Feb 13 04 Vedbaek

Thank you. It is a great pleasure to be here.

I am big fan of history and the folks here are doing a great job in preserving history. It is often said that people who do not study history are bound to repeat it – and this is almost always presented in a negative light – it is said about the wars and strife of years gone by. But we also know that we want to study history because the people in the past actually did some great stuff that is still applicable today (or in some cases better than today) and it would be sheer folly and waste to forget about it.

Also with the knowledge of history, we can better identify historical trends. By identifying the trends honestly and exactly; maybe we'll be able to better predict or shape our future.

For honesty we need artifacts and witnesses and we need to study what they tell us. This is why a society such as the data – archeologists is so important and I very much applaud and support their efforts and I only wish they would have a name that is easier to pronounce. Round of applause for the data archeologists.

The title of my talk refers to the Rosetta stone, so I owe you a picture of it here. You can see it in the British Museum and of course it is very important in that it contains the same text in two languages and three different alphabets so it was key to the deciphering of hieroglyphs.

The metaphor will be of course that the Gier Algol compiler code occupies a similarly central place in the relatively brief but fast changing history of computer software. I hope I'll be able to show you how meaningful and even visionary this software was and how we are still decoding the lessons from it today.

But to appreciate any artifact we have to put it first in historical context.

Last year we celebrated the 100th birthday of John von Neumann, the great mathematician, who worked at the Institute for Advanced Study in Princeton. (slide) Here we see a copy of the first meeting of the electronic computing

project, where the structure of the modern computer started to emerge with the crystal clarity that only a genius of von Neumann's caliber can create.

This idea was very important: von Neumann told his colleagues at the time that he was working on something much more important than the atomic bomb – which was his previous project. The trustees of the institute allocated $100K (slide) to the project even as sugar was still rationed (slide) and despite of the tradition of the institute not to engage in experimental science the IAS computer took shape. (slide)

Together with the expensive computer came expensively prepared software. When I look at this prehistoric software plan I get the same feeling as when I see a beautiful steam engine in the Deutsches Museum in Munchen or the Science Museum in London where even the handrails were machine turned with great care. Our ancestors when they created something new, important, and precious, they dressed it up very the best they could. Here notice not only how clear the depiction of the program flow is but also note the mathematical notations: the absolute values, the subscripts, the Greek variables.

One of the countless contributions of von Neumann to the world was his placing into the public domain not only the basic ideas of the stored-program computer but also the detailed engineering plans for the IAS computer. Many copies of this computer were made around the world as shown in this list (slide).

Please note that the Swedish Besk and the Danish Dask are more-or-less direct descendants of the IAS computer. This is also evident from the general architecture - 40 bits of data with two 20 bit instructions per word.

Now we are here of course in celebration of Dask's 50th birthday – another round birthday. Here are some slides from a talk by legendary Danish programmer Jorn Jensen. Jorn acknowledges the relationship with Besk, but does not mention the connection with the IAS computer – perhaps it was not known to the designers. Several improvements were made to the architecture, (slide) apparently including an emergency backup abacus.

Here we see a drawing of the operator's console of Dask and the 40 bit / 20 bit inheritance is clearly visible.

(slide) I have to admit that I've never seen a Dask but I lived in the time-warp of Hungary so only 10 years later in 1964 I had the pleasure to study in detail and work with the nephew of Dask, the direct offspring of Besm on the list, the soviet made Ural 2 with exactly the same family traits: 40 bits of data, two 20 bit instructions per word. This was a fantastically beautiful machine with the soft and warm glow of the thousands of vacuum tubes and the hundreds of flickering orange gas-filled indicator lights.

Notice the similarity of the consoles, and the single person sitting in front of the computer. The Ural was a personal computer, just like the Dask – you could not own it, or carry it, but when you used it; it was all yours alone – no operating system, nothing came between you and your machine.

By incredible coincidence, the performance and the capacity of the first PCs in the late seventies were very close to the Dask or the Ural: around 100 microsec execution for a 40 bit operation and about 5-10K bytes of RAM. So the techniques developed in the use of this 1950's generation of machines were – at least for the late seventies, these ancient techniques were directly applicable to writing PC software.

(slide) Here is a listing of my first compiler for the Ural with 20 bits of octal code per line. We did not have a printer so I never had a listing, I had to work from hand-written notes only. This listing was finally made much later in America in 1972 mimicking the format of my 1964 notes – the printer was XGP, a pre-prototype of today's laser printer.

(slide) But getting back to the 60's, after working on the Ural for a year, in 1965 I gained access to an offspring of Dask, the Gier. It is only recently that I become aware that the Ural and Gier – my very first two computers - were in fact second cousins. The family resemblance was fading a little: Gier had 42 bit words because of two new flag bits were added and it could have one 40 bit or two 20 bit instructions per word, so at the time I did not notice anything unusual – I thought all computers looked like that.

The people in front of Gier were very important to its history. On the left we see NIB, the director of Regnecentralen, a major visionary force in Danish computing in the 60s. On the right, Professor Peter Naur, the great Danish computer scientist, the editor of the incredibly influential Algol 60 report and the designer of the Gier Algol compiler.

(slide) Algol was not the first high level language, but it was the first that was defined with the mathematical precision and the generality, one might say generosity, that was lacking from the programming practice of the time. In a way Naur, with his name misspelled as the editor of the Algol 60 report, served the same complex role – combining the talents of a mathematician, an engineer, and a teacher – that von Neumann played in his famous report.

So influential was Algol 60 – that one scientist called it an improvement not only over its predecessors but an improvement over its successors as well.

NIB recognized the importance of Algol and supported the development of a compiler and run-time system for it. This effort was led by Peter Naur who was helped by a very able team (slide..) and especially by Jorn Jensen on the left side whose handiwork we will look at today.

I studied the Gier Algol compiler code in Hungary and Mr. Beck and Prof. Naur kindly offered me a job at Regnecentralen starting my carrier in computing. Here is my passport photo at the time not yet 18 years old. (slide)

And here is what I was studying, the Rosetta code. When I show a portion of the Rosetta stone properly aligned, I think you can see what I mean. On the left side we have the priestly, the so called hieratic language, the difficult-to write hieroglyphs, and on the right we have presumably the same message in a newfangled low-brow, easier to write so called demotic language, what we call today high-level language.

Another interesting parallel is that when the writing is difficult, the person of the scribe is very important. (slide) So here we see that on the Rosetta stone the scribe is mentioned by the beautiful scribe symbol – an inkwell for black and red ink with a connected pen – very near to the king's name – which is marked by the so-called cartuche. Similarly the compiler group we just saw, included a scribe, Kirsten Andersen, and her presence made it possible to create a beautiful and enduring artifact that we are examining today. And of course I would call Prof. Naur the king of the enterprise.

So now that we have the historical context, let's look at the Rosetta code in detail. The listing is over 40 years old – which is a lot in computer years.

As I mentioned, the code on the left is the assembly instructions for the Gier. The portions on the left enclosed in square brackets and the portions on the right following the semicolon, as indicated by the green line; are comments – that is they are ignored by the computer and they are there only for the human reader.

So, even though the code on the right looks like a computer program, it is really not, it is called pseudo code and it serves to explain what the real code on the left is doing – so ostensibly we have two descriptions of the same algorithm at hand – a Rosetta code. The care and craftsmanship that went into these comments, let alone the code is truly amazing. It is especially amazing if you consider the sheer difficulty of editing the code on a ten character per second Flexowriter, using 8 level paper tape that is copied as the edits are made in sequence.

This style of commenting was used mostly at Regnecentralen, and it expressed a hope in that the high level language Algol is better suited in explaining the workings of a program, than for example a natural language such as English or Danish would. Other projects at Regnecentralen followed this pattern, for example here is a portion of the RC-4000 operating system that was developed by PBH, PK and myself – but without help from Kirsten Andersen, so it does not look as pretty.

(slide) The code fregment we are looking at is in the parsing phase of the compiler, in particular the calculation of the value of a floating point literal which has a signed exponent. (slide) Algol 60 introduced a very pretty symbol which looked like the numeral ten in the subscript, to separate the mantissa from the exponent in constants (slide) – this required an expensive upgrade of the teletype equipment. In America, Fortran was used instead of Algol. In Fortran one could write the capital letter “E” for the same purpose – less pretty, less elegant, but a much more pragmatic and typically American solution.

(slide) So here is the processing of the exponent on the right where the value of the exponent is in the variable called exp10. Do not forget to ignore the blanks in names and also notice the beautiful and expensive multiply symbol – it is not just a capital X, but another special character. Today we would use the commercial asterisk for multiply and we gave up the blanks in names so that we do not have to underline or otherwise distinguish the reserved words.

Also note that the symbol := was used for assignment – strangely they did not create an arrow symbol for this purpose. Today we use = for assignment and have a lot of bugs because we often confuse equality with assignment.

So here we have loop exp10 times and multiply the number R that many times by 10 or 1/10 depending on the sign of the exponent, as required by mathematics.

Each line on the assembly code represents 42 bits of information – one machine word. It is quite amazing how tricky and short the implementation is, for example at the arrow on the left, we have a single multiply instruction that corresponds to a multiply or divide by 10 depending on the sign of the exponent as it should be, and as it is clearly indicated in the Algol code.

The text in square brackets that the arrow points at is just a comment. The letters “mkf” mean the operation code for floating multiply. The next instruction is “hv”, a goto, or hop in Danish, marking the end of the loop. But where is the if, the conditional? How is this possible?

(slide) The answer is at the place where the sign bit is set, at the upper arrow. We see there, that the actual address of the constant 10 for positive exponents or 0.1 for negative exponents is jammed into the address field of the multiply instruction. Sure it is non-reentrant self-modifying code that could not work in the modern architectures, but it is effective.

Today we would call this an extreme form of hoisting the “if” expression from an important loop all the way up to the two places where the Boolean was set. We may wonder if this was a good idea or not. It was, in the context of the early sixties.

So the comments were necessary not to explain what the code actually did, but rather what the code was intended to do. Comments are for expressing intentions. (slide) On this basis we can actually distinguish good and bad comments. The Rosetta code had good comments. Modern code all too frequently has not so good comments or bad comments. The bad comments simply repeat what the code is doing.

By looking at the comments we can find places where the same intention may be implemented in different ways, depending on the circumstances. We just

saw a Boolean value implemented so that the address of the constant 10 was used to represent the true value and the address of 0.1 was the false value. We do not have to look very far in the scroll to find two more examples of Boolean implementations.

The upper example shows the straightforward implementation with a 10 bit long 0 value being stored to represent false. The lower example sets the 8 bit or the 4th bit from the right in the I register under a mask to represent true. Here I would have put in an extra comment to say that -9 was in fact the logical negation of 8, but no matter. Joern Jensen who wrote this code was like the Shakespeare of coding, he had a very large vocabulary, and he could use different idioms for the same concepts, depending on the situation. Note that the Algol comments remained straightforward in each one of the three cases.

What did all this trickery buy us? (slide) For comparison, today we would simply write the code on the right side and wind up with the compiled code like this. This code takes up about 5 times the number of bytes – which is OK since my laptop machine here has two hundred thousand times more RAM memory than the Gier, so I am still ahead by a factor of 40,000. But 5 times more memory would have made Gier Algol unaffordable at the time.

(slide) But why is this still important? As I mentioned before, another fascinating thing about the comments in the Rosetta code is that they did not have to compile so that the programmers could basically enhance the language – in this case Algol - in any way they saw fit. So by looking for such enhancements we can gain insight of their desires and expectations of the future! Of course we know how the future turned out to be, but my point is that if our former selves successfully predicted the past future in their comments; our present future can be also better predicted using the same method.

The first example shows the spontaneous introduction of something like an enumerated type or maybe a character constant. There was no such thing in Algol, but Joern basically said, I do not care, I just want to express the intention that the assembly constant d14 in fact is an implementation of a carriage return. Joern did not have to explain this new language feature – we are in the domain of compilers so a domain concept should be self explanatory.

It was too bad that in the 1960s good people had to wait 10 years before language technology caught up with their normal needs, such as enumerated types or character constants. But I believe this situation has not changed insofar as we have today too many needs that will not be fulfilled for the next 10 years unless there can be a paradigm shift.

(slide) The next example shows another use of the notation that is clearly an enumerated constant, here used to designate a meta-language token. This is done today legally and routinely. (slide) However the next example for a meta-language token, maybe written by another programmer - is even more aggressive: here the programmer uses underlining as the quoting mechanism to denote the internal token value. So we have the use of formatting for delimiting – a concept that is not in any computer language today – even though it is very common in our informal communications, for example the use of indentation for grouping ideas or the use of fraction lines to define the operands of a division.

(slide) In this example we see another lovely Algol operator for the infix inclusive OR operation – it is not the letter V! Consider that people have spent hundreds of thousands of 1960 dollars on custom hardware to get this quality of notation. Today, with out bitmap displays and Unicode character sets this and much better notations could be had for no extra cost – yet we do not use them.

From the intention we see that the condition involves our friend, the boolean called “introuble” that was set with the 8 bit. (slide) You have to be a machine expert to see that it is checked using the QA flag in the indicator register. It would have been nice to see a comment to that effect there. With my modern editor I can easily make the change, so some things have improved, I am happy to report.

(slide) But the real important point here is the use of a comment inside of a comment! The programmer, in effect, lost his nerve, he has chickened out. The comment simply says that alarm 53 prints the error message that says “delimiter error”.

(slide) Strange how the programmer is avoiding the use of a string constant which was in fact in the Algol language, although its use was left undefined, and also the Flexowriter equipment did not have the good looking nestable string quotes that Algol demanded – so maybe that was the reason. Today we

have are no longer concerned about nested strings and use directionless string quotes.

So this example of a comment within a comment is important because it reinforces our hunch that intentions are first put into comments and that there is always a hierarchy of intentions.

Which raises the question: Now that today we can in fact write the program in Algol or some other high level language, what language should we use in the comments? The point is that abstractions form a staircase, for every level, there may be a level below – which we may call an implementation, and a level above – which is a more intentional level.

(slide) For the Gier Algol programmer, the next level of intention above was in fact the hand written notes that were attached by a paper clip to the code listing. This is the third language of the Rosetta code – the intention behind the Algol behind the assembly code.

Today, parsing is one of the few domains that is well understood and we can in fact automatically generate parsers not only from a table like this but also directly from the syntax equations which are at an intentional level one higher than depicted on these hand-written notes, (slide) and which could be copied directly out the Algol 60 report.

(slide) So we have seen a number of examples of the trend that comments contain intentions and there is always a next level of comments and therefore a next level of intention.

(slide) Unfortunately, outside of a few scattered examples of domain specific languages, the state of programming has not changed much since the Rosetta code which is quite shocking in view of the explosive developments in hardware where we are ahead of the Gier by 5 and 6 orders of magnitude.

And what is even more shocking about these programs is that they do not give any hints as to what the problem is all about. You may have heard of steganography, the secret writing where the secret message is mixed into a very large amount of foreign data, such as a photograph, so that it is hidden like a needle can be hidden in a haystack. Programmers today are unwitting steganographers, they effectively hide the information about their problem by

mixing it up with thousands of lines of implementation detail, and this self-defeating activity makes programming the bottleneck on the digital horn of plenty.

I'd like to conclude the talk by mentioning briefly how my company Intentional Software Corporation is approaching this issue, and how it relates to what we learned from Rosetta code.

A key fact that tends to be overlooked – is that the purpose of software is to implement the intentions of its creators. It is overlooked perhaps because this is not yet actionable; meaning that there is nothing we can do to about it.

(slide) Von Neumann was very aware of this and in his last book on computers, published in 1957 after his tragic death – he consistently refers – to the problem to be solved in intentional terms. In contrast he never used the terms program or software.

But as we saw in the Rosetta code the intentions are found at best only in comments or in notes where they can not be processed by a computer.

(slide) This slide summarizes the current process where the computer activity is limited to a compilation of the high level language code – into the software product. The code is the only portal into the nirvana of machine processing, the right side where things are a billion times faster, cheaper, and more reliable than the left side. And the programmers are the gatekeepers of the portal, protecting the portal almost as a priesthood; they protect the portal from the demotic influence of subject matter experts and the other stakeholders who really represent the users and their intentions.

Intentional software means that the intentions and the design become the code, or if you want to say it gradually, the code will look more and more like the design. Programmers will not work directly on the code, but will focus their expertise and energy on a generator program that makes the design runnable.

(slide) This is called generative programming, and it is quite different from current practice. The SME's gain direct access to the portal using a very fancy editing tool that is like a supercharged PowerPoint or one of the CAD tools that are used so successfully in the manufacturing industries. Because the SME's are not programmers we do not ask them to write programs. They simply have to

enter their problem using the terms of their own domain, their own notation. So we are not talking about end-user-programming; we are not asking the hospital administrators to learn Smalltalk or Java so that they can implement their concerns and administrative processes in some magic computer language.

(slide) Here is an example of a contribution that could be made by en expert in just-in-time manufacturing. The table shows part numbers and car types and model years. It looks messy because the domain, the real world is messy. A program expressing this intention would include at least this mess (hopefully correctly) and the program would also have to include the mess expressing the implementation environment and this mixture would result in mess squared or mess cubed.

(slide) The similarity between the JIT example and the third language of the Rosetta code is intentional.

(slide) So if the code is not a program, how can you run it on a computer? Well it will be mechanically combined by the programmer's contributions using the generator that the programmers specifically created for this domain, for this problem area. This can be done if the input is precise and consistent – but all professionals are already precise and consistent, it is not like we ask them to learn something strange, like programming.

(slide) So the programmers are not made redundant, but they are freed from their role as gatekeepers and bottlenecks. They are freed from domain work. Indeed, the flip side of asking SME's to learn programming, is asking programmers to learn about domains, to become domain experts. Both requests are unreasonable and can result only in inefficiency and low quality.

Programmers are also freed from repetitive work, because their work results in a reusable generator, not expressed as repetitive manual process. Now they can focus on their own area, which is software engineering, on better reliability, performance, on engineering innovation.

SME will have much better control because they can directly affect the details of the software. In most cases they will get the program with their desired domain change or domain improvement at silicon speeds, at silicon cost: in seconds instead of weeks, for price of milli-cents instead thousands of euros,

and without implementation bugs instead of the bugs that a direct programmer interaction would inevitable create.

To read more on Intentional Software, please enter “Biggest Damn Opportunity” into Google. Thank you for your attention. (slide)