Introduction to the Special Issue on Software Architecture DRAFT: Do Not Distribute

David Garlan and Dewayne Perry

I. What is software architecture?

A critical aspect of the design for any large software system is its gross structure represented as a high-level organization of computational elements and interactions between those elements. Broadly speaking, this is the software architectural level of design [?], [?]. The structure of software has long been recognized as an important issue of concern (e.g., [?], [?]). However, recently software architecture has begun to emerge as an explicit field of study for software engineering practitioners and researchers. Evidence of this trend is apparent in a large body of recent work in areas such as module interface languages, domain specific architectures, architectural description languages, design patterns and handbooks, formal underpinnings for architectural design, and architectural design environments.

What exactly do we mean by the term "software architecture?" As one might expect of a field that has only recently emerged as an explicit focus for research and development, there is currently no universally-accepted definition. Moreover, if we look at the common uses of the term "architecture" in software, we find that it is used in quite different ways, often making it difficult to understand what aspect is being addressed. Among the various uses are (a) the architecture of a particular system, as in "the architecture of this system consists of the following components," (b) an architectural style, as in "this system adopts a client-server architecture," and (c) the general study of architecture, as in "the papers in this journal are about architecture."

Within software engineering, most uses of the term "software architecture" focus on the first of these interpretations. Typical of these is the following definition (which was developed in a a software architecture discussion group at the SEI in 1994).

> *The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.*

As definitions go, this is not a bad starting point. But definitions such as this tell only a small part of the story. More important than such explicit definitions, is the locus of effort in research and development that implicitly has come to define the field of software architecture.

To clarify the nature of this effort it is helpful to observe that the recent emergence of interest in software architecture has been prompted by two distinct trends. The first is the recognition that over the years designers have begun to develop a shared repertoire of methods, techniques, patterns and idioms for structuring complex software systems. For example, the box and line diagrams and explanatory prose that typically accompany a high-level system description often refer to such organizations as a "pipeline," a "blackboard-oriented design," or a "client-server system." Although these terms are rarely assigned precise definitions, they permit designers to describe complex systems using abstractions that make the overall system intelligible. Moreover, they provide significant semantic content that informs others about the kinds of properties that the system will have: the expected paths of evolution, its overall computational paradigm, and its relationship to similar systems.

The second trend is the concern with exploiting specific domains to provide reusable frameworks for product families. Such exploitation is based on the idea that common aspects of a collection of related systems can be extracted so that each new system can be built at relatively low cost by "instantiating" the shared design. Familiar examples include the standard decomposition of a compiler (which permits undergraduates to construct a new compiler in a semester), standardized communication protocols (which allow vendors to interoperate by providing services at different layers of abstraction), fourth-generation languages (which exploit the common patterns of business information processing), and user interface toolkits and frameworks (which provide both a reusable framework for developing interfaces and sets of reusable components, such as menus, and dialogue boxes).

Generalizing from these trends, it is possible to identify four salient distinctions:

- **Focus of Concern:** The first distinction is between traditional concerns about design of algorithms and data structures, on the one hand, and architectural concerns about the organization of a large system, on the other. The former has been the traditional focus of much of computer science, while the latter is emerging as a significant and different design level that requires its own notations, theories, and tools. In particular, software architectural design is concerned less with the algorithms and data structures used within modules than with issues such as gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance;

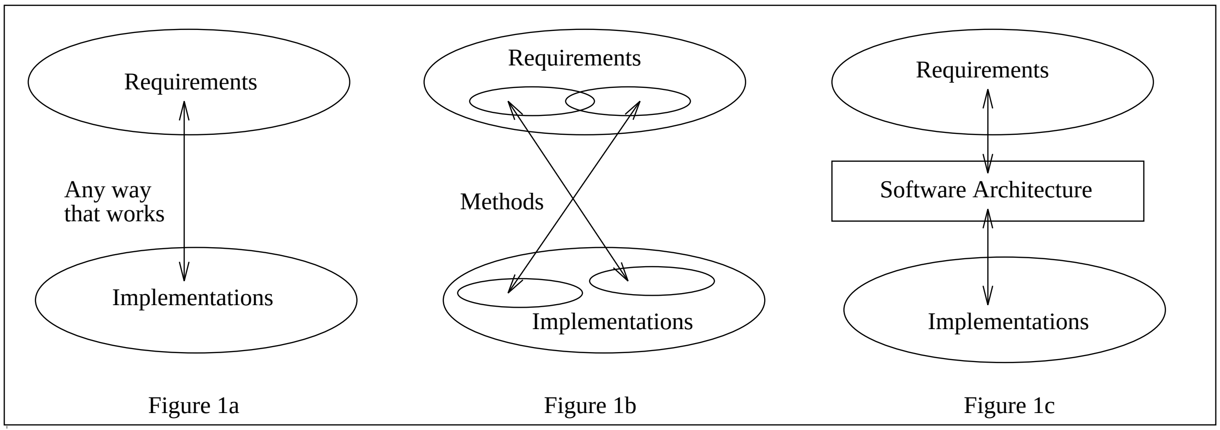

Fig. 1. Design Methods versus Software Architecture

and selection among design alternatives.

- **Nature of Representation:** The second distinction is between system description based on definition-use structure and architectural description based on graphs of interacting components [?]. The former modularizes a system in terms of source code, usually making explicit the dependencies between use sites of the code and corresponding definition sites. The latter modularizes a system as a graph, or configuration, of "components" and "connectors." Components define the application-level computations and data stores of a system. Examples include clients, servers, filters, databases, and objects. Connectors define the interactions between those components. These interactions can be as simple as procedure calls, pipes, and event broadcast, or much more complex, including client-server protocols, database accessing protocols, etc.
- **Instance versus Style:** The third distinction is between architectural instance and architectural style. An architectural instance refers to the architecture of a specific system. Box and line diagrams that accompany system documentation describe architectural instances, since they apply to individual systems. An architectural style, however, defines constraints on the form and structure of a family of architectural instances [?], [?]. For example, a "pipe and filter" architectural style might define the family of system architectures that are constructed as a graph of incremental stream transformers. Architectural styles prescribe such things as a vocabulary of components and connectors (for example, filters and pipes), topological constraints (for example, the graph must be acyclic), and semantic constraints (for example, filters cannot share state). Styles range from abstract architectural patterns and idioms (such as "client-server" or "layered" organizations), to concrete "reference architectures" (such as the ISO OSI communication model or the traditional linear decomposition of a compiler).
- **Design Methods versus Architectures:** A fourth distinction is between software design methods—such as object-oriented design, structured analysis, and JSD—and software architecture. Although both design methods and architectures are concerned with the problem of bridging the gap between requirements and implementations, there is a significant difference in their scopes of concern. Figure 1 illustrates this difference. Without either software design methods or a discipline of software architecture design, the implementor is typically left to develop a solution using whatever ad hoc techniques may be at hand (Figure 1a). Design methods improve the situation by providing a path between some class of system requirements and some class of system implementations (Figure 1b). Ideally, a design method defines each of the steps that take a system designer from the requirements to a solution. The extent to which such methods are successful often depends on their ability to exploit constraints on the class of problems they address and the class of solutions they provide. One of the ways they do this is to focus on certain styles of architectural design. For example, object-oriented methods usually lead to systems formed out of objects, while others may lead more naturally to systems with an emphasis on dataflow. In contrast, the field of software architecture is concerned with the space of architectural designs (Figure 1c). Within this space object-oriented and dataflow structures are but two of the many possibilities. Architecture is concerned with the trade-offs between the choices in this space—the properties of different architectural designs and their ability to solve certain kinds of problems. Thus design methods and architectures complement each other: behind most design methods are preferred architectural styles, and different architectural styles can lead to new design methods that exploit them.

II. Why is Software Architecture Important?

Architectural design of large systems has always played a significant role in determining the success of a system: choosing an inappropriate architecture can have a disastrous effect. The current recognition of the importance of software architecture would appear to signal the emergence of a more disciplined basis for architectural design that has the potential to significantly improve our ability to construct effective software systems.

Specifically, a principled use of software architecture can have a positive impact on at least five aspects of software development.

1. **Understanding:** Software architecture simplifies our ability to comprehend large systems by presenting them at a level of abstraction at which a system's high-level design can be understood [?], [?]. Moreover, at its best, architectural description exposes the high-level constraints on system design, as well as the rationale for making specific architectural choices.
2. **Reuse:** Architectural descriptions support reuse at multiple levels. Current work on reuse generally focuses on component libraries. Architectural design supports, in addition, both reuse of large components and also frameworks into which components can be integrated. Existing work on domain-specific software architectures, reference frameworks, and design patterns have already begun to provide evidence for this [?], [?].
3. **Evolution:** Software architecture can expose the dimensions along which a system is expected to evolve. By making explicit the "load-bearing walls" of a system, system maintainers can better understand the ramifications of changes, and thereby more accurately estimate costs of modifications [?]. Moreover, architectural descriptions can separate concerns of the functionality of a component from the ways in which that component is connected to (interacts with) other components. This allows one to change the connection mechanism to handle evolving concerns about performance, interoperability, prototyping, and reuse.
4. **Analysis:** Architectural descriptions provide new opportunities for analysis, including high-level forms of system consistency checking [?], conformance to an architectural style [?], conformance to quality attributes [?], and domain-specific analyses for architectures that conform to specific styles [?].
5. **Management:** There is a strong rationale for making the achievement of a viable software architecture a key milestone in an industrial software development process. Achieving this milestone involves specifying a software system's initial operational capability requirements, its the dimensions of anticipated growth, the software architecture, and a rationale, which demonstrates that the architecture, if implemented, would satisfy the system's initial requirements and anticipated directions of growth. If one proceeds to develop a software product without satisfying these conditions, there is significant risk that the system will be either inadequate or unable to accommodate change.

The importance of software architecture can also be seen in terms of its broad impact on market drivers that are important for software-intensive businesses. These drivers affect the way businesses plan their software projects and, ultimately, the way they build their software systems.

Given that software has become an integral part of a wide variety of products, and that many of these products have been on the market for some time, there is a broad base of existing software. This base represents a significant investment of capital, and as such should be considered as capital assets. The ability to use these assets is of important to the financial health of software producers. Software architecture, to the extent that it focuses on domain-specific abstractions and the use of various granularities of existing architectural elements, supports the exploitation of these assets.

Interoperability is another market driver that contribute to the successful exploitation of a company's software assets, since it promotes sharing across product lines. Software architecture, to the extent that it is an effective means of establishing a common architectural framework across various domain-related products, represents a significant means for achieving interoperability.

As the research and development costs of software increase, there is increasing market pressure to procure, rather than produce, significant portions of software systems, either by third party production or by the purchase of software commodities. The result of this shift to *consuming* rather than *producing* software places an increased importance on the resulting integration of the developed and purchased components. Software architecture, to the extent that it provides accessible codification of design elements and their correct use in the context of specific architectural styles, can be critically important in assuring that these various components are integrated effectively.

An important software development driving factor is that of "interval reduction." In many segments of software-dependent businesses, market forces call for reduced costs and release cycles. If a company can maintain cost and quality levels while reducing production cycles, it can achieve significant overall reductions in costs, while at the same time improving time-to-market. Software architecture, to the extent that it is able to reduce production time by using existing assets, exploit common architectural frameworks, and establish more effective integration and generation mechanisms, may be able to achieve these time reductions.

III. The State of the Practice

Much of architectural description in practice is largely informal: drawings in which boxes represent processing components, and arrows represent interactions among those components. At best, these pictures present high-level overviews of the software identifying major design components and data/control flows, but they usually provide little insight into the role that the data plays in the computation or the details of the interactions between the components.

However, the growing recognition of the importance of software architecture is leading to much more explicit use of architectural design as currently manifested in the fol-

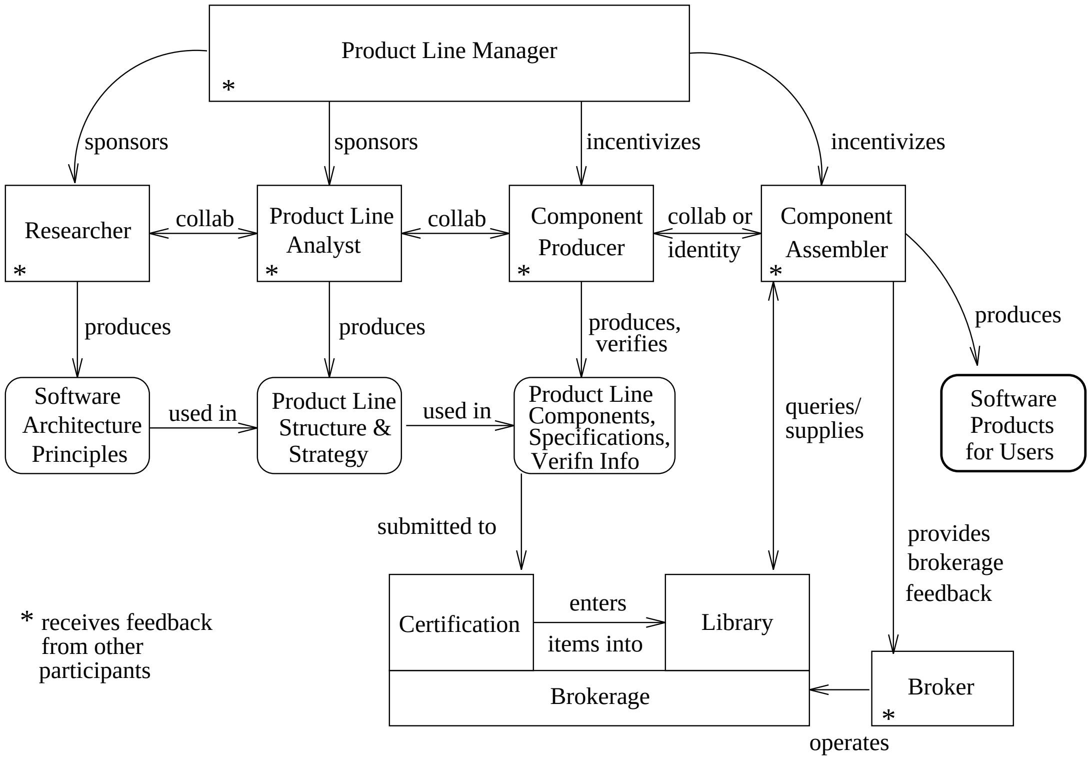

Fig. 2. Boehm-Scherlis Megaprogramming Enterprise Model

lowing approaches:

- standardized components
- product families
- platforms
- domain-specific architectures

The use of *standardized components* arises where software producers recognize that there is a common set of components that are used across a set of products. An example of this is BaseWorx [?], a set of standard components that forms the basis for a set of related operations support systems. Specific systems are produced by adding the specialized components to the standard ones.

Product families are one means of capitalizing product assets and using that asset base to create a family of closely related product architectures. While the instances of a product family all tend to be in the same domain, it is the sharing of components amongst those instances and the generation of those instances that is the driving force in product family architecture.

One approach that is gaining in popularity among many software producers is that of a software *platform*. A platform is a general set of components that form the basis of a variety of related products. These components are usually generic capabilities, such as databases, graphical user interface generators, etc. The components provide means of specialization by either special declarative languages or special-purpose scripting languages. A platform is similar to a set of standardized components, but it is populated with large-granularity components that need to be tailored for the specific software system.

Finally, there is a movement towards *domain-specific architectures* [?]. Most software intensive businesses have domain-specific systems that are vital to their financial profitability. By concentrating on those domains and creating architectural abstractions that are specific to their domain, these companies can combine the best aspects of standard platforms and standardized components to create and specialize their domain-related product families. The platforms are tailored to their domains and the standard components are built to provide both the domain-specific processing and structures, and the necessary glue to weld the many different architectural elements together.

A good example of this approach is the oscilloscope architecture developed at Tektronix, Inc. Product engineers in collaboration with researchers developed a reusable product architecture that provided a customizable framework for instrumentation systems, based on a specialized dataflow model [?], [?]. Other more recent examples include a number of DoD-sponsored projects in domains such as avionics, command and control, and mobile robotics.

This trend towards reuse of architecture-based, product-line assets is leading to new roles, artifacts and relationships in software development organizations. An example of a new organizational model based on architectural reuse is provided by Boehm and Scherlis in their "Megaprogramming Enterprise Model." As illustrated in Figure 2, some of new roles introduced in the model include

- **product line managers**, who oversee the human, financial, and software resources needed for successful development and exploitation of software product lines;

- **product line analysts**, who are concerned with domain analysis, and engineering and evolution of software product line architectures;
- **component producers**, who develop, test, customize, and adapt components;
- **component assemblers**, who identify, assess, and compose components to produce software systems; and
- **"brokers"**, who manage and help populate a library of components and architectures.

IV. The State of Research

While the application of good architectural design is becoming increasingly important to software engineering practice, the fact remains that much of common practice leads to architectural designs that are informal, ad hoc, unanalyzable, unmaintainable, and handcrafted. This has the consequences that architectural designs are only vaguely understood by developers; that architectural choices are based more on default than solid engineering principles; that architectural designs cannot be analyzed for consistency or completeness; that architectures are not enforced as a system evolves; and that there are virtually no tools to help the architectural designers with their tasks.

Current research in software architecture is attempting to address all of these issues. Among the more active areas are:

1. **Architecture Description Languages:** This area addresses the need to find expressive notations for representing architectural designs and architectural styles. In particular, much of the focus of this research is on providing precise descriptions of the "glue" for combining components into larger systems.
2. **Formal Underpinnings of Software Architecture:** This area addresses the current imprecision of architectural description by providing formal models of architectures, mathematical foundations for modularization and system composition, formal characterizations of extra-functional properties (such as performance, maintainability, etc.), and theories of architectural connection.
3. **Architectural Analysis Techniques:** Researchers in this area are developing new techniques for determining and predicting properties of architectures. In particular, progress is being made to understand the relationships between architectural constraints and the ability to perform specialized analyses, as well as abstraction techniques that make analysis practical for large systems.
4. **Architectural Development Methods:** As architectural design becomes better understood, it becomes imperative to find ways to integrate architectural activities smoothly into the broader methods and processes of software development.
5. **Architecture Recovery and Re-Engineering:** The ability to handle legacy code is critical for large systems with long lifetimes. Research is beginning to address extraction of architectural design from existing systems, unification of related architectural designs, abstraction, generalization, and instantiation of domain-specific components and frameworks. In addition, there is increasing research activity addressing issues of interoperability: techniques for detecting component mismatch and bridging those mismatches.
6. **Architectural Codification and Guidance:** While expertise in architectural design is currently the province of virtuoso designers, there is on-going work on codifying this expertise so that others can use it. This has led to an interest in rules and techniques for selection of architectural styles, handbooks of patterns and elements, and curricula for educating software architects.
7. **Tools and Environments for Architectural Design:** Given notations and models for characterizing software architectures, it becomes possible to support architectural design with new tools and environments. Current work is addressing architectural analysis tools, architectural design environments, and application generators.
8. **Case Studies:** Finally, we are beginning to see the emergence of good published case studies of architectural design including retrospective analyses of successful (and sometimes unsuccessful) architectural development. These serve both to increase our understanding of what it takes to carry out architectural design, as well as providing model problems against which other researchers can gauge the effectiveness of their techniques and tools.

V. This Issue

In this special issue on software architecture we are pleased to present seven papers that illustrate many of these emerging research areas.

The first paper, "Architectural Tradeoffs for a Meaning-Preserving Program Restructuring Tool," by William Griswold and David Notkin, presents a case study of the architectural design of a tool for program analysis and transformation. It nicely illustrates how an architectural view of a system helps clarify system issues, and highlights the challenges of combining multiple architectural paradigms in a single design.

The second paper, "A Domain-Specific Software Architecture for Adaptive Intelligent Systems," by Barbara Hayes-Roth, Karl Pfleger, Phillippe Lalanda, Phillippe Morignot, and Marko Balabanovic, describes a hierarchical domain-specific architecture based on a combination of layering, data streams, and blackboards. It shows how a common architectural framework, together with a rich set of building blocks, can provide a powerful tool for system construction.

The next three papers deal with architectural description. "A Syntactic Theory of Software Architecture," by Thomas R. Dean and James R. Cordy, presents a notation that focuses on the syntactic, graphical aspects of architectural patterns.

"Abstractions for Software Architecture and Tools to Support Them," by Mary Shaw, Robert DeLine, Daniel V. Klien, Theodore L. Ross, David M. Young, and Gregory Zelesnik, describes a language and set of supporting tools for architectural description. One of the more innovative aspects of this work is its support for the explicit definition of architectural connectors.

"Specification and Analysis of System Architecture Using Rapide," by David C. Luckham, Larry M. Augustin, John J. Kenney, James Vera, Doug Bryan, and Walter Mann, also defines an architectural description language. In this language, interactions between components can be characterized in terms of event patterns. These can further be used to analyze a running system for conformance to more global rules about legal event interleavings.

The final two papers are concerned with formal approaches to architectural modelling. "A Formal Approach to Correct Refinement of Software Architectures," by R.A. Riemenschneider, Mark Moriconi, and Xiaolei Qian, considers the problem of architectural refinement. The authors argue that refinement should preserve certain structural and semantic properties, and show how this notion leads to the use of conservative extension as a refinement criterion.

"Formal Specification and Analysis of Software Architecture Using the Chemical Abstract Machine Model," by Paola Inverardi and Alexander L. Wolf, explores ways to characterize architectures as reactive models inspired by recent formal work on the Chemical Abstract Machine.

Acknowledgements

We would like to thank Barry Boehm, Paul Clements, Robert Monroe, Mary Shaw, and Jeannette Wing for their insightful comments on earlier drafts of this guest introduction.

References

[1] D. Garlan and M. Shaw, "An introduction to software architecture," in *Advances in Software Engineering and Knowledge Engineering, Volume I*, World Scientific Publishing Company, 1993.

[2] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, 1992.

[3] E. W. Dijkstra, "The structure of the "THE"-multiprogramming system," *Communications of the ACM*, vol. 11, no. 5, pp. 341–346, 1968.

[4] D. L. Parnas, P. C. Clements, and D. M. Weiss, "The modular structure of complex systems," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 259–266, March 1985.

[5] R. Allen and D. Garlan, "Beyond definition/use: Architectural interconnection," in *Proceedings of the ACM Interface Definition Language Workshop*, vol. 29(8), SIGPLAN Notices, August 1994.

[6] G. Abowd, R. Allen, and D. Garlan, "Using style to give meaning to software architecture," in *Proc. of SIGSOFT'93: Foundations of Software Engineering*, Software Engineering Notes 18(5), pp. 9–20, December 1993.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1994.

[8] E. Mettala and M. H. Graham, "The domain-specific software architecture program," Tech. Rep. CMU/SEI-92-SR-9, CMU Software Engineering Institute, June 1992.

[9] R. Allen and D. Garlan, "Formalizing architectural connection," in *Proc. of ICSE'16*, May 1994.

[10] P. Clements, L. Bass, R. Kazman, and G. Abowd, "Predicting software quality by architecture-level evaluation," in *To appear in Proceedings of the Fifth International Conference on Software Quality*, (Austin, Texas), October 1995.

[11] D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting style in architectural design environments," in *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, ACM Press, December 1994.

[12] R. P. Beck *et al.*, "Architectures for large-scale reuse," *AT&T Technical Journal*, vol. 71, pp. 34–45, November-December 1992.

[13] D. Garlan, "The role of formal reusable frameworks," in *Proceedings of the First ACM/SIGSOFT International Workshop on Formal Methods in Software Development*, (Napa, CA), 1990.

David Garlan is an Assistant Professor of Computer Science in the School of Computer Science at Carnegie Mellon University. His research interests include software architecture, the application of formal methods to the construction of reusable designs, and software development environments. Professor Garlan heads the ABLE project, which focuses on the development of languages and environments to support the construction of software system architectures. Before joining the CMU faculty, Professor Garlan worked in the Computer Research Laboratory of Tektronix, Inc., where he developed formal, architectural models of instrumentation software.

Dewayne E. Perry is a Member of Technical Staff in the Software and Systems Research Center at AT&T Bell Laboratories. He spent the first part of his computing career as a professional programmer, then combined both research (as a visiting research faculty member in Computer Science at Carnegie Mellon University) and consulting in software architecture and design, and has concentrated on research in software engineering for the past 10 years. His research interests (in the context of building and evolving large software systems) include: software architecture, software process descriptions, analysis, modeling, visualization, and environmental support; software development environments; and the practical use of formal specifications and techniques,

Dr. Perry is a member of ACM and IEEE, a member of the editorial board for IEEE Transactions on Software Engineering, and Co-Editor in Chief of Software Process: Improvement and Practice.