Working As A Commercial Programmer In The Early 1970s

Garth Eaglesfield, author of 'The Programmer's Odyssey'

Working as a commercial programmer in the early 1970s was light years away from how most modern commercial programmers work today. The way we created programs in those days may seem completely foreign, if not unbelievable, to modern programmers. But at that time the Digital Revolution with its mainframe computers and punched card technology was at a comparable stage to the early days of the Industrial Revolution when steam engines and large noisy mechanical machines were the state of the art. This article is based on my own experiences on the south east coast of England during the early 1970s and it attempts to recreate the almost forgotten world in which early commercial programmers plied their trade.

The Industrial Manufacturing Process

For many of us in the 1970s to become a commercial programmer you attended a 7 day COBOL training course, at your employer's expense. As a complete computer novice how much could you learn in that time? Not much really. But after taking the course you faced a pretty bruising encounter with reality as you were thrust straight into learning how things really happened back in the work place. On the training course you wrote your program fragments on coding sheets which were taken away and dealt with by 'computer operations' and you got back printed reports from the same place. You never needed to leave your desk, programming was purely paper based office work.

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

To what extent this would require you to leave your desk and carry out non-paper based tasks varied a lot in different workplaces but in all of them it was a more or less industrial manufacturing process involving a defined sequence of tasks. Different tasks were carried out by different specialists using different tools and the whole process had a production line feel to it and involved a clear division of labour.

This basically sequential way of developing software was later characterized as 'waterfall' development as it was a relentless one-way journey towards the working program. Once set in motion it was impossible, or at least extremely difficult, to make any changes to the requirements.

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

There was a clear pecking order, if not a class structure, in the hierarchy of specialists involved in the program production process and it usually included the following roles.

Systems Analysts

They specified what the program was to do using requirements taken from business users to create a detailed written specification which it was usually completely impossible to amend until the program was in production. Specifications varied from excellent to dreadful. The ideal program specification was like an engineering spec.

Development programmers

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

Computer operators

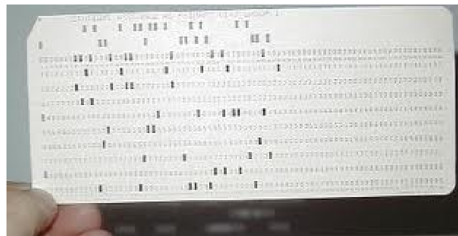
Mainframe computers usually ran for close to 24 hours a day and these shift workers followed elaborate and complex procedures to manage the work load.

Maintenance programmers

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

Key punch operators

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



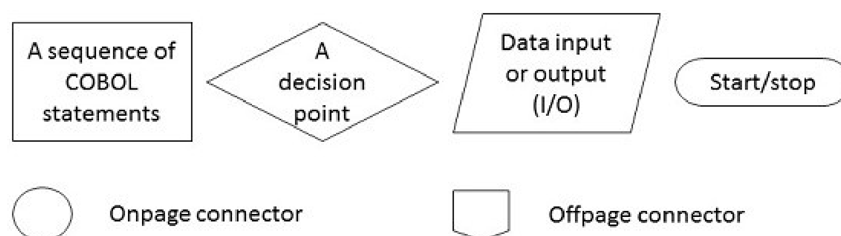
One big happy family then?

Well not always. As with industrial production lines various predictable niggles could creep in. In general the members of each layer of the hierarchy felt superior to those in the layers below and each layer's members blamed any problems they had on the poor quality of the work carried out by those in the layers above. It was very much business as usual in that respect.

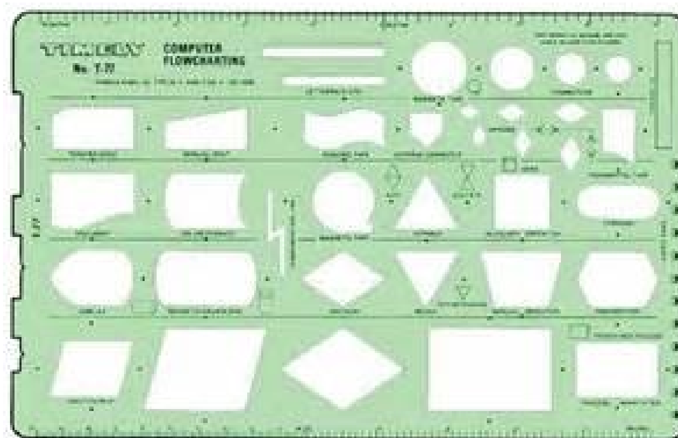
The role of women was similar to their role in many other industries in the early 1970s which is to say confused, sometimes proto feminist, and at times confrontational. The only women who were heavily involved in the program production process were usually the key punch operators who in those pre-feminist days were normally referred to as 'the key punch girls'. However as in most areas of life physical attractiveness would often allow key punch girls to overcome their lowly place in the hierarchy and establish out-of-work social relationships with those higher up

Designing Your Program

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



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



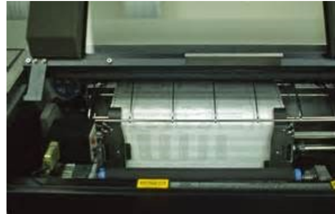
A flowchart template

The Monolithic Sequential Programming Style

A fairly typical COBOL program of the time, which we will use as an example, would be part of a payroll processing batch job. Running first in the batch job it would read in the hours worked by each employee from punched cards prepared by the key punch operators from the employee's paper time sheets. By using pay rates it read in from a magnetic tape it would create an output magnetic tape containing employee gross pay records, which would form the input to the next program in the payroll processing batch job. It would also produce a printed report on the traditional green lined paper detailing each input record, the calculated

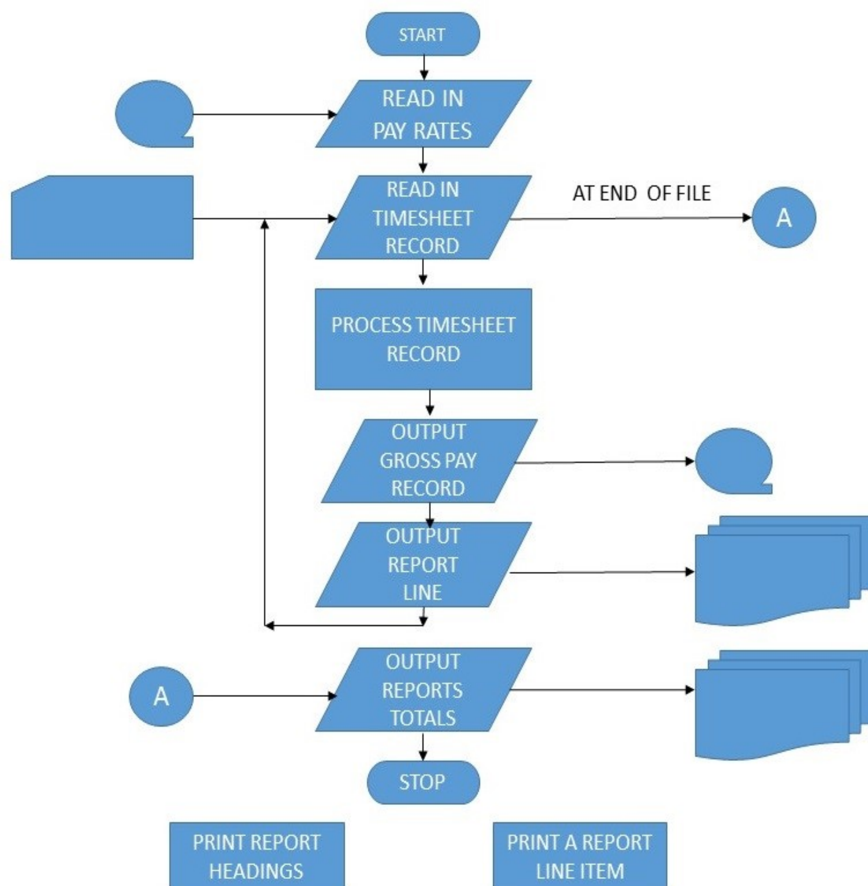
gross pay, any data errors encountered (and input data read from punched cards was notoriously susceptible to errors caused by keying mistakes), and totals.

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



A 132 column line printer at work

The main function of Data In, Data Out programs was processing data, they had no user interface as there were no users, only computer operators. Data was read in from data storage devices such as, obviously, punched cards but also electronic devices like magnetic tapes. The data was processed in the computer's memory, usually referred to as RAM, which in a COBOL program essentially meant the Working-Storage section. Data held in RAM is dynamic and not retained when the program exits thus the need for the Data Out step where the processed data was written out to another static data storage device. A high level flow chart for our example program would look something like this.



The main logic of the program, in this case ‘Process timesheet record’, sat in what was generally called the main loop controlled by a GO TO statement at its end and its detailed design was, hopefully, described in lower level flowcharts. The program was essentially a monolithic block of instructions which could be broken up into paragraphs by inserting a paragraph label prior to a COBOL statement. However the COBOL statements were executed strictly in sequence despite the presence of paragraph labels. The only exceptions were when control was diverted by GO TO or PERFORM statements to an out of sequence paragraph label elsewhere in the program. The processing that occurred before the main loop was often called ‘Housekeeping’ and dealt with opening files and any other start-up tasks. Paired with it was a block of code executed after the main loop had completed which dealt with closing files and writing out totals or other summary data.

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

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

Committing Your Program To Paper

With your overall program design completed you were now ready to actually write your program which in those days really did mean ***writing*** your program. Your writing instrument would be a pencil and your writing paper would be a block of COBOL coding sheets. The example coding sheet below shows the famous minimal ‘Hello World’ program written, unusually neatly it should be pointed out, in COBOL.

Program: PROG01 | Requested by: QUASAR CHUNAWALA | Page 01 of 01
Programmer: QUASAR CHUNAWALA | Date: 27-02-2011 | Identification

```
01  IDENTIFICATION DIVISION.
02  PROGRAM-ID.  PROG01.
03
04  ENVIRONMENT  DIVISION.
05
06  DATA DIVISION.
07
08  PROCEDURE DIVISION.
09      DISPLAY  'HELLO  WORLD'
10      STOP RUN.
11
12
13
14
15
16
```

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

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

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

- Columns 8 to 11 were known as 'Area A' and were used to begin the declarations of high level data structures in the Data Division and to begin paragraph name declarations in the Procedure Division.

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

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

Creating The Source Code Card Deck

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

A Key Punch Operator

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

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

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

Ø and 7

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

This simple but incredibly powerful coding convention followed on from the work of Ada Lovelace in the 1840s. The realization that computers could not only interprete binary values as numeric values to be used in calculations but also as non-numeric symbols. In this case they represented the upper and lower case letters of the alphabet, decimal digits, mathematical operators and simple device control codes such as form feed. As a simple example the character 'A' in ASCII is represented by the binary number 01000001 (hexadecimal 41, decimal 65) so the key punch machine would punch holes in rows 2 and 8 of the current card column when the A key was pressed with no-hole meaning 0 and a hole meaning 1.

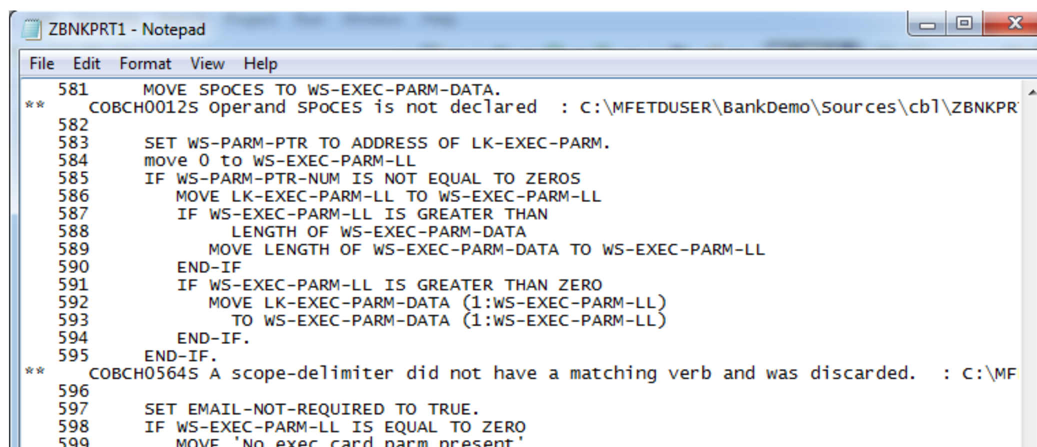
Given the very high value of mainframe computer time during this period it was important that it wasn't wasted finding keying errors in the COBOL source code card deck instead of running valuable production jobs. Usually a listing of the source deck was printed and the programmer would then 'desk check',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

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

Compiling Source Code Into Machine Code

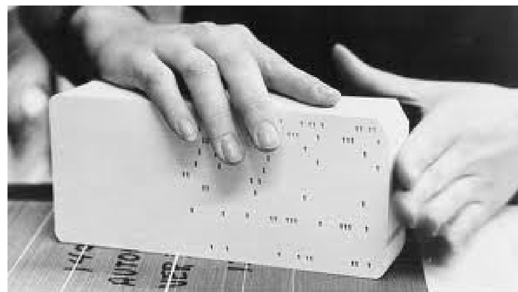
When the desk checking was complete the COBOL source card deck would be submitted to computer operations to be processed by running it through the COBOL compiler. Computer operations such as compilations were carried out by computer operators in the inner sanctum, the holy of holies, the computer room.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

As a result the turnaround on a compilation job was far from predictable but in due course, probably next day, the programmer got back a printout showing any compilation errors and the source deck.



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

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



Managing Your Source Deck

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

A Handpunch

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

Chads

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

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

- ‘7’ = 00110111 (hexadecimal 37)
- ‘1’ = 00110001 (hexadecimal 31)

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

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

Program Testing

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

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

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

The Batch Processing Production Environment

When program testing was completed successfully the program would be released into production and this could be a non-trivial and complicated task. The unit of work in the production environment was a batch job where each job consisted of a number of individual programs that ran one after the other. Each program processed its input data, usually produced a report, and then passed some of its processed data forward to the next program in the job in the form of punched cards or maybe on magnetic tape. A single job, say payroll processing involving our example program, was executed and controlled by a considerable card deck. The card deck could include a mixture of input data, program object code and JCL, the Job Control Language which instructed the operating system on how to execute the batch run (similar to a Unix shell script).

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

The punched card driven production environment was a completely sequential environment executing one card at a time and it was absolutely critical that the cards in card decks were in the right order. The computer operators needed to assemble the correct card decks and the management of punched card decks was a massive part of their job. Dependencies could exist between one batch job and another batch job and so operating procedures needed to be defined and documented to handle such dependencies.

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

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

A batch run card deck with operator's markings

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

A classic teletype with a built-in paper tape reader

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

The magnetic tape library

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

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

A disk pack consisting of separate disk platters

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

A disk farm

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

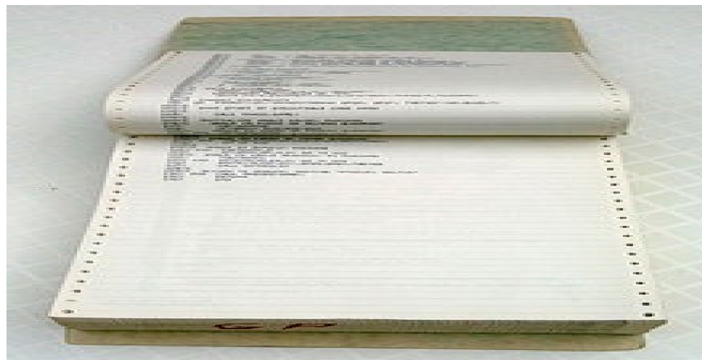
The Program Maintenance Nightmare

Once a program went into production, especially in large organisations, it would often be handed over to the maintenance programmers. My second programming job, taken to boost my salary, was as a maintenance programmer.

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

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

IF PRICE < 0 THEN GO TO PRICE-ADJUST.



A large program listing

Finding PRICE-ADUST could take minutes of visual searching.

This led to one of the many religious wars that sprang up around COBOL, the 'meaningful' versus the 'alphanumeric' paragraph name dispute. The dispute arose over the question of whether paragraph names should be ‘meaningful’ like PRICE-ADJUST, or ‘alphanumeric’ like A-120 and created in alphanumeric order so they would be easy to find. The most ludicrous supporting arguments were offered on both sides, which is usually the case in these religious wars, personally I settled for the hybrid approach and gave paragraphs names like A-120-PRICE-ADJUST.

With more and more commercial programs going into production, program maintenance nightmares like this got worse and worse. In 1968 Edsger Dijkstra published a groundbreaking letter in the American Computer Society’s technical journal entitled “Go To Statement Considered Harmful” in which he recommended the outlawing of GO TO statements and the use of more ‘structured’ programming logic. A dramatic oversimplification of the structured techniques that emerged is that they basically outlawed the use of GO TO statements and to create program logic they only allowed the use of three logical constructs which are now standard in all modern programming languages. Looping constructs such as DO … WHILE/UNTIL; switch or case statements such as SWITCH ….. CASE1, CASE2, CASE3 etc. and monitor/escape constructs such as TRY/CATCH. This eliminated the horrors of the GOTO statement. These three constructs have remained remarkably resilient and I’m not sure any better way of representing program logic has been developed at this time. Predictably it took several years for Dijkstra’s ideas to penetrate into the world of commercial programming.

The Programming Proletariat

Because it was an almost industrial production line process the development and operation of computer programs at this time bore a striking similarity to the role of industrial workers that had been described by Karl Marx in his revolutionary writings which, along with Che Guevara posters, were quite in vogue in the early 70s. Central to Marx's analysis was the fact that the 'means of production' were owned by the (capitalist) employers, not by the (proletarian) workers, so the workers were completely dependent on the capitalists for the means to do their jobs. A car production line, for instance, was simply too expensive for workers to own, only capitalists with large amounts of capital could own them. This analysis also held true for computer programmers since our means of production, the mainframe computers, keypunch units, line printers, disk farms, etc. were all owned by our employers and were completely out of reach of programmers themselves. In any computer project the hardware was by far the most expensive item in the budget whereas software was relatively speaking a cheap item. The idea that a programmer could possess their own means of production and produce their own software independently was simply unthinkable. The first commercial computer programmers were firmly positioned, if not trapped, in the early digital age's proletariat.

The Programmer's Revolution

It goes without saying that as a result of the radical transformation of programming environments that has taken place since the 1970s the opportunities now available to entrepreneurial programmers are almost limitless. It is now easy for a programmer to own the means of production and to produce their own working software. There has been a real revolution for programmers and to paraphrase Karl Marx, they have been released from their chains.

This article was written by Garth Eaglesfield based on an extract taken from his memoir 'The Programmer's Odyssey'.

For book details see http://theprogrammersodyssey.com

For further resources concerning the history of computer programming see http://spanitis.eu/links-articles