Editor in Chief: Hakan Erdogmus ■ National Research Council Canada
hakan.erdogmus@computer.org

Architecture Meets Agility

Hakan Erdogmus

In an *Agile Times* article published several years ago ("Let's Scale Agile Up," *Agile Times*, Apr. 2003), I argued against attempts to scale up agile software development approaches beyond their intended home ground of self-contained projects undertaken by small coherent teams. I still don't agree with advocating an "agile" approach for anything and everything under the sun. If you're developing an interpreter for a domain-specific language whose specification is known, what's the point of insisting on using short time-boxed iterations? However, I've changed my opinions about the need to port the practices and processes included nowadays under the agile umbrella. I've come to appreciate the benefits of moving those practices and processes, selectively and with necessary adaptations and embellishments, to different application domains and project contexts. And the concept of architecture, although still downplayed by many agile advocates, has a major role to play in expanding the traditional scope of agile software development.

The architecture-agility reconciliation was unmistakable at a May 2009 workshop organized by *IEEE Software*, the University of Southern California, and the University of California, Irvine (see http://computingnow.computer.org/sac21). The stamp of endorsement on that partnership was posted simultaneously, but independently, by no fewer than two gurus whose professional lives have straddled the ordinarily separate worlds of the two realms. They are *Software*'s very own Philippe Kruchten and Grady Booch. Such convergence on the part of two pioneers may not be so surprising given that their paths crossed in the distant past in significant ways. But the common threads, along with the different angles, are nevertheless well worth exploring. They should inform the evolution of software practice, whose success depends on the ability to combine complementary ideas to fit the context in which they are applied.

Scope of Software Architecture

Kruchten defines software architecture as the ensemble of many encompassing aspects of a system, including primary decisions about the system's organization, the external attributes that influence those decisions, the structure of the elements that make up that organization, the interfaces of those elements, the system's behavior expressed as collaboration among the elements, and the composition of the elements into progressive subsystems. But software architecture doesn't end there. According to Kruchten, architecture also covers the style that guides the system's organization and the technologies and platforms underlying a system's implementation. However, despite this casting of a seemingly wide, catch-all net, architecture isn't all things important about a software system, either; it focuses on a relatively small, select set of systemwide choices with overarching and lasting implications.

Architecture Phobia

Kruchten explains that in some agile-loving circles, the aversion to architecture as a distinct concept and to architecting as a legitimate activity stems from a number of misconceptions and misguided thinking. Pervasive among the myths is associating software architecture with "big upfront design"

Mission Statement: To be the best source of reliable, useful, peer-reviewed information for leading software practitioners—the developers and managers who want to keep up with rapid technology change.

 0740-7459/09/$26.00 © 2009 IEEE

(BUFD) and massive amounts of design documentation that fall out of sync with the implementation soon after development starts. It's also common to equate the role of software architect with the ivory-tower persona of the all-knowing pointy-haired engineer-turned-manager who believes in throwing a BUFD over the wall for the development team, and expects everything to progress dandily afterwards. The perceived incompatibility between a fixed architecture and the ability to respond to change is yet another reason for architecture aversion. And finally, there's the uncompromising emphasis on results (also known as "working code") at the expense of all the other development artifacts that don't get deployed or aren't as tangible, and consequently don't get as much visibility. All these viewpoints constitute apparently justifiable pretexts for devaluing architecture. Booch and Kruchten, however, offer some common threads that counterbalance such anti-architecture pretexts.

Architecture's Omnipresence

The most prominent theme in Kruchten and Booch's thinking was the view of software architecture as an artifact that persists through a system's lifetime. Booch reminds us that every software-intensive system has an architecture at its soul regardless of the process used to develop the system. The architecture might not be explicit, static, or articulated in a dedicated document, but it's nevertheless always there. It morphs. It lasts as long as the system exists, but different stakeholders see different slices of it at different times.

Whether defined implicitly or explicitly, a system's architecture affects all primary crosscutting decisions, and many secondary ones, about a system through its conception, implementation, and evolution. As such, it may determine the system's fate. Kruchten and Booch therefore suggest that omnipresence should give software architecture a pivotal role in the development process, at least a role that's much more essential than is ordinarily granted in the mainstream agile software development community.

Architecture for System Governance

Architecture may not have directly observable value, but it still has plenty of intangible value as a risk-management vehicle. For effective risk management, some degree of anticipation is necessary and as important as adaptation, says Kruchten. For large projects with uncertain breakeven points, anticipation may be fundamental to continuity and ultimate success. Booch adds that architecture enables active and flexible budgeting with appropriate cost controls, coherent communication among stakeholders, accountability for technical decisions, and sound financial analysis at the right level.

However, to be most effective in governing system decisions, architecture must be visible. Booch advises creating, or reverse-engineering, an explicit representation if such a representation is not available, and keeping it current, or re-creating it as often as necessary when it gets stale. Think of architecture as systemwide knowledge. Capturing that knowledge allows us to revisit important systemwide decisions over time.

Architecting as a Continuous Activity

Simply articulating architecture will not grant us full control over a software system's evolution. The future of a system is influenced by a multiplicity of external and internal factors that constantly steer it in unexpected directions. An explicit architecture will inform that evolution, but more is needed to effectively manage system integrity. This is where deliberate, continuous architecting comes into play to maintain sufficient stability, responsiveness, and integrity. Both Booch and Kruchten advocate iterative and incremental evolution of architecture to counter erosive forces and keep the system in sync with changing conditions. Both strongly advise intertwining and integrating architecting with other development activities. The Rational Unified Process subscribes to this philosophy by promoting an architecture life cycle: architecting starts in the inception phase and rapidly ramps up in the elaboration phase. The bulk of architecting effort is concentrated in the elaboration phase, where the software architecture is revisited and refined iteratively. The effort then gradually reduces, but still spills into the construction and transition phases as the architecture continues to coevolve during the system's implementation, operation, and maintenance.

A New Column: The Pragmatic Architect

I'm proud to announce the inaugural installment of *The Pragmatic Architect,* a new regular column by Frank Buschmann. Frank comes to us from the burning trenches of the software development world as an extremely well-recognized trailblazer with loads of experience and an explosive arsenal in his supply bag. He's coauthor of the famous Pattern-Oriented Software Architecture series. Expect Frank to translate his experience and understanding into plenty of perfectly sized, easily digestible nuggets. Think of Frank's reflections and advice on the practice of software architecture as the ultimate complement to our long-running and popular column *On Architecture* by Grady Booch. Enjoy.

When it comes to how the iterative and incremental development approach could be practically applied to software architecting, Booch and Kruchten take different angles. In strong alignment with the agile mentality, Booch believes in continuous architectural refactoring as a means to dampen a system's natural and inevitable tendency to conceptually degrade (abstractions blurring, concerns scattering and getting entangled, responsibilities being redistributed) over time. The system's degradation is the result of local decisions that neglect global concerns. His emphasis is on simplicity and flexibility: start simply and flexibly and aspire to keep the architecture that way. He also cautions us against the temptation to use architecture to overcontrol the natural progression of a system, for example by indiscriminately barring all locally optimizing choices in favor of globally optimizing ones. Booch doesn't encourage risky big-bang refactoring, but rather managing architectural evolution through small, contained refactorings.

Kruchten is seriously sceptical about the effectiveness of starting with a simple architecture and allowing the "right" architecture to emerge organically through continuous refactoring. While he admits that in some situations this approach might work, he states that architectural refactoring often becomes prohibitively costly very quickly if certain considerations have been neglected early in the process. Kruchten's emphasis is thus on getting the architecture "sufficiently right" early on without necessarily resorting to BUFD. A stable enough architecture permits a modest amount of tweaking, but as in RUP, such tweaking often takes the form of the iterative refinement of earlier high-level decisions about vision, quality attributes, and platform choices. Lower-level decisions are about the decomposition, impact, and realization of higher-level decisions. On one hand, once high-level choices are made, they tend to be hard to reverse. On the other hand, sometimes the "last responsible moment" for a decision is much earlier than we think. Kruchten is mainly concerned about the risk of misjudging that moment. Early architecting effort need not be high, and it doesn't preclude small teams tackling aspects that are deferrable or tentative at the outset. Once systemwide decisions become stable, Kruchten encourages weaving architectural aspects with functional aspects and constantly testing the current architecture with implemented functionality. At that point, architecture and implementation start to coevolve synchronously.

Enough Architectural Representation

The image of page after page of detailed UML specifications is enough to send a cold chill down the spine of most developers. So, does explicit software architecture mean lots of documentation? Booch and Kruchten's answer is a categorical no. They favor a lightweight approach to representing and documenting software architectures. Booch further cautions that although representation is important, when software architecture is expressed in the wrong format or documented excessively, or the representations are used poorly, the result is the illusion of architecting without really doing it. Effective representation of software architecture is possible with simple artifacts. Yes, the artifacts could include specifications written in UML or an architecture description language if appropriate, but a few informal box-and-line diagrams, descriptions of a system metaphor, a succinct document capturing the relevant decisions, and combinations thereof might do the job as well or better. As an example of a lightweight representation suitable for most purposes, Booch gives Kruchten's 4+1 view model *(IEEE Software,* Nov./Dec. 1995, pp. 42–50).

Architectural information can even be embedded in a prototype or directly in the code itself (for example, captured by subsystem interfaces). So, documenting a software architecture might entail as little as referencing objects that the development team creates anyway, together with the rationale behind central underlying choices. However, Kruchten warns against making excuses to just live in the code: except in the simplest of scenarios, rising some degree above the code level is probably necessary.

In essence, software architecture can both be lightweight and serve its purpose so long as the information pertinent to future decisions governing the system's evolution are easily accessible when they're needed. Fine. The explicit manifestation of the architecture may be small, but how much architectural information deserves to be articulated and kept around? What is too much or too little? Well, at the risk of repeating a cliché, that depends on the context. Kruchten lists the existence of a de facto architecture and the project's size, criticality, age, rate of change, business model, governance model, team profile, and team distribution among the important context factors. As in other process choices, these factors determine the baggage that a development team must haul along.

You'll read more about the role of architecture in the agile software development context in an upcoming focus section scheduled for publication in early 2010. Guest editors Pekka Abrahamsson, Muhamed Ali Babar, and Philippe Kruchten are gearing up to assemble a lot of leading-edge material on this timely topic. On a related thread, a 2011 focus section guest-edited by Maja D'Hondt, Juan Fernández-Ramil, Yann-Gaël Guéhéneuc, Tom Mens, and Martin Robillard will zoom in on the link between software evolution and maintaining stakeholder satisfaction. You're also likely to catch glimpses of architecture and agility intermingling in The Pragmatic Architect, which debuts in this issue.