A Principled Way to Use Frameworks in Architecture Design

Humberto Cervantes, Autonomous Metropolitan University, Mexico City

Perla Velasco-Elizondo, Autonomous University of Zacatecas

Rick Kazman, University of Hawaii

// A holistic approach for architecture design uses both top-down concepts and implementation artifacts, which are bottom-up concepts. In doing so, it reduces the mismatch between the abstract concepts generally found in existing design methods and the technical ones architects use every day. //

ARCHITECTURE DESIGN APPLIES design decisions to satisfy a set of architectural *drivers*—the most important functional, quality attribute, and constraint requirements that shape a system.[1] Architecture design is notoriously difficult to learn and even harder to master; good architects typically have gray hair.

If a process is important and difficult, a common approach in many fields, including software engineering, is to try to systematize it to ensure predictability, repeatability, and high-quality outcomes. To this end, a number of architecture design methods have appeared during the last decade. For example, *attribute-driven design* (ADD) provides detailed steps for architects, including entry and exit conditions, to perform design.[2] Such methods provide structure to complex, often daunting problems.

However, such methods are difficult to apply because they deal in abstractions. For example, ADD primitives are tactics and patterns, whereas architects daily deal with additional primitives including commercial frameworks such as JavaServer Faces, Spring, Hibernate, and Axis.

Here, we present a more realistic, holistic approach to architecture design. It starts with architecturally significant requirements—drivers and constraints. It then systematically links them to design decisions and systematically links those decisions to the implementation options available through commercial frameworks.

We stress the importance of considering technologies—specifically frameworks—as first-class design concepts. We illustrate this through a real-world case study that highlights how frameworks are selected within ADD iterations and how architectural drivers are connected to the selection of frameworks. Initial qualitative results have been obtained from the successful application of this approach in several projects in a large software development company in Mexico City.

Design Decisions

Architectural design is performed by applying design decisions to satisfy a set of architecturally significant requirements, typically called architectural drivers. Architecture design decisions fall under several categories.

Allocation of Responsibilities

The allocation of responsibilities involves identifying the important responsibilities, including basic system functions, architectural infrastructure, and satisfaction of quality attributes and determining how these

0740-7459/13/$31.00 © 2013 IEEE

responsibilities are allocated to non-runtime and runtime elements.

Coordination Model

The coordination model involves identifying the elements of the system that must coordinate (or are prohibited from coordinating), determining the properties of the coordination, such as timeliness, currency, completeness, correctness, and consistency, and choosing the communication mechanisms that realize those properties.

Data Model

The data model involves choosing the major data abstractions, their operations, and their properties, organizing the data, and compiling any metadata needed for consistent interpretation.

Management of Resources

The management of resources involves identifying the resources that must be managed and determining the limits of each, determining which system elements manage each resource, how resources are shared, and the strategies employed when there's contention for or saturation of resources.

Mapping among Architectural Elements

The mapping among architectural elements involves the mapping of modules and runtime elements to each other—that is, the runtime elements that are created from each module; the modules that contain the code for each runtime element; the assignment of runtime elements to processors; the assignment of items in the data model to data stores; and the mapping of modules and runtime elements to units of delivery.

Binding Time Decisions

The binding time decisions involve establishing the point of time in the life cycle and the mechanism for achieving a variation in an architectural decision. Binding time decisions introduce allowable ranges of variation; each of the decisions in the other six categories have an associated binding time decision. This variation can be bound at different times in the software life cycle by different entities—from design time by a developer to runtime by an end user.

Technology Choice

Technology choice involves

- deciding which technologies are available to realize the decisions made in the other categories;
- determining whether the available tools to support this technology choice (IDEs, simulators, testing tools, and so on) are adequate for development to proceed;
- determining the extent of internal familiarity and external support available for the technology (courses, tutorials, examples, and availability of contractors);
- determining the side effects of choosing a technology, such as a required coordination model or constrained resources; and
- determining whether a new technology is compatible with the existing stack.

This final category is critical to system success. It is ubiquitous—every architect of every system must make this choice or deal with its consequences if it's given as a constraint—and fraught with uncertainty. Ideally, an architecture design method should help reduce this uncertainty. However, most methods don't provide guidance for technology choices.[3]

To illustrate this, consider ADD, which is a well-established method for designing an architecture.[2] ADD comprises the following steps:

1. Confirm that the requirements information is sufficient.
2. Choose a system element to decompose.
3. Identify candidate architectural drivers.
4. Choose a design concept (patterns and tactics) that satisfies the drivers.
5. Instantiate architectural elements and allocate responsibilities.
6. Define interfaces for the instantiated elements.
7. Verify and refine requirements and make them constraints for instantiated elements.
8. Repeat these steps for the next element.

Step 4 is critical. Architecture patterns and tactics guide the architect from these key requirements to a design concept. Consider, for example, the tactics hierarchy for the quality attribute of performance (see Figure 1). An architect designing a high-performance system is guided from coarse-grained requirements, such as controlling the demand for or managing resources, to specific tactics, such as managing the sampling rate, introducing concurrency, and bounding queue sizes. On this basis, an architect might then choose a pattern that instantiates these tactics, such as

Most methods don't provide guidance for technology choices.

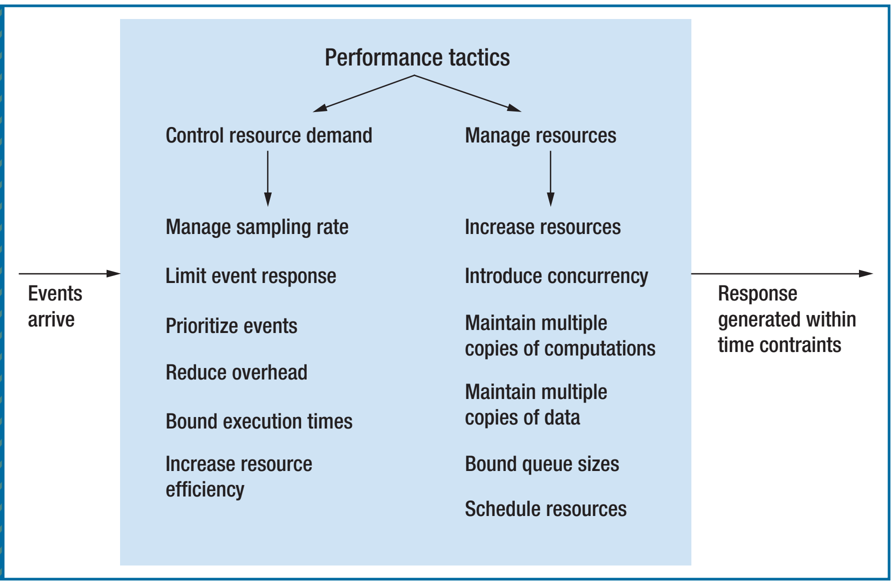

FIGURE 1. The tactics hierarchy for the quality attribute of performance. Each tactic is an architectural design primitive aimed at either controlling the demand for a resource or for managing the resource. (Reprinted by permission of Pearson Education.[1])

TABLE 1 The architectural drivers for a system for buying bus tickets.

Driver	Description
Primary use cases	Buy ticket Search timetables
Quality attribute scenarios (ordered by priority)	Performance. A user performs a timetable search during peak load (5,000 users are accessing the system); the system processes the query in under 10 seconds. Security. A user submits personal data in a form; 100 percent of this data is transferred and stored using encryption. Usability. A user fills out a form incorrectly. After submitting, the system highlights all erroneous fields and provides instructions for correcting them. Modifiability. A developer adds a user interface language to the system during maintenance. The language is added successfully and no recompiling is necessary.
Constraints	The initial release's time to market is six months. The system must be compatible with Internet Explorer 8+, Firefox 3+, Chrome 6+, and Safari 5+. The system must support access from iOS and Android in a later release. The system must work with a legacy relational database for which the customer had acquired a license. The development team is small and has experience in Java, JavaServer Faces, Spring, and Hibernate.

Concurrent Pipelines, Leader/Followers, or Thread Pool.[4]

But where to from there? At some point, architects must choose a set of technologies and, more specifically, frameworks to instantiate system aspects (such as user interface development and persistence).

Frameworks as First-Class Design Concepts

Frameworks incorporate many patterns and tactics. (For more on frameworks, see the sidebar.) Mapping these patterns and tactics onto the ones employed in the architecture design—the output of ADD—might not be straightforward. For example, the .NET framework provides services for clustering, load balancing, implementing a RAID (redundant array of independent disks), rolling upgrades, and so forth. What's the relationship between these services and the patterns and tactics that address performance?

Our answer is that patterns and tactics typically instantiate the services that frameworks provide. For this reason, frameworks need to be considered first-class concepts during design. Architects must recognize this. In doing so, they simultaneously design both top-down, using abstract patterns and tactics, and bottom-up, by selecting concrete realizations of those design concepts within frameworks.

A Case Study

To illustrate the value of using frameworks as first-class concepts in design methods, we present a case study of the greenfield development of a system to buy bus tickets. This is a typical enterprise application in which many users interact with the system through Web browsers or mobile apps. They perform processes, such as checking bus schedules, that act according to information from a database.

Before starting architecture design, the development team elicited and analyzed these architectural drivers:

- *Primary use cases* constitute a subset of the system's use cases that describe the critical functionality needed for achieving the system's most important business goals.
- *Quality attribute scenarios*

FRAMEWORKS

A framework is a reusable software element that provides generic functionality, addressing recurring concerns across a range of applications. Frameworks increase productivity by letting programmers focus on business logic and end-user value rather than underlying technologies.

Frameworks abstract some combination of application, language, hardware, networking, storage, or operating system characteristics. One aim of abstraction is to reduce the cognitive burden on the programmer, who needs to learn only the framework rather than all the underlying details. The abstractions should change more slowly than the details, easing portability and evolution. Moreover, these abstractions, being shared among many systems, will more likely be heavily tested, reducing or entirely eliminating classes of bugs.

You can use frameworks by either incorporating or extending the functionality they provide. For example, a programmer might use existing widgets from a GUI framework to build a user interface. He or she might, however, not find exactly the right widgets with the right functionality. In that case, the programmer might choose to extend an existing widget to augment its default functionality through mechanisms such as inheritance or configuration options such as XML files or annotations. This extension wouldn't actually modify the framework (which typically isn't allowed).

Software frameworks are related to patterns and tactics in that frameworks instantiate these concepts. The patterns and tactics that a framework instantiates allow particular quality attributes to be satisfied.

Consider Hibernate, a popular object-relational mapping framework. Hibernate instantiates several patterns, including Identity Field, Metadata Mapping, Lazy Load, and Unit of Work[1]—as well as tactics for promoting modifiability and performance. Object-relational mapping hides the fact that an object model persists in a relational database. Mappers achieve this by isolating the code that translates between these two models, which is an instance of the modifiability tactic of increasing semantic coherence. Furthermore, the mapper is typically parameterized through configuration files, which is an instance of the modifiability tactic of deferring binding. Hibernate also implements tactics that promote performance; these include resource management tactics such as fetching strategies and using caches.[2]

Another example of design concept instantiation is Spring, an extensive framework for developing enterprise applications. At its core lies the *container*, a mechanism that lets developers create and connect components—*beans*—using the Dependency Injection pattern.[3] Spring instantiates such patterns as Abstract Factory, Prototype, and Proxy.[4]

Spring also instantiates tactics: using an intermediary and deferring binding promote modifiability, introducing concurrency promotes performance, and exception handling and transactions promote availability.

Other examples of frameworks in widespread use include JEE (Java Enterprise Edition), .NET, MCF (Meta Content Framework), Oracle ADF (Application Development Framework), Cocoa, Ruby on Rails, iBatis, and JUnit.

References

1. M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002.
2. G. King et al., "Improving Performance," *Hibernate—Relational Persistence for Idiomatic Java*, Red Hat Middleware, 2004; http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/performance.html.
3. M. Fowler, "Inversion of Control Containers and the Dependency Injection Pattern," blog, 23 Jan. 2004; http://martinfowler.com/articles/injection.html.
4. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.

describe quality attribute requirements in terms of the system's responses to specific stimuli in a measurable way.

- *Constraints* limit the space of design and implementation decisions.

Table 1 shows the results.

The architect employed ADD, following the steps we listed earlier, and he also used patterns[4] and tactics[2] catalogs. Table 2 summarizes the initial iterations of the design process performed by the architect.

The first iteration aimed to decompose the entire system (ADD step 2). This was because this system was greenfield development, and the primary concern was to define the overall system structure and allocate responsibilities to elements that team members could develop independently. The drivers for this iteration (ADD step 3) included the constraints 1.1–1.4 in Table 2. The architect decided to create a layered system that separated user interface aspects from the rest of the system. This would allow easy support for access through Web browsers or mobile devices as the system evolved. The architect created a presentation layer, a business logic layer, and a database access layer that aimed to isolate

TABLE 2

Five of the initial design iterations.

Iteration	Drivers (ADD step 3)	Element to decompose (ADD step 2)	Design decisions (ADD step 4)	Rationale
1	1.1. Web access and future support for mobile-app constraints 1.2. Object-oriented system (Java) and relational database constraints 1.3. Small development team 1.4. Time-to-market constraint	The system as a whole	Layers pattern	Promotes 1.1
			Database Access Layer pattern Domain Model pattern	Influenced by 1.2
			Domain Objects pattern	Influenced by 1.3 and 1.4
2	2.1. Primary use cases 2.2. Testability	Layers	User Interface Domain Objects pattern Application Service pattern Data Mapper pattern	Enables 2.1
			Separate-interface-from-implementation tactic	Promotes 2.2
3	3.1. Performance scenario 3.2. Search timetables use case	Database access layer	Lazy Load pattern	Promotes 3.1 Enables 3.2
4	4.1. Performance scenario 4.2. Primary use cases	Business logic layer	Introduce-concurrency-tactic-through-Leader/Followers pattern Transactions tactic	Promotes 4.1 Enables 4.2
5	5.1. Security scenario	Database access layer	Maintaining-data-confidentiality tactic	Promotes 5.1

the object-oriented system from persistence details in the relational database. Finally, the architect employed the Domain Objects pattern to allocate modules to the development team. Elements derived from the domain model were used to decompose the layers into modules that could be concurrently developed.

For the second iteration, the architect established a coordination model among the domain objects to support the primary use cases. The domain objects became the elements to decompose; the primary use cases became the drivers. The architect specialized the domain objects using patterns according to their layer. In the presentation layer, domain objects were responsible for receiving user input and displaying information. In the business logic layer, domain objects became application services. In the database access layer, domain objects became data mappers. To promote testability, the architect also applied the tactic of separating the interface from the implementation. The system requirements didn't explicitly state this requirement, but the architect knew from experience that supporting it was critical.

For the third iteration, the architect wanted to achieve the performance scenario in Table 1. He focused first on the database access layer. He improved performance by using the Lazy Load pattern to avoid retrieving unnecessary information from the database when an object with dependencies was retrieved. Although this decision promotes performance, by itself it's insufficient because other factors affect performance, such as dispatching multiple requests simultaneously.

So, to achieve the performance scenario, the architect made decisions elsewhere in the system. For example, in the fourth iteration, he applied the tactic of introducing concurrency on the business logic layer to support simultaneous requests along with transactions and their associated isolation levels. Design continued for additional iterations to satisfy the other drivers, such as security. The final result was an elegant design—one that satisfied all the drivers—that was still based on solid conceptual foundations and a proven design method.

At this juncture, the architect faced a challenge: How would he map existing frameworks to the previously produced design? This activity isn't always straightforward because a mismatch frequently occurs between the patterns and tactics that the frameworks incorporate and the ones selected in the design. For example, consider the business logic layer (iteration 4 in Table 2),

TABLE 3

Three design iterations using frameworks as part of the design process.

Iteration	Drivers (ADD step 3)	Element to decompose (ADD step 2)	Design decisions (ADD step 4)	Rationale
1	1.1 Web access and future support for mobile-app constraints 1.2 Object-oriented system (Java) and relational database constraints 1.3 Small development team 1.4 Time-to-market constraint	The system as a whole	Layers pattern	Promotes 1.1
			Database Access Layer pattern Domain Model pattern	Influenced by 1.2
			Domain Objects pattern	Influenced by 1.3 and 1.4
2	2.1 Primary use cases 2.2 Testability 2.3 Team experience with framework constraint 1.4 Time-to-market constraint	Layers	Application Service pattern	Enables 2.1
			Separate-interface-from-implementation tactic	Promotes 2.2
			JavaServer Faces, Spring, and Hibernate frameworks	Influenced by 2.3 and 2.4
3	3.1 Performance scenario 3.2 Search timetables use case	Database access layer	Use Hibernate's support for lazy associations Use Hibernate's cache support	Promotes 3.1 Enables 3.2

where the architect employed the tactic of introducing concurrency by creating a thread pool using the Leader/Followers pattern, which could affect additional components and relationships. Suppose the architect uses Spring to implement the design. In Spring, you can easily introduce concurrency by annotating the source code. When the architect selects Spring, some components and relationships previously designed for the Leader/Followers pattern become unnecessary, so a mismatch appears in the architecture. To correct this situation, the architect must revisit the design, which at this point might already be partially documented, and modify it to reflect the changes necessitated by the framework. However, this results in unnecessary rework.

Applying Our Approach to the Case Study

You can overcome the problem of mapping a purely conceptual design to the frameworks that implement it by considering technologies, and specifically frameworks, as first-class design concepts up front—that is, frameworks must be considered jointly with patterns and tactics during architectural design. This is particularly worthwhile in domains such as the one in our case study, because many frameworks exist to address such systems' design objectives, supporting many quality attribute concerns such as security, concurrency, and distribution.

Table 3 shows how the design process can be performed after frameworks are adopted as first-class design concepts. In iteration 2, the architect selected several frameworks that address objectives in the layers. Many frameworks exist for addressing domains such as Web-based user interfaces (for example, JavaServer Faces or Struts) or object-oriented models' persistence into relational databases (for example, Hibernate or iBatis). However, the architect selected frameworks on the basis of two of the constraints in Table 3: team experience with frameworks and time to market. These constraints complicated and restricted the selection pool of frameworks because the learning curve for frameworks introduces risk. This list of constraints, however, is small; in general, you will likely consider many criteria when selecting frameworks.

The choice of framework will affect further iterations. For example, decisions to satisfy the performance scenario in Table 3's third iteration differ from those in Table 2. In Table 3, because the architect chose Hibernate as the persistence framework, he addressed performance by configuring the provided framework parameters.[5] In this case, Hibernate incorporates the Lazy Load pattern but also incorporates tactics such as support for a cache (an instance of the tactic of maintaining multiple copies) that allow improved performance.

As in Table 2, design continues along other iterations. For Table 3, however, design decisions at each iteration range from selecting new patterns and tactics to configuring options provided by the frameworks selected in previous iterations.

Discussion

Our approach has been applied successfully in several projects in a large company in Mexico City that develops software for government and private customers.

Observed Benefits

Using our approach simplified the adoption of a method for systematic design—in this case, ADD. The company's architects had never received theoretical training on architecture and were oriented toward selecting technologies rather than using a method. Seeing how the knowledge they already had could fit in the method greatly helped them embrace it.

Also, design activities now produce architectures that can be implemented more straightforwardly, preventing mismatch and thus avoiding rework.

In addition, architecture design produces an executable architecture in addition to documentation (http://epf.eclipse.org/wikis/abrd/core.tech.common.extend_supp/guidances/concepts/executable_arch_D4E68CBD.html). This executable architecture is useful for testing, evaluation, and training.

Finally, during architectural evaluations, the evaluation team—other architects from the company—frequently uncovers risks associated with specific aspects of the selected technologies.

Selection Criteria

Framework selection should take into account criteria such as

- the development team's level of knowledge of each framework,
- the framework type (commercial or open source),
- the framework's maturity and level of support from its community (if it's open source),
- the type of license the framework employs and its effects on the design,[6] and
- the level of tool support (for example, the availability of plug-ins to simplify development using frameworks in popular integrated development environments such as Eclipse).

You must also consider the tradeoffs—the consequences for the drivers. For example, including a framework might affect the binary executable's size. Because frameworks typically include many functions, they tend to be large. If you're using only a small part of the framework, you might ask whether the framework justifies the larger application size. The framework selection could also affect aspects such as the project plan. If you select a new framework, the team will need training. If the project schedule doesn't include training time, this will negatively impact delivery dates and project profitability.

No clear criteria exist for determining when an architect should transition from designing with patterns and tactics to designing with frameworks. Many architects are familiar with several frameworks. They usually select one when they recognize a problem that can be addressed by a framework they're familiar with—for example, Web-based user interfaces. Generic frameworks, such as enterprise application frameworks, are usually selected very early during design because they cover many aspects of an application. Specialized frameworks are typically selected later, when a particular concern is addressed—for example, communicating with devices using a specific protocol.

Frameworks' Relationship to Requirements

Constraints are important in framework selection. Often, however, important constraints aren't explicitly captured as project requirements because they "belong" to the development environment. An example of this is employee skills—the development organization doesn't want this constraint to appear in a requirements document.

Certain quality attributes, such as testability, also aren't commonly described explicitly. Testability might not appear as part of the requirements if elicitation activities focus on end-user-facing requirements. An experienced architect will, however, know that achieving high quality will require thorough testing and will make design decisions to support this attribute. An example of this decision is in the second iteration in Table 2. One possible remedy is to have an internal requirements document that complements the standard requirements specification with information that is not visible to customers but still must be taken into account.

Frameworks might generate new requirements—for example, if you plan on using Ajax (Asynchronous JavaScript and XML), then your customers will need a Web browser that supports JavaScript. In this way, frameworks can limit your architecture design options. However, they might also create new opportunities, introducing the possibility of new features. For example, suppose an architect selects a GUI framework that automatically scales and adjusts its layout for different display sizes and aspect ratios. The organization might decide to capitalize on this as an opportunity to deploy the interface on mobile phones, even if this previously hadn't been a requirement.

Architecture design methods, such as ADD, describe an idealization of how architects perform their duties in real projects. These methods might seem complicated for practitioners who can't easily match abstract design concepts to their everyday experience, particularly with respect to technologies. Extending existing architecture design methods to consider frameworks as first-class design concepts is useful to overcome this situation. One benefit of this approach is that you can directly implement the resulting design; you don't have to match the design's output with the implementation technologies.

Although we still need to perform a quantitative evaluation of our approach's benefits, we've applied it in several large-scale, real-world projects. We've observed that it adds clarity and transparency to the relationship between requirements and constraints on one hand, and the design outcomes on the other. This, in turn, promotes architects' adoption of design methods and substantially reduces rework time.

References

1. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., Addison-Wesley, 2012.
2. R. Wojcik et al., *Attribute-Driven Design Version 2.0*, tech. report CMU/SEI-2006-TR-023, Software Eng. Inst., Carnegie Mellon Univ., 2006.
3. C. Hofmeister et al., "A General Model of Software Architecture Design Derived from Five Industrial Approaches," *J. Systems and Software*, vol. 80, no. 1, 2007, pp. 106–126.
4. F. Buschmann, K. Henney, and D. Schmidt, *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*, John Wiley & Sons, 2007.
5. G. King et al., "Improving Performance," *Hibernate—Relational Persistence for Idiomatic Java*, Red Hat Middleware, 2004; http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/performance.html.
6. I. Hammouda et al., "Open Source Legality Patterns: Architectural Design Decisions Motivated by Legal Concerns," *Proc. Int'l Academic MindTrek Conf.: Envisioning Future Media Environments*, ACM, 2010, pp. 207–214.

ABOUT THE AUTHORS

HUMBERTO CERVANTES is a professor of software engineering at the Autonomous Metropolitan University, Mexico City. He's also a visiting researcher at Quarksoft, the company where the method discussed in this article was applied. His research interests include software architecture design methods and their adoption in industrial settings. Cervantes received a PhD in software engineering from Université Joseph Fourier. Contact him at hcm@xanum.uam.mx.

PERLA VELASCO-ELIZONDO is a professor of software engineering at the Autonomous University of Zacatecas. Her research interests include software composition, architecture-centered software development, and software engineering education. Velasco-Elizondo received a PhD in computer science from the University of Manchester. Contact her at pvelasco@uaz.edu.mx.

RICK KAZMAN is a professor of information technology management at the University of Hawaii and a principal researcher at Carnegie Mellon University's Software Engineering Institute. His primary research interests are software architecture, design and analysis tools, software visualization, and software engineering economics. Kazman received a PhD in computer science from Carnegie Mellon University. He's a senior member of IEEE. Contact him at kazman@hawaii.edu.

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

IEEE computer society

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. Visit our website at www.computer.org.
OMBUDSMAN: Email help@computer.org.

Next Board Meeting: 13–14 June 2013, Seattle, WA, USA

EXECUTIVE COMMITTEE
President: David Alan Grier
President-Elect: Dejan S. Milojicic; **Past President:** John W. Walz; **VP, Standards Activities:** Charlene ("Chuck") J. Walrad; **Secretary:** David S. Ebert; **Treasurer:** Paul K. Joannou; **VP, Educational Activities:** Jean-Luc Gaudiot; **VP, Member & Geographic Activities:** Elizabeth L. Burd (2nd VP); **VP, Publications:** Tom M. Conte (1st VP); **VP, Professional Activities:** Donald F. Shafer; **VP, Technical & Conference Activities:** Paul R. Croll; **2013 IEEE Director & Delegate Division VIII:** Roger U. Fujii; **2013 IEEE Director & Delegate Division V:** James W. Moore; **2013 IEEE Director-Elect & Delegate Division V:** Susan K. (Kathy) Land

BOARD OF GOVERNORS
Term Expiring 2013: Pierre Bourque, Dennis J. Frailey, Atsuhiro Goto, André Ivanov, Dejan S. Milojicic, Paolo Montuschi, Jane Chu Prey, Charlene ("Chuck") J. Walrad
Term Expiring 2014: Jose Ignacio Castillo Velazquez, David. S. Ebert, Hakan Erdogmus, Gargi Keeni, Fabrizio Lombardi, Hironori Kasahara, Arnold N. Pears
Term Expiring 2015: Ann DeMarle, Cecilia Metra, Nita Patel, Diomidis Spinellis, Phillip Laplante, Jean-Luc Gaudiot, Stefano Zanero

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 • **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 BOARD OF DIRECTORS
President: Peter W. Staecker; **President-Elect:** Roberto de Marca; **Past President:** Gordon W. Day; **Secretary:** Marko Delimar; **Treasurer:** John T. Barr; **Director & President, IEEE-USA:** Marc T. Apter; **Director & President, Standards Association:** Karen Bartleson; **Director & VP, Educational Activities:** Michael R. Lightner; **Director & VP, Membership and Geographic Activities:** Ralph M. Ford; **Director & VP, Publication Services and Products:** Gianluca Setti; **Director & VP, Technical Activities:** Robert E. Hebner; **Director & Delegate Division V:** James W. Moore; **Director & Delegate Division VIII:** Roger U. Fujii

revised 22 Jan. 2013

IEEE