••••••For CW Consideration June 22 1989

THE ETERNAL PRINCIPLES OF SYSTEMS ENGINEERING

by Tom Gilb,

Independent Consultant, Tom @Gilb.com
and Ormerudveien 4C , N-1410 Kolbotn Norway. Tel + (47 66) 801697, Mobile +47 92066705, website (where this paper is) www.gilb.com

INTRODUCTION

The normal function of this invited lecture is to attempt to take a look at the future. I know a few things about the future. It will surprise us. Some of the predictions about it will be wrong. You should not place too much trust in predictions about it.

When I was about 25 years old, I had already experienced my knowledge being “obsoleted” by three new generations of technology: Punched cards, IBM 1401 disk drive solid state computers and IBM 360 operating system computers with high-level languages.

When I was 25 years old, this was fun - learning new complex systems all the time. But I began to wonder whether it was good for me in the long term. Would I really want to learn totally new complex languages and systems when I was fifty years old? Would I be competitive with younger virgin minds in learning new things then?

My sister gave me a clue by sharing a book she was reading. It was by Jevons, an Englishman, called "The Principles of Science". The striking thing about the book was that it had been written about a century earlier - and in spite of rapid change in the science area, it remained totally unchanged, totally un-updated.

I began to realise that there were two kinds of knowledge: those things with a short "half-life", and those with a life-long usefulness. I began to realise that I should invest some more time with the life-long types of knowledge. I also knew that I liked being in the computer industry, and so I specifically wanted long-life knowledge related to being in the industry. I could find no obvious teachers, books or guidelines, so that what followed was a long term process of self-education, which continues even today, and is quite exciting. For example last year, I experienced that a concept known as "Fagan's Inspection" which was developed for software engineering quality control in 1972-4, was applicable on a large scale to all manner of aircraft engineering drawings and flight test plans at Douglas Aircraft. Nobody had ever done this before. In fact I made another recognition in this same connection: aircraft engineering drawings are also "software". They sure aren't airplanes yet! So any method or process which applies to "software" might apply to them. It is

really exciting to discover the power of solving difficult problems by using eternal principles and methods. I would like to share the following with you:

1. The concept that some kinds of knowledge have more power and long term value for you.

2. Some specific examples of those principles.

THE ETERNAL PRINCIPLES

The very first principle would seem to be that:

"SOME KNOWLEDGE LASTS LONGER"

The consequence of this is that:

1. we should actively try to distinguish between short-term knowledge and that knowledge which we can use for the longer term.

2. new problems, with new technology, can be dealt with to some degree by using these "classical" ideas. Direct, personal and extensive experience of new technology will not be the only way to understand and control it.

Are students being taught this distinction in their computer science studies? I think not. My conclusion is that both students and teachers need to make a personal decision to understand the distinction between "temporal" knowledge (that which becomes obsolete in time), and "classical" knowledge" (that which tends not to become obsolete with the passage of time).

THE VALUE OF KNOWLEDGE

Obviously an obsolete principle or piece of knowledge (like the cost of a piece of software last year) holds little value to the individual. A piece of knowledge which may very quickly become obsolete (like whether an airplane you are going to board has a bomb on it) may have extremely high value for a short time period only.

So validity of knowledge (whether it is still true or not) is not the only criteria upon which we must base our decision as to whether to learn it or not. We must consider the potential value of the knowledge.

The principle for this is:

LEARN THINGS WHICH WILL BE VALUABLE

This poses a problem. How can we know what will be valuable many years from now, with changing technology and changing values in society?

MY GRANDFATHERS FILM PRINCIPLE

When I was about ten years old, my mother confided that my grandfather, Philip, had a habit of refusing to see films which were less than twenty-five years old. He reasoned that if they were still being shown, they must be worth seeing. I thought that he was quite strange at the time. But, having wasted too much of my precious time watching unmemorable television and films, I can now appreciate his point. Age is not the only principle we can use to separate potentially useful from useless ideas. But it is a helpful starter.

I recently picked a "new" book (1985) to read. But its title "THE TAO OF LEADERSHIP" (by John Heider) indicated that it was based on ideas from the "Tao Te Ching" (of about 520 B.C.). It was indeed worth the brief time required to read it. One of its specific recommendations was that one should spend more time reading the "classics".

If something is simply "old" it may be useless to you. If it is old, and people have seen fit to reissue it, then that is a sure sign that you may find value in it.

I have not seen any attempt to republish the IBM computer manuals of my youth. But the philosophy ideas of Rene Descartes, which I learned at the same time - about tacking big problems by dividing them into smaller one - has proved enduringly useful knowledge. I was able to use them immediately in organising my computer programming (no advice on "structured programming" was given in 1960, except by Descartes). I have been able to use Descartes advice on almost all my problems since that time.

SOFTWARE ENGINEERING AND SYSTEMS ENGINEERING

I believe that software engineering is a sub-set of system engineering. It must be so because software is itself only one component of larger systems. Software has no value except when it is functioning in systems with hardware and human beings. The fact that most software engineering today is not true engineering of any kind - it is merely computer programming with a fancy title - does not detract from that observation. Here is the most general statement of the principle:

SOFTWARE ENGINEERING IS A SUBSET OF ENGINEERING.

I have found this principle powerful in identifying useful knowledge for software engineering.

Here are the underlying principles:

SOFTWARE ENGINEERING IS A PART OF SYSTEMS ENGINEERING

SYSTEMS ENGINEERING IS A PART OF ENGINEERING

ENGINEERING IS AN ITERATIVE PROCESS OF TRYING TO MEET RISK-FILLED DESIGN OBJECTIVES USING DESIGN PRINCIPLES.

(I owe this insight to Prof. Billy Koen of University of Austin, although the exact formulation is mine)

The consequence of this set of principles is that engineering principles and processes which have been found valuable in engineering disciplines for many years, are probably worth learning and transporting to the software field for our own use.

One of the most powerful illustrations of this is in "Fagan's Inspection" process. It has been largely responsible for a number of remarkable results in software engineering. For example it was

the major force behind IBM Federal Systems Divisions "zero defects" result in the last six Shuttle missions of 1985, for over 500,000

lines of code of real time software. Yet, when we trace the history of Fagan's inspection, we find that it is directly (according to Fagan, and any deeper analysis) related to the statistical quality control technology of Deming, Juran and Shewhart - going back to at least 1920.

SOFTWARE ENGINEERING PRINCIPLES

There are, I am afraid, too many people who still assume that "software engineering" is primarily "programming". Of course if the entire world defines it that way, it is. However I must admit that I refuse to condone such misuse of the the term engineering. I will therefore only speak with regard to a proper engineering discipline as it applies to software.

THE MOST POWERFUL SOFTWARE ENGINEERING PRINCIPLES

I would define the most powerful software engineering principles as those which have shown clear practical large-scale and real-world results up to now. Results which can be documented by name and date and extent of the accomplishment.

The collection of principles is so interdependent that it is difficult to say whether they are independent principles or are all part of the same concept. Let me try to express the highest level in as few words as possible:

EARLY PROCESS CONTROL

The problems encountered in industry are not "to write a program". The problems are how to get software systems working at acceptable quality levels, within limited resources (time, people, money, hardware).

The problems are in some cases solvable by a sufficiently ingenious super-programmer. But if such a genius is given improper task definition, or if the task exceeds what any one person can do alone, then we still have problems.

Benjamin Franklin, in Poor Richard's Almanack, over two hundred years ago captured one of the principles we are beginning, only beginning, to appreciate in modern software engineering.

"A STITCH IN TIME SAVE NINE" ("AN OUNCE OF PREVENTION IS WORTH A POUND OF CURE")

The key principle in software engineering seems to be that any way in which we can discover requirements definition problems, design problems, or implementation problems early, we will benefit from that by far more than the cost of learning of our problems.

EARLY DETECTION BEATS LATE CORRECTION

This seems to be the powerful principle in a world constrained by finite resources, which requires reliable systems. It would not be important if cost of building and maintaining a system did not matter. It would not be important if the quality of a system was immaterial.

THE HIGH OPPORTUNITY TO DISCOVER PROBLEMS EARLY IN SOFTWARE

In software engineering it has been shown that even in military quality systems, over two-thirds of the bugs found in operation of the software were traced and found to have existed in the systems documentation before the code was even written! The opportunity to clear up the dirt is there, but our fault detection mechanisms are usually too poor to find the problems which we then must suffer later.

(Ref. Boehm: Software reliability, North Holland 1978).

THE HIGH COST OF NOT BEING EARLY

Professional software development must be organised into a series of activities which refine initial customer or marketing product objectives into increasing levels of detail until a complete and useful product is available to satisfy the users and customers objectives. When an error is made at an earlier stage of the development, and not discovered until later, then the cost of that error will become greater as time goes on. At IBM, and elsewhere, the measured cost of letting an error slip out of the design stages and into the code testing stages was a factor of twenty more cost at the later stage to deal with the problem thus created. Further, if that same error was allowed to slip past the testing stages, the cost of dealing with it when the customer reported it would become 67 times (1978, IBM Santa teresa labs, Horst Remus) to 82 times (1979 same source) more expensive than if it had been dealt with during the design stage (where 2/3 of the errors occur).

The chief defense against these errors is intensive quality control procedures on the early documentation. Such techniques as reviews and walkthroughs have been found too weak to do the job. Rigorous Fagan's Inspection has been the primary tool for this cleanup at IBM and elsewhere. But, rigorous analytical efforts to spot human error will fail if the language of specification is unclear.

THE NEED FOR A HIGHER LEVEL SPECIFICATION LANGUAGE

In parallel with the increase in analytical rigour for early software documentation (primarily the many documentation stages before coding, later including source coding) we have recognised that we need to improve our way of expressing requirements and design.

We acknowledge the need for a rigorous language to express program logic to a computer. We acknowledge the need for a rigorous definition of data and of telecommunication protocols. The machine will rarely let us off the hook. Human beings will always try to interpret something said or written, but they are sharply limited by the information given, and the shared culture or agreements between sender and receiver.

To worsen this problem, human being make a lot of errors when writing software specifications. I expect to find between five and twenty-five errors or problems of exact interpretation per page in normally well written documentation today. This number goes down to about 1 defect per page or less when a culture systematically learns and

competitively tries to improve its performance. Most software engineering cultures today are unaware of this problem and have no measuring method or improvement device in this area, such as statistical process control over systems documentation using Fagan's inspection method.

AN AMBIGUITY EXAMPLE

If I were to say to you " I want to know the fastest timing for the 100 meters", you would have interpretation problems. It this a running or swimming or rowing event or what, it it for children, seniors, airplanes or grasshoppers? The range of interpretation possibilities is too great to be useful. If i say i want you to build a state of the art software system with regard to bug maintainability - you would have exactly the same problem. The definition of what you need to engineer (programming has almost nothing to do with the answer here) might range from a performance level of fixing bugs of "within six months" to " within 6 microseconds". How is a poor software engineer suppose to know what is required? And, until they know what is required, how can they begin to consider the large number of design possibilities for solving this design problem?

This leads to a fundamental principle:

UNAMBIGUOUS DEFINITION SHOULD BE GIVEN WHEREVER POSSIBLE

This principle requires more training than almost any software

engineer is presently given. I find that they have virtually no training into the precise definition of the most important quality requirements of their systems. The result is that they are unable to design engineer towards these targets, and they are unable to test and validate that these targets have been met before delivery. The result of this is that before the users are satisfied, substantial delay occurs, costs mount up and the software engineering profession rights has a very poor international reputation.

A constructive principle can be stated to remedy this situation:

ALL CRITICAL RESULTS SHOULD BE STATED IN MEASURABLE AND TESTABLE WAYS

This principle contains many potential problems. How do we know if we have identified all critical results? How do we know if a result is critical to success or failure? How do we state such results as " portability", "integrity" or "usability" in measurable and testable ways? How can we measure these things economically and practically at early stages of design, before we even have a working system?

I cannot give you all the answers here. But I can assure you that practical answers to these questions have been found, documented, and are all well within the reach of determined common sense.

THE EXTENDED PROBLEM OF DEFINITION LANGUAGE FOR SOFTWARE SYSTEMS

This set of principles does not alone solve the problem of appropriate software engineering principles in the area of requirements and design languages. I have found that we have a terribly difficult time relating the many elements of a software project with one another. Many problems are caused by our inability to understand how things are related to each other. We recognise the problem at the level of the algorithm. We have cross-reference listing to help us, for example.

But, software requires and is dependent upon much more than source code! One Uk computer manufacturer had forty-six different types of documentation for which they had written standard work-process descriptions. I don't think that clearly related things like the user manuals were included either. We found that they had major problems stemming from such simple things as the user manuals did not properly describe the programs. The programmers said that the documentation was at fault. Unfortunately consumer protection law says that the customer has a right to what you describe in the manuals! The real problem was one of too little control over the correlation between various documents (like code and the user manuals) as they developed and as they changed after initial delivery. We call this "configuration management" but in practice we do not do enough about it to prevent silly discrepancies in large numbers. So another fundamental software engineering principle has to be:

ANY CHANGE MUST KNOW ALL IMPACTED COMPONENTS AND CHANGE THEM ACCORDINGLY

It would seem to be perfectly good common sense, but few take it seriously. They all pay dearly, but most are not aware why they have the problems they do. One of the practical consequences I find of this principle whenever i do software engineering is:

* all elementary requirements and design statements must have a unique tag.

* frequent direct cross reference to these tags must be made in any part of the software documentation which is the source of a refinement.

* computer-aided tools must support the finding of all related cross-references rapidly and cheaply. If not, people will push ahead and take too-great risks in ignorance of the consequences.

DESIGN ENGINEERING OF SOFTWARE

The managing director and technical director of a large UK computer manufacturer asked me in 1983 to tell them why they were failing to provide software products which the customers were happy with. I spent two weeks looking around the company, and gathering evidence for a top management presentation. I told them two things.

1. They did not define the qualities they needed for customers in measurable ways so that software development labs knew exactly what their task was.

2. No software engineers were trained in the translation of quantified quality and resource results into products which met those specifications. (Hardware engineers were trained to do this).

The management adopted my first recommendation (quantify objectives) excellently. But one year later I pleaded with the technical director: "Excellent specification is worthless without the ability to translate (engineer) these requirements into real products!"

He did not disagree, but pleaded management overload at the time.

Three years later, he took me to dinner at Runneymead and said:

"You were always right about our lack of software engineering design capability. I can see that our product line is suffering from lack of it today. The problem is urgent. We are going to change"

He then delegated the problem to people who did not understand the problem or the solution and the problem continues to this day. It is not an easy problem to solve since there is no software engineering education which addresses the problem. Software engineers do not learn formally how to design towards a "99.98% availability" target, like systems engineers do. They learn irrelevant dogmas and myths like " structured programming is good for everybody" and "higher level languages are better".

I sometimes think that the best way to train software design engineers would be to train them as conventional engineers and architects first. At least the principles of design would be understood!

The principle here is that:

COMPETITIVE DESIGNS ARE ENGINEERED, NOT CODED AND TESTED, INTO EXISTENCE.

Make no mistake about it, this principle is very difficult to communicate to todays software engineers. They do not even share the definitions of the words used to state it with the rest of the world. To them "design" is functional specification. "Engineering" is programming and hacking. "Competitive" still means tighter code.

The one institution which has given explicit recognition to this is Bell Labs (AT&T Technical Journal march/April 1986, special issue on quality) - but they understand far more than software engineering.

CONCLUSION

Had I more time, I should like to share more principles with you. You will find about 150 more principles in my book “Principles of Software Engineering Management”.

It is my hope that this short lecture has managed to increase your appreciation for the more eternal things in life.

I hope you will return to work or study and ask: “Am I learning or teaching things with long term validity - or might I be wasting my time, as Tom did early in his career?”