Towards AI-Native Software Engineering (SE 3.0): A Vision and a Challenge Roadmap Review - By Ahmed El Shatshat

Problem to be Solved

Brooks notes the fact that software development is slow, cumbersome, and suffers from many structural deficiencies that have origin in the essences of software development. Because of these issues, there is much hope for a solution that will revolutionize software development. However, this concept of a silver bullet is mythical, and impedes actual progress in software development practices. Furthermore, solutions touted as silver bullets are often revealed to provide some gains, but not at the order of magnitude one would expect from silver

New Idea

Brooks instead rejects the silver bullet as a concept, using those essences of software to argue that the difficulties of software development are far too tightly coupled with software development as a medium for there even to be the possibility of a silver bullet. The essences he identifies are complexity, conformity, changeability, and invisibility.

Software is much more complex than their size would suggest. As a result, it is that much more difficult for one to comprehensively wrap their head around every facet of a software system. Digital systems have a complexity an order of magnitude greater than most other systems, which further contributes to human error in this regard; there's no domain that is so unbound by the logic of structural space in the same way software is.

In addition to this, software systems must conform with human institutions and systems already in place. This further adds complexity as a designated software system is forced to follow the convention of any domain it is being applied to. In addition to this, software itself needs to conform to real world structures through use of data structures and encoded logic, which further increases difficulties.

Software is also much more malleable than products from other domains; it is much more feasible to write a new driver for a new printer than it is to enforce a uniform driver interface across all printers. This forces software into a situation where it must conform to a complex cultural matrix across industries and standards.

Finally, the invisibility of software makes it much more difficult to create comprehensive models in the same way one can for physical systems. It is very difficult to ensure that all stakeholders and developers on the same page in terms of design and functionality even with the use of models we have today, and such miscommunications can propagate issues that lead to the death of a project.

Solutions such as high-level languages and unified programming environments have provided clear benefits, yet are empirically seen to not be silver. To even have a chance at a silver bullet, the essence of what makes software difficult must be addressed, not simply the technical aspects of the process

Positives

Revolutionary Presentation of Software and its Issues: There's a reason why this paper is one of the most widely cited papers written in software. Even from the larval stages of software development as we know it today, Brooks makes bare the essence of software development, and zeroes in on how this essence is what makes software development difficult; any solution attacking contingent aspects of process or symptoms of that essence will not make the great gains imagined.

Timeless in its Message: Brooks expresses ideas that are still highly relevant today, even cleanly addressing concepts to the bleeding-edge of software today. Furthermore, many ideas he posits, despite having been demonstrated to be effective, are not being used as ubiquitously as they perhaps should.

A Time Capsule: It is interesting to see how the software field manifested in the mid-80s, in a time when high-level languages were just beginning to be iterated upon. With the gift of hindsight, one is able to read the ideas Brooks has in the larval stages of the development of many software tools and practices that today are commonplace, and perhaps reevaluate how they are being manifested today; indeed, this paper is almost like a requirements document for future software practices.

Negatives

Antiquated in Nature: hile the age of the paper has its benefits, it cannot be ignored that it is still nearly 40 years old. As such, many of the ideas aren't quite actionable, given they've already become integral parts of software development culture.

Untested Thoughts of a Single Author: hile not much of a negative, the paper is largely a collection of Brook's musings on software development as he sees it. It happens that I also agree with much of what he says, but there's not much in the way of proof that what he's saying is necessarily correct.

Future Work

In terms of future work, It would be interesting to see what has changed over 40 years (since 1986) in terms of the guidelines

outlined by Brooks; what was he correct about, what did he miss, and what has been ignored?

Further work on addressing the identified essentials of software development that are responsible for impeding development would also provide novel approaches to the problem.

- For example, are there ways in which software can be structured in a way that is more resilient to change
- Despite the unvisualizability of software, can we develop structural models that are more in-line with the epistemological models software developers employ?
- What aspects of software development are still victim to arbitrary conformity, and can we abrogate such conformity to streamline the development process without loss of quality?

Rating

I give the paper a 5/5, this paper is visionary in what it strives to do, and the impact of the paper speaks for itself in terms of how effectively it achieved its goal.

Discussion Points

1. Which the essences of software development outlined in Brooks paper do you believe are the primary cause for issues and accidents in the software development process today?
2. What recommendations by Brooks do you feel have been inadequetely implemented by the software industry at large, despite their demonstrated or apparant usefulness?
3. Are you a believer in a theoretical silver bullet that we simply haven't discovered, or do you believe that no silver bullet will be found, merely incrementally better solutions?

Summary of Discussion

In the class discussion, we further ruminated on the essences identified by Brooks, and reconciled the temporal contextual differences between his time and ours.

Prof. Godfrey noted that there has been much work done in the way of software visualization since Brooks time; however, their primary function is as a communication aid and are employed most frequently as informal diagrams to try and explain ideas to a fellow stakeholder. However, architectural diagrams such as class diagrams have been very useful in structurally situating software constructs with each other.

Furthermore, it was noted that it is difficult even with models to confirm that the epistemological model within each individual stakeholder's mind is firstly competently interpreted vis a vis the diagram, and secondly that the epistemological model's held in each mind are the same. It is impossible to know whether this is the case or not until some sort of working prototype has been created; this could be why Brooks notes the benefit of prototyping early.

AI as it exists today is also in a very different form that it was in Brooks' time. Prof. Godfrey notes the shift from building ground truth models which were designed to model the entire world to a more statistical approach was wrought by the impact the advent of the internet had on the field. Previously, one was unable to simply access wholesale datasets; with the internet, one can simply do a search and find more data than they may know what to do with. However, many of the ideas Brooks discusses in relation to the AI systems of his day are still relevant to the AI systems of today.

We also touched on the concepts from the other paper from this same week discussing SE 3.0, and ask if that theoretical view of software engineering in it's most optimistic iteration could be a silver bullet. I disagreed, given that even with the improvements noted in the paper by Hassan et al. it would not remove the issues that stem from human error in design and communication. Realistically, it would be similar to the gains achieved from moving from low-level Assembly to high-level programming languages.

Prof. Godfrey ended the discussion with an elucidating point: most revolutionary jumps in software development come from addressing the "dumb stuff" that we as software engineers didn't need to be doing in the first place; we just didn't know better at the time.