Chapter 7
The Growth of Structure

Evolution is an ascent towards consciousness.
Therefore it must culminate ahead in some kind of supreme consciousness.
Pierre Teilhard de Chardin

The main purpose of this chapter is to illustrate the exponential chararacteristics of the growth curve, which occurs throughout Nature, with the growth of structure in the computer industry in particular. We can thereby see how evolution could overcome the limits of technology, outlined in Chapter 8, 'Limits of Technology' on page 619, taking humanity out of the evolutionary cul-de-sac we find ourselves in today, outlined in Chapter 9, 'An Evolutionary Cul-de-Sac' on page 643. Chapter 6, 'A Holistic Theory of Evolution' on page 521 showed how the accelerating pace of evolutionary change can be modelled by an exponential series of diminishing terms, which has a finite limit at the present time, the most momentous turning point in some fourteen billion years of evolutionary history.

But because of science's ignorance of the Principle of Unity—the fundamental design principle of the Universe—science has no satisfactory explanation for this exponential rate of evolutionary change. As a consequence, we are managing our business affairs with little understanding of the evolutionary energies that cause us to behave as we do. Even when the Principle of Unity is presented to scientists, they tend to deny that it is a universal truth, because it threatens their deeply held scientific convictions, the implications of which we shall look at in the Epilogue.

In the meantime, we can use the Principle of Unity to examine this critical situation. In essence, evolution is a process of divergence and convergence, of analysis and synthesis in the noosphere. But when we analyse without subsequently integrating the divergent parts that are produced, the result is a fragmented mind. Because of academic specialization, we can see sections of the jigsaw puzzle, but not how they coherently fit together in the Big Picture. Our mechanistic conditioning thus prevents us from realizing our fullest potential as divine, cosmic beings, living in Wholeness.

The Unified Relationships Theory provides this synthesis of everything. The URT views the Universe in the abstract terms of structure, form, relationships, and meaning, rather than the primary concepts of science today: space, time, matter, and energy. This means that meaningful structure-forming relationships are universally causal, being created by Life or the Logos, arising directly from our Divine Source. With these abstract concepts, we can see that evolution is not just a biological process. The development of the stars and atoms—the large and the small—in the physiosphere and our learning or mental development in the noosphere are as much evolutionary processes as the evolution of the species in the biosphere.

In other words, we can view all fourteen billions years of evolution since the most recent big bang as a seamless continuum of development in the Eternal Now. And as all evolutionary processes follow the same underlying pattern, it is sufficient to study our own learning processes to understand evolution as a whole through self-inquiry. In terms of business information systems modelling, it is thus necessary to use our self-reflective Intelligence to include this modelling process in the territory being modelled, as explained in Part I. It is in this way that evolution can become fully conscious of itself, leading to superconsciousness, in conformity with Teilhard's law of complexity-consciousness: the greater the complexity, the greater the consciousness.[1] But if we are not to be overwhelmed by this complexity, we need to acknowledge the simple role of the Principle of Unity in this developmental growth of structure.

Growth of computer technology

So what about the information technology industry, which dominates all our lives today? Where are we on the growth curve of this industry and where is it likely to lead in the coming years? Well, we cannot possibly answer these questions unless we are willing to adapt to the accelerating pace of evolutionary change and thus be liberated from the seven pillars of unwisdom that influence so much thinking on this subject today.

The key point to recognize here is that the programmable computer, which forms the hub of the IT industry, is a tool that enhances our mental faculties in some sense, not our physical abilities. We can say this because the computer provides a means of storing data, not unlike our memory, and a means of processing this data, somewhat analogous to our thinking, reasoning, and calculating skills.

On this point, it is interesting to note that the Greek word *logos*, which is the root of both *logic* and *logistic*, could mean both 'reasoning' and 'reckoning'. However, in a culture that is obsessed with quantitative measure, the emphasis is more on the latter skill than the former, as the word *computer* indicates. The Swedes call the computer *dator*, meaning 'a machine for processing data', a more accurate term.

The computer is not the first tool that we have invented to extend our mental abilities. The first such tool is generally regarded as the abacus, which is probably of Babylonian origin. "The word *abacus* is probably derived, through its Greek form *abakos*, from a Semitic word such as the Hebrew *ibeq* ('to wipe the dust', noun *abaq* 'dust')".[2] The tool was so named because originally people used a board or slab covered in sand in which they made marks to help them with their calculations. It was only later that the tool evolved into one where counters were strung on wires. In Roman times, these counters were stones moving in grooves on the board, hence the English words *calculate* and *calculus*, from the Latin *calculus*, meaning 'a stone'.

Then in the seventeenth to nineteenth centuries, there was a flurry of activity as a series of inventors built various kinds of calculating machines. The first of these appears to be a German astronomer and mathematician, Wilhelm Schikard, who built what he called a 'calculating clock' in 1623. This invention was followed by a calculator, called the 'Pascaline' or 'Arithmetic Machine', designed and built in 1642 by Blaise Pascal, the French mathematician-philosopher, for his father, who was a tax collector. Over fifty of these machines were built over the next ten years.[3]

Then in 1671, the German mathematician-philosopher, Gottfreid Wilhelm von Leibniz, designed a calculating machine called the Set Reckoner, this machine being built two years later[4]. But Leibniz's thoughts went further than this. He dreamed of a machine for mechanizing reason by manipulating symbols that represent concepts, a goal that computer scientists are still trying to realize.[5]

Not much more seems to have happened in the development of calculating machines until 1820, when Charles Xavier Thomas de Colmar of France invented the first mass-produced calculating device to come on the market, called the Arithmometer (progress in the eighteenth century was still on the AB section of the growth curve). The Arithmometer was based on Leibniz's technology and was so successful that it was still in production in 1926.[6]

However, theoretically, at least, the greatest breakthroughs came through Charles Babbage, an English mathematician, who held the Lucasian chair of mathematics at the University of Cambridge, as Isaac Newton had done before him, for some eleven years in mid-life.[7] About 1821, Babbage first conceived of a mechanical device that could automate long, tedious astronomical calculations. Babbage was concerned that the logarithm tables used for navigation at sea contained many errors. This was because they were produced by 'computers', the name given to the people who operated calculating machines.

Babbage gave two different descriptions of the circumstances that gave rise to the idea of calculating mathematical tables automatically by machine, the one he gave in 1834 probably being more accurate than his recollections in his autobiography published thirty years later. Babbage was with John Herschel at the Astronomical Society in London when the former ex-

claimed, "I wish to God these calculations had been executed by steam." To which Herschel replied, "It is quite possible."[8] It was from this chance remark that Babbage devoted the rest of his life to the design of machines that could automate calculations.

Babbage called his first calculating machine the Difference Engine because it was based on the mathematical method of differences.[9] By the summer of 1822, he had constructed a prototype of this machine and began to show it to his friends and colleagues with the purpose of obtaining funding to build a fully operational machine.

It was this machine, or a development of it, that Ada Byron, the poet Byron's only legitimate child, saw on 5th June 1833 on a visit to Babbage's home when she was just seventeen. Her mother, Byron's widow, commented at the time in a letter, "We both went to see the *thinking machine* (for such it seems) last Monday." Lady Byron then went on to say, even though she was quite an accomplished mathematician, "I had but faint glimpses of the principles by which it worked." Ada Byron, it seems, had no such difficulty. Her friend, Sophia Frend, records in her memoirs, "Miss Byron, young as she was, understood its working, and saw the great beauty of the invention."[10]

However, despite receiving Government grants to build this machine, for a variety of reasons this project was never completed in Babbage's lifetime[11]. A partial reason for this is that in 1833 Babbage had what must be regarded as one of the greatest leaps in imagination in the whole history of human learning. He conceived of a machine with the power "to combine together general symbols in successions of unlimited variety and extent", which would embody "the science of general reasoning".[12] Babbage was thus attempting to realize Leibniz's dream with what he called the Analytical Engine because it covered the whole field of analysis in mathematics, an extension of what we called calculus at school.

The key difference between the Analytical Engine and the Difference Engine was that the former was general-purpose, while the latter was special-purpose. The Difference Engine could only work with a particular set of problems, albeit quite common ones. For many expressions, such as logarithmic, exponential, and trigonometric, can be expressed in terms of a polynomial series, whose successive terms tend to zero. So it is possible to use the mathematical method of differences as an aid in calculating these expressions, calculating them to as many decimal places as are needed.

However, the design of the Analytical Engine *does* take us a step nearer to Wholeness and the Truth, to a universal science of reason. Rather than incorporating a particular algorithm into the structure of the machine, Babbage envisaged a universal machine that could handle a wide variety of different algorithms, not unlike the modern computer.

The Analytic Engine consisted of two principal parts. The first was a mill that could execute all four basic arithmetical operations directly. It was called a mill on analogy with the cotton mill. The mill was not unlike the central processing unit (CPU) in computers today.

Secondly, there was a store to hold the data being processed, consisting of columns of up to forty discs, each column being equivalent to one storage unit in a modern computer. Unlike the Difference Engine, the discs were able to rotate independently of each other, the angular position of each disc indicating a value of 0 to 9 for the decimal position in the column.

The input, both instructions and data, was provided on punched cards, called Operations and Variable cards, using the card-reading technology of the Jacquard loom. As Ada Lovelace, as Ada Byron was to become,[13] wrote later, "We may say most aptly, that the Analytical Engine weaves algebraic patterns just as the Jacquard-loom weaves flowers and leaves."[14]

Ada Lovelace had one great difficulty in her relationships with her contemporaries: she was highly intelligent. She was able to see in the essence of forms and structures what few of her contemporaries could see, even when what she could see was pointed out to them. One of her biographers, Doris Langley Moore, says that she lived "a hundred years too soon".[15] But this is an underestimate. It is only in recent years, with the identification of Indigo children, that superintelligence has been recognized in a positive light, rather than as a mental disorder.[16] I feel sure that if Ada had been living today, she would have been one of the few people who understood The Principle of Unity, that Wholeness is the union of all opposites.

Ada Lovelace is credited with being the first computer programmer. But she wasn't really. In describing the Analytical Engine, Babbage, himself, had to sketch out elementary programs illustrating how the machine could be used. So I see Ada more as a technical writer, clearly describing a system that the originator does not necessarily have the skills to do, rather like the relationship between technical writers and some program designers today.

Being ignored by his contemporaries in England, Babbage accepted an invitation from a number of mathematicians and engineers in Italy to present his work in 1840 in Turin.[17] One of the mathematicians there, Luigi Federico Menabrea, subsequently wrote a sketch of the Analytical Engine published in French in Switzerland two years later. This sketch contained the outline of a program to solve a pair of simultaneous equations in two variables, which, I guess, could be called the first published program.[18]

Unbeknownst to Babbage, Ada translated this sketch into English. When Babbage heard about it, he wondered why she had not written an original piece about a machine with which she was 'so intimately acquainted' and suggested that she do so. She jumped at this task, writing notes to the translated sketch that were three times longer than the original. The translation and appended notes were published in *Taylor's Scientific Memoirs* in September 1843, when Ada was just 27.[19]

One of the major aims of her notes was to clear up the many misconceptions that her contemporaries held about the Difference Engine and the Analytical Engine: "No very clear or correct ideas prevail as to the characteristics of each engine, or their respective advantages or disadvantages."[20] Nothing much has changed even today. There is still no clear understand-

ing of the essential differences and similarities between human beings and computers, a critical situation that led me to begin this research project in 1980. For instance, Ray Kurzweil has said that by the end of the century there will no longer be any clear distinction between humans and computers,[21] which is utter nonsense, of course.

Regarding Ada's concerns, she was at pains to point out in Note A, which goes deepest into the essential nature of these two machines, they are two quite different species of animal. As she said, "the Analytical Engine does not occupy common ground with mere 'calculating machines'. It holds a position wholly its own."[22]

Ada could even see that the Analytical Engine was so abstract in the way it handled symbols that it could be used not only in mathematics, but also in musical notation. This led her to say, "The engine might compose elaborate and scientific pieces of music of any degree of complexity or extent". Here, Ada seems a little confused. When Menabrea said, "although it [the Analytical Engine] is not itself the being that reflects, it may yet be considered as the being which executes the conceptions of intelligence,"[23] Ada responded with a cautionary note. She said:

> It is desirable to guard against the possibility of exaggerated ideas that might arise as to the powers of the Analytical Engine. In considering any new subject, there is frequently a tendency, first, to *overrate* what we find to be already interesting or remarkable; and, secondly, as a natural reaction, to *undervalue* the true state of the case, when we do discover that our notions have surpassed those that were really tenable.[24]

The claim that Ada was the first programmer is based on a significant program included in her paper to calculate Bernoulli numbers. She did not create this program from scratch herself; Babbage gave her an outline to work with. However, she did find an error in Babbage's notes that he had not found himself. There was also one mathematical error in Ada's memoir that neither she nor Babbage spotted. This was caused by a misprint in Menabrea's original French article.[25] So while she was both highly intelligent and paid meticulous attention to both detail and conceptual clarity, she was, nevertheless, subject to error like the rest of us.

In the event, the Analytical Engine was never built,[26] and neither did it have much influence on the development of the stored-program computer one hundred years later.[27] Its interest today is to show that we are still sleepwalkers, having very little understanding of the machine that was invented over half a century ago.

It was to be one hundred years before the first programmable computer actually came to built, not in the USA or in the UK, as many believe, but in Germany. In Berlin, Konrad Zuse applied to patent an electromechanical automatic calculator in 1936.[28] However, it was not until 1941 that he was able to build the third version of his machine, known as the Z3, today regarded as the world's first general purpose program-controlled computer, the machine being controlled by a paper tape.[29]

Obviously, because of the war, Zuse knew nothing of the work of the Americans at the time, and vice versa. Anyway, the next most notable advance was the Harvard Mark I, again an electromechanical device, designed by Howard Aitken and built by IBM. This machine, which became operational in May 1944, was also controlled by an external program on paper tape or punched cards, I am not sure which.[30]

Then came a number of electronic machines, which greatly increased the speed of calculation. The most notable of these was the ENIAC (Electronic Numerical Integrator and Computer), designed by the Moore School of Electrical Engineering at the University of Pennsylvania.[31] The ENIAC, which contained 19,000 valves, became operational in the autumn of 1945.[32]

However, the ENIAC was not controlled by a paper tape or cards. It was necessary to set up plugboards and banks of switches, even sometimes to reroute cables.[33] So setting up these machines was a time-consuming process.

What was needed was a *stored-program* computer, in which the instructions could be stored in the memory of the machine, along with the data being processed. The implications of this epoch-making change is that a program could not only operate on numerical data, but also read and modify itself during the course of execution, a subject that we looked at in the previous chapter. For this issue is key to understanding the essential differences between human beings and computers.

It is not clear exactly when the idea of a stored-program computer first appeared in human consciousness. Ada Lovelace refers in her memoir to the Analytical Engine being able to operate on symbolic data as well as numerical data.[34] But it is not clear whether she envisaged that this symbolic data is the program itself, the symbols of mathematics, such as *Mathematica* is able to handle today, or something else entirely. Neither is it possible to determine whether Babbage conceived of the idea of a stored program.[35]

Both Konrad Zuse and Alan Turing seem to have seen the possibility of a stored program in the papers they wrote in 1936. However, the idea seems to have fully emerged within the ENIAC group at the Moore School in 1944 or 1945.[36] They envisaged a computer called the EDVAC (Electronic Discrete Variable Arithmetic Computer), which was described in a draft paper, apparently never finished, by the eminent mathematician, John von Neumann, published on 30th June 1945.[37] In 1946, Alan Turing, who had met von Neumann during the war, also proposed that the National Physical Laboratory in England build a stored-program computer called an Automatic Computing Engine (ACE).

However, neither the EDVAC nor the ACE were the first stored-program computers to be built. This honour went to the MADM (Manchester Automatic Digital Machine), constructed at the University of Manchester in England in 1948 by a team led by M. H. A. Newman.[38] However, this was more a prototype than a practical machine. The first practical

stored-program computer was the EDSAC (Electronic Delay Storage Automatic Calculator), built by a team led by Maurice Wilkes at the University of Cambridge, also in England.[39] This machine ran its first program on 6th May 1949. In 2 minutes 35 seconds, it computed a table of squares from 0 to 99 and printed out the results.[40] The Computer Age was truly born.

The first commercial stored-program computer to be built was the UNIVAC (Universal Automatic Computer), which was bought by the US Census Bureau in 1951 to process the mass of data that they were being deluged by. The UNIVAC was designed by J. Presper Eckert and John W. Mauchly, who had worked at the Moore School, but who left to form a fledgling company bought by Remington Rand, the electric razor company.[41]

But it was not until the presidential election between Dwight D. Eisenhower and Adlai E. Stephenson the following year that the capabilities of the computer caught the public imagination. As the results of the election were being processed at election headquarters, someone in the media asked the UNIVAC for a prediction of the result. The machine predicted a landslide victory for Eisenhower, something that no one believed so the prediction was not broadcast immediately.[42] It was only after the result was published that the media rather sheepishly admitted that the machine seemed to know more about the future than they human beings did, thus leading to much hype around the computer, which even today has not died down.

With the birth of the stored-program computer, we can say that we have now reached the point B in the growth curve for computer technology. The complexity of computer hardware then began to accelerate due to significant advances in electronics. This was aided and abetted by the growth of structure in databases, programming, and modelling, which we look at in later sections in this chapter.

The key component in the first computers was the thermionic valve, a device that was invented in 1904 by J. A. Fleming and improved on two years later by Lee de Forest.[43] The early computers were thus very large machines, filling whole rooms.

Then in 1947, a team of engineers at Bell Laboratories invented the transistor, a device that was smaller, more reliable, and used less power than the valve[44]. It was this invention, above all, that was to make the computer a practical business machine. This was well demonstrated by the second generation of computers that appeared in the late 1950s and early 60s, of which the most successful was IBM's 1401.

The next major change occurred in the 1960s when engineers discovered that computer components could be assembled together on or within a continuous substrate. The integrated circuit (IC), commonly known as the silicon chip, came into being. Since then, these integrated circuits have been further compacted in an accelerating manner giving rise to even more advanced technologies called large-scale integration (LSI), very large-scale integration (VLSI), and very high-speed integrated circuit (VHSIC). As each change in technology has

been introduced by the data-processing industry, a new generation of computers has emerged.

The most noticeable effect of these electronic developments has been that each generation of computers has been smaller than the previous one. This phenomenon has come to be known as Moore's law. In 1965, Gordon E. Moore, cofounder of Intel the chip manufacturer, wrote an article called 'Cramming more components onto integrated circuits' for the journal *Electronics*, in which he stated that the number of transistors on a silicon chip doubles every year.[45] This came about because the surface area of a transistor was halving every twelve months. In 1975, Moore revised this trend to two years, or, as some say, eighteen months, it doesn't matter which. Table 7.1 provides a set of figures that shows Moore's law at work with the chips manufactured by Intel.[46]

Year	Transistors on Intel chip
1972	3,500
1974	6,000
1978	29,000
1982	134,000
1985	275,000
1989	1,200,000
1993	3,100,000
1995	5,500,000
1997	7,500,000

Table 7.1: *Illustration of Moore's law*

Now the overall effect of this phenomenon is twofold. Not only has the number of chips doubled every year or two, the speed of the circuit has also doubled, for the simple reason that the signals between the components have a shorter distance to travel. In business terms, this has meant that the price-performance of computers—the amount of performance available per dollar—has effectively quadrupled every year or two, which has, of course, greatly expanded the marketplace.

These characteristics of computers have led to the economics of the data processing industry being quite different from any other. To give an analogy, if the car industry had developed in a similar manner, cars would now cost just a few dollars and be able to travel at several times the speed of sound. (How the passengers would fit in is quite another matter!)

But as nearly everyone now recognizes, silicon-based technology is limited. This fact was brought vividly home to me in the mid-seventies when I attended an IBM hundred-per-cent club conference, for those salespersons and managers who had reached their sales quota for the year. One of the speakers at the conference held his hands in front of him about 30 cen-

timetres or one foot apart and said, "this is one light nanosecond". It takes 10^{-9} of a second for light to travel this distance. So miniaturization cannot continue forever along this line.

Gordon Moore is well aware of this fact for he told a meeting of the world's top chip designers and engineers on 10th February 2003, "No exponential is forever." But he then went on to say, "Your job is to delay forever."[47]

Well is this possible? Will quantum computers, molecular electronics, nanotechnology, or other exotic technologies one day replace conventional silicon chips, leading to the beginning of another growth curve, not unlike the envelope of growth curves in Figure 6.9 on page 541 in Chapter 6, 'A Holistic Theory of Evolution'.

Maybe. But this is not the point. The capabilities of the computer are not a function just of raw processing power. Neither is the cognitive power of the computer viz-à-viz human beings dependent on the development of ever more complex algorithms, leading to what James Martin has called 'alien intelligence', something that human beings cannot understand.[48]

The principal reason for this is that computers are essentially symbol processing machines. They can only handle what is explicit—the appearance of things—not the implicit essence of forms and structures, which, in human terms, we often call intuition, in contrast to reason. So the growth curve of computer technology is inherently limited by the capabilities of symbol processing.

To continue this theme, who needs even the processing power that is projected to become available in the next few years. I was using a four-year old computer to write this chapter in 2007. At the time I bought it, it was the top of its range. Today, the top of the range is a computer running many times faster. But do I really need this additional processing power? The computer I have satisfies my needs quite well, although I really need some more memory.

Some CIOs have also recognized this situation. Do people whose primary use of the computer is word processing really need a 10 GHz machine, or even a 2 GHz one? Of course, professional graphics designers, creating and editing still or moving pictures in two and three dimensions, make much use of the enhanced computing power, as do computer programmers and scientists. But these people are a minority today.

Growth of program structure

When human beings began programming computers in the 1950s, they had to do so directly in the machine's instruction set. For instance, a move instruction could be `1011, B` in hexadecimal. Even though this was comparatively high in the hierarchical structure of computers—as we see in Section 'Computer structure' in Chapter 8, 'Limits of Technology' on page 624—this was extremely cumbersome, requiring the programmer to perform many tasks that the machine could do far better, such as calculating addresses.

So assembly programming languages emerged that provided a symbolic representation of the instructions as mnemonics (such as `mov`), which were then translated into the machine's opcodes for execution. Each type of computer had a different set of instructions. So each required its own assembler.

But assemblers did not allow programmers to think in the conceptual terms of the problems that they were endeavouring to solve. So high-level languages began to emerge. Among the first of these languages was FORTRAN (FORmula TRANslation), for scientific applications, developed by John Backus, becoming available from IBM in 1957,[49] and COBOL (Common Business Oriented Language), developed by CODASYL (COmmittee/COnference on DAta SYstems Languages) led by Grace Hopper, for business applications.[50] Because FORTRAN had been developed at IBM, then rapidly becoming the dominant force in the data-processing industry,[51] a group of computer scientists in Europe, seeking a 'universal' computer language, introduced ALGOL (ALGOrithmic Language) in 1958, which was the progenitor of PASCAL, named after Blaise Pascal (1623–62), among other languages.[52]

Two other significant developments arose at this time. First, in the example of FORTRAN, functions were provided to perform standard algorithms, such as returning the maximum of a list of numbers and calculating the sine of an angle. This was an extremely important development, the beginning of creating 'building blocks' for program developers. Comparing the task of program development with designing and building a house, pioneering programmers did not have a set of bricks, tiles, doors, windows, light switches, and so on, with which to construct their programs. They needed to create each of these themselves, often 'reinventing the wheel'.

Secondly, operating systems, which provided overall control of the resources of the computer system, became available. The first one I worked with in 1964 was IBSYS, the OS for IBM's 7094, its most powerful computer at the time, which had initially been developed by an IBM customer: General Motors' research division,[53] which was not an uncommon happening. Sometimes customers of computer manufacturers were ahead both of their suppliers and the theoreticians in academia.

Then in 1966, IBM introduced OS/360, designed to run on its System/360 range of computers, so-named to indicate a sense of wholeness, for there are 360 degrees in a circle. By some magic, this later evolved into System/370 and System/390. In terms of reducing the semantic gap between the machine and the human programmer, one of the most important features of OS/360 was the 'device independence' of input/output (I/O). To some extent, programs could be written without specifying the hardware device that they were reading from or writing to. This could be specified in data definition (DD) statements in the Job Control Language (JCL), interpreted at run time. So sequential data, for instance, could be read from cards, tape, or disk without changing the program.

This was all very fine. But there was still one outstanding problem that inhibited the growth of structure of computer programs. Programs are generally seen as a sequence of instructions to the computer. However, sometimes this sequence needs to change depending on some condition or other. To allow for such situations, computers include jump or branch instructions in their instruction set. In high-level languages, these switching instructions were generally implemented by a `goto` instruction.

However, the use of this instruction could lead programs to look like plates of spaghetti, which were notoriously difficult to debug and maintain. The breakthrough came in 1966 in one of the most important papers in the history of the data-processing industry. In that year, Corrado Böhm and Giuseppe Jacopini from Italy wrote a paper, today known as the 'Structured Program Theorem', in which they proved mathematically that all programs could be written with just three control structures.[54]

The first is simply a sequence of instruction executing one after another. In terms of structure, these sequences could be grouped together in functions or subprograms, executed sequentially. The key concept here is a process box, with only one input and output, as shown in Figure 7.1:[55]

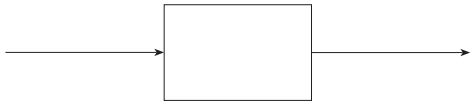


Figure 7.1: *Programming process box*

Secondly, which was the next instruction block to be executed could be selected by testing a Boolean variable, as in Figure 7.2.[56] Note that this is also a process box with one input and output. Thus the process blocks in this diagram could also be selection blocks in a nested, hierarchical fashion, most simply implemented as if-then-else instructions in high-level languages, or just `if-else` in C. When a choice has to be made between several options, to avoid too many levels of nesting, C provides a `switch-case` statement. There are similar facilities in other high-level languages.

The third control structure is the loop or iteration block, as shown in Figure 7.3.[57] In C, this construct is implemented with `for`, `while`, and `do` blocks. However, on their own, these statements are not sufficient to avoid the use of the `goto`, which actually exists in C. Sometimes, it is necessary to interrupt the normal flow of control. In C this is done with a `break` statement, which causes an exit from the innermost loop or `switch` statement, and `continue` statement, which jumps directly to the next iteration of the loop.

However, like so many ground-breaking ventures in all fields, this paper was initially ignored by the mainstream of the data-processing industry. It was left to a few pioneering individuals to promote the benefits of structured or modular programming. Foremost among these was Edsger W. Dijkstra from the Netherlands who wrote a famous letter in 1968 called

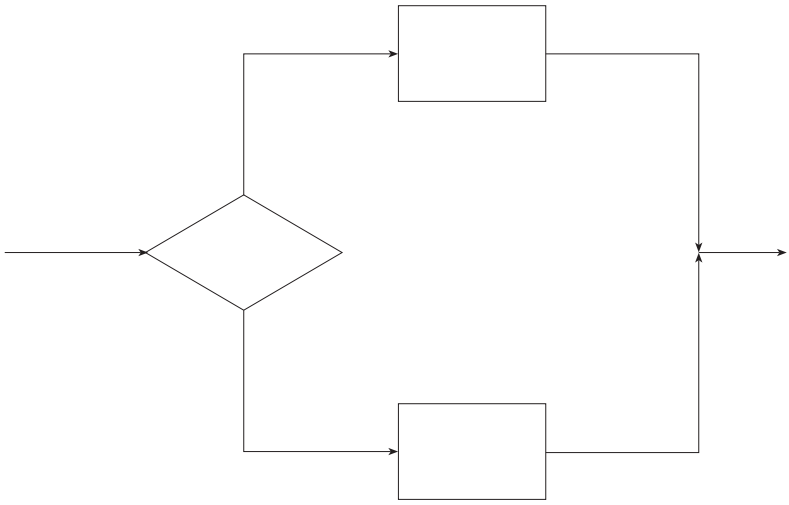


Figure 7.2: *Programming selection block*

'Go To Statement Considered Harmful', in which he described the 'disastrous effects' of the go to statement, and that it should be abolished from all-high level languages.[58]

One who took up the challenge of creating structured programming languages was Niklaus Wirth from Switzerland, the chief designer of Pascal, and several other programming languages. To show how these languages could be used, in 1971, Wirth published a classic paper called 'Program Development by Stepwise Refinement'.[59] Using the example of the 8-Queens problem on a chessboard, he showed how programs could begin with a very high-level description, the program then being developed in a sequence of refinement steps. "In each step, one or more instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of an underlying computer or programming language."

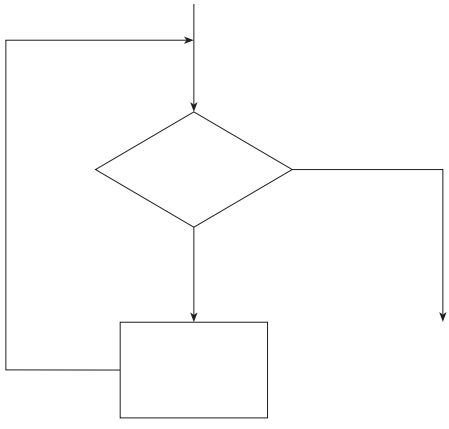


Figure 7.3: *Programming iteration block*

This is a splendid example of human learning as a structured evolutionary process. The whole is visualized from the start in simple terms, which then evolve through a process of divergence and convergence until the structure is complete. A familiar example from the biosphere is the development of an organism from a single fertilized cell, as we see in Figure 6.4 on page 532 in Chapter 6, 'A Holistic Theory of Evolution'. The Unified Relationships Theory and implicate relational logic have become manifest by just such a holistic evolutionary process.

Another major contributor to the development of structured programming was C. A. R. Hoare, from the UK, who was the co-author with Dijkstra and Ole-Johan Dahl, from Norway, of an influential book *Structured Programming*, published in 1972. Dahl, along with Kristen Nygaard from the Norwegian Computing Center, was the originator of the next great step in the growth of program structures. But before we move on to this, it is interesting to note that all these pioneers came from various countries in Europe.

From these theoretical and programming foundations, several people, from both sides of the Atlantic, began to develop tools and techniques that could bring structured programming into human consciousness. These included Larry Constantine and Edward Yourdon from the USA, co-authors of *Structured Design*, Michael A. Jackson from the UK, author of *Principles of Program Design* and initiator of Jackson Structured Programming (JSP), and Glenford J. Myers from IBM, author of *Reliable Software Through Composite Design*, all first published in 1975. In the late 1970s, IBM also introduced a tool called Structured Programming Facility (SPF) running under the Timesharing Option (TSO) of MVS, the then latest version of its mainframe operating system.

But back to the Norwegian Computing Center. As early as the mid 1960s, Kristen Nygaard and Ole-Johan Dahl, together with Bjørn Myhrhaug, designed a computer language called SIMULA (SIMUlation LAnguage)[60] intended to describe complex dynamic systems.[61] The key concepts in SIMULA are class and object, where objects are instances of classes, just as entities are instances of classes in IRL. It is these concepts that gave rise to object-oriented programming and modelling techniques in computer science, bringing these processes close to the way humans think and hence to how the Universe is designed. However, computer scientists are not generally aware that they are implicitly using Integral Relational Logic in this process.

At its simplest, the concept of class can be considered as an extension of the basic [passive] data types in programming languages that match to data types in the hardware. In C, examples are `char`, 'K'; `int`, 7564; and `float`, 3.142857. These can be extended into arrays, called strings when the data type is `char`. Data types can be further expanded in C by the use of `typedef`, which extends the identifiers that can be used to denote data types, making them more meaningful in specific situations, and `struct`, which allows for complex data structures

to be defined. Explicitly stating the data types of variables in programs is an important factor in making them more robust. (Active data types in C are operators, such as +, and functions, the 'heart and soul' of the language, which can sometimes be treated in a similar manner to passive data types. For instance, they can be passed as arguments to other functions.)

In an object-oriented programming language, a class includes not only a set of data attributes, but also a set of allowable operations on that data, called member functions in C++, for instance.[62] "A class is a set of objects that share a common conceptual basis."[63] Examples are **book** in a library lending system and **flight** in an airline booking system. To take a simpler example, **rectangle** could be a class with data attributes breadth and height. Member functions could calculate the area and circumference of a particular instance of rectangle. If other attributes are added, such as position, other functions could move, rotate, or draw a rectangle in a drawing program. Data attributes thus determine the shared semantics of a set of objects and functions define their behaviour. So classes contain both passive and active data types, described in more detail on page 625 in Chapter 8, 'Limits of Technology'.

One great benefit of classes is that they provide the basic building blocks for the development of information systems. Once one programmer creates a class, other programmers can use or reuse it, greatly increasing productivity. So unlike the early days of programming, developers no longer need to create their own bricks and tiles in building their 'houses'; they have masses of class libraries available to them. It is therefore not surprising that developers today are inspired by the architect Christopher Alexander's pattern language[64] in developing holistic and 'ecological' systems.[65]

Classes can be reused in this way because programmers do not need to be concerned with how classes are actually implemented. In principle, at least, they are provided with an external interface that is separated from implementation details, called encapsulation or information hiding. Another major benefit of object-oriented programming is that classes form hierarchical structures. Data attributes and functions in a high-level class can be inherited by subclasses lower down the hierarchy. For instance, class **polygon** could have an attribute area, which could be calculated in different ways depending on the type of polygon. Programmers can thus work at whatever level of abstraction suits them, leaving the system to determine what algorithms should be used in which particular instance.

The object-oriented paradigm has also had a profound effect on the way that human beings communicate with computers. The Macintosh desktop metaphor was partially inspired by the Smalltalk programming environment, which built on ideas in SIMULA, developed by a group of researchers led by Alan Kay at Xerox Palo Alto Research Center (PARC) in the 1970s. For instance, icons on the desktop are objects with various attributes, such a creator and type, and functions, such as open. In Word, a character can have various attributes, such as **bold** or underline.

But the Apple Macintosh introduced an even-bigger change in the human interface. In the 1970s, computer programs tended to be menu-driven; users were given a list of options from which to choose. The computer was in charge. This changed with the Mac, which introduced an event-driven approach, giving the user control over the program, in theory, at least. Application programs were written to respond to different events, such as characters entered from a keyboard or mouse clicks on a button or menu.

The practice, however, falls some way short of this ideal. Most particularly, I struggle today with Microsoft products. Not only have they become so complex that I no longer fully understand their underlying conceptual model, Microsoft programmers try to second-guess what I want to do, automating some actions, which I then need to spend time undoing. In the years since I bought my first Mac in 1986, I have used three programs for writing and publishing: Microsoft Word and Adobe FrameMaker and InDesign. But none of these programs on their own provide me with all the tools I need to write this book with full ease-of-use and functionality. Because of the intense competitiveness of the marketplace, evolutionary convergence is still struggling with divergence, leaving users with a set of fragmented tools that do not fully meet their integrative requirements.

Growth of systems modelling structures

Writing computer programs is but one part of designing and developing the information systems that modern enterprises need to manage the complexity of their business organizations. What is needed here is a fully integrated model of all the processes taking place within an organization and of the entities on which these processes operate, independent of whether these processes are performed by machine or human being.

Not surprisingly, the modelling methods and tools that have evolved over the years show the same growth of structure as any other evolutionary process. These have led to the Unified Relationships Theory, which provides a comprehensive model of the psychodynamics of the whole of society, including the modeller's own creative thought processes, and hence of the whole of evolution from Alpha to Omega. Some of these modelling techniques became extinct species, unable to evolve any further, while others have been the basis for further development or have survived more or less intact as primitive species.

One of the extinct species is Business Systems Planning (BSP), which was one of the triggers that led to Integral Relational Logic, as I describe on page 19 in Part I, 'Integral Relational Logic'. As I explain, the difficulty of representing an APL program dynamically creating and destroying other programs in a BSP process-entity matrix was a major factor that led to me to discover the essential difference between human beings and computers, for this led me to see that computers cannot program themselves without human intervention, as I describe in the Section 'Computer-driven program development' in Chapter 8, 'Limits of

Technology' on page 633.

On the other hand, one basic species that is still in use today is the flowchart, introduced by Frank Bunker Gilbreth, an early advocate of scientific management, to members of the American Society of Mechanical Engineers (ASME) in 1921. A flowchart is a

> graphical representation of a process, such as a manufacturing operation or computer operation, indicating the various steps that are taken as the product moves along the production line or the problem moves through the computer. Individual operations can be represented by closed boxes on the flowchart, with arrows between boxes indicating the order in which the steps are taken.[66]

Actually, flowcharts were used both for modelling systems and programs, and suffered from the same problems as the programs themselves using the `goto` statement. The rhombus symbol was used to represent choices in flowcharts, leading them to great complexity, often stretching over many sheets of paper, with links between the sheets. As computers still had very limited graphical facilities in the 1960s, these flowcharts were very difficult to maintain and could easily get out-of-date.

When flowcharts got really complicated with multiple choices to be made, these could be elegantly represented in a concise tabular form called a decision table, which corresponded to `if-then-else` and `switch-case` statements in a structured programming language. Here is a simple example for a printer troubleshooter.[67]

		Rules							
Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognized	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Table 7.2: *A decision table*

Structured programming also led naturally to a set of structured system analysis and design methodologies. Many innovators contributed to these techniques, including those already mentioned: Larry Constantine, who invented data flow diagrams, Ed Yourdon, Glenford Myers, and Michael A. Jackson. Others who also made major contributions were Wayne Stevens,[68] Peter Checkland,[69] and Chris Gane and Trish Sarson.[70]

One of the methods that arose from these theoretical foundations was Structured System Analysis and Design Method (SSADM) developed by Learmouth and Burchett Management Systems (LBMS) in conjunction with the UK government's Central Computer and Telecommunications Agency (CCTA) in the early 1980s. Such a structured approach was absolutely

essential because the UK government was the largest user of information systems in the UK and had suffered some very expensive failures with taxpayers' money in the 1970s, although I do not remember the details.

SSADM is based on a staged approach to systems analysis and design, beginning with a feasibility study, continuing with requirements analysis and specification, which leads to a logical systems specification, which can then be physically implemented in a particular computer system.[71] There are three basic techniques in SSADM: logical data modelling, creating a data model of entities and the relationships between them; data flow modelling, describing the way data moves through a system; and entity behaviour modelling, documenting the events that affect each entity and the sequence in which these events occur.[72]

Another interesting structured design technique was developed by the U.S. Air Force Program for Integrated Computer Aided Manufacturing (ICAM), which sought to increase manufacturing productivity through systematic application of computer technology. This technique is called IDEF, initially standing for ICAM DEFinition language, later becoming Integration DEFinition, when the technique became standardized. IDEF was based on Structured Analysis and Design Technique (SADT), originally developed by Douglas T. Ross of SofTech in 1972.[73]

There were initially three basic modelling techniques in IDEF:[74]

IDEF0, used to produce a 'function model'. A function model is a structured representation of the functions, activities or processes within the modelled system or subject area.

IDEF1, used to produce an 'information model'. An information model represents the structure and semantics of information within the modelled system or subject area.

IDEF2, used to produce a 'dynamics model'. A dynamics model represents the time-varying behavioural characteristics of the modelled system or subject area.

IDEF0 is perhaps the most interesting. It is a strictly hierarchical modelling method, based on a decomposition technique. At the highest level is a diagram called A0, which encapsulates the function of the entire system in a single box, such as Figure 7.4:

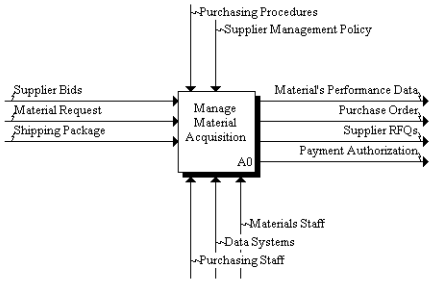


Figure 7.4: *IDEF0 A0 top-level chart*

The horizontal lines mark inputs and outputs. Arrows connected to the top side of the box are controls, specifying the conditions required for the function to produce correct outputs.

Arrows at the bottom represent mechanisms, identifying some of the means that support the execution of the function.[75] An A0 box would then be decomposed into A1, A2, A3, etc, and then into A11, A12, A13, etc.Today, IDEF has been extended up to IDEF14,[76] although I do not know to what extent the additional modelling techniques are being used.

With the development of these structured analysis and structured design methods, there was a great need to use the computer itself to help with the documentation of systems. In the early days, systems analysts suffered from the problem that you never see a tailor in a new suit of clothes. This problem began to be solved in the early 1980s with the introduction of computer-aided software engineering (CASE) tools, a term coined by Nastec Corporation in 1982. Today, there are a multitude of different CASE tools covering a wide range of modelling techniques. However, there is not yet an educational tool for IRL, which enables the modeller to model her or his own thought processes.

The introduction of object-oriented programming languages naturally led into object-oriented systems analysis and design techniques, taking these methods as close as possible to the underlying structure of the mind and hence of the Universe. One of the pioneers in this field was Peter Coad, founder of Object International in 1986,[77] and co-creator with Edward Yourdon, a structured design pioneer, of the Coad/Yourdon of object-oriented analysis[78] and design.[79]

Other pioneers were Grady Booch, who developed the Booch method[80] at Rational Software around 1990, and James Rumbaugh, who developed the Object Modeling Technique (OMT)[81] at General Electric Research and Development Center.[82] Rumbaugh joined Booch at Rational Software in 1994 to develop the Unified Modeling Language (UML).[83] They were joined there by Ivar Jabobson, who invented use cases as a way to specify functional software requirements when working for Ericsson in Sweden in the 1980s.[84] In 1987, he had left Ericsson to set up Objective Systems, a company where he created Objectory also known as Object-Oriented Software Engineering (OOSE).[85] Ericsson purchased a substantial stake in this company in 1991, renaming it Objectory AB, and then sold it to Rational Software in 1995.[86]

It was a natural evolutionary process for all these different object-oriented modelling tools to be integrated into a single system, for they were all addressing essentially the same problem. After twenty-five years struggling to develop practical modelling tools, IBM bought Rational Software in 2003 for 2.3 billion dollars,[87] which shows how central to the management of the complexity of modern corporations these modelling tools have become. Not unnaturally, the UML has become the de facto information systems modelling tool in business today.

Although the designers are not aware of this fact, the UML is implicitly based on IRL, which models both the structure of the Rational Unified Process (RUP), an iterative software development process covering the software development life cycle,[88] and beings in the UML itself, which are called things, relationships, and diagrams. Things and relationships corre-

spond to nodes and arcs in a graph in mathematics, displayed in diagrams. So the UML operates at a very high level of abstraction.

Things are classified as structural, behavioural, grouping, and annotational things, the first two being the nouns and verbs of the language. The other two are organizational and explanatory elements. Relationships are the basic relational building blocks in the UML. There are four of them: dependency, association, generalization, and realization. The overall architecture of the UML can be looked at through five interlocking views, documented in nine or more different types of diagram, of which the class model is the most important, as described in Chapter 1, 'Starting Afresh at the Very Beginning'.

There is no need to go any further into these levels of complexity, for the purpose of this chapter is illustrate the growth of structure, with the UML representing a very high point in the growth of the complexity of business information systems. This evolutionary process is not only modelled in Integral Relational Logic, but this growth of complexity-consciousness has also led to IRL's development as a universal modelling method of the utmost abstraction and generality.

Growth of data storage structures

Not that the UML was the immediate precursor to IRL, which began to emerge in consciousness in 1980, some fifteen years before UML. It is only in retrospect that some of the structures of UML have been used to describe IRL. Actually, it was Ted Codd's relational model of data that was IRL's immediate predecessor, as explained in Part I, 'Integral Relational Logic'. So in the next section, we look briefly how this modelling technique fits into the growth of data structures, perhaps the most fundamental of all growth processes in human society, for this relates to how we have long organized our records, giving birth to the sense of linear time.

However, before we look at how we organize our records today, it is useful to put modern storage technology into its historical perspective. As human memory is both fragile and personal, we have used a variety of materials over the years to store our records, in the widest sense of this word. The earliest records we have discovered from about 5,000 years ago in Mesopotamia were inscribed on clay tablets before they were fully dry. This material was to be in use for about two thousand years alongside papyrus, before the latter eventually won the day.[89]

Papyrus, which was in plentiful supply along the Nile, was, of course, the primary writing material of the ancient Egyptians and later of the Greeks and Romans[90]. However, it had one major disadvantage: it could not be folded without cracking.

So records on papyrus were written on rolls, which could get quite long. This meant that the user had to rewind the roll after reading it.[91] It was also not easy to find a specific reference

on the roll. The papyrus roll, which was in regular use until the ninth century AD and sporadically until the twelfth century,[92] was thus rather like a magnetic tape in the early days of the data-processing industry or a cassette tape in the music industry.

The major alternative to papyrus during ancient times was parchment, invented about 190 BC, although raw leather had been used as a writing material long before this date. It seems that Ptolemy V of Egypt was afraid that the library of Eumenes II of Pergamum in Greece (now in Turkey) would outstrip his own book collections at Alexandria.[93] So Eumenes invented parchment (derived from *Pergamum*), a greatly refined form of leather made from sheep, goat, or calf skins, the last being called vellum.[94]

Parchment had several advantages over papyrus. "A sheet of parchment could be cut in a size larger than a sheet of papyrus; it was flexible and durable, and it could better receive writing on both sides."[95] Furthermore, it could be folded and stitched together to form a codex, not unlike our modern books. The codex was the world's first random access device, because it was possible to go directly to a page of information, not possible with a roll of papyrus, which could only be read sequentially.

Eventually paper was to replace both papyrus and parchment, although it took a long time in coming. Paper was invented in China about 105 AD. However, it did not reach the Middle East until the eighth century and it was not in common use in Europe until the fifteen century, when the invention of the printing press greatly increased its demand.[96]

For most of this time, paper was made from rags. It was not until the beginning of the nineteenth century that wood pulp became the most common source of fibre for paper-making. At about this time, the first paper-making machine was invented that could use this wood pulp to make a continuous sheet of paper wound round cylinders. Before this time, each sheet of paper was individually made by hand.[97]

During all these millennia, it was human beings who both wrote and read records on these various materials. Then, in the 1880s, this situation changed radically. During this decade, Herman Hollerith invented a machine to punch and count cards, which he used in the 1890 census in the USA.[98]

This was not entirely a new invention. At the beginning of the nineteenth century, Joseph-Marie Jacquard had enhanced the punched card, invented fifty year earlier, to automatically control the patterns of weaving of cloth in a loom.[99] And Charles Babbage adopted this invention as an input/output device for his analytical machine, which never actually got built in his lifetime.

Hollerith's machine was so successful that in 1896 he organized the Tabulating Machine Company, incorporated in New York, to manufacture these machines; through subsequent mergers this company grew into the International Business Machines Corporation (IBM).[100]

Punched cards were to play a key role in business data processing until well into the 1970s. They were eventually replaced as a storage medium by magnetic devices, such as tapes and disks, and as an input medium by MICR and OCR machines, human beings at visual display units (VDUs), and by computers being directly connected to each other in networks.

But this was not the only thing that changed at this time. During the 1950s, the stored-program computer began to be used as a business machine. This invention introduced the most fundamental change in the storage of our records in five thousand years. For the computer is an extension of our minds. And by storing our records in computers, we were, in effect, extending our collective memory.

This was a gigantic step. For while some people can remember a large number of telephone numbers, for instance, for myself, I can barely remember more than about five at a time. And how many of us can remember a whole directory of telephone numbers, never mind the vast quantities of other records we keep in our computers?

Of course, if we were to make use of this extended memory, we needed to find ways to organize and access it. At first, this facility was fairly limited because records were stored on magnetic tape, a sequential medium. So tapes had the same limitations that papyrus rolls had had two or three thousand years earlier.

This situation led to computers getting quite a bad name. For when customers rang companies enquiring about their orders or bills, they were often told something like, "Sorry, I don't have that information, it's on the computer."

What changed this was the invention of random access devices, like disks and drums, which IBM generically called direct access storage devices (DASD). With these devices, it was possible to go directly to a particular record of interest, thus enabling customer service personnel and other staff immediate access to the stored data.

The first computer disk storage system, the RAMAC (random access), was invented by IBM in 1956. It was displayed at the 1958 World's Fair in Brussels, where it was used to answer questions on world history in ten languages.[101] But it was not until the mid sixties that it became a practical proposition in business. Even then, it was severely limited. The first (removable) disk drive I worked with around this time was the IBM 2311, with the capacity of 7 MB, about four floppy disks familiar in the 1980s and 90s. Since then the capacity of random access storage devices has grown exponentially, so that today the capacity of these disks is measured in gigabytes and even terabytes.

Growth of data structures

When programmers began to use random access devices to store data, they inevitably needed to work closely with the machine. At the basic physical level, random access devices are divid-

ed into sections with various names such as blocks, clusters, sectors, and pages. These sections typically range in size between 512 and 4096 bytes, corresponding to pages in a book.

It is the job of the disk management software to organize these pages, most commonly through a hierarchical directory structure familiar to us on the desktops of our personal computers. It is this directory structure that is being created when we format our hard disks, with HFS or HFS+ on Mac OS, FAT16, FAT32, or NTFS on Windows, HPFS on OS/2, and UFS on UNIX, including Mac OS X.[102] At this level, the disk stores folders and files, analogous to the filing systems we use to organize our papers. Hence the desktop metaphor introduced in the Macintosh in 1983 or thereabouts.

However, what we are really interested in is the structure of the data in these files. In the 1960s, a number of access methods were developed to facilitate this process. The ones I was familiar with came with OS/360. There were three main types.

The simplest, SAM (sequential access method), merely mimicked on disk the sequential processing of a tape drive or card deck. At the opposite end of the spectrum, there was DAM (direct access method), which provided random access to records typically using some form of hashing algorithm, determined by the programmer. ISAM (index sequential access method) came between these two extremes. It sought to get the best of both worlds: to enable programs to process files sequentially when this was necessary, while at the same time providing direct access to individual records through a set of hierarchical indexes. However, these access methods were still more oriented to the machine than the programmer. What was needed was for the human view of the data to be completely independent of the data structures in the machine.

Thus the concept of data independence arose, which can be seen as an extension of an earlier concept: device independence, as mentioned on page 581. Data independence goes one step further than this. The aim is to develop a system whereby programs are independent of the way that the data is stored and accessed. In this way, databases can be designed so that they provide a flexible and accurate representation of the changing nature of the business. If it is decided to change these data structures and access methods, there is less need to change programs or recompile them.[103]

The means by which data independence is provided is through database management systems (DBMS). The purpose of the DBMS is to provide a single repository of all the data in an organization so that it can be managed as a coherent whole. This is not to say that the database is physically in one place. Databases can be distributed across an organization. But the aim is to make all these physical databases look like a single unit.

The overall effect of a DBMS is that files of data do not belong to individual departments or the applications that automate the processes in these departments. A central role of a

DBMS is therefore to provide access to this integrated repository to all authorized users in the organization, or, in these days of the Internet, to users throughout the world.

To perform this integrating activity, two key roles emerged in many organizations: database administrator and data administrator. The former is responsible for technical management of the DBMS, itself, while the latter takes a more business-oriented view of the overall information structure of an organization.

The network approach

The pioneering figure in database management systems was Charles Bachman, who worked for General Electric in the 1960s and later Honeywell. He developed a DBMS called Integrated Data Store (IDS).

The word *integrated* is interesting here. It indicates that right from the outset of random access devices, the emphasis was on integrating data that had previously been held in separate files. It is a clear example of the convergent tendencies in evolution.

The basic data structure in IDS, as Bachman explained in a seminal article published in 1965,[104] was the chain, which permitted groups of related records to be linked together, as Figure 7.5 shows. Each record in the chain contained a pointer to the next record, a facility that was quite impossible until the introduction of random access devices.

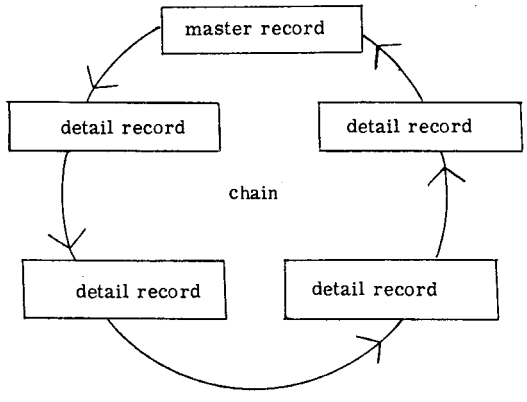


Figure 7.5: *IDS chain of records*

One of the first uses of IDS was in the management of engineering parts lists, commonly called 'bill of materials', the foundation of a manufacturing business. Typically, the products that businesses sell consist of a multitude of parts and subassemblies, which can be incorporated into many different products. Organizing and managing all these records was quite a challenge, especially as the number of products and their subassemblies could be in the tens of thousands.

This network approach to data management was then picked up by the Data Base Task Group (DBTG) of CODASYL (Conference On DAta SYstems Languages), the organization that had introduced the programming language COBOL in 1959.

The DBTG did not set out to develop a DBMS per se. Rather they sought to define a Data Description Language (DDL), which could describe the structure of a database independent of any programming language, and a Data Manipulation Language (DML), which could be embedded in a host language such as COBOL or PL/1, for accessing the records in the data-

base. The aim was to set an industry standard that could provide the foundation for the development of DBMSs.

Charles Bachman was naturally an influential member of this committee, which included representatives from many of the leading computer companies at the time, including IBM, Honeywell, Burroughs, and Univac, and some leading business corporations, such as General Motors and Bell Telephones.

The DBTG produced two reports, in 1969 and 1971, defining what they hoped would become the industry standard.[105] During the 1970s, several DBMSs were developed based on the DBTG proposal, of which the leading exemplar was Integrated Database Management System (IDMS) from Cullinet Software.[106]

In essence, data structures in DBTG are very simple. They contain just two constructs, records and a set of links connecting them, called either a DBTG or CODASYL set, to distinguish these sets from sets in mathematics. To use an example given by Chris Date, a leading authority on DBMSs, suppose we wish to record information about concerts, which works are performed at each concert, and who the composer of each work is.

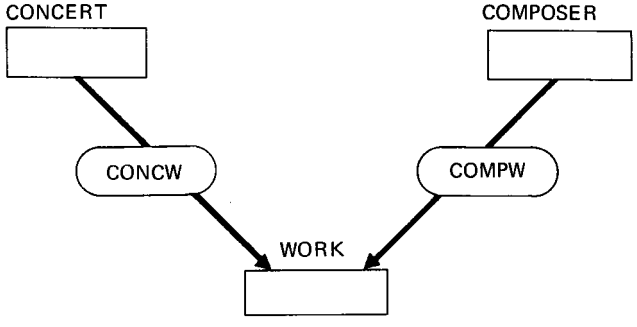
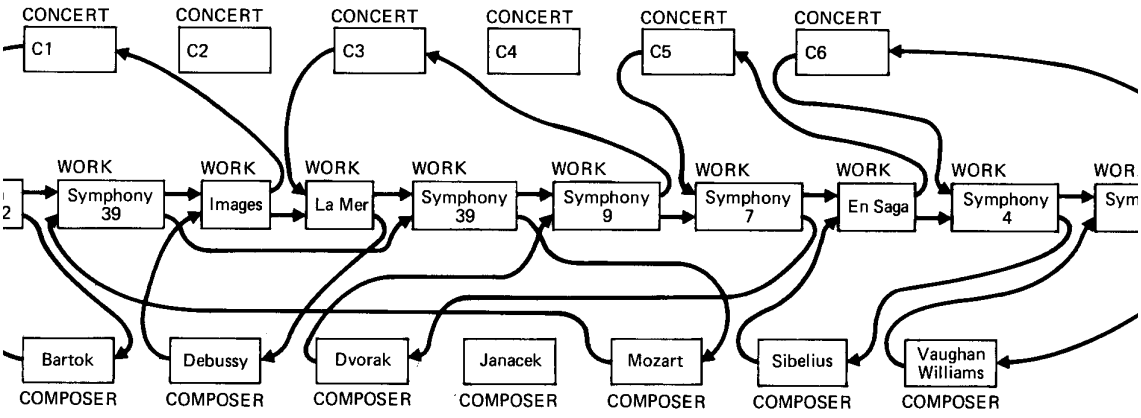


Figure 7.6: *CODASYL's DBTG conceptual model example*

Figure 7.6 illustrates the relationships between the DBTG constructs in this case.[107] The rectangular symbols represent records and the other symbols the DBTG sets of links between them. Notice that these links are unidirectional. A set of links is always between an owner or parent and members or children.

Now this diagram just shows the relationships between the various elements in the database at the abstract, conceptual level. It does not show how individual concerts, works, and composers are linked together. An example of these connections is

shown in Figure 7.7. Notice the chains of links between works and concerts and between works and composers.



: *CODASYL's DBTG instance model example*

The basic weakness of this approach is that while the DBTG model has the property of device independence, as the 1971 report indicates, it does not fully support data independence. This was spelt out quite clearly by Robert W. Engles, one of the IBM representatives on the 1971 committee.[108] His paper on this subject became known as the 'IBM position paper'.[109]

The hierarchical approach

In the meantime, what was IBM, itself, doing? Well, in the early 1960s, it had developed a DBMS called BOMP, designed to support bills of materials in manufacturing industry, just like IDS.[110] However, when IBM came to standardize on a single DBMS in 1969, when they 'unbundled', began to charge separately for software and services, they chose to do so with a product called Information Management System (IMS),[111] a system developed by or with North American Rockwell.[112]

Now the basic data structure in IMS is exactly the opposite of the DBTG model. It is hierarchical rather than nonhierarchical in the network approach. Of course, the DBTG supports hierarchical structures, just as IMS supports nonhierarchical ones. For any DBMS must support both these opposites as they are both inherent in Nature by the Principle of Unity.

However, for myself, I find hierarchical structures easier to understand. For hierarchies are the fundamental organizing principle of the Universe. For example, this book is hierarchically structured. It consists of parts, chapters, sections, subsections, paragraphs, sentences, words, and characters, all organized hierarchically. Arthur Koestler[113] and more recently Ken

Wilber[114] have placed their primary emphasis on hierarchical structures in their unifying theories.

Nevertheless, hierarchical structures have had something of a bad press in recent years because they are associated with patriarchal, authoritarian structures, as in the military, churches, schools, universities, governments, and companies. We can see why this is so from the root of the word *hierarchy*, which means 'chief priest'.

So today there is a powerful movement emphasizing the 'Web of Life', led by such people as Fritjof Capra.[115] For such people, the tree of knowledge is deprecated, a clear sign of dualism at work. For Wholeness is the union of all opposites.

Because not all structures in the Universe are hierarchical in nature, IMS provides two hierarchical views of data. The first describes the physical structure of the database, and the second logical structures, defined through various constructs that correspond to the DBTG DDL. The DML in IMS is called Data Language/1 (DL/1) presumably modelled on IBM's PL/1 (Programming Language/1).

The disadvantage of these two hierarchical views of data in IMS is that the symmetry of the network approach is lost. It is not possible to treat all data elements in the same, consistent manner. So while the latter is symmetrical and rather complex, the former is simpler, but unsymmetrical.

To illustrate the hierarchical structures of IMS, Figure 7.8 shows an example, again borrowed from one of Chris Date's many books.[116] It shows a record type in an education database consisting of courses, their prerequisites, dates on which they are held, the teachers of each offering, and the students enrolled on each course. The parts of the record in the hierarchy are called segments in IMS.

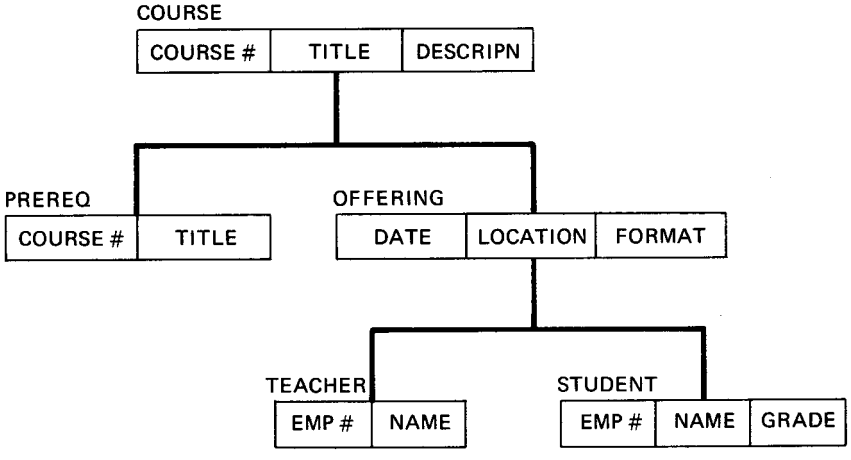


Figure 7.8: *IBM's IMS conceptual model example*

Like the concerts example in the network approach, this diagram shows just the abstract, conceptual relationships between the various segments in the record. It does not show relationships between actual occurrences of segments in the database. Such an example is given in Figure 7.9.[117]

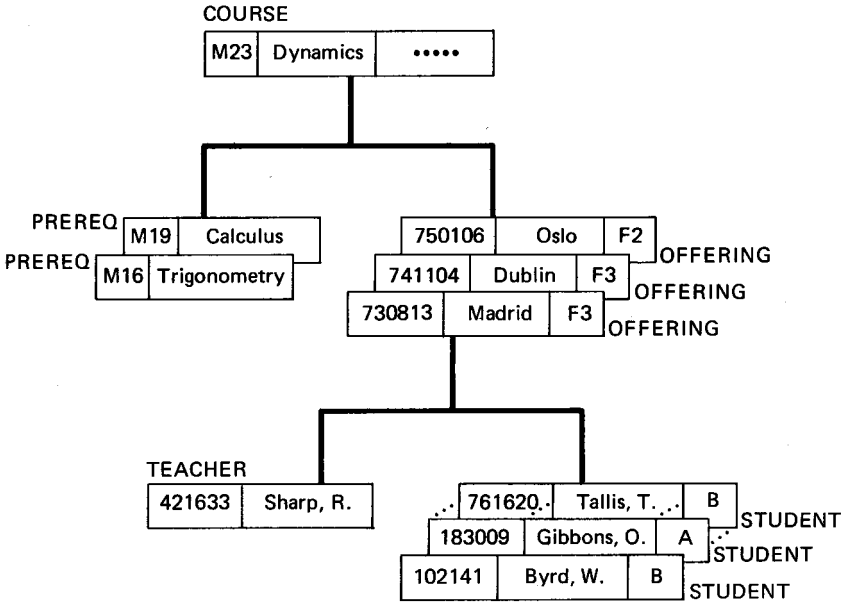


Figure 7.9: *IBM's IMS instance model example*

This diagram shows that, like the DBTG model, IMS does not fully support data independence. For instance, if it were decided to store offerings before prerequisites for some reason, then the programs that access the database would need to be changed. They are not independent of the way that the data is stored.

The relational model of data

The man who solved the problem of data independence was Ted Codd, an English mathematician who joined IBM in the USA after the Second World War, when he had served as a pilot in the Royal Air Force. In the late 1960s, Codd was working as a fellow in IBM's research laboratory in San Jose in California.

While there he realized that all data structures in a database could be seen in terms of mathematical relations, which are simply a type of set. Furthermore, he saw that the members of these sets could be expressed in the language of predicate calculus. He called this approach to database design a relational view or model of data.

Codd published his ideas in two seminal papers in 1969 and 1970 called 'Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks'[118] and 'A Relational

Model of Data for Large Shared Data Banks',[119] respectively. This is the key sentence in both these papers: "It [the relational view (or model) of data] provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes."

The first paper was published as an IBM research report. It therefore had a limited availability and is not well-known, even by database experts. For instance, it is not available from the British Library Document Supply Centre (BLDSC), the largest repository and supplier of research papers in the world.

In this initial paper, which I did not read until 2001, Codd thought that he would need to use second-order predicate logic as the basis of a language to retrieve records from a relational database. However, by the next year, he had simplified the model by eliminating repeating groups in records, like those shown in the IMS education database above.[120] He thereby realized that first-order predicate logic was a sufficient basis for any language designed to query the database.

This second paper was published in the *Communications of the ACM*, ACM being the leading organization of computer scientists in the USA. In it, Codd acknowledged that he was not the first to think of applying relations to data systems. Someone called D. L. Childs had published this idea in 1968. However, as this paper was not mentioned in Ted Codd's first paper, it is likely that Codd had had the idea relatively independently, a clear sign of morphic resonance at work.

It is impossible to overstate the significance of the 1970 paper. Quite simply, it described the key features of a mathematical theory for the basic resource of the data processing industry: data itself. It is the foundation on which the whole of computer science and the information technology industry can be built.

Well, nearly. As Kurt Gödel showed in 1931, mathematics has no foundation within the dualistic framework of Western civilization. In order to give mathematics and indeed all knowledge a solid foundation, we need a Nondual Context that unifies and transcends all opposites, as this book shows.

The reason why this is possible is that data structures in databases are nondeductive in character, as Codd pointed out in a little-noticed second paragraph in his 1970 paper. So Integral Relational Logic, which has evolved naturally from the relational theory of data, is essentially noninferential, and therefore not constrained by the Aristotelian assumptions of mathematics and conventional logic.

Despite the impeccable pedigree of the relational model of data, it fell on very stony ground when it was first published. The blindness and short-sightedness that Arthur Koestler described in *The Sleepwalkers* still pervades many today. It is amazing how long it sometimes takes us to see something that is standing right in front of us.

In the case of the relational model of data, it is not difficult to find the reasons for this blindness and short-sightedness. First of all, as Chris Date, another English mathematician who worked for IBM in the 1960s and 70s, acknowledged, even those who advocated the relational approach did not fully understand it at the time.[121] It has naturally taken many years to gain a clear understanding of its full implications; this is in the nature of evolution.

Secondly, the relational model of data entered the world when two branches of the Establishment, CODASYL and IBM, favoured the network and hierarchical approaches, respectively. They saw the relational model as a threat, as just another way of structuring data; they did not see that the relational model could unify and heal the conflict between the two opposing approaches at the time.

In terms of the DBTG model of CODASYL, this led to a 'Great Debate' in 1974 between Charles Bachman, for the network approach, and Ted Codd, for the relational approach, complete with 'seconds', like an old-fashioned duel.[122]

This is how Robert Ashenhurst reported the session:

> ... the whole session must certainly be put down as a milestone event of the kind too seldom witnessed in our field—Charlie Bachman, representing the Establishment and the folks who gave us the Data Base Task Group Report and many more-or-less functioning data management systems, *versus* Ted Codd, representing the Alternative Way, not extensively embodied in large systems as yet.[123]

As for IBM, they issued a statement of direction in 1971 stating that IMS and DL/1 were to be their strategic database products for the foreseeable future.[124] In the light of the Great Debate they reaffirmed this strategy in 1976. Evidently, IBM wanted nothing to do with either the DBTG or the relational model of data, developed by one of its own employees.[125]

In the event, commonsense prevailed. During the 1970s, a number of organizations, including IBM, explored how the relational theory of data could be used to build relational DBMSs, call RDBMSs. Ted Codd's eleven-page, arcane paper has led to the development of a multibillion-dollar industry, including such major companies as Oracle and Sybase, whose product lines are based solidly on Codd's seminal ideas. Forbes magazine once judged Larry Ellison, the founder of Oracle, the richest man in the world,[126] such is the significance of the relational model of data.

Oracle Corporation, founded in 1977 as Software Development Laboratories (SDL),[127] is the market leader in database systems, and a Fortune 500 company.[128] IBM, itself, did not introduce DB2, its primary RDBMS until 1983.[129] In 1984, Mark Hoffman and Bob Epstein founded Sybase, which went into partnership with Microsoft in 1988.[130] Even though they subsequently diverged, there are many similarities between Microsoft SQL Server running under Windows, and what is now called Sybase Adaptive Server Enterprise (ASE), running mainly under UNIX.

However, these developments led to further debates between the mathematicians and the practitioners. The former, led by Ted Codd and Chris Date, pointed out that in many respects, RDBMSs implemented only some of the principles of the relational model, and even, in some cases, violated some of its rules.[131] We do not need to go further into these rather technical issues here.

Nevertheless, it is useful to look in a little more detail at the relational model, for it helps to give Integral Relational Logic a greater air of authenticity for those who feel threatened by its central proposition: the Principle of Unity.

It is important to note here that the underpinning of the relational model of data is more mathematical than semantic in origin, even though the central concept of relation is based on that of set, which is as much key to semantics as mathematics. For while mathematics is generally considered to be the science of number, quantity, and space, more generally it is the science of patterns and relationships, which underlie the new maths.

What then, is a relation in the relational theory of data? Well, it is something that is familiar to us all. It is a table consisting of rows and columns, as we see on page 193 in Chapter 2, 'Building Relationships'. The table is a most convenient way of organizing sets of related information. Indeed, the clay tablets that were found in Mesopotamia some 5,000 years ago contained tables of information with three columns containing drawings of objects, numerals, and personal names, no doubt the numerals indicating the number of each pot, knife or whatever, and the names stating who owned them.

The formal definition of a relation illustrates a number of other key features of this fundamental data structure. This is the mathematical definition of a relation that Ted Codd gave in his original papers:

> Given sets S_1, S_2, …, S_n (not necessarily distinct), R is a relation on those n sets if it is a set of n-tuples each of which has its first element from S_1, its second element from S_2, and so on. We shall refer to S_j as the jth *domain* of R. As defined above, R is said to have *degree n*. Relations of degree 1 are often called *unary*, degree 2 *binary*, degree 3 *ternary*, and degree *n n-ary*.[132]

In early editions of *An Introduction to Database Systems*, Chris Date provided a similar definition of a relation. But in the seventh edition, he provides a revised definition, which makes a clear and important distinction between the heading and the body of a relation:

> Given a collection of n types or domains Ti (i = 1, 2, …, n), not necessarily all distinct, r is a **relation** on those types if it consists of two parts, a *heading* and a *body*, where:
>
> a.The **heading** is set of **attributes** of the form *Ai:Ti*, where the *Ai* (which must all be distinct) are the *attribute names* of r and the *Ti* are the corresponding *type names* (i = 1, 2, …, n).
>
> b.The **body** is a set of m **tuples** t, where t in turn is a set of components of the form *Ai:vi* in which *vi* is a value of type *Ti*—the *attribute value* for the attribute *Ai* of tuple t, (i = 1, 2, …, n).
>
> The values of m and n are called the **cardinality** and the **degree**, respectively, of relation r.[133]

Two key points are contained within these definitions. First a relation consists of two parts, a heading and a body. These correspond to what are called intension and extension, respectively, in traditional logic. However, the heading is more often called a schema or scheme, which defines the structure of the relation, which corresponds to the epistemological level of IRL.

Secondly, each attribute in the relation draws its value from a domain of values. For instance, an attribute denoting someone's age would typically draw its values from a domain that ranged from 0 to 120 years, let us say. Not that domains of values consist of just numbers. The relational model of data supports both quantitative and qualitative values with equal facility.

For example, a domain of values could consist of the set {red, blue, green, cyan, magenta, yellow} and many other colours. The attribute value would then be a member of this set. But domains of values can contain much more complex members. For example, if we wanted to create a relation containing information about different species of birds, we might want to have attribute values that are still pictures of the birds, video pictures of the birds in flight, and sound clips of bird songs.[134]

Each row or tuple in a relation contains a record of information about a particular entity. The set of entities represented in a relation form a class. A relation can thus be thought of as a propositional function in first-order predicate logic. For instance, in the telephone directory example in Chapter 2, the telephone subscriber relation could be written as a propositional function, where the unknowns are predicates, like this:

The telephone subscriber with name *Name* lives at address *Address* and has telephone number *Telephone number*.

Replacing the unknowns with entries in a particular row in the relation could generate this sentence:

The telephone subscriber with name Jackie Butler lives at address 25 Orchard Way and has telephone number 955-4395.

This record is then a proposition in logic, with the value 'true'. If the telephone number read '429-8490', then this would be a false proposition. So providing all unknowns in propositional functions have a value, a database can be seen as a collection of true propositions.[135] In practice, this is not always the case. Some attribute values may be NULL, a subject that has caused no end of debate among the cognoscenti.

While values in relations are atomic, there is no theoretical reason why domains should not include data structures like arrays and sets, including relations. This would not violate the rule that data elements in relations should be atomic if these structured data types are encapsulated in object-oriented programming terms. In other words, from the point of view of the relational model, the internal structure of a structured domain of values is hidden from the

user, in contrast to the structure of the relation, itself, which is completely open for access.[136]

This situation has led to some confusion in the relationship of the relational model of data to the object-oriented view. Each row or tuple in a relation contains a record of information about a particular entity. The set of entities represented in a relation form a class, which Ted Codd called an 'entity type' in his 1969 paper.[137]

It thus seems obvious that entity type in the relational model corresponds to class in the object-oriented model. However, Chris Date has emphatically called this mapping a 'Great Blunder'.[138] For him, a class in OO languages is just a data type and relations are not domains. This is because in the relational model a relation is a variable that can take different values dynamically, while a domain of values is a relatively static construct. The latter is neither a variable nor a value. To illustrate this, in C a data type can be given a name with the `typedef` statement called a 'tag'.

However, this is not an either-or situation. As far as I can tell as a nonprogrammer, a class in OO languages actually functions like both an entity type and a data type. These languages do not make a clear distinction between these two quite different ways of determining semantics. So Rational Rose Data Modeler, for instance, maps both a Table (entity type) and Domain (data type) in the relational model to a Class in the UML, which seems the correct approach.

This rather technical problem has given rise to a dispute between the OO and relational communities. The former seems to think that they could extend these programming techniques into the database arena, displacing the relational model of data, just as the relational model had displaced the earlier network and hierarchical approaches. However, as far as I am aware, most databases today are still based on the relational model, not the least because of its mathematical pedigree.

Nevertheless, the last time I investigated this issue, around 2002, this confusion had still not been satisfactorily resolved, as evidenced by two major books on this subject. They are *Foundation for Future Database Systems: The Third Manifesto, Second Edition*, written by Chris Date and Hugh Darwen and *The Object Database Standard, Third Edition*, edited by R. G. C. Cattle and Douglas Barry, both published in 2000.

Be that as it may, the relational and object-oriented modelling techniques have merged in Integral Relational Logic, which is not directly concerned with the expression of these structures in computer technology. IRL is more concerned with healing the mind in Wholeness, which we look at in Chapter 13, 'The Prospects for Humanity' on page 1027.

Growth of conceptual modelling structures

The growth of data structures has led naturally to the growth of data modelling structures, which would be better called information, conceptual, or semantic modelling. For as we saw

on page 159 in Chapter 1, 'Starting Afresh at the Very Beginning', in the data-processing industry, information is data with meaning. And all we can really say about the structure of data prior to interpretation by a knowing being is 'Wholeness is the union of all opposites'—the Principle of Unity—and 'The underlying structure of the Universe is an infinitely dimensional network of hierarchical relationships,' as we see on page 250 in Chapter 4, 'Transcending the Categories' and page 217 in Chapter 2, 'Building Relationships', respectively.

These models can be expressed in terms of a data definition language (DDL), such as that in Structured Query Language (SQL),[139] the primary language for defining and accessing relational databases, initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in the early 1970s.[140] But it is more informative to show the relationships between relations in graphical form.

This is not a new idea. John Sowa tells us that in the third century BC the Greek philosopher Porphyry began the practice of conceptual modelling by illustrating the hierarchical structure of Aristotle's categories.[141] Then in the latter part of the nineteenth century, the philosopher, Charles S. Pierce, did much work in developing relational graphs depicting formulæ in first-order predicate logic,[142] independently of Gottlob Frege, who represented such propositions as trees in his seminal work *begriffsschrift* 'concept writing'.[143] (The German words *Begriff* 'concept' and *begreifen* 'to comprehend' are cognate with Swedish *begrepp* and *begripa*, with the same meanings, from the PIE base **ghreib* 'to grip', also the root of *grip* 'grasp, clutch', with a figurative meaning 'Intellectual or mental hold; power to apprehend or master a subject'. So a concept is something that can be held in the mind.)

Today, conceptual models, whether in the field of database design or artificial intelligence, can be seen as applications or extensions of the mathematical concept of graph. This consists of a collection of nodes with lines drawn between them, generally called arcs or edges, not to be confused with the graphs that we drew in school, which provide a visual representation of algebraic equations, or simply collections of data values.

There are many conceptual modelling techniques in use today. One is entity-relationship (ER) modelling, which is sometimes extended into entity-attribute-relationship (EAR) modelling, although *entity* here really means 'entity-type' or 'class'. ER and EAR modelling arose directly from the relational model of data.

Another technique is object-oriented (OO) modelling, already mentioned, which arose from object-oriented programming. Again, this would perhaps be more meaningfully called class-oriented modelling, for it is mainly classes rather than objects that are being modelling. Objects only come into existence when the database is created or programs are executed; they are particular instances of general classes, universals in Plato's terms. Such objects generally do not exist during the modelling process itself, although it is sometimes useful to create an object diagram as well as a class diagram in UML. An example of an object diagram is Fig-

ure 2.10, 'An instance diagram for a family tree' on page 210. Figures 7.7 and 7.9 on pages 596 and 598, respectively, are examples of instance diagrams in the network and hierarchical data modelling methods.

A third method, which has evolved from some conceptual modelling techniques developed in Europe, is called Object Role Modelling. The key distinction between this method and the others is that attributes are treated in exactly the same way as entity types or classes, leading to a more explicit and detailed model. It thus has some similarities with EAR modelling.

Now it is one thing to learn these modelling techniques. It is quite another to learn how to use them in practice. For there are, in general, many ways in which a universe of discourse can be modelled in terms of classes, entities, attributes, and the relationships between them. The best book I have come across on this topic is Bill Kent's *Data and Reality*.

Entity-relational modelling

Credit for the creation of entity-relationship modelling is normally given to Peter Chen, who published a seminal paper on this subject in 1975.[144] However, as Chen himself acknowledged in this paper, Charles Bachman had previously developed a visual modelling technique, which he had published in 1969.

Figure 7.10 shows an example taken from Chen's original paper. Notice that in this notation, there are two forms of node, one depicting concepts and the other relationships between them. This is similar to John Sowa's notation of conceptual graphs in the field of artificial intelligence.[145]

It is interesting to note that Chen subtitled his seminal paper 'Toward a Unified View of Data'. He did not see himself building on the foundations of the relational model of data. Rather, he presented his work as a synthesis of the network model, the relational model, and something called the entity set model, developed by four researchers at IBM.

Over the years, many researchers and practitioners further extended the business information captured by such entity-relationship diagrams. These included Information Engineering, developed by Clive Finkelstein and James Martin,[146] both formerly of IBM; IDEF1 and IDEF1X, developed for the US Air Force;[147] and a number of techniques developed for the vendors of RDBMSs, such as Oracle's CASE*Method developed by Richard Barker,[148] subsequently incorporated into Oracle's Designer product, and Sybase's PowerBuilder, the only modelling tool with which I have had limited experience with in practice. For while these tools were emerging in the marketplace, I was not in paid employment, as my primary focus of attention was in the development of IRL in order to heal my fragmented mind in Wholeness.

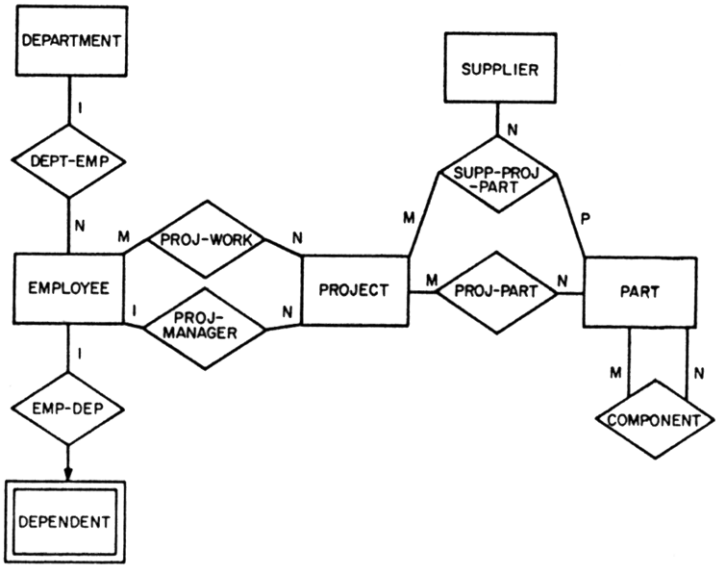


Figure 7.10: *Chen's entity-relational modelling notation*

Figure 7.11 shows some of the key features of an entity-relational model in the CASE*Method notation.[149] The rectangles denote entity types, together with a few of their attributes. There are two major ways of representing relationships: in meaningful sentences and as annotated lines between the entity types. Both methods are provided to ensure that all parties understand the database design as well as possible. This is vital because so many infor-

mation systems have failed to meet the needs of the users because the universe of discourse was not sufficiently well understood at the analysis stage of development.

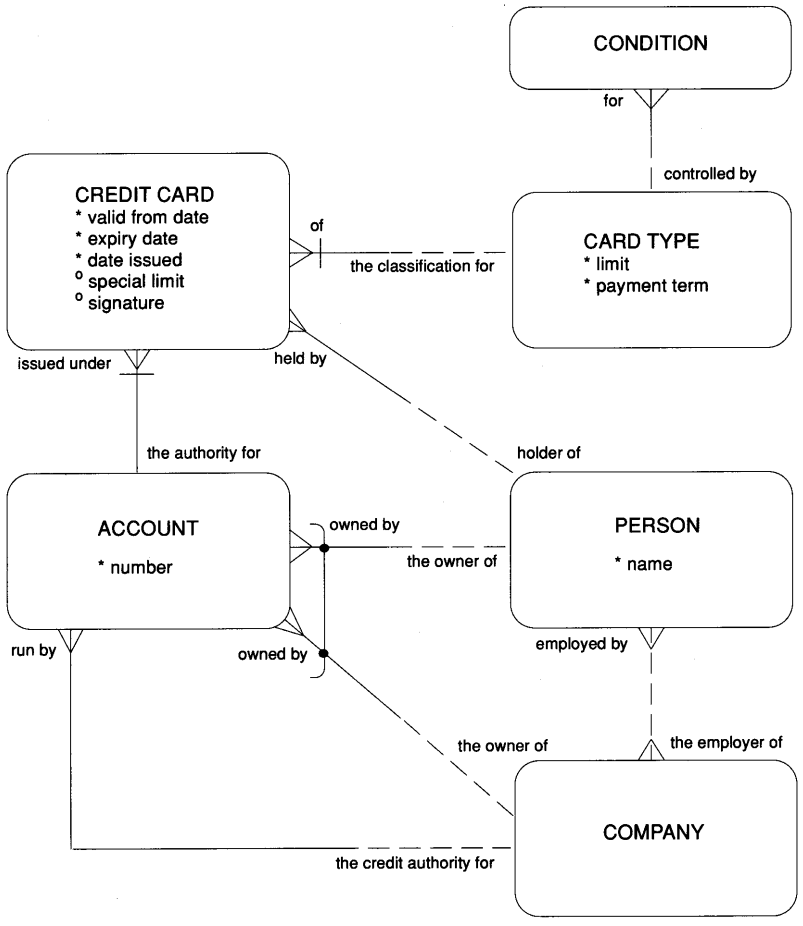


Figure 7.11: *CASE*Method entity-relational modelling notation*

In CASE*Method, relationships are first defined as sentences in this form:

> Each *entity-type one* must be/may be *relationship* one and only one/one or more *entity-type two*.[150]

As David Hay points out, the relationship is either a prepositional phrase or gerund phrase from the names of a relationship and its following entity. Verbs, as such, are not used as definitions of relationships because verbs denote actions, such as functions, not relationships. "This makes it possible to translate a relationship into sentences that not only sound natural but which precisely describe its nature, its cardinality, and its optionality."[151]

For example, in the situation depicted in the diagram above, the two sentences that represent the relationships between Credit Card and Person are:

> Each Person may be holder of one or more Credit Cards.

Each Credit Card must be held by one Person.

Whether or not the relationship is mandatory or not (must be or may be) is depicted by a solid or dashed half line. The cardinality of the relationship (one and only one or one or more) is determined by a symbol at the end of the line. A 'toasting fork' symbol, called a crow's foot, denotes a one or more relationship. A lack of this symbol denotes a one and only one relationship. As you can see from the diagram, the relationships are appended to each end of the relationship line in this notation, although some techniques merge the two reciprocal relationships into one.

Another key concept captured by these relationships is the role that each entity type plays in the relationship. For instance, there could be two relationships between Cars and Persons, one denoting ownership and the other the driver of the car. In the example we are looking at, there are two relationships between Company and the Account that the Credit Card authorizes. One denotes a company that is the owner of a credit card, while the other denotes the fact that a company is the issuer of a credit card, thereby acting as the credit authority for the card.

This example illustrates another complication in conceptual modelling. There are two types of entity that can own a credit card, Persons and Companies. This fact is denoted in the diagram by an arc joining the two ownership relationships, indicating that they are mutually exclusive.

Object-oriented class modelling

To illustrate a class model in UML, Figure 7.12 shows an example taken from the standard textbook on the subject.[152]

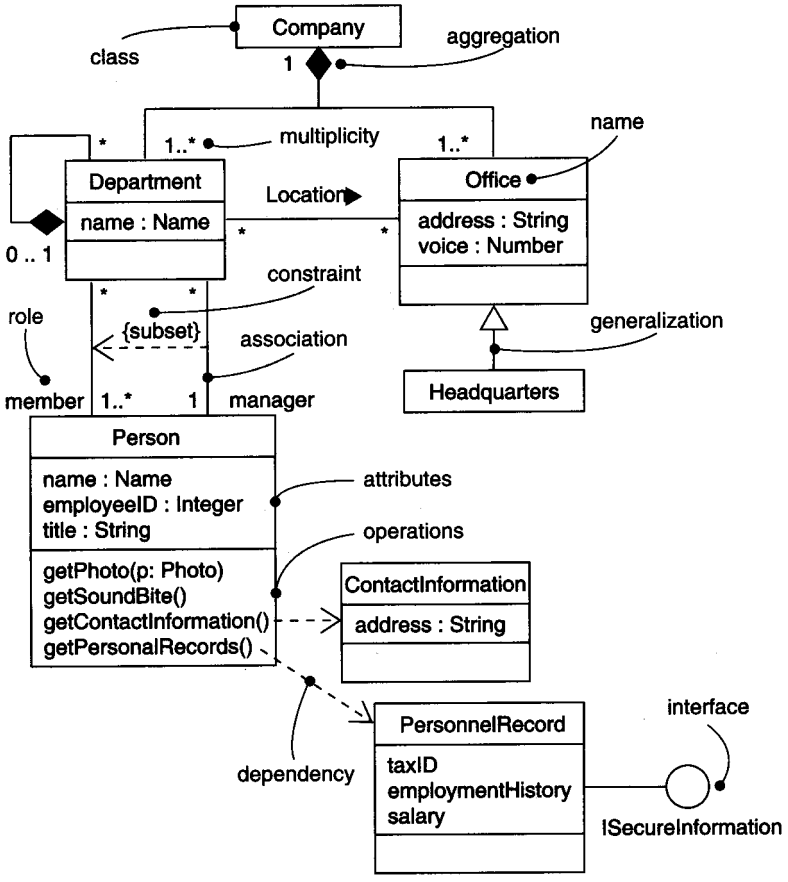


Figure 7.12: *UML class model notation*

Here the boxes represent object classes, most often called just classes, which from a business modelling perspective correspond to entity types in ER modelling. In general, the class box consists of three parts: a name, a list of attributes, and a list of operations that process objects in the class. The last of these define the interface to methods in an object-oriented language, such as C++, and as such are presumably of greater interest to programmers than business modellers.

It is possible to define many different types of relationship in UML, the most important from a business modelling perspective being association and generalization. A subtype of association relationships is the aggregation relationship, which typically has a direct physical mapping, for example in an engineering parts database. Another common example is an or-

ganization consisting of divisions, departments, sections, and teams, which can be depicted as reciprocal relationships, as in the example.

Yet another example of an aggregation relationship is the human body consisting of organs, cells, molecules, atoms, and subatomic particles, which are aggregated in reverse order of the list here. It was from this type of relationship that Arthur Koestler coined the term 'holon', meaning an entity that is both a part and a whole.

However, not all hierarchical relationships consist of holons. Generalization relationships are the most obvious counterexample. For instance, human beings are a type of primate, which is a type of mammal, and so on. Generalization relationships play a central role in OO programming because properties and methods can be inherited by subtypes from a supertype.

Object-role modelling

Another conceptual modelling method that I did not come across until doing some research for this chapter in the autumn of 2001 is object-role modelling (ORM). I was doing a search on the Web for references to conceptual modelling and found a site that published the *Conceptual Modeling Journal*.

The term *Object-Role Model* was coined by Eckhard Falkenberg in a paper presented to an IFIP conference in 1976. In the 1980s, ORM evolved into NIAM, originally aN Information Analysis Method, but now known as Natural-language Information Analysis Method. Research into this method then moved to Australia, where Terry Halpin renamed it ORM and developed a CASE tool to support it. A early version of this tool was called InfoModeler, which was then bought by a company called Visio, becoming VisioModeler. Visio was, in turn, bought by Microsoft, which now markets the product. So even though ORM is not well known, it has now become mainstream.[153]

As far as I have been able to ascertain, ORM would seem to the most comprehensive and thoroughly researched conceptual modelling method available today. All other methods, such as UML and ER, "can be easily abstracted from ORM models", as Halpin states in his scholarly book.[154]

Figure 7.13 shows an example of an ORM model for employees.[155] Note here that entity types and attributes are treated in exactly the same way in the method. Also, like Peter Chen's

original ER method, relationships are shown explicitly in the diagram, in this case as two or more rectangles depending on whether the relationship is binary, ternary, and so on.

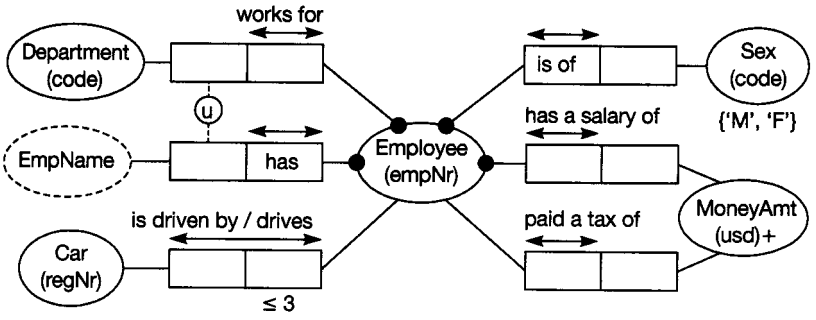


Figure 7.13: *ORM conceptual model notation*

Also, like Richard Barker's ER method, a key feature of ORM is that both facts and constraints are defined in formal sentences in natural language, which helps to ensure that the model matches as well as possible the rules of the business. For instance, we can say that 'Employee with *EmpNr* is of *Sex*', which is essentially a propositional function in predicate logic.

Other conceptual-modelling tools

These are just three examples of conceptual modelling tools. Others I am aware of, but have not studied, are ConceptDraw, Inspiration, and OmniGraffle, available on the Mac. In the 1990s, IBM had an OS/2 product called Thesaurus Administrator/2, which could be used for creating, managing, and using thesauri. No doubt there are other similar semantic modelling tools on the market today.

Thesaurus Administrator/2 was developed at the IBM software laboratory in Germany that produced Storage and Information Retrieval System (STAIRS) in the early 1970s, which I marketed as a systems engineer in IBM's Government sales office at the time. STAIRS was a precursor to the search engines like Google that we see on the Web today, for it could search through unformatted sections of text, such as book abstracts in libraries, looking for a match on one or more words.

It was about this time that I first learned about Ted Codd's relational model of data and wondered whether it would be possible one day to create a generalized modelling method that modelled both formatted and unformatted data. This would be most useful in the UK Government's employment system that I was working on at the time, which required both structured information, such as name, date of birth, and so on, as well as textual description, such as job description.

There is another fascinating stream in the growth of structure here. In the late 1960s and early 70s, we did not have the graphical displays we have today. Text that STAIRS worked

with was mainly entered on punched cards or golf-ball typewriter terminals managed by a software package that evolved into Script, an IBM text formatting language. Script provided the micro-level for a set of macros in the IBM Generalized Markup Language (GML),[156] developed as far back as 1969[157] by Charles Goldfarb, Edward Mosher and Raymond Lorie, whose surnames conveniently had the initials GML.[158] GML was a part of the IBM Document Composition Facility (DCF), DCF/GML being known as BookMaster, a product I used extensively in the 1990s.

Charles Goldfarb then developed GML into the Standard Generalized Markup Language (SGML), a metalanguage in which one can define markup languages for documents, but which is monstrously difficult to understand and use.[159] Nevertheless, Tim Berners-Lee, a physicist working at CERN, developed HyperText Markup Language (HTML) as an application of SGML and the World Wide Web was born in 1989.[160] Another application of SGML is Extensible Markup Language (XML),[161] a general-purpose markup language that enables users to define their own elements much more simply than in SGML. Indeed, XML is so powerful that it is used today as a common language for a wide variety of structured systems.

Metamodels

This is possible because the Universe has a deep underlying structure that is expressed in the most generalized modelling method of all: Integral Relational Logic, which can even model itself, as can a number of the modelling tools that we have looked at in this section. For modelling methods themselves use a set of concepts, which can be modelled using exactly the same set of conceptual modelling techniques as are used to model normal business domains. Thus a metamodel of the method can be produced. Using our self-reflective Intelligence, it is in this way that we can model all business processes, including our own modelling processes, and so create a comprehensive model of the psychodynamics of the whole of society, and hence of evolution from Alpha to Omega.

For instance, there are four basic concepts in entity-relationship modelling, entity type, attribute, domain, and relationship. Richard Barker shows how these can be modelled, reproduced in Figure 7.14, although he uses the term *entity* rather than *entity type*.[162]

For instance, we can say that 'each entity type must be described by one or more attributes' and conversely, 'each attribute may be of one and only one entity'. As all relationships between entity types are binary in CASE*Method, there are two relationships between Relationship and Entity Type. Another interesting point about this model is that both domains, or data types, and entity types are related to each other in a hierarchical set of supertypes, types, and subtypes, as illustrated by the recursive relationships attached to these two entity types, sometimes called a 'pig's ear'!

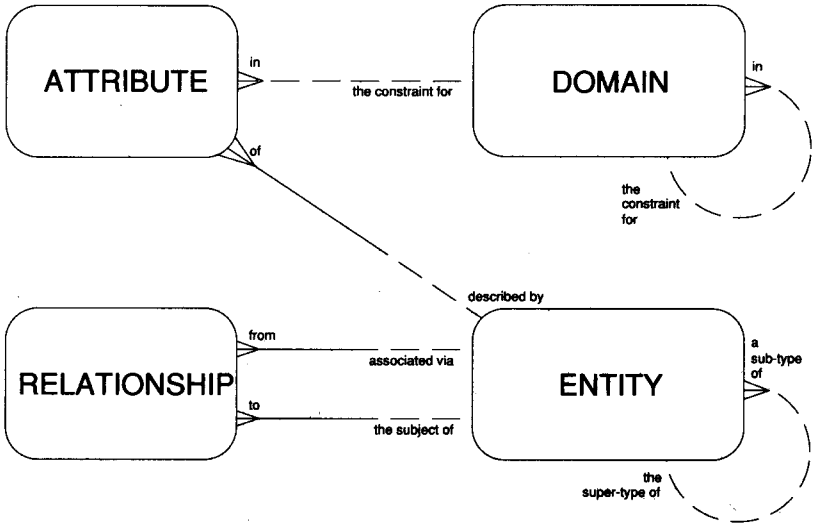


Figure 7.14: *CASE*Method metamodel*

David Hay provides another metamodel in the CASE*Method notation in his book *Data Model Patterns: Conventions of Thought*. In his book, Hay defines several general models of commonly found universes of discourse, including the enterprise itself, things of the enterprise, accounting, and process manufacturing. At the end of the book, he then proposes a generalized model that could model all the patterns illustrated in his book, reproduced in

Figure 7.15, which he calls a 'Universal Data Model'.[163]

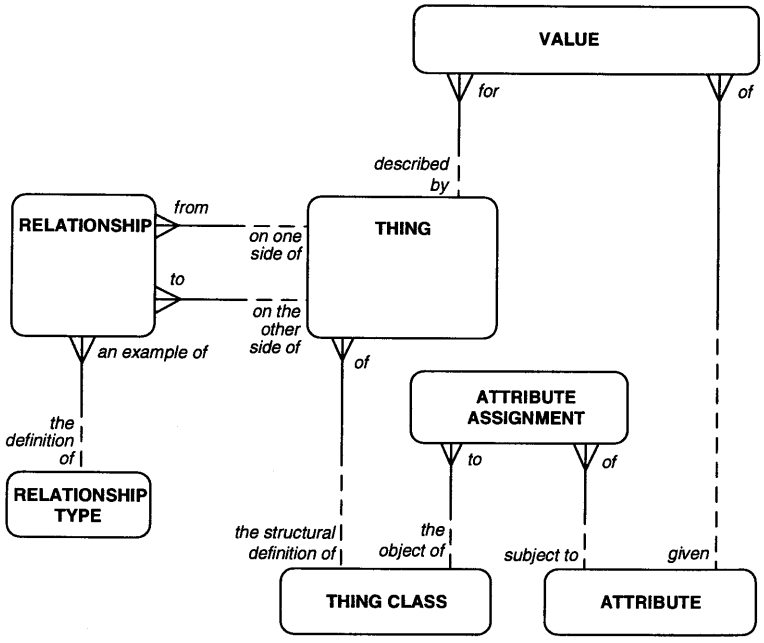


Figure 7.15: *A Universal Data Model*

Hay is clearly moving in the direction of Integral Relational Logic because he suggests that the Universal Data Model "covers all things in the Universe", adding in a footnote, "Well, all right, perhaps not *all* things … ."

These metamodels show that underlying the complexity of the world we live in there is the most exquisite simplicity. Academic subjects that attempt to describe the complexity of systems may be difficult to understand because we do not understand their underlying simplicity. When I was at school, the next year's textbooks could look really complex. But when I understood them, they became quite simple, for I could grasp their subject matter as a coherent whole. The underlying simplicity of the complexity of the Universe is encapsulated in the Principle of Unity and in Figure 1.47, 'Overall design of the Universe' on page 167, which shows that all beings in the Universe are related to all other beings in zero to many ways.

This diagram is both a metamodel and a model, for there is no separation between them. The observer and observed are one. Indeed, we can extend this to a metametamodel and metametametamodel indefinitely. The elegance with this abstract way of thinking is that we are not trapped in an infinite series of regress, which so troubled Bertrand Russell, who developed his theory of types and *Principia Mathematica* to try to avoid the paradoxes that arise from this situation. In mathematics, this series of metamodels is not unlike the infinite series of derivatives of e^x, which are all e^x. Nothing changes in this series.

It we then integrate all the conceptual models that any knowing being has developed or ever will develop, whether formally or informally, explicitly or implicitly, into a coherent whole, we see a picture of the utmost complexity, which nevertheless contains the elegant simplicity of the Principle of Unity at its centre. By the Principle of Unity, we can then allow all these forms, structures, and relationships to dissolve into Ineffable, Nondual, Wholeness, a seamless continuum, with no divisions or borders anywhere, and we are in the bliss of nirvana, which means 'extinction' (of the sense of a separate self). This then is central to the complete union of Western reason and Eastern mysticism.

The end of the growth of structure

What all this means is that we are nearing the end of the growth of structure on this planet at evolution's glorious culmination. The business modelling methods that have evolved into Integral Relational Logic, the foundations and framework for the Unified Relationships Theory, the solution to the ultimate problem in science have reached the saturation point at the top of the growth curve. This book shows not only how to unify the two major business modelling methods, it also shows how we can unify *all* opposites, thus bringing some fourteen billion years to an end at the Omega point of evolution. So IRL provides the framework or infrastructure not only for the data-processing industry, but also for all human activity.

Within this overall framework, we can also see that the underlying architecture of operating systems, which provide the context for all application programming, are now reasonably stable and mature. While changes are still being made to the nucleus and to the application programming interface (API) of the major operating systems, it is difficult to envisage a major new operating system being developed from scratch to replace the UNIX and Windows families. There simply is too much investment in these operating systems for such radical changes to be made.

A similar point can be made about programming languages, the theory and practice of which is now well understood. These too are very close to the top of their growth curve. It is difficult to foresee C, its object-oriented versions, like C++ and Objective C, and other object-oriented languages, like Java, being superseded as the major programming language used by software developers. For these object-oriented languages reflect very closely the underlying structure of the Universe. There is nowhere further for them to grow.

And what about the basic applications used in offices, like Word, Excel, and Photoshop? Are we going to see Word version 67 or even 23? When these applications were first developed, the facilities added to each version were quite significant. For instance, table handling did not appear in Word on the Mac until version 3 if I remember rightly, and was significantly improved in the following version. But how much extra function can software houses add to these applications anyway? And who needs all these functions? Judging from the Word doc-

uments I receive from time to time, many people have still not learnt how to use the tab command.

To encourage us all to use the increased processing power available today, companies are promoting the notion of a digital lifestyle, with a multitude of audio/video products available on the market today. But how much further can the compression algorithms go? These certainly appear to be approaching the top of their growth curve. And how much further can improvements be made in the quality of pictures and sound that we human beings can discern? The quality of my home cinema system is almost as much as I could possibly wish for. Digital television, which has now arrived in Sweden, would no doubt be an improvement. But this is something that I could live without if I had a choice.

If we take just these factors into consideration, ignoring the deeper spiritual, psychological, and ecological changes that are happening in the world today, it is quite clear that the overall growth curve for the IT industry looks something like that shown in Figure 7.16.

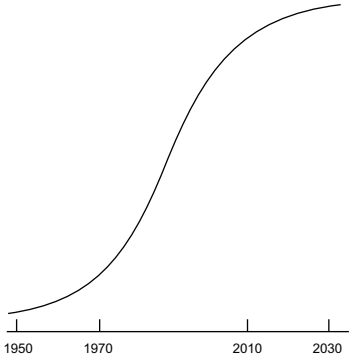


Figure 7.16: *IT growth curve*

It is difficult to be precise about the exact date of point C on this growth curve, because growth processes are generally continuous in nature. However, it is clear that the curve will be travelling in a quite new direction by the beginning of the second decade of this century, which curiously corresponds very closely to the end of the 25-year period of Harmonic Convergence, introduced by José Argüelles, which is also associated with the end of the Mayan Great Cycle, described in Subsection 'The Mayan calendar' in Chapter 6, 'A Holistic Theory of Evolution' on page 546. As many are aware, this harmonic convergence is coming about through the action of morphogenetic fields, a concept introduced by Rupert Sheldrake in *A New Science of Life* in 1982.

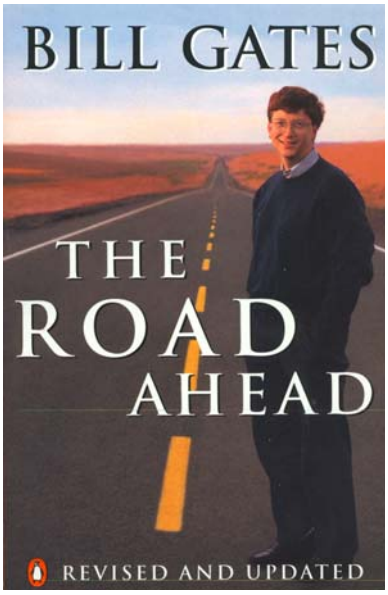


Figure 7.17: *A deluded vision*

It is thus false to assume that technological growth can continue to drive economic growth for very much longer. We are rapidly approaching the limits of technology's ability to enhance our mental and physical capabilities. The overall effect of this evolutionary inevitability will be the total collapse of the global economy, the end of civilization as we have known it for the past several centuries and millennia.

Yet no government in the world is preparing for such a major change in the way that we organize society and conduct our business affairs. The road ahead is utterly different from that envisaged by Bill Gates, illustrated on the front cover of this book.

If the death of Western civilization and the global economy is not to lead to the immediate extinction of the human race, it is thus imperative that we rebuild the infrastructure of society on the seven pillars of wisdom, becoming free of all the delusions that inflict our learning today, as we explore further in Chapter 12, 'The Crisis of the Mind' on page 989. Maybe this book can make a small contribution to helping us through this discontinuity in evolutionary history. For the only choice we have as a species is to turn the attention inwards so that we might realize our fullest potential, both collectively and individually. It makes no sense to put the primary focus of our attention on building machines that extend the capabilities of the human mind, when such a policy is driving humanity insane; it is just technological madness.