

Riding the silver bullet: chasing a moving target

Hafedh Mili, CS Dept, University of Québec in Montréal, Canada

I. Key points

Two things struck me when reading “NSB” (and “NSB-refired”): 1) amazement at how much the technological landscape has changed in the past twenty years and yet 2) admiration at how much of “NSB” still applies today. While the distinction between accidental complexity and essential complexity is a fundamental one, I would draw it differently from Brooks, who argued that [Brooks, 1987]:

the [essentially] hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation

Indeed, while these activities are unarguably harder than the coding itself, I put the boundary between essence and accidents further upstream: the essential complexity is in the requirements domain (problem domain), and not the solution domain (specification of the *software* that will address those requirements, regardless of the abstraction level). Indeed, requirements capture necessarily involves irreducible human processes (human communication, negotiation, consensus building) which cannot be automated or compressed. We may be able to improve productivity by an order of magnitude once, but to use Brooks’ logic, as soon as requirements capture represents more than 10% of the cost of developing and maintaining software, our clip of silver bullets comes empty.

Going back to the specification, design, and testing of the conceptual structures, we believe that substantial gains can be made—and have been made in the past 20 years—corresponding to two dimensions of reuse:

- 1) The nature of the specification language/vocabulary at our disposal to express those structures, and
- 2) The number of such structures that we have to specify/design/test, for a given functionality.

We talk briefly about these two, and then conclude with a brief commentary on some of the points of NSB.

II. Layers of virtual machines

The first generation of programs had to be written against physical machines, and programmers had to worry about managing the physical and software resources of the host machines. Operating systems provided developers with more sophisticated programmable machines that managed a number of resources for the developers, without having to worry about it. Middleware provides yet an additional layer of services, but this time for a “distributed machine”. **With higher level machines, we not only save of programming those services (less code to write), we also save on specifying, designing, and testing them.** With an operating system, I don’t need to specify, design, or program file access to ensure that no two threads or processes attempt to write on the same file. Similarly, with a J2EE container, I don’t need to worry about launching different threads to service multiple client requests: the container does that for me¹. Thus (in theory), I need only focus on the business logic of my components, and that is the only thing I need to specify, design, and test: the other services are provided by the *container*. **In theory (at least), virtual machines (of all levels) reduce the number of interactions that I need to manage between my components so that it grows linearly with the “functional size” of the application, as opposed to its square (n^2).** Clearly, a good number of the applications that are being built today would not be imaginable without these additional layers, regardless of hardware performance and capacity².

With “virtual machines”, we turn *extensive* functionality of the software (i.e. functionality that is implemented by various software components) with *intensive*³functionality, by implementing it as a *pervasive service* offered by the virtual machine⁴.

¹ This is quite different from high-level languages, and I agree with “NSB” that gains to be realized from high-level languages are inherently limited. A high level language provides a more concise way of *invoking* a service, but the service is exposed through the high-level language. By contrast, a *virtual machine* or *container* provides that service in a pervasive fashion, when it is needed, without having to explicitly invoke it.

² Generally speaking, *architecture* was conspicuously missing from “NSB”, both as a complexity issue or as an solution thereof

³ In thermodynamics, an extensive property is a property that is « additive », such as mass m or volume v , whereas an *intensive* variable is usually defined as a partial derivative of two extensive variables, e.g., *density*, which is $\partial m / \partial v$

⁴ In some ways, Kiczales had his contributions in the wrong order: MOP is an evolutionary step farther from AOP, and should have been offered, second, as a cleaner alternative for providing architectural services to the clumsiness of aspects. But that is another story.

In this regard, *domain specific languages* (DSL) will deliver on the hype only to the extent that they turn out to be more than just higher-level languages: in addition to providing higher level vocabulary/alphabet, their *sentences* should embody complex domain-specific behavior that would otherwise have to be specified, designed, and tested. In other words, they should embody domain-specific virtual machines! ☺

III. Component reuse

A somewhat orthogonal reuse dimension to virtual machines is component reuse: the users need A and B, and we have a virtual machine M. We identify a software component that implements A that can execute on M, and we are left to implement B for M. Here the gains are potentially enormous, and we fully agree with Brooks in identifying “Buy [acquire] versus build” as a major factor in breaking the complexity of software.

Note that, given a stable application domain, if our library of domain components starts to have a good functional coverage of the domain, we can start thinking in terms of a domain-specific machine (DSM) ☺

IV. Additional commentary

Fred Brooks can be accurately identified as the first extreme programmer. His recommendation for recognizing great designers has become a reality, at least in the OOPSLA community, in great part through the recognition that good design recipes are worth sharing with the community at large, making great designers virtual “rock stars”.

Finally, a comment about expert systems. In “NSB”, Brooks identified expert systems as one of the many false hopes for the silver bullet. While he—characteristically—identified the essence of the technology (“separation of the application complexity from the program itself”), he focused on the potential uses of the technology as potential aids to developers, and not as a way of implementing business rules. The latter is enhancing the maintainability of business applications in a major way, at least insofar as updates to business rules are concerned.