Software Architecture-Centric Methods and Agile Development

Robert L. Nord and James E. Tomayko, *Software Engineering Institute*

Including architecture-centric design and analysis methods in the Extreme Programming framework helps address quality attributes in an explicit, methodical, engineering-principled way.

The agile software development paradigm and plan-driven approaches each have their strengths and shortcomings. The former emphasizes rapid, flexible development, while the latter emphasizes project and process infrastructure. Many practitioners, particularly of agile methods, tend to view software architecture in light of the plan-driven side of the spectrum. They think that architecture-centric methods are too much work, equating them with high-ceremony processes emphasizing document production. But many elements make up a successful development approach, including process, product, technology, people, and tools. Software architecture is part of product quality and isn't tied to a particular process, technology, culture, or tool.

For the past 10 years, the Software Architecture Technology Initiative at Carnegie Mellon University's Software Engineering Institute has developed and promulgated a series of architecture-centric methods for architecture design and analysis. These methods are now at the point where the SEI is fitting them into popular software development processes. This article explores the relationship and synergies between architecture-centric design and analysis methods and the Extreme Programming framework.[1] We chose to focus on XP because it's one of the most mature and best-known agile practices. (For further information on agile methods, see the "Related Work on Agile Approaches" sidebar on page 48.)

Software architecture and XP

In the XP development model (see figure 1a), the customer records the required system functionality at the beginning of each development iteration in the form of user stories. During planning, the customer and developers determine what functionality to develop for the current iteration. User stories also contain descriptions of test situations. The customer and development team derive these tests from the specifications. The developers design the programming interface to match the tests' needs, and they write the code to match the tests and the interface. They refine the design to match the code's needs.

The design that emerges is a product of the relevant user stories that have been identified. But the architecture depends, for its shape and quality, on the development team's experience. Architecture-centric activities (see figure 1b) can inform and regularize the development process by emphasizing quality attributes and focusing early on architectural decisions.

0740-7459/06/$20.00 © 2006 IEEE

Related Work on Agile Approaches

Recent work shows how you can use architecture-centric activities to balance agile and plan-driven approaches, exploiting their strengths while compensating for their weaknesses. In situations where requirements change rapidly and an agile approach is warranted, architectural concepts can enhance the process of designing a system that will meet its requirements.

Students participating in studio projects in the masters of software engineering program at Carnegie Mellon University have been using the Architecture-Centric Development Method,[1] which uses concepts from the SEI's architecture-centric design and analysis methods, in this way. When developing complex, large-scale applications, many have reported that agile methods must be adapted to include more kinds of architectural information. We see evidence of this in the zero-feature release, the architectural spike, and agile practices that recognize the architect role.

Steve McConnell lists the evolutionary delivery life cycle as a best practice.[2] Evolutionary delivery recognizes an architecture activity and strikes a balance between the control a staged delivery life cycle offers and the flexibility of evolutionary prototyping. The degree to which it moves in one direction depends on the extent to which it accommodates customer change requests. An emphasis on the system features visible to the customer moves it in the direction of evolutionary prototyping, while an emphasis on the system's core architecture (which customer feedback is unlikely to change) moves it in the direction of staged delivery.

Alistair Cockburn uses three indices (communication load, criticality, and project priorities) to characterize projects and recognizes that different project characteristics need different methods.[3] He characterizes Extreme Programming (XP) as being suitable for projects with four to 14 people that have a criticality where defects won't cause loss of life. For larger projects, Cockburn recognizes the architect role.

Barry Boehm and Richard Turner use five dimensions affecting method selection (personnel, dynamism, culture, size, and criticality). They partition the dimension space into two home grounds—agile and plan-driven approaches—within these dimensions, recognizing that hybrid approaches exist at the boundary.[4] They demonstrate a solution for adapting XP to develop complex, large-scale applications by introducing elements of plan-driven methods. These elements include high-level architectural plans to provide essential big-picture information and using design patterns and architectural solutions rather than simple design to handle foreseeable change. Including architecture in this way might also delay refactoring. However, investing in architecture means that it will take longer to get to code, because the first iteration is what some people call a *zero-feature release*. In such a release, the architecture is in place, but the organization doesn't deliver any user-visible features to the customer.

Paul McMahon offers insights into extending agile approaches to large, distributed projects by employing an agile architecture.[5] With Don Procuniar and Dennis Rushing, he demonstrates this approach's success in a case study.[6]

References

1. A.J. Lattanze, *The Architecture Centric Development Method*, tech. report CMU-ISRI-05-103, Inst. for Software Research Int'l, Carnegie Mellon Univ., 2005.
2. S. McConnell, *Rapid Development: Taming Wild Software Schedules*, Microsoft Press, 1996.
3. A. Cockburn, *Agile Software Development*, Addison-Wesley, 2002.
4. B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, 2004.
5. P.E. McMahon, "Extending Agile Methods: A Distributed Project and Organizational Improvement Perspective," *CrossTalk: The J. Defense Software Eng.*, vol. 18, no. 5, May 2005, pp. 16–19.
6. D. Procuniar, P. McMahon, and D. Rushing, "AVCATT-A: A Case Study of a Successful Collaborative Development Project," *Proc. 2001 Interservice/Industry Training, Simulation and Education Conf.* (I/ITSEC), Nat'l Defense Industrial Assoc., 2001, pp. 920–930.

The Quality Attribute Workshop (QAW)[2] can help the development team understand the problem by eliciting quality attribute requirements in the form of scenarios. Basing the scenarios on business goals ensures that the developers address the right problems. The Attribute-Driven Design (ADD) method[3] defines a software architecture by basing the design process on the prioritized quality attribute scenarios that the software must fulfill. It helps identify the most important things to do to ensure that the design is on track to meeting key quality attribute concerns and delivering value to the customer. The Architecture Trade-off Analysis Method (ATAM)[4] and Cost-Benefit Analysis Method (CBAM)[3] provide detailed guidance on analyzing the design and getting early feedback on risks. The development team can use incremental design practices to develop a detailed design and implementation from the architecture. Architectural conformance and reconstruction techniques ensure consistency between the architecture and implementation.

These SEI methods can enhance XP practices. Using these methods results in an architecture-centric approach: architecture connects business goals to the implementation, quality attributes inform the design, and architecture-centric activities drive the software system life cycle. These methods make developing software easier and more consistent. Although designing the architecture is integral to the approach, the level of detail can be flexible.

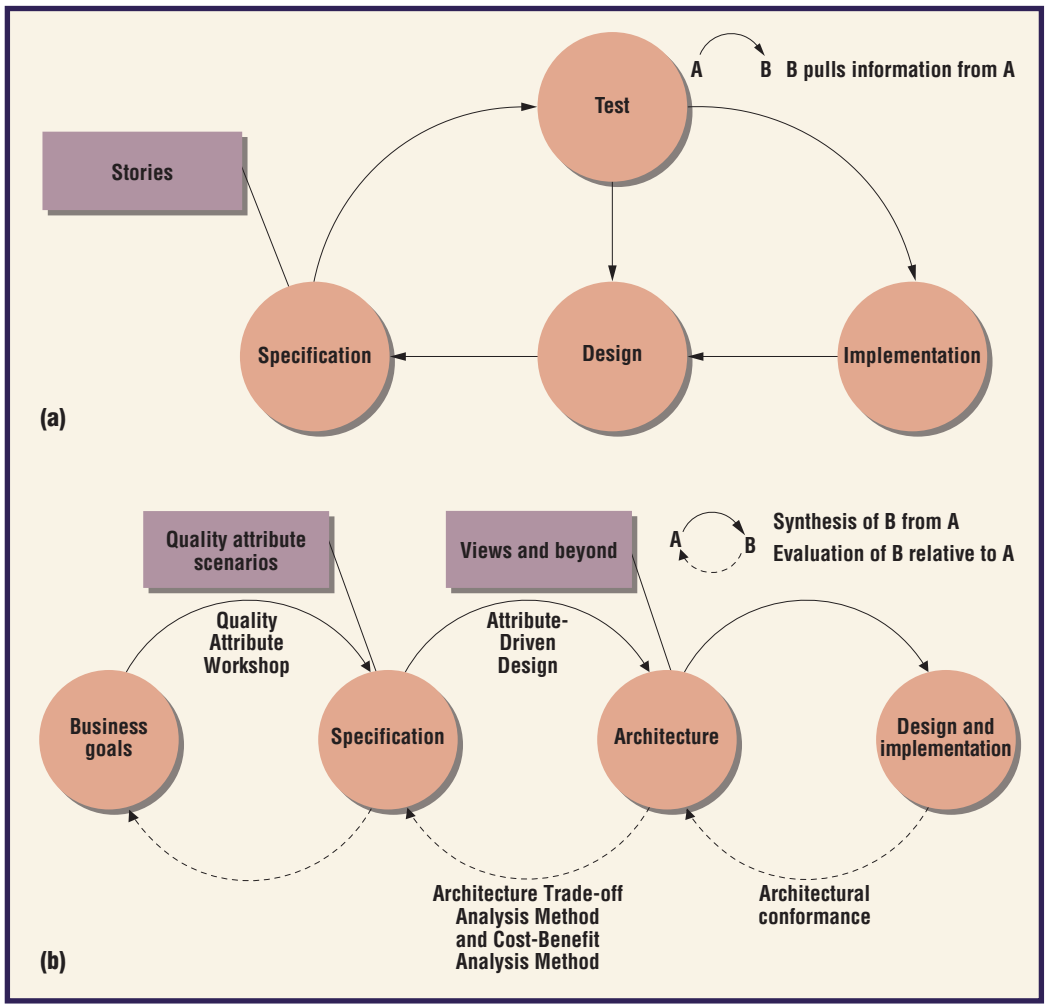

Figure 1. Development models: (a) Extreme Programming and (b) architecture-centric.

Identifying requirements: The Quality Attribute Workshop

In an XP project, system analysts elicit, capture, document, and analyze requirements. After examining user stories at each iteration's beginning, system analysts lead and coordinate requirements elicitation and modeling by outlining the system's functionality and delimiting the system. The result is a specification of details for the system's functionality.

The QAW, held early in the development process during XP story production, can help show quality attribute requirements in the form of scenarios. The QAW is a facilitated method that engages system stakeholders early in the life cycle to discover a software-intensive system's driving quality attribute requirements. The QAW is system-centric, focuses on stakeholders, and occurs before the software architecture's creation.

Quality attribute scenarios are much like user stories. Unless otherwise directed, stakeholders tend to focus on functionality, not on quality attributes. The QAW provides an explicit method for gathering quality attribute information in a six-part scenario format: stimulus, source of the stimulus, artifact, environment, response, and response measure.

A QAW would be appropriate for an XP project's first iteration, helping identify key quality attribute requirements. During a one-day workshop, participants brainstorm and refine a set of scenarios. Stakeholders attending the workshop include the on-site customer and others with an interest in the system—for example, end users, maintainers, project managers, development team members, testers, and integrators. Facilitators who aren't stakeholders act as QAW analysis team members. In subsequent iterations, the developers can collaborate with the on-site customer to elicit and refine additional scenarios as needed.

The ADD method focuses on something XP developers often ignore—the overall system structure that the quality attributes shape.

This approach gives developers guidelines for being more precise about quality attributes. It also enhances the planning and story-generation processes by emphasizing business goals, stakeholders, and quality attributes' role in shaping the architectural design. Business goals are elicited and refined during the QAW. The customer can use them to organize existing user stories, inspire additional ones, or prioritize requirements according to business needs. The scenarios can help determine what's within and what's outside the system's scope and can lead to the creation or refinement of a system context diagram or its equivalent. Scenario generation can also lead to the creation of use cases.

The XP practice of using an on-site customer is sometimes criticized. The idea is that keeping the customer on-site speeds communication and provides a more accurate accounting of changed requirements. Opponents figure that anyone the customer is willing to give up as a permanent loan to the development group is either junior level and lacking experience or too technical. Stakeholders' points of view aren't always easy to obtain; although on-site, the customer is sometimes removed from knowing end users' and other stakeholders' needs. Gathering stakeholders during the QAW complements the on-site customer that XP prescribes. To be successful, the workshop must gather a wide group of stakeholders from the organization. The QAW engages those stakeholders to discover and prioritize the quality attributes. The workshop setting facilitates open communication among the stakeholders and provides a forum where they can discuss conflicts or trade-offs among the requirements.

In addition to these more immediate benefits, quality attribute scenarios continue to provide benefits during later development phases. The development team should keep stakeholders' concerns and any other rationale information they capture in a format they can include in the architecture documentation—usually on a public whiteboard. Stories provide input for analysis throughout the system's life and can drive the system's development.

Scenarios can also help the customer prepare the acceptance test suite that will grow with the product. Typically, customers develop user stories for requirements and then work on acceptance test cases for the end of development. Many customers don't know how to build these test cases. The quality attribute scenarios can give them information, if not encourage them to build the test cases. This way of using the scenarios fits in with XP's "test first" or "build for the test" philosophy; test cases are available to test whether or not the code implements the requirements, early in the development process.

Early design: The ADD method

XP developers grow systems incrementally. When a system doesn't support new functionality, they refactor the design. The first iteration plays a crucial role in defining the system's overall structure, whether it's implicit (the architecture emerges after implementing the first round of stories) or explicit.

The ADD method focuses on something XP developers often ignore—the overall system structure that the quality attributes shape. This focus should occur in the first iteration and recur in later iterations, as substantial changes to the software architecture need to be explored. The development team can keep the architecture, quality attributes, and constraints on a whiteboard in the workroom where everyone can see it.

ADD differs from XP's core practices because it emphasizes addressing quality attribute requirements explicitly using architectural tactics. The quality attributes shape the architecture's structure, and functionality is allocated to that structure. The architecture helps localize the effects of design changes that are caused by changing the functional requirements; the architecture is influenced by the quality attribute requirements and isn't affected by changing functional requirements. The architecture represents the most important design choices. Because this is the first articulation of the architecture, it's necessarily coarse grained. More detailed XP design activities begin where ADD ends.

ADD follows a recursive decomposition process where, at each decomposition stage, the development team chooses *architectural tactics* and patterns to satisfy a set of quality attribute scenarios. An architectural tactic is a means of satisfying a quality-attribute-response measure (such as average latency or mean time to failure) by manipulating some aspect of a quality attribute model (such as performance-queuing models or reliability Markov models) through architectural design decisions. In this way, tactics provide a "generate and test" architectural design model.

The ADD method supports both a breadth-first and a depth-first decomposition approach to design. The order of decomposition will vary on the basis of the business context, domain knowledge, and changing technology. For example, ADD supports an XP approach by allowing an initial breadth-first decomposition for the first decomposition level, followed by depth-first decompositions to explore the risks associated with change through prototyping.

The ADD method creates and documents a software architecture using views. The project's nature determines the views—most commonly, a module decomposition view, a concurrency view, and/or a deployment view. The "Views and Beyond" approach to documenting software architectures describes these views.[5] It also describes a process for choosing appropriate views on the basis of stakeholders' needs. These views allow the developers greater flexibility and the opportunity to defer making more detailed decisions until the proper time. The module view fits XP's iterative development quite well.

Early analysis: ATAM and CBAM

By this time, still early in the process, XP practitioners will work on stories augmented by scenarios and the architectures they developed following the ADD method. They can do ATAM and CBAM right after this to track technical and business risks early in the process and to help prioritize stories for the next release. The ATAM and CBAM help practitioners understand how design decisions they make while creating a complex system architecture will interact.

The ATAM's purpose is to assess architectural decisions' consequences in light of quality attribute requirements and business goals. It helps stakeholders ask the right questions to discover potentially problematic architectural decisions. Developers can then make risks the focus of mitigation activities, such as further design and analysis and prototyping. They can identify and document trade-offs explicitly.

The CBAM helps make the architectural decision analysis performed during the ATAM part of a strategic roadmap for software design and evolution by associating priorities, costs, and benefits with each architectural decision.

The ATAM adds value by defining a step-by-step architecture-evaluation approach that produces risk themes and shows their impact on achieving business goals. The CBAM provides more details on architecture decisions' business consequences, letting the development team make informed choices among architectural options.

Example

To understand how architecture concepts and methods complement XP and address some of its shortcomings, consider an automated teller machine that's part of a bank automation system. The ATM's feature requirements are those the customer desires and are part of the XP user stories. However, other system qualities are important in addition to its functionality. For example, modifiability properties dictate that the system must be easily changeable so that the customer can exploit new platform capabilities, and it must be extensible to let the customer add new functions and business rules. In addition, performance requirements specify that ATM users must get a response from the system in less than 10 seconds.

Through the QAW process, we would refine vague requirements into several six-part scenarios. For example, the following modifiability scenarios would be typical of an ATM system:

- A developer wants to add a new auditing business rule at design time in 10 person-days without affecting other functionality.
- A system administrator wants to employ a new database in 18 person-months without affecting other functionality.
- A developer needs to add a Web-based client in 90 person-days without affecting the existing ATM client's functionality.

To achieve these requirements, XP emphasizes incremental design and relies on the architect's experience and knowledge to create the design. The ADD method provides more details on how we'll need to employ one or more architectural tactics. In the case of modifiability, relevant architectural tactics include Localize Changes and Use an Intermediary. The Localize Changes tactic suggests that we should localize the business rules, database, and client into components. The Use an Intermediary tactic suggests that we should separate these components to insulate them from each other's potential changes. A three-tier client-server model would emerge from applying the Localize Changes tactic; this architec-

The ATAM assesses architectural decisions' consequences in light of quality attribute requirements and business goals.

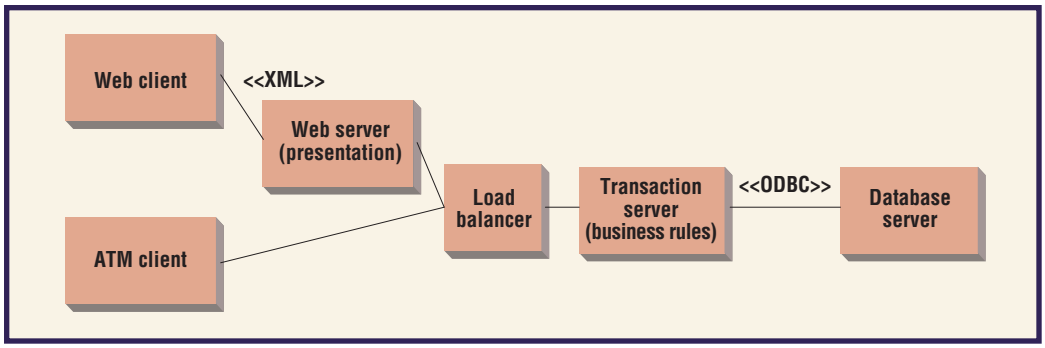

Figure 2. A deployment view of a candidate architecture using UML notation.

ture allocates the client, database, and business rules to their own tiers, localizing the effects of any changes to a single tier. The Use an Intermediary tactic suggests that an abstract interface (such as a data access layer that uses open database connectivity, or ODBC, between the business rules and the database) and a translation layer between the business rules and the client that understands the Extensible Markup Language (XML) should mediate communication between the tiers. Such intermediaries simplify adding new databases or clients. For example, they enable us to add a Web-based client and server to the architecture without affecting the ATM client.

To achieve the "10-second latency on a withdrawal" requirement, we employ a different set of architectural tactics. Because we can't control resource demand with an ATM (or, more precisely, because doing so would be bad for business), we must look toward managing or arbitrating resources to meet performance goals. By employing the Introduce Concurrency and Increase Available Resources tactics, we can choose to deploy additional database servers and business rule servers or to make any of them multithreaded so that they can execute multiple requests in parallel. Once we have multiple resources, we need some way of arbitrating among them. So, we introduce a new component—a load balancer—that employs a resource arbitration tactic such as Fixed-Priority Scheduling or First-In First-Out Scheduling. This component will ensure that the processing load is distributed among the system's resources according to a chosen scheduling policy.

This leads us to the design in figure 2. This example isn't to show a sophisticated architecture's development in its entirety (clearly, we must do more work to turn this into a complete design specification for development), but rather to emphasize how we arrived at the architecture. When using ADD, tactics and a structured set of steps provide design guidance for each tier's creation. In this way, we can create each architectural structure via an engineering process that codifies experience and best practices.

Architectural decisions are complex and interact. For example, we must analyze the degree to which changes in the database schema will affect the business rules, Web server, or client software. Each abstraction layer (XML and ODBC) will mask some class of changes and expose others. And each layer will impose a performance cost. Similarly, adding a load-balancing component will create additional computation and communication overhead but provide the ability to distribute the load among a larger resource pool.

We need a way to understand how design decisions made during the creation of a complex system architecture will interact. XP doesn't address evaluating the design explicitly; code is tested continuously and offers feedback to the design. The ATAM provides software architects with a framework for understanding the technical trade-offs and risks they face as they make architectural design decisions. Additionally, the CBAM helps architects consider the return on investment of any architectural decision and provides guidance on the economic trade-offs involved.

Summary

Table 1 illustrates how the software architecture-centric concepts and methods complement XP activities. Architecture-centric methods provide explicit and detailed guidance on eliciting the architectural requirements, designing the architecture, and analyzing the resulting design.

Including SEI methods addresses quality attributes in an explicit, methodical, engineering-principled way. Architecture-centric methods

Table 1

Extreme Programming and architecture-centric activities

Extreme Programming activities	Value added through software architecture-centric activities
Planning and stories	Business goals determine quality attributes (using the Quality Attribute Workshop): ■ User stories are supplemented by quality attribute scenarios that capture stakeholders' concerns regarding quality attribute requirements. ■ Scenarios help stakeholders communicate quality attribute requirements to developers so that they can influence design. Scenario prioritization and refinement give the customer and developers additional information to help them choose stories for each iteration. ■ During a one-day workshop, additional stakeholders supplement the on-site customer. Cost-effective methods facilitate interaction among a diverse group of stakeholders.
Designing	Quality attributes drive design (using the Attribute-Driven Design Method): ■ A step-by-step approach to defining a software architecture supplements incremental design; the level of detail is flexible. Architecture allows better planning so that developers can better estimate requirement changes' impact. They can plan for change that is foreseen and localize it in the design. ■ Developers do just enough architecting to ensure that the design will produce a system that will meet its quality attribute requirements and to mitigate risks. They defer all other architecture decisions until the appropriate time. ■ Architectural tactics aid refactoring, which is driven by quality attribute needs (such as making it faster or more secure).
Analysis and testing	Design analysis provides early feedback (using the Architecture Trade-off Analysis Method and Cost-Benefit Analysis Method): ■ The development team can use scenarios to evaluate the design and provide input for analysis during testing. ■ Architecture evaluation has a notion of triage to produce just enough information as needed and prioritizes on the basis of business importance and architectural difficulty to focus efforts. ■ Architecture evaluation provides early feedback for understanding the technical trade-offs, risks, and return on investment of architectural decisions. Risks are related back to technical decisions and business goals, giving developers justification for investing resources to mitigate them.

use common concepts—quality attributes, architectural tactics, and a views-based approach to documentation—that lead to more efficient and synergistic use. They also help facilitate communication.

We've found that architecture-centric methods are built on concepts and techniques that practitioners can tailor to an agile approach. Architecture-centric methods can add value to agile methods by emphasizing quality attributes and their role in shaping the architecture's design and by making it possible to adapt agile methods using a hybrid approach to handle larger, more complex systems.[6]

Acknowledgments

The Software Engineering Institute is a federally funded research and development center sponsored by the US Department of Defense. We thank our SEI colleagues Paul Clements, Rick Kazman, Mark Klein, and Rob Wojcik for their insights and Amanda Bejot for help with clarity.

References

1. K. Beck, *Extreme Programming Explained: Embrace Change*, 2nd ed., Addison-Wesley, 2005.
2. M.R. Barbacci et al., *Quality Attribute Workshops (QAWs)*, 3rd ed., tech. report CMU/SEI-2003-TR-016, Software Eng. Inst., Carnegie Mellon Univ., 2003; www.sei.cmu.edu/publications/documents/03.reports/03tr016.html.
3. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley, 2003.
4. P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2002.
5. P. Clements et al., *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2003.
6. R.L. Nord, J.E. Tomayko, and R. Wojcik, *Integrating Software-Architecture-Centric Methods into Extreme Programming (XP)*, tech. report CMU/SEI-2004-TN-036, Software Eng. Inst., Carnegie Mellon Univ., 2004; www.sei.cmu.edu/publications/documents/04.reports/04tn036.html.

About the Authors

Robert L. Nord is a senior member of the technical staff in the Product Line Systems Program at the Software Engineering Institute, where he works to develop and communicate effective methods and practices for software architecture. He received his PhD in computer science from Carnegie Mellon University. He is coauthor of *Applied Software Architecture* (Addison-Wesley, 1999) and *Documenting Software Architectures: Views and Beyond* (Addison-Wesley, 2002). He's a member of the ACM and International Federation for Information Processing Working Group 2.10 Software Architecture. Contact him at the Software Eng. Inst., Carnegie Mellon Univ., 4500 Fifth Ave., Pittsburgh, PA 15213-3890; rn@sei.cmu.edu; www.sei.cmu.edu/staff/rn.

James E. Tomayko, recently deceased, was a teaching professor in Carnegie Mellon University's School of Computer Science and director emeritus of its Master of Software Engineering program, as well as a visiting scientist at the Software Engineering Institute. He received his PhD from the University of Kansas. He coauthored *Human Aspects of Software Engineering* (Charles River Media, 2004). See in memoriam on page 115.