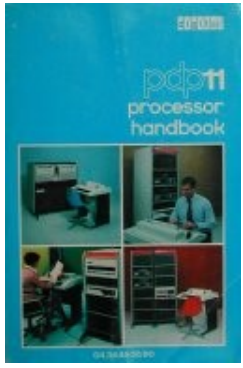
These developments meant that hardware costs as a proportion of an IT project’s budget started to fall relative to the software costs. Previously it was hardware that devoured a huge percentage of any IT project’s budget but now the balance was starting to shift towards software and would continue to do so.

For the next two years, while I worked for CMES on PDP11 projects, I was exposed to so many new programming practices and technology it is impossible to arrange them in a clear timeline as everything was happening to me simultaneously so I present them below in a more or less logical order. The overall effect of all these changes was to begin breaking down the Monolithic Sequential

Programming Style and to start replacing it with an early version of what could be called the Monolithic Modular Programming Style.

## *Train Yourself*

At CMES, programming on a PDP11/70 involved using the low level MACRO-11 assembler language which in turn meant having a decent understanding of the actual PDP11/70 hardware architecture. DEC provided a few manuals the size of paperback books from which we at CMES were supposed to train ourselves, often during billable time at a client site.


Apart from doing the reading you could ask more experienced programmers you were working with to help you out and hope they were willing to share information with you, though not all of them were.

It was very much a sink or swim experience.

## *Dumb Terminals*

(#UI) Something that was crucial in reducing the cost of mini-computer environments and helped popularize them were ‘dumb’ ASCII terminals with which to create relatively cheap online systems. It was their dumbness that made them so much cheaper than ‘intelligent’ IBM 3270 terminals. Unlike the 3270s dumb ASCII terminals did no pre-processing of the keyboard input and did not require expensive coaxial cabling to connect them to the PDP server. Each input keystroke was immediately transmitted to the

PDP and was processed by software running on the PDP computer itself.

A VT100 terminal

Very quickly DEC's VT100 terminal established itself as the dominant dumb terminal and as the de facto standard.

## *Do It Yourself Programming*

My first experience of PDP11/70s was on a project at N. M. Rothchilds merchant bank where to my astonishment there were no key punch girls and no punched cards. We were expected to enter our own source code through dumb terminals using a simple line editor program and it was stored directly into a source code file on disk. For those who can remember MS-DOS, think edlin.

Never having had a typing lesson in my life, like most of my colleagues, this tended to be a rather hit and miss affair at first, but even so it was still quicker and more flexible than relying on key punch girls.

## *RSX11 – A Multi-User Operating System*

DEC's multi-user RSX11 operating system underlay the new programming environment and it too was revolutionary. It supported large numbers of users simultaneously through a command line interface that was presented on their dumb terminals. Thus many

programmers simultaneously could create and edit their own source code files, run compilations, run unit tests and use the other utility programs that came with RSX11. The programming teams I worked with at the time would normally consist of at least a dozen programmers and were easily supported. This was a massive liberation from the restrictions of the coding sheet and punched card world of the mainframe environments I was familiar with and it allowed for much quicker program development. Due to the low cost of PDPs an individual machine was usually dedicated to program development while production work was carried out in parallel on a separate machine so there was no conflict between development and production.

The file system and utility programs supported by RSX11 were roughly comparable with, though better than, MS-DOS as implemented on early PCs, though MS-DOS of course only supported one user. RSX11 provided basic file handling features (edit, create, print, delete etc.) as well as development tools such as compilers. This was a massive step forward for programmers and the programming task.

Some RSX/11 features were well ahead of their time and as far as I know have never been duplicated. For instance files were specified right down to their version number, e.g. myprogram.mac;3 would reference the 3rd version of the myprogram.mac file. If no version number was specified when referencing a file RSX11 would automatically access the one with the highest version number. It also automatically incremented the version number each time the file was saved and the previous version was retained. After editing the above program the saved file would be myprogram.mac;4, but versions 3, 2 and 1 would still be there. This was a real boon for programmers who were mostly untrained typists now entering their own program source code and making frequent typing errors. To prevent the file system filling up with old versions of files an RSX11 parameter allowed you to specify how many previous versions to retain. Amazingly, to me at least, this feature which was present in the early 1970s in RSX11 has never been adopted by subsequent OSs such as Unix or Windows. In those environments overwriting files that you later needed to recover was to become a common problem.

## *Subroutine Components For Programs*

(#PZ) MACRO-11 and similar languages supported the creation of reusable program components commonly known as subroutines that could potentially be used **in many different programs** and this was the beginning of a whole new way of writing programs. Consider this simple example in a pseudo programming language:

EXECUTE some instructions

CALL the CALC-AREA subroutine passing values WIDTH=5 and LENGTH=3 and receiving RESULT=15

Carry on EXECUTING more instructions

The power of a subroutine is that the program code it consists of only needs to be written once but can be called from anywhere in the main part of a program and control will automatically return to the correct place when it completes. Parameter values to pass to the subroutine are specified at the point it is called. The instructions making up a subroutine, in this case CALC-AREA, would be written outside the main program code, possibly even in a completely separate source code file. This was far more flexible than the COBOL PERFORM statement and it was safer too as subroutines did not manipulate data items in a common, or global, data area like COBOL's WORKING STORAGE section. Actually it's more accurate to say subroutines were not supposed to access global data because it was still possible to do so and in fact sometimes they did. But it was a programming practice as much disapproved of as the use of GO TO statements in COBOL. If a subroutine did access global data items then that tied it to that specific program and it could not be reused by another program.

(#PZ) But general purpose subroutines that did not access global data areas could be used in many different programs, unlike a PERFORMed COBOL paragraph, and so libraries containing the object code files of many subroutines became supported by the major operating systems. This led to an additional process in program building called the link phase which was often carried out

by a separate program that ran after the compiler, the link editor. Once your own code had been compiled then the link editor would make copies of any common subroutines you used from the subroutine libraries and combine them together with your code to create the final executable program of object code. The end result was still a single monolithic program but one in which parts of it may have been developed by other programmers.

## *The Basics Of PDP Computer Architecture*

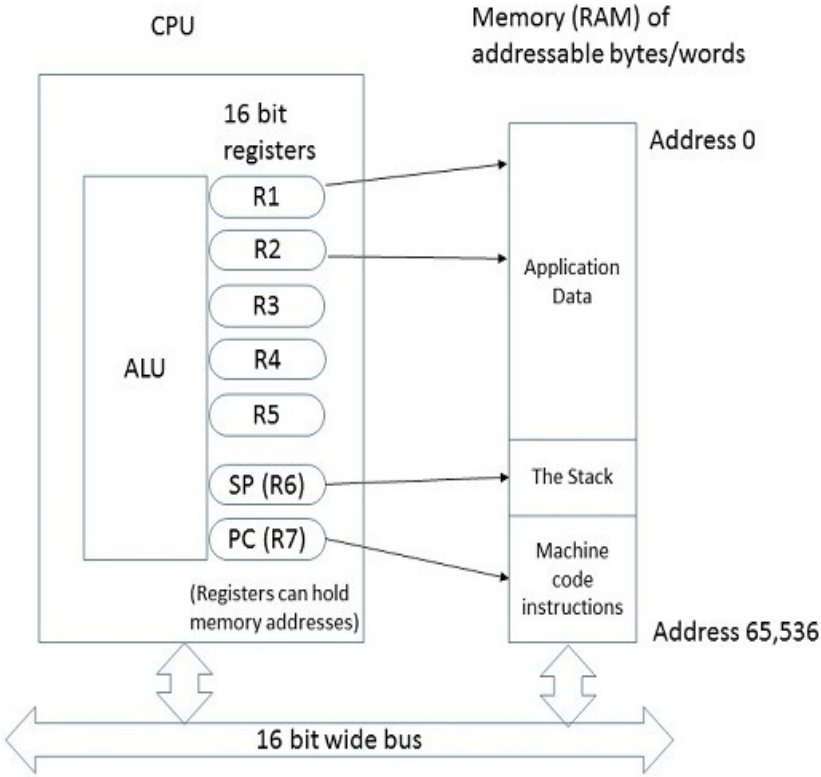
I cannot remember whether any high level languages, like COBOL, were available on PDPs at that time but anyway we did not use them, we used the PDP assembler language MACRO-11. Learning how to write programs in MACRO-11 and getting to understand the underlying hardware architecture of the PDP11 range of computers were two totally interdependent activities. Without understanding the PDP11 hardware architecture the MACRO-11 instructions simply made no sense.

MACRO-11 was not a 'high level' language but a 'low level' assembler language consisting of mnemonic names representing the PDP's actual machine code instructions. Assembler language mnemonics were a one-to-one mapping with the actual machine code and the mnemonic names e.g. ADD, MOV, JMP, etc. made some effort to indicate what the machine code instructions actually did. To keep this current explanation to a manageable size only two or three of them will be used. The translation process from assembler language to machine code was carried out by the assembler program which was similar to a high level language's compiler but faced a far less complex translation task.

The following description of the PDP architecture keeps to the very basics but further details are easily available and some further references are included in Appendix D.

This common basic hardware architecture is essentially the design of the so-called Von Neumann machine, named after John Von Neumann the pioneering US computer designer. The PDP11 hardware architecture was essentially the same as any other computer's architecture and only differed in some details so painstakingly gaining this knowledge was very valuable to programmers.

To execute a program many conditions need to have been met beforehand. It needs to have been assembled down to object code, the machine code needs to have been extracted from the object code file and loaded into memory by RSX11 and any terminals, printers etc. all need to be online and ready. But we will ignore all that and assume it has already happened and that the program's machine code is in memory and ready to go.




The basic PDP11 architecture

There are three core building blocks of the PDP11 computer to be considered.

- The **CPU** (central processing unit) is the brains of the computer and executes the machine code instructions. Conceptually we can divide it into two parts.

- (1) The **ALU** (arithmetic and logic unit) is complex electric circuitry that is driven by the machine code instructions to electrically manipulate the registers and memory. It consists of specialized electrical components such as logic gates. In the Analytical Engine Babbage had designed these as mechanical components.

- (2) The set of seven **registers** are specialized 16 bit components that hold the operational binary data values being used and manipulated by the machine code instructions. These are what Charles Babbage called 'variables' in the Analytical Engine.

- The random-access **memory (or RAM)** is a sequential collection of bytes (8 bits) of memory each capable of holding a binary value up to decimal 255, possibly an ASCII character, and each having a unique address. This is what Charles Babbage called the 'store' in the Analytical Engine.

- The **bus** (which in the PDP was known as the Unibus) connects the CPU to the RAM (and to other devices such as disk drives, communication ports, etc. which we are ignoring here). The bus was 16 bits 'wide' which meant that data could be transmitted along it 16 bits at a time in parallel i.e. all at the same time, with a maximum value therefore of 65,536.

The width of the bus is directly related to how much RAM the CPU can handle and how many bits the registers need to be able to hold. A common use of a register is to hold the address of a single byte of RAM and since the registers consist of 16 bits the maximum address value they can hold is 65,536, or a binary number consisting of 16 1s (hexadecimal FFFF), this value is usually referred to as 64K (one K being $2^{10}$, which is 1024). This determines the maximum amount of memory that can be addressed by the CPU, though software techniques were developed later that allowed this limit to be exceeded. Eventually in later computers bus and

register sizes were increased to 32 bits and now the transition is on to 64 bits. It was originally envisaged that these transitions would happen much faster but another truth about advances in computer technology has proved to be that once systems are in place and working the tendency is to leave well enough alone and not to do unnecessary upgrades.

The other main use of registers is to hold binary numbers that take part in calculations. Whether a binary value in a register is to be treated as an address or a number, i.e. what variable 'type' it is, is determined when specifying the specific machine code instruction the CPU is executing

When a program is loaded into RAM and is being executed the memory it occupies can be divided into three areas.

- The **application data** area is where the application's data values are held and manipulated, roughly equivalent to the Data Division in a COBOL program. Much of the data will have been loaded from disk files and may be written back to them later.

- The **stack** area, this is a special memory area used by RSX11 when a program calls a subroutine and it holds any parameters being passed to the subroutine plus the program address to return to when the subroutine concludes. The use of the stack area is controlled via a special register the Stack Pointer (SP) which holds the address of the next available byte in the stack.

- The **machine code** area contains the instructions for the CPU to execute, i.e. the program itself. Unlike the other two memory areas the contents of this area should not change during program execution. A special register, the Program Counter (PC) always addresses the next machine code instruction for the CPU to execute.

Like everything else in RAM the MACRO-11 machine code instructions are binary values which usually have the format:

Operation code (16 bits); Operand 1 (16 bits); Operand 2 (16 bits)

For example to add the binary number in register 1 to the binary number in register 2, the assembler source code would be:

ADD R1, R2

The MACRO-11 assembler program would translate the mnemonic ADD to the binary value 6000 (hex) which is the ADD operation code (or **op code** as they are usually referred to), plus binary identities for the contents of R1 and R2, the two operands.

However, suppose you wanted the binary values in R1 and R2 to be treated as addresses so you could add the binary value stored in RAM at the first address to the binary value stored in RAM at the second address. The assembler instruction was then written:

ADD (R1), (R2)

Clearly the results of these two operations would be completely different and if an assembler instruction was miscoded so that a binary number was treated as a memory address then unplanned changes would occur at more or less random RAM addresses and data values could be corrupted or, much worse, machine code instructions could be corrupted with catastrophic results. Later ‘strongly typed’ programming languages would be developed where the type checking of variables would take place at compile time to eliminate this kind of error.

More details of the MACRO-11 language are available for those who are interested but one further point is worth highlighting here about working with text strings. Working with a string of text data involved processing it one byte at a time in a loop which each time it executed incremented the address pointer to process the next byte in the string being processed. The loop would, or should, be exited after the last byte of the text string had been processed. However, when, for instance, you were moving a text string from one memory location to another and you failed to exit the loop at the correct point (i.e. once you had moved the last character of the stored text string) the old ‘Garbage In, Garbage Out’ rule applied. You would start to move unknown byte values to unplanned target addresses thus overwriting other data values or even machine code

instructions. Needless to say this would cause catastrophic errors and even today forcing address pointer overruns is still a weapon sometimes used in virus attacks.

Although the computer architecture described above is essentially the same basic architecture developed by Charles Babbage, Alan Turing, Tommy Flowers, John Von Neumann and others it has remained virtually unchanged and yet it has achieved so much. It, or something very like it, is often referred to as the Von Neumann architecture. At its heart are a number of special purpose electronic components that use electronic circuitry to dumbly execute a set of machine code instructions expressed in binary. But on top of that limited foundation generations of programmers using ever more advanced programming tools and methods have built the complex layers of software systems we are surrounded by today.

## *Message Switch Infrastructure Software*

After my assignment at Rothchilds helping develop financial software CMES sent me to Cable & Wireless in 1976 where for the first time I became fully aware that computer software was being developed for far more than sophisticated accounting systems. Software was being created that did some very useful practical things that had been impossible in the past. The project at Cable & Wireless was to implement a store and forward message switch and a team had been put together consisting of programmers from a number of software houses augmented with some freelance programmers. Only a few team members had any experience of what we were doing as it was very early days for this kind of infrastructure software.

In 1976 to connect two devices together, usually 2 telephones though sometimes teletypes, a dedicated electrical circuit between the two devices had to be established and maintained for the length of the conversation. Telephone exchanges were complicated electromechanical switching mechanisms for establishing a single continuous electrical connection between the caller and the call receiver. This meant that if you tried to make a telephone call and got the busy signal that was that, no circuit could be established and you would have to try again later. Since telephone switches were electromechanical hardware they could not store voice messages, in fact they could not store anything.

But the existence of mini computers created the possibility of store and forward text message switches that were computer based and controlled by software. With a store and forward message switch text messages could be sent from one teletype to another via a computer based message switch which could store the message in a disk file during its transmission. If no circuit was available to connect the message switch to the receiving teletype the message could be sent later by the message switch software when a circuit became available. Clearly many many networking and messaging developments subsequently flowed from this basic but revolutionary idea. To name but a few there were packet switched networks, connectionless networks, modern email with all its complexities, transactionally guaranteed message delivery (aka fire and forget), text messaging and much more.

We had a special test rig set up in the computer room and as one of my test tasks I remember the thrill of seeing a message I entered into one teletype appear on another one I switched on a few minutes later. It had actually been stored and forwarded.

## *The Monolithic Component Programming Style*

The developments taking place were breaking down the previous Monolithic Sequential Programming Style and ushering in what could be called the Monolithic Component Programming Style. This would continually evolve over the next decades but this programming style saw the emergence of the early stages of a number of key programming features. These would transform the life of the commercial computer programmer, and indeed the lives of most people in the world, over the coming years.

### *User Interface (front end) versus Business Logic (back end)*

(#UI) Although the UI (user interface) for many years would be little more than an electronic form presented on a dumb terminal it had now been logically separated from the business logic which had previously formed the entire program. There were now real end users who were business employees and who were interacting with computer software and designing that interaction became a new design task. Later the separation of the user interface from the business logic meant the user interface could be located on a PC

which would lead to various competing architectural models and specialised programming roles over the years.

### *Modules and Libraries*

(#PZ) Program source code could now be broken up into separate component files referred to as modules which could be developed by different programmers, compiled separately, and later linked together. One module might contain the program's main routine and the others contain one or more program specific subroutines only relevant to this particular program. After the assembler had run then the linker (or link editor) would be directed to link all the modules together to form the executable program. More widely useful subroutines would be held in object libraries and the linker would search the libraries if it detected any unresolved subroutine calls in a program's modules.

These developments meant it became crucial that the data interfaces between modules or subroutines were clearly defined or else runtime errors would occur when invalid parameter values were passed across the interface, so-called data typing problems. Structured programming techniques did not really support detailed interface design and so new techniques emerged that will be covered later, in particular data flow diagrams (DFDs).

### *Maintenance Dependencies*

As programs could now consist of a combination of different object modules linked together the issue of dependencies between them became critical. It was vital that the latest version of every program component was used in creating a production program. Code management tools would be developed to address these issues.

### *Multi-user data access*

As multiple online users could be accessing the same data files simultaneously all the associated issues, in particular that one user could overwrite another user's update, began to appear and require solutions. ACID transactions, TP Monitors and shared relational databases would emerge to handle these problems.

### *Advanced debugging techniques*

Because of the new flexible and more rapid program development techniques it was easy to temporarily insert additional debugging statements into a program to help diagnose errors. At the same time other debugging techniques became available that led to the development of ever more powerful online debugging tools.

# 1977 - The Microprocessor Revolution

When I first joined the software house CMES I had hoped I would experience a lot of new programming environments on different projects but I soon realized that things didn't work like that in the consulting world. Experience is everything and I was now an experienced PDP11 programmer so that would be how I was deployed. But at Cable and Wireless I worked alongside some freelance programmers and realized they were earning far more than me and had more choice as to what they did. Something else making me want to move on was that microprocessors were now featuring fairly regularly in the computer press and I wanted to get experience with them. If that wasn't enough reason then the background music to all my thinking was punk rockers Johnny Rotten and the Sex Pistols who were celebrating Queen Elizabeth II's 25th anniversary with their stirring rendition of "God Save The Queen". Things were happening, it was definitely time for a change.

The first complete Von Neumann machine implemented on a single silicon chip, that is a genuine microprocessor, was produced by Intel in 1971, the Intel 4004. But it was not until 1976-77 that microprocessors started to get frequent mentions in Computer Weekly and Computing and by then the Intel 8080 was getting all the attention. In 1977 after months of scanning the job ads in Computer Weekly I finally saw one for freelance programmers that included the word 'microprocessor' and I promptly submitted my CV, by snail mail of course. Subsequently I received an invitation to an interview with Micro Focus, a company I had never heard of.

## *Those Crazy Guys At Micro Focus*

Although I was somewhat aware of microprocessors I was not really aware of the related business developments taking place in America. In 1976 Steve Jobs and his associates started Apple Computer, producing the first fifty Apple 1s in his parent's spare bedroom and by 1977 they were successful enough to shift production from the spare bedroom to the garage. At the time Bill Gates was setting up Microsoft's first HQ in Alberquerque and

producing software for the Altair hobby computer. But this new way of doing business was totally unknown to me and pretty much unheard of in the UK.

Because of this I was astonished to turn up for my interview at Micro Focus and discover the ‘office’ was in fact somebody’s house in South London in a normal residential street. A very affable Paul O’Grady (POG as I would come to know him from his memo signature) led me into his front room and proceeded to tell me all about Micro Focus as a company. He also told me what they were trying to do and to a journeyman consultant programmer like myself it all sounded incredible. There were 3 partners in Micro Focus who had struck a deal with ICL to write a COBOL compiler to be used on a Point Of Sale device I’d never heard of, the ICL 1500, which had only 8K bytes of memory and used a microprocessor chip that was unknown to me.

An ICL 1500 Point of Sale device

This latter was not too surprising as the only chip maker I’d heard of at that point was Intel but even in those days 8K bytes was a pretty limited amount of memory. Even more astonishingly Micro Focus intended to do some of the development on an Intel 8080 based development machine, the Intel MDS (Microprocessor Development System). The MDS was normally used for developing Intel firmware and therefore focused on producing machine code for the Intel 8080, not for the ICL 1500. One of POG’s partners, Brian Reynolds had the MDS over at his house just off Notting Hill’s trendy Portobello Road, not even in a proper office.

I struggled to take all this on board. They were going to develop software on their own machine and just in case that wasn't enough they also intended to write the COBOL compiler in COBOL. I could immediately see a significant problem in trying to do that. It seemed rather like the Indian rope trick. How would you do the very first compilation? How could a compiler written in COBOL compile itself by being passed through an uncompiled version of itself?


The Intel Microprocessor Development System (MDS)

I was astonished, impressed and confused. Which planet did POG come from? Was he dreaming? Was he crazy? Was I dreaming? Was I crazy? But if I wanted to get involved with microprocessors, which I did, this seemed to be the only game in town, so what could I do? POG said I could go over to Brian's house one half day a week and familiar myself with the MDS machine, unpaid of course. So I started going over there once a week to get familiar with the MDS and its ISIS operating system while still keeping my day job with CMES at Cable and Wireless.

If I thought MACRO-11 was a low-level programming language it was nothing compared to Intel 8080 Assembler language. On the Intel 8080 data was generally processed one byte at a time and had to be loaded from memory into a register, processed in the register, and then stored back into memory. This led to many many lines of assembler code and the ISIS operating system was also very basic compared to RSX11. I learnt all this the hard way when I lost a three hour editing session and discovered there was no way of recovering it, so I had to enter it all again. ISIS was a very minimal OS with few utilities and, for example, I would later have to write my own file dump program as there was no such utility included with ISIS.

- It helped that I was an experienced COBOL programmer and when I met the compiler designer and third partner at Micro Focus, Stuart Lang, it turned out that because of that, and my PDP11 experience with dumb terminals, (#UI) I could provide some input to thinking through the implementation of the screen handling. Stuart had done postgraduate work at Cambridge University that included modern techniques for compiler development and had the complete design worked out in his head. (#UI) But at that time COBOL had no way of providing a forms type UI which was a required extension to CIS COBOL so that programs running on the ICL 1500 could use its small display. This extension was eventually implemented by extending the ACCEPT and DISPLAY verbs and making a special use of COBOL's FILLER data items so that input forms could be handled and formatted.

Micro Focus was the first example of a phenomenon I would encounter several times again during my career, a successful start-up company involving three people filling three key roles. I don't recall ever encountering a successful startup without them. The three key roles were, and probably still are, in no particular order:

- a visionary with a vision of what was to be achieved
- a technologist who understands the vision and knows how to implement it
- a business person/diplomat to handle all the practical matters needed by the other two

The three partners at Micro Focus filled all three roles, sometimes in an overlapping manner.

## *The CIS COBOL project*

Along with a crew of three or four other contract programmers who would write all the ICL 1500 assembler code, Brian, POG, Stuart and I all started work in what was rather euphemistically called an industrial facility in west London's distinctly untrendy Acton. We had two very basic rooms there and I imagine Steve Jobs and his team were more comfortable in his

parent's garage than we were, but it sufficed. The team was completed by Barry Oakley who was another COBOL programmer moonlighting from his day job in Southend, so I had to be careful if I needed to call him during the day.

This project turned out to be the first time I was involved with data being transmitted over a phone line, it was one of the many technical innovations I was being introduced to. In order to do this a modem had to be used at both ends to convert the binary values from the computer into audible tones that could then be transmitted and received over a normal dialed telephone connection. But how to connect the computer's modem to the phone line? In those days in the UK phones were wired directly into the wall and any messing about with those wires was likely to get you into trouble. There was no handy jack plug connector to attach the modem to, instead an acoustic coupler had to be used that was attached to the hand unit of the phone.

An acoustic coupler

When all the connections were good this system worked satisfactorily, though slowly, but the errors were not uncommon and having to rerun a transmission was not unusual.

Stuart's design for the CIS (Compact Interactive Standard) COBOL compiler used all the latest ideas including it being written in COBOL itself, a decision which would soon dominate my waking hours. When doing its job of compiling a COBOL program it translated the program's COBOL statements into an artificial 'intermediate code', not machine code. The intermediate code had been designed by Brian to be extremely compact and common

operations were represented by single byte op codes. The intermediate code could then either be translated into the target computer's machine code or an interpreter could be written for the target machine to execute the intermediate code instructions at runtime. This latter approach was adopted for CIS COBOL and was famously adopted years later by the Java programming language with its use of JVMs (Java Virtual Machines). At the time there were many different CPU architectures with each architecture requiring its own compilers to produce the machine code instructions it supported. Because of this situation producing compiler software that could easily be moved to different architectures, 'ported' being the term for that, was very big news and offered a huge gain in efficiency. A partial list of the different CPU architectures around in those days included Intel 8080, Intel 8086, IBM, ICL, Honeywell, HP, and many others.

As the project got under way several tasks were carried out in parallel. The ICL 1500 assembler team set off writing the interpreter to execute the intermediate code that the compiler would translate COBOL source code into. Meanwhile Barry was writing, in COBOL on a regular mainframe, the first stage of the compiler, the lexical scanner and syntax checker. At the same time I was using the Stage 2 macro processor, a piece of software from a US university that Stuart had got hold of on a paper tape, to write the compiler-compiler. The compiler-compiler was the temporary software required to translate the COBOL compiler which was written in COBOL into intermediate code for the first time, after which it could compile itself. I loaded the Stage 2 source code through the paper tape reader of a teletype, got it running on the MDS and started to write macros to convert the compiler's COBOL source code into a loadable ICL 1500 file of intermediate code.

It quickly became apparent that it wasn't going to be achievable in a single pass against the COBOL source file so I fell back on batch processing and eventually constructed thirteen different Stage 2 macro files. When Stage 2 ran one of the macro files it took a single input file and transformed it to a single output file which was then used as input when Stage 2 ran the next macro file in the batch, the good old 'Data In Data Out' approach. When Stage 2 ran the first macro file it took an input file of COBOL source code and when much later it ran the thirteenth macro file it produced an ICL 1500 file of intermediate code as output. As I was doing this

I was constantly coming up against limitations on what could be achieved in terms of COBOL functionality and informing Barry of them so he could tailor his COBOL code accordingly. Meanwhile in the next room I could hear ‘bad cop’ Brian exhorting the ICL 1500 assembler team to cram their code into ever smaller memory spaces.

Finally the compiler-compiler was finished and I started writing the COBOL compiler’s final stage the intermediate code generator in COBOL from Stuart’s design specification which made heavy use of table driven code to keep the size of the compiler as small as possible. This, added to the work Barry had done on the lexical scanner and syntax analyser, would complete the COBOL compiler. The initial idea that the CIS COBOL compiler plus the interpreter could be squeezed into 8K bytes proved to be out of the question but by organizing the compiler as an in-memory section with three overlays that ran one after the other it was made possible.

After much effort by everybody we had finally done it, we’d created a COBOL compiler for the ICL 1500 that ran in 8K bytes.

## *Moving To The Intel 8080*

With the CIS COBOL compiler itself now compiled into intermediate code by the compiler-compiler it could be rehosted on to the Intel 8080 by writing an interpreter in Intel 8080 assembler. This was my next task which was made much easier since I could use the existing ICL 1500 interpreter as the specification and the team who had produced that version had already solved most of the design problems.

When the compiler was up and running on the Intel 8080 Peter Brown, who was the Professor of Computing at Kent University, wrote a full page article in Computing on the work we had done. The article is reproduced in full in Appendix B and is still a very readable description of what portable compilers are all about. It was published on May 11th 1978 and it gives a good feel for the different way in which software entrepreneurs and their programmers were viewed in those days. The Micro Focus directors didn’t even get name checked and I was merely the 'one man' referred to by Peter Brown. That was to be the extent of the

15 minutes of fame that had been promised to me, and everybody else in the world, by Andy Warhol in 1968.

## *The End Of My Microprocessor Road*

I had now gained the microprocessor experience I had been looking for but in the process I had become aware of the limited nature of the tools that were available for working with chips like the Intel 8080. It was really working a bit too close to the hardware for me. At that time the term 'hacker' became current in the programming fraternity in the UK but with a different meaning to now. It referred to a programmer who really got down to a low level of coding close to the hardware and I did not aspire to become a hacker.

I decided to head back into the mini world and I took a short term contract in a PDP environment. The money was better too.

# 1979 – Minicomputers Proliferate

Although the PDP11 series dominated the minicomputer world in the same way that the IBM 370 dominated the mainframe world nevertheless there were other options appearing in the minicomputer market. This was driven by the commercial availability of the basic building blocks with which to build computers such as integrated circuits and microprocessors. In the light of this availability many electronics companies decided that they should include minicomputers in their product lines such as GE, GEC (General Electric Company, UK), Packard Bell, Data General, AT&T, ICL, IBM and others. This meant there were many computer environments around that nobody had much experience of.

## *Software In The Signal Box*

In 1979 I was offered a contract on a project using a GEC 4080 mini with the software being written in CORAL 66, so another exercise in self education would be required. However the project was attractive for two reasons, first, the project was to be led by structured programming expert Alan Cohen and second the project was to implement a railway signaling system, and that sounded interesting.

GEC was the giant of the UK electronics companies and they had a joint venture with General Signal (US) called GEC-General Signal. The project was to create the signaling system for a railway line running from Sao Paulo to Rio de Janeiro in Brazil. Computers were now getting out of the office and message switching systems and deep into the real world, making what could be life and death decisions. Writing computer programs could now have major effects in the world we lived in, things were getting serious.

The signaling project involved working with experienced railway signaling engineers and for them the idea of using computer software to control railway signals was deeply concerning, if not horrifying. They viewed replacing electromechanical systems by digital software with alarm. The nightmare scenario for a signaling system failure involved the set of signals controlling access to a set of points where trains can be switched to another track. Trains

approaching the points from different directions must be tightly controlled by signals and any operational errors in the signaling system could cause head on collisions or derailments. The synchronisation of the setting of the points and the operation of the signals is crucial. Traditionally the engineers had used a sophisticated electromechanical system they called an interlock which made it mechanically impossible for the points and the signals to be out of sync. Even if there was a power failure there was a system of counterweights attached to the electrical relays to make sure they would all fall into safe positions due to gravity. Moving to an electronically based system worried the engineers very much indeed.

The use of Alan's structured programming techniques on the project proved to me their power, particularly in the reduction of logic errors and software bugs. Even though our six person team was a varied collection of contract programmers the structured approach meant our work fitted together easily and the bug count was very low. The time I had spent as a maintenance programmer had given me a distinct nervousness about the abilities of some of my fellow IT workers but there was no denying that Alan's logical and structured approach combined with his calm manner really worked. He and his methods could definitely be relied on and meeting Alan would prove to be a milestone in my career which would only become clear to me many years later.

Following this project there seemed to me to be little future in gaining any more experience on the GEC 4080 or in CORAL 66 for that matter, which turned out to be true. I decided to return to the DEC world where significant new developments were taking place.

## *The PDP Gives Birth To The VAX*

In 1978 DEC had released the first of their new VAX range of computers which were a major evolutionary, if not revolutionary, step forward from the PDP range. They were often referred to as super-minis as they were capable of heavy duty processing and were being positioned as a realistic alternative to mainframes. They led to the increased acceptance of non-mainframe computers by major business enterprises. A VAX had a word length of 32 bits and thus supported a far larger address space than the PDP, over 2 billion bytes, this meant programmers no longer needed to use

overlaying techniques when writing large programs. VAX machine code, represented mnemonically by the VAX MACRO language, contained many complex op codes that were as powerful as high level language verbs. For example VAX MACRO had CASE op codes that were essentially identical to COBOL's GOTO DEPENDING verb. The existence of these powerful op codes helped enormously when writing high level language compilers for the VAX as, for instance, GOTO DEPENDING translated to a single VAX op code rather than to the at least ten op codes required by CIS COBOL on the Intel 8080. As a result there were many high level language compilers available on the VAX. Computer chips that supported machine code instruction sets of this complexity were known by the new acronym CISC (Complex Instruction Set Computers/Computing). Like so many developments in computing the CISC chip architecture would eventually lead to another religious war, the CISC vs RISC war. We'll get to that later.

The operating system for the VAX range was the VAX/VMS operating system which in its day, and perhaps even to this day, was the Rolls Royce of operating systems and had many advanced features. DCL, the command language for VAX/VMS, was a very extensive English like language in which it was possible to write command procedures that were as functional as many a program written in a high level language. However the command procedures were not compiled but were interpreted at run time and so were relatively slow in execution. In later years when Microsoft finally decided to replace their MS-DOS based operating systems with the proper multi-tasking operating system Windows NT they poached the legendary software engineer Dave Cutler from DEC to head up the project. He had been the lead architect of VMS and he was the obvious candidate for the job by a massive distance. In fact in many quarters Windows NT was seen as essentially a rewrite of VMS.

## *The Device Driver Challenge*

At the time as a freelance programmer I'd acquired a modest reputation as someone who could take on programming tasks in unknown environments, figure them out and deliver working code. As a result during 1980, following the signaling project, I was approached to write a VAX/VMS 1.6 device driver for an unsupported magnetic tape unit. VMS 1.6 was a very early release of the OS to say the least and so yet again, like my experience of

Stage 2 as described by Peter Brown, I was in a world of my own. Fortunately, the VMS documentation was of a reasonable standard though inevitably some crucial elements remained undocumented and were a nightmare to figure out.

A first for me was that I was given a VT100 terminal and a modem so I could do the initial development from home by dialing in to a computer time sharing service in central London. Writing a device driver for an external device introduced me to several programming techniques and issues that were new to me. They included hardware interrupt handling, my first experience of event handling; the use of memory queues to pass data to another lower priority process for subsequent processing, my first experience of inter process communication (IPC); writing time dependent code sequences that had to execute in less than a specified number of microseconds; directly controlling a physical device; and using online debugging techniques in real time.

Writing the device driver was quite an experience and involved some nail biting problems with undocumented features but finally it was done and the tape unit worked OK under VAX/VMS. By then I felt I knew the VAX and VAX/VMS about as well as was possible but this was the last time I ever felt that I knew everything about a computer and its operating system. In the future as computer environments became more and more complex it became increasingly difficult for a sole working commercial programmer to achieve this level of familiarity. With the notable later exception of PCs running MS-DOS of course.

## *The 70s Draw To A Close*

(#AI) Meanwhile in 1979, twelve years after Marvin Minsky had said “Within a generation … the problem of creating artificial intelligence will substantially be solved.” AI research was encountering many problems. So many in fact that Hubert Dreyfus published the second edition of “What Computers Can't Do” with a new lengthy introduction. But while AI development was struggling hardware developments seemed to be accelerating in line with Moore's Law.

# 1981 New York City – Computing USA

In 1981 I travelled across the Atlantic Ocean to the United States and settled in lower Manhattan, downtown New York City. The city was still in the aftermath of its big financial crisis, the roads were full of pot holes and the subway was completely covered in graffiti but it was a very exciting place to be and was beginning to emerge from its crisis years. Urban chic was beginning to appear and young Wall Street workers were starting to be found in funky clubs and bars in Alphabet City, avenues A to D. Fellow British ex-pat Jo Jackson caught the feeling of those times in NYC very well in his 'Night and Day' album which was released the following year.

I quickly discovered that the main demand for programmers in NYC was on Wall Street in the financial industry and not in the technical areas I was now most experienced in. My CV at the time did not look at all promising for finding work but I did eventually find some COBOL compiler work at Advanced Computer Techniques (ACT) after a chance meeting with an IT recruiter and freelance agent in a New York bar.

## *Welcome to the Big Apple*

ACT was a company run by Charles Lecht, a well known American computer guru and the Wikipedia entry for ACT says *"Lecht was a colorful and flamboyant character with an idiosyncratic sense of style, who went around on a motorcycle and was described as a "showman" by colleagues, customers, and competitors alike."*. He certainly ran a friendly if somewhat disorganised working environment and my colleagues ranged from male Vietnam veterans to diva like female managers and everything in between.

I worked on a COBOL compiler which was being written in Pascal, so there was yet another language to learn. Interestingly Pascal source code compiled down to an intermediate code (remember Micro Focus and CIS COBOL) known as p-code, so that technique was now becoming more widely used. Pascal was a strongly typed language so what a binary value represented could be made unambiguous, unlike in loosely typed languages where a

binary value might be an address, a numeric value, ASCII character(s), and so on. Incorrect interpretations of what a binary value represented, programming errors in fact, were common causes of catastrophic run-time errors but in strongly typed languages these errors would usually be caught by the compiler. The COBOL compiler we were working on generated machine code rather than intermediate code and so the emphasis was on translating COBOL verbs into the minimum number of machine code instructions possible. My earlier programming of the interrupt handling in the VAX/VMS device driver came in handy in creating those short efficient code sequences.

Even though programming languages were proliferating there was still a lingering belief that it would be possible to design a single programming language that could be used in all situations. A language aiming to fill this role was being designed at the time and compilers for it were being written, it was called Ada, as a tribute to Ada Lovelace. However history has shown that even if there could theoretically be a single Esperanto like programming language it was not going to be Ada.

## *Do Androids Dream Of Electric Sheep?*

(#AI) It was while working at ACT that a co-worker told me about Hubert Dreyfus's book “What Computers Can't Do” and I finally came to read it. It said that all AI work up to this point wrongly viewed intelligence as a data processing problem so that if enough data was held in complex data structures and was manipulated by clever enough software algorithms then the Turing Test could be passed. Dreyfus rejected this and said that intelligence required things a data processing machine could not do. His classic example was chess playing software where massive amounts of computer power could play chess impressively because of their ability to rapidly look a long way ahead in the chess game and consider the results of making millions of different moves. Chess grandmasters however can just ‘see’ the correct move and this ability to ‘see’ the correct move could not be analysed in terms of data processing. Dreyfus thought AI would need to operate in the same way, but would that really be necessary in order to pass the Turing Test? As long as the correct result was arrived at did it matter how it was achieved?

A possibility raised in the book was that perhaps Dreyfus's own version of AI could only be achieved with a 'protein computer'. But what did that mean? Dreyfus claimed that only a protein computer, a computer with a 'real' organic body, could truly experience feelings such as pain and that was necessary in order to make truly intelligent decisions. It's an interesting thought but does passing the Turing test involve, say, feeling pain or just being able to simulate feeling pain. Does it matter?

In the world of culture this was already being discussed and in Ridley Scott's legendary cult movie 'Blade Runner' (1982), from Philip K Dick's novel 'When Androids Sleep Do They Dream of Electric Sheep?', the bioengineered human-like replicants represent a test case. Ridley Scott's artistic focus had moved beyond the Turing test for AI to the more complicated (and totally speculative) 'empathy' test for AI. Could a replicant have real emotions and thus feel real empathy for other living creatures rather than merely simulating it? To identify rogue replicants that were illegally posing as human beings they were tested for lack of empathy using a 'Voight-Kampff' machine. Blade Runners (replicant hunters) used the machine to test suspected replicants. (Replicants of course could have had REPLICANT stamped across their foreheads when they were created but the novel and the film would have been rather boring if that was the case.). Whereas in the movie '2001' the question of whether HAL had emotions was never asked, when it came to replicants their lack of emotions and therefore empathy was their defining characteristic.

Once again artists were ahead of the technology and already focusing on whether artificial beings could have real feelings, a question that is yet again under discussion. This question was not really answered in the movie but the distinctly unprofessional relationship that developed between 'super' replicant Rachael (Sean Young) and human (or was he?) Blade Runner Rick Deckard (Harrison Ford), implied a very ambiguous conclusion.

Yet another approach to these questions was offered in 1984 in the movie The Terminator. The anti-hero in this case was a cyborg, perhaps best described as protein organics enhanced with robotic computing technology. Could this hybrid technology and/or

replicants represent the future and perhaps make the whole AI discussion irrelevant? The jury is still out.

## *The IBM PC Arrives*

The IBM PC was first released in 1981 and a new era of the digital age arrived that was defined by its capabilities, as well as its lack of them.

The IBM PC was based on the Intel 8088 microprocessor and featured two 8 inch floppy disk drives for data storage and to hold the MS-DOS operating system, there was no hard drive. At the time I was busy getting settled in New York so I had no direct experience of the PC but its implications and effects made it a game changer.


An 8 inch floppy disk

# MS-DOS

Since the time I had used the ISIS II operating system on the Intel 8080 at Micro Focus a number of competing

microprocessor operating systems for Intel chips had been created. Widely accepted as the best was CP/M which could reasonably be described as a cut down version of RSX11, it was the best around and was the accepted standard. There was even a multi-processing, multi-user version of it called MP/M. It came from a company called Digital Research and it seemed to be a no-brainer as IBM's choice for their PC. But instead a barely known, rather flakey, single user, and very technically limited OS called MS-DOS was chosen by IBM. Why? Well the legend goes that IBM went to Digital Research to do a deal for CP/M but DR's owner Dr Gary Kildall was flying his personal airplane and was so confident they would have to do a deal with him that he carried on flying and kept them waiting. When he finally landed his plane the irritated IBM representatives had gone home and they eventually did a deal with Bill Gates. The rest is history. The truth, or not, of this incident is still shrouded in legend and the Wikipedia article on Dr Kildall lays out some of the background.

## Back To Batch But On To Freedom

Ironically the PC brought back the batch environment whereby a single user executed single programs one after another. But now a programmer could develop their own software and given the very basic nature of MS-DOS there was plenty of scope for developing utility programs. A well-known entrepreneurial programmer who took full advantage of this was Peter Norton whose popular Norton Utilities added much needed functionality to the limited MS-DOS environment. There were many others too.

The PC also quickly became a programming engine initially through Turbo Pascal and BASIC which were soon joined by other languages such as C and COBOL as their compilers were ported to MS-DOS. Now mainframe and server programs could be partially, or completely in some cases, developed on the PC. Programmers really could now own the means of production and could finally break free from their reliance on their employer's computer hardware. The programmer's revolution had happened.

## The User Becomes The Tester

Unfortunately not all companies creating software for the PC were as thorough as Peter Norton and many took the view that if a

program had bugs in it the worse that could happen was the PC would crash and the sole PC user might lose their data. But the users were just consumers who probably had fairly low expectations of PC software so fairly buggy software was often released to grab market share while subsequent upgrades corrected the bugs the users had uncovered. It was long way from the predefined test plans and expected results of the original batch days.

## Lipstick On A Pig

(#UI) The IBM PC introduced the hugely influential technique of putting 'Lipstick on a pig', or 'Lipstick on a corpse' as more pessimistic people characterised it. You could run a terminal emulator on a PC and connect it to a server such as a VAX (you could connect them up to a mainframe too but that was a more complex situation) and use it as if it was, say, a VT100 terminal. With some additional programming you could intercept the communication between the front end PC and the back end server and present simple graphical elements on the PC screen, such as menus and drop downs. Nothing was changed in the back end logic but the user interface, probably a traditional form display originally, could be presented in a more graphical format. (#UI) You had succeeded in putting front end lipstick on a back end pig.

This technique was my first glimpse of the emerging evolutionary archeology of software as it componentized further and broke apart into front end and back end components:

- Existing software that has been in production for long enough to be practically error free and is critical to the business will be retained rather than replaced if humanly possible. The world of back end production software is essentially conservative.

- Because of the above tendency new front end software generally gets created using lipstick-on-a-pig techniques so as not to disturb the back end software. The pressure to develop it comes from the users and the desire of programmers to use the latest software development tools.

- (#UI) Eventually the front end/back end interfaces become such a mess that efforts are made to introduce a standard

‘layer’ between the two, nowadays often known as an API (application programming interface), to be used by all programmers. Defining the standards is usually a commercial battleground and soon becomes a religious war. Different commercial organizations try to control the market by imposing their own de-facto custom standard while industry and/or academic organisations try to rise above the fray and define an open standard. Results vary and we will see plenty of examples of this.

## The Killer App!

The IBM PC popularized the first ‘killer app’, the spreadsheet, and it would be many years later before another app was produced of comparable importance, perhaps not until the web browser.

The first spreadsheet was Visicalc but it was Lotus 123 that became predominant. Spreadsheets allowed normal business users to do some basic programming by using mathematical operators (such as AVG for average) to define formulae in the spreadsheet cells that calculated results by using values from other cells in the spreadsheet. They could also enter text values to help create printed reports which could also contain graphical output. Famously many trading desks on Wall Street got their own PCs and by themselves, or with the help of contract programmers, started to develop basic software systems independently, and out of the control of, the corporate IT department. The quality of the software produced was often questionable and the implied attack on the authority of the IT department is a corporate political issue that rumbles on to this day.

## Dialling In

As noted you could run a (usually) VT 100 terminal emulator program on a PC and then directly attach it to a server machine, say a VAX, as a replacement for a real physical VT 100 terminal.

But more interestingly you could attach it to a server over a telephone line using a modem, in the same way I had used a dumb VT100 terminal from home when developing the VAX/VMS driver.

An RJ11 modular jack plug

Even better, in the US phones were not hardwired into the wall but attached to the wall via a modular jack plug, the ubiquitous RJ11, and a wall mounted socket, so this was easy to do.

A wall mounted RJ11 socket

The hassle of using an acoustic coupler was not necessary.

## *A Niche On Wall Street*

Meanwhile at ACT I was only viewing compiler work as a stop gap measure, though it was very good experience as it gave me a chance to become familiar with the considerable differences between the US and UK office working environments. I kept looking for some way to get Wall Street experience on to my CV and eventually found a niche on Wall Street in 1982/3 on a project at Citibank to move an existing funds transfer network from PDPs to VAXs. Many financial networks had been implemented on PDP11 hardware with software written in MACRO-11 and there was a trend, probably promoted by DEC, to upgrade them to the more modern VAX/VMS and to use a high level language. Management on Wall Street always liked to be using the latest technology simply because it was a good marketing tool when selling financial services to potential customers whereas traditional IT departments were

generally more conservative with an ‘if it ain’t broke don’t fix it’ attitude. Accordingly technical decisions were frequently made, so it was often claimed, on the golf course. If a non-technical Wall Street board member was playing golf with a computer vendor’s sales manager it could be an easy sell, like a lamb being led to the slaughter in fact. Not the best way to make technical decisions but many of them were only explainable in that way. However the financial services companies always had the resources, i.e. money, to make these decisions work whatever the cost. But frequently, including in this case, the existing system was working more or less satisfactorily and it was absolutely mission critical. So the potential for disaster was high and to minimize this risk Citibank had started a project to evaluate the capabilities of the software and hardware available in the VAX/VMS environment.

One issue to be investigated was the performance of inter-computer networking and my task was to look at the performance of DEC’s DECnet networking software so I started by pouring over the DECnet manuals. This was very early in the history of data networks when there was no such thing as a universally accepted network protocol in the way TCP/IP would become one day. As usual in computing various incompatible vendor dependent networking products were being developed including DEC’s DECnet, IBM’s SNA and Xerox’s XNS. Even so at the US Department of Defence they were already unhappy with this situation and had started to work on TCP/IP. I learned a good deal from my encounter with DECnet in particular how to configure and initiate network server processes (a totally new concept to me) in the sending and receiving machines and how to use what we now call a networking API in two application processes, one in each machine, to send data to each other. This was not a trivial task in those days and it kept me fully occupied, it opened up a whole new world to me, networked super-mini-computers and their networking software. It was also the first time that I worked with two simultaneously running application programs in different computers communicating with each other without using any files to store the data.

Being so focused on networking I was not giving much thought to the other technology developments taking place and though I knew the IBM PC had been released in August 1981 I had not had any real experience of it. However, at Citibank our

programming team, the usual varied collection of contractors, were astonished one morning to see a team member arrive carrying, actually more like struggling with, the first IBM PC compatible ‘transportable’ computer from Compaq.

He was rather overweight and it was a steamy New York summer day so he was pouring with sweat and looked like a leading candidate for a heart attack. But he proudly lowered it on to his desk and gave us a demonstration.

The First Compaq (Trans)Portable PC

The key thing was you could run a VT100 terminal emulator program on it and then attach it directly to the VAX in the place of one of the VT100 terminals we were all using. This was all very nice though given the size of the screen it was hard to get too excited about it, but we were missing the crucial point. Most terminal emulators included the capability of uploading files from the PC (the Compaq) to the server machine (the VAX) and, far more importantly, downloading files from the server to the PC. It was now possible to get a copy of any code you’d written on the server into a file on your very own PC. This was probably, well certainly, legally dubious but it started to clear the way for individual contract programmers to create their own collections of source code. A more or less acceptable technique that became popular was to amass a collection of useful code snippets you had written on various projects and then massage them together into a ‘new and original’ subroutine library that could be sold as a ‘tool set’. The programmer’s revolution was continuing.

(#UI) Around the same time, in 1984, our team went to an electronics store just off Wall Street one lunchtime and watched a demo of the brand new Apple Mac. It was amazing to see a

graphical user interface (GUI) for the first time, which was widely claimed to be a direct rip-off of Xerox's PARC labs GUI research work. However first impressions were that the Mac was just too small for serious work.

[***Information point:*** This seems the appropriate moment to mention the major role XEROX's PARC research labs have played in the development of computing. They introduced many innovations that were taken up by others including laser printers, WYSIWYG (What You See Is What You Get) text editors, GUIs, the Ethernet, the 'true' object oriented programming language SmallTalk and the MVC (Model View Controller) software architecture, plus many others.]

Meanwhile in the cultural world the first cyberpunk novel was published, 'Neuromancer' by William Gibson. Even though a slightly crazy co-worker recommended it to me at the time it was only several years later I read it. (It seems I was always a few years behind in getting around to reading the crucial books.) Once again an artist was way ahead of reality and so, while in the real world I was struggling with DEC's early networking software, in the fantasy world of Neuromancer the existence of a sophisticated worldwide 'cyberspace' was imagined. (#UI) In cyberspace 'console cowboys' equipped with 'cranial jacks' could 'jack in' directly to the 'Matrix' virtual reality system. Cyborgs meet networks. The number of subsequent books and movies that have taken ideas from Neuromancer is incalculable.

But back in the real world I was becoming aware of how much attention the Unix operating system was getting in the computer press and I began to pay more attention to it.

## *Unix Takes On VMS*

Up until now DEC, with its PDP and VAX computers and RSX11 and VMS operating systems, was dominating the server market but this situation was now changing, in large part because of work that had been carried out at Bell Labs. Bell Labs had been a pioneer in developing new hardware components like the transistor but had increasingly taken the lead in a lot of software

developments. Because of the many different computer hardware environments at this time, all with their different sets of machine code instructions, software portability was highly desirable and this goal was pursued at Bells Labs. In Peter Brown's article about CIS COBOL in Appendix B the theory behind portable compilers is described and by using many of the same techniques Bell Labs developed the highly portable Unix operating system by:

- Creating a compilable high level language, C, that nevertheless had enough low level features to be used for systems programming thus largely replacing the use of machine dependent assembler code.

- Producing a C compiler in a highly portable form using the techniques described in appendix B.

- Creating the highly portable Unix operating system by writing it in C.

This allowed the Unix operating system to be implemented on any hardware platform, more or less, by simply porting the C compiler to it, a fairly straightforward task.

In the late 1970s the Unix operating system's source code was licensed to many academic and commercial organisations by Bell Labs resulting in its appearance on many server brands but under different names, such as Solaris (Sun), AIX (IBM), HP-UX (Hewlett Packard) and so on. Although all these variants were based on Unix they were not compatible and you could not just take a program from, say, Solaris and recompile it under AIX and expect it to work. Computer manufacturers still did their utmost to tie you to their hardware by incorporating non-portable elements in any software they produced. Nevertheless as a result of the licensing of an almost off the shelf operating system many minicomputer and workstation manufacturers were now in the Unix market, some new like Apollo and Sun, and some older like IBM.

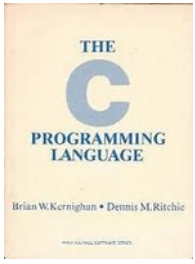
## *On The Road To Bell Labs*

I felt that I needed to get Unix and C on to my CV and where better to do that than at its birthplace where it all started? Bell Labs was in New Jersey on the other side of the Hudson river and would

involve some travel but I knew that other contractors from Manhattan worked there so I reasoned that there must be some way of coping. The Holy Grail of the various Bell Labs, which were scattered around New Jersey, was the Murray Hills lab, it was the real Unix/C holy of holies. But it was hard to get into and so I eventually got a contract at the Whippany lab. It was closer to Manhattan and was more of a traditional telephone company engineering lab but they used Unix and C so that would get it on to my CV. The project I worked on was the software for a diagnostic device used in network line testing.

Commuting to Whippany from Tribeca was not a problem as I managed to get a seat in a van that did the journey for about a dozen Bell Labs employees and contractors. It picked up a few of us from a pizza place on Canal Street close to the Holland Tunnel that opened early in the morning. Van passengers could volunteer to drive and they got a fare reduction though only a few did so and I myself declined. As well as the reduced fare the driver chose the radio station we listened to which almost led to blows if they tuned to one of the new shock jocks like DJ Howard Stern. It also created the concept of 'van music', uncool music I would only admit to knowing to my friends by saying "Oh yeh, I heard it in the van". Nevertheless plenty of great music was being released at the time by Cyndi Lauper, Madonna, Dire Straights, Prince and many others.

At the Labs I was given a copy of the now legendary 'White Book' written by Kernighan and Ritchie accompanied by a Unix manual and I set about absorbing them both.




Entering your C source code for a program was done with the ubiquitous Unix editor vi from a dumb terminal. When vi was developed at Bell Labs the quest for portability had been continued by using ordinary characters for edit commands so that it could be used with any dumb terminal's keyboard. Those keyboards of

course did not have all the control keys that modern keyboards come with such as arrow keys, Home, End, Page Up, Page Down, and so on. As with everything in the Unix world the vi key combinations ranged from simple and intuitive to complex and barely comprehensible. It operated in 2 modes, insert and command modes, and in command mode, for instance, it used the normal keyboard characters ‘h’ ‘l’ ‘k’ and ‘j’ as substitutes for the absent left, right, up and down arrow keys.

Compared to the VMS operating system and to high level programming languages like COBOL, Coral 66 and Pascal, Unix and C certainly represented a much lower level, closer to the hardware and more like assembler coding. There were relatively few reserved words in the C language with the main ones being for, data classes (**int, char, double, float**), data types (**signed, unsigned, short, long**), looping verbs (**for, while, continue, break**), decision verbs (**if, else, switch, case**) and control verbs (**goto, return**). C took the componentization of computer programs several steps further through the use of functions, a specific form of subroutine. Functions, returned a single value that might then be part of an equation. You could write a code snippet like:

```
area = calculate-area(length, width);
```

Where calculate-area was a function you, or somebody else, had written which returned the size of an area. Functions could also be included in equations.

```
double-area = 2*(calculate-area(length, width));
```

Because of this feature a large part of a C program consisted of calls to functions you had written yourself or were resident in an object library that would be accessed at link time. There are many standard library functions which are always available in Unix implementations and it is easy to think they are part of the C language but that is not true. These include file handing functions such as fopen(), print functions such as printf() and character string functions such as concat().

As well as the C reserved words and the standard library functions the C language included a large number of built-in

operators which provided operations all the way down to the individual bit level:

- Arithmetic Operators (+, -, *, /, ++, etc.)
- Relational Operators (==, !=, >, >, etc.)
- Logical Operators (&&, ||, !)
- Bitwise Operators (&, |, <<, ^, etc.)
- Assignment Operators (=, +=, <<=,)
- Misc Operators (sizeof(), &, *, ? )

As a result C language programs could become as complicated as assembler programs, if not more so, but then C had been specifically designed for systems programming. This gave rise to the joke that C was the only high level language in which you had to write more lines of code than if you'd used assembler. C programs could be so notoriously incomprehensible that an "International Obfuscated C Code Contest" was started that selected the most incomprehensible examples each year. However, far from discouraging the use of C it became popular and widespread as a commercial programming language. It seems to be true that the more complex a programming language is, the more that programmers want to use it.

## *Unix Development*

The Unix command language was somewhat similar to C and by using it you could create complex Unix shell scripts to automate processing tasks. However, compared to VMS's DCL the Unix command language was far less readable and more formula like and so shell scripts, like C programs, could be written in a fashion that made them very hard to understand. There were a large number of Unix commands available to use in a script varying from the fairly trivial to the highly functional and Unix command names were famous for bearing little if any relationship to what the command did. A well known example is the command for listing the files in a directory, the Unix command is **'ls'** and not **'dir'** as in most other command languages.

Building executable programs from C source code in the Unix environment normally involved linking together various different object modules that were compiled separately and probably stored in object libraries. When calling an 'external' C

function that was resident in a different module C programs would use #include files (files merged into the C source code at compile time) of source code statements defining the interfaces, or APIs, for the external functions. These interfaces were not fully resolved until the link phase.

This meant that the interdependencies between #include files, C program source code files, C program object files/libraries and executable program files could become very complicated. In order to build an up to date version of an executable program it was vital that all the component files involved in its creation were themselves up to date. This required that all files had been created using the latest versions of any files on which they depended. These inter-dependencies were defined in a 'make file' created using vi which was used by the Unix **make** command to check if any file was created at an earlier date than files on which it depended. If this was true the make command would recompile and/or relink it as directed by commands embedded in the make file. By repetitively doing this all the program components would be brought up to date as would the executable program itself.

As C source code files consisted of text entered through vi a C source code listing was very basic and lightly formatted but it was perfectly satisfactory for the program development process. However this was not acceptable for professional looking documentation or business documents and this was in the days before WYSIWYG (What You See Is What You Get) word processors. As a result software type setting products appeared to fill this gap and at the Whippany lab the Unix product we used was Latex. Latex took as input a text file, usually prepared in vi, that contained both the text for output and the print formatting commands. This was essentially a precursor to HTML which of course underlies everything you see on the Internet. Like HTML Latex depended on the insertion of formatting tags in the document's text, for example to increase the font size you would key in the tag \LARGE. Many tags existed in Latex that were the equivalent of current HTML tags and like HTML files Latex files were not always easy to read.

## *The Instinet Trading System*

After a year of commuting out to Bell Labs the project there was winding down and I'd now learnt C and Unix so I looked for a contract closer to home, back on Manhattan, and I found one at Instinet in 1985 in their mid-town offices.

The Instinet system was possibly the first ever electronic trading platform and made use of some real time interfaces provided by the NASDAQ stock exchange to allow a few hundred traders to trade stocks quoted on NASDAQ online in real time. The system consisted of two PDP11s which, surprisingly, resided in a cupboard in the system programmer's home in Massachusetts. They connected to a network of a few hundred terminals including Instinet's own trading floor in Manhattan over leased lines. It was a typical PDP implementation and everything was implemented in largely undocumented MACRO-11 code and that included the business logic, the rudimentary database, the network software, the VT100 screen and keyboard handling, the interfaces to NASDAQ, everything.

The system's creator was an English guy who lived and worked in Massachusetts who was quite possessive about 'his' system and was less than happy that Instinet had been acquired by two Wall Street investors, Bill Lupien and Fred Rittereiser, who saw the potential of online trading. This was at a time when many Wall Street people insisted that only human beings located on physical trading floors could make financial markets work properly. Bill and Fred had seen beyond that conservative and largely self serving objection and would eventually be proved right by the acquisition of Internet by Reuters. They had taken on David Rosensaft (memo signature DNR), a way ahead of his time electronic trading system visionary, to create a team to modernize the system or at least to make it more presentable.

(#UI) One programmer was busy putting lipstick on the pig by presenting the VT100 interface in a PC terminal emulator and enhancing it to include drop downs and other menu items. This, somewhat ironically, slowed down the ability of the traders to make trades, a phenomenon that often accompanied the introduction of GUIs over the next few years and led to the development of keyboard shortcuts for power users so they could bypass the GUI.

It's a good example of the not uncommon problem that though programmers and designers may think certain technology features are cool, users may hate them.

Given my compiler experience I was asked to design a trading language allowing some level of automated trading to be defined, though how it was to be executed remained unclear. But it meant I had the opportunity to learn the compiler tools lex and yacc. Various team members were engaged in other modernisation tasks, all somewhat speculative and none of which would actually modernize the back end system. The system's creator was allegedly converting all the MACRO-11 to C to at least make the code slightly more readable and portable. Given my English roots and accent I was sent up to Massachusetts to see him and try to get some, any, information from him and perhaps even an agreement that he would work with the modernization team. But predictably even though we had a pleasant enough meeting there was no way he was going to give up any control.

There were some ongoing efforts to sell Instinet and whenever potential buyers were being entertained they would be taken through our work area and given an optimistic view of what we were working on. Occasionally we'd get to sit in on the sales presentation and I always enjoyed watching Fred Rittereiser's pitch. He was an ex NYPD cop and would inevitably drop in phrases like 'back then when I was out there dodging bullets' which would really excite the potential buyers who were mostly desk bound financial types. Once he speculated on a day when perhaps everybody in the world could participate in online trading, which at the time was a completely insane idea. He admitted he couldn't get his head round the thought but felt instinctively that should it ever happen the whole system would quickly grind to a halt because of the massive volume of opposing trades being executed.

Even prior to the enhancements we were working on the very existence of the Instinet system had already been putting front end lipstick on the back end pig. It did so in a way that has plagued many business modernisation efforts since then. Although it allowed trades to be carried out in milliseconds by the Instinet front office system (the lipstick), clearing and settling the trades in the back office system (the pig), still happened in the traditional way. Clearing and settlement still involved paper and took days or even weeks to

happen so of course massive backlogs built up. The front office was now operating hundreds of time faster than the back office. This same performance mismatch has plagued many attempts to update corporate and particularly government business processes as it still does to this day. Neo-Dickensian paper based back office admin systems absolutely must be replaced by modern digital equivalents before the real benefits of new faster front office systems can be realised.

## *The Wall Street Shuffle*

It was in 1974 that 10cc released their excellent song “The Wall Street Shuffle” but it would be considerably later before I finally managed to shuffle on to the real Wall Street myself.

(#AI) Before I did so, and 15 years after saying the AI problem would be solved within 5 years, in 1982 Marvin Minsky, head of AI research at MIT told a reporter “The AI problem is one of the hardest science has ever undertaken”. But incredibly it was against this background of setbacks and failure that Wall Street was seduced into falling in love with AI. It was not the first time, and it wouldn't be the last, that the financial industry would leap onto a technology bandwagon with little idea of what it was all about. I came to believe that because the financial industry makes so much money the idea of pouring some of it into technology that might allow it to make even more money was a no-brainer.

Suddenly if you could claim AI experience either practical or academic you were hot property and could command an eye-watering daily rate on Wall Street. The platforms of choice for this AI work were the Sun Microsystems workstations since PCs running MS-DOS were woefully underpowered. In fact, as Dreyfus had shown and as I managed to see for myself, nothing special was happening here, it was mostly list processing of a fairly boring nature. Still, it was good for AI consultants and for a Wall Street executive to be able to say on the golf course to potential customers that “we have a crack team working with AI” was very good for business.

But the good news for non-AI programmers, like myself, was that by 1985/6 having a group of computer ‘wizards’ of any description programming away was a must have for the big Wall

Street companies. In 1986 as the Instinet modernization project slimmed down and the company was lined up to be sold to Reuters I got a contract at Goldman Sachs on Wall Street itself through some of the guys I'd met at Citibank. I'd made it to Wall Street proper at last.

## *The Great Vampire Squid*

In 2010 Matt Taibbi published a highly incendiary article in Rolling Stone in which he observed:

*"The first thing you need to know about Goldman Sachs is that it's everywhere. The world's most powerful investment bank is a great vampire squid wrapped around the face of humanity, relentlessly jamming its blood funnel into anything that smells like money."*

Quite an interesting thought I'm sure we'd all agree.

But in 1986-8 when I consulted at Goldman did it feel like that to me? Frankly, not really. Rather it seemed more like a somewhat shambolic group of extremely well paid and well connected young individuals who had somehow got their hands on an unlimited budget and massive expense accounts. I well remember standing in a conference room looking through the glass wall across the huge crowded open plan office and Steve Krolak, a fellow technical consultant soon to become a good friend, turning to me and saying "I wonder where all the grownups are?". That's what it was like. But make your own mind up, I can only tell you what I witnessed while I was there.

With the emergence of PCs and mini-computers fairly uncontrolled software development had been taking place at many of the Wall Street investment banks around the trading desks and therefore beyond the reach of the IT departments which mostly handled back office processing. Goldman was no exception and had a frightening amount of undocumented software in a VAX/VMS environment and for some of the modules the source code had been lost entirely and only the object code was available.

(#PZ) At the same time modules containing key financial calculations such as their 'options model', which was a must have

for investment houses at the time, were incorporated into many different programs. It was critical that all these programs were using the latest version of the options model and it was far from clear that this was the case. In an effort to tidy things up they recruited an experienced systems guy and turned him loose to try to sort things out. As was the case at many Wall Street firms he could, and did, easily take on a dozen freelance consultants as they didn't show up on the firm's headcount. Glory days for consultants. When I was interviewed I was asked to name the three methods of IPC (inter process communication) in VMS and I answered correctly (a system semaphore, a shared memory segment with access to it controlled by a system semaphore, a mailbox (not what that means nowadays)). It was the first, and only, time I was ever asked this and was completely irrelevant to the work I would do at Goldman.

(#PZ) My first assignment was to implement an object module library to help make sure that the most recent versions of object modules were being used by all software. VAX/VMS had its own versions of the Unix make software in the form of two products, CMS (Code Management System) and MMS (Module Management System). Between them they provided all the functionality of the Unix Make command which I incorporated and provided to the librarian through a VT100 terminal. (#UI) I created the user interface with a menu based system I'd written in the DCL command language.

As 1987 progressed our team of contractors kept growing and by 19th October 1987, Black Monday, we numbered twelve. By then our boss had decided that myself and Steve Krolak would co-lead 'the project', a rather hazy concept of his that the existing portfolio management system should be completely replaced with more modern software developed from the ground up. Whether he had the backing to do this from senior Goldman management was not exactly clear and as markets crashed on Black Monday the team became extremely nervous about our future as contractors at Goldman. But although our boss was quickly terminated nobody seemed to pay us any attention so we held a team meeting and decided Steve and I would look for somebody new to report to, keep working, and hope for the best. This we did rather successfully and carried on for another 18 months before the axe finally fell.

How could that be? I think it was because part of our role was as window dressing to convince clients, and even the Goldman management, that serious intellectual effort underlay the fancy graphs and reports that they were shown. But in fact they may well have been produced by Lotus 123 based on data that came from, well, who knew where? I decided that computer generated reports and graphs were part of the mythological apparatus used by Wall Street's financial fortune tellers to create their gnomic predictions about the financial future. It was remarkably similar to the way that in the world of Odysseus ancient Greek seers interpreted animal entrails when their leaders consulted them on whether or not to take a specific course of action.

Whatever the truth of the matter we got to use some key technologies that were emerging at the time in particular the object oriented programming paradigm, SQL and relational database technology.

## *The Basics Of Some Crucial Developments*

During the 1980s, at Goldman Sachs and on other assignments, I came face to face with many of the new technologies and programming practices that were constantly emerging. There was a lot happening and the rate of change in computer technology seemed to be ever accelerating. These new techniques and technologies were a result of the growth and increasing complexity of computer systems and the ever-widening areas in which they were being deployed. Computer programs had come a long way from their basic 'Data In. Data Out' origins. Three of the key changes behind these developments were:

- The role of the data being processed had become far more important in program and system design because of its increasing volume, the need for concurrent access to it, and the need to protect it from poor programming techniques.
- Interactions were increasingly required between running programs that were active within the same computer using IPC, or active in different computers connected by networks.
- (#UI) The ability to develop complex graphical user interfaces on PCs and workstations was dramatically transforming user's experiences and leading to a corresponding increase in user's expectations.

Some of these developments are discussed below.

## Relational Databases and SQL

In the early days of commercial computing the data being processed had been stored on magnetic media in relatively simple and modestly sized file structures and they had only been accessed by one program at a time. But the rise of online systems had introduced the problems of concurrent access of a single file by multiple users and the increasingly large files, or data sets, being used by programs. The major fear was if two programs updated a data set at the same time one of the updates would be overwritten by the other and therefore not applied. These ever-growing data sets were starting to be called databases.

To solve the concurrent access problem the technical concept of a transaction that exhibited the ACID properties (atomicity, consistency, isolation, durability) was defined in the late 1970s. Though it should be noted that the word ‘transaction’ is often used in a far less precise fashion. The classic example of a transaction is a funds transfer from one bank account to another which involves both a DEBIT operation (on the source account in the database) and a CREDIT operation (on the target account in the database). The problem is if something goes wrong in between the two operations, or with one of them, then inconsistent data will be stored. By defining both updates as part of a single isolated atomic transaction this problem is solved since you begin the transaction first, prior to both actions, and then either commit it or roll it back (restore all the data to its previous values) after both updates have completed successfully, or not. No other transaction can interfere with this transaction while it is happening as it is isolated. A transaction is indeed atomic, it is ‘all or nothing’ and it required the relevant programming language components to be created for beginning, committing and rolling back transactions. Creating the infrastructure software to create and manage transactions was an extremely complicated task and was not standardized, and nor was the structure of the new databases. Some kind of proprietary tree structured data set was often used with its own difficult to use library routines to access and manipulate the individual data records.

As a result of this chaotic situation a new standard database type was defined, by Codd and Date, using mathematical principles involving set theory, this was the relational database. The essential organizational principle of a relational database is that all data is arranged in a set of tables each one consisting of rows and columns, like spreadsheets. Each row in a table contains an individual data record and each cell/column contains an individual data field.

Another major organizational principle is that no row in a table can have 'repeating fields'. This latter principle can be demonstrated as follows – in older file structures a 'Father' record might have, say, a repeating field for 'Child'. But it was impossible to predict how many Child fields should be created, so usually a design decision was made that 'nobody' had more than, say, 10 children so 10 fields would be enough. But then either a lot of disk space would be wasted, at a time when it was expensive, or sooner or later people would come along who did have more than 10 children. This was unacceptable to Codd and Date and the relational model so two separate database tables would be created instead, a 'Father' table and a 'Child' table. Everybody's children would be added to the same Child table, one record per child, with a column in the table to hold the identity of the child's parent, who would have a record in the Father table. More and more children could be added to the Child table as extra rows without causing problems.

Analysing, or modelling, data in the relational manner to develop a design for the database tables was known as normalizing the data and was done using Entity-Relationship (ER) diagrams. As a 'process guy' rather than a 'data guy' I never really got beyond knowing enough about ER diagrams to be able to interprete them if really necessary. The normalization process would frequently lead to the creation of large numbers of data tables in the database. In order to manage relational data (create, delete, update, query) a new language was required and in an attempt to create a standard language for manipulating data in relational databases SQL (Structured Query Language) was defined. SQL statements could range from the relatively simple to the horrendously complex as could E-R diagrams, so another IT priesthood emerged who could do one or both tasks. At this same time software companies old and new released RDBMSs (Relational Database Management Systems) of which some survived and some did not with Oracle,

DB2 and SQLserver being three well known survivors. An RDBMS supported the structuring of data into tables, the use of SQL, transaction management and much more, they were extremely complex pieces of infrastructure software.

To retrieve all of the data relating to, for instance, a single customer a quite complex SQL query would be defined and embedded in the program's C, Pascal, or whatever, source code. When executed at run time such an SQL query could involve the RDBMS software searching many tables for all the data records pertaining to that particular customer. The data fields from all those tables then needed combining into a single result set. Since relational databases and therefore SQL were based on set theory SQL queries produced result sets which would contain multiple records unless the query had been specifically written to exclude all but one result. The work carried out by the RDBMS to find and assemble the requested data from a potentially large number of different database tables can loosely be described as involving 'database joins' (though database purists will be horrified at this imprecise use of the word 'join').

Although SQL, like COBOL, was supposed to be a standard that could be used on any database it suffered the same fate as COBOL. The database vendors quickly compromised the portability of SQL by including proprietary features that locked you in to their particular RDBMS, but more of that later. A more pressing problem was that the performance of relational databases could be truly horrific with data queries taking minutes to complete due to the multiple database joins taking place. This problem was exacerbated by the poor quality of some early relational database designs and of the early SQL statements used to access them. Writing efficient SQL was a rare skill and it is still an issue today. Ironically a new process had to be added to the relational database design process called de-normalisation which aimed to reduce the number of tables in the database by undoing the effects of over-enthusiastic normalization by breaking some of the relational rules. At Goldman Sachs they had resorted to an additional method of improving relational database performance by acquiring a Britton Lee database machine, a piece of hardware that was solely dedicated to relational database access.

But despite these problems the RDBMS bandwagon kept rolling and soon everybody was using them.

## Inter Process Communication (IPC)

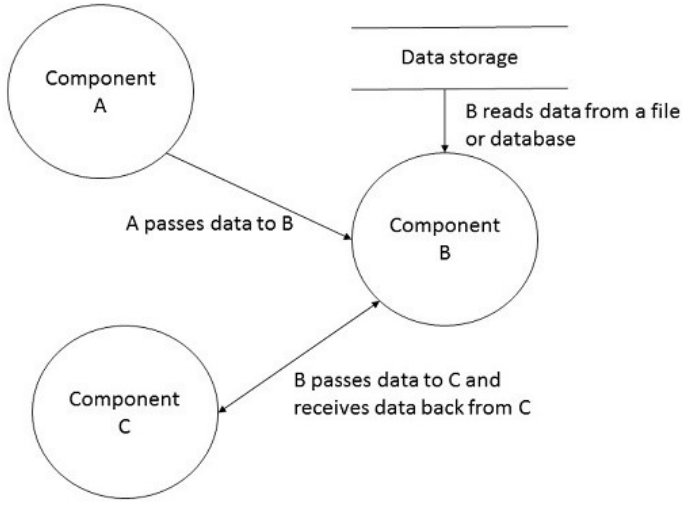
As systems became more complex there arose more and more requirements for programs that were being executed (at which point they become known as processes) at the same time to communicate with each other within the same computer. Inter process communication was known, unsurprisingly, as IPC and there were two basic approaches to doing it.

The first method involved using system semaphores which were a special set of 'system' single bit flags that could be accessed by any running process possessing the necessary permissions. They were accessed using a set of special assembler instructions which prevented any problems of multiple access through the use of hardware interlocking. The system semaphores could be used for basic synchronization whereby process A would cease doing application processing, perhaps by executing a small loop, until process B changed a specific semaphore bit from 0 to 1, known as setting the flag.

Semaphores could also be used to control access to a global data section, an area of memory that could be used by more than one process through special address mapping functionality. Usually one process would insert data into it then set a specific system semaphore which was monitored by another process that then took the data from the global data section and carried out further processing.

(#SOA) Another approach to IPC involved using memory based queues which usually operated in a FIFO (first in first out) mode. Process A would add entries to the memory queue and process B would remove entries from the queue and process them. The other basic queue mode was FILO (first in last out), basically the same as stack processing, though more complex options would emerge. As the same low level program might be using IPC and also communicating across a network using, for instance, DECnet and perhaps doing some SQL for database access, the opportunities for catastrophic errors were high.

# Tracking The Data Flows




Early program design largely revolved around program logic with the data itself being somewhat secondary but Ed Yourdan and Rick De Marco developed a methodology that focused on data and the transformations it went through as it flowed through a program. Data flow diagrams (DFDs) were the basis of the methodology and a snippet of a DFD would look something like the example.

DFDs concentrated the design focus on the data interfaces between program components. This usually meant subroutines and the parameters they expected as these were the source of many errors when the interface wasn't defined and documented clearly. DFDs were purely concerned with data interfaces and were accompanied by detailed definitions of each data flow, the arrows in the diagram. DFDs did not imply anything about the logic and the order of program execution and about this time I realized I was more of a 'process' kind of designer/programmer than a 'data' kind of designer/programmer. I never became expert at creating DFDs properly but I later found them useful when using them, well mis-using them to be accurate, for showing IPC connections in systems involving interacting programs.

## Object Oriented Design & Programming

The use of subroutine and C function software components in ‘imperative’ or ‘procedural’ programming was now common. Good coding practice dictated that such components should never access global variables that could also be accessed by other components. Nevertheless it was often done and could cause catastrophic run time errors. Partially in response to this and perhaps also in response to the complexities and performance issues of SQL a new type of software component was invented – the software object. The ideas of object oriented design (OOD) and object oriented programming (OOP) gained currency and led to the creation of new programming languages and programming techniques and even a new model of database.

Very briefly, the major principle of OO was to hide all data structures and data access methods from the application programmer inside software ‘objects’ which provided ‘methods’ (basically a new name for subroutines) with which to actually manipulate the object. The programmer was to think of manipulating objects and never to consider manipulating data directly. There were also many other principles involved in OO such as inheritance, classes, polymorphism and more, which led to a whole new priesthood who could talk the talk. All these arcane subjects can easily be researched elsewhere.

As a very simple OO example suppose a clock was implemented as a software object. The first program action necessary would be to create the clock object using the new() method, common to all kinds of objects, which would return an object ID to be used when carrying out operations on that specific clock object with clock object method calls. The methods implemented to manipulate clock objects would no doubt include set-time(), query-time(), set-alarm(), cancel-alarm() and so on. The programmer making use of the clock object would have no idea of how or where the relevant data was defined and stored. They were only aware of a clock object that supported certain methods. As with subroutine and C function components software objects could be reusable and shared between many programs.

This new approach, or ‘new paradigm’ as the in-phrase was at the time, initially became very successful in client programs

involving user interfaces. Things like calendars, checkboxes, radio buttons, drop down menus and so on could be implemented as objects and then easily incorporated into an application program's code. New object oriented programming languages such as C++ and SmallTalk were created and older programming languages had object oriented additions made to them. Software objects in general resided in memory throughout the execution of a program thus increasing performance but objects could be saved (or 'persisted' in the new terminology) to a database and later restored from a database. This could be a relational database but it was not always a good fit and a number of object oriented database managers, OODBMSs, emerged.

It has become clear to some people since the early days of OO that for some applications it works well whereas for others the procedural model works better. For other people – well, it's yet another religious war.

## The GUI Meets The Laser Printer

With the development of GUI based WYSIWYG word processing software and the production of laser printers it became possible to use a PC or workstation to produce professional quality documents and this gave rise to a new class of software and job description, 'Desk Top Publishing' (DTP). WYSIWYG products running on PCs also marked the beginning of the end for complete hardware systems dedicated to word processing like the highly successful Wang 1200 WPS.

## CISC versus RISC

Another religious war sprang up with the introduction of RISC (Reduced Instruction Set) chips as an alternative to CISC chips like those used in the VAX. As we saw the machine code for the VAX contained some very complex op codes that could implement whole high level language verbs like CASE, SWITCH and so on. But the set of machine code instructions for RISC chips was stripped right down to a limited set of basic operations that could be executed very quickly and it was claimed that computers based on RISC technology performed better. But to use RISC chips

required that operating systems and compilers had to be rewritten to handle their RISC machine code instruction sets.

Whereas a compiler for a CISC chip might need to only translate a CASE verb into a single CISC machine code instruction, on a RISC chip it would need to translate it into many machine code instructions from a different instruction set.

I'm not sure whether this war has ever been completely decided but RISC environments were successfully created, perhaps most famously Sun Microsystems SPARC chipset, which are still alive and well today.

# 1988 - Distributed Computing

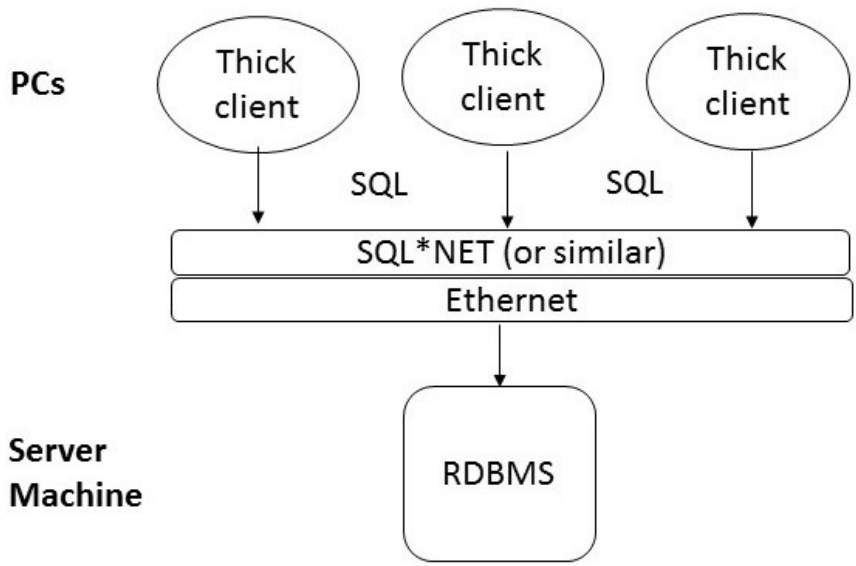
When Goldman Sachs finally brought the axe down on our ‘project’ and terminated our contracts I found work again with DNR at Transvik. But first we should catch up on developments in network technology.

## *Networks Start New Religious Wars*

By the mid-80s the use of several PCs linked via a LAN (Local Area Network) to a server machine holding shared data, usually a relational database, was commonplace. These networks only operated over limited distances so were generally deployed in smaller companies or in the individual departments of large business enterprises. Predictably a number of existing proprietary LAN technologies and standards were competing and the main contenders were Ethernet (developed at Xerox PARC and backed by DEC) and Token Ring (IBM’s offering). However various other players were involved including Novell with Netware and Microsoft with Microsoft LAN and as usual the various LAN de facto ‘standards’ fought it out in the marketplace while ‘official’ standards were put down on paper though nobody really stuck to them. A joke of the time was that the good thing about standards was that there’s so many to choose from. In practice Ethernet would fairly quickly win the LAN war which would lead to the universal use of the RJ45 modular connector for connecting PCs to the Ethernet.

Incredibly the RJ45 Ethernet plug was similar to the RJ11 modem plug and got mistaken, even causing damage to equipment.

The typical LAN configuration was to have tens of PCs, occasionally hundreds, connected via a LAN to a single server mostly acting as a database host. The database manufacturers implemented SQL networking software, such as SQL*Net from Oracle, that ran over the LAN and allowed programs running on PCs to include embedded SQL statements to access data held in the database on the server in a secure transactional manner. This became widely known as **client server** computing where the business logic all executed in an application, known as a 'thick' client, running on all the PCs on the LAN while the back end server hosted the RDBMS. This worked fine in small environments but as we'll see it was totally incapable of scaling up to support high levels of users, in particular to Internet levels. It also introduced a management problem as every PC had to have its own copy of the application software so if any changes were made to it new copies had to be installed on every PC, hopefully in an organized fashion.




Client-Server, (or 2-tier or thick client/thin server, architecture)

A problem that arose with writing the programs to run on the thick clients was that they contained the GUI, the business logic,

and SQL statements. These three elements required completely different kinds of programming skills and knowledge. It was an unholy mixture and the number of programmers who were proficient in all three disciplines was small, if it existed at all, and as a result much mediocre poorly performing thick client software was written. In addition GUI developments had led to the new programming technique of event driven programming and experience in this was not widespread, though those of us who had written interrupt driven routines had a headstart.

The client server architecture was known more technically as the 2-tier, or thick client thin server, or first generation client-server, architecture

(#SOA) In parallel with the development of LANs the development of WANs (Wide Area Networks) continued and focused on connecting mainframe and server machines together over large distances, often very large distances. I had already experienced DECnet at Citibank and the competing technologies and standards from IBM and others were still widely used. This contributed to the need for a more flexible and scalable architecture than LAN based 2-tier client-server could support and so the development of a more robust 3-tier client-server architecture was happening. In the 3-tier architecture (aka thin client/thick server, or second generation client-server) ‘thin’ clients handled the GUI on the front end devices and accessed shared business logic running on ‘thick’ back end servers, or perhaps mainframes. The business logic would handle all the database access and the database itself was sometimes hosted on another set of hardware situated ‘behind’, or ‘below’ the servers. It thus became possible for GUI, business logic, and SQL specialists to focus more clearly on their areas of expertise. For a fuller discussion of 3-tier see the section on The Basics Of Tuxedo.

IBM’s mainframe CICS, referred to as a TP (Transaction Processing) Monitor, was by far and away the best known software supporting the 3-tier architecture. In a CICS managed environment the business logic was written in COBOL (plus SQL for database access) and was implemented as a collection of ACID compliant mainframe transactions accessible simultaneously from large numbers of 3270 terminals handling the UI. In the world of Unix

servers similar products were beginning to appear that would see TP Monitors evolve into what would later be called middleware.

2-tier and 3-tier muddled along together and would do so for several years while various integration strategies could be used to create hybrid environments combining 2-tier and 3-tier architectures. But eventually it would lead to a full blown religious war that was not finally resolved until the arrival of the Internet.

## *The Transvik Project*

The Transvik project, led by DNR, was an attempt to create an online trading system from scratch using the best technology available. Although based in New York it was funded by the Swedish millionaire Jan Stenbeck, a person who gets a brief mention in 'The Girl With The Dragon Tattoo' by Stig Larsen, his fifteen minutes of fame. Having seen a demonstration of the Internet system he decided to develop his own electronic trading system from scratch based on modern technology. He hoped to deploy it in some of the European stock markets and he recruited DNR to implement his vision.

PCs were simply not capable of supporting the processing required for a trader's workstation so actual workstations, often Unix based, from vendors like Sun, Apollo, IBM, etc. were a possible choice. But at Transvik a 100% DEC VAX/VMS environment was selected with DEC workstations connected over an Ethernet LAN to a number of DEC servers that carried out the business logic of order matching and execution. At that time software was being developed by several vendors to simplify writing programs to communicate with each other within the same machine or across a LAN. The objective was to hide any use of memory queues, semaphores and shared memory sections from the programmer. DEC was working on the PAMS messaging system which provided a relatively simple API with verbs like PAMS_SEND, PAMS_RECEIVE and so on. To configure a system to use PAMS, or similar products, was still non-trivial but once it was done programmers could write code fairly easily. This has been an ongoing trend in programming, to make programming easier by making as much functionality as possible part of the configuration of the underlying infrastructure software.

A number of competitive messaging products would appear and given the number of different platforms to be supported their creation involved considerable resources. For instance, if a messaging product was to support messaging between VMS machines and all the different Unix machines it required rewriting the messaging software for each supported platform. It might even involve taking care of any cross platform message translations required, e.g. from ASCII to EBCIDIC. Over the coming years it would somewhat inevitably be IBM's MQ Series that came to dominate the market in terms of platform coverage. MQ Series supported crucial features such as transactional messaging and guaranteed message delivery which became known as 'fire and forget' messaging.

Code for the Transvik system was written in C and used PAMS for communications between the front end workstations and the back end servers and this proved sufficient to create a fully functional pilot system. (#UI)

The Transvik Workstation

However the pilot system highlighted a number of problems - system management complexity, the transactional integrity of the trades being executed, the LAN distance restrictions, and the design of the user interface. During one demo a petulant trader (weren't they all?) poured a cup of coffee over the mouse because doing a trade now required eight mouseclicks instead of three key strokes as previously.

It became clear that the scope of the project was greater than had been originally envisaged and so Jan Stenbeck brought in a crew of Texas based consultants led by Tom Martinson. They were to take the project forward and transform the pilot system into a robust and manageable production system and soon a lot of the old faces from Goldman Sachs, including Steve Krolak, came on board as contractors to get this done.

(#SOA) Because of his background in telephony systems Tom was focused on real time communications and was unhappy with a number of deficiencies in PAMS, which was still at a relatively early stage of its development. Features for management, resilience and recovery were basic and in addition the underlying messaging mechanism was asynchronous. This meant that to execute a trade a workstation would send a PAMS message to the back end and then wait to receive a PAMS message back indicating the trade's successful execution or its failure. Taking care of course not to wait forever if a failure occurred along the way. Failures could occur in the PAMS infrastructure or the Ethernet network or the back end business logic which included data storage handling. But these different failures were not managed or reported in an integrated fashion and the whole process of executing a trade was certainly not executed as a single ACID transaction. To compound the situation the data being stored on the back end servers was not being held in a fully functional database with proper transaction control.

The decision was taken to move the data into a relational database and the one selected was the Ingres database, a leading RDBMS at the time with a good reputation. What to select as the replacement for PAMS was a far more difficult question and required going out on a limb to find an answer. In the early 1980 s AT&T had developed a product called Tuxedo for their own internal use which could be viewed as 'CICS for Unix'. Its main design aim was, like CICS, to support very high transaction rates in a robust secure fashion but unlike CICS to support them on one or more Unix servers, including AT&T servers, which was far cheaper than using CICS on a mainframe. As an internal product running on AT&T servers Tuxedo was relatively easy for AT&T to support and develop but as there were many different flavours of Unix now running this posed a major challenge. If Tuxedo was to be released as a commercial product it would need to be ported to, and supported

on, platforms such as Unisys, ICL, IBM, HP, Sun, Data General, Bull, etc., which AT&T was not inclined to do. Instead they licensed the Tuxedo source code to the various computer manufacturers so they could do it themselves.

But not all manufacturers, for instance IBM, were interested in taking it on so AT&T also licensed Tuxedo to several independent software companies, such as IMC (Information Management Company) in New Jersey, who saw commercial opportunities in this situation. The successor to a more generalized software consultancy, IMC had come to principally focus on Tuxedo opportunities and they featured the traditional successful start-up troika, in this case Glenn Rose (business), Peter Dessart (vision) and Lee Morrow (technology). Unfortunately they had never ported Tuxedo to run on VAX/VMS but Transvik did a deal with them to do the VMS port so it would be possible to move the Transvik system off PAMS and on to Tuxedo.

The key features that Tuxedo provided were manageability, synchronous communications, transactional control and a full API for interprocess communication on, or between, machines. In its current form Tuxedo has all the modern features you would expect but its key features were already present in those early days. The business logic was written as a set of application services written in C, for example EXECUTE-ORDER, and were linked into server processes running under Tuxedo control. An important point here is that Tuxedo application services could be other things than ACID database transactions, they could, for instance carry out calculations, or translations, and so on. This was one of the ways in which TP Monitors began their evolution into general purpose middleware.

(#PZ) (#SOA) The application services were advertised within the Tuxedo environment in such a way that a Tuxedo client could call a service without having any knowledge of which machine it was running on.

A client's service call using tpcall() from the Tuxedo API looked something like:

**return-code = tpcall("EXECUTE-ORDER", input data buffer, results data buffer, status code);**

This was a synchronous call, technically known as a remote procedure call (RPC), which seemed like calling a subroutine in the same program except that the service being called might be running anywhere within a Tuxedo environment that involved many networked machines.

This took the componentization of programs a giant step forward from only using subroutines and C functions as component parts of a monolithic program. Now a program could make use of program components, known as services, that were provided by completely independent and separate programs with no shared code involved. Incredibly it would be many more years before this model, the SOA (Service Oriented Architecture) model, became widely accepted.

Despite the various challenges it faced the Transvik system finally got implemented and it was during its implementation that I got an IBM PC at home and started to explore what it could and could not do. One of the things it did allow me to do was to subscribe to one of the pioneer dial-up online services Compuserve. On Compuserve I got my first personal email address, which was a meaningless two part number something like 234567, 123987. However there were so few people I knew who were also on Compuserve that my email communications were few and far between.

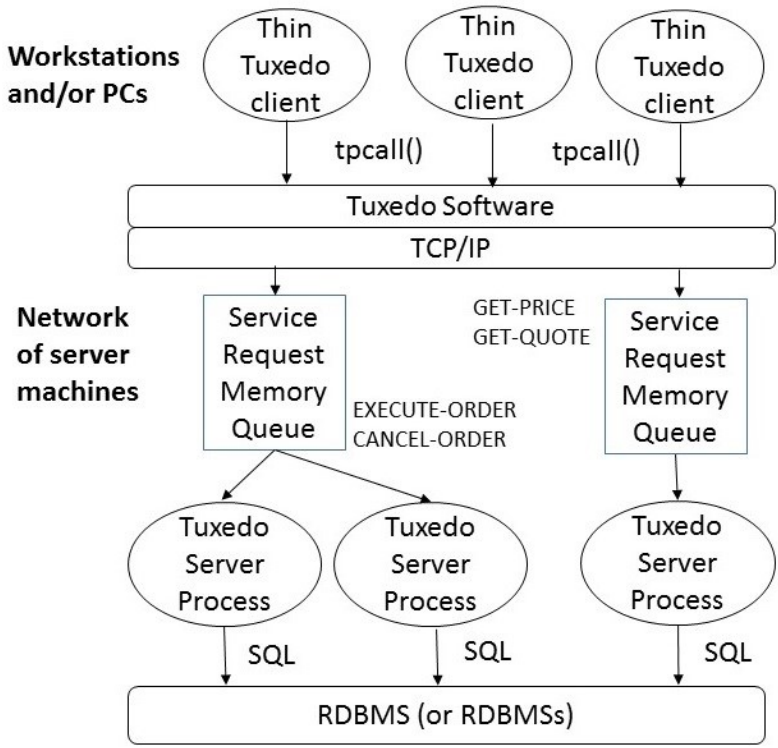
As the Transvik project wound down DNR was already dreaming of a workstation that could just sit there quietly making money without any human intervention but that was another vision ahead of its time and ahead of its funding cycle. We also often had telephony guys wandering around our office from Tom's earlier projects who were dreaming about a new kind of semi-mobile phone system. They were thinking of installing antennae at major urban sites such as rail stations that could support calls on portable phones, though only when they were within the antenna's limited range. It was to turn out differently of course.

## *The Basics Of Tuxedo*

(#PZ) (#SOA) In a Tuxedo application that was distributed across a number of machines all the complexity was taken care of by the Tuxedo infrastructure and its configuration. The Tuxedo API

was relatively simple to use and the programmer needed no knowledge of the shared data sections, semaphores, memory queues and networking software that underlay it.

Under the covers a Tuxedo RPC call made by a client process involved a message containing the input data being queued to a server process advertising the application service for the requested business logic. The server process could be running on any of the machines that were part the Tuxedo configuration and the message would be automatically delivered to the correct queue on the correct machine by the Tuxedo infrastructure. It would then be dequeued and processed by the service, which probably involved SQL access to a relational database, and a message would be sent back to the client containing the results plus a status code reflecting any problems encountered. The message formats could be defined in a number of ways including Tuxedo's own FML (Field Manipulation Language) which can be seen as a precursor of XML and allowed great flexibility in defining message formats.



Tuxedo clients would each be running on their own workstations, or one day PCs, and presenting a GUI interface to the users with the server processes running on one or more networked server machines (the word ‘server’ on its own being ambiguous in this discussion) with the database probably hosted on another separate machine. The outbound path of a service call is summarized in the somewhat oversimplified diagram.

Because the business logic resided in the server processes it was only the server processes that needed open connections to the database rather than every single client. This key feature of the 3-tier architecture, sometimes known as database connection multiplexing, was the basis for Tuxedo’s scalability that went way beyond that of the 2-tier model.

By taking care to write the Tuxedo services in standard C and only accessing the database using standard SQL a Tuxedo application could be made highly independent of which relational database was being used, which of course the database vendors hated. Tuxedo exhibited a complete SOA a good two decades before SOA was finally recognized as the correct foundation architecture for modern distributed applications.

# 1991 Chicago – Electric Blues

In mid-1991 a few months after the Transvik project wound down I got a contract at A.C.Nielsen the ratings agency in Deerfield just outside Chicago. Deerfield was a Chicago suburb known to some of its residents quite justifiably as Drearfield and after a few months of living at the Deerfield Residence Inn the Drearfield effect kicked in big time. So I moved into Chicago within sight of Lake Michigan and started commuting out to Nielsen each day. Being in Chicago was curiously like being back home in Hull and Liverpool in northern England when I was younger. Chicago is often characterized as 'the city that works' and it really did have the feel of a north of England working city. It was much more down to earth than New York. To add to its many attractions electric blues music provided the background to Chicago life. It was to be heard on the radio and in the many blues clubs like Buddy Guy's Legends, the club owned by Muddy Waters' old guitarist. When asked by the Chicago Tribune newspaper who was his favourite guitarist Buddy Guy replied that it was Eric Clapton. That Buddy-Eric connection is a perfect expression of the powerful bond that exists between Chicago and England based on the electric blues and the power of that bond made me feel like I was arriving home.

## *A Two Tier/Three Tier Hybrid At Nielsen*

Nielsen was best known for collecting and publishing data on the viewing figures for TV shows but it also did the same for the volumes of consumer product sales. In those days the methods for collecting and analyzing the data were quite primitive and involved a good deal of manual intervention and Nielsen wished to use IT to automate at least some of the data collection process. This was a very very early precursor of 21st century Big Data and Data Analytics. As the first step of the modernization project they had undertaken a large data modeling exercise and created a lot of documented E-R diagrams plus an internal organization to manage the data model and the DB2 database created from it. Although data modelling was not really my thing it turned out that an enhancement to E-R diagrams was the creation of entity-life-cycle (ELR) diagrams which, broadly speaking, described the 'life' of a specific data item,

say an account number, from its birth (creation) to its death (deletion). Happily for me the ELR diagrams used the same structured programming notation that had been developed by Michael Jackson and Alan Cohen so I was qualified for at least that.

As things turned out I did very little data modelling because what seemed to be missing was any definition of the business processes that needed to be carried out with the data and that was far more my kind of thing. The IC project, which I co-led, was all about interpreting the bar codes scanned at supermarket check outs when customers paid their bills. Nielsen processed vast amounts of this data with the data records typically consisting of the sales date, a bar code, the sales amount and maybe a text description of the product. But although at the time the use of bar codes was somewhat standardized it was nowhere near standardized enough to automate the collection of barcode data, the same bar code might be used differently in different sales locations. Therefore if necessary the sales transaction records had to be manually reviewed with the aim of identifying the correct product for each record. Our project was to create a PC based system to assist the reviewers in doing this.

Our users, the data reviewers, worked in Fond du Lac, some distance away in Wisconsin, the land of the Cheeseheads, and it needed serious navigation efforts by the IC crew to get us to this faraway destination for the occasional meeting or demo.

The IC crew set off for Fond Du Lac

The implementation environment was a hybrid of a 2-tier and 3-tier architecture. A front end GUI application for PCs was being written in SQLWindows which was one of several development tools available at the time for creating 2-tier GUI applications that accessed relational databases, in our case SQLserver, hosted on a Windows server. These tools made heavy use of the fact that a relational database table could easily be loaded directly into a visual control very similar to a spreadsheet by using simple SQL.

(#PZ) (#SOA) The IC business logic made use of both CICS transactions and SQLserver 'stored procedures'. Stored procedures were a workaround developed by the database vendors to allow some shared business logic to be stored and executed on the database server. Stored procedures were in fact executed by the RDBMS and thus sharable by all the clients in a 2-tier application. To do so stored procedures combined standard SQL with non-standard SQL extensions that supported, for instance, conditional statements to test for true or false and thus allow a basic level of logic to be included in SQL. For SQLserver this hybrid language was called TSQL whereas, for instance, in Oracle it was called PL/SQL with various other names being adopted by the other RDBMS vendors. Needless to say all these SQL hybrids were different and they were all proprietary as they constituted another attempt by the database manufacturers to enforce vendor lock-in. The stored procedures were held in the database and executed by the Relational Database Management System, not the obvious way to implement business logic and certainly not designed for complex business logic. For the database vendors the key was that by writing your shared business logic in their proprietary SQL extension you completely locked yourself in to their RDBMS.

The IC SQLserver database held mostly static reference data and was refreshed by a regular upload from a DB2 database resident on an IBM mainframe. The DB2 database had been created using the large data model defined at the start of the modernization project and was also accessible using CICS transactions written in COBOL which could contain more complex business logic. The final data component of the IC project was the sales transaction data and this was transferred to the Deerfield mainframe from a mainframe in Green Bay Wisconsin overnight. It was a somewhat complicated hybrid architecture but not untypical at the time and it was certainly better architected and engineered

than many other similar systems and because of that in the end it worked. Due to the hybrid nature of the implementation environment a variety of different skills were required and the IC project team was another example of a very mixed bag of characters including in this case a significant Russian ex-pat component.

The IC project ran relatively smoothly and after about 18 months it completed successfully which was duly celebrated in a haze of B52s, the team's favourite cocktail.

## *There's A Storm Coming In*

In Chicago I was living on the 43rd floor of an apartment building close to Lake Michigan and I remember the first time of many that I saw threatening stormclouds rolling in across Lake Michigan that would soon engulf the building in rain, thunder and lightning. I should have been prepared because in 1984 the cyborg movie 'Terminator' had famously concluded by a character stating ominously that "there's a storm coming in". By 1993 there were not only threatening stormclouds over Lake Michigan they were also gathering over the software applications that were being increasingly distributed across a mixture of different server platforms.

(#SOA) The first stormcloud was the unmanageable complexity implied by large networks of mixed platform servers. I saw at Nielsen firsthand the integration effort involved in combining PCs with servers and mainframes, LANs with WANs, 2-tier database stored procedures with 3-tier CICS transactions, and this was in an environment with only a small number of end-users. In a large perhaps global network these issues could quickly become overwhelming. As Unix, VMS and Windows servers became ever more widely used to do serious processing it was obvious to some people that distributed network architectures needed to be rationalised and standardized.

Networks needed to be able to include a mixture of different servers possibly all running different versions of Unix and different RDBMSs, and to include mainframes in the mix too, all in a transparent manner. It was another example of vendor-specific solutions versus standards-based solutions and in 1984 the OSI (Open Systems Interconnect) project at ISO (International

Standards Organisation), had published a seven layer model for network software, usually referred to as the OSI Model.

**The OSI Model's Layers**

7. Application Layer
6. Presentation Layer
5. Session Layer
4. Transport Layer
3. Network Layer
2. Data Link Layer
1. Hardware Layer.

The idea behind the OSI model was that networking software on all platforms would allow it to carry out network communication. On the sending machine the software in each layer would use software from the next layer down via its API until the base of the stack was reached at which point the actual communication across the network would take place. Once the communication arrived on the receiving machine it would be transferred up network software that followed the same model to the receiving process. Although this model never really took off in its entirety it was very influential in network design and began to lead to the development of layered network architectures.

(#SOA) Another major stormcloud approaching was network scalability. The two tier LAN based architecture was inherently unscalable because each copy of the business application running on its separate PC needed to have its own open connection to the database. Although some of the RDBMS vendors claimed their products could handle 2,000 open connections this was a claim greeted with some skepticism and even if true it certainly represented the upper limit. But in the 3-tier architecture with the application's business logic running in server processes on server machines each open database connection could be shared by many clients thus drastically reducing the number of database connections required. But this was before the Internet and World Wide Web came along and their need for huge global networks would settle the argument once and for all. So the argument raged on between proponents of two tier client server (mostly database

vendors) and proponents of three tier architectures (mostly infrastructure software companies).

(#SOA) Mixing different platforms handling transactions across a network was yet another big stormcloud that was gathering. The issue of handling distributed transactions, for instance between my bank account in a London bank's database and yours in a New York bank's database, had been addressed by the invention of two phase commit (2PC). This was a protocol used by RDBMS instances in a networked environment whereby the different database instances on different network nodes would first each 'vote' on whether to commit, or abort, a transaction on its local database (phase 1) before it was actually committed by all of them (phase 2). Each database vendor, predictably, implemented 2PC in a different way so a problem arose when my bank account was in an Oracle database but yours was in a DB2 database. 2PC was implemented differently by Oracle and IBM so a single transaction simply could not be defined. To solve this problem the vendor independent XA protocol was defined by X/Open whereby if an RDBMS supported XA then it could participate in a transaction with any other RDBMS that also supported XA. Eventually all the major database vendors started to provide XA protocol support as a paid for extra feature of their RDBMS and so finally a single transaction could include, say, both an Oracle and a DB2 database. This of course required the presence of an XA transaction coordinator as part of the software infrastructure. Other computing resources could also comply with the XA protocol and participate in XA transactions, for instance IBM's MQSeries messaging software is a famous example and XA compliant print spoolers have also been created.

So stormclouds were definitely gathering on the horizon and before too long they would be coming in.

# 1993 Back In NYC – Pushing 3-Tier

In late 1993 when the Nielsen project wound down in Chicago I had to decide whether to push on further west, to travel across the Pacific Ocean to Hawaii, or to make Chicago the westernmost destination of the odyssey. Perhaps I was getting too far from home or perhaps I couldn't hear the Sirens calling from Hawaii clearly enough but whatever the reason I decided to turn back and return to New York City.

Back in NYC I soon met up again with Steve Krolak and in 1994 when freelance contracts were very thin on the ground we both ended up becoming employees at IMC, the Tuxedo consultancy we'd first encountered at Transvik. Tuxedo was making slow but steady progress in the market place in the US and IMC were beginning to need consultants who could provide professional services to the sites who had purchased Tuxedo. IMC were located in New Jersey, close to Tuxedo's birthplace, so commuting through the Holland Tunnel became my daily routine again like when I was at Bell Labs. This time though I was driving Glenn Rose's old Saab which I got to borrow as part of my IMC employment package.

## *Selling 3-tier*

The Tuxedo product itself was now owned by Novell who had acquired it as part of their purchase of Unix System Laboratories (Bell Labs) from AT&T in 1993. It was widely rumoured that Novell had never heard of Tuxedo until they found themselves the owners of it, however they continued to support it on Unix, now called Novell Unix. IMC was now doing Tuxedo implementation and customer support on many of the other platforms including IBM, Data General, Unisys, Sequent, Sun, HP, even Windows NT which was released in 1993. On the west coast of the US a company called Independence Technologies was also a Tuxedo consultancy and somewhat competitive to IMC. Tuxedo was now at release 4.2 and was multiplatform, highly scalable and also an XA transaction coordinator. In addition to all of this its performance was excellent as shown by the fact that even Oracle, a vociferous supporter of 2-tier, always used it when running the Transaction Processing

Council's industry standard TPC-C benchmark for online transaction processing (OLTP), as they still do today. The class of software that Tuxedo belonged to was still normally referred to as TP Monitors but in OLTP circles there was a growing awareness by some people that an evolution towards more general purpose 'middleware' was taking place.

Because of its origins at Bell Labs Tuxedo was quite widely used by the US telcos (telecommunication companies spun off when Bell Telephone (later AT&T) was broken up into the 'Baby Bells' in 1984) but it had not really penetrated other market sectors such as finance, logistics, health, retail, etc. This was the challenge facing the IMC sales guys, who were of course in practice all guys. Although larger distributed systems were starting to be implemented there were not yet enough of them to be a compelling argument for 3-tier. In addition although implementing a new 3-tier application was not difficult unfortunately there were very few programmers who knew how to do it or system designers who could design 3-tier applications properly. When it came to transforming 2-tier applications into 3-tier applications the situation was even worse in terms of available skills.

The main uptake for Tuxedo at the time was almost always in companies that had a talented software architect who understood the limitations of 2-tier client server and such individuals were few and far between. In fact I'm not sure the term 'software architect' even existed at the time. If any similar person existed in a company they were sometimes called 'software fellows' after the style of university research fellowships. Far worse was if they were hiding under a more traditional title such as system designer, systems analyst etc. and any architectural concerns they might have were being ignored. In fact the 2-tier client server crowd tended to dismiss the whole concept of a software architecture supported by middleware as unnecessary, their view was - just put it all in the database.

It was a talented software architect, who I believe was indeed known as a software fellow, who led to the biggest win for Tuxedo at IMC in those days, and he worked for Federal Express. He saw that 2-tier was far too limited to put in place the kind of applications they were planning and some form of middleware was required. Somewhat surprisingly IBM were very supportive to IMC

as they had just released the SP2 machine, a number of single board computers in the same box linked by a high speed backplane. But they really had no appropriate middleware to offer for developing applications distributed across the various computers in the box. Tuxedo was a perfect fit and Steve Krolak was sent down to Memphis to do a proof of concept project which was a great success.

Meanwhile I was going out to several sites to do what nowadays would be called pre-sales work. I would demo basic Tuxedo functionality and try to show that creating 3-tier applications was not rocket science. On most sites my reception varied from lukewarm to openly hostile as the new style of programming was correctly seen as a threat to PC based 'thick client' 2-tier programming. It would still be a few years before the tide really started to turn. While doing these assignments I became aware through hands on experience that Sequent was a computer vendor making a real effort to put Tuxedo into Dynix environments, it's version of Unix. They had some very nice hardware offerings and seemed to be a computer manufacturer that was going places and I stored that fact away for future reference.

There were two things I learned about Tuxedo at IMC that were to stay with me and be very useful. First, as one of the salesmen said to me, "The thing is, it works!", it seems an obvious claim but when so many other distributed software vendors could not confidently say that their product worked well it was a striking claim. Second, the IMC motto was "The answer is always yes!", whatever you wanted, Tuxedo could do it, somehow.

It was around this time that I got my first cell phone as telephony engineers had finally solved the problems I'd heard about in the office at Transvik through cellular technology. It would prove to be a huge technological step forward but the full effects of it would not be felt until the 21st century.

## *The Distributed Application Programming Style*

(#SOA) Tuxedo of course was not the only game in town for creating distributed 3-tier applications for mainly Unix servers. These products were usually still referred to as TP Monitors, however their evolution into 'middleware' was underway. Other

similar products in the market place included NCR's Top End, IBM's Encina, DEC's ACMS and ICL's TPMS, however none of them covered the number of platforms that Tuxedo did. IBM's approach to platform coverage was to make Encina dependent on DCE (Distributed Computing Environment) which was an attempt by the OSF (Open Software Foundation) to set down standards for distributed applications and application services such as security, distributed directories and so on. The idea was that each computer vendor would add DCE to its operating system and then all systems running that OS could participate in a distributed and mixed environment with any other DCE supporting computers. It never really took off as once again individual vendors did not want to support vendor independent standards. What helped Tuxedo was that the one standard it depended on, which was rapidly becoming ubiquitous and in reality vendor independent. This was TCP/IP the Transport Layer (layer 4) in the OSI model and everybody, even Microsoft, eventually started to support it. Tuxedo rested upon TCP/IP and as long as TCP/IP was supported on a platform Tuxedo could be ported to it.

All the products mentioned above supported 3-tier applications and they all provided a management capability for managing distributed applications spread across a network.

(#SOA) The defining features of a 3-tier application are:

**It uses a thin client not a thick client**: In 2-tier applications all the business logic is implemented in the PC 'thick' client which must then be resident on every client PC whereas in 3-tier the business logic is on the server side written in a real standard programming language such as C, COBOL, etc. and not some vendor's proprietary SQL extension. Business logic is presented in the form of shared services (sometimes called transactions, which may not always be accurate technically) that are called by the 'thin' client usually by using some form of RPC (remote procedure call). This gives the application resiliency, load balancing, scalability, failover and it allows business logic upgrades to take place even while the application is live.

**It runs in a full SOA (Service Oriented Architecture) environment**: SOA is nothing new, TP monitors implement SOAs

in which a client can call a service without knowing where it is located or how it is implemented.

**It takes advantage of database connection multiplexing:** Only the shared services require open database connections which can be shared by all the clients so many more clients can be supported than in 2-tier applications and far higher transaction rates can be achieved.

**It uses the database's 2PC and/or XA:** the clients and services can use full ACID transaction control across distributed databases and, if XA is supported, across mixed vendor's databases.

(#PZ) The SOA model further reinforced the idea of componentized applications except that now the components weren't all linked together to form a single monolithic executable image but largely consisted of calls to services that were running in other independent executable images distributed across the network. The services model would one day become ubiquitous.

## *You Can't Go Home Again*

Meanwhile back in England in 1994 my father turned 89 and I felt it was time go home and be there when he reached 90. After almost fourteen years in the US I would finally return home to England but like the journey home of Odysseus himself it was not that easy. For one thing DNR would regularly insist on quoting to me Thomas Wolf's ominous line "You can't go home again" and I was also a bit nervous about telling the guys at IMC I was going back to the UK as I really hadn't been with them that long. But I needn't have worried, this was America. When I told Peter Dessart I was returning to England without the slightest hesitation he said I must start a Tuxedo business in the UK and check out the unexplored leads the IMC salesmen knew of in Europe. God bless America! It was while I was gathering the UK and European leads from the IMC sales guys that I first heard about a very large Tuxedo project at the UK's Employment Service. It was a project that would turn out to play a hugely influential role in my future.

# 1994 London – Bringing It All Back Home

In October 1994 on what would turn out to be the eve of the Internet explosion I travelled back across the Atlantic Ocean to the UK. The Cool Britannia and Britpop phenomena were kicking off there with bands such as Blur, Pulp and Oasis who included backward references to the 1960s in their music. In a Rolling Stone reader's poll for the 10 best Britpop songs 'Don't Look Back in Anger' by Oasis was number 2 and was described as follows.

*Bob Dylan's Don't Look Back movie hit theaters in 1967, and 12 years later David Bowie released his song "Look Back in Anger." The titles were mashed up on the first Oasis song to feature Noel Gallagher, who considers it one of his finest compositions, on lead vocals. He has since described the tune as a cross between the Beatles and Bob Dylan.*

In similar fashion Bob Dylan's album Bringing It All Back Home hit record shops in 1965 and in this case it was 29 years later that its title was mashed up with the contents of the suitcase I brought back from America.


The contents of the suitcase I brought back included one set of Tuxedo manuals, the IMC email and telephone directories and the

details of some possible leads. It didn't seem like a lot of resources and I was nervous about the prospects. But this mashup was going to work just as well as the 'Don't Look Back In Anger' mashup did. I really was Bringing It All Back Home. All of it.

## *GA Systems*

In London I linked up again with Alan Cohen and in November we formed GA Systems Limited as a Tuxedo consultancy. Having just returned from the high tempo 'can do' business world of the USA trying to do business in the 'sorry you can't do' UK was a real let down and quite dispiriting. However there was nothing to do but to get over it and get on with it. An even bigger challenge though was that the UK was still very much locked into the 2-tier client server application world with its Cyclops like tunnel vision view of software architecture. Very few people were looking at 3-tier application development.

As is usually the case in IT many careers and billions of pounds of software and hardware were invested in one way of doing things, 2-tier, so to change to a new paradigm like 3-tier was going to be a long drawn out process. GA Systems was firmly in the 3-tier camp and back then that was a fairly lonely place to be. Also I was about to turn 50 and Alan already was 50, we were also both going grey, so the obsession with 'cool' young tech-dudes, who were usually 2-tier disciples, did not bode well.

One of the first UK contacts I followed up on was ICL who had a Tuxedo source code license and when I met with the remnants of their Tuxedo team they were friendly but told me 'you're 2 years too late'. They had tried pushing Tuxedo but with very little success and had largely abandoned it. (#SOA) Over the coming months I would hear the same story several times and it seemed to me that in the UK the TP Monitor features of Tuxedo had been over-emphasized and its expanding use as a general SOA middleware platform under-emphasized. It was the latter that was going to provide the way forward.

To try and start building some momentum for Tuxedo and 3-tier we organized a seminar on January 27th 1995 called 'Second Generation Client/Server' and managed to attract 30 attendees, though we knew half of them personally.

We also visited other computer vendors and found Unisys to be more positive about Tuxedo than ICL. But it was when we made contact with Sequent Computers, the vendor I had come in contact with in the US, that things looked more promising. Sequent in the UK had adopted a far more American way of doing business and their sales force were still pushing Tuxedo and it was being used on some small projects, so a little consulting work came our way from them. In addition the London office of the US development tool maker JYACC who had tools for creating GUI Tuxedo clients were supportive to GA Systems.

Meanwhile in the US the IMC sales guys had landed a couple of Tuxedo proof of concept projects in Europe, one in Athens for a local Greek IT consultancy and one in Madrid for HP. In both places local consultants had failed to get Tuxedo running properly and so its use was hanging in the balance. We were contracted to redo the proof of concept projects and get Tuxedo up and running successfully and as a result I spent some time in both Athens and Madrid and through these projects GA developed the first version of a Tuxedo load test tool. These two assignments taught me a crucial lesson the hard way. Whereas in the USA whichever state you were in you could always use an RJ11 jack plug to connect your modem to the phone network and dial up a remote system, every European country used different jack plugs for modem connectivity. To consult in Europe meant carrying around something like a dozen different modular telephone jack plugs. Incredible.

Meanwhile Alan was creating our three one day Tuxedo courses with which we hoped to increase Tuxedo skills and Tuxedo awareness in the UK. We started to run them and began to attract modest class sizes for them and we also hosted another public seminar to keep the Tuxedo ball rolling.

## *LMS – The Big One*

(#SOA) I'd first heard rumours in the US of a big, the biggest ever, Tuxedo project, the Labour Market System (LMS) that was being planned by the UK government's Employment Service. It was to support the new Job Seeker's Allowance which the government had introduced with a big fanfare and the project was rumored to involve 25,000 PCs in 1200 local offices each with a local server which would access a centralised data centre. Happily for the UK

taxpayers the talented overall system architect, Phil Mullis, realized that the only viable solution was a 3-tier architecture and that the best, robust and proven middleware product to support that was Tuxedo.


GA's first encounter with the LMS project came in 1995 when competitive benchmarks were run for the contract to provide the 1200 servers for the local offices and we were contacted by IBM to carry out this benchmark for them. It was specified that Tuxedo 5.0 which was not yet officially released was to be used and that meant getting it from Novell themselves since they had yet to release the 5.0 source code to the source code licensees like IMC. The big new feature in Tuxedo 5.0 was the ability to support many interlinked Tuxedo domains, basically the manageable unit of Tuxedo, instead of just one monolithic domain. Unfortunately this capability had not been thoroughly tested and the documentation recommended using a product feature that it was later admitted contained bugs and caused errors at high transaction rates. With less than stellar benchmark results all around IBM eventually lost the bid on price.

It was thought that the computer hardware for the LMS data centre had already been selected and that it was going to be ICL. This was not surprising since ICL was a UK government supported computer vendor who were usually the automatic and unopposed choice on government contracts. However it seemed that ICL was not getting the performance required by the LMS and so the Employment Service decided to run a benchmark competition for the data centre hardware. This was an unusual development in a government contract but Computer Weekly was now regularly reporting massive and expensive development disasters on various large scale government IT projects and this may have led to some rethinking. The benchmark was to involve ICL and Sequent Computers and to use specimen Tuxedo service calls provided by the LMS project team. As GA Systems was already known to Sequent we were called upon to provide the Tuxedo support for their benchmark team in Weybridge. I hired a car for 3 days to do my daily commute but it was to be 6 weeks later before I would actually

return it to the rental company. Sequent provided a Dynix expert, a Sequent hardware expert and an Ingres (the RDBMS being used) expert and when I was first introduced to them the Ingres guy said to me that it would be Tuxedo that was the bottleneck in the benchmark. This was a standard objection from the 2-tier crowd, that putting Tuxedo in front of the database would slow it down, but he, and they, were completely wrong about that.

Over the next 6 fairly hellish weeks the benchmark team built an optimised benchmark system that when the benchmark was carried out beat the ICL team's benchmark system quite handsomely. As a result Sequent won the contract to provide the hardware and software in the LMS data center which was located in Sheffield. But wait, the insanity of the UK government's famously restrictive procurement regulations was about to intervene. Sequent Computers weren't on the government's authorised supplier's list! What to do? Eventually the Employment Service bought the Sequent computer equipment through HMSO, Her Majesty's Stationery Office. You couldn't make it up.

At the Sequent team's victory celebration I received a snazzy Sequent golf umbrella for my contribution but more importantly Sequent also contracted with GA Systems to do the load testing in support of the LMS development team in Sheffield. The load testing was to be done using a dedicated Sequent server, known as the injector, which would run our load test tool and fire a high volume of transactions into the LMS Tuxedo application running on the data centre computers. Any fragility in the Tuxedo services being developed would be rapidly revealed and could be corrected well before the system was rolled out across 1200 sites. I started spending a lot of time in Sheffield where my personal workload quickly became too heavy for me to support alone and GA Systems was asked to provide another resource on site.

Alan began trawling the contract agencies in London to try and find somebody, anybody, with Tuxedo experience and amazingly he found the perfect support programmer to assist me in Keith Thomas. Keith had worked at Bull Information Systems in their Tuxedo team and had done a lot of Tuxedo work both on Tuxedo internals and on Tuxedo applications. Up until this moment I had not been aware that Bull had made such a significant investment in Tuxedo but like ICL they had eventually given up on it and

disbanded their Tuxedo team. It was a fantastic stroke of luck to have met Keith at this exact moment as he was a very talented programmer. It was not long before Keith became the 3rd partner in GA Systems so we now had the threesome that I believed was necessary for a successful start-up company. In addition Keith was in his late 20s, so we no longer all had grey hair.

While we worked on the LMS project GA Systems also contracted to develop an SNMP agent for Tuxedo and we brought in a couple of contractors to help with that. We were also hearing from the guys at IMC in the US that a new well financed 1995 start up in California, BEA Systems, might be looking at Tuxedo as a possible foundation for their ambitious development plans. Warburg Pincus were their financial backers so this was serious stuff. We started to cautiously think that perhaps Tuxedo's moment, and the 3-tier model's, had finally arrived.

This was also the time when the world wide web began to make an impact and its take off event is usually identified as the release of version 1.0 of the Mosaic web browser in September 1993. Mosaic has been labelled the killer app of the Internet in the same way Lotus 123 was the killer app of the PC. Originally Mosaic ran on Unix but was later ported to the Mac and Windows platforms and the rest is history. It was the first graphical browser which displayed text and images together. At first web sites tended to be technical or academic but increasingly businesses started to adopt them though at the time they were just information sources, like online publicity brochures, with no transactional capabilities.

Meanwhile in early 1996 we handed over our load test tool specifically configured for the LMS system to the Sheffield project team for them to complete the final stages of load testing.

There was an increasing awareness developing that client-server was having major scalability problems and of course this meant 2-tier client-server. There were numbers of government IT disasters being reported and an infamous private sector IT disaster had been National Westminster bank's incomprehensible decision to base their branch network on an early release of Windows NT. Eventually IBM was called in to rescue the project which they did, it was said at the time, by installing a proper network of IBM's AIX

servers in the branches and hanging the Windows NT servers off them.

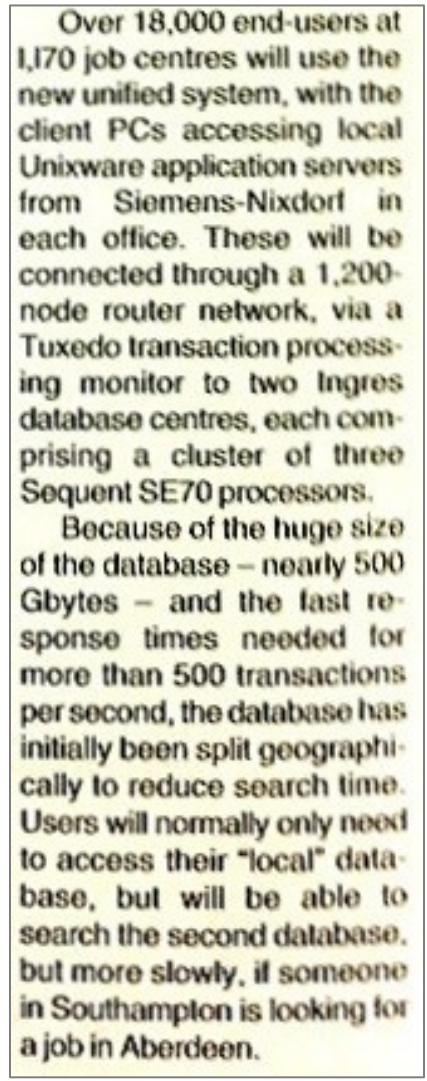
With all these problems being reported in June 1996 Computer Weekly had a look at the problem in an article titled ‘Thinking Big’.


Thinking Big – Computer Weekly

The article made a big play of a 2-tier Oracle application at Europcar that supported 2,500 users, though it was admitted this was after a very shaky start and repeated system crashes, and no doubt some heavy technical support from Oracle. Nowhere was it even suggested that 2-tier could go beyond this number of users. This was contrasted with using a TP Monitor like Tuxedo, which was characterized as a ‘mainframe solution’, and it was erroneously stated that it was difficult to use because it demanded that the application services were written in low level languages. A somewhat eccentric view of COBOL and C. But this was how deep the bias towards 2-tier had become and how blinkered some technical journalists were. However the Europcar project was the high water mark for 2-tier client-server.

Meanwhile the LMS project completed on time and went live on July 1st 1996, a few months after GA Systems had been acquired by BEA Systems.

It was perhaps the only public sector project to come in on time and run without any major problems, particularly without scaling problems. Computer Weekly were so amazed that a project of such scale had succeeded (over 18,000 users supported by only 200 open database connections) they ran a 2 page story titled, revealingly, ‘Against All Odds’.



Over 18,000 end-users at 1,170 job centres will use the new unified system, with the client PCs accessing local Unixware application servers from Siemens-Nixdorf in each office. These will be connected through a 1,200-node router network, via a Tuxedo transaction processing monitor to two Ingres database centres, each comprising a cluster of three Sequent SE70 processors.

Because of the huge size of the database – nearly 500 Gbytes – and the fast response times needed for more than 500 transactions per second, the database has initially been split geographically to reduce search time. Users will normally only need to access their "local" database, but will be able to search the second database, but more slowly, if someone in Southampton is looking for a job in Aberdeen.

From 'Against All Odds' – Computer Weekly

Although the story several times mentioned Tuxedo and how crucial it was in the success of LMS it was still being referred to dismissively as a TP Monitor. Tuxedo was mentioned in a way that suggested it was a prehistoric monster, like the Charybdis and Scylla faced by Odysseus, that had somehow escaped from the

ancient world where mainframes lived. The concept of middleware was still a long way from being understood and accepted by technical journalists or by IT developers in the UK.

## *BEA Systems*

BEA Systems, the Californian start-up we had been hearing about, had been founded to focus on the middleware infrastructure software that was going to be needed to support the gathering storm of large scale, mixed platform, distributed applications. BEA stood for **B**ill Coleman (vision), **E**d Scott (business) and **A**lfred Chuang (technology) and in the UK the confusion between BEA Systems and the former BEA (British European Airways) would haunt the company for several years. Bill, Ed and Alfred were not three hot young techie dudes but were three mature ex-Sun Microsystems business-savvy technologists and a standard joke was that BEA was not a start-up born in a garage but one born in a condominium (UK = an upmarket private apartment). To GA Systems this seemed like a good sign.

Although the Internet was already in existence its full potential had not become apparent and the most likely platforms for large distributed transactional applications were thought to be networks of Unix servers which would very likely be accessing data and services on IBM mainframes.

BEA quickly identified Tuxedo as the only available middleware that could be used for this purpose and as their business plan was to grow very quickly by acquisition they started acquiring all the Tuxedo focused companies they could find around the world. Initially this involved IMC in the US, Independent Technologies in the US, GA Systems in the UK (see Appendix C), USL in Paris, followed by companies in Finland, South Africa and other countries. They also acquired VISystems a US company that had developed Unix to IBM CICS connectivity products. The other part of the BEA business plan was to negotiate aggressively with Novell to get complete control of the Tuxedo product and source code. This was successfully achieved and the whole Tuxedo engineering team would eventually join BEA.

As part of the GA Systems acquisition I went to California in early 1996 to meet people from the embryonic BEA organization

and no doubt like many people in the same situation I had been briefed by my partners to try to get as much cash as possible for our company. But that wasn't the way BEA were going to work, the bulk of the offer was in the form of stock options which happily I was familiar with from my days in the US. It turned out the cash amount being offered in my case would just about settle the credit card bills I'd amassed in helping to get GA Systems up and running. When I was introduced to Ed Scott he told me that the aim was to build BEA to a $1billion a year software company within 4 years so the stock options represented excellent value even though BEA was not a listed company at the time. Bill Coleman added his view that this would involve a lot of hard work plus the ability to improvise when necessary in order to achieve that aim, but in four years time 'people like you' won't want to be part of BEA. By then it would be established as a large corporate enterprise with all kinds of rules and endless forms to fill in and a hierarchical management structure, something I was not psychologically equipped to deal with. But by then, he said, your options will have vested and you can get out.

I was somewhat sceptical and would certainly have liked more cash but GA Systems were already totally committed believers in Tuxedo and the BEA principals appeared to be business savvy people with seriously ambitious plans for the product. Nothing like this had ever been proposed in the world of Tuxedo before so this could finally be the moment. There was really no choice, we just had to commit to them.

In fact they were to be proved completely right, their predictions of how BEA would go from start-up company to international corporation would all come to spectacular fruition over the next 4 years as predicted. The bible for carrying out this ambitious plan for BEA's founders was 'Crossing the Chasm' by Geoffrey Moore and if you do a search on the book's title you will see just how influential it has been in many companies. In essence the book says that for technology companies it is vital that they have proper sales, distribution and support infrastructure in place ***before*** product sales take off exponentially. If not then the company will fail at just the moment it ought to succeed as it will be unable to process the rapidly expanding volume of orders.

To start putting the UK part of that infrastructure into place the BEA office in London, actually a Regus rented office in Harrow, opened with just 6 people in March 1996. There were the three former GA Systems partners acting as Tuxedo consultants plus two salesmen, both completely ignorant of Tuxedo, and a sales manager with some knowledge of it.

In the UK, and elsewhere in the world, BEA began to grow and in the UK and in late 1996 we moved out to High Wycombe to get more office space. Tuxedo sales in the UK were hard work but they were starting to happen as the need for large mixed platform distributed systems became necessary for more and more industries. However a persistent problem in sales campaigns was that the Tuxedo product, now being clearly marketed under the banner of 'middleware', a class of SOA infrastructure software that the 2-tier crowd said was completely unnecessary, was quite a hard and technical sell. Arguing the case with a potential customer's systems people, who were almost certainly all 2-tier proponents, needed significant technical knowledge. So quite early on I was moved over from consulting to a pre-sales role to support the sales guys and help them get over the problem. After a career as a consultant it was a whole new world for me mixing with all those 'trust me I'm a salesman' guys but over time a significant BEA pre-sales team was recruited. Meanwhile BEA continued making acquisitions in particular Top End, NCR's TP Monitor which was perhaps the only meaningful competitor to Tuxedo. It was acquired from NCR and proceeded to be merged into Tuxedo.

Tuxedo development continued at a rapid pace and as application integration technology was always in the news at the time the integration possibilities of Tuxedo were now highlighted in sales campaigns. The SOA nature of Tuxedo meant that services running in Tuxedo server processes could be created that in turn accessed services in 'foreign' back end systems, such as IBM CICS, via BEA Connect gateway processes. These foreign back end services could then be exposed within the Tuxedo environment as just more Tuxedo services. New features were also being added to Tuxedo such as Publish and Subscribe communications, the technique that would become the foundation for social media platforms.

Object oriented programming was now an accepted technique using languages such as C++ to create and use memory based objects all resident within a single program executable and Tuxedo itself now allowed services to be written in C++. But the notion of distributed objects was also getting attention which would allow a program to make a remote object method call to an object resident in a process somewhere else in the network. A number of companies, such as Iona Technologies in Dublin, had produced basic ORBs (Object Request Brokers) that included this capability and followed the CORBA standards from the OMG (Object Management Group). BEA decided to give Tuxedo a CORBA 'personality' which it developed under the code name Iceberg and eventually released under the product name M3. To facilitate this development BEA bought DEC's ORB ObjectBroker, along with DEC MessageQ, the renamed PAMS which I'd first met at Transvik. The well tested infrastructure services provided by Tuxedo such as load balancing, resilience, transaction control, manageability and platform independence were, if present at all, extremely rudimentary in other ORBs and were expected to give M3 a huge advantage.

While this was happening the Web was becoming more and more widespread, in a form sometimes referred to as Web 1.0, mostly as a method of distributing business information via web servers such as IIS and Apache. By now all major companies had web sites, usually functioning rather like an online company brochure. A flourishing business in reselling key domain names that companies had failed to register themselves had also sprung up and in fact BEA could not initially acquire the domain name bea.com and so in the early years operated under the domain name beasys.com.

Some transactional web actions were being implemented over the Internet using clumsy workarounds to pass transaction requests through the web server and back to traditional back end transaction processing systems but they were far from satisfactory. As some of these workarounds became more formalized the concept of an **application server** to which a web server could hand off requests to execute business logic appeared, with several competing products in the market.

The Java programming language was also becoming very popular in web environments and was fully realized in 1995. Somewhat ironically Java implemented the same portability

features described by Peter Brown in 1978, 20 years earlier. Java compilers translate the Java source code into a portable machine independent intermediate code (called bytecode) which is then executed by machine dependent interpreters known as JVMs (Java Virtual Machines) written for each different platform. Sometimes it really seems like there's nothing new under the sun. Java is also an object oriented language and is not too dissimilar to C++ in many ways.

It was Java's portability that largely led to it becoming a Web star as JVMs could be built into web browsers thus allowing Java applets to be downloaded as part of a web page and executed within the browser on the client machine. Somewhat on the back of this development Java application servers started to appear on the market whereby server side Java code could be accessed over the web via web servers.

By 1998 the world of transactional web applications, usually referred to as Web 2.0, was ready to take off but several competing and/or complementary technologies were jostling for position, some of them being:

- Tuxedo – tcp/ip SOA middleware based on solid TP Monitor infrastructure
- M3 – tcp/ip OOA middleware based on Tuxedo with a CORBA interface
- Application servers – executable business logic accessed from web pages via web servers
- Java – portable language supporting applets running in browsers and used to create application servers that were easily ported

## *The Internet Storm Comes In*

It was really in 1998 that finally the Internet storm came in. There was a clear commercial demand for the Web to be more than just an information delivery system and to support transactional 'web applications' where the UI was presented in the browser and could access back end business logic running somewhere 'behind' the web server. This would come to be called Web 2.0. Commercial transactions were already being executed over the Internet, though not very elegantly, and the 'dot com' boom was beginning to take off in the stock market, particularly on NASDAQ. With their focus on

transaction processing and large multi-platform distributed applications this was exactly what BEA was hoping for but they had not correctly identified the technology that would be used. It was not to be CORBA enhanced TP monitors but the new Java ‘application servers’. Having backed the wrong horse BEA used their tried and true acquisition strategy to short cut their way to market domination, to become the ‘gorilla’ as Geoffery Moore described it.

(#SOA) There were various Java application servers around many of which could be downloaded over the Internet and of all the contenders BEA chose to acquire WebLogic in 1998 because not only did they have a Java application server they were also committed to Java 2 Enterprise Edition (J2EE). J2EE defined a set of services that had to be provided by a Java application server if it was to be considered as enterprise class. It went far beyond viewing Java as a programming language and specified a complete Internet based transactional middleware environment for Java. It included a full list of services the application server must provide such as the Java Messaging Service (JMS), the Java Transaction Service, the Java Naming Service, JDBC database access and many others, in the early releases it consisted of about 12 services. Providing these kinds of services was second nature to the Tuxedo engineers as was the provision of other infrastructure features like load balancing, manageability and resiliency. The enhancement of the WebLogic application server using experience gained with Tuxedo was a win-win situation and so BEA WebLogic was born.


As the BEA WebLogic product gained traction BEA rebranded itself and replaced its original logo designed by the founders to one designed by professionals, which was a common development in start-ups, and they finally got hold of the domain name bea.com

With the Internet taking off the wisdom of following Geoffrey Moore's advice proved itself as BEA had already put in place the necessary company infrastructure to support the booming BEA WebLogic sales. BEA was on the crest of the Internet infrastructure wave. With a very solid product perfectly placed to take advantage of the development of Web 2.0 BEA UK's customer list began to include major companies like BT.

## *Dot.com Hype and Dot.com Reality*

As the year 2000 approached the dot com frenzy got crazier and crazier and prices on NASDAQ went ever higher. The companies whose stock was being quoted were often, at best, wildly speculative and supporting BEA WebLogic sales calls took me to some surreal sales meetings.

Typical were a couple of guys who were going to set up a dog food delivery web site. Wow, great idea! Mundane details such as the logistics infrastructure required to actually deliver the dog food were not allowed to dampen expectations. They'd be dealt with in phase two!

Mobile phones were now ubiquitous and so 'location dependent services' were also a favourite idea even though there were no GPS satellites up there to locate you. Details, details!

It was another golden era of putting lipstick on the pig and, as usual, the front end browser based UI was way ahead of the old back end systems such as logistics and financial transaction processing. A situation that always leads to problems and it most certainly did this time as web site crashes became commonplace.

# The 21st Century

The first major computer event in the 21st century, in the year 2000, was the event that never happened. The great Millennium Bug meltdown. As the world waited and the clocks struck midnight on the 31st December 1999, aircraft in flight did not crash, nuclear power stations did not go into meltdown and spacecraft did not fly into the sun. But as with so many other technology related anxieties a great many consulting companies had made a great deal of money preparing for this non-event.

As for me, my four busy, crazy years with BEA in the UK office – now with over 100 taxpaying employees – were coming to an end at the same time as Ed Scott's prediction came true and BEA Systems achieved revenues of $1billion. More importantly, as predicted by Bill Coleman, it had developed into precisely the kind of large company I was not temperamentally suited for. I resigned to take some time off. Meanwhile, a 21st century meltdown that, unlike the Millenium Bug problem, really was long overdue began to happen, the end of the dot-com boom. It was becoming ever more possible to see the pig lurking behind the lipstick, a flashy browser based UI that was usually little more than a prototype. The NASDAQ was soon falling like a stone.

Over the next decade of course many of the business and technical predictions that had been made prematurely during the dot-com era did eventually take place but far more slowly than the over ambitious predictions. The following quote about the development of 3D printing, from Vyomesh Joshi chief executive of 3D Systems describes this phenomenon perfectly.

*"[3D printing] …. shared the problem of other technology markets where claims run ahead of reality. "People overestimate what will happen in the next two years and underestimate what will happen in the next five years,"*

## *The View From Outside The Trenches*

Since leaving BEA in 2000 I've taken on some consulting assignments for friends and acquaintances but as the years have rolled by I have let them tail off. My experience of 21st century

computing is patchy with much of it resulting from keeping up to date through the technical media rather than actually being in the trenches using it.

There are many younger commentators who have a much better grasp of the contemporary computing world than me and who produce well thought out articles about it. My own look at 21st century technology in the remaining sections mostly consists of considering how specific current developments have, arguably, resulted from what went before. For in-depth analysis of the current state of computing technology you should seek out the contemporary commentators.

## *New Players And Old Players*

As the 21st century has progressed a whole new stable of major technology players have emerged like Amazon, Google, Facebook, Twitter and salesforce.com, though arguably Google and Facebook are more accurately described as advertising agencies. But as these new companies have grown they have increasingly relied for further innovation on the acquisition of smaller and newer start-up companies. At the same time the older established technology companies have acquired newer companies to help them innovate, such as the Microsoft acquisition of Skype. The scenario laid out to me by Bill Coleman in 1994 still holds good in technology companies. As they grow they tend to ossify and need fresh injections of innovative people from start-ups. In 2017 the successes and failures of these businesses both old and new are still playing out in the market place.

## *What Of Tuxedo?*

BEA itself survived the dot com bust because it actually had useful saleable software and in April 2008 Oracle finally admitted that they did need middleware and were not really skilled in creating it themselves and acquired BEA to bring BEA WebLogic into their Fusion middleware line. They got Tuxedo too. So what of Tuxedo? Well it still works. The current TPC-C benchmark numbers produced with Tuxedo are quite astonishing and can usefully be compared to the pitiful transaction capabilities claimed for the latest financial services technology fad blockchains. Tuxedo services can now easily be exposed over the Internet as web services where they

exhibit the defining properties of REST services. Tuxedo's language support has been extended and now includes web favourites PHP, Python and Ruby thus giving applications written in these languages the full support of the Tuxedo infrastructure. So Tuxedo is alive and well in the age of the Internet and the cloud.

## *The Users Who Came In From The Cold*

(#UI) Back in the days of batch processing there were no end-users of software apart from computer operators and so there was really no such thing as a UI (user interface), let alone a UX (user experience). But as front end software separated from back end software and began to be presented on both intelligent and dumb terminals, then PCs, then browsers, the UI gained in importance.

It has always been a mystery to me how in the 1990's millions of consumers were persuaded to buy expensive, difficult to manage PCs running software with, at best, passable UIs originally designed for business users. At the time all they really wanted to do was send emails and create simple documents. But the appearance of smartphones and tablets running easy to use, cheap and/or free apps, that is to say actual ***consumer*** devices, has brought consumer users in from the cold.

The final implications of this are still far from clear but there are now at least two broad classes of computing technology users, business users and consumer users, and consumer users demand a good UI on an easy to use and maintain device.

## *Mobile Technology*

The advances in mobile technology both in smartphones and tablets and in the back end services that support them provided by infrastructure such as GPS satellites has been completely transformative. The location dependent services that were over ambitiously promised during the dot com boom are now taken for granted.

A major transformative change made possible by mobile technology has been the ability to roll out mobile networks easily and quickly which has had a major effect in transforming 3rd world

countries and economies after years of largely ineffective foreign aid.

Thanks to mobile devices and social media, with Facebook being the gorilla in this market of course, fifty years after Marshall Mcluhan predicted the coming of the global village it now exists. Soon everybody everywhere will be able to communicate in real time with images, video, text and voice, regardless of the language they use.

The social and cultural implications of this are already dramatic and will continue to be and they are much discussed. People who are growing up in the global village most certainly have different views on privacy, friendship, intimate relationships and what is required to become socially accepted than people from my generation can perhaps ever truly understand.

## *Smartness Everywhere*

It was also fifty years ago that Mcluhan predicted that one day consumers would be able to order products that were built to their individual specification. This reality moves ever closer with the appearance of 3D printing and 3D milling devices plus the emerging Internet of Things (IoT). What is the Internet of Things? It's the idea that one day every manufactured item will be 'smart', that is to say it will contain minituarised computer hardware and software and will be able to receive and send control commands as well as status and performance data.

Many homes already have smart electricity meters that can be instructed to send meter readings back to the electricity company without any human intervention. There are also smart cookers that you can command to start cooking from your mobile phone. Smart production lines are already being talked about thus furthering the possibilities for customized manufacturing processes.

With a world of smart things sending real time status and performance data over the Internet there will be a tsunami of data that needs storing and analysing. This is the world of Big Data and Data Analytics. Whereas at Nielsen in 1992 we struggled to assist the manual data entry of bar code information from supermarket till receipts this will all happen 'under the covers' in future. One of the

technology implications of Big Data is the need for ‘new’ tools that allow us to work with it and this has led to the (re)appearance of non-SQL databases. Whatever next.

Smartness is also penetrating our reality, or augmented reality, or virtual reality, which are yet more not very well defined or agreed upon phrases to ponder.

## *‘The Medium Is The Message’*

Mcluhan’s dictum that the medium is the message holds true for the new technology world. Although data content is sometimes singled out for attention with the fact, apparently, that the two most popular forms of Internet content are pornography and crazy cat videos. But that’s really missing the point. The point is not the content of web data but that all this web data is now available to everybody and what this is doing to culture, society, politics and human individuals. It’s much discussed of course.

One interesting example is how search engines have effected the way we organize information. In the past information tended to be organized in hierarchical structures such as folders and subfolders and the phrase ‘drill down’ was used to describe navigating your way through the hierarchy to get to the information you wanted. But nowadays with such powerful search engines and the ability to attach meaningful tags to any kind of data the question of whether or not the underlying structure is hierarchical becomes irrelevant. As does the ability to navigate such structures. This must surely be effecting the way we think.

## *Open Source*

The widespread availability of reliable and cheap, even free, software source code has been revolutionary and was not a widely predicted phenomenon even quite recently. Perhaps the biggest open source event has been the appearance of Linux, an open source version of Unix running on Intel hardware. Linux has rapidly put an end to the Unix wars between players such as Sun, IBM, HP and so on, who all now offer Linux. It seems that the open source movement tends to support the adoption of standards and even the creation of new open standards.

Happily the days of wasteful religious wars driven by competing vendor specified proprietary standards are largely over, which is good. Though of course Apple still insists on using its own non-standard cable terminators and other accessories.

## *Networks And SOA*

(#SOA) Networking standards have become pretty much settled with Ethernet, TCP/IP and HTPP unchallenged and SOA is without doubt the dominant software architecture layered on top of these network standards. OOA has not really become a major player in networked applications. The cloud has built on the principles of 3-tier SOA to create a multi-tiered service based architecture with 2-tier client server relegated to the sidelines.

(#PZ) Today's Internet apps are all about using business services provided transparently by back end systems that reside somewhere in the cloud. The rapid growth of service related acronyms pays tribute to this movement such as SaaS (Software As A Service), IaaS (Infrastructure as a Service), PaaS (Platform as a Service) and so on. Even the 'service' architectural software component itself is now being further componentized and the current talk is of microservices and even microkernels. Along with the proliferation of services have come the necessary specifications of the APIs through which to call them.

Wirelessness is pushing forward with WiFi and Bluetooth continuing to drive the move to get rid of unnecessary cables for connecting devices to the network and it is now possible to charge some devices without the use of cables.

## *AI*

(#AI) AI is back in the news in particular the idea that instant messaging (IM) systems will be able to take natural text, or actual speech, and by using specific chatbots allow you to do what you want to do (say book an airline ticket) without you ever realizing you're actually accessing some back end reservation system. This all sounds curiously like the original Turing test which was to be carried out using teletypes except that now you will use a mobile phone and an instant messaging app. That's progress I suppose. Machine learning is another buzz phrase getting a lot of attention.

However articles are now starting to appear pointing out that there is really no agreement on just what is meant by phrases like ‘artificial intelligence’ and ‘machine learning’. Some clarity would certainly be welcome.

## *And The Final Stormcloud Is …The Cloud!*

(#SOA) The ‘cloud’ has become a much hyped concept though in reality it is basically a term for a globally available standards based network (the Internet) which contains globally addressable service providers (web sites, SaaS providers, etc). If in doubt just think of it as the Internet or the Web finally achieving the potential which was over optimistically predicted for them almost twenty years ago.

An architecture underlying much of the cloud, though being increasingly challenged by ARM, Qualcomm and others, is the Intel chip hardware architecture, which can host either Windows or Linux thus giving it an enormous reach. This has played very neatly into Microsoft’s own cloud offering, Azure, which can be viewed (well by me at least) as the basic Tuxedo architecture with Tuxedo server processes being replaced by Intel VMs (Virtual Machines) running Windows or Linux.

The other big player in cloud offerings is of course Amazon with AWS and other major technology players have their own offerings too, though some say they have left it too late.

## *21st Century Programming (Coding)*

# Programming Languages

How is the craft of programming faring in the 21st century? In the 21st century’s first decade people began to realise that nowhere near enough programmers were being trained in western countries. Programmer training had somehow fallen off the radar screen. One result was that in Cambridge in England some computer scientists decided to form an organisation to create a low cost computer that young enthusiasts could use to teach themselves programming, this became the Raspberry Pi. The success of the Raspberry Pi has been phenomenal and at the same

time people like Bill Gates have been getting involved in starting successful organisations like code.org. Similar organisations have sprung up elsewhere to encourage people to become programmers and coding bootcamps have been mushrooming.

It seems that, perhaps at five minutes before midnight, programming has been rescued and is growing again, though it is now more commonly referred to as coding. With the profession hopefully secured what is happening in the way of programming languages? One idea from the early days, that a single language could replace all others, has been proved to be completely misguided as languages have continued to proliferate. Some people talk about thousands of programming languages.

Many articles get published with titles like 'The Top 10 Programming Languages' but they all use different surveying techniques and do not always agree on what constitutes a programming language. However there is a very stable group of languages that regularly appeared in most of these articles in 2017 and they are:

- Java
- C
- Javascript
- C++
- C#
- Perl
- Python
- Ruby
- PHP
- SQL

Objective-C (Apple) and HTML5 sometimes appear too.

It's interesting how successful Javascript continues to be and anecdotally it may even be rivalling Java (the two languages not being related despite the names similarity). Interesting too is C's continuing high popularity which shows that procedural programming is alive and well and that OOP has by no means replaced it.

## Programming Techniques

This odyssey has followed the development of programming techniques from monolithic mainframe program executables, through modular program executables first common on servers, on to modular program executables making remote procedure calls to other independent programs in the 3-tier architecture. Today we have cloud clients in front end devices calling services from the multi-tiered cloud which may call each other and may need to access transactional legacy systems. Service providers now publish strictly defined APIs.

The programming techniques and programming languages required by programs in the different layers - front end apps, cloud based services, back end legacy systems, require different techniques with most current focus and discussion seeming to be placed, as usual, on the front end, the lipstick. The phrase 'full stack' is now in use and is intended to include software components other than the front end lipstick but in most cases it only covers as far as the servers. The mainframes doing most of the transactional processing heavy lifting that provides the core services are usually ignored.

(#PZ) (#SOA) It appears that the SOA architecture is pushing towards further componentization and minituarisation with services calling other services that become increasingly small. This extract from a post by Matt Miller (@mcmiller00) on 23rd January 2016 describes this development:

*"Today's digital innovators can trace a similar historical path that starts with mainframe computers and monolithic applications and then, step-by-step, reveals software's interchangeable parts until we arrive at today's cloud-based era of microservices and continuous integration.*

*Microservices is an approach to building software that shifts away from large monolithic applications toward small, loosely coupled and composable autonomous pieces. The benefit of this abstraction is specialization, which drives down costs to develop and drives up agility and quality — while operating much more resilient systems."*

## Devops

When production line program development was alive and well the waterfall development process drew a hard line between programs in development and programs in operational production, though this was always compromised by the need for maintenance programming. Once real users started to interact with programs this distinction became harder and harder to maintain as real users tend to want real enhancements made to programs right now. An effort was made to allow more flexibility through Rapid Application Development (RAD) and more recently with the much touted Agile development techniques. This all seems to have climaxed in Devops which accepts that development and production can no longer be kept separate.

## Apps

(#PZ) Apps were initially small software clients running on smartphones though the term has become harder and harder to pin down.

In some ways apps are the culmination of the programmer's revolution, almost anybody can now develop and market a consumer app.

Consumer apps usually provide more highly specific functionality, say your banking needs, than web sites or enterprise apps. But this has led to the existence of millions of apps. Is this any better than having millions of web sites? Another religious war could be brewing up over this.

# Postscript/O Brother Where Art Thou?

We have reached the end of the odyssey. But where have we arrived?

Well in some ways, like Odysseus himself, we have finally come home, right back to where we started. Some things have changed, we now have a massive multi-layered global network of increasingly smart devices, including the ultimate smart device the human being. Yet ultimately it's all based on the same old Von Neuman computer architecture and on the two digits 0 and 1, the binary system. The whole extraordinary software edifice rests on that foundation, which is really mindboggling.

Have we learned anything on our odyssey that suggests this is likely to change? Perhaps a new computer hardware architecture will finally emerge? Articles regularly appear about quantum computing devices and IBM have announced a project to research cognitive SyNAPSE chips but so far these are still very much research projects. It is also being said that the minituarisation of silicon chips by Intel and others will reach its limit by 2020 and Moore's Law will finally come to an end.

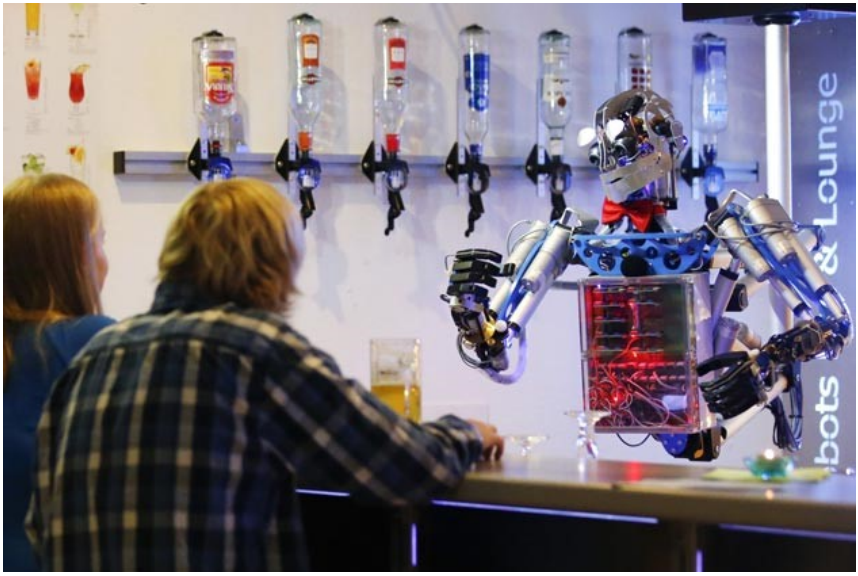
On our odyssey we, like Odysseus, have encountered dangerous and worrying phenomena, in particular whether any conceivable developments in computer technology will put the human race in danger. It's hard to believe that based on existing system components a self-conscious sentient being will come into existence like HAL in 2001 and threaten our own survival. Throughout history major technological advances have tended to become models for explaining human beings, always incorrectly. In the digital age this has meant starting to see human beings as just computing machines. Will this prove to be true or false? The jury is still out.

But if we are in danger from our technology it seems to me most likely that it will come from an artificial living creature/ cyborg/ replicant enhanced with implanted computer chips and network interfaces. Created perhaps by using advanced tools such as DNA editors? The UK TV series 'Humans' is an interesting attempt to confront some of these issues.

Questions, questions. As the philosophically inclined outlaw replicant Ray Batty so presciently remarked in BladeRunner.

It would be so good to be able to discuss these questions over a few beers with Pythagoras and Aristotle who started the whole ball rolling over two millenia ago. If only it were possible.

But perhaps it will be possible one day in the future when reality, virtual reality and augmented reality, whatever they all are, have merged.




If it happens, the drinks are on me.

# Appendices

# *Appendix A – The ASCII character set*

| ASCII Printable Character Conversion Table | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Binary | Dec | Hex | Char | Binary | Dec | Hex | Char | Binary | Dec | Hex | Char |
| 010 0000 | 32 | 20 | space | 100 0000 | 64 | 40 | @ | 110 0000 | 96 | 60 | ` |
| 010 0001 | 33 | 21 | ! | 100 0001 | 65 | 41 | A | 110 0001 | 97 | 61 | a |
| 010 0010 | 34 | 22 | " | 100 0010 | 66 | 42 | B | 110 0010 | 98 | 62 | b |
| 010 0011 | 35 | 23 | # | 100 0011 | 67 | 43 | C | 110 0011 | 99 | 63 | c |
| 010 0100 | 36 | 24 | $ | 100 0100 | 68 | 44 | D | 110 0100 | 100 | 64 | d |
| 010 0101 | 37 | 25 | % | 100 0101 | 69 | 45 | E | 110 0101 | 101 | 65 | e |
| 010 0110 | 38 | 26 | & | 100 0110 | 70 | 46 | F | 110 0110 | 102 | 66 | f |
| 010 0111 | 39 | 27 | ' | 100 0111 | 71 | 47 | G | 110 0111 | 103 | 67 | g |
| 010 1000 | 40 | 28 | ( | 100 1000 | 72 | 48 | H | 110 1000 | 104 | 68 | h |
| 010 1001 | 41 | 29 | ) | 100 1001 | 73 | 49 | I | 110 1001 | 105 | 69 | i |
| 010 1010 | 42 | 2A | * | 100 1010 | 74 | 4A | J | 110 1010 | 106 | 6A | j |
| 010 1011 | 43 | 2B | + | 100 1011 | 75 | 4B | K | 110 1011 | 107 | 6B | k |
| 010 1100 | 44 | 2C | , | 100 1100 | 76 | 4C | L | 110 1100 | 108 | 6C | l |
| 010 1101 | 45 | 2D | - | 100 1101 | 77 | 4D | M | 110 1101 | 109 | 6D | m |
| 010 1110 | 46 | 2E | . | 100 1110 | 78 | 4E | N | 110 1110 | 110 | 6E | n |
| 010 1111 | 47 | 2F | / | 100 1111 | 79 | 4F | O | 110 1111 | 111 | 6F | o |
| 011 0000 | 48 | 30 | 0 | 101 0000 | 80 | 50 | P | 111 0000 | 112 | 70 | p |
| 011 0001 | 49 | 31 | 1 | 101 0001 | 81 | 51 | Q | 111 0001 | 113 | 71 | q |
| 011 0010 | 50 | 32 | 2 | 101 0010 | 82 | 52 | R | 111 0010 | 114 | 72 | r |
| 011 0011 | 51 | 33 | 3 | 101 0011 | 83 | 53 | S | 111 0011 | 115 | 73 | s |
| 011 0100 | 52 | 34 | 4 | 101 0100 | 84 | 54 | T | 111 0100 | 116 | 74 | t |
| 011 0101 | 53 | 35 | 5 | 101 0101 | 85 | 55 | U | 111 0101 | 117 | 75 | u |
| 011 0110 | 54 | 36 | 6 | 101 0110 | 86 | 56 | V | 111 0110 | 118 | 76 | v |
| 011 0111 | 55 | 37 | 7 | 101 0111 | 87 | 57 | W | 111 0111 | 119 | 77 | w |
| 011 1000 | 56 | 38 | 8 | 101 1000 | 88 | 58 | X | 111 1000 | 120 | 78 | x |
| 011 1001 | 57 | 39 | 9 | 101 1001 | 89 | 59 | Y | 111 1001 | 121 | 79 | y |
| 011 1010 | 58 | 3A | : | 101 1010 | 90 | 5A | Z | 111 1010 | 122 | 7A | z |
| 011 1011 | 59 | 3B | ; | 101 1011 | 91 | 5B | [ | 111 1011 | 123 | 7B | { |
| 011 1100 | 60 | 3C | < | 101 1100 | 92 | 5C | \ | 111 1100 | 124 | 7C | \| |
| 011 1101 | 61 | 3D | = | 101 1101 | 93 | 5D | ] | 111 1101 | 125 | 7D | } |
| 011 1110 | 62 | 3E | > | 101 1110 | 94 | 5E | ^ | 111 1110 | 126 | 7E | ~ |
| 011 1111 | 63 | 3F | ? | 101 1111 | 95 | 5F | _ | | | | |

The 8 bit (one byte) ASCII character set uses the values from decimal 32 to decimal 126 to represent printable characters. Other values represent device control characters for terminals and printers and other special characters.

## *Appendix B - Peter Brown On CIS COBOL*

This appendix contains a readable transcript of the above mentioned Computing article followed by a scanned image of the original.

**Transcript**

Computing – 11th May 1978

# CIS Cobol

*'Software developers stand more chance of long term success if they make an initial investment in portability and can then implement their software on whatever machines are successful. Portability brings advantages to the user. He can get his compilers moved to a new machine, at a price. More importantly he can, if his compiler offers a standard language, move his program from one compiler to another'*

**Peter Brown, Professor of Computing at Kent University**

The pace of change in microprocessor hardware is great; and this presents a challenge to those who write software for micros, since the software continually needs changing to fit new environments. One way of meeting this challenge is to produce portable software. Portable software involves making an initial investment at the design stage to ensure that the software will be easy to convert to new environments. This investment is not trivial; typically portable software costs twice as much as machine-dependent software, and takes twice as long produce. However each time the portable software is implemented on a new machine there is a healthy return on the original investment. If the number of implementations reaches a dozen or so, the implementors can retire to the Caribbean.

One small British company that has embarked on this trail is Micro Focus. It has produced a portable compiler for Cobol on micros. It is called CIS Cobol, and, unlike cross-compilers, actually runs on the micro that it is compiling for. Clearly the demand for good Cobol will be almost boundless, so there is ample scope for

success. Not surprisingly, however, there is plenty of competition in such an attractive market. In particular CAP in the UK has produced MicroCobol and there are numerous microprocessor Cobols being produced in the United States.

There are many interesting technical problems facing a small portable Cobol compiler. How is it made portable, and how is the portablity process started? What level of language can be offered? How can the compiler be made fit in a small space?

The purpose of this article is to explain how CIS Cobol has tackled these problems. Compared with other software, compilers are hard to make portable. This is because not only does the compiler need to be moved to a new computer, but the action of the compiler itself needs to be changed in order that it produce code for the new computer. This problem has been successfully attacked by research workers, and several techniques suitable for use in production compilers have been developed. The most popular is the use of a machine-independent intermediate language. The first part of the compiler, often called the translator, maps the source language into the intermediate language. The second part is either a generator which translates the language down to machine code, or an interpreter, which interprets the intermediate language on the object machine. The attraction of these techniques is that the translator is machine-independent and can be coded in any suitable portable high-level language. The second part of the compiler is usually re-coded for each machine. (If it is an interpreter the second part could be coded in a machine independent high-level language like the translator, but it is not usually done this way, for reasons of efficiency.) Because of this recoding, the work needed to implement a portable compiler on a new machine is not trivial. However the work is, on average, only about one-third that of developing a new compiler from scratch. Moreover there is proportionately less danger of the host of bugs that permeate completely new compilers when they are written in a shoddy manner.

CIS Cobol uses the intermediate language technique. Since the CIS Cobol intermediate language is designed for micros, where space is tight, it is normally interpreted. Moreover, again because space is tight, the interpreter is hand-coded in assembly language for each machine. There is, however, no reason why the intermediate language should not be compiled if it was to run on a

larger machine. The CIS Cobol translator is itself written in CIS Cobol. 'Writing a compiler in itself' is a classic technique of compiler writers. It makes porting much easier. The encoding of the translator can be converted to the intermediate language by using any existing version of the compiler. Given this, once an interpreter (or code-generator) for the intermediate language has been written for a new machine, then the translator is available immediately without further ado and the compiler is complete. There is only one problem: if you need an existing implementation to help build a new one, how do you build the first implementation? In particular how is the CIS Cobol translator converted for the first time from CIS Cobol to the intermediate language?

Micro Focus solved this problem by using a macro processor. General-purpose macro processors can be used, within certain strict limits, to translate one language to another, and hence provide a natural tool. Micro Focus chose to use the Stage 2 macro processor. **One man** was given the task of writing the macros. He worked in a world of his own. He started with no knowledge of Stage 2, and moreover never met anyone who knew Stage 2 until the project was complete. I finally broke the spell by acknowledging some passing acquaintance with Stage 2. Yet he got his macros to work, and achieved this in ten weeks. Granted the macros do not run very fast. Because of space limitations with the micro on which Stage 2 runs, the macros require thirteen separate passes. The end result is that it takes thirty hours of computer time to process the translator - just right for a week-end. When the compiler can compile itself the macros will not be needed any more. Their author will heave a sigh of relief, with a mite of regret that his creation has died. The macros will have provided a quick (in people's time, not computer time) and dirty solution to a pressing problem.

It is not only the compiler writers who want portability. The end user wants it too. The best aids to portability for the user are the international standards. These standards are far from perfect, but life is much more difficult without them. Thus the Cobol user, unless he is happy to be locked in to a single hardware or software supplier, would do well to get an ANSI standard Cobol compiler.

CIS Cobol is compatible with the ANSI standard, but, being a small compiler, the language offered is not up to the minimal Level I of the standard. (Although the language does not fully cover Level

l, it actually includes a few more advanced features from Level 2. The extras are good to have but do not fully compensate the omissions from Level 1.) The implication of this to the user is that if he gets Cobol programs from elsewhere he will probably have to reprogram parts of them to run on CIS Cobol. In the opposite direction, running CIS Cobol programs on larger Cobol compilers, there should be no special problems.

CIS Cobol is specially designed for interactive work on a display terminal. The Accept and Display verbs can be used to transfer blocks of data and from the screen. The format of the data is described as ordinary Cobol records.

The first implementation of CIS Cobol was done in conjunction with Dataskil. It runs on the ICL 1500 Transaction System, and needs only 8K bytes with overlaying from backing storage. All sorts of devices were needed to shoe-horn the compiler into such a tight space. One was to make the intermediate language especially concise, so that every bit earned its keep. Since both the compiler and the code it produces are represented in this intermediate language, this conciseness is one of the keys to success.

The ICL 1500 implementation was completed in six months, with a peak of ten people working on the project. The completion of the project was achieved by 'a nice piece of human engineering; a director of Micro Focus was chained to his desk until the work was finished'.

The second implementation has been done on the Intel 8080, and follows similar lines. The minimum storage requirement has been increased to 16K, and this has given scope for streamlining the compiler, improving its user interface and augmenting the language facilities. Compilation speed on the Intel 8080 is about thirty CIS Cobol statements per minute.

An advantage of the portability of the compiler is that if it is moved to an Intel 8086, it can easily be made to exploit the extra features of the newer machine.

CIS Cobol is not currently sold direct to the end user - Micro Focus is not set up to deal in this way - but is sold to vendors of

computer systems. The Intel 8080 implementation has already been sold to one such vendor, and, since more and more companies are selling systems based on micros, there is plenty of promise for further sales.

Portability is going to help many software implementers along the road to a life of luxury.

LANGUAGES

# CIS Cobol

'Software developers stand more chance of long-term success if they make an initial investment in portability and can then implement their software on whatever machines are successful. Portability brings advantages to the user. He can get his compilers moved to a new machine, at a price. More importantly he can, if his compiler offers a standard language, move his program from one compiler to another' – Peter Brown, Professor of Computing at Kent University.

Andy Wise

The pace of change in microprocessor hardware is great; this presents a challenge to those who write software for micros, since the software continually needs changing to fit new environments. One way of meeting this challenge is to produce portable software. Portable software involves making an initial investment at the design stage to ensure that the software will be easy to convert to new environments. This investment is not trivial; typically portable software costs twice as much as machine-dependent software, and takes twice as long to produce. However each time the portable software is implemented on a new machine there is a healthy return on the original investment. If the number of implementations reaches a dozen or so, the implementers can retire to the Caribbean.

One small British company that has embarked on this trail is Micro Focus. It has produced a portable compiler for Cobol on micros. It is called CIS Cobol, and, unlike cross-compilers, actually runs on the micro that it is compiling code for. Clearly the demand for a good Cobol compiler on micros will be almost boundless, so there is ample scope for success. Not surprisingly, however, there is plenty of competition in such an attractive market. In particular CAP in the UK has produced MicroCobol and there are numerous microprocessor Cobols being produced in the United States.

There are many interesting technical problems facing a small portable Cobol compiler. How is it made portable, and how is the portability process started? What level of language can be offered? How can the compiler be made to fit in a small space?

The purpose of this article is to explain how CIS Cobol has tackled these problems.

Compared with other software, compilers are hard to make portable. This is because not only does the compiler need to be moved to a new computer, but the action of the compiler itself needs to be changed in order that it produce code for the new computer. This problem has been successfully attacked by research workers, and several techniques suitable for use in production compilers have been developed. The most popular is the use of a machine-independent intermediate language. The first part of the compiler, often called the translator, maps the source language into the intermediate language. The second part is either a code-generator which translates the intermediate language down to machine code, or an interpreter, which interprets the intermediate language on the object machine. The attraction of these techniques is that the translator is machine-independent and can be coded in any suitable portable high-level language. The second part of the compiler is usually re-coded for each machine. (If it is an interpreter the second part could be coded in a machine-independent high-level language like the translator, but it is not usually done this way, for reasons of efficiency.) Because of this re-coding, the work needed to implement a portable compiler on a new machine is not trivial. However the work is, on average, only about one-third that of developing a new compiler from scratch. Moreover there is proportionately less danger of the host of bugs that permeate compilers when they are written in a shoddy manner.

CIS Cobol uses the intermediate language technique. Since the CIS Cobol intermediate language is designed for micros, where space is tight, it is normally interpreted. Moreover, again because space is tight, the interpreter is hand-coded in assembly language for each machine. There is, however, no reason why the intermediate language should not be compiled if it was to run on a larger machine. The CIS Cobol translator is itself written in CIS Cobol. 'Writing a compiler in itself' is a classic technique of compiler writers. It makes porting much easier. The encoding of the translator can be converted to the intermediate language by using any existing version of the compiler. Given this, once an interpreter (or code-generator) for the intermediate language has been written for a new machine, then the translator is available immediately without further ado and the compiler is complete. There is only one problem: if you need an existing implementation to help build a new one, how do you build the first implementation? In particular how is the CIS Cobol translator converted for the first time from CIS Cobol to the intermediate language?

Micro Focus solved this problem by using a macro processor. General-purpose macro processors can be used, within certain strict limits, to translate one language to another, and hence provide a natural tool. Micro Focus chose to use the Stage 2 macro processor. One man was given the task of writing the macros. He worked in a world of his own. He started with no knowledge of Stage 2, and moreover never met anyone who knew Stage 2 until the project was complete. I finally broke the spell by acknowledging some passing acquaintance with Stage 2. Yet he got his macros to work, and achieved this in ten weeks. Granted the macros do not run very fast. Because of space limitations with the micro on which Stage 2 runs, the macros require thirteen separate passes. The end result is that it takes thirty hours of computer time to process the translator — just right for a week-end. When the compiler can compile itself the macros will not be needed any more. Their author will heave a sigh of relief, tinged with a mote of regret that his creation has died. The macros will have provided a quick (in people's time, not computer time) and dirty solution to a pressing problem.

It is not only the compiler writers who want portability. The end user wants it too. The best aids to portability for the user are the international standards. These standards are far from perfect, but life is much more difficult without them. Thus the Cobol user, unless he is happy to be locked in to a single hardware or software supplier, would do well to get an ANSI standard Cobol compiler.

CIS Cobol is compatible with the ANSI standard, but, being a small compiler, the language offered is not up to the minimal Level 1 of the standard. (Although the language does not fully cover Level 1, it actually includes a few more advanced features from Level 2. These extras are doubtless good to have but do not fully compensate the omissions from Level 1.) The implication of this to the user is that if he gets Cobol programs from elsewhere he will probably have to reprogram parts of them to run on CIS Cobol. In the opposite direction, running CIS Cobol programs on larger Cobol compilers, there should be no special problems.

CIS Cobol is specially designed for interactive work on a display terminal. The Accept and Display verbs can be used to transfer blocks of data to and from the screen. The format of the data is described as ordinary Cobol records.

The first implementation of CIS Cobol was done in conjunction with Dataskil. It runs on the ICL 1500 Transaction System, and needs only 8K bytes, with overlaying from backing storage. All sorts of devices were needed to shoe-horn the compiler into such a tight space. One was to make the intermediate language especially concise, so that every bit earned its keep. Since both the compiler and the code it produces are represented in this intermediate language, this conciseness is one of the keys to success.

The ICL 1500 implementation was completed in six months, with a peak of ten people working on the project. The completion of the project was achieved by 'a nice piece of human engineering; a director of Micro Focus was chained to his desk until the work was finished'.

The second implementation has been done on the Intel 8080, and shows similar lines. The minimum storage requirement has been increased to 16K, and this has given scope for streamlining the compiler, improving its user interface and augmenting the language facilities. Compilation speed on the Intel 8080 is about thirty CIS Cobol statements per minute.

An advantage of the portability of the compiler is that if it is moved to an Intel 8086, it can easily be made to exploit the extra features of the newer machine.

CIS Cobol is not currently sold direct to the end user — Micro Focus is not set up to deal in this way — but is sold to vendors of computer systems. The Intel 8080 implementation has already been sold to one such vendor, and, since more and more companies are selling systems based on micros, there is plenty of promise for further sales.

Portability is going to help many software implementers along the road to a life to luxury.

MAY 11 1978

COMPUTING EUROPE 17

Computing – May 11th, 1978

## *Appendix C – BEA and GA*

This appendix contains a readable transcript of the above mentioned Computer Weekly article followed by a scanned image of the original.

### **Transcript** Computer Weekly - 16th May 1996

### BEA Systems buys its way to the top of Tuxedo world - *J. Green-Armytage*

BEA Systems has bought Unix Systems Laboratories, of France, rounding off a series of deals to give it pole position in the critical arena of middleware software for Unix-based systems.

Last week's acquisition of USE Europe's largest distributor of Tuxedo, brought another 35 employees to swell BEA Systems' total head count to 270. Its first move outside the US was the acquisition of **GA Systems**, a consultancy specialising in online transaction processing in London. However, the cornerstone of BEA Systems' strategy was its deal with Novell at the end of January to acquire the development and marketing rights to Tuxedo, the key middleware software for handling transaction processing in Unix.

Backed with $100m (£66m) from investment bank Warburg Pincus, which owns 75% of the company, BEA acquired leading US suppliers of Tuxedo solutions Information Management Company and Independence Technologies. It then bought VISystems which developed products for Unix to be compatible with IBM's CICS middleware software.

Ed Scott, executive vice-president for worldwide operations, said the string of deals meant "the fragmented pieces of the Tuxedo world have come together under a single flag". Scott conceded that the market for BEA's products was small but he said it would grow to top $1bn a year by the end of the century.

The company has set up a London office, to be run by Stephen Allen, while Joe Menard, who had been in charge of Tuxedo operations at Novell, will be senior vice-president for European operations based in Brussels.

BEA Systems was founded in January last year by **B**ill Coleman, **E**dward Scott and **A**lfred Chuang.

Computer Weekly - 16th May 1996

SCOTT . . . bringing the Tuxedo world together under one flag

# BEA Systems buys its way to the top of Tuxedo world

**J. Green-Armytage**

BEA Systems has bought Unix Systems Laboratories, of France, rounding off a series of deals to give it pole position in the critical arena of middleware software for Unix-based systems.

Last week's acquisition of USL, Europe's largest distributor of Tuxedo, brought another 35 employees to swell BEA Systems' total head count to 270. Its first move outside the US was the acquisition of GA Systems, a consultancy specialising in online transaction processing in London.

However, the cornerstone of BEA Systems' strategy was its deal with Novell at the end of January to acquire the development and marketing rights to Tuxedo, the key middleware software for handling transaction processing in Unix.

Backed with $100m (£66m) from investment bank Warburg Pincus, which owns 75% of the company, BEA acquired leading US suppliers of Tuxedo solutions Information Management Company and Independence Technologies. It then bought VISystems which developed products for Unix to be compatible with IBM's Cics middleware software.

Ed Scott, executive vice-president for worldwide operations, said the string of deals meant "the fragmented pieces of the Tuxedo world have come together under a single flag".

Scott conceded that the market for BEA's products was small but he said it would grow to top $1bn a year by the end of the century.

The company has set up a London office, to be run by Stephen Allen, while Joe Menard, who had been in charge of Tuxedo operations at Novell, will be senior vice-president for European operations based in Brussels.

BEA Systems was founded in January last year by Bill Coleman, Edward Scott and Alfred Chuang.

# *Appendix D – Some Useful References*

### Author's research

See **http://spanitis.eu/links-articles** to access the author's curated research collection and the blog on the same web site. Also keep an eye on **http://theprogrammersodyssey.com/blog**.

### Computing history sites

UK National Museum of Computing

### Influential Books & Articles

- Education and Ecstasy – George Leonard –1968
- The Gutenberg Galaxy – Marshall Mcluhan - 1962
- Understanding Media – Marshal Mcluhan - 1963
- What Computers Still Can't Do - Hubert L. Dreyfus
- Rise of the Machines - Thomas Rid
- Homo Deus: A Brief History of Tomorrow - Yuval N Harari

### Seminal Computing Texts

Notes – Ada Lovelace - 1843
Go To Statement Considered Harmful - Edsger Dijkstra - March 1968 Communications of the ACM

### PDP11 References

Wikipedia contains good information under the headings RSX11, MACRO-11 and PDP.

### Computers In Groundbreaking Novels

The Tin Men – Michael Frayn - 1965
Neuromancer – William Gibson – 1984

# Notes

*"Well-paced account, enlivened by weaving in coming-of-age tales, and the music of the time. Who would have thought a book on the history of programming could be so enthralling?"*

Andrew Lewis, Fujitsu UK