\SLCLUNCH.HTM (DoD Technical Talk)

INTERIM METHOD OF USING TWO CHARACTERS FOR FOUR-DIGIT YEARS

DoD Software Technology Conference
Salt Lake City, UT -- 1999 May 04

By Bob Bemer

Keywords: Y2K, Conversion, Bigit, Virtual Machine, Data-driven

[illegible]

Abstract: When nearly 40 years' of date data are "century-disadvantaged", and programs may fail around 2000 because of that, one can either:

a. Find and change all date data to 4-digit values, at the same time reworking all programs that use them, or
b. Rework only programs, so that assumptions of century values may be made (temporarily, and every time it is used), or
c. Leave source programs undamaged, find major date inputs and outputs, modify them to carry a century value still within two digits, and augment the object program with a virtual machine that can handle them with 4-digit properties.

Even then, much testing and validation lie ahead, which should be:

- Independent; foxes guard chicken coops poorly.
- Superfast, for little time remains.
- As different as possible from your own method.
- Working well enough to be a contingency plan if mishaps occur.
- Done with future dates, not just those prior to 2000.
- Helpful in converting date data to the right form eventually.

Method (c), looking from a data (not program) view, can have these attributes, and carries all of these dividends naturally.

GROUND RULES FOR THIS PAPER

You will have noticed that no slides and audiovisual aids are set up for this presentation. So think of me as a bit of an evangelist.

I am going to try to keep your interest and attention by its basic content, as evangelists do. Imagine one of them showing slides:

- Now this is a head-on shot of the gates of Hell.
- (click) This is our new reception room.
- (click) Here are the statistics on fueling our various hot rooms and torture chambers, by fuel type and rates of consumption.

Got it? This paper is about a qualitative problem, with just a few quantitative statistics sneaking in. Let's go!

FUNDAMENTALS OF THE Y2K PROBLEM

Coming into the year 2000 we have a unique problem, so let's review the fundamentals to get an unbiased picture.

We all know that computing is about programs and data. Either can be temporary (we throw it away after use) or permanent.

> E.g., some data occurs only on the way to an answer, and we don't need it. But often we use data at the moment, and also keep it for future uses. In a database.
>
> Sometimes we throw away the program, but not often. The year 2000 problem tells us that we should have thrown away more programs than we did!

We all also know that programs and data can be wrong.

> For programs we first make a rough test, and then put them to use, making further corrections as needed. But over the life (sometimes a very long life) of a program it grows and changes. Sometimes the original program was not annotated very well, and this gets worse as a succession of people get into the act.

Data can be wrong in many ways -- totally or partly. Either way that's not good.

> An FTC database once had three Standard Oil of Ohio vice presidents with remarkably similar names -- Wolgemuth, Wohlgemuth, Wohlegemuth. Spelled differently, of course!
>
> For the Year 2000 situation we have date data that is partially, but fatally, wrong. That is to say --
>
> > The year values are "century-disadvantaged"

You can see the hook coming from that use of "temporary". Obviously an effective solution to the Year 2000 problem is to throw old data away!

> But that solution is not good for people that save and reuse data, like the Egyptians recording grain sales on clay tablets two or three millennia ago.
>
> Among those objecting to this would be NOAA (they predict future weather from past), the Census Bureau, surely Social Security, hospitals, scientists, property records, genealogists and just about everyone else.

So if we're going to keep on using computers, eventually we're going to have to fix the data! For years, that means deciding once and for all what century they belong in. But we have a few problems in doing that.

> An identification process must precede the conversion process, whichever is chosen. And here is our first stumbling block.

Everyone tells the press that "dates are hard to find". Not so.

> What they should say is that "usages of dates" are hard to find in the program. And each date gets used 20-40 times. To contrary, "dates" are fairly easy to find, especially from the input/output formats. With such multiples, no wonder many usages are missed.
>
> Source code search for date variable usage may be just a little more difficult that identifying males and females uniquely by:
>
> - hair length or style
> - pants vs. skirts
> - earrings
> - companion type, etc.

Object identification is more direct -- specifics left to imagine.

SOME PRESENT STATISTICS

Before examining the various processes used to fix the problem, let's look at a situation that's near, if not dear, to our hearts. That is -- what is the Y2K problem costing the U.S. Government?

The Office of Management and Budget once said that the Federal Government would need some $4.7 billion to fix the situation. The figure keeps going up. Some industry experts say it could go to to $30 billion. Actually, everyone is now uncertain what Y2K compliance will cost, because each previous cost projection keeps getting superseded by a higher one.

Congressman Stephen Horn's (R. Calif.) report card on the Federal Government's progress on Y2K repairs says costs aren't the only concern. A survey he commissioned showed some departments so far behind that the State Department, for example, may not be done until the Year 2014!

Only three agencies received good grades: the Social Security Administration, the Veteran's Affairs Department, and the National Science Foundation. But the SSA and VA's progress may go for naught because the study also gave the Treasury Department a D-. Without checks to send, the SSA and VA aren't going to be of much use, no matter how well-prepared their own computers are.

The stats are grim, but you're all well aware of the magnitude of the problem. I'd like to challenge each and every one of you to ask yourself honestly: Are you ready? Given the need to interact with other agencies who themselves may not be prepared, and the conflicting daily reports, do you honestly feel you will be completely prepared at midnight, January 1, 2000?

If you gave yourself a report card, what would your own grade be? Have you seen your own systems running completely with aged dates? If so, have you seen the results when interfaced with other outside agency systems? Do you know for certain how many minor systems are interfaced with each of your mission-critical systems? Are all of these completed and thoroughly tested?

If you cannot answer all of these questions with certainty - with an A+ in all areas - you are not really completely ready.

Now to the repair processes at our disposal.

PROCESS (0)

Let's define process zero as getting all new programs, prefabbed with guaranteed full Year 2000 compliance. After all, the famous guru Ed Yourdon (who has labored long in support and defense of "structured programming" and such methodologies) has said about the present situation [1]:

> "We had the impression that not only would the hardware be thrown out in seven or eight years, the software would be thrown out, too."

This has been tried, and usually canceled. Standardized business programs are sold, but one pretty much has to change to their data forms, formats, and method of operation. No time remains. And that still won't heal your mountain of century-disadvantaged data. How can the century values be recovered for such data?

PROCESS (a)

Find and change all date data to 4-digit values, at the same time reworking all programs that use them.

This one is easy to dispose of. Unless you were as smart as Social Security, and started 10+ years ago, it is impossible to make this sort of fix in the remaining time. Period!

Not only must the programs be rewritten, but you need to set up a second team to dig into the database, make a best guess as to century, and expand the data. But not all started that early.

PROCESS (b)

Rework only programs, so that assumptions of century values may be made (temporarily, and every time it is used)

This very common method is what we have been reduced to, for several reasons. But one reason stands out.

Thinking is hard. We all know that. So sometimes we try to see if it can be avoided. One way is to look around at everyone else, and see if a substantial number of them have come to the same conclusions. If so, we adopt that thinking with little effort, because it is obvious that it is what we call "conventional wisdom". Wow! We've solved another hard problem with minimal thinking expenditure.

Conventional wisdom is not a modern invention. Thinking has been hard for millennia, to use that now-popular term. That is how it happened that the Romans all used lead to make the pipes that transported their water. Of course is wasn't very wise wisdom, as a large percentage of them died too soon, from lead poisoning.

Somewhat later, conventional wisdom was applied to investment. There was a time when the price of tulip bulbs could only go up, so the conventional wisdom was that if you bought at some price you could always sell for a higher price. I am just waiting for a NASDAQ IPO for a tech company called TULIP.COM. They don't dare float PONZI.COM.

Conventional wisdom always applies to where you build or buy a house. In movieland it's on top of a cliff facing the ocean. Of course when it rains hard your house loses a lot of altitude. In the Southeast the cliffs are few, so they build on the edge of the ocean, where inevitable hurricanes send them scurrying to get the Federal Government to reimburse them for their foolish choice.

Maybe this doesn't sound much like talking about a computer problem, but it is. Have patience.

As the certified first and second person in the world to warn publicly about the Year 2000 problem, first in 1971 and again in 1979, I have a certain proprietary interest in its effects, and what we're going to do about it. That interest is only heightened by the fact that, by inventing the Picture Clause of COBOL, I made it actually quite easy to use 4-digit year values.

But we didn't, in the main, except for the Mormon Church and some insurance companies, and I don't want to hear anyone putting the blame on the programmers again. Maybe a little blame, but mostly it was at management insistence. You might recall that in early days of computing the management knew less about computers than they do now, when there's been time for them to come from the ranks of programmers themselves.

Like Alan Greenspan, that self-confessed programmer of 2-digit years!

And I don't want to hear any more people or newspapers give the stupid argument that it was the fault of punch cards. Punch cards were really a very extensible medium, and if you don't want to buy that I can show you a way to scrunch a 4-digit year in 2 columns. It's a forerunner of my method.

And I don't want to hear that we didn't worry because the programs wouldn't be around that long. To borrow a phrase from the political arena, "It's the data, stupid!" Not programs per se. I mentioned the Egyptian clay tablets. Data persists! People gain economic advantage from studying it. Sometimes they improve their lives. But in the 40 years of COBOL we've been amassing "Century-disadvantaged" data!

Throw all those programs away that process 2-digit years, and we're still stuck with the data, and how to move it into 4-digit synch with our new programs.

So what I classify as "conventional wisdom" is "windowing". Although relatively faster than Process (a), which was doing expansion to 4 full digits for years, windowing has drawbacks. Among them are:

- It is an evanescent decision, having to be made over and over again. Never mind any slowdown. How can this lead to fixing the data for good?
- It depends upon choosing a pivot year. After a year and a half of asking, I still don't know an official pivot value for the U.S. Government. Medicare uses many different values. Hope they can remember which goes with which data and with which programs.
- It depends upon modifying source programs. Fine if you still have them, and they correspond fully to the object programs you're running, and documented so well that a newcomer can understand what they're doing -- Good luck!
- And modifying those source programs depends upon finding all places where 2-digit years have been used.
- Not to mention what you have to do to build up a 4-digit year form for all of

those over a half million interchanges of data that the GAO found in the Federal Government.

At this point I'll say that the goal of compilers was wonderful, but along the way we botched up two things:

1. Letting everyone into the act without proper training, exposition, and control, spawned programs that were not sufficiently self-explanatory to those that might need to understand them thirty years later.
2. Additions and changes were ordered by the owners as they found other needs and purposes. And these changes were not documented by re-editing the source program. Just as house builders, when they deviate from the plans, seldom tell the architect so he can revise them.

That's why we speak of "legacy". Grandma left us her Victrola, but we don't know how to get a new song to play on it. The writer Ellen Ullman, in her paper "Why Programmers Can't Fix It", says:

> "Over time, the only representation of the original knowledge becomes the code itself, which by now is something we can run but not exactly understand." ... "(there is ...) no one left who understands."

This is a serious deficiency applicable to windowing, and we'll go back to it in talking about object code fixes.

More difficulties of windowing are:

- It doesn't get you to 4-digits eventually. That entire process is deferred. Will we have another period of panic as any present effectiveness of windowing decays?
- It takes people to make the changes, and they work at people speed, which is not computer speed.
- We have now proven absolutely that this method, overall, introduces NEW errors to your programs. which is why at least 50% of total repair project time must be in testing.

On this last point let's see the statistics. Lines of Code are not a great measurement tool, but it's a much-used one.

INTRODUCED ERRORS

Many measurements have been made, by different groups, and not under the same controlled conditions. So results are obviously disparate. But so are programmer capabilities.

On the number of new errors in supposedly fixed and tested code:

- Ref. 585 found one new error in 166 lines of code.
- Ref. 640 found one new error in from 2 to 4 lines of code.
- Ref. 2568 found one new error in every 20 lines of date code when fixed manully, and one in every 200 lines for automatic tools.
- Ref. 3342 found the one new error average to be 22 lines of code, but it varied from 1 line (!) to 46.
- Ref. 3678 found the one new error variation to go from 5 to 200 lines of code.

One of the best papers on this topic is Ed Yourdon's "Deja Vu All Over Again" [2]. His statistics show about one error in every 10-20 lines of code supposedly tested after repair. This presumably includes both missed instances and newly introduced errors, as they're hard to tell apart.

He quotes Capers Jones on "bad fixes" averaging 1 in 25 lines of date code, which testing reduced to 1 in 83 lines, still leaving 30% bad.

But the question is whether your interlocked operations can afford this newly-bad code. And how many more errors would be found by the tester of the tester?

> Reminds me of the Persian poet-philosopher in the book "Hajji Baba", who had invented a perpetual motion device that needed only one more small addition to make it go 'round forever.

This problem became well-known to the press in just this last month and a half. It is why testing has always been figured to be at least as much work as changing the programs.

PROCESS (c)

Leave source programs undamaged, find major date inputs and outputs, modify them to carry a century value still within two digits, and augment the object program with a virtual machine that can handle them with 4-digit properties.

There are several ways proposed to do this, in various stages of the patent process. But patents are immaterial now. Time has overtaken. If anyone wanted to pirate a scheme it would not be possible to program it in time!

Our company has a product in this class, and while sales arguments may be found at our booth I am both inclined and adjured not to make such here. I'll be generic whenever possible. But four cautions are needed up front:

- Our particular product is limited to, and developed for, programs of IBM's MVS COBOL, on 360/370/390 class mainframes. As advertisers say, other restrictions may apply. But this makes for a pretty big battlefield in the war.
- We can't do a thing for PCs and embedded chips, nor for computers of other makes or marques.
- I admit that this is where my main expertise lies, together with my gut beliefs and instincts. I have no animosity to other ways, even as I point out their problems. For any doubting Thomases among you, I also admit that developing our solution took longer than we had anticipated and proved to be an extremely complex project, to say the least.
- However, this is not a problem today, because our solution not only works, it also runs up to 40 times as fast as other methods. In addition, it serves very well as a testing and contingency method, which is where most entities are now.

The principles of the method have been given on our Website.

The key ingredient is what I have called the Bigit, which is a character with a numeric part like any other, but where the zone part of the display character for the decade position is altered to reflect different centuries -- up to 6 of them, so you can get to 2199 now.

Why the funny name? It comes from the fact that when I invented the escape sequence I contrived to put it into the public domain so IBM would not have that monopoly. As a result I did not get much credit for it until recently, and in particular never made a nickel from it. So this time I sneaked my surname into it -- "Bigit" stands for "Bemer Digit"!

I must admit some skepticism has surfaced about this method. This may be due in part to my joke about a "silver-plated bullet" in the first article about it, in the Wall Street Journal. But --

> "Every truth passes through three stages before it is recognized. In the first, it is ridiculed. In the second, it is opposed. In the third, it is regarded as self-evident."
>
> Schopenhauer

Today should be third-stage day!

A MAJOR DISTINCTION IN METHOD

The remarks of Ellen Ullman lead to my dictum:

Fixing Year 2000 problems via source code change requires people to know HOW the source code logic works, and WHY. Many difficulties lie in this path.

Via object code, the computer already knows HOW it works -- that's what it is running! And it doesn't CARE WHY!

A DATA-CENTRIC ANALOGY

Think of the database as a book, and the several programs that operate upon it as reader groups, where each program can have several readers.

Finding the dates in the book is our job, and is not all that difficult. Finding every reader of the book in the source program is their problem, as each reader makes a "use of dates".

PROS AND CONS ON WINDOWING vs. OBJECT CODE

Definite Advantages of Object Code Solutions

- Data marked once and for all, as needed, via several info sources. It is the object code that get changed so it can deal with the marked data.
- Being permanently marked, no pivot year is needed.
- No modifying source programs for date purposes.
- Your programs are still in your secure environment.
- No particular need to find date usage in source programs.
- The computer is used to make most of the changes, and it works at a 10:1 to 40:1 advantage. This is one of the reasons Capers Jones dubs the method "the last hope of the laggard".
- Because the bigitizing process changes data to add century, it can at the same time add a differential of years, thus enabling testing with future dates.

Thought Bad, But Isn't

- When source programs must be modified for other than date purposes (like adding or changing functionality) simply run them through the Bigit process again at end of compilation. Added time is not even noticeable (a GIGA Group expert went public with a misunderstanding of this, which we deplore).
- Extra run time, because we instrument ALL instructions that might process dates -- whether they actually do or not. When real data is run through the programs, we can turn off this instrumentation for those instructions that do not process dates. This process is now operating very efficiently.

Neutral

- You'll still have to build a 4-digit year for some interchange, but this is done from the permanent markings. (The program I wrote had 8 machine-language instructions!)
- It can help to get you to 4-digits eventually.
- We have now proven absolutely that this method works as a high-speed remediation and testing product that fixes the problem so elegantly that, in retrospect, many people will no doubt be asking, "Why didn't I think of that?"

Drawbacks

- If you're using your COBOL source code for more than just the IBM family that the object code method is suitable for, then applicability drops off.
- If your data is shared with other applications, you'll need utilities that we provide to expand the compressed date to the format needed by the application.

THE MISSION-CRITICAL SHUFFLE

Pleading lateness, never admitting neglect, the general worldwide trend is toward triage. Some percentage of software components are labeled "mission-critical", because without them operating correctly there can be massive failures and seriously bad effects.

This leaves systems not so classified as relatively unimportant and ignorable. But Yourdon keeps warning of interconnection in the entire enterprise, and says you had better fix them, too!

Ian Hugo, the UK guru, in a private communication to only a few newspaper experts on Y2K, and others in critical places, describes this as a "movable feast that is primarily a factor of of what there is time and resources to do ..."

He says "It then gets a bit Orwellian, some applications being more mission-critical than others". He calls "mission-critical" a movable demarcation line through a continuum of application importance. It's placement at any one time depends on time left and potential for political embarrassment ..."

Of interest to this audience, he said that "the UK Ministry of Defence through 1998 was consistently declaring 10,812 systems needing fixing, of which 6000 were classified as mission-critical. Last December 5000 of the 6000 critical systems simply disappeared off the reporting radar screen ..."

Here is how the object code method can prevent this problem.

- Now that a much faster method is available, reconsider those classifications. Suppose you have 20 systems, of which 6 are ready to go. You report a 30% completion, which causes your management and theirs to jump on you.
- So you downgrade half, to 10 mission-critical systems. Now you're looking better, at 60% completion.
- But suppose you apply the object code remediation to 3 of the 10 axed systems, and they are quickly compliant?
- Put them back in the mission-critical group, be 69% complete, and accept the praises of your management!

By the way, Don Estes, a famous tester, reports that when top management comprehends the Y2K problem, the pension plan is the first Mission-Critical system that gets repaired.

HOW DO WE FIX IT PROPERLY?

The very best way, which I hope to see used in the future when we have passed this emergency, is to do all internal calculations in days, and only change to cultural date forms for input and output from and to people. People in the large don't calculate in floating point, but computers do, and for very good reason. Working internally only in days has the advantage that the earth rotates once every day no matter what calendar system you subscribe to, and there are many, in addition to the Gregorian.

Doing internal calculation by days is a common denominator like communicating in ASCII, and then switching to EBCDIC or Kanji or a full set of Chinese characters. It's rather like the way all international airline operations are carried out in English, for safety.

I have proposed a way. It's called XDay. You can see it as a part of our product, looking forward to "do-it-right" time.

A WORD ON TESTING

Testing is the biggest Year 2000 trap. After you've changed your software to work after 2000, can you prove inexpensively that it works?"

- You've probably changed your source code. You may have introduced new errors (what an understatement!).
- You need the most independent check you can get. If you have used windowing, nothing is more independent and effective than changing the object code, to see if both methods get the same answers.
- The Bigit method is cheap in cost, cheap in total effort by a factor of 10 or more from what you would have expected. Cheap in saving your business by the deadline.
- With this method you can check future answers as well as today's.
- If your other efforts need rework, and there is no time to do so, you have a contingency solution in the test method.
- The method is called Vertex 2000 TM. You may have already heard of it in the (inter)national press, and on national television.

CASE STUDIES

1) Remediation

The product based upon these principles was used for remediation at the Hacienda, the Puerto Rican equivalent of the Treasury Department. Repair was over a 4-month period. In a videotape segment of "World Business Review", hosted by Caspar Weinberger, Chairman of Forbes, an executive with Hacienda avers that they could not have fixed the problem any other way. Unless of course they had started 10 years ago, as the US Social Security did.
A copy of this segment may be requested from Bigisoft at its booth here.

2) Testing Other Repairs (IV&V)

The product in its testing mode was used on a program from a US Government Agency. After a visit we were given a Batch COBOL program of about 3000 Lines of Code, using 2 input and 5 output files. The programmer for production support of this program helped with step A below.

A. Thu -- 30 min onsite, locate dates in the input data (*)
B. Fri -- 4 hr to move application to Dallas MVS site
C. Fri -- 30 min of mechanical remediation
D. Fri -- 2 hr 30 min of regression and compliance tests
E. Fri -- 2 hr to prepare jobs for demo to agency
F. Mon -- 45 min for Agency site demo retracing the remediation steps, with several questions interspersed

* Viewing the output reports with bigits present, where a date without a bigit showed a missed input date, showed that the application programmer had not pointed out one input date.

RECAPITULATION

In conclusion I'm going to follow the classic advice to speakers. "Tell 'em what you've told 'em"!

Source code methods have certain drawbacks. Object code methods have some advantages to consider very carefully. Here are the pro and con features:

Can't Scrap Data

- the easy way to fix Y2K is throw away the data, and have no more problems
- is this feasible for you?
- might install new programs from outside -- can you do that in time, or change your operation mode?
- are outside programs safe? Gartner just said 81% bad!

Speed

- can do all - no triage - beat deadlines with confidence
- fix more systems
- avoid the non-mission-critical trap for systems that may have been overlooked or intentionally written off

Deadlines

- you have time left (Yourdon uses a late runner analogy as I do)
- cost-effective alternative for late or failing systems
- if n times as fast as others, you have n times that many months until the deadline

Resources

- minimum, as people are scarce
- minimum, as computer time is scarce
- low dollar drain
- parallelism with other methods

Security

- no outsiders need be introduced
- no source exposure
- source code changes optional only
- non-intrusive

Reliability

- COBOL constructs are limited, and so are analyzable

Adaptability

- completely IBM environment
- well-documented
- no new methods or jargon
- learn fast (our 8-hour course)

Contingency

- solves any problem with SEC regulations
- serves both test and contingency purposes
- suppose you try your way, and it's a near thing, but no cigar -- what will you do if time runs out?
- experts all advise testing and a contingency plan
- what if secondary (non-M-C) programs fail unexpectedly?

Testing Dividend

- V2K gives 2 for 1! Validation AND contingency, as the test itself provides a backup solution!
- V2K is like crossfooting on a spreadsheet
- orthogonal check on same data is best possible

Precedent

- COBOL object optimizers
- ASCII-EBCDIC autoconversions
- you're going to have to change the data anyway, to 4-digit years, to get it right!

Social Security changed to 4

Transition

- book and readers
- windowing is impermanent, introduces new errors
- today's incoming stats show incomplete fixing -- are new crises coming?
- windowing doesn't FIX the problem
- Bigits are permanent and unambiguous marking
- on IV&V, can ignore your changes
- introduce no new errors in logic

New Errors

- source code, documentation, and design knowledge lost
- statistics on present situation indicate danger
- to fix for Y2K, must know what the program does and why! Your computer WAS running it -- it didn't have to know what the program did, and it didn't care why!
- if it ran OK before bigitizing it probably still will
- the test period is drastically reduced

Public Relations

- be a promise keeper, with V2K help
- be a hero/heroine to management and the public
- make the press releases be true

Finally -- it's still not too late, and V2K gives you an edge -- hurry!

Support -- IBM is campaigning for proper testing (avoid suits)

REFERENCES

1. D. Hayes, F. Buller, "Chances to Fix Problem Frittered for Decades", Kansas City Star, 1998 Sep 27, p. A-18 lead.
2. Ed Yourdon, "Y2K Software Projects: Deja Vu All Over Again". http://www.yourdon.com/articles/y2kdejavu.html

GARY NORTH REFERENCES

- [585] 1997 Oct - Inevitable Errors, Inevitable Shortage of Capacity http://www.sentrytech.com/sqt1.htm
- [640] 1997 Oct - The Number of Errors Per Lines of Repaired Code
- [2568] 1998 Sep - Thorough Testing Costs Four Times What Remediation Costs
- [3342] 1998 Dec - Supposedly Fixed Systems Turn Out to Be Error-Filled http://www.softwaremag.com/Nov98/sm118dt.htm"
- [3678] 199- http://www.itpolicy.gsa.gov/mks/yr2000/y2kconf/papers/paper59fp.htm

SPEAKER BIOGRAPHY

Robert W. Bemer

In 1999 March, Bob Bemer marked 50 years as a computer programmer. He was Manager of Systems and Software Integration for General Electric, head of Software and Field Support for Bull GE in Paris, Director of Systems Programming for UNIVAC, and IBM Director of Programming Standards.

He created the ID and Environment Divisions, the Picture Clause, and the name of COBOL. Known as the "Father of ASCII", he invented the Escape Sequence allowing the world's symbols to coexist. The "software factory" concept is his, and his team built the second FORTRAN processor.