

Making Architecture Visible to Improve Flow Management in Lean Software Development

Robert L. Nord and Ipek Ozkaya, Carnegie Mellon University

Raghvinder S. Sangwan, Pennsylvania State University

// Release plans that give as much emphasis to architecturally significant tasks as to feature-based, high-priority functionality can achieve better outcomes by avoiding conditions that lead to wasted time and effort. //



AN ARCHITECTURE EMBODIES those design decisions that influence a system's quality attributes.¹ Changes to an architecture involve modifying a system's gross topology as well as

its communication and coordination mechanisms. Therefore, any defects in an architecture (such as lack of desirable security, latency, or scalability) can necessitate enormous rework. An

overly flexible architecture can lead to unnecessary work or overproduction. The time required to create such an architecture can also delay downstream activities.

Each software development paradigm treats architecture with a different focus. Phase-based software development methodologies, such as waterfall or Rational Unified Process allocate dedicated time and focus to different development activities, conducting them mostly in order (for example, requirements, analysis, architecture, development, and testing).² Dedicating large chunks of time to architecture up front might not only delay downstream activities but could also easily lead to overproduction waste, where the effort spent might not pay off.³

Iteration-based agile development methodologies, such as Scrum, work with predetermined two- to four-week sprints, each focused on feature-based, high-priority tasks.⁴ When there's no up-front effort on architecting, developers must periodically conduct refactoring for necessary changes to accommodate new features. As the system grows, conditions will necessitate rearchitecting, which might require significant rework beyond the expected limits of refactoring. Sprint 0 aims to overcome this problem.⁵ Allocating architectural tasks to a sprint backlog, to allow architecture and feature development to progress in concert, has also been an increasingly recognized practice.⁶

Although architecture in itself is important, prolonging the architectural work needed to support development can impede progress. A significant challenge is to determine what size increment of an architecture is best for a given iteration to manage the

LEAN DEVELOPMENT IN MANUFACTURING



Much of the lean thinking in software development can be traced back to Toyota’s manufacturing approach. However, there are some obvious differences between a manufacturing environment, with its focus on repetitive delivery of identical parts and products, and the software development environment, in which variation is necessary to create value for end users. Manufactured products have obvious phase gates, marked by the handoff of engineering drawings to production. Software development, however, is an iterative process where design is an ongoing activity throughout the development life cycle; the demarcation between design and production activities is much less distinct. Despite these differences, the focus of lean thinking on eliminating waste has shown great potential for improving software development project management.¹

Software architecture has a particularly strong influence on some of the waste categories from lean manufacturing (see Table A),² but its current practice offers little explicit guidance in lean software development.

The three types of waste discussed in Table A are closely related. Overproduction of architecture can create delay. Not enough production

of architecture can result in patched solutions leading to defects. Fixing these defects would lead to delay. Together, they all lead to wasted time and effort.

References

1. P. Middleton and D. Joyce, “Lean Software Management: BBC Worldwide Case Study,” *IEEE Trans. Eng. Management*, vol. 59, no. 1, 2012, pp. 20–32.
2. M. Poppendieck and T. Poppendieck, *Implementing Lean Software Development: From Concept to Cash*, Addison-Wesley, 2006.

TABLE A

Architecture-related waste.

Waste category	Relevance to software architecture
Overproduction	Overproduction waste occurs when an excess of goods is produced. In software development, this can result from implementing low-priority or nice-to-have requirements, or creating an architecture for such requirements or for unforeseen futures without a sound basis.
Delay	Delay waste refers to the period of inactivity downstream in the process that occurs because an upstream activity doesn’t deliver on time. This type of waste occurs when an organization aims to collect all architecturally significant requirements and plans to architect for them, or aims to build utmost flexibility into the system, leading to overanalysis and prolonged time spent designing the architecture.
Defect	Defect waste occurs when the delivered software doesn’t meet the expected quality. Defects related to architecture can result in paramount rework when rearchitecting is needed.

development flow. In this article, we show how the flow management concept from lean software development can provide a framework for balancing the allocation of critical architectural tasks to development effort.

Release Planning: MSLite Case Study

MSLite is a hardware-based field system that automatically monitors and controls a building’s internal functions, such as heating, ventilation, air conditioning, access, and safety.⁷

During its first year, the MSLite project used an object-oriented analysis and design methodology. It

captured requirements as use cases, which it grouped into functionally cohesive packages. It then distributed use case packages among different teams for development. Because the use cases weren’t fully elaborated, the team didn’t yet fully understand dependencies among the different use case packages. So, as the design emerged, so did new dependencies across packages, which created ad hoc communication streams among the different teams. Much to their chagrin, the teams found themselves in situations where they had to either wait for some yet-to-be-completed work by another team or perform duplicate work. Such

interdependencies created conflicting solutions for cross-cutting concerns, such as dealing with variability in building automation devices and handling latency issues associated with the transporting events that such devices generated. As the design evolved and the teams continued to discover interdependencies, managing the workflow became so challenging that the project was stopped midyear.

In the second year, the project was undertaken from scratch, using architecture-centric methods.⁸ Building on use cases from the previous year, the project team made explicit the systemic quality

attribute requirements that have the most significant influence on a system's architecture. The team used these requirements to design an architecture that reflected a better understanding of dependencies (both functional and systemic) among the MSLite system's different elements before distributing these elements for development. Thus, the architecture became a mechanism for coordinating tasks among the different teams, resulting in a significantly improved workflow.

Our Experience

Being familiar with the MSLite project, we were motivated to use lean software development's flow management concept (for more information, see the "Lean Development in Manufacturing" sidebar) as a framework for understanding how to balance the allocation of critical architectural tasks to a development effort. Figure 1a shows our adaptation of Don Reinertson's visualization of optimum batch size to demonstrate the effect of architecture-related overproduction, delay, and defect waste on the overall project, and how the size of architecture increments affects the costs associated with waste.⁹ To demonstrate the batch size, Figure 1b shows how we tested our adaptation by looking at the amount of *cumulative cost*—or the running sum of the cost incurred (both implementation and rework) up to a given release—spent on architecting for MSLite data.

As Figure 1a illustrates, architecting in many smaller increments reduces the cost of delay that results from waiting for an entire architecture to be completed. However, rework is more costly, because it might involve rearchitecting. Having fewer large increments reduces the cost of rework, but at the expense of adding to the cost of delay for the larger architecture increment to be completed.

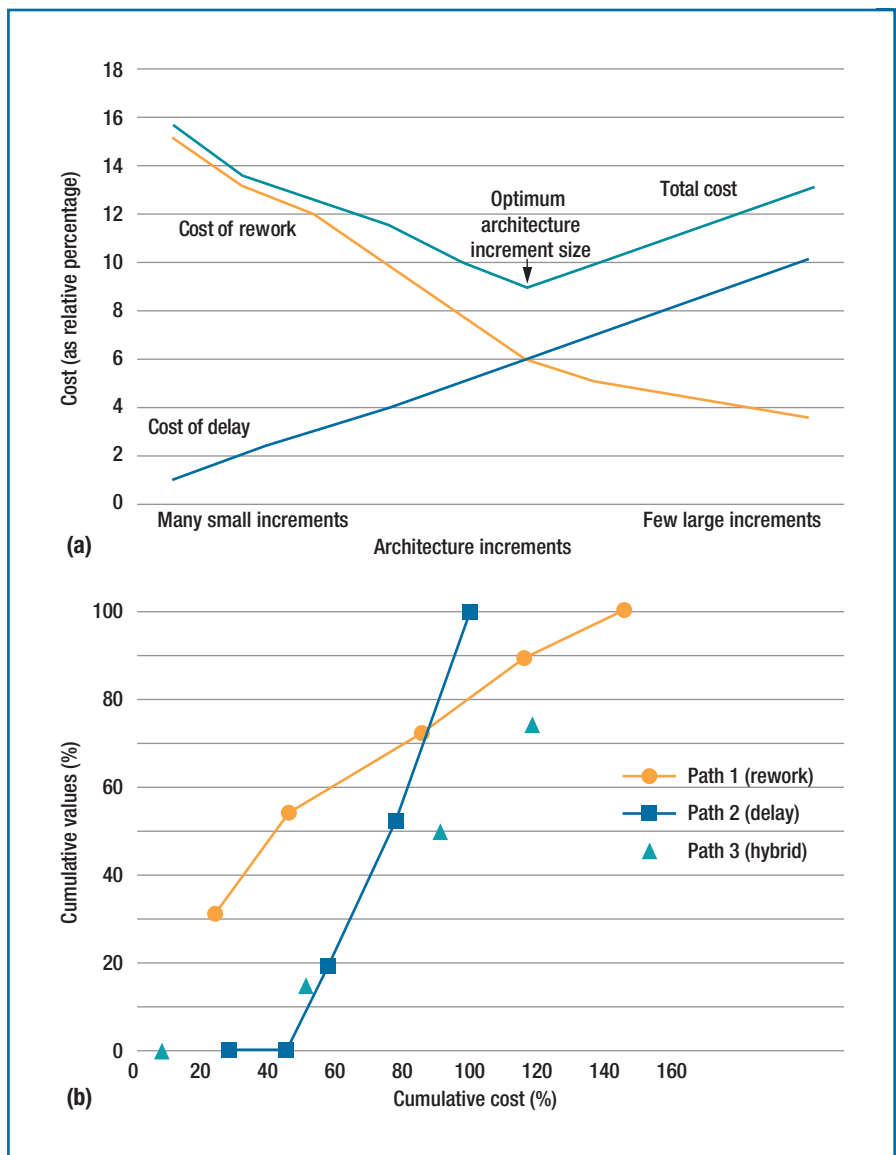


FIGURE 1. How waste affects product development flow. (a) Costs associated with architecture-related waste. (b) The MSLite system's influence on cost of rework. The cumulative cost of the system independent of rework is depicted as 100 percent cost on the x-axis. The cost exceeding 100 percent reflects the cost of rework.

Delay costs are incurred because of waiting or delay waste. Rework costs are incurred because of defect waste as well as overproduction waste.

During its second year, the MSLite development path fell somewhere in the middle of the total cost curve. In trying to understand and improve the process (architecture and increment

size), we modeled the delay and rework costs for this path in terms of the value of capabilities delivered over the total effort. To model the cost for the path, we looked at the MSLite project's development plan, the software architecture's module view, and its code (the DLLs).

Cumulative value is a running

TABLE 1

Flow management in delay- and rework-focused release-planning strategies.

	Path 1: Use delay avoidance to plan the release	Path 2: Use rework avoidance to plan the release
Throughput	Starts out fast and then slows, owing to high cost of rework	Starts out slow while putting infrastructure in place and then increases as new stories are rapidly released
Work in process (WIP)	Highest-priority stories and acceptance test cases; stories prioritized by business value without consideration of architecturally significant risk	Architectural elements that minimize dependencies, prioritized by dependencies without business value consideration
Cycle time or waste	High rework cost due to rearchitecting, because focus is on stories initially without considering architecturally significant acceptance test cases (representing quality requirements); leads to defects and high rework	Low cost of rework because quality and architecture support are initially considered
	Low cost of delay because architecturally significant requirements aren't initially considered	High cost of delay owing to up-front architecting; high-priority stories are delayed in favor of infrastructure
Advice on how to improve flow	Reduce cost of rework by expanding focus on stories, defects, and refactoring to include acceptance test cases, architectural technical debt, and rearchitecting	Reduce cost of delay by leveraging smaller architecture increments

sum of the value of delivered capabilities (all the user stories and architecturally significant acceptance test cases) up to a given release. Figure 1b depicts this on the y-axis. The pace with which value is delivered in each release is indicative of the cost of delay. For instance, there is a high cost of delay early on in the first release of the MSLite development path, path 3, where no value is delivered.

Flow Management

To see the influence of architecture on cost, we adapted path 3 using two strategies of development (modeled as paths 1 and 2 in Figure 1b) to realize the MSLite system's requirements. Path 1 demonstrates release-planning strategies focused on delay-cost avoidance, and path 2 demonstrates rework-cost avoidance. These strategies represent the two extremes for decisions in iteration planning.

Avoiding delay cost. Path 1 reduces the cost of delay at the expense of rework, focusing on small architecture increments. This path uses the priority of stories and acceptance test cases to guide the work (referred to

as *enhancement agility*¹⁰). Using an agile approach with many short iterations delivering small increments keeps the delay cost low because there's little waiting. However, architectural issues aren't visible to the end user and are likely to be initially missed or ignored, resulting in a high rework cost later on, when they can no longer be avoided. Therefore, the total cost is quite high.

Figure 1b shows this. Path 1 starts out with a high *throughput* (a ratio of work in process [WIP] and cycle time¹¹) during the first two releases, then delivery tapers off as subsequent releases take longer. The WIP limits are fixed at three stories or acceptance test cases. The drag on throughput is an indication of high rework costs to deal with the growing complexity of dependencies. This shows the effect of small architectural increments (which Figure 1a also demonstrates toward the extreme left of the total cost curve); small architectural increments create rework waste. Path 1 in Figure 1b is typical of efforts where waste elimination and short-term value delivery aren't balanced by consideration of the longer-term effects of the project's rework cost.

Avoiding rework cost. Path 2 reduces rework at the expense of delay cost. It focuses on a few large architecture increments, using rework cost as a guide. Using a phase-based approach and carefully considering architectural dependencies can minimize or eliminate rework cost but at the expense of delaying activities downstream while the architecture is put in place. The total cost is therefore high.

Figure 1b demonstrates this, where path 2 starts out with low throughput of end-user value early on, owing to the delay cost while the team focuses on putting the architecture in place. Figure 1a also shows the effect of large architectural increments toward the extreme right of the total cost curve.

Once the architecture is in place, throughput does improve as the team settles into a rhythm of releasing high-value capabilities at regular intervals. The WIP limits change, however, because later releases must deal with an overloaded allocation of stories with the assumption that the architecture is ready. Although this might appear to achieve high throughput—a higher number of stories yields higher value for the release—it's already on release

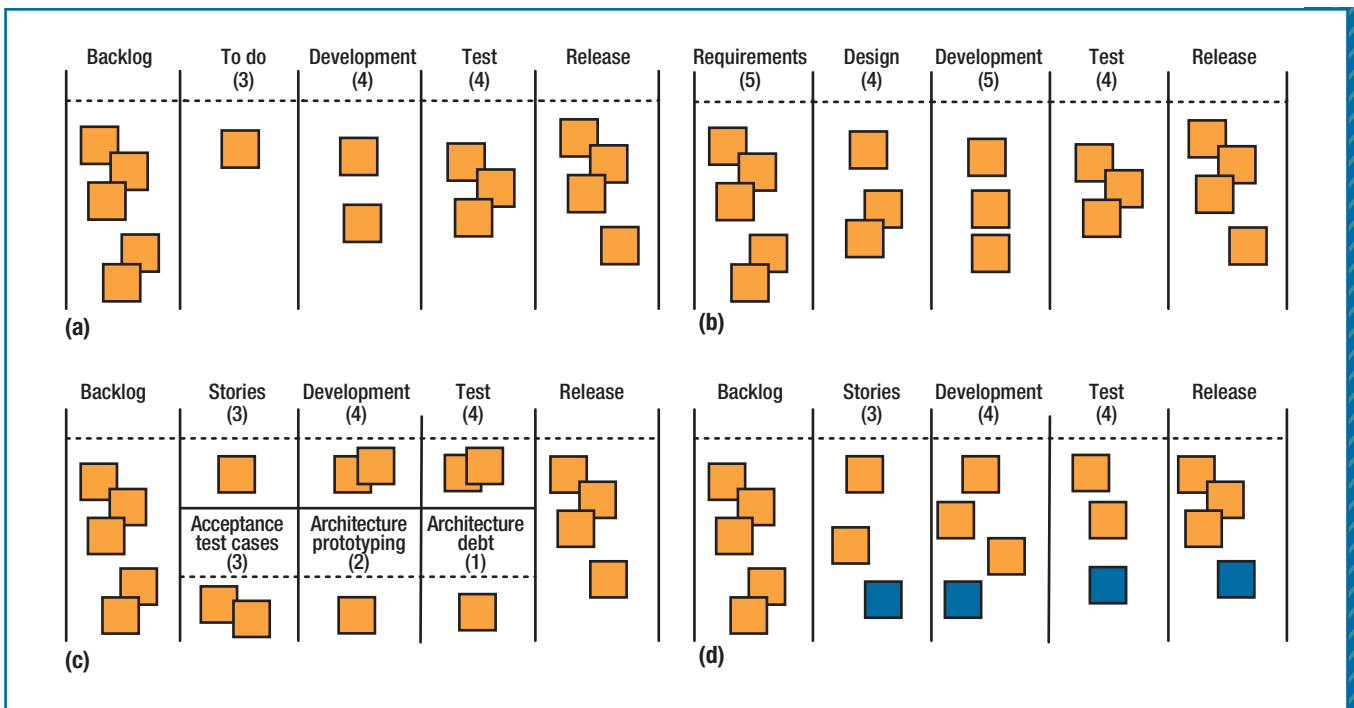


FIGURE 2. Different designs of our Kanban board for software development. (a) An agile-development-focused board. (b) A phase-based board. (c) Visualizing architecture with parallel horizontal swim lanes. (d) Visualizing architecture with color coding; architecture-related tasks are highlighted in blue.

5, and there’s an added complexity of having to deal with the maximum throughput in the final release.

A comparison. Although we constructed paths 1 and 2, they’re grounded in the actual system’s building blocks (requirements and architectural elements) and can be seen as variations of path 3. We wanted to see how the architecture-centric development path of MSLite would compare to these paths and provide insights for the best strategy for allocating architectural tasks to a development effort. Table 1 compares these two extreme iteration-based agile and phase-based approaches on the basis of how they treat lean principles.

Recommendations for Practitioners

There’s no one-size-fits-all approach for achieving maximum throughput value with minimal resource costs.

Focusing on flow management to analyze different approaches, however, can provide insight into how to improve management for both cycle time and WIP. One such insight involves improving the visibility of architecture-related aspects of a development project, as we demonstrated with the MSLite case study modeling different strategies for decision-making in iteration planning. Balancing the two opposing forces of delay-cost and rework-cost minimization will result in an improved outcome, as shown by the actual development path of the MSLite system.

The phase- or iteration-based approaches to software development typically don’t provide tools to visualize architecture’s role in managing developed features’ throughput. Therefore, we designed a Kanban board to elevate the criticality of the architecture and show how it supports or inhibits throughput.

Kanban is a scheduling system that helps just-in-time delivery by emphasizing pulling work items from process queues with work limits.¹²

Typical Kanban boards focus on feature, story, or task-level allocation of WIP limits—these limits are the numbers in the columns in Figure 2, mimicking the agile development context as shown in Figure 2a. Kanban is also used to support a phased-based approach where the WIP limits are organized according to an order of design, development, and test (see Figure 2b).

Our Kanban board design highlights architecture-related aspects to include infrastructure development, which is a critical aspect of improving throughput. There could be different versions of such a board involving parallel horizontal swim lanes or color coding, as demonstrated in Figures 2c and 2d, respectively.



ROBERT L. NORD is a senior member of the technical staff in the Research, Technology, and System Solutions Program at the Software Engineering Institute of Carnegie Mellon University. His research interests include effective methods and practices for software architecture. Nord has a PhD in computer science from Carnegie Mellon University. He is a distinguished member of ACM. Contact him at rn@sei.cmu.edu.




IPEK OZKAYA is a senior member of the technical staff in the Research, Technology, and System Solutions Program at the Software Engineering Institute of Carnegie Mellon University. Her research interests include empirical methods for improving software development efficiency and system evolution with a focus on software architecture practices, software economics, and requirements management. Ozkaya has a PhD in computational design from Carnegie Mellon University. She serves on the advisory board of *IEEE Software*. Contact her at ozkaya@sei.cmu.edu.



RAGHVINDER S. SANGWAN is an associate professor of software engineering at Pennsylvania State University. His research interests include analysis, design, and development of software systems, their architecture, and automatic and semiautomatic approaches to assessment of their design and code quality. Sangwan has a PhD in computer and information sciences from Temple University. He's a senior member of IEEE and ACM. Contact him at rsangwan@psu.edu.

Our experience in experimenting with different release plans demonstrates how lean software development has the potential to merge architecturally significant tasks with feature-based high-priority functionality development, unlike agile software development methodologies, where sprints can create artificial boundaries and story slicing becomes increasingly challenging. By visualizing architecture-related tasks that contribute to feature throughput, including tasks that span multiple sprints, teams can achieve an effective flow-based development environment.

The following actions should be kept central to the development effort: first, capturing architecturally significant requirements as acceptance test cases enables them to be more easily visible on a backlog or a Kanban board. In a lean environment, enforcing WIP limits on these acceptance test cases assists with improving flow by managing the overproduction waste associated with overarchitecting. Second, when determining story priorities, considering the dependency of the stories on architecturally significant tasks enables the pulling of related stories earlier in development and assists with flow. Finally, in addition to monitoring story development, monitoring the changing quality of the system via its architecture, possibly captured as technical debt or rework-related defects, enables a response to the cost associated with rework-related waste. 

A parallel swim lane for architecture-related tasks helped us achieve architecture development in concert with feature development to improve throughput, mainly by enforcing WIP limits on activities such as eliciting architecture-focused acceptance test cases, architecture prototyping, or rework to pay back architecture debt.

Enforcing technical debt WIP limits increases the visibility of rearchitecting for quality and also ensures that finding and fixing defects don't come at the expense of overall system quality. Similarly, both architectural prototyping and pulling through architecturally significant requirements (captured as acceptance test cases) should be in balance with story development.

Acknowledgments

This material is based on work funded and supported by the Department of Defense under contract number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

Carnegie Mellon University makes no warranties of any kind, either expressed or

IEEE
Software


NEXT ISSUE:

**Technical
Debt**

implied, as to any matter including, but not limited to, warranty of fitness for purpose or merchantability, exclusivity, or results obtained from use of the material. Carnegie Mellon University does not make any warranty of any kind with respect to freedom from patent, trademark, or copyright infringement. This material has been approved for public release and unlimited distribution except as restricted by copyright. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013 and 252.227-7013 Alternate I.

References

1. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., Addison-Wesley, 2012.
2. W.W. Royce, "Managing the Development of Large Software Systems," *Proc. IEEE WESCON Conf.*, IEEE, 1970, pp. 328–338.
3. J. Coplien and G. Bjørnvig, *Lean Architecture: For Agile Software Development*, Wiley, 2010.
4. K. Schwaber and M. Beedle, *Agile Software Development with Scrum*, Prentice Hall, 2002.
5. D. Leffingwell, *Scaling Software Agility: Best Practices for Large Enterprises*, Pearson Education, 2007.
6. J. Madison, "Agile Architecture Interactions," *IEEE Software*, vol. 27, no. 2, 2010, pp. 41–48.
7. R. Sangwan et al., *Global Software Development Handbook*, Auerbach Publishers, 2006.
8. R. Sangwan et al., "Integrating Software Architecture-Centric Methods into Object-Oriented Analysis and Design," *J. Systems and Software*, vol. 81, no. 5, 2008, pp. 727–746.
9. D. Reinertsen, *The Principles of Product Development Flow*, Celeritas Publishing, 2009.
10. N. Brown, R. Nord, and I. Ozkaya, "Enabling Agility through Architecture," *Crosstalk*, Nov./Dec. 2010, pp. 12–17.
11. J.D.C. Little and S.C. Graves, "Little's Law," *Building Intuition: Insights From Basic Operations Management Models and Principles*, D. Chhajed and T.J. Lowe, eds., Springer, 2008, pp. 81–100.
12. D. Anderson, *Kanban: Successful Evolutionary Change for Your Technology Business*, Blue Hole Press, 2010.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.

IEEE computer society

EXECUTIVE STAFF

Executive Director: Angela R. Burgess; **Associate Executive Director, Director, Governance:** Anne Marie Kelly; **Director, Finance & Accounting:** John Miller; **Director, Information Technology & Services:** Ray Kahn; **Director, Membership Development:** Violet S. Doan; **Director, Products & Services:** Evan Butterfield; **Director, Sales & Marketing:** Chris Jensen

COMPUTER SOCIETY OFFICES

Washington, D.C.: 2001 L St., Ste. 700, Washington, D.C. 20036-4928
Phone: +1 202 371 0101 • **Fax:** +1 202 728 9614
Email: hq.ofc@computer.org

Los Alamitos: 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314
Phone: +1 714 821 8380 • **Email:** help@computer.org

MEMBERSHIP & PUBLICATION ORDERS

Phone: +1 800 272 6657 • **Fax:** +1 714 821 4641 • **Email:** help@computer.org

Asia/Pacific: Watanabe Building, 1-4-2 Minami-Aoyama, Minato-ku, Tokyo 107-0062, Japan
Phone: +81 3 3408 3118 • **Fax:** +81 3 3408 3553
Email: tokyo.ofc@computer.org

IEEE OFFICERS

President: Gordon W. Day; **President-Elect:** Peter W. Staecker; **Past President:** Moshe Kam; **Secretary:** Celia L. Desmond; **Treasurer:** Harold L. Flescher; **President, Standards Association Board of Governors:** Steven M. Mills; **VP, Educational Activities:** Michael R. Lightner; **VP, Membership & Geographic Activities:** Howard E. Michel; **VP, Publication Services & Products:** David A. Hodges; **VP, Technical Activities:** Frederick C. Mintzer; **IEEE Division V Director:** James W. Moore, CSDP; **IEEE Division VIII Director:** Susan K. (Kathy) Land, CSDP; **IEEE Division VIII Director-Elect:** Roger U. Fujii; **President, IEEE-USA:** James M. Howard

PURPOSE:

The IEEE Computer Society is the world's largest association of computing professionals and is the leading provider of technical information in the field.

MEMBERSHIP:

Members receive the monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others interested in the computer field.

COMPUTER SOCIETY WEBSITE: www.computer.org

Next Board Meeting: 5–6 Nov., New Brunswick, NJ, USA

EXECUTIVE COMMITTEE

President: John W. Walz*
President-Elect: David Alan Grier;* **Past President:** Sorel Reisman;* **VP, Standards Activities:** Charlene (Chuck) Walrad;† **Secretary:** Andre Ivanov (2nd VP);* **VP, Educational Activities:** Elizabeth L. Burd;* **VP, Member & Geographic Activities:** Sattupathuv Sankaran;† **VP, Publications:** Tom M. Conte (1st VP);* **VP, Professional Activities:** Paul K. Joannou;* **VP, Technical & Conference Activities:** Paul R. Croll;† **Treasurer:** James W. Moore, CSDP;* **2011–2012 IEEE Division VIII Director:** Susan K. (Kathy) Land, CSDP;† **2012–2013 IEEE Division V Director:** James W. Moore, CSDP;† **2012 IEEE Division Director VIII Director-Elect:** Roger U. Fujii†

*voting member of the Board of Governors †nonvoting member of the Board of Governors

BOARD OF GOVERNORS

Term Expiring 2012: Elizabeth L. Burd, Thomas M. Conte, Frank E. Ferrante, Jean-Luc Gaudiot, Paul K. Joannou, Luis Kun, James W. Moore, William (Bill) Pitts
Term Expiring 2013: Pierre Bourque, Dennis J. Frailey, Atsuhiko Goto, André Ivanov, Dejan S. Milojicic, Paolo Montuschi, Jane Chu Prey, Charlene (Chuck) Walrad

revised 22 May 2012