"How to find solutions to the problems addressed by the SEMAT effort"

Alistair Cockburn

I have long been on record that the 1968 definition of "software engineering" was flawed. "The end of software engineering and the start of economic-cooperative gaming", published in Computer Science and Information Systems in Feb, 2004, contains a detailed critique of the 1968 work and a suggested replacement model.

The paper explains the key ways in which the 1968 model fails:

> "Software engineering" was introduced as a model for the field of software development in 1968. This paper, reconsidering that model in the light of four decades of experience, finds it lacking in its ability to
> * explain project success and failures,
> * predict important issues in running projects, and
> * help practitioners formulate effective strategies on the fly.
> The failure of the software engineering model leaves our field needing an underlying model that
> * explains why successful projects succeed and failing projects fail,
> * intrinsically names topics known to be important to project success, and
> * leads the person on a live project to derive sensible advice as to how to proceed.

Those failures also explain why the term "software engineering" does not provide a sound pedagogical base for teaching newcomers to our field.

These flaws simply have to be fixed. We can't injure another generation of our constituents by failing on these points.

The paper goes on to propose a model that works:

> An alternative underlying model for software development is presented: Software development as a series of resource-limited, goal-directed cooperative games of invention and communication

Since 2004, I have added three elements to the model: craft professionalism, lean processes, and design-as-knowledge-acquisition.

Craft Professionalism teaches us about lifelong learning, deepening mastery over time, and about developing skills in a medium. For software development there are a number of significantly different craft specialities, each working in its own medium.

The **Cooperative Game** model teaches us about the human component – trust, morale, strategies, the importance of things like invention techniques, decision-making conventions, and the properties of communication.

Lean Processes provide useful mathematics to the design assignment, once we recognize *unvalidated decisions* as the unit of internal inventory. Having made this adjustment, large amounts of manufacturing theory drops in our laps.

Design as Knowledge Acquisition teaches us that we can pay to learn, and structures our efforts to optimize both knowledge gained and business value accumulated. This view helps get rid of the tension between no-design-up-front rabid agilism and all-design-up-front conservativsm.

These four elements are all described in more detail in papers and talks, as given in the references.

The above theory is fairly sound and fairly complete. It does a good job of handling the criticisms / goals set out at the top:

* It explains project success and failures quite well,
* It predicts important issues in running projects quite well,
* It helps practitioners formulate effective strategies on the fly,
* It provides a sound pedagogical basis for teaching.

My thanks go to Philippe Kruchten for pointing out that Professional Ethis is significantly missing from the above elements. It might fit under Craft Professionalism, but since I am not an expert in this area, I hesitate to move there, and look forward to longer discussion.

The point of the above section is to illustrate a way to constructively critique the 1968 definition of software engineering, and to indicate that the topic under consideration is not as empty as the SEMAT call-for-action indicates.

Having made that point, the larger point remains – how do we work to settle on an agreed-upon definition.

I have wishes and proposals in this regard:

1. Look at what engineers **'do'**, not what they **build**.
2. Catch up with the state of the art in what is conventionally called **engineering**.

In more detail, here they are:

1. Look at what engineers *'do'*, not what they *build*.

Too many people look at a factory running with statistical controls in place, and say "That factory is engineering." The factory is ***not*** the engineering. *"Engineering"* was the verb, the activity of designing the factory. When chemical and process engineers get together to design a factory, their *designing* does not operate under statistical control processes; they are not more likely to get their schedule, cost and quality right than their programmer counterparts – they are quite as likely to overrun their time and cost budgets, to argue with each other, to do design-by-committee, or it's alternative, most-senior-loudmouth-wins. In other words, engineering-in-action looks a look like software-development-in-action. This similarity is not to be overlooked.

In order to make sense of the term "software engineering", therefore, we need to examine the *activity* engaged in by other engineers. The book *SkunkWorks* illustrates this very well, as

does Donald Schön's *Reflective Practitioner*. There are people today studying the act of engineering, and we should talk with them.

This leads immediately to the second point:

2. Catch up with the state of the art in what is conventionally called engineering.

There are people today studying the act of engineering, and we should talk with them.
We need to engage those people, find out how they think about engineering-the-activity from the solid-engineering perspective.

I can name two such people to get us started: *Dr. Alexander Laufer* has been working in and studying civil engineering projects for decades, and has been working with NASA for years. His examples, which closely match my experiences with software development, are taken from the building of airports and hospitals. The other is *Stephen C-Y. LU*, David Packard Chair in Manufacturing Engineering at USC and chair of the working group of *Engineering as Collaborative Negotiation*.

These are similar people are also investigating models of engineering. It would be foolish for us as software practitioners to once again speculate on our own as to what "engineering" entails, without seeing what experts in other engineering fields feel it entails.

Summary

When we get 'there', our result should

* explain why successful projects succeed and failing projects fail,
* intrinsically name topics known to be important to project success,
* lead a person on a live project to derive sensible advice as to how to proceed,
* provide a sound pedagogical base for teaching newcomes to the field.

A draft model meeting those criteria exists: craft, cooperative gaming, lean process with decisions as internal inventory, and design as knowledge-acquisition. That model should be critiqued as any other model to find its holes. It is notably missing "professional ethics", so it is already known to be incomplete. It is a model looking for competing models.

As as group, we should do two specific things as the earliest part of going forward:

1. Look at what engineers **'do'**, not what they **build**.
2. Catch up with the state of the art in what is conventionally called **engineering**.

References

Alistair Cockburn: The end of software engineering and the start of economic-cooperative gaming at Keynote presentation at AGILE 2009, Foundations for Software Engineering, Software engineering in the 21st century presentation, From Agile Development to the New Software Engineering,

Stephen C-Y. Lu: Engineering as Collaborative Negotiation, Minutes of the ECN-WG Meetings 2003-2006