

Position Paper:
Anticipating Change in General-Purpose Software Systems

David Notkin
Computer Science & Engineering
University of Washington
Seattle WA 98195-2350 USA
notkin@cs.washington.edu

The “science of design: software-intensive systems” is an intriguing topic that has extraordinary breadth. To broaden it slightly further, it is important to add the word “engineering”, which The Call for Position Papers doesn’t use in a substantive way. At its heart, the goal of engineering is to design (and implement) artifacts that improve society, so there is an inherent connection between the “science of design” and “engineering.”

It is broadly documented why software-intensive systems differ from many other kinds of engineered systems. Here I will focus on a single reason: the necessity for rapid change to meet the evolving needs of users of these systems. And within that, I’ll argue a single point: anticipation of change in software-intensive systems is inherently more difficult than in most other kinds of engineered systems, and that this has challenging consequences to the design of these systems.

Belady and Lehman [1] and Parnas [2] are among those who have documented the nature and difficulties of change in software systems. The need for change arises primarily from evolving needs of users, with these needs arising for reasons ranging from individual preferences to technological improvements to social and economic shifts.

It is widely accepted that the best way to accommodate potential changes is to effectively anticipate them. Parnas’ work is foundational in this regard: information hiding [3] and layering based on the uses relation [4] provide principles for software system design given an anticipated set of changes.¹

We have developed few methods for effectively anticipating change; this, of course, compromises our abilities to design effectively.² I believe that one of the reasons for this is that many software-intensive systems are more general-purpose than most other engineered systems, and this makes anticipating change much more difficult – and more important. A classic example of a program written for a specific task that ended up being used much more broadly is ‘make’ [5]; although the core principle hasn’t really changed (“compare dates and execute a rule depending on the result”), the ways in which ‘make’ has been extended and stretched and pulled are extreme. Another example is spreadsheets, which are used in truly remarkable ways that their inventors never anticipated: they are used not just for budgets, but also as small databases, as checkbook

¹ The recent work on Extreme Programming argues otherwise. In essence it asserts that we are unable to effectively predict change, so instead one should at every point use the simplest possible design for a software system. Then, once changes are needed, one should restructure the design to meet the needs. This may indeed be a fruitful approach – and it certainly questions conventional wisdom – once it matures, but it’s still quite early and the outcome is unclear.

² One method that is widely accepted, at least in principle, is data abstraction. One key aspect of data abstraction is to allow the representation of data to be changed by a module without required changes to its clients. This is valuable (although there is little or no literature indicating how valuable), but regardless is not the only valuable kind of potential change.

registers, and much much more. And each usage puts different pressures on particular features, encourages other changes, etc. And there are untold additional examples in the software realm of unexpected use. Contrast this with airplanes, for example: very rarely are there radical changes in the intended function, the basic size, the underlying infrastructure (for instance, airports), and so forth. (The ways in which airplanes are implemented vary more significantly over time, but the intended and basic usage does not.)

What are the consequences of this observation with respect to design of software-intensive systems? I see at least two. First, we need to become more aggressive about understanding, empirically, particulars about how software systems actually change over time. This is not a question of code churn – that’s an interesting, but different question – but rather a question of how to understand and categorize and, thus, better predict, future needs that cause software to be changed. Second, we need to look at this in the context of the historical observation that much software is used in unexpectedly general-purpose ways. This deeply complicates this issue, and thus requires insights and ideas beyond those currently in play.

However, improvements in our understanding of future changes, even if incremental improvements, can lay the foundation for better design and for new kinds of design of software.

Biographical Information

I received a Ph.D. from Carnegie-Mellon University in 1984, and an Sc.B cum laude with honors from Brown University in 1977. I have been a faculty member in the Department of Computer Science & Engineering at the University of Washington since 1984, now serving as the Bradley Professor and Chair. My awards include a 1988 U.S. National Science Foundation Presidential Young Investigator award, election as an ACM Fellow in 1998, and the 2000 University of Washington Distinguished Graduate Mentor award. I served as the program chair for ACM SIGSOFT '93: First Symposium on the Foundations of Software Engineering, and as program co-chair for the 17th International Conference on Software Engineering. I have served as an associate editor for *ACM Transactions on Software Engineering and Methodology*, the *Journal of Programming Languages*, and the *IEEE Transactions on Software Engineering*. I have graduated a number of Ph.D. and M.S. students who are active in the field.

References

- [1] Belady, L. and Lehman, M.M.. A model of large program development. In *IBM Systems Journal* 15, 3 (1976), 225-252.
- [2] Parnas, D.L. Software aging. In *Proceedings of the 16th International Conference on Software Engineering* (May 1996), 279-287.
- [3] Parnas, D.L. On the criteria to be used in decomposing systems into modules. In *Communications of the ACM* 15, 2 (1972), 1053-8.
- [4] Parnas, D.L. Designing software for ease of extension and contraction. In *IEEE Transactions on Software Engineering* SE-5, 2 (1979).
- [5] Feldman, S. Make--A computer program for maintaining computer programs. *Software-Practice and Experience*, 9, 4 (1979), 255-265.