Opportunism is required to meet software demand

point

Opportunism Is Required to Meet Software Demand

Michiel van Genuchten, *Eindhoven University of Technology*

Companies can't develop from scratch all the software that products and applications will need over the next few years.

The amount of software in many products and applications is rapidly increasing. For example, the software in a mobile phone is expected to grow from 2 million lines of code in 2008 to 10 million in 2010. A car will contain 100 million LOC in 2010 (R.N. Charrette, "Why Software Fails," *IEEE Spectrum,* Sept. 2005). For reference, Windows Vista is approximately 50 million LOC. It's not possible or economically viable for original equipment manufacturers (OEMs) to develop and supply all this software. Industries can't afford to be too selective in collecting parts of their software stacks from various sources. They'll have to employ combinations of embedded software, commercial off-the-shelf software, and open source software to meet the demand. They'll also have to apply multiple development approaches, one of which will be opportunistic software development.

Some, like me, have applied opportunistic development without knowing the term. Over the past five years, I was part of the management of a business that earmarked software stacks, released them from their embedded architectures, and offered them as open system software on various platforms, such as mobile phones and personal computers (M.v. Genuchten, "The Impact of Software Growth on the Electronics Industry," *Computer*, Jan. 2007). Two examples:

- Video players originally developed for PC platforms now run on hundreds of millions of phones.
- Video-enhancement algorithms originally developed for embedded architectures for televisions are now deployed on PC x86 architectures.

We've been accused of junkyard development: get software for free and sell it on another platform in another industry without doing any real engineering ourselves. It's clear that starting with a working application does help. However, making the software perform within a software architecture and on a hardware platform for which it wasn't intended is real engineering work. Creating and sustaining a profitable business in terms of business development and marketing is still more work.

The main question is, to what extent can opportunistic software development help meet the increasing need for software? In my opinion, opportunistic software development isn't about forgetting all the good engineering practices put into place over the past decades. Calling it opportunistic won't make bugs disappear. Victor Basili and Dieter Rombach distinguished between construction and analysis of software ("The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. Software Eng.*, vol. 14, no. 6, 1988). If we want to reuse opportunistically, we must be able to quickly analyze massive amounts of software. We can benefit from better methods and tools for software analysis. Software metrics, reviews, static code analy-

Continued on page 82

counterpoint

The Case for Planned Reuse

Robert Baillargeon, *General Motors*

From automobiles to consumer electronics, manufacturers are challenged to meet customer preferences for individualized experiences within the constraints of mass production. In the past, automobiles were physical systems, characterized exclusively by the mechanical parts of assembly. Today, the industry has advanced to the point where we can best describe future vehicles as software-intensive systems. The tight coupling between the physical and cyber environments that typify these vehicles enables the enhancement of the user experience and of system performance.

The complexities of tightly coupled systems expose software's critical role in managing both variability and scale in their production. As software-intensive systems move from large to ultralarge scale, a mandate for innovation in both product and process goes with them. Developing individualized designs in the automotive domain isn't an effective practice. This is reflected in vehicle systems development methods that attempt to employ various forms of reuse. Central to the automotive challenge is maintaining predictability and sustainability while developing multiple products, so planned reuse plays a critical role in the future of these software-intensive systems.

Opportunistic reuse has been proposed as the path to this future, but I doubt whether its time as an effective method has arrived. A simple question remains: is it effective to reuse software outside the context of its original use? The answer is unclear. This approach has worked in some instances, but I question its ability to provide sustained or predictable success. Software developers often wish to obtain reuse from scavenged software because the investment costs are minimal. However, they must balance their desire with sound reasoning. We can look to the issues that David Garlan, Robert Allen, and John Ockerbloom raised in "Architectural Mismatch: Why Reuse Is So Hard" (*IEEE Software*, vol. 12, no. 6, 1995). They identified the source of reuse difficulty as a mismatch between assumptions and architecture. With only a slight extension, I would say that reuse is enabled by unified assumptions, consistent architecture concepts, and composable behaviors (that is, interface-type compatibility that avoids undesirable feature-interaction behaviors in component assembly). Successful planned-reuse methods actively engage in these attributes to encourage success, whereas opportunistic success reflects only that the attributes exist.

Planned reuse can meet predictability and sustainability software system requirements that opportunistic reuse can't yet guarantee.

Frameworks are a lightweight form of planned reuse. They address mismatch by defining a common development context for unifying assumptions and architecture. Although frameworks don't prescribe planned reuse, they do formalize concepts that are critical enablers. Observe the success of such general frameworks as Java 2 Enterprise Edition and Corba, which encourage composition and reuse. Furthermore, domain-specific frameworks, such as Autosar (automotive open

Continued on page 82

point

continued from page 80

sis, and automated regression testing will all contribute to make opportunistic software development a responsible engineering practice.

Education must change to achieve responsible opportunism. Typically, universities don't train students to analyze software and reuse it opportunistically. Often, an assignment starts with a few specification lines, after which the student is expected to write an elegant piece of code, preferably starting from scratch. Some teachers are now providing more real-life exercises, such as starting with a program of a hundred thousand lines of code that will require analysis and extension.

Opportunistic software development has implications beyond engineering. For example, look at software licenses. It's rewarding for an engineer to opportunistically discover a piece of software and put it to use. However, almost every piece of software comes with a license agreement. Here are three examples of what can go wrong if opportunistic software development leads to opportunistic licensing:

- Many software license agreements don't allow reusing or reengineering software. Doing it anyway can have legal implications.
- Reusing a piece of software might infringe on someone's patent.
- Exposing your company's intellectual property by opportunistically accepting a GPL can be a career breaker.

These examples don't mean that opportunistic development is impossible. They do mean that engineering opportunism must balance with business realities.

We need opportunism to meet the increasing software demand, but the implications of opportunistic software development go far beyond engineering. Allowing and implementing opportunism must therefore be a business decision. Business managers will make business decisions, engineers will develop opportunistically, and the customers will decide which company brought the best software to market.

Michiel van Genuchten is a part-time professor of software management at Eindhoven University of Technology. He's worked in industry since 1987, at companies including Philips Electronics and GroupSupport, a software company he founded. Contact him at genuchten@ieee.org.

counterpoint

continued from page 81

system architecture), establish industry standards for software—in this case, jointly developed by automobile manufacturers, suppliers, and tool developers. Such standards show continued value in framework approaches, which unify assumptions and architecture. Yet despite these frameworks' success, they remain limited by insufficient attention on behavioral composition. Consequently, after selecting components, developers still face a significant integration effort to resolve interface-type compatibility and behavioral interaction.

Software product lines (SPLs) represent a powerful method to execute planned reuse. The paths to SPLs are varied, but organizations that have instituted this practice see progress toward rapid, predictable, and sustained reuse. SPLs extend the success of frameworks in common architectures and assumptions via the addition of direct development of behavioral composability. This development of composable behaviors requires significant forethought and investment in the architectural patterns of feature interactions, but the results have been impressive. Instead of managing product variability and integration reactively, which occurs as a matter of course in opportunistic methods, SPLs require a planned practice with known results and qualities. As compared to other development methods, integration in SPLs resolves to an activity similar to feature selection because behavioral interactions have been addressed in the product-line creation. Through the direct engagement of reuse factors, SPLs have developed the predictable, sustainable reuse patterns that commercial success requires.

The minimal investment costs will always make opportunistic reuse alluring, but the transition from scattered points of success to an applied method remains uncharted. In frameworks and, more significantly, in SPLs, I see a foundation of practice and success—both firsthand and in the public literature. However, in a domain like vehicular transportation, where consistent and sustainable reuse are emphatic objectives, opportunistic reuse doesn't yet provide sufficient evidence of its capabilities. Until compositional supports are available to evaluate the potential in ad hoc construction, I believe opportunistic reuse cannot become a sustainable industry practice. On this basis, I conclude that effective opportunistic reuse remains more related to desire than to practice.

Robert Baillargeon is a staff researcher at General Motors Research, where he leads software engineering practices research. Contact him at rcbaillargeon@acm.org.

Michiel Responds

Opportunistic software development shouldn't result in opportunistic software. Opportunistic behavior of a car's airbag isn't acceptable.

I'm not against planned reuse, but planed reuse won't be enough if the software doubles every two years and goes beyond what we imagined in complexity. This situation calls for multiple methods. The open source movement can teach an important lesson. They've built very reliable systems without planning the reuse top-down. Many factors, one of which is extensive reviews of every line of code, have produced high-quality open source software.

We need multiple methods and a lot of business common sense. The amount of acceptable opportunism in software development will vary with intended use, application, and industry. In cars (Robert's world) and in high-volume consumer products, the costs of failure are high. However, we can break down the software stack and make different trade-offs for those pieces with lower costs of failure. For example, a nonfunctioning DVD in the back of the car might result in unhappy children, but it's less of a problem than a failing braking system.

Opportunistic software development will happen, anyway, because of the huge demand for software. One contribution of this special *IEEE Software* issue is to name it and open discussions of how to overcome its weaknesses.

Robert Responds

It's true that software engineering relies on some measure of opportunism to be successful. However, rather than a method, it's more appropriately identified as a difficult engineering task requiring significant effort. Transitioning opportunistic reuse to a predicable engineering method will require automated reasoning about the reuse attributes—that is, the dimensions of architecture, assumptions, and behaviors that I mentioned earlier. All practitioners would like composition validation to be a trivial analysis with a Boolean response of success or failure. However, software engineering's science and practice simply aren't capable of reasoning with certainty across these dimensions today. Consequently, opportunism leaves us with the challenging cycles of build-test-fix, which often invalidate much of our gains from reuse.

On the other hand, planned reuse methods, such as SPLs, have achieved the level of engineering practice that results in predictability, quality, and productivity. We must use the gains from these planned practices as a basis for developing the science to account for, and predict, the success of our opportunistic endeavors. Until we have a science to analyze opportunism's validity, planned reuse will dominate the practice of delivering predictable high-quality products.